A Spread Spectrum Sonar with Noise-Locked Loop-Based Entrainment

by

Joseph L. Richards

Submitted to the Department of Physics and the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Master of Engineering in Electrical Engineering and Computer Science and Bachelor of Science in Physics at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY June 2000

© Joseph L. Richards, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ............

Department of Electrical Engineering and Computer Science

May 19, 2000

Certified by ..................................................

Neil A. Gershenfeld

Associate Professor of Media Arts and Sciences

Thesis Supervisor

Accepted by ............

Arthur C. Smith

Chairman, Department Committee on Graduate Students

Accepted by ..................................................

David E. Pritchard

Senior Thesis Coordinator, Department of Physics
A Spread Spectrum Sonar with Noise-Locked Loop-Based

Entrainment

by

Joseph L. Richards

Submitted to the Department of Physics and the
Department of Electrical Engineering and Computer Science
on May 19, 2000, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Physics

Abstract

Although most computation is done digitally to allow for arbitrarily long, error-free computations in the presence of noise, there are some problems that are discouragingly expensive (in terms of cost, power, complexity, and other factors) to solve digitally but can be efficiently solved using analog nonlinear systems that show noise immunity. These systems can combine the beneficial stability of digital computation with the flexibility and power of the analog domain. We discuss an example of such a system, the “analog feedback shift register” (AFSR) and the related “noise-locked loop” (NLL). The AFSR is a nonlinear analog system that can entrain to a pseudo-noise (PN) sequence generated by a certain linear feedback shift register (LFSR). When an AFSR is forced by copy of its PN sequence, it entrains to that input and generates a local copy of the noise sequence. This system shows promise for use in spread spectrum code acquisition and tracking. We have tested a simulation of this device in a prototype spread spectrum sonar suitable for use in motion tracking and other user interface applications. We describe the dynamics of the AFSR, discuss how it is related to the NLL, and compare its performance with digital correlation detection in our test system.

Thesis Supervisor: Neil A. Gershenfeld
Title: Associate Professor of Media Arts and Sciences
Acknowledgments

I would like to thank everyone who has in any way helped in the production of this thesis. Due to space constraints, I shall not attempt to record an exhaustive list. It does include my parents and step-parents, without whose love and support I could not have survived five years at MIT, my brother Josh, and my other step- and half-siblings who help make Bloomington a more interesting place to call home.

Very special thanks to Elina, whose cleverness, wisdom, love, and support have been invaluable in the years I have known her.

And of course, I am deeply grateful to my mentors, colleagues, and friends at the Media Lab who have been especially helpful in the development of this work. In particular, I wish to thank Matthew Reynolds for having the patience to help me learn so much over the last five years, Ben Vigoda for his groovy attitude and guidance down the chaotic road of nonlinearity, Neil Gershenfeld, for providing the resources and leadership without which this thesis would have been impossible, and the rest of the PHM (and pals) gang who contribute to the funky karma down here in the basement.

Finally, I would like to thank the Flying Karamazov Brothers, whose collaboration with the lab provided inspiration to work on sonar and practice juggling.

Thanks everyone. You have my gratitude.
Contents

1 Introduction ................................................. 11
  1.1 AFSRs and Noise-Locked Loops .......................... 14
  1.2 Spread Spectrum Sonar .................................... 14
    1.2.1 Sonar Implementation History .......................... 15
    1.2.2 Tracking System Objectives ............................ 16

2 Nonlinear Dynamics ........................................... 17
  2.1 Linear Systems ............................................ 18
    2.1.1 Forcing Terms ........................................... 19
    2.1.2 Systems of Equations .................................... 20
    2.1.3 Evolution of Linear Systems ............................ 21
  2.2 Nonlinear Systems .......................................... 22
    2.2.1 Multiplicity of Fixed Points ........................... 23
    2.2.2 Limit Cycles and Bifurcation ........................... 24
    2.2.3 Iterated Maps ............................................ 25
    2.2.4 Chaos .................................................... 26
    2.2.5 Stability of Solutions ................................... 27
  2.3 Time Delay Systems ......................................... 28
  2.4 Connections to Computation ................................. 29
    2.4.1 Translating Computation into Dynamics ................. 30
    2.4.2 Lower-Dimensional Universal Computation .............. 31
List of Figures

2-1 Phase space plot of $\dot{x} = x^3 - x$. The arrows indicate the direction of flow. Note that there are two unstable fixed points ($\pm 1$) and one stable fixed point (0). ............................................. 24

3-1 Schematic of an LFSR. ......................................................... 34

3-2 PLL schematics. (a) is a general PLL topology; (b) is an example circuit, borrowed from [2] .............................................................. 38

3-3 Schematic of an autonomous NLL. ........................................... 39

5-1 Example circuit to generate a phase ramp that resets on the all-“1” state of the AFSR. This will reset to approximately $V_0 = V_{gs}$, where $V_{gs}$ is the FET gate-source voltage. The slope of the ramp will be $I/C$. 58

6-1 Correlation peaks for a single transmitter and no other ultrasonic source. Peaks are clearly visible above the baseline and occur with the expected separation of 150 samples, or 15 milliseconds. .............. 60

6-2 Example of correlation peaks in the presence of noise. This sample was recorded with an SNR of approximately -13.2 dB. .................... 61

6-3 Correlation peak height versus SNR. ...................................... 61

6-4 Correlation peak position versus distance. ............................... 62

6-5 Correlation peaks for two transmitters located side-by-side. As expected, there are now two peaks per 15 millisecond LFSR period. ... 64
6-6 AFSR with $\epsilon = 0.25$ and $G = 3.0$ entraining on captured sonar data. The top plot shows both the noisy input data and the AFSR output. The bottom plot shows the error between the two signals. The ring-up transient in the first 20 chips is typical of the AFSR and reminiscent of a damped simple harmonic oscillator.

6-7 Plot of the error signal over a long time scale. The growth in error shows loss of entrainment due to accumulated clock skew.

6-8 AFSR entraining with a noisy sonar signal. Despite the slow variation of the input signal amplitude, the AFSR sits near the $\pm 1$ rails.

6-9 AFSR phase estimation using the phase ramp. Upper plots show the ramps as two 7-bit AFSR entrain to differently lagged versions of the same data. The bottom plot is the difference between the two ramp values, $\Delta \phi$. 
Chapter 1

Introduction

Any system that performs some operation to produce a set of outputs based on a set of inputs can be thought of as a computer. Today, that label is almost always reserved for referring to digital computation devices. This is because "real" computation is almost exclusively performed using digital systems — analog electronics are generally reserved for signal processing and other tasks that are straightforward to perform in that domain.

During the early days of electronic computing, there was a real competition between analog and digital computation. Clearly, that battle was won by digital computation. A critical reason for this is the noise resistance of digital computation. The quantization in a digital computer is performed so as to provide a noise buffer region; in essence, the digital computer performs error correction after each operation. According to Shannon's channel capacity theorem, arbitrarily low error rates can be achieved in a communications channel up to some capacity that depends on the noise level in the channel. By treating the digital computer as a communications channel, thus, for a given finite gate error rate, it is still possible to compute for arbitrarily long times without accumulating error in the result of the computation [14, 15].

However, this noise immunity comes at engineering cost. Digital systems allocate a great deal of their available state space to the buffer zones to protect their results. In addition, the energy required per bit of computation is high. A result of their computational generality is increased complexity and often inefficiency at solving
specific problems.

Analog computation can address some of these issues. For some computations, it is possible to design simple analog circuits that produce the desired output. Often, because they are tailored to a specific problem, they will be more efficient at solving that problem than the digital alternative. The crucial issue is protecting the computation from noise. By carefully designing the dynamics of the system, it can be possible to produce an analog system that is stable in the presence of errors — that is, it can tolerate mistakes without diverging from the correct computational trajectory. This observation is the inspiration for the work in this thesis. Computational universality requires nonlinearity, and as a result there are deep connections between nonlinear dynamics and computation.

Nonlinear dynamics is a broad area of study with useful applications in almost every academic discipline. Mathematics, physics, biology, economics, to name a few, have active nonlinear dynamics research communities. Although modern digital computers were instrumental in exploring the behavior of these systems, until recently there were few direct applications of nonlinear dynamics in computer science.

In 1990, Cristopher Moore showed that certain nonlinear dynamical systems were universal computing structures — a billiard ball in a three-dimensional potential is Turing complete [8]. This discovery opened an exciting connection between nonlinear dynamics and computation. It makes apparent the possibility of using dynamical systems to perform computational tasks. By finding appropriate nonlinear physical systems, tasks that we currently perform on power-hungry, general purpose digital computers can instead be performed by the dynamics of analog systems with fewer components. This offers the potential promise of significant power savings because the computational structure is customized to the task at hand.

This link between physics and computation runs both ways — not only are physical systems useful for performing computational tasks, but computer science can provide physical insights. These computationally interesting systems are chaotic in a strict sense. Chaotic systems are characterized by an extreme sensitivity to initial conditions; Moore's systems take this to the extreme where even when initial
conditions are known precisely, long-term predictions are impossible. By understanding the computational behavior of these systems, however, it is possible to describe their long-term behavior in ways that conventional physical methods cannot. Thus, information theory can provide useful predictions about their behavior.

One of the first examples of a nonlinear dynamical system with important computational characteristics is the analog feedback shift register (AFSR), introduced by Gershenfeld and Grinstein in 1995 [3]. This structure uses dissipative dynamics to entrain a local pseudo-random noise (PN) source to a received copy of a remote noise generator. This is one of the key operations required for a spread spectrum data receiver.

“Spread spectrum” refers to a variety of communications techniques that transmit a narrow-band data signal over a much wider communications band [1, 10]. This type of communications offers a number of advantages over other modulation schemes, including reduced susceptibility to jamming (either by hostile third parties or inadvertent jamming from adjacent communications channels), efficient bandwidth use for channel sharing, a coding gain due to multi-bit encoding of data symbols, and reduced analog filter complexity (because precise filters are not required to partition the bandwidth). Originally developed for military communications in hostile environments, spread spectrum is now a widely used technology in the commercial sector. Popular applications include digital cellular telephones, global positioning system (GPS) receivers, and other wireless communication networks.

In this thesis, we develop two distinct but related lines of research. First, we consider the dynamics that govern the AFSR and related dynamical structures. By studying the governing nonlinear dynamics, we develop an understanding of its behavior and computational properties. Second, we describe the design and implementation of a spread spectrum sonar tracking system that uses the AFSR to perform part of the demodulation and tracking computations.
1.1 AFSRs and Noise-Locked Loops

The analog feedback shift register (AFSR) as described by Gershenfeld and Grinstein is a discrete-time, continuous space dynamical system — i.e., a clocked analog system [3]. The AFSR is an analog nonlinear system that is related to the linear feedback shift register (LFSR) often used to generate pseudo-random bit sequences. The dynamics of the AFSR are such that when it is forced by a particular LFSR noise sequence, its internal state entrains to that of the LFSR that generated the noise. The operation of synchronizing to a received pseudo-random bit sequence is an important task in spread spectrum receivers. The smooth entrainment properties of the AFSR suggest that it may be useful in such an application.

A major issue for the AFSR is that its clock must be synchronized with the bit clock of the sequence it is receiving in order for its dynamics to work. In order to remove this requirement, a continuous-time version of the AFSR’s dynamics is desired. The noise-locked loop (NLL) is a dynamical system that may meet this requirement [13]. The NLL has a structure and behavior similar to the phase-locked loops (PLLs) that are present in many communications systems. A PLL is an analog structure that synchronizes a local oscillator with an incoming oscillator by attempting to maintain a constant phase error between the two signals. An NLL performs an analogous task with pseudo-random bit sequences instead of sine waves.

1.2 Spread Spectrum Sonar

As a test platform for the dynamical system under study, we have developed and constructed a spread spectrum sonar system. We chose this particular platform for a number of reasons. First, because most spread spectrum communication systems operate at much higher radio frequencies the engineering required to design these systems is much more complicated. By operating at audio frequencies, we are able to concentrate our efforts on the NLL design itself. Second, sonar tracking systems are useful for tracking objects in a number of situations; because of the advantages
that spread spectrum offers for spectrum sharing and interference rejection a spread spectrum sonar would be a useful device for its own sake. It is hoped that the simple sonar we have developed will be extended for use in a tracking system suitable for use as a computer user interface.

1.2.1 Sonar Implementation History

In a recent collaboration with the Flying Karamazov Brothers (FKB), a juggling troupe, we developed a sonar system capable of tracking four dancing jugglers on a floor roughly 30 feet across. The sonar channel was shared using a time domain multiple access (TDMA) scheme with a master clock distributed via radio. In preassigned time slots, a transmitter worn by each juggler would transmit a brief “ping” to be received by microphones located around the perimeter of the floor. By measuring the time of flight to each microphone, the location of the juggler could be determined. Because we were able to measure the time of flight to within less than a microsecond, we could theoretically measure positions to within $10^{-3}$ feet — much more accurately than a human can control his position. In practice, our resolution was somewhat coarser than this, but still better than that of the jugglers.

While the spatial resolution was much better than required, the temporal resolution proved to be an issue. The relatively low velocity of sound in air (343 m/s, or about 1 foot per millisecond) that made our measurement method possible also forced a non-negligible delay between emitting a ping and making the time of flight measurements. Since the pings from the various transmitters were distinguishable only by the time slot during which they were “heard” by the microphones, a transmitter could not emit a ping until the previous ping had cleared all the microphones. For our 30 foot by 30 foot floor, the master clock was limited to 30 Hz, resulting in approximately 7 Hz updates for each juggler. This coarse temporal sampling was clearly noticeable and resulted in a somewhat awkward system since the position updates were rather slow compared to the rate at which humans move.
1.2.2 Tracking System Objectives

Our goals in pursuing a spread spectrum sonar tracking system were inspired by our disappointment with the update rates attainable using the TDMA scheme in the FKB sonar system. There were two types of time lag in the FKB system that were objectionable: inter-measurement delays and overall latency. The former results from the TDMA scheme's requirement that only one sonar transmitter speak in each time slot and could potentially be remedied by using a different channel-sharing scheme. The latter results from the finite velocity of sound and is unavoidable — the position estimate cannot be updated before the signal travels from the transmitter to the microphone.

To circumvent the update rate limitations, a means for permitting multiple transmitters to speak during the same time slot is required. Two multiple access schemes are available that permit this: frequency division multiple access (FDMA) and code division multiple access (CDMA). Although either system could be used to allow essentially continuous position updates, spread spectrum-based CDMA offers a number of advantages over FDMA, such as process gain [1].

