2023 American Control Conference (ACC)
San Diego, CA, USA. May 31 - June 2, 2023

A Tutorial on Real-Time Computing Issues for Control Systems

Daniel Y. Abramovitch*, Sean Andersson, Kam K. Leang, William Nagel, Shalom Ruben

Abstract— **This paper presents a tutorial on the elements of computation in a real-time control system. Unlike conventional computation or even computation in digital signal processing systems, computation in a feedback loop must be sensitive to issues of latency and noise around the loop. This presents some fundamental requirements, limitations, and design constraints not seen in other computational applications. The logic of presenting such a tutorial is that while the computer technology changes at a rapid pace, the principles of how we match that technology to the constraints of a feedback loop remain consistent over the years. We will discuss the different computational chains in a feedback system, ways to conceptualize the effects of time delay and jitter on the system, and present a three-layer-model for programming real-time computations. The tutorial also presents some filter and state-space structures that are useful for real-time computation. It concludes with an overview of the different sample rate ranges currently used in some typical control problems and a short discussion of how business models affect our choices in real-time computation.**

I. Motivation: Why Talk About Computation?

While computation is critical to any digital control system, our CAD tools have gotten so effective that fewer and fewer control engineers are competent programmers outside of Matlab or Python. This leaves us to rely on the prepackaged real-time implementation tools provided by manufacturers. While these tools allow one to go from model to real-time without writing C or C++ code, their need for generality and for hiding the computational complexity from the user often consume much of the real-time resources of a chip. The control engineer who hits that limit without understanding computation is left relatively helpless. A basic understanding of how computation-for-feedback issues affect the performance of real-time systems will greatly expand the performance achievable by many algorithms.

In this paper we will focus on the pieces of computation that need to be in place in any real-time control system. While we are focusing on control systems where the main computation is done by a digital computer (broadly any computation done via programmable digital logic, including processors and field programmable gate arrays (FPGAs)), there is always a role for analog electronics in interfacing these means to the real world. Figure 1 shows the main "computations" done in a feedback loop as abstracted blocks. We can think of four chains where processing happens:

- The physical system to real-time computer "input" signal chain and its computations.

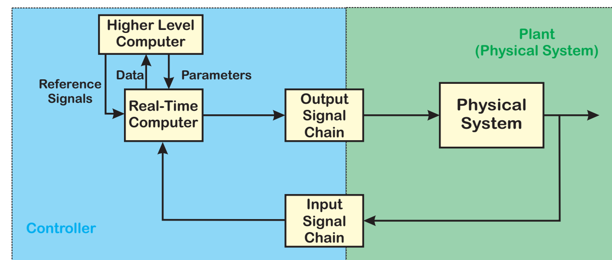

Fig. 1. An abstracted view of the main computational divisions in a feedback system.

- The real-time computer to physical system "output" signal chain and its computations.
- The physical system "computations" and discussions of various model types.
- Inside the real-time computer itself: how to think about computer architectures and programming in the context of real-time control systems.

We describe the physical system "computing" because it is transforming its input signals into output signals, and thinking in this metaphor helps us both understand the system and generate models. We think of the computational path inside the real-time computer as a signal chain because once again signals are transformed from the input of that chain to the output, handed from one routine and data format to another. Each has its own potential latency, jitter, and noise. We typically only think about those of the plant. Bode's integral theorem [1], [2], [3] teaches us that once noise gets into a loop we can only adjust where we amplify it. Causality means that once latency enters a loop, we cannot eliminate it. Jitter just makes all of this worse.

Finally, any discussion of computation issues must be made relative to the physical system time constants. How a control engineer looks at computation for an atomic force microscope (AFM) sampled at 2 MHz is different from what they face on a pressure control loop sampled at 1 Hz. This vast speed range drives dramatic changes in computation. For this reason, this tutorial talks about the principal drivers of each signal chain. Knowing that they exist but contribute little to critical loop dynamics is a far better situation than being oblivious to them until after they have handcuffed a brilliant new control algorithm.

Reasonable people may argue that the first three blocks can be wrapped into a pure zero-order hold (ZOH) equivalent, and dealt with via direct design [4], but this both obscures the sources of those discrete-time features and

*Daniel Y. Abramovitch is with Agilent Technologies, Santa Clara, CA USA (abramovitch@ieee.org)
Sean Andersson is with Boston University, Boston, MA USA
Kam K. Leang is with the University of Utah, Salt Lake City, UT USA
William Nagel is with Widener University, Chester, PA USA
Shalom Ruben is with the University of Colorado, Boulder, CA USA

979-8-3503-2807-3/$31.00 ©2023 AACC

removes the opportunity for co-design. For these reasons, we will try to understand the chains physically, separately, and in their relation to the overall loop design.

In this paper, we will often refer to Moore's Law [5], the general rule proposed by Intel co-founder Gordon Moore that the amount of logic one can pack onto a silicon chip doubles roughly every 18 months. This prediction has been remarkably accurate over the decades, not only through the march of technology but also because the chip makers themselves have felt pressured to make sure they meet the line (in logarithmic space) [5]. In this paper, Moore's Law will be used as a generic stand-in for the rapid advance of computer and electronic technology. Similarly, "Newton's Laws" usually refers to Newton's Laws of Motion [6], but in this paper the term is shorthand for scientific modeling and inference, i.e. what drives the real world system.

Each of the blocks in Figure 1 affects the performance of the feedback loop. Moore's Law has made the left side of Figure 1 much more powerful. Modulo being able to attack far more physical problems than before (again Moore's Law), the right side is governed by Newton's Laws. The following should be considered fundamental to understanding computation for feedback control systems.

> Newton's Laws take precedence over Moore's Law, and they always will. *However,* Moore's Law helps us read the fine print of Newton's Laws, helps us get computation close to the real physics, close to the real model.

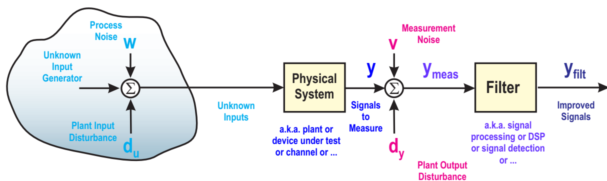

Fig. 2. A filtering structure for looking at processes.

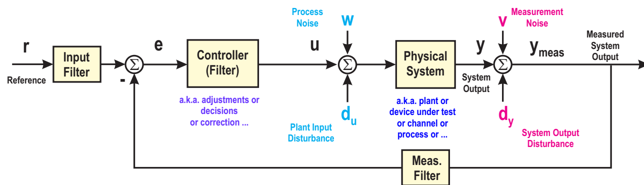

Fig. 3. A feedback structure for physical processes.

Finally, it is worth discussing the mental framework for filtering, diagrammed in Figure 2 with that of feedback, diagrammed in Figure 3. The key differentiator of the filtering framework is that we can never assume any access to the input driving the physical system. Consequently:

- ⇒ Noise and disturbances are modeled solely at the output.
- ⇒ This fundamentally limits input-output modeling.
- ⇒ Nothing we do in our filtering will affect the process.
- ⇒ Because of this, we have to assume that the physical process has to be reasonably behaved on its own.
- ⇒ The filtering context is insensitive to latency.

Conversely, a feedback framework only exists if we assume that we have access to some inputs to drive the physical system.

- ⇒ Noise and disturbances are modeled at both input and output of the physical system.
- ⇒ Our models use both plant outputs and inputs.
- ⇒ We adjust our measurements to better drive our inputs.
- ⇒ Our inputs can and most likely will change the behavior of the physical system.
- ⇒ We are sensitive to latency.

This matters because much of what has been written about real-time computation is from a filtering framework. Access to inputs to the system drives sensitivity to latency, and this changes the entire perspective.

II. Time Delay and Sampling

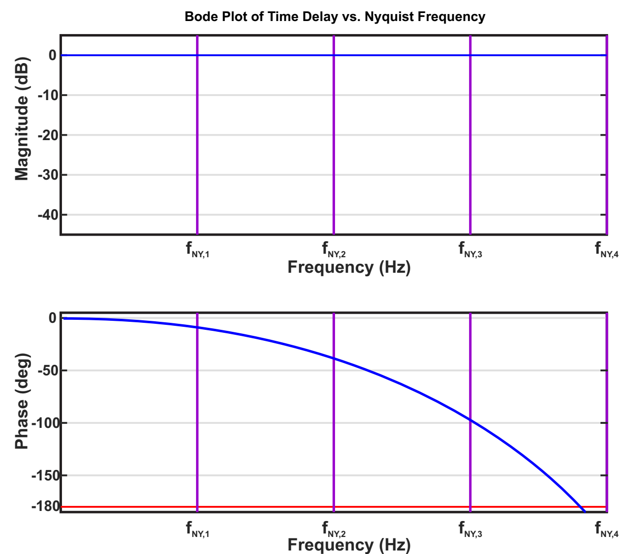

Fig. 4. Bode plot of physical time delay versus Nyquist rate

Time delay (latency) in a feedback loop is one of the key limiting factors of closed-loop performance [7]. Latency in time is negative phase in frequency, and without phase margin, feedback control is untenable. In a feedback loop, we can think of four general sources of time delay:

a) physical properties of the system,
b) sensor/actuator effects,
c) conversion delays, and
d) computational and sample rate delays.

We will focus on conversion and computation delays in Sections IV and VII, respectively, since these are things that we may affect by better real-time system design (hardware and software). Here we group sensor, actuator, and plant delays together as delays that we cannot alter merely with electronics. We will discuss the consequences of this delay. We especially want to make obvious what fast sampling can and cannot do to handle this delay.

A delay of Δ seconds is typically modeled in the s-plane as:

$$D(s) = e^{-s\Delta}. \quad (1)$$

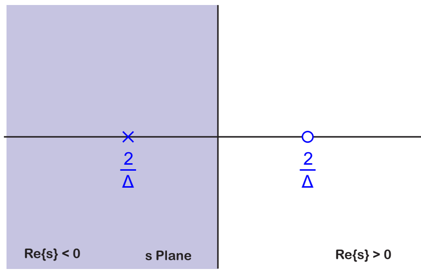

Fig. 5. First order Padé approximation of time delay on the s-plane. Note the non-minimum phase (NMP) zero.

We can evaluate (1) at $s = j\omega$ to generate a Bode plot as shown in Figure 4, but for our analysis we usually like to have a rational transfer function. It is common to use a Padé approximant [8] for this, and while many variations are possible, a first order numerator and denominator effectively illustrate the issues we must contend with. For a first order Padé approximation of (1), we get

$$e^{-s\Delta} \approx \frac{1 - \frac{s\Delta}{2}}{1 + \frac{s\Delta}{2}} = \frac{\frac{2}{\Delta} - s}{\frac{2}{\Delta} + s} \tag{2}$$

This simple and reasonable approximation of delay has given us a stable pole and non-minimum phase (NMP) zero, as diagrammed in Figure 5.

It is common for control engineers to say that we need to simply sample faster to deal with the time delay, but fast sampling does not make physical delay disappear. This is illustrated in Figure 4, which shows a Bode plot of physical delay versus different Nyquist frequencies. The faster sampling has just pushed the Nyquist frequencies to the right, up to areas of higher phase lag due to delay. That does not solve anything, although it gives us more room in the frequency space to apply phase-lead to compensate for some of this delay.

