Study of Feasability for Phase Difference Extraction Using Software Defined Radio in Location Analysis

Paul Miller

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons, and the Remote Sensing Commons

Recommended Citation
https://scholarworks.wmich.edu/masters_theses/176
STUDY OF FEASIBILITY FOR PHASE DIFFERENCE EXTRACTION USING SOFTWARE DEFINED RADIO IN LOCATION ANALYSIS

by

Paul Miller

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science
Computer Science
Western Michigan University
June 2013

Thesis Committee:

Ala Al-Fuqaha, Ph.D., Chair
Dionysisos Kountanis, Ph.D.
Moussa Ayyash, Ph.D.
Here is a method for using the phase of an HF radio signal for use in location analysis. This computation is implemented in a software defined radio processing block in the GNU Radio environment. The signals analyzed are received by an Ettus Research USRP SDR. We created a phase analysis system called WMU Rootsync, which compares the roots of a received signal to the roots of a generated reference signal for the phase analysis. This research describes a prototype method for phase analysis only. Future projects may use these ideas differently than presented here. We intend for future projects to use this phase information to measure distance for use in location analysis in turn used to help persons navigate in areas where GPS or other alternatives may not function or may otherwise be impractical.
TABLE OF CONTENTS

LIST OF FIGURES .............................................. v

CHAPTER

I. OVERVIEW ................................................. 1

II. LITERATURE REVIEW ................................. 4
    GPS ....................................................... 4
    Cricket ............................................... 5
    RSS and ToF .......................................... 8
    Near-Field EM Ranging ............................. 9
    Broadcast AM Radio ............................... 11
    Software Defined Radio ......................... 13
        GNU Radio ....................................... 15
        Ettus: USRP ................................. 18

III. ROOTSYNC PHASE COMPUTATION .............. 20
    Simulator and Basic Models ................. 20
        Debuging and Output Log Files ........... 23
        Carrier Signal Extraction ............... 26
        Rootsync ....................................... 32
        Extract Signal .............................. 37
        EMEdf Extract .............................. 38
Table Of Contents–Continued

CHAPTER

Simulation Testing Results .................. 40
Non-Simulated Testing ....................... 40
Least Square Fit ......................... 45
Relevant Root Notched Filter ............... 47
Laguerre Frequency Filter .................. 51
Audio Domain Tests ....................... 56
WMU Radio Station ....................... 59

IV. CONCLUSIONS AND FUTURE IMPROVEMENTS 63

REFERENCES .................................. 68

APPENDIX .................................. 69
2. Correct positioning of beacons ......................... 8
3. AM signal 833 Hz on a 10 kHz carrier, continuous ... 14
4. AM signal 833 Hz on a 10 kHz carrier, 250 kbps ...... 15
5. AM Modulator ........................................... 22
6. Two Signals .............................................. 23
7. Example theta graph using candlesticks ............... 26
8. Phase locked loop carrier extraction ................... 28
9. Basic AM demodulation configuration ................... 30
10. On average, the roots build up at the carrier frequency 31
11. The root is somewhere between the two root samples. 34
12. Extract Signal .......................................... 37
13. Extract Multiply Extract Theta Out ..................... 39
14. WMU Mobile Science Station ............................ 41
15. 9.6 samples per root .................................... 42
16. Phase roll output of Rootsync on live radio $\theta_1$ and $\theta_2$ 43
17. 8 kHz signal with mixed signal at 96 kbps .............. 50
18. Unwrapped $\theta$ roll from USRP over 80 seconds ....... 52
List Of Figures–Continued

19. Laguerre Filter Hardware Schematic .................. 53
20. The GNU Radio model for the phone sync tests ...... 58
21. Audio domain 2.7kHz run ............................ 59
22. WMU 1,000,000 Hz Radio ............................. 60
23. The Final USRP Sync digram ......................... 61
24. WMU 1,000,000.00 Hz Radio sync run ............... 62
25. Protocol for location by phase analysis ............... 66
CHAPTER I
OVERVIEW

Localization is the science of determining the location of objects relative to one another such as: Where is this device in relation to this building? Where is this device on the face of the earth? Everyone is familiar with GPS, now the paradigmatic example of localization. GPS gives operators another answer for free (which is not at all a requirement of localization in general): Where is this device in time?

In the first chapter we cover localization as it appears in the literature and as it relates to this thesis. The first chapter covers the Global Positioning System, a global positioning network used for navigation and entertainment alike. Next, we describe the MIT Cricket System. It is used for localization inside structures (GPS is not designed to do this).