The FKB tracking system offered nearly two orders of magnitude better spatial resolution than our application required. Unfortunately, the TDMA scheme did not offer any means for trading off this accuracy for benefits in other design criteria. Our CDMA scheme, however, permits exploitation of a tradeoff between spatial accuracy and coding gain.
Chapter 2

Nonlinear Dynamics

The first major triumph in the study of nonlinear dynamical systems was Newton's solution to the two-body gravitational problem. His efforts resulted in a satisfying analytical result — from a set of initial conditions, a precise trajectory for the two masses could be calculated. Clearly, the next step for a clever physicist or mathematician was to find a similar solution to the three-body problem, or perhaps the general $N$-body problem. The problem, however, refused to yield an analytic solution. This was not due to insufficient effort on the part of the researchers. The problem turns out not to have any analytical solutions at all. Other than by numerical solution of the equations of motion, one cannot determine precise trajectories for the system over time. Poincaré showed that it is possible, however, to ask general, qualitative questions about the behavior of the solutions and derive useful results [12]. Is there a stable solution, or will the solar system eventually collapse or drift off to infinity? Are there multiple stable solutions? How stable are the trajectories? If we made errors in our measurements of the initial conditions, is it possible that the solar system will collapse instead of remaining safe?

Finding answers to these questions is certainly important, but making detailed studies of the properties of nonlinear systems is difficult without being able to accurately visualize the solutions. The development of powerful electronic computers in the middle of the twentieth century sparked a sudden interest in the study of nonlinear dynamical systems. Computations that were previously intractable could
suddenly be performed easily, so detailed numerical simulation was finally possible. For this reason, the field of nonlinear dynamics exploded during the 1960s.

This relationship between nonlinear dynamics and computational devices appears to run very deep — already, it has been well established that powerful computers are useful for studying nonlinear dynamical systems. More recently, it has been discovered that nonlinear dynamical systems can in fact themselves be powerful, universal computational devices [8]. This observation is the basis for our interest in exploring nonlinear dynamics in this thesis.

2.1 Linear Systems

Physical systems can be characterized by a multidimensional state vector and a nonlinear relationship between that state and its various derivatives. Over a wide range of observation, all physical systems are nonlinear systems. Within a restricted domain, however, a linear approximation is often sufficient to characterize the behavior of the system. Because these linear systems are much more amenable to analysis, such approximations are made when possible.

Even when the system of interest will be studied over a domain that cannot be described by a linearization, finding local linearizations near equilibrium points of the system can be a very helpful step toward developing an understanding of the complete system behavior. Thus, understanding the characteristics of linear dynamical systems is crucial to understanding dynamical evolution when nonlinearities are introduced.

Not all features of nonlinear dynamical evolution can be extrapolated directly from linear system responses. For example, oscillatory limit cycles are an important structure in the solutions to many nonlinear systems, yet are fundamentally different from structures found in the trajectories of linear systems. In addition, chaos is a property found only in nonlinear systems and is without analog in the linear regime.

Most systems studied in traditional physics and differential equations courses are linear time-invariant (LTI) systems. That is, the dynamics can be characterized by
some m-th order homogeneous differential equation

\[ A_m \cdot \frac{d^m \vec{x}}{dt^m} + A_{m-1} \cdot \frac{d^{m-1} \vec{x}}{dt^{m-1}} + \cdots + A_1 \cdot \frac{d \vec{x}}{dt} + A_0 \cdot \vec{x} = 0, \quad (2.1) \]

where \( \vec{x} \) is an \( N \)-dimensional state vector and the \( A_i \) are \( N \times N \) matrices. To completely specify such a problem, a set of initial conditions must be provided. For an \( m \)-th order \( N \) dimensional system, a total of \( m \times N \) initial conditions must be provided to determine a unique solution. In most physical situations, the equations of concern are of order two or less. Various orders of derivative can be understood to represent different time scales, and it is rare for phenomena at widely different time scales to exhibit strong effects in a single system.

Because the time, \( t \), does not appear as an explicit parameter in this equation, the system response will be independent of the time origin chosen — the system will evolve in the same way at any time, given the same initial conditions. Most systems are not truly time-independent, although over the time scales of interest, time-independence is often a reasonable assumption. A time-independent system is also known as an autonomic system, in reference to the fact that its evolution occurs independent from external influences.

The crucial property of a linear system is that its solutions can be superposed to produce another solution. That is, if \( \vec{x}_1 \) and \( \vec{x}_2 \) are solutions, then \( \vec{x}_3 = a \vec{x}_1 + b \vec{x}_2 \), where \( a \) and \( b \) are constants, is also a solution. It is clear that such a property cannot be true of physical systems in general. Such a system must allow, for example, arbitrarily large displacements, whereas nearly all physical systems have some constraint on this maximum displacement. Indeed, for large displacements, many nominally linear systems begin to demonstrate nonlinear behavior.

### 2.1.1 Forcing Terms

If we replace the zero in the right-hand side of the homogeneous LTI equation (2.1) with some function of time, \( \vec{F}(t) \), the nature of the problem is changed somewhat. The equation now has an explicit time dependence, so it is now non-autonomous,
reflecting the fact that the system is now connected with the outside world. Because of this, the function \( \bar{F}(t) \) is called a forcing term. The resulting equation is

\[
D \cdot \ddot{x} = \bar{F}(t),
\]

where \( D \) is a derivative operator that includes the various orders of derivative involved. A solution to this equation known as a particular solution and denoted \( \ddot{x}_p(t) \). Solutions to (2.1) are known as homogeneous solutions, \( \ddot{x}_H(t) \). A general solution to the forced equation (2.2) is \( \ddot{x}_p(t) + \ddot{x}_H(t) \), since \( D \cdot \ddot{x}_H(t) = 0 \) by (2.1). A forcing term corresponds to an external tug on the system — for example, a push on an otherwise free-swinging pendulum.

### 2.1.2 Systems of Equations

As we remarked earlier, an \( m \)-th order differential equation in \( N \) dimensions requires \( m \times N \) initial conditions. This means that, for example, a second order, one-dimensional system has two degrees of freedom. Likewise, a first order, two-dimensional system also has two degrees of freedom. This suggests a connection between order and dimensionality. In fact, it is possible to trade off between these characteristics. For example, the second order one-dimensional differential equation

\[
\ddot{x} + a\dot{x} + bx = 0
\]

is equivalent to the system of first-order equations

\[
\begin{align*}
\dot{x}_1 &= -ax_1 - bx_2 \\
\dot{x}_2 &= x_1
\end{align*}
\]

(2.3)
which we can rewrite in explicit vector form

\[
\dot{x} = \begin{bmatrix} -a & -b \\ 1 & 0 \end{bmatrix} \cdot \bar{x} .
\] (2.4)

By making similar substitutions, it is possible to convert any higher-order differential equation into a system of first-order equations.

Using a similar technique, a non-autonomous system can be treated as an autonomous system. In a one-dimensional first order version of equation (2.2), let \(x_1 = x\) and \(x_2 = t\). Then we can write the equations

\[
\begin{align*}
\dot{x}_1 &= -ax_1 + F(x_2) \\
\dot{x}_2 &= 1
\end{align*}
\] (2.5)

If the forcing function is linear in time, this will still be a linear system, but in general it will become an autonomous nonlinear system of equations.

### 2.1.3 Evolution of Linear Systems

Given a linear, autonomous, first order system of \(N\) equations, which by our previous discussion is a completely general linear system, we can always find a solution. The equation to be solved is

\[
\dot{x} = A \cdot \bar{x} .
\] (2.6)

If \(A\) has \(N\) distinct eigenvalues, \(\lambda_i\), with corresponding eigenvectors, \(e_i\), we can write the general solution

\[
\bar{x}(t) = \sum_i k_i e_i e^{\lambda_i t} .
\] (2.7)

If \(A\) has any repeated eigenvalues, a solution can still be constructed, although its form is slightly different. If \(A\) is a singular matrix, then at least one of its eigenvalues
will be zero, which means that a vector constant in time is a solution to the system. By diagonalizing $A$, it is then possible to reduce the system to an equivalent system of lower order.

The linear system described by (2.6) has exactly one fixed point or equilibrium point, at $\vec{x} = 0$ if $A$ is nonsingular. At this point, $\dot{\vec{x}} = 0$, so the trajectory is a single point. If $A$ is singular, then there will be a connected set of fixed points.

If we consider the case $N = 2$, a linear second order system, we can graphically depict the behavior of the solutions. Understanding these classes of behavior is a useful method for finding approximate solutions to nonlinear systems through phase plane analysis. The nature of the trajectories is determined by the eigenvalues. The various possible cases are tabulated in Section 2.5 of [11].

### 2.2 Nonlinear Systems

By relaxing the linearity requirement, a much wider variety of systems can be studied. Unfortunately, this also increases the difficulty of analyzing their behavior and properties. In general, analytic solutions to these systems do not exist. Other techniques can be used to study the properties of such systems, but one cannot usually hope to write down their solutions succinctly. The general form for a nonlinear differential equation is

$$\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \dot{\vec{x}}(t))$$ \hspace{1cm} (2.8)

Without an analytic form for $\vec{x}(t)$, studying its trajectories requires some sort of approximation beginning from specific initial conditions. The most general approach is to numerically integrate the equations of motion and plot the resulting values. The tremendous speed of modern computers has made this a very valuable technique for understanding nonlinear systems. This “brute force” method is not without pitfalls, however — great care must be taken to ensure that small numerical errors do not accumulate and introduce large errors in the trajectories. When a trajectory passes near
to a boundary between different classes of behavior, for example, slight numerically-induced errors can cause a solution to "jump" from the correct path to one that is qualitatively different.

Beyond being without analytical solution, nonlinear systems can exhibit many behaviors not found in linear systems. While a linear system is limited to a single fixed point (or a connected set of fixed points), most nonlinear systems possess multiple fixed points. In addition, a nonlinear system can exhibit limit cycles, bifurcation, and chaos. There are deep connections between nonlinear systems and computation, which is our primary interest in studying these mathematical constructions and the physical systems they represent.

### 2.2.1 Multiplicity of Fixed Points

Whereas a linear system is limited to either a single fixed point or a single connected set of fixed points, nonlinear systems often have multiple separated fixed points. For example, the system

\[ \dot{x} = x^3 - x \]  

has fixed points at \( x = \{0, \pm 1\} \). As shown in Figure 2-1, 0 is a stable fixed point and \( \pm 1 \) are unstable. In one dimension, this is the extent of the complexity that a solution may present. Since all trajectories are restricted to a single line in phase space, paths have no way to pass each other — every trajectory either ends at a stable fixed point or blows up to infinity. This is because trajectories in phase space cannot cross except at singularities (fixed points). A point in phase space completely specifies the state of the system, so if two trajectories intersect at one point, they will follow the same trajectory forever.

In two or more dimensions, the picture becomes more interesting. Because trajectories can now swirl around each other without crossing, complicated paths through phase space become part of the solution set. In the vicinity of each fixed point, the behavior of the trajectories can be determined by linearizing the system about the
fixed point (assuming there are no “hard” nonlinearities, such as a discontinuity). By connecting trajectories between the various fixed points, it is possible to construct rough estimates of the phase portrait of the system. This is demonstrated in most differential texts; see [11], for example. This method is restricted to use in second order systems due to the difficulty of visualizing the trajectories in higher dimensional spaces.

2.2.2 Limit Cycles and Bifurcation

Limit cycles and bifurcations are examples of phenomena that do not occur at all in linear systems. A limit cycle is an isolated, closed curve trajectory in phase space. This is distinct from a center in that a limit cycle is isolated. That is, a center is a set of closed trajectories around an equilibrium point — trajectories neighboring one closed path are also closed paths. A limit cycle, however, is neighbored by qualitatively different trajectories that move away from the limit cycle as \( t \to \infty \) or \( t \to -\infty \). The Van der Pol equation is an classic example of a nonlinear system with a limit cycle [11, 12].
Depending on the behavior of its neighboring trajectories, a limit cycle can be classified as either stable, unstable, or semi-stable. If all trajectories that enter some region close to the limit cycle converge to that cycle, then it is stable. If all nearby trajectories diverge from the cycle, then it is unstable. If the nearby trajectories on one side converge to the cycle while those on the other side diverge from it, then it is semi-stable.

The oscillation of a nonlinear system due to a stable limit cycle is qualitatively different from that of a linear system orbiting around a center. Because the limit cycle is an isolated trajectory, the oscillations will occur at only one frequency and one amplitude. Due to its stability, any initial conditions (within the basin of attraction to the limit cycle) will cause the system to oscillate with the same amplitude. An oscillating linear system will oscillate at an amplitude determined by its initial conditions. It is for this reason that stable oscillators require a nonlinear device as their source.

A bifurcation is a qualitative change in the fixed points of a nonlinear system brought about by a change in parameters. For example, consider the system

\[ \dot{x} = ax - x^3 \]  

When \( a \leq 0 \), we find a single stable fixed point at \( x = 0 \). For \( a > 0 \), there are two additional fixed points at \( x = \pm \sqrt{a} \). The fixed points at \( \pm \sqrt{a} \) are stable, while that at \( 0 \) has become unstable. This system is said to exhibit a supercritical pitchfork bifurcation at \( a = 0 \). There are a number of other types of bifurcations for first order systems, and a wide variety of types in higher order systems. Bifurcations can be understood to cause phenomena such as hysteresis and phase changes.

### 2.2.3 Iterated Maps

So far, the methods we have discussed for modeling nonlinear dynamics have involved differential equations. These mathematical constructions operate in continuous time to govern the evolution of various quantities of interest. Another way to describe evo-
olution is through an *iterated map*. An iterated map is a rule, \( \bar{F}(\bar{x}) \) that is iteratively applied to the state vector of a system to determine its next state. That is,

\[
\bar{x}_{i+1} = \bar{F}(\bar{x}_i) .
\]

(2.11)

Obviously, this method for describing evolution operates in discrete time. Many of the properties we have described for nonlinear differential systems are also present in iterated systems. Iteration is a popular method for studying dynamics because computers are very efficient at repeatedly evaluating functions.