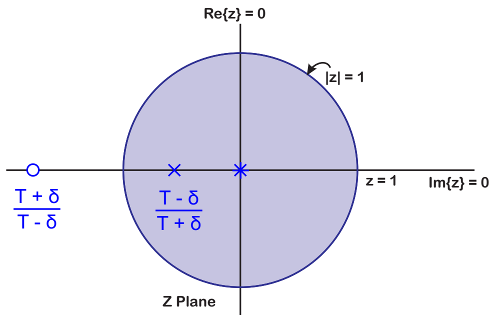

Fig. 6. Padé approximation of time delay on the z-plane. The full sample delays result in a pole at $z = 0$. The partial sample delay is handled by a first order Padé Approximation. If we have M unmatched poles at $z = 0$, then we will have M zeros at $|z| = \infty$.

We can return to the Padé approximation and discretize it to look at this in the z-plane. However, we will first break up Δ into full and partial sample periods, i.e.

$$e^{-s\Delta} = e^{-s(MT_S+\delta)} \tag{3}$$

where $\Delta = MT_S + \delta$ and $0 \leq \delta < T_S$. We will use z^{-1} for full sample period delays and the Padé approximant for the partial delay, δ.

Using a trapezoidal rule (TR) on (2) (replacing Δ with δ) results in $D_\delta(z) = 1$ for $\delta = 0$ and $D_\delta(z) = z^{-1}$ for $\delta = T_S$, which matches intuition. In between these endpoint values of δ we get:

$$e^{-s\delta} \approx \frac{\frac{2}{\delta} - s}{\frac{2}{\delta} + s} \xrightarrow{TR} \frac{(T_S - \delta)z + T_S + \delta}{(T_S + \delta)z + T_S - \delta} = D_\delta(z) \tag{4}$$

for $0 \leq \delta < T_S$. The poles and zeros get exposed by reordering this as:

$$D_\delta(z) = \left(\frac{T_S - \delta}{T_S + \delta}\right)\left(\frac{z + \frac{T_S+\delta}{T_S-\delta}}{z + \frac{T_S-\delta}{T_S+\delta}}\right). \tag{5}$$

The result of this discretization is diagrammed in Figure 6. We see that we have mapped the sub-sample portion of our delay to a stable pole and NMP zero. However, we have M poles at $z = 0$. These are so benign in digital signal processing (DSP) environments (the filtering context of Figure 2) that signal processing engineers like to refer to their Finite Impulse Response (FIR) filters as "all-zeros" filters as they only have zeros in z^{-1}. For them, it is a harmless misstatement; a simplification.

The lack of matching finite zeros in the z-plane for the M poles means that there are M zeros at $|z| = \infty$. People who close feedback loops know that on any version of the root locus [9], [10], [4], [11] the closed-loop poles go from the open-loop poles to the open-loop zeros. This means that at some point, M closed-loop poles will be going to those M open-loop zeros at $|z| = \infty$, obviously outside the unit circle. Fast sampling has not saved us from closed-loop poles streaking towards instability. It merely allows us an opportunity to put more compensation inside the unit circle to allow us to push our closed-loop system a bit further. It is up to us to properly use that opportunity.

III. Understanding Phase Delay, Phase Noise, and Jitter

We should now discuss timing uncertainty, a quantity known alternately in different fields as phase noise [12] or jitter [13]. Phase noise is a frequency domain term related to time through the relation:

$$\Delta\theta = e^{j\omega\Delta t}. \tag{6}$$

Usually, this is considered relative to a particular frequency, such as a carrier frequency: $\omega = \omega_C$. Jitter is a time domain term, usually relative to some sample period, T_S:

$$\text{jitter} = \frac{\Delta t}{T_S}. \tag{7}$$

For digital systems, jitter is a more common concept than phase noise. Much of what we try to minimize is time delay and jitter because each of these can badly affect our control systems and can be the result of bad computer architecture.

A few more figures illustrate this idea. In Figure 7 we see the difference between phase delay and phase noise in a sinusoidal signal. Phase delay is a predictable lag,

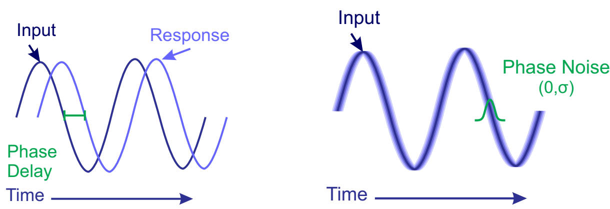

Fig. 7. Phase delay and noise as seen in a sinusoid.

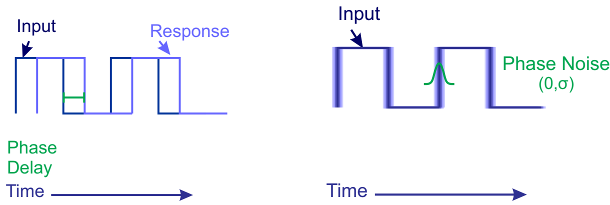

Fig. 8. Phase delay and noise as seen in a square wave.

phase noise is not. Phase noise makes the exact timing of any part of the signal unknowable. Phase noise is usually characterized by distribution, e.g. a Gaussian. Note that neither phase delay nor phase noise change the maximum or minimum level of the signal, but uncertainty in when signal happens translates into uncertainty in the signal value.

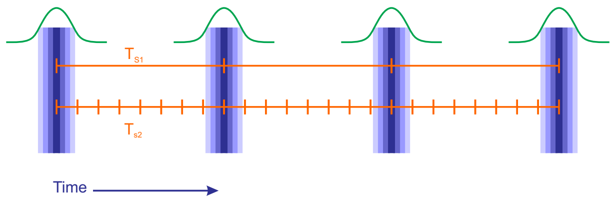

Fig. 9. Jitter is usually defined relative to a sample period.

Moving to Figure 8, we see many of the same properties of phase delay and phase noise seen in sinusoids apply to square waves. Now, the sharp edges mean that timing uncertainty results in uncertainty in a logic level. Logic levels trigger operations inside programmable logic (PL) or a processor, so jitter results in uncertainty in when operations will be triggered. Whether or not this is important to us is highly dependent on the amount of jitter relative to our sample period. Figure 9 illustrates how the same amount of timing uncertainty that would be insignificant for the longer sample period, T_{S1}, covers the majority of shorter sample period, T_{S2}. The moral of this is that if our physical system time constants allow for a slower sample rate, then we may be far less sensitive to jitter than we would be in a system that requires a sample rate several orders of magnitude higher than the first one.

We get to the last visualization of how jitter can affect us with Figure 10. Our illustration here is to show that if our controller computation times not predictable, we may miss the next sample instants and lose samples. In the illustration of Figure 10, the predictable portion of controller computations, T_{comp}, takes up most of a sample period, T_S. Jitter in the exact calculation time makes missing samples a probabilistic problem. This can even be seen in student

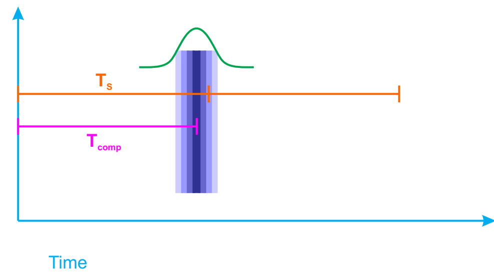

Fig. 10. Jitter, added onto computational time, may make us miss samples.

laboratory systems [14]. Some systems are passive enough that this does not cause problems for our loop performance. However, for lightly damped, high speed, unstable, and/or nonlinear systems, this could be disastrous.

The lesson here is that for high performance systems with relatively short sample periods, we want to minimize computational jitter. As much as possible, we want to have:

- deterministic computations,
- deterministic memory access,
- deterministic sampling, and
- deterministic communications in the digital system.

These desires frame how we discuss each of the computation chains previewed in Figure 1 of Section I.

IV. The Input Signal Chain: The Real World to Computation

For most real systems, there are multiple inputs and outputs, but for simplicity of concept and visualization, we will stay with a single-input, single-output explanation here.

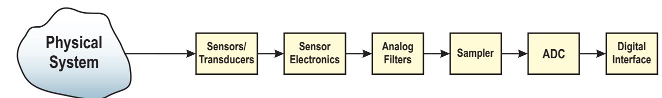

Fig. 11. An abstracted view of the input signal chain from the physical system to our computation. This one is specific to use in a feedback loop.

Figure 11 shows a lot of component blocks we often ignore in considering the implementation of getting measurements from the physical system into digital controllers. Sometimes they are simply bundled inside a turn-key system, but for our purposes we want to discuss them individually. *Note that the technology changes over time, but the basic functionality of the blocks does not.* The path from the physical system to our computer algorithm starts with a sensor.

Sensor/Transducer: A lot of science and engineering are employed to convert some physically sensed phenomenon into a calibrated, repeatable, electrical signal. Sensors have their own dynamics, in terms of linearity, time constants, and noise properties. The physical environment often determines what sensors are available and what they cost.

Sensor Electronics: These are especially suited to the sensor and environment of the sensor. They typically are packaged with the sensor, but their characteristics and limitations sometimes need to be considered apart from the sensor itself. They are often specialized to handle tough environments,

extreme levels of temperature, pressure, moisture, ambient noise, speed, and voltages and currents. Ideally, they get signals into the low voltage, well regulated electronics where we like to do our small-signal filtering.

Analog Filters: While there are many elements that can perform a "filtering function", we focus here on analog electronic filters. These are combinations of operational amplifiers (op amps), resistors, capacitors, inductors, diodes, transistors, and other components that allow us to implement mathematical filtering functions inside of circuitry.

Why do we need analog filters when we can digitally filter signals inside our computers using only high level languages and none of that wiring and solder? One short answer is that we cannot digitally filter everything. A second short answer is that we can do a much better job of digitally filtering signals that have been cleaned up and normalized by some analog filters. Signals outside of our sampling bandwidth need to be managed with analog filters. Of particular interest to control engineers are anti-alias filters (Section IV-A).

Sampler: The role of a sampler or sample-and-hold is to capture the well-conditioned analog signal (while minimally affecting it) and hold it long enough for the analog-to-digital conversion (ADC) of that signal to be done.

ADC: Analog-to-Digital Converters (ADCs) are often grouped with samplers when discussed in controls textbooks, but there are many cases in which the sampler can capture signals at a higher rate than a single ADC can convert them. (This phenomena led to a class of digital oscilloscopes called "equivalent time" digital oscilloscopes [15].)

ADCs convert sampled signals into a digital computer compatible signals. Note that we tend to think in terms of floating point numbers when we analyze control signals, but ADCs (and DACs) deal with fixed-point representations. Their quantization is nonlinear, but usually modeled as noise [16]. The number of bits of accuracy ties to cost and conversion time. We will discuss this more in Section IV-B.

Digital Interface: How the converted signal is delivered to the controller is another fundamental source of possible delay and jitter. There are significant design tradeoffs between the number of signal lines to a converter chip. While we may dream of having dedicated, parallel interfaces to each ADC in a system, the costs of laying out 16+ parallel lines for each analog input and keeping their timing aligned at high speed means that many of the interfaces are serial, with the inherent delay of serialization at the ADC and deserialization at the processor. Is each ADC dedicated to one signal or multiplexed (Section IV-B)?

A. Anti-Alias and Oversampling

PES Pareto [3], [17], [18] gave us a way to find the most critical noise sources affecting the error. While noise gets shaped once it's in the loop, we can read "Bode's fine print" if we attack it at the source, before it enters the loop. Almost always, this requires an understanding of analog electronics and how they interact with sampled data system. Here, we focus on the issues posed by anti-alias filters, which are typically implemented as analog low-pass filters (LPFs) with cutoff frequencies at or above the Nyquist frequency.