The system we have built tries to utilize wavelengths in the Broadcast AM Radio range, so naturally we describe AM in detail, including a brief description of the transmitter we built to use in our experiment phase.

Lastly, the review chapter talks about Software Defined Radio. In this section we discuss SDR in general and also the hardware and software choices we made. We choose GNU Radio for its completeness as a digital signal processing package and we choose the Ettus USRP as the SDR for its
sensitivity, sample rate, and especially for its ease of use with GNU Radio.

Following the literature review we talk about the software and models we have developed and empirical results we have recorded. The method we invented and discuss there is called Rootsync; so-called because it seeks to measure phase differences by synchronizing a generated signal to a received signal using the roots of the signals as references. (A root of a signal is a position in time where the signal has a 0 amplitude.)

We have built a simulator for our early work (which we discuss), followed by the basic blocks we use and the problems we encountered using them. We then talk about how we overcame the problems with the basic blocks using statistical (actually stock market time difference) techniques – namely, least squares fit linear regression analysis and then Laguerre filters as described by Ehler in [2].

After the description of the simulator, we provide a narrative of negative results, what we tried, and why each idea did or did not work. We first discuss our empirical results in the radio domain, which did not go well. After this, we tested in the sound domain using audio and a PC sound card rather than radio, which did turn out very well. Then finally we tested again in the radio domain again using a radio of our own design as the signal source where we achieve a positive result. We have shown that the methods we developed do function for phase extraction. Further
work will be required to construct a usable system, but phase difference extraction is possible.
CHAPTER II
LITERATURE REVIEW

GPS

The Global Positioning System (GPS) Standard Positioning Service (SPS) is a space based location system provided for free (not counting taxes) by the US government.

GPS today consists of 24 geometric slots in the sky. Each slot has at least one satellite which beams a spread spectrum signal at 1575.42 MHz (civilian use channel) and 1227.6 MHz (encrypted military and space exploration channel). The signal contains navigation and clock data encoded via BPSK (binary phase shift keying) modulation. The data in the signal contains the satellite location and upload time. If the upload time is older than about a day, then the satellite accuracy may fall below the expected 95% accuracy – called normal operation. [8] provides a handy logarithmic chart of the accuracy decay in the appendices. It seems the satellites will often operate without uploads for longer than 14 days (the maximum in the chart) – which puts the error for that satellite above 400 m.

Receivers compute their position based on the satellite position(s) and the delay (due to the speed of light) of the received signal. On the civilian channels, the expected accuracy will never be better than about 20 m, by
design. Using various tricks and guesses based on known locations (the
locations of roads and cellular antennae, etc.) GPS software can often
reduce the 20 m error quite a bit. Thanks to GPS, outdoor navigation is
largely a solved problem.

Since GPS broadcasts at very low level (−130 dBm under clear skies
and as low as −150 dBm indoors), it often does not function usefully inside
buildings and structures. The accuracy is also normally too low to give
the exact position of persons inside structures. It can give users a rough
idea of their location, but imagine a situation where a device should give
an instruction such as, “the door to your destination is a few feet to your
left” and GPS likely cannot solve the problem.

Cricket

What if we have an interest in finding our way inside a large unfamiliar
structure? This has special implications for visually impaired persons for
whom the usual signage (one need only consider room numbers and floor
numbers) is likely less helpful than it is for people that can easily read
the signage. This concern may also apply to persons unable to read the
language of the signs for any other reasons (e.g. travelers in a foreign
country).

One of the most well developed solutions for indoor localization is the
MIT Cricket system. It employs a network of beacons arranged throughout a structure, which transmit radio and ultrasound signals that listener devices use to locate themselves.

The beacons are very inexpensive PIC microncontrollers with SAW radio transmitters producing RF at 418 MHz and ultrasonic transmitters operating at $40kHz$. Operators must arrange a large number of these beacons around structures since the operational range is roughly 30 ft and ultrasound cannot travel through obstacles. The very limited range of the beacons is intentionally built into the design. Obstacles distort the signals and confuse the receivers in this system. The shorter range helps to guarantee the receiver is working with line-of-sight beacons.

![Figure 1: RF-A:US-I vs RF-A:US-A](image)

Isolating the correct beacon is achieved by transmitting a code at a
very slow rate over the RF signal. Figure 1 shows a timing diagram of this process. In the diagram, the listener receives both the RF-A (radio frequency signal A) and the RF-I (interfering signal). Because US-I (ultra sound interfering signal) is received before the RF-A code is complete, the system can rule out RF-I as a valid code while reading RF-A and US-A.