A mathematical structure called a *Poincaré section* creates a bridge between the study of continuous time dynamics and the study of iterated maps. A Poincaré section can be used to study a quasi-periodic trajectory in phase space. One slices a surface perpendicular to the orbits and records successive points of intersection with this surface [6]. This, in effect, defines an iterated map that is closely related to the continuous time system. In particular, a fixed point in the Poincaré section corresponds to a periodic trajectory in continuous time. In the case of a system with periodic forcing, a Poincaré section is frequently made by sampling the state of the system once per period of the forcing function, as in the *one period later mapping* described in [5].

### 2.2.4 Chaos

"Chaos" is a buzzword that has received a great deal of attention in recent years. A system exhibits chaos when it follows an unpredictable orbit through phase space. A chaotic orbit traces an infinite, non-repeating path through a finite region of phase space. In order to fit the infinitely long trajectory into a finite region, distinct regions of the orbit pass very near to each other. As a result, minute errors in initial conditions quickly amplify into radically different trajectories. This extreme sensitivity to initial conditions is the best-known property of chaotic systems, and the reason that weather predictions are unreliable.

A chaotic orbit is very much different in nature from a random orbit. A chaotic
orbit is governed by deterministic laws — in principle, precise knowledge of the initial conditions would permit accurate predictions of the long-term behavior of the system, through numerical simulation. However, in practice, we are limited to computations at finite accuracy. The dynamics of a chaotic system amplify our errors exponentially, so our predictions very quickly become wildly incorrect.

The minimum subspace of a chaotic system’s phase space required to contain all of its long term orbits is known as an strange attractor of the system. The attractor for Lorenz system of equations, one of the first chaotic systems to be studied extensively, is discussed in detail in [12]. The trajectories in a strange attractor remain in this finite region of phase space, yet diverge from each other exponentially quickly.

This is as much as we will discuss chaos in this thesis. We will be making every effort to ensure that the systems we study are not chaotic.

2.2.5 Stability of Solutions

Although we cannot generally find explicit solutions to nonlinear differential equations, the situation is not entirely hopeless. Frequently, it is possible to demonstrate qualitative properties of the solutions of a system.

Lyapunov Methods

Lyapunov’s direct method is one of the most powerful methods for demonstrating the stability of a nonlinear system on a particular trajectory. Without a general solution for the motion of the system, it is difficult to make use of ordinary methods for illustrating stability. In physical systems, Lyapunov’s method is closely related to energy minimization.

To use Lyapunov’s method, we attempt to find a Lyapunov function, \( V(x) \). \( V(x) \) must be positive definite in a ball around the phase space equilibrium point, which means that \( V(x) > 0 \) for all \( x \neq 0 \), and \( V(0) = 0 \). In addition, the derivative of \( V(x) \) along trajectories of the system, \( \dot{V}(x) \) must be negative semi-definite, meaning \( \dot{V}(x) \leq 0 \). If such a function exists, then the equilibrium point is stable. For a proof
of this theorem and related commentary, refer to [11].

The major problem for applying the direct Lyapunov method is finding the function, \( V(x) \). Once a function is found, it is generally straightforward to demonstrate its properties and prove the stability of the system. However, not having found a suitable \( V(x) \) does not imply that the system is unstable.

### 2.3 Time Delay Systems

In the systems we have discussed so far, the dynamics are completely controlled by the local, instantaneous state of the system — the information is contained in its variables and their derivatives. While this is fundamentally true of the evolution of a closed system it is frequently convenient to introduce time delays into the dynamics of a system. For example, in a feedback control network with a non-negligible processing delay, the feedback applied at time \( t \) is controlled not by the state of the system measured at \( t, \Phi(t) \), but by \( \Phi(t - \tau) \), where \( \tau \) is the delay in the feedback loop. Such a dependence can arise whenever there is a feedback path whose propagation delay is comparable to the dominant time scales of the system. This frequently occurs in industrial control networks, high-speed electronic systems, and chemical reactions, among others [4, 11].

In a time delay system, the differential equation of concern becomes

\[
\dot{x} = f(\Phi(t), \Phi(t - \tau_1), \Phi(t - \tau_2), \ldots, \Phi(t - \tau_n))
\]  

(2.12)

where \( \Phi(t) = \tilde{x}(t) \otimes \tilde{x}(t) \) is the complete state vector. While this modification to (2.8) seems innocent enough, it has deep implications for analysis of the system. A differential equation like (2.12) is a nonlinear delay differential equation (DDE).

How many degrees of freedom does the DDE possess? In the ordinary nonlinear system, (2.8), the number of degrees of freedom is \( N \), the number of elements of \( \tilde{x} \) — a complete initial condition can be specified by determining these \( N \) variables. In addition, this means that distinct trajectories in its \( N \)-dimensional phase space
cannot cross. In the DDE, however, trajectories that meet in this same space at time $t$ can diverge if their values at some time in the past were not the same. Thus, two trajectories can cross and a higher dimensionality is present. In fact, evolution is determined by the initial conditions over the interval $[t - \tau_{\text{max}}, t]$, where $\tau_{\text{max}} = \max_{n}\{\tau_{n}\}$. The initial condition must be specified as a function $\tilde{\phi}(t)$ over this interval. Thus, the system actually has infinite degrees of freedom. Noise limitations place limits on our ability to access the information in these degrees of freedom, however. In real systems, limited frequency responses will limit the dimensionality to some finite bandwidth.

2.4 Connections to Computation

It is clear that there are connections between nonlinear dynamics and computation. This is trivially obvious if one considers that, at the transistor level, every decision made in any digital processor is made by the nonlinear response of a circuit. In principle, then, a Pentium processor is nothing more than a complicated nonlinear function that controls the dynamics of the output signals, subject to the forcing provided by its input signals. Treating the system as a discrete time, discrete state machine is only an approximation. As clock speeds increase, this approximation becomes less and less valid and the continuous time dynamics become more and more visible. A “discrete time” system is one whose outputs are changing significantly only in the periodic neighborhoods of the clock transitions, due to switched high gain.

Thus, we see that extremely complex nonlinear dynamical systems are capable of computing universally. Such a system is truly a marvel of engineering — that so complex a device can be designed and can operate successfully and repeatedly speaks well of the power of hierarchical design and abstraction as tools for controlling complexity. However, these tools do not necessarily produce the most efficient computational device for a given problem. Furthermore, because these devices are designed to perform more-or-less arbitrary computations, their dynamics are not necessarily selected to efficiently perform a specific computation. Efficiency concerns are among the primary
reasons that simple computations are performed in analog hardware when possible — it is much more efficient to use a diode and a capacitor to detect an AM signal than it is to replace those devices with a high-speed DAC and DSP to perform the detection in software.

As we discussed in the introduction, more involved computations, however, are usually exclusively in the domain of digital methods. Methods for performing sustained, error-free computation on digital processors have been known for decades. Is the loss of efficiency a painful but necessary price for error-free computation? In at least some situations, the answer is "no." By carefully selecting the dynamics of a system, it is possible to perform some operations more efficiently and no less effectively than in a digital processor. It has been shown that low-dimensional dynamical systems (i.e., much less complex than a PC) can be universal computational structures, so it is always possible to implement a computation in a dynamical system [8]. However, not all computations are significantly more efficient in the analog domain. However, in some cases, ensuring fault tolerance can be done in without incurring the full penalty of a digital computation. The task becomes identifying these systems and suitable dynamics for implementing them, certainly not an easy challenge.

2.4.1 Translating Computation into Dynamics

What exactly does it mean to compute with a dynamical system? How does one read the answer? When one uses a computer to solve a problem, a surprisingly deep process is occurring. The combination of writing software and designing the hardware to run that code is, in effect, translating the algorithm into a set of dynamics. The dynamical evolution of the computer is equivalent to the evolution of the algorithm under some complicated transform. Reading the answer at the end of computation requires that one determine the final physical configuration of the system and invert the transform to understand what it means in terms of the original algorithm.

With this picture, computation by a dynamical system is performed by allowing the system to evolve over time. The trajectory of the system in phase space defines its internal states during the computation. One can translate notions of computer
science into dynamical language. For example, the important "halting problem" has a clear dynamical interpretation. The halting problem is to determine whether a program will reach a particular state (the "halt" state) given a certain input. It can be shown that there is no general solution to this problem. In dynamical language, we ask whether a system's trajectory from some initial condition will ever pass through a particular point in phase space — the "halt" point. In general, this cannot be solved. One can run the dynamics and hope to show that the system halts, but one can never be certain that the system does not halt, for it might pass through that point in the next moment.

2.4.2 Lower-Dimensional Universal Computation

A surprising result was shown by Cristopher Moore when he demonstrated that a particle in a 3-dimensional potential can be a universal computing device [8]. Although this universality has been demonstrated, there is still the terrible task of finding dynamics to solve problems. To date, programming nonlinear systems has been done in a backwards fashion. Rather than choosing a problem and searching for the dynamics to solve it, research has started from an interesting set of dynamics and searched for a problem that those dynamics solve.

Digital computers are structured specifically to permit a programmer to operate with semantically understood constructs, such as loops, branches, variables, and other intellectual devices. Does there exist a method for "programming" dynamical systems in such a familiar way, without losing the benefits that drove us away from digital computation in the first place?
Chapter 3

AFSR and NLL Entrainment

The AFSR is a nonlinear iterated map that was introduced by Gershenfeld and Grinstein in 1995 [3]. It uses dissipative nonlinear dynamics to entrain to an PN sequence generated by a particular LFSR. This smooth entrainment, similar to the ringing up of a forced harmonic oscillator, offers potential computational benefits. An NLL is a continuous time generalization of the AFSR introduced in 1999 by Benjamin Vigoda [13].

3.1 AFSRs

An AFSR is an example of an analog nonlinear dynamical system that is capable of performing a specific computational task. In particular, an AFSR can produce a clean, in-phase copy of a noisy PN sequence, an important computational task for the demodulation of spread spectrum signals. The means by which this computation is achieved — smooth dissipative entrainment — is the reason that this method of computation is exciting. The operations required to perform this task in a conventional digital computing paradigm call for a powerful, power-hungry GaAs DSP with a clock rate much higher than the transition rate in the PN sequence chip rate. In the dynamical approach, nothing runs faster than the chip rate — the complex high-speed digital operations are smoothly performed by the analog nonlinear dynamics.
3.1.1 LFSRs

A linear feedback shift register (LFSR) is a digital circuit that can be used to generate a PN sequence. An LFSR is a shift register whose input is some function of its state bits in the previous time step. Obviously, if one would like to generate a PN sequence with an $N$ chip repeat time, one could initialize an $N$ bit LFSR and feed its output directly back to its input. For PN sequences with a meaningful length, however, this would require a prohibitively long shift register. Less obviously, it is possible to generate a sequence with an $N$ bit repeat time using a shift register of length only about $\log_2 N$ \cite{10}.

The typical LFSR configuration used is called a Fibonacci LFSR. As shown in Figure 3-1, the new value shifted into the register is the sum, mod 2, of several of the bits from the previous register state. The bits involved in the sum are said to be “tapped.” By selecting appropriate taps, it is possible to generate a maximal length sequence — that is, one that visits every state except the all-zero state (which is a dead-end). This corresponds to a PN sequence of length $2^N - 1$. While there is no general method for finding these tap sequences, a large number have been found and tabulated \cite{9}.

3.1.2 Moving to the AFSR

As the name suggests, the analog feedback shift register (AFSR) is an discrete time analog system related to the LFSR, and was introduced in \cite{3}. The AFSR is essentially
an analog generalization of the LFSR, with the output of the LFSR as a stable attractor for the AFSR output. From an LFSR, replace the digital shift register with an analog equivalent — that is, a "register" with analog values. Now, since the mod 2 operation is only defined for integer arguments, it becomes necessary to find a new function to combine the values coming in from the taps. In order to mimic the behavior of the LFSR, this function, $\psi$ must agree with the mod 2 operation for integer arguments. We have some flexibility with the intermediate values, but in order to guarantee stability, we will see that certain criteria must be met.

The AFSR then operates in the same fashion as the LFSR. On each clock transition, the values in the register shift, with the output of the function shifted into the open position. The shift operation combined with $\psi$ provides a function that maps one AFSR state ($\vec{x}_i$) into the next ($\vec{x}_{i+1}$). We see that an $N$ bit AFSR is an iterated map on an $N$ dimensional space.

### 3.1.3 Guaranteeing Stability

In dynamics language, our stability criterion requires that the LFSR orbit (a traversal of the corners of a hypercube in $N$-dimensional space) be a stable limit cycle of the trajectory of the AFSR. To satisfy this requirement, it is necessary that $\psi$ have stable fixed points for integer arguments [3]. This requires

$$\left| \frac{d\psi}{dx} \right|_{x \in \mathbb{Z}} < 1. \quad (3.1)$$

This also requires that there be an unstable fixed point somewhere between successive integer values. Any function that meets this requirement will have an attracting basin around the orbit of the associated LFSR.

While this describes a wide class of functions, not all functions that provide this stable limit cycle are equally suitable for our purposes. As we will see shortly, when we consider entrainment, other criteria will become important to ensure acceptable performance.
3.1.4 Entrainment

The AFSR picture we have drawn thus far is an interesting academic excursion, but if its only product is an analog system that mimics a cheaper, simpler digital system, there would be little engineering interest. The crucial difference between the AFSR and the LFSR is the ability of the AFSR to wander in a continuous region of phase space versus the discretized domain of the LFSR. The attracting sub-orbit that we have carefully guaranteed draws the trajectories over a period of time, but this rate is controlled by the slope of \( \psi \).

While the AFSR is exploring these intermediate stretches of phase space, by applying a forcing function in addition to the autonomous dynamics already described, we can nudge the system toward some direction in phase space. If the forcing function is a copy of the LFSR sequence that the AFSR is trying to follow, then an interesting phenomenon occurs. The AFSR now energetically favors trajectories where the adjustment due to its internal dynamics is in the same direction as that due to the forcing. As a result, the orbit of the AFSR will come into phase with the forcing LFSR sequence, generating an identical copy. Thus, the AFSR can smoothly entrain to an input sequence [3]. This can be performed by updating the AFSR with \( 1 - \epsilon \) times its computed next value and \( \epsilon \) times the incoming forcing signal.