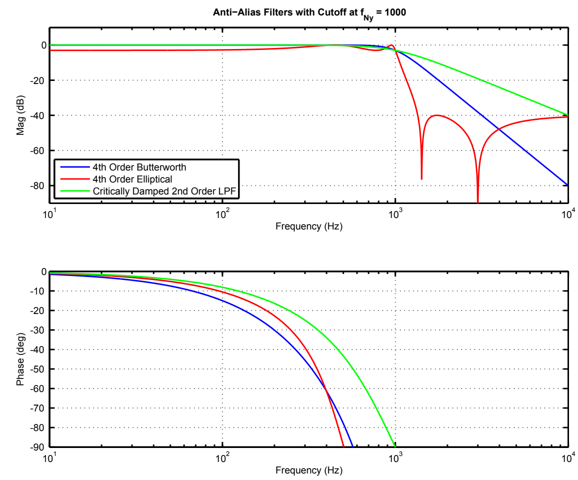

Fig. 12. Frequency responses of various anti-alias filters. All filters have a DC gain of 1, with the passband ending at the Nyquist Frequency ($f_{Ny} = f_S/2$ = 1kHz here). Under an assumption that the sample frequency is 10 or 20× the open loop crossover frequency, we can examine the filter phase response, which can significantly degrade the phase margin of the system, as documented in Table I.

Filter	**Phase at** $f_{Ny}/10$	**Phase at** $f_{Ny}/5$	**Attenuation at** $10f_{Ny}$
4th Order Butterworth	$-15.0°$	$-30.1°$	-80.1 dB
4th Order Elliptical	$-10.6°$	$-22.8°$	-40.9 dB
2nd Order Butterworth	$-8.12°$	$-16.4°$	-40.1 dB

TABLE I
PHASE PENALTY OF REPRESENTATIVE ANTI-ALIAS FILTERS FROM FIGURE 12. THE CORNER FREQUENCY IS CHOSEN TO BE AT THE NYQUIST FREQUENCY, $f_{Ny} = f_S/2$.

From our first digital controls class, we are told that we need to apply anti-alias filters in order to avoid aliasing of higher frequency signals into our control bandwidth [4], [19]. What is often overlooked are the effects, particularly in terms of negative phase in the passband, of such filters. Table I and Figure 12 show the effects of three simple anti-alias filters, with their corner frequencies at the Nyquist rate. The simple take away from these is to note the substantial amount of negative phase imparted in order to get attenuation above the Nyquist frequency. The loss of 30° of phase margin at 1/5 the Nyquist rate (or 1/10 the sample rate) is nothing to take lightly. The anti-alias filter can have strong phase lag, and this itself can severely limit the achievable bandwidth or destabilize the system. The selection of anti-alias filters must be combined with the selection of sample rate and in this case it seems that what we might think of as "oversampling" is fundamental to achieving desired closed-loop performance.

B. Analog to Digital Converters

Simply defining a block as an Analog to Digital Converter (ADCs) is not specific enough. The lowest latency version ADC could be a single dedicated converter on a parallel digital bus, eliminating any delay based on serialization/deserialization. Parallel operation is "expensive" for two reasons. First of all, it requires far more digital lines be laid out on the circuit board containing the ADC and the path to the processor, consuming valuable board real estate. The second cost is that as signaling speeds have gone up, keeping parallel digital signals phase aligned on these buses has gotten substantially more difficult. For these reasons, high-speed serial buses have become increasingly popular inside computing environments, and despite the cost of serialization and deserialization, they may have lower overall latency than the available parallel solutions. Unlike the filtering context (Figure 2), those of us in the feedback context (Figure 3) need to be highly aware of both the transmission speed and the latency of these channels.

At the other end of the speed/dedicated line spectrum is the shared ADC, comprising a single multiplexed sample and hold handling many input lines and presenting these sequentially to the converter. The ADC does each conversion and then puts the result on a serial line to the processor. The resulting architecture likely has significantly higher latency than the parallel one. Control and system theory knowledge must guide the design: the more cost effective but slower architecture may have delays that are orders of magnitude shorter than the physical system time constants. In such a case, demanding the parallel interface is wasteful and unnecessary. It is better to yield on these channels quoting the lack of need so that we can be taken more seriously when we need to press for the top architecture.

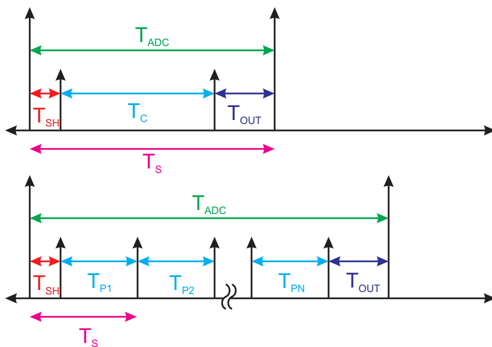

Fig. 13. Diagrams of sample timing. The lower diagram shows the pipelining of the digital computation needed for conversion.

It is also worth looking inside of the ADC itself. The ADC and sample and hold timing can be diagrammed as in Figure 13. Here, T_{SH} represents the sample and hold time, T_{OUT} represents the transmission on the digital interface to the processor, and T_C represents the digital computation time in the ADC conversion. What is often overlooked is that many ADCs speed up their sample rate by pipelining this conversion time into smaller digital processing blocks. Even though the time for any individual sample to reach the processor is longer, the sample period, T_S, can be shortened. This is yet another example of architectures that are excellent for DSP applications (filtering context), but have very negative effects on a system that is sensitive to latency (feedback context). When someone not attuned to latency makes the choice, they can unknowingly consume 90% of the phase margin and bandwidth. Such errors cannot be fixed by any algorithm, and so it is critical that engineers informed by a knowledge of control principles be involved in the design of the input signal chain. It is not necessary to be the expert in the latter, but only to be conversant enough so as to inform the experts of the latency sensitivity.

V. The Output Signal Chain: Computation to the Real World

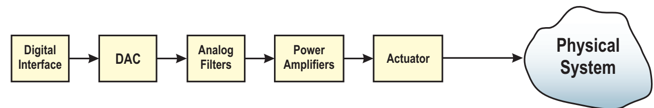

Fig. 14. An abstracted view of the output signal chain from our computation to the physical system.

A diagram of an output signal chain using a Digital to Analog Converter (DAC) is shown in Figure 14. Its components usually include:

Digital Interface: How the converted signal is delivered from controller to the DAC is another fundamental source of possible delay and jitter. There are significant design tradeoffs between the number of signal lines to a converter chip. While we may dream of having dedicated, parallel interfaces to each DAC in a system, the costs of laying out 16+ parallel lines and keeping their timing aligned at high speed means that many of the interfaces are serial, with the inherent delay of serialization at the processor and deserialization at the DAC. Is each DAC dedicated to one signal or multiplexed? These issues will be discussed in more detail in Section V-A.

DAC: Digital-to-Analog Converters (DACs) convert the computer signal into a well regulated analog voltage. They face many of the same quantization issues as ADCs. We will discuss these more in Section V-A.

Analog Filters: On the output chain analog filters are often used to remove digital artifacts or smooth the analog version of digital signals. For example, a digitally produced sine wave will have steps if we look closely enough, but passing it through a low-pass filter (LPF) or band-pass filter (BPF) can considerably smooth that signal.

Power Amplifier/Drive Electronics: Theses scale up the voltages and currents produced by the DACs to drive the actuators.

Pulse-Width Modulation (PWM): A slightly different output chain substitutes PWM in place of a DAC, as diagrammed in Figure 15. PWM is surprisingly common in industrial environments. The use of PWM depends on the dynamics of plant being far slower than the electronics (often true). Its utility lies in part in the fact that a single, serial,

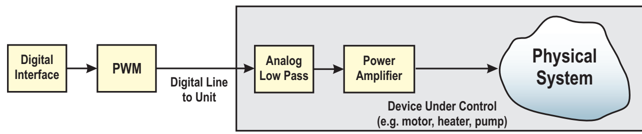

Fig. 15. An abstracted view of the output signal chain from our computation to the physical system using pulse width modulation (PWM).

binary line is very useful in noisy industrial environments. In most of these applications, the drive electronics have been made part of the actuator/device itself.

Actuator: This part pushes on the real world.

A. Digital to Analog Converters

Generally speaking, DACs seem architecturally simpler than ADCs, but this does not mean they do not often include similar pipelining to that shown in Figure 13. Again, control engineers need to be involved in the selection of these components so that we can avoid having our phase margin wrecked by digital pipelining in the conversion circuits.

B. Pulse Width Modulation

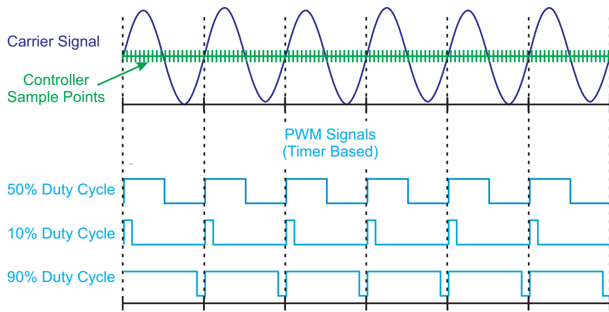

Fig. 16. Classic PWM converts a multi-bit number into a stream of 1s and 0s whose duty cycle on the carrier encodes the number.

A classic view of PWM is shown in Figure 16. A multi-bit number is modulated onto a carrier on a 0-1 digital line such that the duty cycle represents the number. It has the advantage of a noise insensitive, inexpensive single line. The systems driven by this are slow and the system itself integrates/low-pass filters the modulated signal to restore the the original multi-bit number. In essence, the PWM then becomes an inexpensive DAC, but it finds application in a lot of industrial systems such as pumps, heaters, and motors.

VI. The Plant's "Computation"

The plant itself is the most fundamental piece of computation in our loop, and the one that sets and limits what all the other pieces can and *must* do. In actuality, the plant is doing some form of physical computation, which we try to model with a combination of first principles (a.k.a. "physics based" or "Newton's Laws") and data driven approaches. The plant itself is – for this discussion – fixed, but the models we choose to apply to it are quite varied. We can have:

- A linear, time-invariant (LTI) model for control design.
- A model used for parameter identification.
- A linear, time-varying model, or one with uncertain data sampling, for observer design and operation.
- A complex, nonlinear model on dedicated hardware as a digital twin of the system for health monitoring and simulation.

The point of mentioning these (and many other) forms is that Moore's Law allows us to have many of these running in parallel on the same system.

VII. The Computer Itself

We have delayed getting to this section until the other prerequisite computational pieces had been described. At this point, we focus on the broad issues of computation in and out of the real-time environment. Section VII-A will introduce and discuss a highly useful Three-Layer Model of computation in a real-time environment. Section VIII-F will get more basic, in the sense of how our choices of filter structure can also affect jitter, latency, and numerical stability.

A. The Three-Layer Model

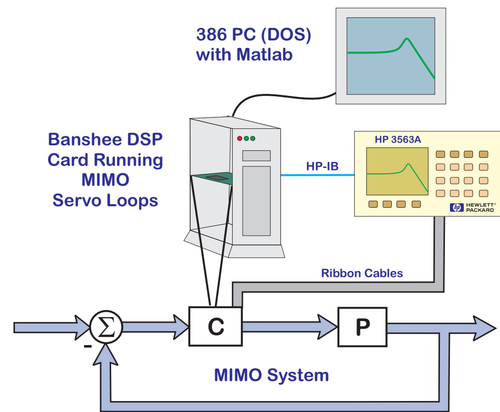

Fig. 17. The Banshee Multivariable Workstation (BMW).

This section will present a three-layer model of computation for real-time systems. This split in functionality seems quite common when one examines many real-time systems, but without abstracting out the different layers and their purposes, the smearing of the boundaries can add a lot of confusion about how they should be programmed. In other words, it's always been there, but we didn't see it.