In [9] (to which this entire section refers), we find detailed beacon position diagrams and strategies. The system is complicated to set up. One must account for various room configurations and beacon positions, making sure the beacons are at the correct distance and cover all parts of various architectures. Figure 2 shows the correct positioning of beacons as described in [9], showing example room with various partitions. The paper covers many details of correct positioning, which seems to involve some trial and error. The basic idea is to make sure there is always direct line of sight to at least a couple beacons.

When everything is set up just right, an accurate read from two beacons can give an accurate position in a building. Readings are sent in to a so-called Floorplan server, which can display an active map to the end user. The Floorplan server must be preprogrammed with the exact location of the beacons so the beacon distance information can be used to describe a location in the building (on the floor plan).
Another commonly used method for indoor localization involves using the received signal strength of a signal. This technique normally requires line of sight or a signal that can pass through walls. It is sensitive and considered somewhat inaccurate. The power of a signal can fluctuate due to interference or even temperature changes. This is described in [6].

The above paper also discusses time-of-flight (ToF) readings. Using ToF, the systems estimate distance and positions using wave propagation speed as a constant and measuring the delay to the receiver. Some systems (like those at Nanotron GmbH) offer products that do this bidirectionally (locator tags are transceivers and offer a signal to location servers as well as the other way around). Nanotron, specifically, calls this SDS-ToF (symmetric double sided two way ranging). ToF also requires line of sight, since
multi-path reception will clearly confuse the timing measurements.

In [6] the authors describe a technique for combining RSS with ToF for the purpose of making a measurement better than the sum of the parts. Their technique relies on statistical probabilities. Basically, they divide the room into tiles and use RSS to tiles probably containing the localizer elements. They also choose tiles using ToF and consider which tile has the best probability of containing the localizers. In their experiments, this technique was shown to be much more accurate than either RSS or ToF alone.

Near-Field EM Ranging

There exists a patent filing [7] that lists the details of using orthogonal signals in near field to locate tags relative to beacons – products from this patent (called NFER RTLS) are produced by a company named Q-Track. They describe a system with beacons that propagate AM wavelength RF. The system uses differences between the magnetic field propagation and the electrical field propagation to estimate distance between the beacon and the locator tags. Technically, the patent filing talks about any two characteristics, but they specifically mention the electric field (E) and the magnetic field (H) most of the time and rarely mention any other characteristics.
Further, the specific difference they use is the phase difference $\lambda$. The phase difference they are measuring is between these two parts of the signal rather than between two different transmitters or after some timing delay. Much of the patent discusses the various antenna configurations they use to exaggerate or detect the differences. For instance, they use a loop antenna and a dipole antenna at the transmitter – the dipole helps to broadcast the E field and the loop antenna works with the H field. As the range increases, the phase delta decreases which is shown in their equation 1. The phase delta is used to compute the distance in their products, so they sometimes refer to the range in degrees of $\lambda$.

$$\Delta \phi = \phi_H - \phi_E = \frac{180}{\pi} \left( \cot^{-1} \left( \frac{\omega_r}{c} - \frac{c}{\omega_r} \right) - \cot^{-1} \left( \frac{\omega_r}{c} \right) \right) \quad (1)$$

They also carefully explain that at far field distances (generally one wavelength away from the transmitter), the magnetic field and electrical field become phase synchronous. This gives a maximum of 3000 ft range for low wavelength transmitter frequencies. However, they also say the range in AM frequencies (where they actually use this in empirical testing and products) is around 900 ft. Their testing showed they can achieve better than 0.001$\lambda$ accuracy.
Broadcast AM Radio

Amplitude modulated radio is the oldest form of audio broadcast radio ever used and it is still widely used today. The modulation is very simple – the input audio is mixed with a carrier wave of the desired broadcast frequency. The modulated signal is limited to roughly 10 kHz of broadcast bandwidth. Frequencies are divided into three wavelengths, called: long wave, short wave, and medium wave.

The operating range of “medium wave” broadcast frequencies are designated as roughly 500 kHz through 1600 kHz. So-called “short wave” signals are designated to go as high as 26 MHz and “long wave” goes as low as 150 kHz though the 500 – 1600 kHz range is the range of typical off the shelf AM radio receivers.