In order for this entrainment to be useful, it must be achievable in the presence of a noisy forcing function. For example, in a spread spectrum communications system, the received data might be an LFSR sequence modulated by lower-frequency data and we would hope to use the AFSR to recover this sequence to perform the despreading. It has been shown that for two tap AFSRs, replacing the sum (mod 2) with the operation

\[
x' = G [(x_{t1} - x_{t2})^2 - (x_{t1} + x_{t2} - 1)^2 + 1],
\]

(3.2)

where \( x' \) is the new register value, \( t1 \) and \( t2 \) are the indices of the taps, and \( G \) is some gain factor, results in an AFSR that stably and rapidly entrains in the presence of noise. The important property of this function is that it is flat in the region of

36
\( x_{t_1} = x_{t_2} = 0.5 \) for any value of \( G \). This balances the tendency to shift out to the limits (where the function should be clipped or otherwise rolled off) with the freedom to explore different phases by moving slowly around the center, thereby increasing the probability of finding the global energy minimum of the entrained state [13].

### 3.2 NLLs

The AFSRs described in the previous section are only usable in coherent receivers, where the transmitter's chip clock is derived from the received signal before demodulation occurs. This is because the application of the iterated map must occur at the same time as the incoming forcing sequence makes its transitions in order to guarantee entrainment. In many cases, it is desirable to construct non-coherent receivers, as they are simpler in design. Getting rid of extra clock transitions will reduce the power requirements, as well. For these reasons, as well as in the interest of intellectual curiosity, we would like to find a continuous time analog of the AFSR — one whose dynamics are dictated by a differential equation that operates without a clock. The noise-locked loop is the name given to such a device, in reference to its similarity to the phase-locked loops (PLL) that are common in communications systems.

#### 3.2.1 Phase-Locked Loops

A phase-locked loop (PLL) is a structure that is found in many communications systems. Shown schematically in Figure 3-2(a), a PLL consists of a phase detector circuit, a voltage-controlled oscillator (VCO), and a feedback path with a transfer function known as the loop filter.

The phase detector produces an error signal that is, ideally, proportional to the phase difference between its two inputs. This error signal is fed to the VCO to set the frequency of the output signal. This output signal is then fed back, through the loop filter, to the input of the phase detector. The PLL "locks" when its output is maintained at a constant phase relative to its input. In order to maintain this state, the output frequency must be equal to the input frequency. The loop filter must be
Figure 3-2: PLL schematics. (a) is a general PLL topology; (b) is an example circuit, borrowed from [2]

carefully designed to achieve a stable PLL.

For the example loop filter shown in Figure 3-2(b), the phase detector signal, $V_{PD}$ satisfies the equation

$$\frac{1}{K_{PD}K_{VCO}} \frac{d^2V_{PD}}{dt^2} + \frac{R_2}{R_1} \frac{dV_{PD}}{dt} + \frac{1}{R_1C}V_{PD} = 0 \quad (3.3)$$

Thus, the dynamics of the PLL are governed by the dynamics of a damped simple harmonic oscillator. By selecting appropriate component values, it is possible to produce critically damped dynamics and achieve the fastest possible settle time [2].

### 3.2.2 Discrete into Continuous Time

Lifting dynamics from discrete time into continuous time is, in effect, inverting the task of generating a Poincaré section. The AFSR map can be thought of as the Poincaré section of a continuous time dynamical system. Any system that passes through the Poincaré surface in the proper sequence is a potential continuous time
AFSR. Other properties of those trajectories are important, however. For example, in a physical implementation, all quantities must be finite. In addition, we would like the trajectory to be a limit cycle at the bottom of a basin of attraction wide enough to entrain to noisy PN sequences, but not so wide as to mistakenly entrain to unrelated noise sequences.

Vigoda proposed a promising candidate system in his Master's thesis [13]. His simple system is controlled by the relationship

$$\frac{dx}{dt} = 4x(t - \tau_1) \cdot x(t - \tau_2) - x(t),$$  \hspace{1cm} (3.4)

where $\tau_1$ and $\tau_2$ are set to be the taps of the AFSR at the desired chip rate. This system has the straightforward circuit implementation shown in Figure 3-3. This implements an autonomous analog nonlinear system that generates a PN sequence output, and has been implemented. To extend this to a forced system, one would introduce some influence from the received PN sequence into the loop. There are several different points in the loop where this forcing could be injected and it is not clear which is optimal. This configuration is similar to the PLL both in form and in behavior, and hence has been labeled the noise-locked loop.

### 3.2.3 Stability Concerns

Demonstrating the stability of the NLL is an important step in turning it into a useful device. Although one can verify by inspection that it is plausible that the
LFSR sequence is a stable orbit of the system, proving this is difficult. The library of techniques for investigating nonlinear delay differential equations, such as (3.4), is unfortunately not very extensive.

One concern is high frequency noise control. We have discovered a notion of "pseudo-noise harmonics" that must be suppressed in a useful NLL system. Whereas in the AFSR, the chip rate is set by the clock that drives the map iteration, in the NLL there is no explicit control of the rate of transitions. If we, for example, construct an NLL with \( \tau_1 = \tau \) and \( \tau_2 = 4\tau \), the obvious interpretation is to consider this a four bit wide continuous AFSR with a chip rate of \( 1/\tau \) and taps in the first and fourth bits. However, it could just as well be a seven bit wide AFSR with chip rate \( 2/\tau \) and taps in the first and seventh slots, or an infinite number of longer sequences with increasing chip rates. We believe that these higher "harmonics" will prove to require more energy to sustain in the NLL and will hence die out, leaving only the fundamental sequence. However, this is not a trivially true statement. Filtering beyond the basic dynamics of the system may be required to ensure that high frequency PN modes are not excited by noise.

3.2.4 Circuit Implementation

To date, a demonstration of the autonomous NLL described by (3.4) has been successfully constructed by Vigoda. The entrainment property has yet to be demonstrated in either practice or simulation, however. This is an important area to push, as the promise of a simple means for entraining PN sequences is extremely enticing. At the low frequencies that the hardware has been constructed, implementing the delay lines is somewhat difficult. At higher frequencies, delay lines can simply be lengths of transmission line that retard the signals along the different paths (although dispersion is a serious concern). At audio frequencies, however, the wavelengths are prohibitively long. In Vigoda’s implementation, these delays were achieved using bucket-brigade units (BBUs), which are long chains of switched capacitors that can be used to delay analog values. These are clocked systems, however, so that circuit is not purely continuous time. The frequency of the clock much faster than any of the other time
constants in the system, which should minimize the effects of the quantization.
Chapter 4

Spread Spectrum and CDMA
Channel Sharing

Spread spectrum techniques convert a narrow-band, high power density signal into a wide-band, low power density signal. One result of this is an enhanced resistance to noise and jamming signals (from hostile forces or from neighboring friendly communications channels). This resistance to jamming can be used to permit multiple channels to efficiently share a finite bandwidth allocation. The high degree of orthogonality between messages on various channels leads to a high degree of crosstalk rejection. This is the basis for code division multiple access (CDMA) channel sharing. CDMA offers a number of engineering benefits and is finding frequent application in modern communications devices, most notably in cell phone networks.

4.1 Spread Spectrum

Spread spectrum (SS) refers to a number of coding methods that convert a relatively narrow-band signal into a wide-band signal with a much lower power density. This wide-band signal is transmitted through the physical medium (usually as an electromagnetic signal) and received. At the receiver, a demodulator "despreads" the signal, converting it back to a narrow-band signal that is (hopefully) the same as the original [1].
Most of these techniques were originally developed for military applications. In these settings, anti-jam capability and low probability of intercept (LPI) are crucial. It turns out, however, that these properties are extremely valuable to communications systems in general, even when there is no hostile entity attempting to detect or prevent communications. For example, the LPI property means that a third party actively attempting to detect transmission is unlikely to succeed. If such an active listener is unable to detect the signal, then the level of electromagnetic interference (EMI) caused by the SS signal must be small. This is a very important consideration in designing modern communications devices under strict EMI regulations. Anti-jam capability is also directly applicable to non-military systems. In multiple access systems, one must be concerned with the jamming between neighboring subchannels. The anti-jam properties make it resistant to this type of interference. [10]

4.1.1 Spread Spectrum Methods

There is a variety of methods for spreading a signal for transmission. The general strategy for SS communication is to subdivide a communications channel into $K$ subbands. $K < K_s$ of these subbands are selected in each time slot ("chip") and the available energy is split among them, leaving the remaining ones unused. It can be shown (see [10]) that the effective jamming energy, $E'_J$, is

$$E'_J = \frac{E_J K_s}{K},$$

where $E_J$ is the actual energy transmitted by the jammer. Thus, with jam energy as a metric, $K_s = 1$ is an optimal strategy. Under other metrics, such as error rates, other choices for $K_s$ may be desirable, however [10].

The two main classes of SS communications systems are frequency hopping (FH) and direct-sequence (DS) systems. In an FH SS system, the spreading is accomplished by modulating a carrier frequency that is changed (or "hopped") in each time slot. In each chip time, $k$ bits from a binary pseudo-random noise (PN) sequence are used to select one of $2^k$ carrier frequencies distributed over the available bandwidth.
The receiver then uses an estimate of the PN sequence to determine which carrier frequency to use for demodulation during each chip.

DS SS systems forego the use of a spreading carrier frequency, instead directly modulating the data signal by a higher-rate PN sequence. This effectively spreads the bandwidth by the ratio of the PN bit rate to the data bit rate. Demodulation is performed at the receiver by generating an estimate of the PN sequence and using this to demodulate each chip.

Both classes of systems can be classified as coherent or non-coherent, depending on whether or not they specifically recover the clock used at the transmitter. In a coherent system, the receiver attempts to synchronize itself to the chip clock. Non-coherent systems perform the demodulation without explicitly generating a local synchronized chip clock.

FH systems are typically popular for military applications because interference between adjacent channels within a band is only observed when they randomly occupy the same frequency channel [1]. In commercial situations, such as cell phone networks, DS systems are far more popular. This owes to better spatial bandwidth sharing (i.e., bandwidth pollution from one cell to its neighbors is lower than in FH systems) and simpler receiver designs.

4.1.2 Process Gain

Two well-known characteristics of spread spectrum communications systems are process gain or coding gain and jamming margin. The former is a measure of the effective signal-to-noise ratio (SNR) increase due to the use of a particular modulation scheme over the physically present SNR in the communication channel. The latter is essentially the processing gain minus system losses and an allowance for minimum useful SNR at the system output.

Coding gain is a direct result of using a wider bandwidth than is strictly required to transmit at the data rate. Coding gain can be understood using Shannon’s channel
capacity theorem. By that theorem, the channel capacity, $C$, in bits per second is

$$C = W \log_2 (1 + \text{SNR})$$

where $W$ is the bandwidth. Thus, as the bandwidth increases with a fixed SNR, the data capacity increases linearly (assuming suitable codes can be found). From (4.2), one can show that a spread spectrum system will have a process gain, $G_p$, given by

$$G_p = \frac{W}{R}$$

where $W$ is the bandwidth and $R$ is the data rate. The related jamming margin, $M_j$, is

$$M_j = G_p - (L + \text{SNR}_{\text{min}})$$

where $L$ is the system loss and $\text{SNR}_{\text{min}}$ is the minimum SNR required at the output [1].

### 4.1.3 High-Resolution Tracking

In a DS SS system, the spreading code is ideally one that is highly self-orthogonal. That is, it should have an autocorrelation spectrum that has a sharp peak when offset by a multiple of its repeat length, and near zero values for all other lags. This offers a benefit in ranging applications because the phase of the signal can be precisely identified by correlating the received version with a reference [1].

In a time-of-flight ranging system, a pulse is transmitted and the distance to the reflecting object is inferred by measuring the time elapsed before the reflection is measured. In a DS SS ranging system, the number of chips of lag between the transmitter’s sequence and the received pulse sequence can be used to see phase delays down to the chip rate [1]. It is clear that higher chip rates will enable higher resolution measurements. This easily accessible phase property is one that we will use in our sonar positioning system.
4.2 CDMA

CDMA refers to the exploitation of the orthogonality of spreading codes for the purpose of sharing a wide channel among multiple transmitters and receivers. CDMA has recently come into fashion as a strategy for sharing the channel in cell phone networks. The interest in CDMA comes both from the particular benefits over the competing strategies (TDMA and FDMA) and the "freebie" benefits that result from using a spread spectrum system to implement CDMA.

4.2.1 CDMA vs. FDMA or TDMA

An important question in channel sharing is which method of channel sharing is appropriate for a system. The channel capacity is fixed by the bandwidth, data rate, and available transmission power. Shannon’s capacity theorem is on a solid theoretical basis, so it is reasonably safe to say that no channel sharing method can produce free channel capacity. From this perspective, CDMA, FDMA, and TDMA are more or less interchangeable.

Of course, application-specific requirements may make one method preferable over another. CDMA and TDMA systems require some method of clock distribution (which may mean deriving the clock from a received signal) to ensure that transmitters speak in turn, whereas FDMA systems generally need only set up communications at the start of each conversation. On the other hand, FDMA systems require sharp filters that can be difficult to design and costly or bulky.

Both FDMA and TDMA are more susceptible to a rogue transmitter jamming the entire channel than is CDMA. If a TDMA transmitter becomes confused (or is maliciously modified) and speaks out of turn, it will effectively jam anyone who was officially permitted to speak in that time slot. Likewise will an FDMA transmitter jam others if it uses the wrong frequency allocation.

This property is the major reason for the interest in CDMA bandwidth sharing in communications systems. Additionally, it is particularly well-suited to ranging applications as we discussed above. It should be noted that by the metric of number
of users per bandwidth available, both TDMA and FDMA systems beat CDMA [1].

4.2.2 Useful “Side Effects”

If one opts to use an SS CDMA system, in addition to the channel sharing, one inherits the other properties of SS communications. Many of these properties are very useful in many applications. For example, the low power density of a spread spectrum system translates into reduced interference with other systems who are trying to share the same bandwidth. The nontrivial data encoding makes “casual” eavesdropping difficult, ensuring private communications.
Chapter 5

Sonar Tracking

The goal of the research described in this thesis is to produce a sonar tracking system suitable for use as a user interface device. Our particular goal is to permit a few objects to be simultaneously tracked while maintaining acceptable update rates and latency.

5.1 Design Goals

We selected design goals that would allow our sonar tracking system to be useful as a human interface device. Although there is a wide range of interface methods, each with its own particular requirements, we concentrated on a scenario whose requirements seemed generally useful.

Our model application was an interface for a computer juggling instructor. The specific goal was to track the positions of three balls traveling at typical juggling velocities over a typical juggling position range. We used the requirements of this application to estimate the capability necessary for our system to prove useful.

