2008 American Control Conference
Westin Seattle Hotel, Seattle, Washington, USA
June 11-13, 2008

Efficient and Flexible Simulation of Phase Locked Loops, Part I: Simulator Design

Daniel Y. Abramovitch

Abstract— **Although phase-locked loops (PLLs) are arguably the most ubiquitous control loop designed by humans, system theory analysis seems to lag behind the practice of implementation. In particular, full simulation of PLLs is rare. This paper will explain the reasons for this and offer an efficient and flexible simulator for PLLs. This part presents the simulator requirements and design. Part II [1] presents post processing methods and shows a design example.**

I. Introduction

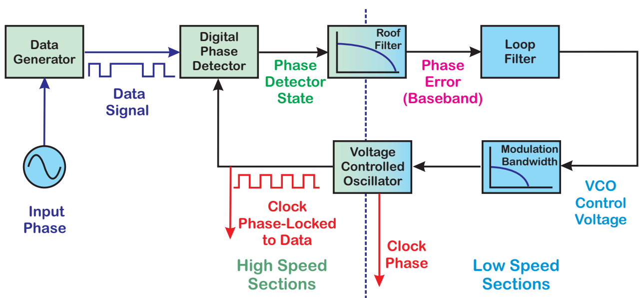

Fig. 1. Simulation block diagram for a classical digital phase locked loop. On the left side of the diagram: data, VCO, and phase detector simulated with component level blocks that are very efficient. On the right side of the diagram: filters and modulation bandwidth are simulated using designs from Matlab that are very flexible and derived from lab measurements.

Simulation of phase-locked loops varies dramatically depending upon the type of loop involved, but generally is beset by the fundamental issue that PLLs operate in two time scales. The first is the very fast time scale of the input signals to the phase detector (namely the reference or data signal) and the output of the oscillator (typically a voltage controlled oscillator (VCO) for analog systems and a numerically controlled oscillator (NCO) for digital systems). The second is the relatively slow time scale of the signal's phase; often called *modulation domain*. The actual loop itself operates in both domains, although examining the signals at various points in the loop would make one more apparent than the other. Because of these two domains, PLLs are inherently stiff systems, and these are difficult to fully and accurately simulate. In modern communication and computer systems, the clocks run in the multiple megahertz to gigahertz range while the loop bandwidth itself may be only several kilohertz.

The stiffness of the problem – the 3 to 6 order of magnitude difference in time scales – provides significant difficulties in achieving all of these objectives. Generally, a simulation step size which is small enough to clearly observe the dynamics of the phase detector makes it difficult to observe the dynamics of the entire loop. In particular, the simulation time needed to observe the baseband behavior of the phase error and the VCO phase leads to long simulation times and massive amounts of stored data.

For this reason, it is quite typical to break up the simulation of PLLs into two pieces. A given simulator will work in one frequency range.

- First, a simulator is written for the high speed sections. A time domain/signal space simulation is done on loop components using very high frequency signals. That is, the data input, VCO, and phase detector are simulated together with no feedback from the output of the phase detector to the VCO. Instead, various phases are introduced for the various types of input data to verify that the phase detector behaves as desired.
- Once the behavior of the phase detector has been verified, a modulation domain/phase space simulation at relatively low frequencies on the baseband model of the complete PLL is done. The VCO is replaced by an integrator and the phase detector is replaced by its baseband model. Instead of an actual data and clock input to the phase detector, only an input phase modulation and a VCO phase are used. This simulator is used to analyze the loop properties.

This two step solution eliminates most of the issues with the stiffness of the PLL, but it suffers from the fact that neither simulation gives a picture of the complete model. This two piece simulation method breaks down when the loop bandwidth is very high and when the designer is concerned about interactions between high and low frequency sections.

- The effect of the input signal (not just it's phase) can have a strong effect on the behavior of PLLs. For example, in a modern communication signal using NRZ data, the absence or presence of a transition indicates a bit of data. However, digital phase detectors respond differently to data streams with lots of transitions as opposed to those with few. Thus, the effect of the data on the phase behavior cannot be studied.
- The high frequency components have parasitic signals present. For example, the simplest PLL phase detector will respond to an input signal at frequency f_0 by producing a baseband component and a signal at $2f_0$. The modulation domain simulation cannot include this effect and thus, the effectiveness of the various filters

Daniel Abramovitch, Nanotechnology Group at Agilent Laboratories, 5301 Stevens Creek Blvd., M/S: 4U-SB, Santa Clara, CA 95051 USA, danny@agilent.com

in the loop cannot be tested.