The long waves are intended to bounce off the ionosphere (and are therefore also called skywaves), while the shortwave broadcasts mainly travel along the ground and are therefore called groundwaves. Medium waves can bounce off the ionosphere better during the night, and because of this, many medium wave broadcast licenses require operators to reduce power after sunset. If they did not reduce power, their range might increase dramatically and they might interfere with other long wave signals (otherwise too far away during daylight) in unintended ways [10].
The frequencies we are working with in this project are in the medium wave ranges. These frequencies are licensed, of course, but the FCC allows amateur and even unlicensed commercial broadcasts in these frequency ranges as long as certain conditions are met [1]. Just a few of the rules include: the power must be less than 100 mW, the antenna (including the radio) must be less than 10 ft long, and that one must never claim a callsign (e.g., WKFR, KAZO). Lastly, any interference with any licensed broadcast must result in the immediate termination of the unlicensed broadcast.

AM radio waves have several useful features that interest us. First, the frequency bands used by broadcast AM easily travel through walls and buildings. By contract, GPS is fairly useless inside buildings and the frequencies used by wifi (2.4 GHz range) tend to bounce off walls and structures causing multipath fading and other attenuation. Likewise, the modulation leaves the carrier frequency fairly unadulterated. Again, by contrast, other signal types alter the portions of the signal in which we are interested in. Frequency modulation alters the frequency (as the name suggests) and phase shift keying alters the phase – wifi employs a mixture of both known as QAM, which we shall not cover here [11].

Our ultimate goal in this project was to employ the use of phase detection on AM radio signals. Our current experiments are focused on carriers we generate locally (at the 100 mW level) rather than ambient commercial
broadcasts because of problems we encountered with signal accuracy and multi-path reception – which is covered later in this paper.

The advantages of this approach, as already suggested, are that with only three transmitters one could locate a receiver anywhere inside a building or complex of buildings.

Software Defined Radio

This project is a Software Defined Radio (SDR) system. Generally speaking, SDR is digital signal processing done in software. Traditionally hardware-only signal manipulations (mixing, multiplying, amplifying) are computed on a general purpose processor after being converted to samples rather than being implemented in an analog fashion as hardware components.

A typical SDR receiver would contain a radio and enough equipment to translate the radio signal into a string of samples to feed to a computer (or embedded processor). This is normally accomplished by setting up a clock to regularly take a snapshot (a sample) of the wave amplitude. The samples are simply evenly spaced (temporally speaking) amplitude measurements. Two diagrams follow, one shows a modulated signal continuously and the other shows the samples taken (generated in this case) at 250 ksp (kilo samples per second).
Figure 3: AM signal 833 Hz on a 10 kHz carrier, continuous

The second diagram shows how difficult it can be to determine the location of the roots (where the signal has an amplitude of 0) – which are values of great concern to us later on in this paper. The minimum sample rate which has the resolution available to analyze a signal is called the Nyquist sample rate. Nyquist is double the frequency one wishes to analyze. The paradigmatic example of why this sample rate is required is a spinning car tire. If the sample rate is at exactly the frequency of rotation, one could not determine if the tire was spinning at all [10]. There is no practical maximum. As a rule, the more samples there are, the more accurate the analysis can be. In our work, we try to use the highest sample rate possible without
Figure 4: AM signal 833 Hz on a 10 kHz carrier, 250 ksp

cauing buffer overruns in our SDR device driver.

GNU Radio

GNU Radio is a suite of digital signal processing library functions, sample processing blocks, and graphical user elements. It is designed to implement software defined radio models simply and extensibly. A standard default install has most of the functionality needed to complete tasks traditionally left to hardware: filtering, amplifying, signal analysis (via FFT), resampling, buffering, synchronizing, modulating/demodulating, etc.

Processing functions are divided into blocks. A block takes a number of inputs (usually samples or scalars), and outputs the same. Source blocks
generate samples (or other values) by reading them from devices (USRP, sound card, etc.) or files. Sink blocks dump samples to radios, devices, and files. So the three input/output modalities are: source, stream, and sink. There are four other modalities to consider, and they refer to the way the I/O synchronizes. The simplest type is the sync block. Sync blocks output the same number of samples that they input. Decimating blocks consume more inputs than they output. Interpolating blocks output more than they consume. Finally, General work blocks consume and emit samples as desired within the block. The I/O rate decisions for general work blocks are entirely software based and can change while the program is running.