The first author's first experience with this model was in building something called the Banshee Multivariable Workstation in the early 1990s at Hewlett Packard Labs (HPL) [20]. The hardware was an old DOS based PC to hold the upper layer, with a Banshee Floating Point DSP board produced by Atlanta Signal Processing (ASPI) to run the real-time operations. The intent was to have the floating point DSP run the real-time computations while in the PC we could run Matlab. This is diagrammed in Figure 17. With the initial work of Carl Taussig, it was interfaced to the HP 3563A Control Systems Analyzer (CSA), an augmented version of HP's 3562A DSA, that was capable of both analog and digital frequency response function measurements and curve fits

[21], [22], [23], [24]. The CSA had a superset of the features of the DSA, specifically enhanced to work with discrete-time systems. Significantly, measurements could be coordinated from the host computer and completed measurements and/or parametric curve fits were uploaded to the host computer to be used in Matlab.

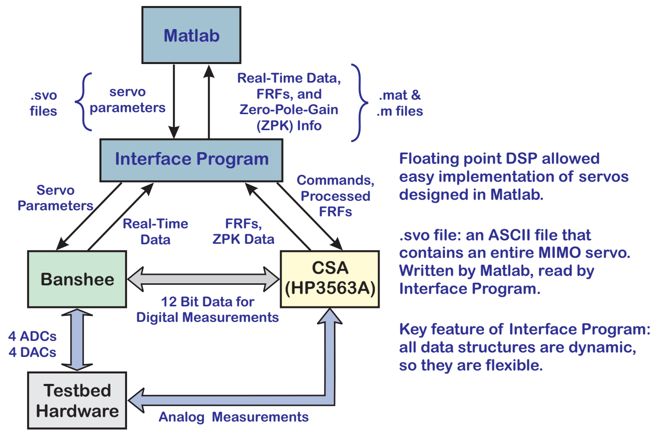

Fig. 18. The BMW: a functional block diagram.

Schematically, the software architecture diagrammed in Figure 18 reveals three distinct layers. The most ad-hoc was the Interface Program, used so that Matlab could interact with a real-time DSP. The considerable work to build the Interface Program (mezzanine layer) smoothed out the interactions with the upper level decision functions (Matlab) and the Hard-Real-Time (Banshee DSP Board interacting with disk drive testbed).

This first seemed like a special case, but over the years, the three separate layers kept showing up. At the time technology limits, recognition, and the difficulty of programming to the model tended to make projects avoid it. However, the three-layer-model remains and the rest of this section will provide a more universal vision for it, diagrammed in Figure 19.

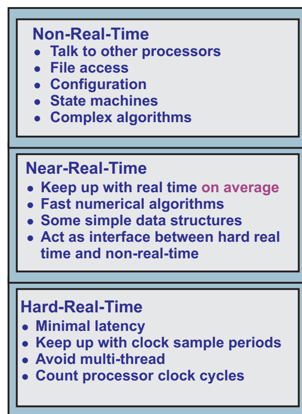

Fig. 19. A three-layer computing model useful for understanding the computing needs of control systems.

Figure 19 presents a more general form of a three-layer compute model that serves well for understanding programming in real-time systems. At the bottom is the Hard-Real-Time, in which we are counting clock cycles to make certain we complete computations between sample points. At the top is the Non-Real-Time, which is the computing environment most programming classes are oriented towards. In between is the least well known mezzanine layer, which functions to keep the two other layers happy. It must be able to outrun the Hard-Real-Time on average, but perhaps in a burst mode. It must also have enough memory flexibility to deal with the Non-Real-Time. We will discuss the different forms of this model and how programming against it improves the performance of embedded systems and their real-time layers. We wil now drill down to discuss the individual layers in more depth.

The first layer to discuss is the real-time layer, which we further delineate as *Hard-Real-Time*, and diagrammed at the bottom of Figure 19. The added qualification is intended to emphasize that the processing must meet the physical system's timing. Missed timing/clocks are bad, especially when controlling devices or trying to make sense of sampled signals. This is the reason for the discussion of Section III. The faster the physical world – relative to the processing – the simpler and more deterministic the processing must be.

This layer often features a lot of simple but time-critical tasks. It is possible that the tasks are not that fast, but they do have to happen within a tight time window. To assign one "large" processor to one of these simple tasks is wasteful. (Metaphorically we are using a sledge hammer to push in a thumbtack.) To handle all of these time-critical tasks, we either want a lot of lightweight processors in parallel or want to multiplex in time using a single powerful processor.

FPGAs shine here as one can build small "single task" processors, i.e. *Application Specific Processing*. In this layer we avoid fancy memory access/caching due to uncertainty in timing. Memory management using caches to hold the most frequently/recently accessed data is fine in other layers, but the possibility of a cache miss – when one requests data that happens to not be in the cache – can increase the time of that memory access by several orders of magnitude. This is a major potential contributor to the operational jitter diagrammed in Figure 10. For systems with dynamics that must be tightly controlled, this is unacceptable. Simple, deterministic memory access is critical and so we write algorithms to rely on on-chip memory whenever possible.

We next move up to the Non-Real-Time layer, at the top of Figure 19. This is the layer that we learn about in most programming and computer science (CS) classes. Non-Real-Time systems are what most people think of as computers and smart devices. Most users will only interact with embedded systems through this layer, and as it has no critical timing, it needs only be responsive enough to not annoy users. Tasks in the Non-Real-Time layer interact with other systems, interact with users, and generally have a lot of multi-tasked functionality. This is the land of multi-tasking operating systems (often a form of Linux) and Graphical User Interfaces (GUIs).

Programming-wise, this is the land of lists, Python

servers, web pages, recursive algorithms, database searches, programs that manage lots of dynamic memory, and caching. We can (and should) use much more complex code/algorithms/memory management in this layer than we could ever do in the Hard-Real-Time layer. We can afford to represent our signals and numbers in single or double precision floating point representations. It's what most folks consider programming, and there are lots of tools to aid the developer. Furthermore, nothing that we program is time critical. Because folks are so much more comfortable in this layer, lots of companies make money offering to take care of the lower layers [25]. This is certainly a viable approach in many situations, but giving up knowledge of how to program for the lower layers makes it nearly impossible for us to port simpler versions of our algorithms down closer to the data.

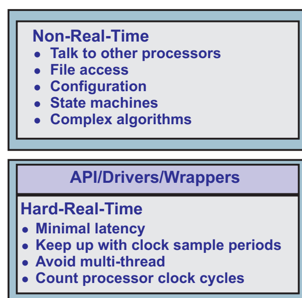

Fig. 20. When Non-Real Time is so much faster than the real world

There are times when the processing power is so fast – compared to the physical world dynamics being handled – that folks try to do everything from the top level. In this case many of our computational latency and jitter concerns from Sections II and III are not significant for our closed-loop system. This is diagrammed in Figure 20. There are still real-time, "touch the world" blocks that are encapsulated into little blocks with application programming interfaces (APIs). In some cases, small, inexpensive processors (e.g. Arduino, Raspberry Pi) are so cheap that it is worth trying to do all parts of a simple problem on them.

These systems certainly save us from having to deeply consider the middle layer and the cycle-counting programming typical of the Hard-Real-Time layer, but there are two inherent dangers here. The first is that this architecture only really works if the speed of the "main processor" swamps the physical system dynamics and the processing needed. The second is that there is always a temptation to add more applications, threads, and processing requests to main processor. All of a sudden, we are violating the speed assumptions. Almost everyone who has owned a computer or a smart phone has experienced exactly this latter phenomenon: they were fast when first purchased, but seemed to slow to a crawl in the years that followed. It is unlikely that the circuits got slower. Instead, more and more operations were being packed into the same piece of silicon.

Yet another common approach is *the advanced tool approach*, diagrammed in Figure 21. The idea is that to get more Hard-Real-Time processing without having to code in a Hard-Real-Time way, there are increasing numbers of advanced development tools. Examples include:

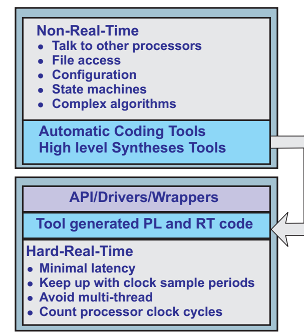

Fig. 21. The advanced tool approach

- Simulink® to FPGA synthesis (HDL Coder).
- MathWorks RT Workshop® (MATLAB®/Simulink®/Stateflow® $\Longrightarrow$ real-time hardware)
- Xilinx High Level Synthesis Tools
- dSpace HIL and auto-code (MATLAB®/Simulink®/Stateflow® $\Longrightarrow$ real-time hardware)
- National Instruments HIL simulation
- Hardware targets designed to use these flexible tools

The blocks these tools access on the real-time targets are very standardized "Lego® blocks". This has its own overhead in amount of real-time processing and PL used. Often the generality comes with a significant speed hit. However, this method is available and popular with academics and research folks. The overhead usually makes it far less suitable for product.

Having passed through these variants, we are now in a position to have a deeper discussion of the full three-layer-model displayed in Figure 19. What has been missing from our previous examples has been a fully fledged middle/mezzanine layer. The Banshee Multivariable Workstation (BMW) had a definitive mezzanine layer, as denoted by the Interface Program of Figure 18. In that example, it served as a bridge between the highly sophisticated layer of Matlab design and data analysis and the Hard-Real-Time of the DSP board and testbed.

It turns out that the processing, data, timing, and memory models of the Non-Real-Time and Hard-Real-Time are so different that some sort of rubber/glue layer is almost always needed. A term that serves for this is Near-Real-Time, or Mezzanine Level Processing. It has to outrun the Hard-Real-Time – but only on average. It features a lot of memory buffers and message queues between the two other layers. This middle layer wants to ensure that it never leaves the Hard-Real-Time layer without inputs in its queue and that it clears the Hard-Real-Time's output queues fast enough that they are never completely full. Doing so means that the most time-critical layer, the one that has to keep up with Newton's Laws, never has to slow down because of the rest of the embedded computing system. We can use more complex memory/processing/algorithms in the Near-Real-Time. Once tasks in this layer get past their initialization and allocate

their memory, they go faster than the Hard-Real-Time (hence the need for buffers). Programs in this layer can also handle a lot of monolithic streaming data.

Depending upon the speed of the physical system dynamics, this Mezzanine Layer can be handled with either a dedicated Linux thread, a Real-Time Operating System (RTOS), or a bare metal (no operating system) implementation. However, unlike the layers on either side, far less is formalized about this layer. Some of this is due to the very purpose of this layer: to form a smooth interface between the others, however this can be viewed as the *Wild, Wild West* of programming.

Even if one recognizes the layers of Figure 19 and programs them differently, there are the issues of how to combine them and how to share data between them. Traditionally, the choices have been:

- To combine multiple layers on one processor. However, the swapping and the multi-tasking could cause complexity and timing problems. By putting everything on one processor, it is difficult to give the Hard-Real-Time layer the priority it needs and wasteful to have a single large processor handling so many simple real-time tasks.
- To put different layers on different processors/chips/boards. While this preserves the independence between the layers, the hand-offs of data and control between layers is usually a bottleneck. The interfaces end up being either low overhead and slow (old parallel buses) or higher overhead and fast (fast serial buses). (Fast parallel buses between boards are disappearing from everything but video controllers.) Crossing boundaries is often a nightmare, as we are having to create compatible channels between very different levels of programming. Furthermore, being on different boards usually implies that the processors are on different clock domains. This gives yet another opportunity for longer and less predictable delay between different computation layers. However, moving the data between the different layers is a key enabler to being able to program for each of the layers. Moving the data around cleanly is also necessary for making processing agnostic, that is, moving the processing to the layer which is most appropriate for the data and the sample rate.