Furthermore, there are further issues with not simulating the entire loop over both frequency bands, particularly in cases where the time constants of the data and phase are not so separated. This can happen when either the loop bandwidth is very high, so as to be within a factor of 20 of the signal frequency. Furthermore, digital phase detectors are only linear in the baseband (signal phase space). The interaction of their dynamics with that of the loop are worth studying. Finally, Bang-Bang phase detectors have no limit on their bandwidth. Thus, it is hard to conclusively state that their dynamics are separate from the filter dynamics. The signal phase space model of the Bang-Bang phase detector is a relay (signum function) and this model also has frequency content up to high (infinite) frequency. Thus, a complete loop simulation helps us to truly understand these.

Modern computers should allow us to think differently. There is no reason why a full time domain simulation cannot be run for a large number of samples so that both the high frequency behavior and the low frequency behavior can be studied. This paper presents such a simulator.

II. Simulator Requirements

It is common for scientists and engineers to generate simulations of their specific systems in a single, monolithic program. However, to make this simulator generally applicable to multiple types of PLLs, there were several requirements that would have to be addressed.

Looking at Figure 1, there are high speed blocks and low speed blocks. Each of these could represent multiple possible technologies. For example, a phase detector may be analog or digital, may be a simple XOR gate or a more complicated clock-data recovering phase detector. The data generator may need to generate digital or analog data, and may need to model various disturbances to the phase of that data. The loop designer will want to model multiple types of filters for loop shaping. Thus, a simple set of requirements for a simple yet generally useful simulation include:

- **Modularity:** An obvious requirement is modularity, in the high speed components, the low speed components, and the hybrid components (the roof filters and the VCO models). Inherent in these are interfaces to the other blocks that are consistent enough so that exchanging these components is straightforward.
- **State knowledge:** One caveat of modularity is that each of these modules must be able to preserve their own internal state, so that information is not lost between time steps. Without this capability, the simulator must either work on a large set of global variables or must pass a large set of parameters to each of the routines. Furthermore, the size of the parameter list needs to be adjustable, since the coefficients and state information for a tenth order filter needs more memory than that for a second order filter. (This may seem obvious once stated, but is the kind of thing that has to be considered explicitly when building such tools.)
- **Extensibility:** There should be the ability to add new phase detectors and signal generators that are extensions of older ones. Part of this comes from modularity, but another part comes from an iterative design approach.
- **External post processing capability:** While there is a need and desire for the simulation loop to run quickly to generate the closed-loop data, there would be tremendous benefits from making the simulation data accessible to other environments for post processing.
- **Filter design:** From a loop shaping perspective, we want flexibility in the filter design for loop filter. Once the high speed components are set, a good simulation will allow the designer to drop new filter designs into the loop easily.

Figure 1 shows a block diagram for a simulation of a clock-data recovery (CDR) loop. In this case, the loop to be simulated is a classical digital PLL (CDPLL) [2], which receives digital input and a square wave clock as inputs to the phase detector, but does all of the filtering and clock generation using analog components. This particular simulation example provides a series of issues that must be dealt with in PLL simulation:

- The data input signal and the VCO generated clock must approximate a set of digital signal values.
- Any data generated for the simulation must run in the time domain.
- The phase detector must respond to these digital levels and produce an output. The output of the phase detector will generally have a baseband component and a component of higher frequency than the inputs.
- The simulation must have a small enough time step to accurately represent these signals. In particular, for most digital phase detectors, the minimum resolution of the phase will be directly proportional to the minimum time step of the simulation.
- The filtering and the control of the VCO generally take place at significantly lower frequencies than the operations of the phase detector and data generator.
- Simulations must be run long enough to allow the dynamics in the modulation domain to be examined.

III. A Hybrid C/C++/Matlab Simulator

For reasons mentioned in Section I, most standard simulator packages fall short of what is needed. The larger, detailed circuit simulation packages that can simulate each component in detail are far too cumbersome for a full loop simulation of any length. The modulation domain packages lack the time domain information.