5.1.1 Spatial Resolution and Range

The first class of design requirement is the spatial characteristics necessary for a useful interface device. During the course of ordinary three-object juggling, a ball
is contained within a radius of approximately one-half meter from the center of the juggler. For novice jugglers, however, extending this range to a full meter radius is often necessary. This requires a tracking diameter of approximately 3 meters.

The typical motion of a juggled object, even in the hands of a partially-trained, clumsy juggler, does not fill anywhere near the full volume. In order to properly judge a juggled pattern, resolution on the order of 2 cm is necessary. Obviously, higher resolution would enable finer judgment of the performance of a juggler-in-training and would therefore be desirable. On the other hand, a usable system could be implemented with somewhat coarser resolution.

5.1.2 Temporal Resolution and Latency

Human perception places limits on the update rates and latencies required to make a useful tracking system for use in interactive juggling applications. Another limit is brought about by the time scale of the motion of the balls in the air.

First, we consider the motion of the balls being juggled. For an ordinary toss in a three-ball pattern, a typical height of about \( h = 0.5 \) meters would be attained. Such a toss will spend

\[
t = 2\sqrt{2h/g} = 2\sqrt{1/9.8} = 0.64 \text{ seconds}
\]

from throw to catch. Because the motion will be parabolic, at least three samples per toss will be required to uniquely determine the trajectory of the throw. This sets an absolute minimum sample rate of 4.7 Hz for three objects. Of course, many jugglers will juggle a faster pattern than this estimate, so faster sampling would be beneficial.

It is interesting to note that, typically and counterintuitively, as the number of objects being sampled increases, the sample rate required will actually decrease. This is because each throw in the pattern must be higher in order to allow the juggler enough time to make the extra throws necessary. Claude Shannon, one of the pioneers
of both information theory and the mathematical study of juggling, showed that

\[ N = 2 \frac{D + T}{D + B} \quad , \tag{5.1} \]

where \( N \) is the number of objects being juggled; \( D \) is the "dwell time," or time that a ball rests in a hand between catch and throw; \( T \) is the "throw time," or time that a ball is in the air on a throw; and \( B \) is the "between time," or time that a hand spends empty between a throw and a catch \([7]\). Rearranging,

\[ T = \frac{N}{2} \left( B + D \right) - D \quad . \tag{5.2} \]

The rate throws per hand is given by \((B + D)^{-1}\), so for a fixed rate of throws, \(B + D\) must be held constant. Assuming that there is very little between time, this means that \(D\) is independently constant, so the throw time, \(T\), goes linearly with the number of balls for a fixed juggling rate.

While approximately 5 Hz would be a sufficient sampling rate to record the trajectories of the juggled objects, this precludes providing real-time interactive feedback about the gestures involved in the juggling pattern. The motion of the balls while held is not a simple parabola, so a higher sampling rate is necessary to characterize the activity in this part of the pattern. Because one of the critical skills for successful juggling is efficient, controlled motion between a catch and the subsequent throw, this portion of the juggling pattern should be captured accurately. Using (5.1) with \(T = 0.6\) seconds and \(B = 0.5T\) as an example, a dwell time of \(D = 0.3\) seconds results. This is the amount of time during which the juggler is gaining control of the caught object and preparing for its throw. If we somewhat arbitrarily request at least 10 samples during this crucial period, a sampling rate of 33 Hz is necessary. A target update rate of approximately 30 Hz then would be a reasonable goal. This should provide acceptable perceptual feedback as well.

The other timing concern is the sampling latency. Continuous position updates would be of little value if they could not be reported until a few minutes after the associated events occurred. Physically, some degree of latency is unavoidable because
the information required to make our measurements propagates with a finite velocity. In order to be useful, the latency due to measurement propagation should be less than the duration of the gesture that is being analyzed. This requirement would permit a response to a gesture as quickly as the gesture could be recognized, maximizing the perceived causal relationship between gesture and system response. Experience with the FKB sonar tracking system suggested that a latency of approximately 30 milliseconds of latency made this relationship unclear. In the context of a juggling training system, rapid response would be desired so that advice could be given immediately after a mistake was made. Based on our above estimate that a typical juggling gesture would last 0.3 seconds, latency of, say 10 milliseconds should be acceptable.

5.2 Sonar Hardware

The tracking system required the design and construction of a prototype sonar system. In our proof-of-principle test implementation, this consisted of a single receiver and three transmitters. With three transmitters, an estimate of the channel sharing properties of the system could be estimated. By using one of the transmitters to send random ultrasonic noise, noise response of the system could be gauged. A full tracking system would require at least four receivers and a host processor to perform the triangulation and display. This would require only a straightforward extension of our hardware and ordinary triangulation routines.

5.2.1 Transmitter

A schematic of the sonar transmitter can be found in Appendix A. The transmitter is based on a PIC16C76 microcontroller, which runs at 10.24 MHz. A 40 kHz carrier is produced from the system clock by a CMOS divider. External CMOS logic is provided to permit straightforward BPSK or OOK modulation of the 40 kHz signal. In addition, the PIC can exert absolute control over the sonar output line, permitting arbitrary binary modulation.

The sonar output line is amplified by a push-pull bipolar amplifier stage to provide
the power required by the transducer. The transducer is a piezoelectric ultrasonic microphone. This transducer has a bandwidth of approximately 4 kHz centered on 40 kHz.

The PIC firmware generates a PN sequence for transmission. This PN sequence is the output from a 4-bit 2-tap LFSR, which is the 15-chip sequence “100011110101100.” At a chip rate of 1 kHz, this sequence is transmitted using OOK modulation. Because the bandwidth of the transducer is narrow, no filtering is required to convert the square wave signal from the carrier clock to smoother pure carrier tones. The high Q of the transducer is a sufficient filter to block the unwanted harmonic content.

5.2.2 Receiver

A schematic of the sonar receiver can be found in Appendix A. An ultrasonic microphone matched to the transmitter's transducer is used to convert the ultrasonic power back to electrical impulses. A common emitter amplifier and two op-amp gain stages amplify the incoming signal. The OOK signal is then diode detected to remove the carrier, leaving only the received data.

A PIC16C76 on the receiver can sample this received data using its on-board 8-bit ADC to permit conventional correlation analysis of the LFSR sequence. This digitization is performed at 10 kHz, or 10 samples per chip. The analog output is also buffered and passed to an audio jack to allow easy connection to other hardware. In particular, this output can be used to bypass the PIC correlator and use the raw analog signal as the forcing input to our NLL prototype.

The output from the PIC processing can be transmitted via a 115.2 kbaud RS232 serial link to a PC. In addition, a DAC daughter card we designed can be used to output the results of the transmission for direct viewing on an oscilloscope. The latter option proved extremely valuable, as high-speed serial communications to our test PC (running the Windows 98 operating system) proved to be unreliable.
5.2.3 Correlation Processing

As a reference, we implemented a correlation detector in the PIC on the receiver. For each sample recorded, a correlation value is computed and output. The value computed for sample interval $k$ is

$$C_k = \sum_{i=1}^{15} L_i \sum_{j=1}^{10} S[(k - 149) + 10(i - 1) + (j - 1)] ,$$  \hspace{1cm} (5.3)

where $L_i$ is the LFSR sequence, with each $L_i \in \{1, -1\}$, and $S$ is the set of samples, with $S[n] \in \{0, 1, 2, \ldots, 255\}$. Because the computational power of the PIC on the receiver board is limited, fully computing this sum at 10 kHz is unachievable. However, only two of the samples involved in the sum actually change from one time step to the next, suggesting a more efficient differential implementation. We can derive this algorithm as follows. Expanding (5.3) for $k$ and $k + 1$, we find

$$C_k = L_1 (S[k - 149] + \cdots + S[k - 140]) +$$
$$L_2 (S[k - 139] + \cdots + S[k - 130]) +$$
$$\cdots +$$
$$L_{15} (S[k - 9] + \cdots + S[k]) ,$$  \hspace{1cm} (5.4)

and

$$C_{k+1} = L_1 (S[k - 148] + \cdots + S[k - 139]) +$$
$$L_2 (S[k - 138] + \cdots + S[k - 129]) +$$
$$\cdots +$$
$$L_{15} (S[k - 8] + \cdots + S[k + 1]) .$$  \hspace{1cm} (5.5)
Subtracting (5.4) from (5.5),

\[ C_{k+1} - C_k = L_1 (S[k - 139] - S[k - 149]) + \]
\[ L_2 (S[k - 129] - S[k - 139]) + \]
\[ \cdots + \]
\[ L_{15} (S[k + 1] - S[k - 9]) \] (5.6)

Rearranging, we find

\[ C_{k+1} - C_k = S[k - 149] (0 - L_1) + S[k - 139] (L_1 - L_2) + \]
\[ \cdots + S[k - 9] (L_{14} - L_{15}) + S[k + 1] (L_{15} - 0) \] (5.7)

Thus,

\[ C_{k+1} = C_k + \sum_{i=0}^{15} S[k - 149 + 10i] (L_i - L_{i+1}) \] (5.8)

where we add the boundary conditions that \( L_0 = L_{16} = 0 \). This reduces the task of our correlator to 15 additions per time step, since the \( L_i - L_{i+1} \) can be precomputed for a fixed LFSR sequence.

The PN sequences that we transmit have the property that their autocorrelation function demonstrates a sharp peak at \( \tau = 0 \). Of course, because they are periodic sequences, this peak will also occur for \( \tau = \pm nT \), where \( T \) is the period of the sequence.

### 5.3 Object Tracking

The output of the system in which these sonars would be used is a set of positions of the transmitters in the tracking space. To generate these measurements, the relative phase of the received signal at several points in space is measured. Because the chip clock rate is known, these phase measurements can be converted into time lags, then
using the speed of sound in air as a conversion factor, these time lags can be converted into differences in distance from the receivers to the transmitter. The algorithms used for inverting these measurements into 3-dimensional positions are beyond the scope of this thesis. However, it is worth noting that because we are not operating with a global phase origin, we must add one receiver more than would otherwise be necessary in order to determine the unknown phase of the transmitter.

The repeat time of the PN sequence gives an upper bound on the range of the tracking system in a phase-based tracking situation. By constraining the range to $r < (v_{\text{sound}}) (t_{\text{repeat}})$, the relative phase offset will be uniquely defined.

Regardless of the method used for tracking the phase, there are several tradeoffs inherent in the spread spectrum sonar tracking system we envision. First, the latency of the system increases with increasing measurement range. Clearly, we cannot update a measurement until the signal from the transmitter is received, and this will increase as we move the transmitter farther from the receivers. Another tradeoff is between bandwidth and sensitivity. The height of the correlation peak of the PN sequence increases as more bits are included in each sample. Thus, the measurement peak becomes better defined as we increase the bandwidth of the system.

5.3.1 Tracking Using the Correlator

Determining a phase estimate using the correlation detector is a straightforward process. Because the LFSR autocorrelation spectrum has a distinctive peak when the lag is a multiple of the repeat length, tracking the position of the highest peak in the correlation spectrum gives a measure of the relative phase. Thus, the phase estimation algorithm operates by first recording the correlation spectrum over a single repeat time, then finding the peak and using its offset as the phase measurement. In a very noisy system, it may be desirable to average over several repeat times to increase the certainty of the phase measurement. This is a knob that can be tuned to optimize the tradeoff between update rate (and the ability to see transient behavior) and precision.

The accuracy to which we can estimate the phase is dependent on the certainty
of the location of the peak. This is limited by the computational power available and by the bandwidth of the signal used in transmission. As the chip rate increases, the location of this peak becomes better defined. In addition, more bandwidth allows for sharper transitions, reducing the uncertainty in the location of the peak. The computational power available is important because it is possible (and desirable) to oversample the received signal to allow for sub-chip phase measurements and clock misalignment. Because faster sampling requires more correlation operations, more computational power becomes a limitation for measurement precision.

5.3.2 AFSR-Based Tracking

The primary issue for the AFSR is generating a relative phase based on the AFSR state. One method for doing this is to maintain a slow ramp that increases monotonically until the AFSR reaches a particular state (say, all "1" values). Then, the ramp drops to zero and works its way up again. This can be implemented using only simple, inexpensive, low-power hardware and gives a usable phase reference. An example circuit is shown in Figure 5-1.

The discrete time AFSR is somewhat tricky to use in this model. By itself, the AFSR cannot see below a single chip time — its dynamics are completely synchronous with the chip clock. A real-world AFSR would require a method for clock estimation, anyway, perhaps implemented via a PLL that would operate in parallel and drive the AFSR dynamics. In this situation, the PLL would provide fine timing information while the AFSR would distinguish time lags over multiple chips.

A better solution would be to enable the AFSR to entrain to the clock rate as well. This is just the task that the NLL is sought so perform. Thus, for an effective analog spread spectrum sonar tracking system, a continuous time AFSR is really the optimal embodiment.
Figure 5-1: Example circuit to generate a phase ramp that resets on the all-“1” state of the AFSR. This will reset to approximately $V_0 = V_{gs}$, where $V_{gs}$ is the FET gate-source voltage. The slope of the ramp will be $I/C$. 
Chapter 6

Results and Analysis

6.1 Correlation Detection

6.1.1 Single Object Performance

Identifying the phase of the PN sequence from a single transmitter was the first step in evaluating the performance of our prototype sonar. Our test setup for this task consisted of a single sonar transmit board and its transducer located at one end of a test bench and the receiver/correlator board and its microphone at the other. A second transmitter connected to a function generator was used to inject ultrasonic noise to measure the noise rejection capabilities of the sonar.

The DAC spectrum output from the receiver board was used to verify system function and observe dynamic behavior. For analysis purposes, 1.5 seconds of spectrum data was output through the serial port of the correlator board and recorded on a PC. The data presented in this section were collected via this method and printed. Unless otherwise specified, the x-axis is measured by sample index (at 0.1 milliseconds per sample) and the y-axis is the 8-bit value reported by the correlator.

Data and Numerical Analysis

We first verified the ability of the sonar to detect the transmitted signal and correctly compute the correlation spectrum for the received signal. For our PN sequence with a
repeat length of 15 chips and our 1 kHz chip rate, we expected a correlation peak every 15 milliseconds. The measured spectrum (Figure 6-1) clearly shows this behavior. The clean regularity of this signal illustrates that there is little noise pollution in this region of the ultrasonic spectrum. The peaks corresponding to alignment of the received sequence with the phase of its reference LFSR sequence are approximately 14 dB stronger than the next largest positive peaks. While this gives some measure of the level of certainty of correct identification of the phase offset, it should be noted that the entire optimal correlation sequence is a known function. As a result, it would be possible to track more features of this pattern to increase accuracy.