The key to a solution allows for separate hardware segments for each layer while still enabling fast transfer of data between layers. In this respect, chips which feature a System on a Chip (SoC) provide such a solution. Multiple chip families from makers such as Altera (now part of Intel) and Xilinx (now part of AMD), provide multiple high level ARM processors on the same die as programmable logic (PL), on-chip-memory (OCM) as well as communication and interfaces for off chip memory [26], [27]. This allows multiple levels of computing on one piece of silicon (processing, programmable logic, memory, buses), and the chip makers are moving towards advanced synthesis tools to "compile" hardware to meet the system timing requirements (or alert the designer when they cannot be met).

VIII. Control Algorithm Programming

The prior sections have focused at a high level on the many issues that can encumber the proper execution of control algorithms. In this section, we will discuss specifics in how we code our filters and our state-space structures to minimize issues such as latency, jitter, and numerical instability in the code. There are many wonderful texts describing the different mathematical benefits of different controller structures. What is sometimes left to the reader is how to implement those structures (a.k.a. write code) and how the particular properties of the compute engine place limits on how we write that code.

> We cannot emphasize enough the principle that for algorithms to make it from the notepad, through Matlab or Python, and into systems that work, they must be debuggable. This goes beyond commenting one's code (although this helps) but goes into the filter and state-space structures into which we fit our schemes. Structure, compartmentalization, and documentation matter. The clueless schmuck who looks at your code in 6 months or 2 years will likely be you. Throw that schmuck a bone.

A. *It's a Filter*

In most human-built implementations of feedback controllers, the computations take the form of weighted values of functions of prior inputs to the plant, outputs from the plant, reference signals, and auxiliary sensor inputs to the controller block. In other words, they implement one or more filters. Whether implemented using analog circuits [28], digital logic, or computer code, controllers involve filters (and decision trees). We will focus on the digital versions of these, starting with computer code but branching into digital logic as a special "hardware implementation" of code. Even a state-space structure can be considered under this "filter" rubric [11], [4], [29], [30].

When we talk about filters in code, we most often mean that the filter is in a subroutine (or function or subprogram, but for our purposes here, these are the same). While filters may be coded in-line in small systems, best practices of compartmentalizing coding dictate that most filters will be implemented in subroutines. The subroutines might themselves have their own subprograms, but for what we are trying to explain, one level of subprogram is enough. Considering the most simple (and common) case of a linear, constant-coefficient filter, we will have parameters into the routine that need to be shared from the top level down to the routine and some that go back. As subroutines generally have a separate data space from that of the calling routine, the parameters need to be either:

- **Global:** Shared between all routines and the top level program.
- **Passed:** These are passed down via the *stack* during the subroutine call.

- **Static/persistent:** These keep their value even after the routine ends so that on the next call they remember their previous value.
- **Part of an instance of a class:** This is static data local to an instance of a class that gets initialized through the class, but also maintains its value until the instance is deleted.

Of issue here is what data needs to be known in the filter routine and which does not. In our LTI filter example, we can imagine the common form for a polynomial form IIR filter as:

$$\begin{aligned} y(k) &= -a_1 y(k-1) - a_2 y(k-2) - \ldots - a_n y(k-n) \\ &\quad +b_0 u(k) + b_1 u(k-1) + \ldots + b_n u(k-n). \end{aligned} \quad (8)$$

In this filter, we have have three sets of parameters:

- **Filter Coefficients:** Our values of $\{a_i, b_j\}$ define what we think of as a filter as they are the filter coefficients. Most often, these are defined before the filter is ever run and they do not change.
- **Previous filter inputs and outputs:** This signifies the $\{y(k-i)\}$ and $\{u(k-i)\}$ values that are known prior to the current time step.
- **Current filter inputs and outputs:** This signifies $u(k)$, the current input at step k, and $y(k)$, the output of the filter to be computed based upon the coefficients, the prior inputs and outputs, and the most recent input to the filter.

In a very simplified understanding, only $u(k)$ is a new input to the filter, and only $y(k)$ is a new output. Given the right programming methods, no other parameters from above need to be passed in to or out of the filter at any one step.

This is a very good spot to mention the difference between batch mode programming and sequential programming. In batch mode programming, we present all of the data to the subroutine or program at the entry point. It applies the parameters and generates all of the data to be output. In this mode, the routine would be called only once. It would execute its operations on the large data space, then return the result in (probably) an equally large data space. We are used to this type of programming for much of our off-line data processing, e.g. when we run filter() or filtfilt() in Matlab. While the filter of (8) might only process one new input at a time, in batch mode processing all the data would be passed down to the filter routine which would loop through repeatedly before returning a column of results.

However, this type of programming makes no sense in the most critical operations of a feedback controller. Instead of a batch of data, we have new data coming in at each time step (probably) and need to generate a response at each time step (probably). The filter routine only sees a tiny bit of new data at each step and must respond to it. It is this incremental use of filters and programs that we will focus on, since these are the ones that will be running in a feedback loop that must run "forever". In this mode of programming, passing the coefficients and old inputs and outputs on the parameter stack every time the filter routine is called is tremendously wasteful of computation time. It does not advance our algorithm; only increasing our computation delay while adding no functionality. In such programs, it is far more efficient to have both filter coefficients and signal values (delays/states) as static/persistent data so that only new information gets passed on the parameter stack.

B. The Wire

Besides being a classic HBO show, *The Wire* is the name the first author gives to the first filter subroutine one writes in any computer environment. The role of *The Wire* is akin to the role of the various *"Hello world!"* programs in programming environments: it is the first proof of concept that helps debug the basic structure of the code. *The Wire* is simply a filter that takes the input and passes it to the output. In terms of (8), all coefficients are 0 except for b_0 which is 1. By first coding *The Wire* engineers can debug their filter code structure without worrying about numerics, time constants, and discretization.

Once *The Wire* is working, the next version involves having all coefficients at 0 except for b_0 and b_1 which are both set to 0.5. Once this simple FIR averager is working, the next version sets all coefficients to 0 except for $b_0 = 1$ and $a_1 = -0.5$ to set up a simple IIR low-pass filter. With these, one can test impulse and step responses and generally debug most of the filter structure. After that, one can get more complicated with the numerics.

C. Numerics, Parameterization, and Operations

The successful implementation of algorithms into filter routines relies on doing a variety of different things well, or at the very least, not screwing up a whole bunch of small things. Experienced programmers (just like experienced practicing engineers) realize that every implementation will need to be debugged, and so a key aspect of coding is to design the code in such a way that it can be easily debugged. This is one of the reasons for breaking code up into smaller subprograms or subroutines. Taken to the next level, we get to Object-Oriented Programming (OOP) where much of the data and code specific to that data, are wrapped up into a class. (Discussing OOP in detail is beyond the scope of this tutorial, but some parts are keenly relevant to this discussion.)

Those used to programming in feature rich, Non-Real-Time environments, can often not see the need for such careful attention to structure, but for any practical programming system, structure comes before numerics. Once the data handling is properly handled, then one can look at how the algorithm handles the numerics. This is not an either-or situation: the structure puts one in a position to better understand the numerics and the numerics then may well mandate a change to the structure.

A feature of programs that needs to be understood is the cost of different mathematical operations. In most modern computation architectures, additions, subtractions, and

multiplications take 1–5 clock cycles while operations such as division or computation of transcendental functions take on the order of 30+ clock cycles (depending upon the particular processor). Accesses to on-chip memory take 1–2 clock cycles and so certain real-time operations are far better accomplished via a local look-up-table (LUT) and interpolation, than by following the complete algorithm.

When working with Hard-Real-Time, one might often need to give up on floating-point operations in the name of resources and speed. For example, using Xilinx's DSP48E blocks in PL, one can perform a multiply of a 25 and 18-bit twos-complement number and accumulate with another 25-bit number in 5 clock cycles with one block. To perform the same calculation using floating-point requires 10 clock cycles and four of these same blocks [31], [32]. Thus, when the precision is less critical than speed and resources, one may very well opt for fixed-point operations.

D. Understanding Sampling and Discretization Methods

While there are many ways to discretize a linear model, the Zero-Order Hold Equivalent (ZOH) [4] has been the most used form for many years. As it is the default method for Matlab's `c2d()` function, this dominance has only increased. The ZOH equivalent provides exact matches at the sample instances, but with it comes a loss of physical understandability [18]. Its use only makes sense when one is discretizing the entire plant model in one step. However, as John Madden famously taught, "One size doesn't fit all" [33].

For example, most proportional, integral, derivative (PID) controllers are discretized using a backwards rectangular rule (or backwards rule, BR) equivalent [18]. This will be discussed in Section VIII-E, but it is worth noting that the only place in Matlab where one finds the BR built in is in the PID design tools. Matching higher end resonant structures is far more intuitive if one breaks them into second-order transfer function blocks (biquads). For such lightly damped second order sections, discretizing them with pole-zero matching is not only highly accurate, but preserves the physical intuition for each biquad (Sections VIII-G and VIII-H). If one wishes to model the rigid body modes of a system in such a way as to directly extract both position and velocity directly from the discrete-time state space, a trapezoidal rule (TR) equivalent makes a lot of sense (Section VIII-I).

Finally, while the "sampling fast" mantra does bring a lot of benefits, it can also bring forth several issues. The first is that if one samples so rapidly that the measurement and quantization noise is larger than any signal change in the sample instant, the signal to noise ratio (SNR) of our measurements drop, with consequentially bad results. Secondly, because all discretization is calculating or approximating some version of e^{sT_S}, the shrinking value of T_S will squeeze more poles and zeros into a pack around $z = 1$. This may not matter when the numbers are represented in double-precision floating point, but for real-time systems with single-precision floating point or even fixed-point coefficients, this can be disastrous. Trimming a few bits of the numeric representation can flip poles and zeros from inside of to outside of the unit circle in unpredictable ways [34]. Several schemes have been proposed to shift the design space back to a more continuous-time like representation, including the δ parameterization [35], [36] and the τ parameterization [37]. The Δ coefficients used in the Multinotch (Section VIII-G) [34], change the coefficients to be more accurate but do not alter the signal space. For biquad structures, Δ coefficients provide greater accuracy of coefficients [38] but do not do anything about signal growth. The signal growth limiting properties of the δ parameterization can actually be traced to the fact that T_S ends up scaling many of the signals. As this shrinks, it compensates for having more values in the accumulator [37], [39]. The issue with this scaling by T_S is negligible when the numbers are represented in floating point, but potentially disastrous when T_S is in an unscaled fixed point number. All of this is to say that "sampling fast" should be done fully aware of the potential downsides.

E. PIDs

While many controls researchers see the PID controller as "Brand X" controller against which to compare their research [18], it is still the case that most practical controllers are (or start with) some variant of a PID. There are many good discussions on various forms, aspects, and tuning of PID controllers [40], [41], [42], [19], we will focus here on some of the important computational features. One point to be clarified is that while PIDs are considered "standard" there is no one standard PID form. There is an attempt to consolidate many of the forms into one of four different archetypes in [43], and [44] adds the ISA standard form PID equation [45]. We will focus on one form from [43] that has nice discretization properties.