It seemed that the only way to be able to do a full closed-loop simulation of the time domain response was to generate one in some high level language. However, it seemed that in order to get the loop efficiency needed, CAD environments and block diagram simulation tools had to be avoided.

The compromise was to write a modular simulation in C++. This was chosen because the author already had some useful data generation routines in C, but found that in order to get the extensibility needed for the simulation, an object oriented approach was needed. Furthermore, the object oriented approach provided the ability to preserve the state of each of the components, since each of the components could be build upon a class library that had its own static variables.

Thus, the advantages of the approach chosen were:

- The simulation was built on simple component models in C++. This made the actual loop execution very efficient and fast.
- High speed components such as phase detectors could be built from simple components in a class library. This allowed a large set of loop types to be simulated without altering the specific structure of the loop.
- Loop filter designs could be imported from Matlab via a simple ASCII file interchange format.
- Loop simulation data could be saved to Matlab for post processing using the Matlab API. Extensive data analysis and plotting were then done in Matlab.

With this architecture, we can do complete loop simulations in the time domain without having to simplify the model any further. This allows simulation of long runs of data in reasonable time. The long runs of data allow the PLL simulation results to be compared to long measurements of hardware in the laboratory. Specific examples of tests that can be run are:

- Measuring data induced noise in the PLL signals. Specifically as PLLs are used in jitter measurements, it is helpful to know how the data induced noise in the PLL affects the overall measurement of jitter in a signal.
- Measuring the effects of loop design on jitter of the PLL signals, both those passed by the PLL from the input and those generated by the PLL itself.
- Measuring the spectral resolution of a particular PLL architecture. That is, the ability to estimate the spectral resolution of a given PLL architecture, and how it is affected by input noise and the data patterns.

These are all tests that cannot be run on a conventional PLL simulation system, simply because the overhead of the simulation makes long data runs impractical. At the other end of the spectrum, simple one-off simulations lack the flexibility to test a variety of architectures.

Thanks to the object oriented nature of the simulator and the use of Matlab to generate filter designs (described in Section V-A), it is possible to add almost any of the PLL components of a PLL block diagram to this simulator, albeit with a bit more work. This may involve more work than adding a block to a simulator such as Simulink, but the block once added will have minimum overhead. In the following sections, features of this simulator, as they apply to different loop blocks, will be described.

IV. Fast Components

The class libraries for the fast components are built on the idea that the simulation will be more accurate if it is built upon the actual solution of simple differential equations, rather than on doing numerical integration of those components. Thus, devices such as digital phase detectors are composed of simple component class library. There are classes for logic gates, latches, and flip-flops. These include the ability to add first order dynamics (transport lag and propagation delay) to each of these objects. The primitive objects are run as ideal logic elements, followed by code that propagates the true state. This allows us to add realism at the lowest logic level of the simulation, if that is needed. The transport lag and propagation delay are parameters of the logic family, which can be set in the initialization of the simulator. Furthermore, the time constants were adjusted for digital logic, in that rather than representing the time it takes to go e^{-1} times the distance to go, it represents the 50% distance from one state to another[1].

A. Phase Detectors

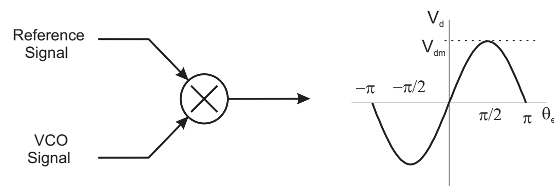

Fig. 2. Block diagram for a simple mixing phase detector, most often found in analog PLLs and often mimicked in software PLLs [3].

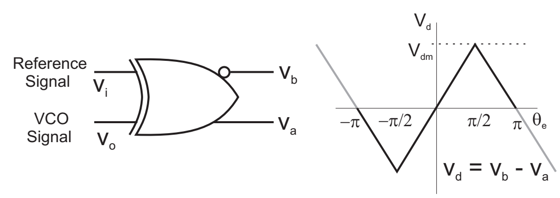

Fig. 3. Block diagram for a simple XOR phase detector. The XOR phase detector behaves as a mixer would behave if the inputs to the mixer were saturated. Thus, it is the digital signal "analog" of a sinusoidal phase detector.