The noise resistance of the single transmitter case was next tested by turning on the frequency generator. The generator was set to sweep frequency from 38 kHz to 42 kHz, covering the full bandwidth of the sonar transducer. By setting the sweep time to approximately 1 millisecond, a good approximation to white noise was achieved. An example spectrum is shown in Figure 6-2. Figure 6-3 shows the effect of varying levels of noise power.

Figure 6-1: Correlation peaks for a single transmitter and no other ultrasonic source. Peaks are clearly visible above the baseline and occur with the expected separation of 150 samples, or 15 milliseconds.
Figure 6-2: Example of correlation peaks in the presence of noise. This sample was recorded with an SNR of approximately -13.2 dB.

Figure 6-3: Correlation peak height versus SNR.
A qualitative observation of the behavior of the correlation spectra was made as the transmitter was slowly moved relative to the receiver. For this portion of the testing, the scope was triggered at the beginning of each repetition of the LFSR sequence by the transmitter. This provided a fixed phase relationship between the signal and the beginning of the spectrum sweep. As expected, the spectrum could be observed to slide forward and backward in phase as the transmitter was moved. Unfortunately, the height of the peaks varied noticeably as the transducer was moved. Because this was visible in motions of only a few inches over foot-scale distances, we believe this was caused by the high directionality of the transducers. The spectra remained clearly identifiable, however. To quantify the observation of the sliding spectra, we recorded measurements of the first correlation peak’s position as the position of the transmitter was moved to known distances. This data is shown in Figure 6-4.
Further Analysis

The spectrum presented in Figure 6-1 is very encouraging. The correlation sequence is very clearly visible without significant distortion. If such a spectrum could be generated by this system in practice, it is clear that accurate tracking could be performed.

The noise response of the system is also encouraging. The correlation peaks remain clearly identifiable to around a -10 dB SNR. Below that, while the peaks do become comparable to other features of the correlation spectrum, qualitative observation suggests that tracking could potentially still be performed. The zero crossing between the correlation peak and the maximal anticorrelation (the next largest peak in magnitude, but in the opposite direction) is identifiable down to at least -18 dB.

The variation in spectrum with distance and relative orientation of the transducers was more sensitive than in an optimal setting. The inclusion of automatic gain control (AGC) in the sonar front end should minimize this effect. In the case of multiple transmitters, the situation becomes complicated by the possibility of one transmitter's being located much closer to the receiver than another. We will discuss this below.

6.1.2 Multiple Object Performance

The ability of the sonar to track multiple objects is crucial to its utility for our applications. In our model for channel sharing, otherwise identical objects are identified by the random startup phase of their chip clocks. The length of the chip sequence becomes important for determining how many peaks can be measurably distinct.

For this set of measurements, two transmitter boards and a noise source were used to produce the ultrasonic signals. The relative locations of the transmitters was varied to control the relative power received from each. Each board maintained its own clock and no synchronization was performed between the transmitters and the receiver.
Data and Numerical Analysis

A first test of multiple transmitter detection was performed by placing the transmitters side-by-side. Because our sonar front end does not use AGC, we adjusted the gain manually to ensure that saturation did not occur. The resulting spectrum is shown in Figure 6-5. As we expect, with two transmitters, there are two distinct peaks in each 15 millisecond period. Each repeats after every 15 milliseconds, and the phase relationship between the two is approximately constant over the measurement period. The stronger and weaker peaks are approximately 5.7 dB and 3 dB over the next highest positive peak, respectively. Again, other features in the correlation sequence could be tracked to increase confidence in the phase estimates.

Further Analysis

While the peaks in Figure 6-5 are rather clear, the very short (15 chip) PN sequence that we are using for tracking appears to be too short. Already, the peaks are difficult to distinguish from each other, and in many cases, we could not clearly identify a sec-
ond set of peaks to assign to the second transmitter. In addition to the issue of finding space in the correlation spectrum to differentiate more transmitters, we appeared to be having trouble due to input saturation. This problem could be ameliorated by using AGC in the front end, or by using a zero-crossing detection method instead of our correlator implementation.

### 6.1.3 Implications for a Tracking System

The theoretical resolution of our system is approximately 1/10th the chip period times the speed of sound, or approximately 3.4 cm. Our observations indicated that this level of resolution was, in fact, within reach. With our 15 ms PN repeat time, we our maximum range was approximately 5.1 m, with a 67 Hz update rate, independent of the number of objects being tracked. We are happy with these measures of performance.

Unfortunately, we fell short on one of our major goals. First, it is apparent from our experimentation that tracking three objects is beyond the capabilities of this system. However, by increasing the PN sequence length by a factor of two, we believe we could successfully locate at least three distinct peaks in the correlation spectrum. This would still be capable of meeting the 30 Hz update rate design goal we set.

The power consumption of the transmitter we used was approximately 160 mW (18 mA at 9V). This is somewhat troublesome as adding heavy batteries to an object to be juggled quickly adds undesired weight. However, our transmitter was certainly not optimized for power savings, so we expect that savings could be achieved.

### 6.2 AFSR Entrainment

Because an AFSR requires a clock recovery mechanism, we were unable to construct a hardware AFSR to perform sonar entrainment. Instead, we used our sonar to sample the transmitter’s signal and captured these samples on our PC. Then, using a MATLAB model of the AFSR and manual clock alignment, we tested the entrainment capabilities of the algorithm.
Figure 6-6: AFSR with $\epsilon = 0.25$ and $G = 3.0$ entraining on captured sonar data. The top plot shows both the noisy input data and the AFSR output. The bottom plot shows the error between the two signals. The ring-up transient in the first 20 chips is typical of the AFSR and reminiscent of a damped simple harmonic oscillator.

6.2.1 Simple Entrainment

In our first experiment, we captured sonar data from a transmitter with no ultrasonic noise source. Using an AFSR model similar to that used in [13], with $\epsilon = 0.25$ and $G = 3.0$, we were able to successfully and repeatably entrain. An example entrainment series is shown in Figure 6-6.

We were somewhat surprised to note that after cleanly entraining and remaining in perfect synchrony with the input sequence for several hundred chips, the AFSR would catastrophically drop out of entrainment. The resulting increase in error signals is shown in Figure 6-7. We believe this is due to slight clock alignment errors caused by our inexact, global clock alignment method. Our sampled sonar data was captured at approximately 10 samples per chip. We downsampling this to generate the data for use in the AFSR simulation, which makes a transition only on chip boundaries. Over a few hundred chips, the difference between “approximately 10” samples per chip and the true number accumulated enough phase error to drive the
AFSR out of synchronization. By carefully adjusting the sample rate, we found that downsampling by an average of $10.133 \pm 0.005$ samples per chip was close enough to maintain entrainment over about 500 chips. We believe that in a real system, a local clock estimator would be capable of maintaining the AFSR clock to suitable accuracy since local corrections could be made as the clocks shifted relative phase.

### 6.2.2 Entrainment with Noise

In a second test, we added noise to the received sonar signal using our frequency generator. The noise power was set to approximately the same level as the transmitter power. Our AFSR simulation successfully entrained to this data as well. One issue was that the noisy data tended to pull the AFSR states away from the rails through $\epsilon$. If $\epsilon$ was set small enough that this was not observable, then there was not enough signal injection to entrain. Careful balancing of $\epsilon$ and the internal gain is necessary to minimize this noise injection. In addition, before using the AFSR signal for further computations, it appears to be advisable to first force its output to the rails to produce a true digital signal. The entrainment sequence from this test is shown in Figure 6-8.

Figure 6-7: Plot of the error signal over a long time scale. The growth in error shows loss of entrainment due to accumulated clock skew.
6.2.3 AFSR Phase Tracking

We did a brief simulation of the phase tracking AFSR described in Chapter 5. In addition to running the AFSR, on each update cycle we incremented our ramp function to provide a measure of the expired PN phase. Upon achieving the AFSR state where all chip values are “1,” we reset the phase ramp to a fixed reset value. In this simulation, we placed two AFSR receivers some distance from the and allowed them to entrain to a single LFSR transmitter whose signal was retarded appropriately. We then plotted the difference between their phase ramps. The results of our simulation are shown in Figure 6-9.

We define the phase ramp function as

\[
\phi(t) = \begin{cases} 
V_0 + at, & \text{if } 0 \leq t < t_{\text{rep}}; \\
\phi(t - t_{\text{rep}}), & \text{if } t \geq t_{\text{rep}}; \\
\phi(t + t_{\text{rep}}), & \text{if } t < 0.
\end{cases} \tag{6.1}
\]

Then, if we have two transmitters separated by a propagation time \( \tau \), one ramp will
Figure 6-9: AFSR phase estimation using the phase ramp. Upper plots show the ramps as two 7-bit AFSR entrain to differently lagged versions of the same data. The bottom plot is the difference between the two ramp values, $\Delta \phi$.

be $\phi_1(t) = \phi(t)$, and the other $\phi_2(t) = \phi(t - \tau)$. It can be shown that the difference between these is the time dependent value

$$\Delta \phi = \phi_1(t) - \phi_2(t) = \begin{cases} a\tau, & \text{if } \tau \leq t < t_{rep}; \\ a(\tau - t_{rep}) & \text{otherwise}. \end{cases} \quad (6.2)$$

In order to convert this function into a constant measure of the phase offset between the two, we should add or subtract $a t_{rep}$ to the computed $\Delta \phi$ in order to keep it in the range $0 \leq \Delta \phi < a t_{rep}$.

### 6.3 Conclusions

We have developed a prototype spread sonar device that meets most of the design goals outlined in Chapter 5. This goal was met using modest hardware resources, even for the digital correlation receiver. Meeting the full set of design goals with this digital system would require somewhat more computational power, but is still well
within reach.

In addition, we have for the first time demonstrated the dynamics of the AFSR performing a useful task using data from a real world system. Although this was shown using a MATLAB simulation on captured data rather than in a more exciting hardware implementation, this is still an important step toward constructing a real hardware AFSR. We have shown that the dynamics of the discrete time AFSR are very sensitive to clock errors, an observation that has serious implications for the future direction of AFSR research. In many ways, this discovery serves to underline the value and importance of lifting the AFSR into a continuous time system to obviate the need for separate clock recovery.

Unfortunately, we were unable to develop the dynamics of the continuous time NLL to demonstrate entrainment. Research to this end appears promising and the rewards should such a system be perfected are great. There are a great many applications that would benefit from the option to use a high-speed, low-power, low-parts count, low-cost spread spectrum communications network. The path to this end is one through largely unexplored territory in the domain of nonlinear delay differential equations and time delay systems. As a result, we expect there to be a great many discoveries and insights waiting to be found.
Appendix A

Schematics and PCB Diagrams

This appendix contains the electrical schematics and PCB diagrams used to fabricate the hardware we used in the work described in this thesis.
A.I. I Transmitter

A.I. II Schematic
A.1.2 PCB Top Layer
A.1.3 PCB Bottom Layer
A.2.2 PCB Top Layer
A.2.3 PCB Bottom Layer
Appendix B

PIC Code

This appendix contains the PIC code for the Microchip PIC16C76 microcontrollers that ran the sonar transmitter and the correlator/receiver board.

B.1 Transmitter Code

```plaintext
; spread spectrum sonar transmit code
; joey richards (bigjoe@mit.edu) feb 2000
;
list p=16c76 ; list directive to define processor
#include "p16c76.inc" ; processor specific variable definitions

_.CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON & _HS_OSC

; a few useful macros
; CHBANK n chooses the bank n
CHBANK macro which
   if which == .0
      bcf STATUS, RP0
      bcf STATUS, RP1
      exitm
   endif
   if which == .1
      bcf STATUS, RP0
      bcf STATUS, RP1
      exitm
   endif
   if which == .2
      bcf STATUS, RP0
      bcf STATUS, RP1
      exitm
   endif
   if which == .3
      bcf STATUS, RP0
      bcf STATUS, RP1
      exitm
   endif
endm
```
test for a value; branch if \[ rfile \] == value, else fall through
bashes w, status

\textbf{BRANCH\_IF} macro \texttt{rfile, value, tgt}
\begin{verbatim}
    movlw value
    subwf rfile, W
    btfsc STATUS, Z
    goto tgt
\end{verbatim}

; hardware definitions
; first three: \texttt{sctrl} (sonar control)
; \texttt{octrl} (output control)
; \texttt{sdata} (sonar data)
; these control the output to the sonar transmitter.
; for \textit{ook}, \texttt{octrl high}, \texttt{sdata high}, \texttt{sctrl gets data}
; for \textit{bpsk}, \texttt{sctrl and octrl high}, \texttt{sdata gets data}
; for arbitrary output control, \texttt{sctrl low}, \texttt{octrl} = \texttt{sdata}

\#define \texttt{SCTRL PORTB, 3}
\#define \texttt{OCTRL PORTB, 4}
\#define \texttt{SDATA PORTB, 2}

\texttt{srst} shuts resets the counter and shuts off the output waveform

\#define \texttt{SRST PORTB, 1}

; \texttt{usart pins}
\#define \texttt{RX PORTC, 7}
\#define \texttt{TX PORTC, 6}
\#define \texttt{LED0 PORTB, 5}
\#define \texttt{LED1 PORTB, 6}
\#define \texttt{LED2 PORTB, 7}

; more meaningful names for LEDs
\#define \texttt{CHIP\_LED LED2}
\#define \texttt{CHIP\_LED\_REG PORTB}
\texttt{CHIP\_LEDMASK} \texttt{equ B'10000000'}

; software variables
\texttt{w\_temp} \texttt{EQU 0x70} ; variable used for context saving
\texttt{status\_temp} \texttt{EQU 0x71} ; variable used for context saving
\texttt{fsr\_temp} \texttt{equ 0x77} ; variable used for context saving
\texttt{lsfr\_bits} \texttt{equ 0x72} ; lsfr output bits (bit 7 is newest)
\texttt{r\_lo} \texttt{equ 0x73} ; lsfr state variables
\texttt{r\_hi} \texttt{equ 0x74} ; lsfr state variables
\texttt{q\_lo} \texttt{equ 0x75} ; lsfr coeffs
\texttt{q\_hi} \texttt{equ 0x76} ; lsfr coeffs
\#define \texttt{CURRENT\_CHIP lsfr\_bits, 7}
; default lsfr taps are (3, 0)
\#define \texttt{LFSR\_DEFAULT\_Q\_LO b'0000000'}
\#define \texttt{LFSR\_DEFAULT\_Q\_HI b'0000000'}