Design: Modern PIDs are typically designed in continuous time and implemented in discrete time. Almost universally, discretization is done using the BR, where

$$s \longrightarrow \frac{z-1}{T_S z} = \frac{1-z^{-1}}{T_S}, \text{ and} \tag{9}$$

T_S is the sample period. Consider one continuous-time form, *Explicit Time with No Derivative Filtering* from [43]:

$$C(s) = K_P + \frac{K_I}{T_I s} + K_D T_D s, \tag{10}$$

where K_P, K_I, and K_D are the proportional, integral, and derivative gains, while T_I and T_D are integration and differentiation times. Setting $T_D = T_I = T_S$ and applying the BR from (9) yields

$$C(z) = K_P + \frac{K_I}{z-1} + K_D \left(\frac{z-1}{z} \right). \tag{11}$$

We see that modulo the T_I and T_D factors there is a tight correspondence between (11) and (10). Using this parameterization and the BR has not only led to a very intuitive correspondence between the continuous and discrete-time PID parameters, but it has also made the non-proper derivative term in (10) proper in (11). In essence, the conservatism of the BR has inserted a needed low-pass filter. Operationally,

this means that the discrete PID is internally stable, so long as the integrator is not operated when the actuator is in saturation. This last issue is one of the likely reasons why PIDs are broken out from the rest of the filter blocks in most industrial systems. Being able to isolate the integrator for anti-windup purposes is extremely helpful.

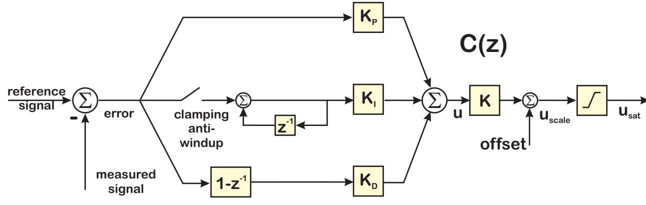

Fig. 22. Conditional integration/integrator clamping

Anti-Windup: We know from the Final Value Theorem that in order to track an input step with 0 steady state error, we need an integrator in the forward portion of the loop [11]. The issue for an integrator in the controller is that it is only stable in the presence of the feedback loop. Saturation – typically at the actuator – breaks that feedback. The breaking of the loop can cause a buildup of error in the integrator (wind up) causing the actual error to take much longer to settle when the system comes out of saturation.

An advantage of the PID structure is that the integrator can be isolated so that anti-windup schemes can be implemented. The workaround of integrator anti-windup involves some method of detecting the saturation and then using this to change/limit the input to the integrator. The two prevalent methods are back-calculation and conditional integration (also known as integrator clamping). Back calculation aims to remove from the integrator input a portion of the controller output that did not get past the saturation block. If properly scaled, this should drive the input to the integrator to 0, eliminating the windup. Most descriptions of this seem to focus on the integrator or the PI action, giving the impression this is usually reserved for systems where the D portion is not enabled. Back calculation has an advantage in that it can be readily applied in continuous or discrete-time implementations. Conditional integration, diagrammed in Figure 22, implements a decision tree to zero out the input to the integrator during saturation. As such, it is more easily understood with discrete-time implementations. It also has the advantage that even in saturation, input to the integrator may be allowed if the sign of the error is different from the output of the integrator, thereby moving away from saturation.

Derivative Filtering: Depending upon the application, various forms of filtering are often recommended. When the plant dynamics are substantially slower than the computation, one can apply fairly aggressive low-pass filtering to the entire PID to limit electronics noise while not affecting the needed closed-loop performance [41]. For higher speed systems such as mechatronics, we may stick with only filtering the derivative term [18], [43].

The point of this is that the most common PID blocks should be implemented in their own special filter subroutine. The form of discretization matters and can affect the intuition one keeps about the overall control system. Furthermore, this block can contain some anti-windup code, not found in most other filter blocks.

One more feature of integrators is that even without saturation and windup, their internal signals can get quite large compared to other filter signals. When using fixed point number formats, we often need to allocate extra bits just for the integrator accumulation.

F. Filter Structures and Latency

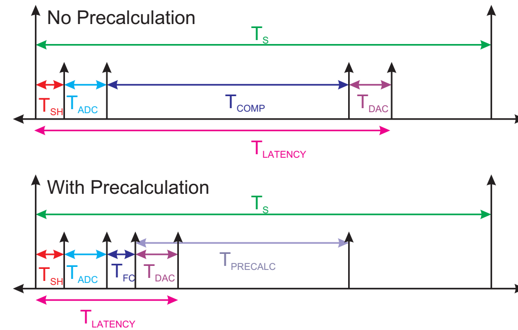

Fig. 23. Input and output timing in a digital control system. The top drawing is without precalculation; the bottom drawing is with.

As Section IV-A showed the phase-margin killing effects of careless selection of anti-alias filters, this section deals with computational latency. In particular, Figure 23 illustrates how the lack of precalculation makes the closed-loop latency dependent on the controller filter size. Restructuring the calculation to push as much as possible into precalculation makes the computational latency fixed and shorter. It is relatively straightforward to apply precalculation on a controller implemented as an IIR filter as in Figure 24, but polynomial filters have poor numerical properties, particularly when the filter has lightly damped poles and zeros. Since these are common in mechatronic systems, we want to implement our control filters using a biquad cascade that has better numerical properties than a polynomial filter [46].

G. The Multinotch

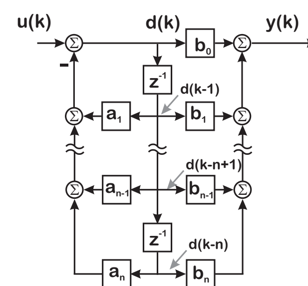

Fig. 24. n^{th} order polynomial filter in Direct Form II configuration [47].

The development of the Biquad State-Space (BSS) starts with the Multinotch (MN), a way of turning a polynomial form IIR filter as diagrammed in Figure 24, into a cascade of biquads with the direct feedthrough coefficients factored

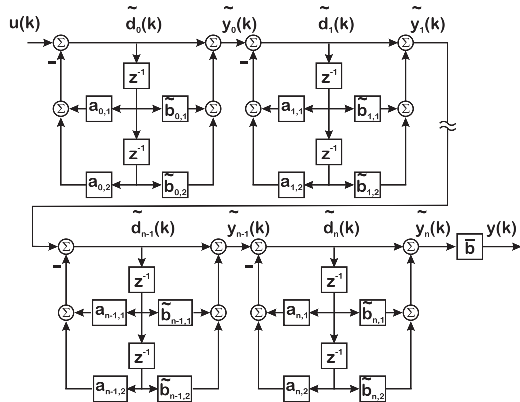

Fig. 25. The updated biquad cascade, with factored out b_0 terms.

out to the end, as shown in Figure 25 [34], [46]. With this factorization, one can discretize each biquad individually so that the discretized biquads have a one-to-one correspondence with the continuous-time biquads. By judicious choices of which poles and zeros from the physical model are assigned to each biquad, one can minimize effects of the signals of any one biquad on the others. The Multinotch is a highly efficient digital filter because it not only has greater numerical stability than standard polynomial and state-space forms, but also allows for precalculation of most of the filter, minimizing the latency between reading a sample and responding to it (Figure 23). This particular form of the direct feed-through scaling allows for precalculation, but others are available if we want the internal states of the filter to be scaled as they are in the system.

H. The Biquad State-Space (BSS)

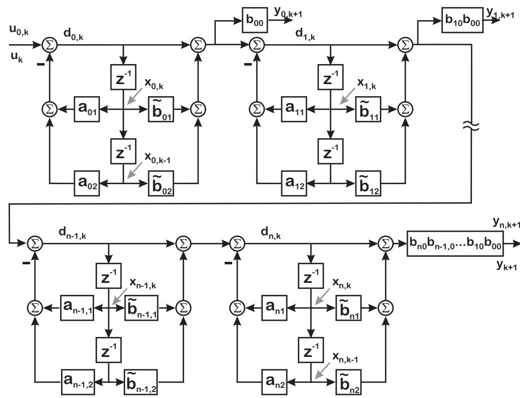

Fig. 26. The updated discrete biquad cascade, with factored out $b_{i,0}$ terms and scaling the output of each block.

One of the easiest ways to clear a room full of practicing mechatronic control engineers is to suggest that they employ state-space methods for the control of their structure with many lightly damped resonances. State-space models of highly flexible systems can present severe numerical issues. The models derived from physical principles often lack structure. Canonical form models, are compact, but obscure any physical structure and can have coefficients that are highly sensitive to model parameters. What is needed is a

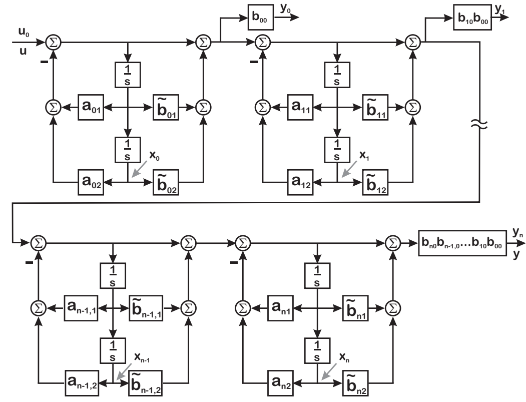

Fig. 27. The analog biquad cascade, with factored out $b_{i,0}$ terms and scaling the output of each block. This is completely analogous to the digital form of Figure 26.

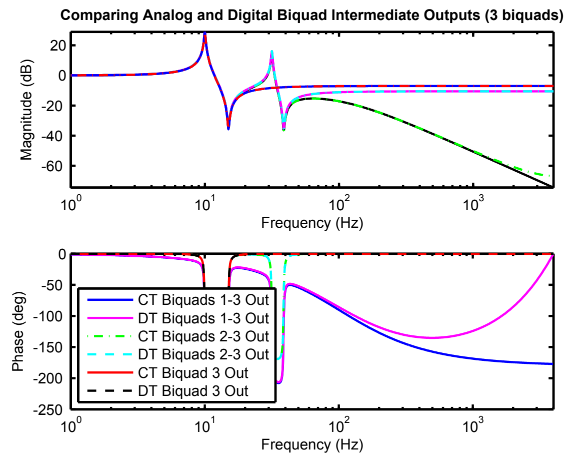

Fig. 28. BSS with three biquads including a low-pass filter in Biquad 1. This plot compares the Bode responses of the individual CT and DT biquad sections. The outputs of biquad 3 and biquad 2 show the magnitude and phase flattening out at high frequency (due to the matched number of poles and zeros). Once the response of biquad 1 is added in, we see the low pass roll off. At each biquad output, the match between continuous and discrete responses is incredibly close, a unique and useful feature of this structure.

form that has the compact representation of the canonical forms, the physicality of the forms derived from physical equations, and maintain numerical accuracy and physical intuition, even after discretization.

The first of these is the Biquad State-Space (BSS) [48], [49], based on the Multinotch (MN) of Section VIII-G. The BSS captures the endearing characteristics of the Multinotch while providing the flexibility of model based control. A significant feature of the BSS is the ability to move easily between the states of the continuous and discrete-time forms.

The digital version of the BSS shown in Figure 26 looks very similar to the Multinotch, although as we are more focused on accurate modeling than precomputation, we scale the outputs of each biquad to get to the proper output states. This form results in a block upper triangular state transition matrix [48]. If one were to mistakenly substitute $1/s$ for z^{-1}, one would end up with the continuous-time structure of Figure 27. Furthermore, if one were to discretize the structure of Figure 27 one biquad at a time, then one would

end up with the structure of Figure 26, with the added advantage that the signals at the outputs of the biquads would correspond between the analog and digital versions. Figure 28 demonstrates this with a 3-biquad system, where one of the biquads implements a LPF [50]. Note the tight correspondence between the outputs of both analog and digital biquads. The roll up in phase in the DT plot is due to mapping the LPFs continuous-time zeros at $s = -\infty$ to $z = -1$.

I. Rigid Body Modes and the Bilinear State-Space (BLSS) Structure

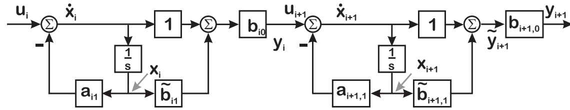

Fig. 29. Continuous-time bilinear state-space (CT-BLSS) form.