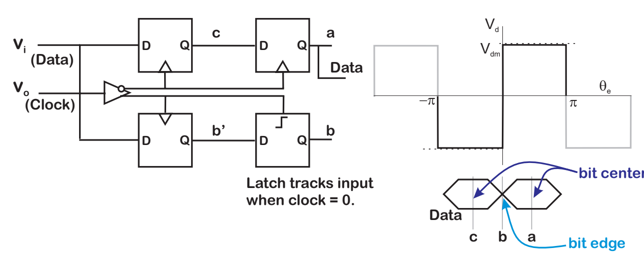

Fig. 4. Block diagram for a Bang-Bang phase detector used in clock-data recovery PLLs.

The first step for any PLL design is the phase detector. Without a means of detecting the relative phase of an input signal and some sort of clock, there is no PLL. So, the design of phase detectors is critical. As described in [2], phase detectors vary greatly by the type of input signals that

[1]This suggestion made by Rick Karlquist of Agilent Labs.

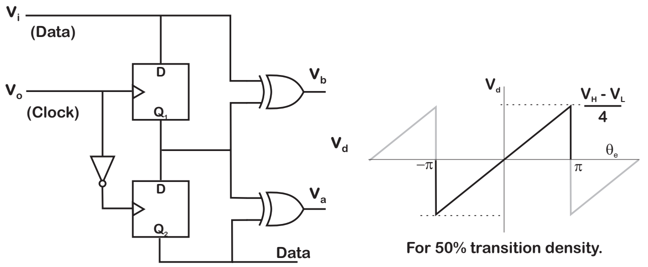

Fig. 5. Block diagram for a Hogge phase detector used in clock-data recovery PLLs.

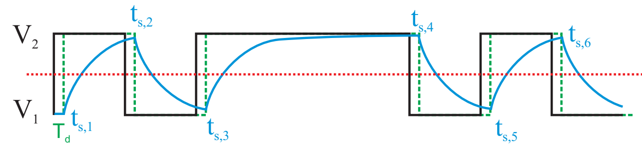

Fig. 6. Diagram of witching logic levels simulated in the circuit. The ideal switches are delayed, then passed through a first order low pass filter. The logic is considered switched when the output voltage rises above the halfway threshold.

they deal with. Signals that deal with modulated sinusoids can be examined with a sinusoidal phase detector, which one gets by mixing the input and clock signals. At the other end of the spectrum are complex logic phase detectors such as the Alexander (or Bang-Bang) phase detector and the Hogge phase detector.

From these primitive objects, I am able to construct classes for phase-detectors. Among the phase detectors available for simulation are:

- Mixer (sinusoidal): memoryless, ideal element used in pure analog PLLs [4]. The starting point of most analyzes is shown in Figure 2. This responds well to zero centered input signal.
- Exclusive OR (XOR): memoryless, ideal element used in classical and digital PLLs [4]. Unlike the mixer, this is typically used with digital logic levels. This is shown in Figure 3.
- Hogge: linear phase detector using flip-flops and gates that can recover phase from NRZ data [5], [6]. This requires that the VCO clock period be the same as the data (bit) period. This is shown in Figure 5.
- Bang-Bang: binary phase detector using flip-flops that can recover phase from NRZ data [7], [8]. This requires that the VCO clock period be the same as the data (bit) period. This is shown in Figure 4.

What is important to note about these different phase detectors is that some require digital logic blocks, some require flip flops, and all require some math. By having a class library of primitives for these components, it is fairly easy to construct and test any of these (or other) phase detectors.

B. *Simulating Analog Behavior of Digital Logic*

An important piece of the realism of the simulation was suggested by Rick Karlquist of Agilent Labs. The underlying circuit implementation of digital logic involves switching based on voltage levels where these levels rise and fall in an analog way. This analog rise and fall can usually be modeled by a simple differential equation.

The propagation delay, T_d is the time between the application of as signal to the circuit input to the time when the circuit input starts to change. This resembles the classic transport delay of linear systems. The switching time, τ_s determines the time it takes for levels to move from one logic level to 50% of the distance between logic levels. This is shown in Figure 6.