; starting buffer values
\#define \texttt{LFSR\_DEFAULT\_R\_LO 0x00}
\#define \texttt{LFSR\_DEFAULT\_R\_HI 0xF0}
\texttt{data\_byte} \texttt{equ 0x78} ; the data in the queue for output
\#define \texttt{CURRENT\_BIT data\_byte, 0}
\texttt{sonar\_mode} \texttt{equ 0x79} ; data about the sonar setup
\texttt{ook} \texttt{equ 0x01} ; mode bits -- only 1 should be set
\texttt{bpsk} \texttt{equ 0x02} ; at a time; if none set, defaults to
\texttt{direct} \texttt{equ 0x03} ; \textit{ook}
\texttt{chip\_timer\_init\_l} \texttt{equ 0x7A} ; timer init values for chip timer
\texttt{chip\_timer\_init\_h} \texttt{equ 0x7B} ; should be 65535 = 2.5MHz/f\_chip
\#define \texttt{CHIP\_INIT\_1KHZ\_LO 0x00} ; 1khz timer init
\#define \texttt{CHIP\_INIT\_1KHZ\_HI 0xF6}
;********************************************************************
; processor reset vector
clrf PCLATH ; ensure page bits are cleared
goto main ; go to beginning of program

ORG 0x000 ; interrupt vector location
movwf w_temp ; save off current W register contents
movf STATUS,w    ; move status register into W register
movw status_temp ; save off contents of STATUS register
movf FSR, W      ; move far into W
movwf far_temp   ; save off far

; dispatch appropriate interrupt handler from here
btfsc PIR1, TMR1IF ; timer 1 overflow
call do_chip ; means chip time expired
movf far_temp, w ; retrieve copy of FSR
movwf FSR ; restore pre-isr FSR contents
movf status_temp, w ; retrieve copy of STATUS register
movwf STATUS ; restore pre-isr STATUS register contents
swapf w_temp,f
swapf w_temp,w ; restore pre-isr W register contents
retfie ; return from interrupt

;----------------------------------------
; Main Routine :
;----------------------------------------

main
 ; first the boring startup stuff
CHBANK 0

clrf PORTA
clrf PORTB
clrf PORTC

CHBANK 1
movlw 0xFF ; portA tris
movwf TRISA
movlw 0x01 ; portB tris
movwf TRISB
movlw 0xBF ; portC tris
movwf TRISC
movlw 0x07 ; turn off port A ADC
movwf ADCON1

; set up the usart for asynchronous 57600
movlw 0x10 ; 57600 at 16mhz clock
movwf SPBRG
bsf TXSTA, BRGH
bcf TXSTA, SYNC
bsf PIE1, RCIE
bsf PIE1, TXIE

; set up chip timer (tmr1)
CHBANK 0
bcf T1CON, T1CKPS1 ; 1:1 prescaler
bcf T1CON, T1CKPS0
bcf T1CON, T1OSCEN ; no t1 oscillator
bcf T1CON, TMR1CS ; use system clock
bcf T1CON, TMR1ON ; halt the clock for now
movlw CHIP_INIT_1KHZ_LO
movwf TMR1L
movwf chip_timer_init_l
movlw CHIP_INIT_1KHZ_HI
movwf TMR1H

81
movwf chip_timer_init_h

CHBANK 1
bsf PIE1, TMR1IE

CHBANK 0
movlw LFSR_DEFAULT_Q_LO ; set up the LFSR
movwf q_lo
movlw LFSR_DEFAULT_Q_HI
movwf q_hi
movlw LFSR_DEFAULT_R_LO
movwf r_lo
movlw LFSR_DEFAULT_R_HI
movwf r_hi
clrf IFSR_bits ; zero its output
bsf SRST
nop
nop
nop
bcf SRST
movlw 0x01 ; sonar in ook mode
movwf sonar_mode
movf chip_timer_init_l.w ; get value to reinit timer
movwf TMR1L
movf chip_timer_init_h.w ; get value to reinit timer
movwf TMR1H
bcf PIR1, TMR1IF ; clear the interrupt flag

movlw CHIP_LED_MASK
xorwf CHIP_LED_REG, F ; blink the led
call IFSR_update ; update the IFSR
call compute_output ; update the sonar output
return

; ----------
; interrupt subroutines ;
; ----------

; do_chip is called when the chip timer expires
do_chip

movf chip_timer_init_l.w ; get value to reinit timer
movf TMR1L
movf chip_timer_init_h.w ; get value to reinit timer
movf TMR1H
bcf PIR1, TMR1IF ; clear the interrupt flag
movlw CHIP_LED_MASK
xorwf CHIP_LED_REG, F ; blink the led
call IFSR_update ; update the IFSR
call compute_output ; update the sonar output
return

; ----------
; IFSR update routine, called at the start of each chip
IFSР_update

bcf STATUS, C ; clear carry bit
rrf r_hi, F ; rotate bits in the shift register
rrf r_lo, F ; and put the low bit into carry
btfss STATUS, C ; carry bit is the next data bit
goto IFSR.do_output ; jump if bit was 0
; If we got here, then the bit popped out of the IFSR was a 1.
; That means that for each bit in the register, we want to add
; q_i to the bit just shifted there. This is the same as
; doing an XOR with each bit, which is much faster
movf q_hi, W ; get coeffs
xorwf r_hi, F ; and xor
movf q_lo, W ; get coeffs
xorwf r_lo, F ; and xor

lsr_do_output
; make absolutely sure that nothing has touched the carry bit
; between here and the previous rrf
rrf lfsr_bits, F ; shift the next bit into the output bin
bcf STATUS, C ; in case I forget somewhere else...
return

compute_output combines the current data bit with the current chip value
; and presents it on the sonar output. Should be called whenever either a
; baud or a chip expires
compute_output
btfsc CURRENT_BIT
goto invert_chips ; here, the data bit is a 0 so we pass the chips straight through
btfsc CURRENT_CHIP
call do_sonar_1
btfss CURRENT_CHIP
call do_sonar_0
goto output_computed
invert_chips
; here, the data bit is a 1 so we invert our chips
btfsc CURRENT_CHIP
call do_sonar_0
btfss CURRENT_CHIP
call do_sonar_1
output_computed
return ; done

; these routines do the right thing as far as outputting sonar bits
;
do_sonar_0
btfsc sonar_mode, ook
goto do_ook_0
btfsc sonar_mode, bpsk
goto do_bpsk_0
btfsc sonar_mode, direct
goto do_direct_0
;
; default is ook (if nothing else specified)
do_ook_0
bsf OCTRL
bcf SDATA
bcf SCTRL ; sctrl = data
goto did_sonar_0
do_bpsk_0
bsf OCTRL
bsf SCTRL
bcf SDATA ; sdata = data
goto did_sonar_0
do_direct_0
bcf SCTRL
bsf OCTRL ; octrl = /data
did_sonar_0
return
do_sonar_1
btfsc sonar_mode, ook
goto do_ook_1
btfsc sonar_mode, bpsk
goto do_bpsk_1
btfsc sonar_mode, direct
goto do_direct_1
;
; default is ook (if nothing else specified)
do_ook_1
bsf OCTRL

83
B.2 Correlator Code

; correlator board code
; April, 2000 Joey Richards bigjoe@mit.edu
;
; clock should be a 16MHz crystal
;
; correlates 10kHz adc samples with 4-bit maximal length
; if sequence and spews correlation values out the serial
; port at 115.2kbaud
;
list p=16c76 ; list directive to define processor
#include <p16c76.inc> ; processor specific variable definitions

_CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON & _HS_OSC

; BANK n chooses the bank n
CHBANK macro which
if which == .0
  bcf STATUS, RP0
  bcf STATUS, RP1
  exitm
endif
if which == .1
  bcf STATUS, RP0
  bcf STATUS, RP1
  exitm
endif
if which == .2
  bcf STATUS, RP0
  bcf STATUS, RP1
  exitm
endif
if which == .3
  bcf STATUS, RP0
  bcf STATUS, RP1
  exitm
endif
endm

; hardware definitions
#define I2C1 PORTC, 4 ; control lines for parallel bus
#define I2C2 PORTC, 5 ; (in case we try to wire multi-pics)
                 ; (port B is the data byte)
#define DEMOD1 PORTA, 0 ; these are accessed through the ADC
#define DEMOD2 PORTA, 1
```c
#define LED0 PORTA, 2
#define LED1 PORTA, 3

; vbl defs
; 0x20 - 0x6A, 0xA0-0xEA reserved for data storage

w_temp EQU 0x70 ; variable used for context saving
status_temp EQU 0x71 ; variable used for context saving
AARGBO equ 0x6B ; for fixed point routine
AARGB1 equ 0x6C ; aargb0(MSB)-aargb2(LSB)
AARGB2 equ 0x6D
BARGB0 equ 0x6E
data_origin equ 0x7B ; pointer to start of samples
corr_low equ 0x73 ; correlation buffer
corr_high equ 0x74
TEMPB0 equ 0x75 ; for fixed point routine
TEMPB1 equ 0x76
tmp equ 0x77 ; used in interrupt!
tick_counter equ 0x78
mystatus equ 0x79
new_sample equ 0x7A
laps_left equ 0x7B ; number of times to loop the output cycle
tmp2 equ 0x7C
mattcount equ 0x7D

; mystatus bits
tick equ 0x00 ; 0 bit of mystatus
start equ 0x01 ; hmm

; this next macro is from the microchip 16x8 fixed point
; multiplication libraries
UMUL1608 macro
    Max Timing: 1+6+7*11 = 84 clks
    Min Timing: 1+2*8+4 = 21 clks
    PM: 1+2*8+4+6*7 = 63 DM: 4
    variable i = 0
    BCF STATUS, C ; clear carry for first right shift
    while i < 8
        BTFSC BARGB0, i
        GOTO UM1608NA#v(i)
    endw
    CLRF AARGBO ; if we get here, BARG = 0
    CLRF AARB1
    RETURN

UM1608NA0 RRF AARGBO, F
RRF AARGB1, F
RRF AARGB2, F

    variable i =1
    while i < 8
        BTFSS BARGB0, i
        GOTO UM1608NA#v(i)
    endw
    UM1608A#v(i) MOVF TEMPB1,W ADDWF AARGB1, F MOVF TEMPB0,W
```
```
BTFSC STATUS, C
INCFSZ TEMPBO, W
ADDF AARGBO, F
UM1608NA v(i) RRF AARGBO, F
RRF AARGB1, F
RRF AARGB2, F

variable i = i + 1
endw
endm
;;;; end fixed point macro

<table>
<thead>
<tr>
<th>ORG 0x000</th>
<th>; processor reset vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>clrf PCLATH</td>
<td>; ensure page bits cleared</td>
</tr>
<tr>
<td>goto main</td>
<td>; go to beginning of program</td>
</tr>
</tbody>
</table>

| ORG 0x004 | ; interrupt vector location |
| movwf w_temp | ; save off current W register contents |
| movf STATUS, w | ; move status register into W register |
| movwf status_temp | ; save off contents of STATUS register |

; first, see which interrupt fired
btfsc PIR1, RCIF | ; USART reception
call do_serial_in
btfsc INTO1, T0IF | ; timer
call do_timer

movf status_temp, w | ; retrieve copy of STATUS register
movwf STATUS | ; restore pre-isr STATUS register contents
swapf w_temp.f
swapf w_temp.w | ; restore pre-isr W register contents
retfie | ; return from interrupt

; interrupt handlers
;*****
;* do_serial_in
;*
; Interrupt handler that deals with inputs over the serial line
;
do_serial_in

movf RCREG, W
movwf tmp
bcf RCSTA, CREN
bsf RCSTA, CREN

moviw 'g' | ; 'g' -> start sending a spectrum
subwf tmp, W
btfsc STATUS, Z
goto start_spectrum
;
else
goto serial_handled

start_spectrum
bsf mystatus, start
goto serial_handled

serial_handled
return

;*****
;* do_timer
```

86
**
; interrupt handler that deals with the 10kHz clock tick
;
do_timer
bcf INTCON, T0IF : clear int flag
movlw 65 ; leave 200 ticks
movwf TMR0 ; to get timing right
bsf mystatus, tick ; signal that we've ticked
return

main
clrf PORTA
clrf PORTB
clrf PORTC
CHIBANK 1
movlw 0xC3
movwf TRISA
movlw 0x00
movwf TRISB
movwf 0x8F
movwf TRISC
movlw 0x04
movwf ADCON1 ; only 0,1,3 are analog
movlw 0x24 ; high-speed serial, txen
movwf TXSTA
movlw 08 ; set baud rate ~= 115.2k
movwf SPBRG
bcf OPTION_REG, T0CS ; timer stuff
bcf OPTION_REG, PSA
bcf OPTION_REG, PS2
bcf OPTION_REG, PS1
bcf OPTION_REG, PS0
bsf PIE1, RCIE ; turn on serial receive interrupts
CHIBANK 0
movlw 0x89 ; adc on, fosc/32, input 1
movwf ADCON0
movlw 0xF0 ; enable serial port
movwf RCSTA
bsf RCSTA, CREN
bsf INTCON, T0IE : set up the interrupts
bsf INTCON, PEIE
bcf INTCON, T0IF
bsf INTCON, GIE ; enable global interrupts

clrf tick_counter
clrf corr_low
clrf corr_high

movlw .150
movwf corr_low ; just a tmp buffer for now
clrf data_origin
buf_clear_loop
clrf new_sample ; clear out the sample buffer
call store_sample
decfsz corr_low, F
goto buf_clear_loop

clrf new_sample
clf corr_low
clf data_origin

; the running loop:
pseudocode_loop
call get_adc_sample ; adc_in --> new_sample
call compute_corr_step ; update corr total
call store_sample ; new_sample --> data array, update data_origin

87
btfss mystatus, start
goto dontstart
movf data_origin, F
btfss STATUS, Z ; only start sending spectra at
goto dontstart ; phase = 0
movlw .150 ; spectrum is 150 ticks long
movwf tick_counter
movlw .100 ; lots o' bits
movwf laps_left.
bcf mystatus, start
call xmit_corr ; output corr to RS232 @ 115200bps and DAC
dontstart
call xmit_corr ; output corr to RS232 @ 115200bps and DAC
wait_for_tick
bcf LED0
btfss mystatus, tick
goto wait_for_tick
bcf mystatus, tick
bcf LED0
goto pseudocode_loop