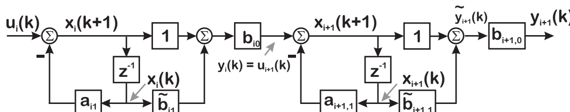

Fig. 30. Discrete-time bilinear state-space form (DT-BLSS).

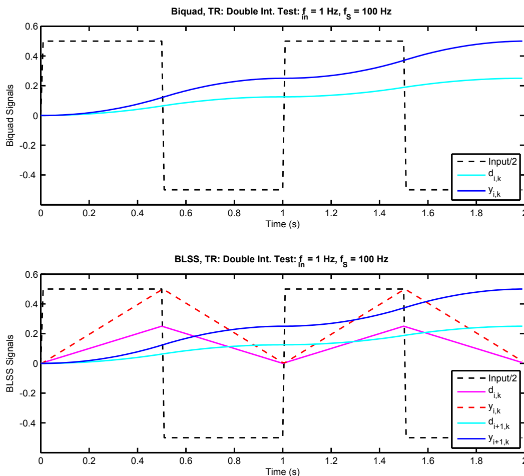

Fig. 31. Double integrator with square wave input. Implemented using a trapezoidal rule equivalent biquad (top) and BLSS (bottom).

For all the advantages of the BSS for flexible modes, we still need to find some way not only to represent rigid body modes, but also to have the internal states of those structures correspond to the internal states of the rigid body, e.g. velocity and position. Furthermore, we would also like that rigid body state-space structure to have an equivalence between the continuous and discrete-time forms. This is accomplished via the Bilinear State-Space (BLSS) structure [50]. Figures 29 and 30 show continuous and discrete versions of the BLSS [50] which accomplishes that, and can be combined with a cascade of biquads into one overall state-space structure. This is shown in the simulation of Figure 31. After all, it is a bit embarrassing to go through all the mathematical machinery of state space, and not be able to access the discrete states for both position and velocity.

IX. Example Bandwidth Ranges, Applications, and Platforms

In this as in many other sections, there are general statements based upon where the state of the art of technology ("Moore's Law"[5]) is at the time of the writing. We expect that the specific numbers and ranges will change over time, but the basic idea of what defines a range should remain. The best advice here seems to be to paraphrase Sun Tzu [51]: "Know your time constants, and know your dynamics, and you can close 100 loops without disaster."

We will provide some platform examples, in a direction of ever increasing speed. Of common interest is the tradeoff between preemption – being able to run multiple tasks by multiplexing – and timing certainty. As we move down the list, we move towards more and more dedicated hardware to a specific computation and less opportunities for preemption. The second trend as we move down the list is that there are a decreasing number of electronic components between the physical system and the electronic computation. The increase in speed comes along with a potential increase in cost and almost always decrease in flexibility. However, the march of technology means that as the years pass, more physical systems with faster time constants move up these layers. One path through the different computation technologies is:

On a Linux Thread: Linux is a free, customizable operating system. Among the many variants are complex versions for servers and simple versions that run much of the embedded systems in the world. It is not uncommon to select a Linux thread to run some of the slower real-time applications. Because a real-time Linux thread can be given higher priority than Non-Real-Time threads, this can reduce the delay and jitter to acceptable levels for relatively slow applications.

On a Real-Time Operating System (RTOS): An RTOS is the next level up in capability. This is a compromise between maintaining a preemptive operating system with its ability to manage memory, communications, and scheduling, with the priority of real-time tasks. Many applications that use digital signal processor (DSP) chips will fall into this area.

On a bare metal (minimal/no OS) chip: In this case, our need for precise control of timing has overridden our desire for the convenience of an operating system. Many of these processors are relatively small and made for running a single process with minimal interruptions.

On an FPGA or other PL: Programmable Logic (PL) started as a way of prototyping custom integrated circuits (custom ICs) or of generating the glue logic that tied many processing chips together. As the capabilities have grown, so have the PL chips and market. What FPGAs give is a chance to generate custom hardware processors for specific mathematical tasks. Instead of multiplexing these tasks in time as one would have to do on a single processor, they are multiplexed in space on the FPGA. Some algorithms are relatively simple and can be implemented very efficiently in relatively simple, customized logic. Thus, we are not wasting the processing power of a large processor by having it switch

to many simple tasks.

On custom digital or mixed signal IC: A custom chip can include mixed analog and digital signals without incurring any of the (albeit small) overhead of an FPGA chip. The cost of laying out a custom chip means that this solution is only feasible for either high-cost applications or for mass market applications. In either case, something has to make up the cost of chip layout to gain that extra speed.

In analog electronics: The fastest speed systems often require us to give up on digital methods in the Hard-Real-Time layer. With this move to fully analog implementation comes the loss of flexibility, reproducibility, and updatability that are a key advantage of digital methods.

An alternate view starts with the general speed ranges that we can group by sample rates, often with overlapping edges:

Low End Speed $f_S \leq 1Hz$: The typical applications include thermal systems, pressure control, biological reactors, and chemical process control. Such slow systems can often be handled as a Linux thread with the API approach of Figure 20 or the advanced tools method of Figure 21.

Next Level $1Hz \leq f_S \leq 100Hz$: In this zone we currently have rigid systems; typically medium to large mechatronic systems, or small toy class systems. We might find personal robots in this region. While the sample rates are pushing the boundaries, one might still find some of these handled by Linux threads. The more complex or safety critical systems might run on an RTOS, and the chips involved might move from low end processors to DSP chips.

Next Level $10Hz \leq f_S \leq 50kHz$: Typical applications here might include fast rigid body and/or mechatronic systems, such as motion stages, fast robotics, flight, safety, automotive, and disk storage. These would be usually handled with an RTOS or bare metal computing environment. The hardware has moved away from conventional or inexpensive processors fully in the range of DSP and/or FPGA chips.

The Need for Speed $50kHz \leq f_S \leq 50MHz$: Typical applications requiring these sample rates might include high-end instrumentation, mid-level electronic test, and high-speed small mechatronics, such as Atomic Force Microscopes (AFMs). These might use DSP chips at the lower end FPGA implementations at the higher end, and run with minimal operating system interference.

X. Business Models and Bandwidth

Control theory might seem unified, but the space to implement is dramatically varied. As was described in Section IX, the physical system time constants are a main determinant of the required sampling rates, and these in-turn affect the version of the Three-Layer-Model from Section VII-A that we will program against.

This section is about how much computing power we can afford to apply to any given control application. The computing cost limits may or may not affect the implementation of our control algorithms, but it is good to have some idea when those limits occur. The earlier Sun Tzu paraphrase applies well here.

For very slow applications (e.g. pressure, temperature, or chemical and biological process control) the computer is so much faster than the process dynamics that we can forget about latency inside the digital unit. This is where lots of compute intensive learning algorithms are first tried. A great example of this is Model Predictive Control [52], [53], which finds a natural home performing optimization between the relatively-slow sample instants of chemical process control systems. At the same time, the types of dynamic structures for which the control system must compensate are different. Chemical engineers rarely think about resonances, but are keenly aware of transport delay and the limitations on their sensors and actuators. Similarly, the long time constants mean that frequency domain measurements are almost meaningless to this group. As a result, the raw processing costs will be relatively low. Such bandwidth requirements can often be met with the API method diagrammed in Figure 20. Where the money will be spent is on the input and output signal chains, where – depending upon the system – the environment in which those electronics operate often determine the cost.

At the other end of the spectrum are intense, expensive applications (e.g. fighter planes, space launch and spacecraft, wafer scanners). For these systems, the cost of processing is a tiny part of the machine cost. These types of systems might have substantial numbers of lightly damped dynamics, as well as substantial instrumentation challenges. Time constants may be short, prompting higher sample rates and more stringent computational requirements. However for these systems, the engineering teams are large and (relatively speaking) resources are flush. Latency, noise, performance limits all matter, but the proverbial checkbook is open. These systems are characterized in part by being so expensive that each device can be tuned by a team of engineers. Such systems can afford high-end, high-cost processing, and plenty of engineers to design and program each of the layers in Figure 19.

Perhaps the largest price-performance demands come from consumer level systems. This may present the largest challenge and opportunity. The prevalence of feedback-based devices in our everyday use require a lot of performance in relatively inexpensive processing solutions. These must be increasingly self-tuning and self-diagnostic. The low unit cost mandates that we cannot afford to have engineers touch every device. That being said, their penetration to the public is far more visible, so reliability is critical. The path for these computation systems is usually to start with more complex, powerful chips and algorithms in the early test phases, and then port the simplified versions down into the low-cost hardware. While the Hard-Real-Time layer is made as efficient as possible, it is common in these systems to thin out the top two layers to reduce cost. After all, when it is cheaper to replace a device than to diagnose and fix it, this makes economic sense. (We are ignoring the more complete accounting costs on the environment due to throwing away, rather than repairing or recycling broken devices. We do not endorse this incomplete accounting; we simply view

it as a current – and flawed – business practice for many operations.)

Consumer level devices where the feedback loop is used as a selling point include fuzzy logic washing machines and rice cookers. Two salient features of these are that the system dynamics are relatively simple and benign and that the fuzzy logic control is advertised as a feature. They are highly unlikely to do damage to anyone or anything even with a flawed or failing feedback loop. Such systems feature hard limits on internal temperature, cooking or cleaning time, and the like, so that even if the feedback fails, the system will shut down with nothing more than a ruined dinner. If one views fuzzy control as more of a control implementation than a control design method [54], then we can realize that a key feature of these devices was the implementation of feedback on a benign but improvable process.

Few consumers are aware of the key role of feedback in their hard disk drives and optical disks [55], [56], [57], [58]. At the same time, devices that visibly depend upon feedback for their fundamental operation are rapidly becoming prevalent in our society, from all types of robots, to drones, to self-driving modes in vehicles, to self-driving vehicles themselves. These are not systems with benign dynamics and so the proper implementation of feedback on a low-cost embedded platform is critical. As control engineers we need to understand both how to fit our algorithms into such inexpensive platforms and how to justify our push for some computational head space in those systems for improved diagnostics and code revisions. The team will likely only have one or two engineers with any knowledge of feedback. Being able to communicate design considerations to everyone from business types, to chemists, to software designers is a critical skill.

Finally, with all the discussion of machine learning (ML) and artificial intelligence (AI) in the public consciousness, we can neither shy away from these discussions, turn off our connection to the laws of physics, nor pout in the corner because we are not getting the same attention. Anya Tsalenko, an expert in machine learning, big data, and artificial intelligence at Agilent Labs often points out that the main advancement between the neural network methods of the 1980s and 1990s and today are a massive increase in the amount of training data and computer power available now. What was once a parallel scheme run on a single Intel processor is now a massive parallel scheme tuned with terabytes of data on GPUs (Graphics Processing Units) and implemented in parallel on FPGAs. Still, the demonstrations all seem to focus on systems for which the dynamics are orders of magnitude slower than the processing, obviating the need for discussions of jitter and latency through the system. We have tried to make the case in this tutorial that control engineers cannot ignore these factors in how we think about computing. This is a function of the problem itself, rather than our choice of computing platforms. As ML/AI systems are increasingly applied in real-time feedback loops, they will likely face the same computing issues discussed here.

XI. Summary

This paper has tried to show how the computation in a feedback loop extends far beyond the processor itself, and into four signal chains in the loop: the plant's "computation", the input signal chain, the output signal chain, and the computer itself.

Each of these is a potential source of delay, noise, and jitter. They can each wreck the performance of even the best control algorithm. If we simply block these together into a single discrete-time plant model, we loose the opportunity to optimize each of these signal chains so as to improve the overall performance of the feedback loop.