To model these switching times, we use the solution of a first order differential equation:

$$V(t_{k+1}) = (V_{NL} - V(t_{s,i}))\left(1 - e^{-\frac{t - t_{s,i}}{\tau_s}}\right), \qquad (1)$$

where

- V_{NL} is the new desired voltage level. In a binary logic system, there are two voltage levels for logic, and thus, V_{NL} can be V_L or V_H (low and high) depending upon what the original level was.
- $V(t_{s,i})$ is the output voltage of the circuit at the time of a logic switch.
- $t_{s,i}$ is the time of the i^{th} level switch
- τ_s is the switching time constant of the circuit.

Thus, each circuit block has an input section based on "analog" inputs from other circuit blocks. The decision based on an input is delayed by T_d, but with this delay the decision threshold is applied to those inputs. This determines the ideal logic state of the ideal logic circuit. If this logic state represents a switch from the previous logic state, then the output of the circuit is determined by (1). Note that as (1) is the solution of the differential equation, it does not depend on the system sample rate for accuracy. This allows the simulation time to be slower away from a switch point than it would need to be otherwise.

C. *Multi-Rate Simulation of Digital Components*

Early simulations showed that all the digital phase detectors suffered from the ability to resolve time properly. This was a limitation due to the time quantization of the simulation and the fact that the clock and data signals into the phase detectors were binary (0/1). To remedy this, a multi-rate simulation feature was used which works as follows:

- At a time, t, the simulator predicts forward one simulation time step, $t + T_{sim}$, to see if the output of either the VCO or the reference input will change.
- If it detects a change in any of these, it ups the sample rate of the simulation by a multi-rate factor, MR_Factor.
- It runs the VCO and the reference generator using this faster sample rate and then feeds the outputs of these into either
  - an averaging filter or

– a single pole analog low pass filter.

The averaged/analog filtered output is then sent to the loop filter. The increased time resolution is effectively converted into a voltage resolution which the loop filter, running at a slower rate, can make use of.

D. Data Inputs

The data inputs will vary greatly depending upon the type of loop to be modeled. A loop that merely has to lock to the phase of a largely sinusoidal signal will need fairly simple inputs. On the other hand, simulating communication systems would require something that looks like digital data. Often, this is generated using Pseudo Binary Random Sequences [9], [10]. In this case, the digital data is generated with a sequence generator, where the bit frequency is substantially below the clock frequency of the simulator. Another component of this class can then be used to disturb the phase of the data signal in a prescribed way. For example, noise can be added to the time of a transition. Alternately, a sinusoidal variation can be added to the transition time. Because the true data is known, and the variations added are known, this allows for a significant amount of post processing once the simulation has run.

E. Analog Filters

There are some extra features to this simulator that make it more accurate than it would be otherwise. I have added a class of first order analog filters. These are analog in the sense that their output is taken from the closed form solution to a first order differential equation which describes the filter (including time constants), as discussed in Section IV-B. This allows me to do some averaging in some convenient places. Furthermore, at any point in time, the output of the filter is accurate in the sense that one simply reads the solution for a given time, rather than over a quantized time interval.

F. Oscillators

There is also a class of oscillators which can be either sinusoidal or square wave type. The oscillators can be used as data inputs, modulators, or as VCOs. To use as VCOs, one simply needs to set the phase of oscillator using the output of the PLL loop filter. The oscillator classes define routines (called methods by the object oriented programming folks) that allow the user to set the input phase in radians or degrees. Radians are useful, because these are the physical units that the control voltage to a VCO would use.

Currently, the sine wave oscillators are zero centered and the square wave oscillators swing between 0 and 1. This matches their typical use in analog and digital systems, respectively. However, it is not difficult to define an offset for either one that will change what it swings around.

V. Slow Components

A. Filters

There is also a filter class which can include FIR and IIR filters. The filters themselves are read from a specially formated ASCII file with a **.flt** or **.svo** extension. The key feature here is that the filter can be written from Matlab, allowing a filter analysis and design to be done in Matlab and then have this transferred easily into the fast simulator. So, an ideal filter response can be generated in Matlab, say by matching the measured frequency response of a prototype, and then a discrete equivalent can be formed and saved to the *.flt* or *.svo* file.

The salient feature of these file formats is that they allow a linear system that has been written from Matlab to be read from a C/C++ program without the latter having any prior knowledge of the system organization. The *.svo* format was created for general MIMO systems, while the *.flt* format is a simplified version that focuses on SISO transfer function forms of FIR and IIR discrete filters.