;*****
;* compute_corr_step
;**
; This routine updates the stored correlation buffers after a new
; sample is received. Hardcoded for the 4-bit 2 tap LFSR sequence
; ...1000 1111 0101 100...
; Note: it is crucial that the addwf to compute the offset for the
; argument to get_wth_sample immediately before calling that function
; to preserve the carry bit
;
compute_corr_step
movf data_origin, W
movwf tmp2 ; our offset

; 1->0 gets -1
; [data_origin is still in W]
call get_wth_sample ; read the sample from memory into w
subwf corr_low, F ; add this to corr
btfss STATUS, C ; did the /borrow bit get set?
defc corr_high, F ; if _not_... then we borrowed so dec

; 11->10 gets +2
movlw .10 ; get the 40th old sample
addwf tmp2, F ; add the sample origin
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

; read the sample from memory into w
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
incf corr_high, F ; if so, then we carried so inc
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
incf corr_high, F ; if so, then we carried so inc

; 41->40 gets -2
movlw .30 ; get the 80th old sample
addwf tmp2, F ; add the sample origin
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

call get_wth_sample ; read the sample from memory into w
subwf corr_low, F ; add this to corr
btfss STATUS, C ; did the /borrow bit get set?
decf corr_high, F ; if ..not.., then we borrowed so dec
subwf corr_low, F ; add this to corr
btfss STATUS, C ; did the /borrow bit get set?
decf corr_high, F ; if ..not.., then we borrowed so dec

; 81->80 gets +2
movlw .40
addwf tmp2, F
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

call get_wth_sample ; read the sample from memory into w
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc

; 91->90 gets -2
movlw .10
addwf tmp2, F
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

call get_wth_sample ; read the sample from memory into w
subwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the /borrow bit get set?
decf corr_high, F ; if ..not.., then we borrowed so dec
subwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the /borrow bit get set?
decf corr_high, F ; if ..not.., then we borrowed so dec

; 101->100 gets +2
movlw .10
addwf tmp2, F
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

call get_wth_sample ; read the sample from memory into w
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc

; 111->110 gets -2
movlw .10
addwf tmp2, F
movlw .150
subwf tmp2, W
btfsc STATUS, C
movwf tmp2
movf tmp2, W

call get_wth_sample ; read the sample from memory into w
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc
addwf corr_low, F ; add this to corr
btfsc STATUS, C ; did the carry bit get set?
icf corr_high, F ; if so, then we carried so inc

; 111->110 gets -2
```assembly
movf tmp2, W
    ; read the sample from memory into w
    call get_wth_sample
    subwf corr_low, F    ; add this to corr
    btfss STATUS, C      ; did the /borrow bit get set?
    decf corr_high, F    ; if _not_, then we borrowed so dec
    subwf corr_low, F    ; add this to corr
    btfss STATUS, C      ; did the /borrow bit get set?
    decf corr_high, F    ; if _not_, then we borrowed so dec

; 131->130 gets +2
movlw .20
    ; get the 110th old sample
addwf tmp2, F
movlw 150
subwf tmp2, W
    ; read the sample from memory into w
addwf corr_low, F    ; add this to corr
btfsc STATUS, C      ; did the carry bit get set?
incf corr_high, F    ; if so, then we carried so inc
addwf corr_low, F    ; add this to corr
btfsc STATUS, C      ; did the carry bit get set?
incf corr_high, F    ; if so, then we carried so inc

; 151->150 gets -1 (this is the new sample)
movf new_sample, W
subwf corr_low, F
btfsc STATUS, C
decf corr_high, F
return

; store_sample
    ; stores the contents of new_sample into the address pointed to by data_origin and
    ; then increments data_origin (with proper wrapping)
movf data_origin, W
sublw .74
btfsc STATUS, C
movlw 0x20
btfsc STATUS, C
movlw 0xA0-0x4b
addwf data_origin, W
movwf FSR
movf new_sample, W
movwf INDF
clrf data_origin
movlw .150
subwf data_origin, W
btfsc STATUS, Z
incf data_origin
clrf data_origin
    ; yep --- wrap

    ; this next instruction is way out of place, but I'm trying to
    ; optimize so you know how it goes
bef LED1
bsf ADCON0, GO ; start the next sample
return

; get_wth_sample
; store-sample
```
Get the sample indexed by tmp2 (and in w)

```assembly
get_wth_sample
  sublw .74 ; if data_origin >= 75, then carry is clear
  movlw 0x20 ; low bank offset
  btfss STATUS, C
  movlw 0xA0-0x4b ; high bank offset
  addwf tmp2, W ; get address
  movwf FSR ; set FSR for indirect addressing
  movf INDF, W ; read the sample

return
```

get_adc_sample

```assembly
get_adc_sample
  btfsc ADCON0, GO ; wait for adc to finish
  goto adc_wait
  bsf LED1
  movlw 0xFF ; compute 0xFF - ADRES
  movwf new_sample ; to de-invert the input value
  movf ADRES, W
  subwf new_sample, F ; 0xFF - ADRES -> new_sample

  movlw .10
  subwf data_origin, W
  movlw 0xFF ; search for me
  clrf new_sample
  btfss STATUS, C
  movwf new_sample

return
```

xmit_corr

```assembly
xmit_corr
  movf corr_low, W
  movwf AARGB1
  movf corr_high, W
  movwf AARGB0
  movlw 0xBA ; add enough to force positive
  addwf AARGB1, F
  btfsc STATUS, C ; check carry bit
  incf AARGB0, F ; increment if carried
  moviw 0x45
  addwf AARGB0, F
  movf AARGB1, W ; set up for multiply
  movwf TEMPB1
  movf AARGB0, W
  movwf TEMPB0
  clrf AARGB2
  movlw 0xDB ; 1.7109 ~ 1.713333333 in 1.7 fixed point notation
  movwf BARGB0
  UMUL1608 ; do the multiply
```

Reads a sample from ADC0 into new_sample

Returns the sample indexed by tmp2 (and in w)
now we want to take bits 23.15 because that is where our value can end up (do the math — \(38250 \times 0xDB = 0x7FD1AE\), so the high bit is always 0)

```asm
rlf AARGB1, F ; we are ignoring everything that doesn't make it into the high byte
rlf AARGB0, F ; we just multiplied by 2
; now the good bits are all in AARGB0
```

```asm
movf tick_counter, F
btfsc STATUS, Z
goto xmit_done ; xmit if it didn't become 0
```

```asm
xmit_wait
btfss PIR1, TXIF ; is the buffer clear?
goto xmit_wait ; nope...
```

```asm
; now the txreg buffer is cleared so we can xmit
movf AARGB0, W
movf new_sample, W
movwf TXREG ; beam it out, scotty
```

```asm
decfsz tick_counter, F ; update the counters
goto xmit_done
```

```asm
movlw .0
decfsz laps_left, F see if there's another lap
movlw .150
movwf tick_counter
```

```asm
xmit_done
; now display that on the dac
movf AARGB0, W
::
```

```asm
movlw 0x7f
movwf PORTB ; port B goes out to dac
bsf I2C1 ; (I only need one of these but I don't
bsf I2C2 ; remember which it is...)
nop
bsf I2C1 ; latch the sample
bcf I2C2
```

```asm
return
END
```
Appendix C

MATLAB Simulation Code

This appendix contains the various MATLAB scripts that were used for the simulations described in this thesis.

C.1 AFSR Entrainment

These files were used to test AFSR entrainment on the sampled sonar data. The first and second script were used to entrain to the clean and noisy sampled data, respectively.

```matlab
% file: afsr1filt.m
% filters the file afsr1.dat and feeds to afsr
% (clock recovery by hand)
%
x = load('afsrl.dat');
offset = 19; % first clock transition
rate = 10.133;
filtseq = zeros(1, ceil(length(x)/rate)+500);
for i=1:round( (length(x)-offset)/rate-100),
crap(round(rate*i)+offset) = 160;
filtseq(i) = x(round(rate*i)+offset);
end
%
now remove the dc
chips = filtseq(1:(length(filtseq)-610));
chips = chips - mean(chips);
%
and scale by its stdev
chips = chips / std(chips);
%
now see if we can entrain!

afsr_seq = run_afsr(chips, 0.25, 3.0, 0);
```
```matlab
subplot(2,1,1);
plot(1:500, afsr_seq(1:500), 1:500, chips(1:500));
subplot(2,1,2);
plot(afsr_seq - chips);

% file: afsr2filt.m
% filters the file afsri.dat and feeds to afsr
% (clock recovery by hand)

x = load('afsr2.dat');
offset = 14; % first clock transition
rate = 10.133;
crap = zeros(1, ceil(length(x)/rate));
end
filtseq = zeros(1, ceil(length(x)/rate)+500);
for i=1:round( (length(x)-offset)/rate),
crap(round(rate*i) + offset) = 160;
filtseq(i) = x(round(rate*i)+offset);
end
% now remove the dc
chips = filtseq(1:(length(filtseq)-610));
chips = chips - mean(chips);
% and scale by its std
chips = chips / std(chips);

% now see if we can entrain!

afsr_seq = run_afsr(chips, 0, 0.25, 3.0, 0);

subplot(2,1,1);
plot(1:500, afsr_seq(1:500), 1:500, chips(1:500));
subplot(2,1,2);
plot(afsr_seq - chips);

% file: run_afsr.m
function S = run_afsr(input, delay, epsilon, gain, noisevar)

S = zeros(1,length(input));
for i = 1:length(input),
afsr(afsr_len+1) = epsilon*rcv(i) + (1-epsilon)*afsr_step(tap1, tap2, afsr, gain);
end
```
\[ S(i) = \text{afsr}([\text{asr}\_\text{len}+1]; \]
for \( j=1:\text{afsr}\_\text{len} \),
\[ \text{afsr}(j) = \text{afsr}(j+1); \]
end
end

% file: afsr-step.m
function \( x = \text{afsr}\_\text{step}(\text{tap1}, \text{tap2}, \text{asr}, \text{gain}) \)
\% \( x = \text{afsr}\_\text{step}(\text{tap1}, \text{tap2}, \text{asr}) \)
\% \% computes the next value to be stuffed into an \text{afsr}
\% in the \(-1/+1\) basis using Ben’s rule
\% \( \text{new} = -(\text{asr}[\text{tap1}] \times \text{asr}[\text{tap2}]) / \min(\text{abs}(\text{asr}[\text{tap1}] \times \text{asr}[\text{tap2}]), 1) \)
\% \% \text{mag} = \text{abs}(\text{asr}(\text{tap1}) \times \text{asr}(\text{tap2}));
\text{if} (\text{mag} < 1e-30)
\( x = 0; \)
\text{else}
\text{sgn} = \text{asr}(\text{tap1}) \times \text{asr}(\text{tap2}) / \text{mag};
\( x = -\text{sgn} \times \min(\text{gain} \times \text{mag}, 1); \)
end

C.2 AFSR Phase Tracking

These files were used to generate the sample plots for the phase tracking AFSR.

% file: trackingtest.m
% test of tracking afsr
\% \% \text{input} = \text{-1+2*lsr}(0, 1, 3000); \% get lsr samples
\text{epsilon} = 0.5;
\text{delay1} = 300; \% distance to first receiver
\text{delay2} = 370; \% distance to second receiver
\text{gain} = 4.0;
\text{noisevar} = 0.0;

\text{input} = \text{input} + \text{noisevar} \times \text{randn}(1, 3000);

[\text{out1, phi1}] = \text{run\_phi\_afsr}(\text{input}, \text{delay1}, \text{epsilon}, \text{gain}, \text{noisevar});
[\text{out2, phi2}] = \text{run\_phi\_afsr}(\text{input}, \text{delay2}, \text{epsilon}, \text{gain}, \text{noisevar});

\text{figure}(1);
\text{clf};
\text{subplot}(2,2,1);
\text{plot}(\text{phi1}(1:1000));
\text{subplot}(2,2,2);
\text{plot}(\text{phi2}(1:1000));
\text{subplot}(2,2,3);
\text{plot}(\text{phi1}(2001:2250) - \text{phi2}(2001:2250));

% file: run\_phi\_afsr.m
function \( [S, \phi] = \text{run\_phi\_afsr}(\text{input}, \text{delay}, \text{epsilon}, \text{gain}, \text{noisevar}) \)
\% \% runs an \text{afsr} on the input sequence lagged by delay samples.
\% \text{noisevar} controls the variance of the noise stuffed in during
\% \text{the delay}
\% \text{epsilon} is the fraction of rcd signal stuffed in
\% \text{gain} is gain on the “sum” operation
\%
% runs an integrator (starting with a random startup value)
% which resets to V0 whenever afsr = 111..1

% afsr_len = 7;
tap1 = 7;
tap2 = 1;

% init the afsr
afsr = randn(1,afsr_len+1);
faze = 5*abs(randn(1,1));
dfazedt = 0.01;
V0 = 0.7;

% lag the input, put in random noise for now
dlycrap = noisevar*randn(1, delay);
rcv = [dlycrap, input];
rcv = rcv(1:length(input));  % clip to same length as input

S = zeros(1,length(input));
phi = zeros(1,length(input));

for i = 1:length(input),
    ok = 1;
    for j = 1:afsr_len,
        if (afsr(j) < 0.995),
            ok=0;
            break;
        end
    end
    if(ok==1)
        faze = V0;
    end
    afsr(afsr_len+1) = epsilon*rcv(i) + (1-epsilon) * afsr_step(tap1, tap2, afsr, gain);
    faze = faze + dfazedt;
    S(i) = afsr(afsr_len+1);
    phi(i) = faze;
    for j=1:afsr_len,
        afsr(j) = afsr(j+1);
    end
end

% file: afsr_step.m
function x = afsr_step(tap1, tap2, asr, gain)
% x = afsr_step(tap1, tap2, asr)
% computes the next value to be stuffed into an afsr
% in the -1/+1 basis using Ben's rule
% new = -(asr[tap1]*asr[tap2])/abs(asr[t1]*asr[t2])
% min(abs(asr[t1]*asr[t2]),1)
% mag = abs(asr(tap1)*asr(tap2));
% if(mag < 1e-30)
%    x = 0;
% else
%    sgn = asr(tap1)*asr(tap2)/mag;
%    x = -sgn*min(gain*mag, 1);
% end
Bibliography