Furthermore, we cannot separate these computation chains from the time constants of the plant itself. We have tried to show how the pain points for a control engineer working on a high-speed mechatronic system are likely very different than those for a control engineer working on a bioreactor. What we have tried to show is that the principles underlying each of these signal chains is the same across applications, which allows us to gain understanding of where the pain points will be as we move our control knowledge between applications. Understanding the role of real-time computation for different time constant problems is critical.

Once we get into the actual computation, we need to understand how different aspects of the problem are addressed by different layers of computation. The proposed Three-Layer Computer Model helps us understand how to program for different parts of the loop.

Finally, none of this can be separated from the business model versus bandwidth tradeoffs. How much computing we can afford to apply on any given control problem will often limit what we can do with our algorithms.

References

[1] H. W. Bode, "Relations between attenuation and phase in feedback amplifier design," *Bell System Technical Journal*, vol. 19, pp. 412–454, July 1940.

[2] G. Stein, "Respect the unstable," *IEEE Control Systems Magazine*, vol. 23, no. 4, pp. 12–25, August 2003.

[3] D. Y. Abramovitch, "A tutorial on PES Pareto methods for analysis of noise propagation in feedback loops," in *Proceedings of the 2020 IEEE Conference on Control Technology and Applications*, IEEE. Montreal, Canada: IEEE, August 2020.

[4] G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital Control of Dynamic Systems*, 3rd ed. Menlo Park, California: Addison Wesley Longman, 1998.

[5] Wikipedia. (2022) Moore's law. [On line; accessed September 21, 2022]. [Online]. Available: https://en.wikipedia.org/wiki/Moore's_law

[6] ——. (2022) Newton's laws of motion. [On line; accessed September 21, 2022]. [Online]. Available: https://en.wikipedia.org/wiki/Newton's_laws_of_motion

[7] D. Y. Abramovitch, "Trying to keep it real: 25 years of trying to get the stuff I learned in grad school to work on mechatronic systems," in *Proceedings of the 2015 Multi-Conference on Systems and Control*, IEEE. Sydney, Australia: IEEE, September 2015, pp. 223–250.

[8] Wikipedia. (2018) Padé approximant. [Online; accessed June 6, 2018]. [Online]. Available: https://en.wikipedia.org/wiki/Pade_approximant

[9] K. Ogata, *Modern Control Engineering*, 3rd ed., ser. Prentice-Hall Instrumentation and Controls Series. Englewood Cliffs, New Jersey: Prentice-Hall, 1970.

[10] ——, *Discrete-Time Control Systems*, 2nd ed. Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[11] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 5th ed. Upper Saddle River, New Jersey: Prentice Hall, 2006.

[12] Wikipedia. (2022) Phase noise. [On line; accessed September 28, 2022]. [Online]. Available: https://en.wikipedia.org/wiki/Phase_noise

[13] ——. (2022) Jitter. [On line; accessed September 28, 2022]. [Online]. Available: https://en.wikipedia.org/wiki/Jitter

[14] S. D. Ruben, "Respecte the implementation: Using NI myRIO in undergraduate control education," in *Proceedings of the 2016 American Control Conference*, AACC. Boston, MA: IEEE, July 6-8 2016, pp. 7315–7320.

[15] Keysight Technologies, "What is the difference? between an equivalent time sampling oscilloscope and a real-time oscilloscope," Keysight Technologies, Santa Rosa, CA USA, Application Note 5989-8794, April 9 2021, [On line; accessed September 28, 2022]. [Online]. Available: https://www.keysight.com/us/en/assets/7018-01852/application-notes/5989-8794.pdf

[16] B. Widrow, "A study of rough amplitude quantization by means of Nyquist sampling theory," *IRE Transactions on Circuit Theory*, vol. 3, pp. 266–276, 1956.

[17] D. Y. Abramovitch, "Determining Kalman filter input noises using PES Pareto," in *Proceedings of the 2021 American Control Conference*, AACC. New Orleans, LA: IEEE, May 2021, pp. 4292–4298.

[18] ——, *Practical Methods for Real World Control Systems*. Self, December 1 2022.

[19] K. J. Åström and B. Wittenmark, *Computer Controlled Systems, Theory and Design*, 3rd ed. Englewood Cliffs, N.J. 07632: Prentice Hall, 1997.

[20] D. Abramovitch, "The Banshee Multivariable Workstation: A tool for disk drive servo research," in *Proceedings of the ASME Winter Annual Meeting*, ASME. Anaheim, CA: ASME, November 1992.

[21] *HP 3563A Control Systems Analyzer*, Hewlett-Packard, 1990.

[22] *z-Domain Curve Fitting in the HP 3563A Analyzer*, Hp 3563a-1 product note ed., Hewlett-Packard, 1989.

[23] *Control System Development Using Dynamic Signal Analyzers: Application Note 243-2*, Hewlett-Packard, 1984.

[24] *Curve Fitting in the HP 3562A*, Product note HP 3562A-3 ed., Hewlett-Packard, 1989.

[25] S. B. Andersson, "Lessons from the advanced tool world," in *Proceedings of the 2023 American Control Conference*, AACC. San Diego, CA: IEEE, May 31–June 2 2023.

[26] Xilinx. (2022) Xilinx adaptive SoCs. [On line; accessed October 4, 2022]. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc.html

[27] Intel. (2022) Intel FPGAs and Soc FPGAs. [On line; accessed October 4, 2022]. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga.html

[28] S. D. Ruben, "Controller implementation via analog computers," in *Proceedings of the 2023 American Control Conference*, AACC. San Diego, CA: IEEE, May 31–June 2 2023.

[29] W. S. Nagel, A. Mitrovic, G. M. Clayton, and K. K. Leang, "Discrete input-output sliding-mode control with range compensation: Application in high-speed nanopositioning," in *Proceedings of the 2022 American Control Conference*, AACC. Atlanta, GA: IEEE, May 31–June 2 2023, pp. 4371–4376.

[30] W. S. Nagel and K. K. Leang, "Discrete input-output state-space models for real-time control," in *Proceedings of the 2023 American Control Conference*, AACC. San Diego, CA: IEEE, May 31–June 2 2023.

[31] *7 Series DSP48E1 Slice Users Guide*, Ug479 (v1.6) ed., Xilinx, August 7 2013.

[32] *LogiCORE IP Floating-Point Operator v6.0*, Ds816 (v1.2) ed., Xilinx, January 18 2012.

[33] J. Madden and D. Anderson, *One Size Doesn't Fit All*. Jove, October 1 1989.

[34] D. Y. Abramovitch, "The Multinotch, Part II: Extra precision via Δ coefficients," in *Proceedings of the 2015 American Control Conference*, AACC. Chicago, IL: IEEE, July 2015, pp. 4137–4142.

[35] R. H. Middleton and G. C. Goodwin, "Improved finite word length characteristics in digital control using δ operators," *IEEE Transactions on Automatic Control*, vol. 31, no. 11, pp. 1015–1021, November 1986.

[36] G. Li and M. Gevers, "Comparative study of finite wordlength effects in shift and delta operator parameterizations," *IEEE Transactions on Automatic Control*, vol. 38, no. 5, pp. 803–807, May 1993.

[37] D. Y. Abramovitch, "A comparison of the δ parameterization and the τ parameterization," in *Proceedings of the 2019 American Control Conference*, AACC. Philadelphia, PA: IEEE, July 2019.

[38] ——, "A comparison of Δ coefficients and the δ parameterization, Part I: Coefficient accuracy," in *Proceedings of the 2017 American Control Conference*, AACC. Seattle, WA: IEEE, May 2017.

[39] ——, "A comparison of Δ coefficients and the δ parameterization, Part II: Signal growth," in *Proceedings the 2018 American Control Conference*, AACC. Milwaukee, WI: IEEE, June 2018, pp. 5231–5237.

[40] K. J. Åström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*. ISA Press, 1995.

[41] ——, *Advanced PID Control*, ser. Oxford Series on Optical and Imaging Sciences. ISA Press, August 15 2005.

[42] T. Wescott, "PID without a PhD," *Embedded Systems Programming*, pp. 86–108, October 2000.

[43] D. Y. Abramovitch, "A unified framework for analog and digital PID controllers," in *Proceedings of the 2015 Multi-Conference on Systems and Control*, IEEE. Sydney, Australia: IEEE, September 2015, pp. 1492–1497.

[44] ——, "Thoughts on furthering the control education of practicing engineers," *IEEE Control Systems*, vol. 43, no. 1, pp. 64–88, February 2023.

[45] V. VanDoren, "Understanding PID control and loop tuning fundamentals," *Control Engineering Magazine*, 2023, [On line; accessed September 30, 2022]. [Online]. Available: https://www.controleng.com/articles/understanding-pid-control-and-loop-tuning-fundamentals/

[46] D. Y. Abramovitch, "The Multinotch, Part I: A low latency, high numerical fidelity filter for mechatronic control systems," in *Proceedings of the 2015 American Control Conference*, AACC. Chicago, IL: IEEE, July 2015, pp. 2161–2166.

[47] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, N. J.: Prentice Hall, 1975.

[48] D. Y. Abramovitch, "The discrete time biquad state space structure: Low latency with high numerical fidelity," in *Proceedings of the 2015 American Control Conference*, AACC. Chicago, IL: IEEE, July 2015, pp. 2813–2818.

[49] ——, "The continuous time biquad state space structure," in *Proceedings of the 2015 American Control Conference*, AACC. Chicago, IL: IEEE, July 2015, pp. 4168–4173.

[50] ——, "Adding rigid body modes and low-pass filters to the biquad state space and multinotch," in *Proceedings of the 2018 American Control Conference*, AACC. Milwaukee, WI: IEEE, June 2018, pp. 6024–6030.

[51] S. Tzu, S. B. Griffith, and B. H. L. Hart, *The Art of War*, reissue ed. Oxford University Press, January 1 1963, iSBN-10: 0195015401 ISBN-13: 978-0195015409.

[52] J. B. Rawlings, "Tutorial overview of model predictive control," *IEEE Control Systems Magazine*, vol. 20, no. 3, pp. 38–52, June 2000.

[53] R. R. Negenborn and J. M. Maestre, "Distributed model predictive control: An overview and roadmap of future research opportunities," *IEEE Control Systems Magazine*, vol. 34, no. 4, pp. 87–97, August 2014.

[54] D. Y. Abramovitch, "Some crisp thoughts on fuzzy logic," in *Proceedings of the 1994 American Control Conference*, AACC. Baltimore, MD: IEEE, June 1994. [Online]. Available: dabramovitch.com/pubs

[55] ——, "Magnetic and optical disk control: Parallels and contrasts," in *Proceedings of the 2001 American Control Conference*, AACC. Arlington, VA: IEEE, June 2001, pp. 421–428.

[56] W. Messner and R. Ehrlich, "A tutorial on controls for disk drives," in *Proceedings of the 2001 American Control Conference*, AACC. Arlington, VA: IEEE, June 2001, pp. 408–420.

[57] D. Y. Abramovitch and G. F. Franklin, "A brief history of disk drive control," *IEEE Control Systems Magazine*, vol. 22, no. 3, pp. 28–42, June 2002.

[58] ——, "Disk drive control: The early years," in *Proceedings of the 2002 IFAC World Congress*, IFAC. Barcelona, ES: IEEE, July 2002.

[59] J. A. Chantel K. Lapins and K. K. Leang, "Real-time control of mobile robotic systems through the Robot Operating System (ROS)," in *Proceedings of the 2023 American Control Conference*, AACC. San Diego, CA: IEEE, May 31–June 2 2023.