Because of the inheritance of object oriented methods (such as used by C++), a general filter class can be defined, with subclasses for FIRs and IIRs. The class constructors of these different types will have similar methods, but adapted to each subclass. One of the advantages of object oriented methods is that the calling routine doesn't have to know whether the filter being created based on the *.flt* file is an IIR or FIR. As the file is being read, the appropriate constructor function (method) can be called once the line containing the filter type is found.

Running the filters in the simulator also require some cleverness, due to the possible need to run part of the simulator faster than the filters. Because the filters are stored as discrete time filters, one cannot simply change their sample rate. Thus, if the simulator sample period changed, it would compromise the filter. Instead, the filters are run as follows:

- When the filter is initialized, the current simulator time is stored into the filter class.
- Using the filter sample period, T_{filter}, the next update time for the filter is calculated.
- At each time step, the PLL simulator checks the current time, t, against the next update time, $t_{NextUpdate}$, for the filter.
  – If $t < t_{NextUpdate}$, the old filter output values are used.
  – If $t \geq t_{NextUpdate}$, the filter is run.

  The filter also has two modes, which either lock to the original update time, or slip the next update time so that it is exactly T_{filter} past the current time. When the filter sample time period is an integer multiple of the simulator sample time period, their operation is identical. However, since we cannot guarantee this, these modes let the user choose how to handle the mismatched times. The latter guarantees that the filter always runs at a rate less than or equal to its designed update rate. The former (which is the mode I prefer), guarantees that on average the filter will run at its designed sample rate.

B. VCO Input

Although the output of a VCO or NCO is at high frequency, the input control voltage is at low frequency in the baseband. Furthermore, there is a modulation bandwidth – essentially a low pass filter on the control voltage. This can be modeled by using a low pass IIR filter on the end of the loop filter. Laboratory spectrum measurements of prototype VCOs can be matched in the frequency domain, and then modeled in the simulator with a discrete equivalent as described in Section V-A.

VI. Post Processing

Post processing will be discussed, along with some example simulation results, in Part II of this paper [1]. Suffice it to say, that time domain plots, frequency domain plots, and histograms are all facilitated by having dumped the fast simulation data into Matlab *.mat* files.

VII. Conclusions

Part I of this paper has tried to show how to generate a flexible and efficient PLL simulation. Part II of this paper [1], will discuss post processing the simulation data and present a design example.

References

[1] D. Y. Abramovitch, "Efficient and flexible simulation of phase locked loops, part ii: Post processing and a design example," in *Submitted to the 2008 American Control Conference*, (Seattle, WA), AACC, IEEE, June 11–13 2008.

[2] D. Y. Abramovitch, "Phase-locked loops: A control centric tutorial," in *Proceedings of the 2002 American Control Conference*, (Anchorage, AK), AACC, IEEE, May 2002.

[3] R. E. Best, *Phase-Locked Loops: Design, Simulation, and Applications*. New York: McGraw-Hill, fourth ed., 1999.

[4] D. H. Wolaver, *Phase-Locked Loop Circuit Design*. Advanced Reference Series & Biophysics and Bioengineering Series, Englewood Cliffs, New Jersey 07632: Prentice Hall, 1991.

[5] C. R. Hogge, Jr., "A self correcting clock recovery circuit," *IEEE Journal of Lightwave Technology*, vol. LT-3, pp. 1312–1314, December 1985.

[6] D. Shin, M. Park, and M. Lee, "Self-correcting clock recovery circuit with improved jitter performance," *Electronics Letters*, vol. 23, pp. 110–111, January 1987.

[7] J. Alexander, "Clock recovery from random binary signals," *Electronics Letters*, vol. 11, pp. 541–542, October 1975.

[8] R. C. Walker, "Designing bang-bang plls for clock and data recovery in serial data transmission systems," in *Phase-Locking in High-Performance Sytems - From Devices to Architectures* (B. Razavi, ed.), pp. 34–45, New York, NY: IEEE Press, 2003.

[9] S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, June 1981.

[10] W. J. Gralski, *SONET*. McGraw-Hill Series on Computer Communications, New York, NY: McGraw-Hill, 1999.