Most things can be built in the GNU Radio Companion, a user interface designed for connecting the blocks together. When a program designed graphically in the GRC is run, the GRC generates the appropriate combination of python and C++ code and saves it to files. It then compiles them, and then runs the program in an optional attached GUI. A typical example might read from a device, like a USRP radio (see section III.2 for more details). Builders would connect the sample source to a suite of filters, for example: an automatic gain control, an xlate filter (made with FIRs) that translates the signal down toward baseband, and a low pass filter that cuts off samples at higher frequencies than desired. Here, the builder might
attach an oscilloscope or an FFT graphical element in order to “see” the signal – e.g., an input AM radio signal. The builder could then apply a magnitude function to get the amplitude, resample to the rate of the sound card and sink to the audio device. In this example, the user could then trivially listen to the radio broadcast to which the USRP is tuned. All of these things are built into GNU Radio and an experienced operator could get this AM radio example working in just a couple minutes.

As explained above, GNU Radio uses more than one programming language. Most of the processing blocks are written in C++. The C++ code uses the popular Boost libraries extensively (one might consider Boost another language). The processing blocks are “glued” together using the Python programming language. Lastly, the build system uses a language called Cmake. Cmake is not really a programming language in the traditional sense, but learning Cmake is a must in order to construct custom blocks. GNU Radio can also compile FPGA embedded blobs that can be injected into radio controllers and, a feature that we do not use in this project yet, but would be useful in the future.

We use this project exclusively for our research because it can do everything we need, including building custom blocks that implement our models for this research. In fact, the primary contribution of this research is a block called Rootsync, which we describe in detail (including pseudo
Ettus: USRP

The Ettus USRP2 is an SDR that is intended for use with GNU Radio. Ettus maintains a suite of open source software drivers for using the USRP with GNU Radio. The driver is called UHD, which stands for USRP Hardware Driver, and which manifests in the GNU Radio Companion as a pair of source/sink blocks with settings used to set the tuner in the radio.

The USRP pushes a stream of sample data to GNU Radio through the driver source block. It can also accept a stream of samples and mix them up to a desired frequency for output. The device can produce (and collect) streams with sample rates as high as 50 Msps – which it delivers to host applications over a Gigabit Ethernet port.

Users must select a target frequency range (e.g., 2.3 – 2.9 GHz, 2.4 – 2.5 GHz, 4.9 – 5.9 GHz, 400 – 4400 MHz, etc.) and install a daughter-board of the desired frequency range. In this way, the USRP can be made to operate anywhere from DC to 6 GHz for both input and output.

The board we are using is the LFRX Daughterboard (which can tune DC – 30 MHz). Since the maximum sample rate of the USRP is 50 Msps, with this board it should be possible to listen to all of the broadcast AM Radio channels at the same time. In practice, this is difficult to do because
most AM antennae have tuners in them that act as a bandpass filter, blocking all but one channel. This makes sense for normal usage – why would anyone ever listen to two at the same time? In SDR however, one useful application is recording huge swaths of spectrum to files to sift through later or at a more convenient location. The USRP normally excels at this. In shorter wavelengths (e.g., ISM 2.4 GHz), recording the entire band to a file would indeed be possible – mainly due to antenna possibilities.

The USRP is full duplex and can transmit with the same bandwidth (50 Msp/s) by which it receives. It is supposedly possible to run a complete cellular LTE basestation through a single PC and a USRP. We have never tried to do this, but it seems reasonable that it could be done [3].

The usage of the USRP we describe in this paper is modest and does not begin to use a fraction of the USRP capabilities. However, the device does suit our purposes perfectly and provides the accuracy and reliability we need to conduct our experiments.
CHAPTER III
ROOTSYNC PHASE COMPUTATION

Simulator and Basic Models

We discussed possible approaches before we really began constructing experiments and models. It became clear that we needed a realistic simulation in which we could test without having to construct a physical apparatus. We intended all along to conduct tests on real radio hardware, but before we could pick which ideas to test in real life, we needed a method to evaluate the various techniques we were discussing.

Creation of realistic radio signals is a simple process, although there are considerations that can cause simulated results to diverge from results of analyzing real signals. Our simulation is built such that external audio is expected to be fed in. We choose a songs with a variety of frequency ranges and rhythms to make sure our signals were as realistic as possible. We initially made no attempt to simulate multiple path fading, interference and frequency drift.

First we bring the audio into our flow graph from a wav file. Any audio source would work here, but again, noisy music with a variety of frequency ranges was chosen to make the signal as realistic as possible. We did not (for example) wish to use Gaussian noise since it does not in any way resemble
the content of a real radio signal – after all, most radio stations broadcast audio intended for humans, not carrier waves or carrier waves mixed with Gaussian noise. The frequencies from the audio are set as the real part of a complex number, with the complex component set to zero. The resulting complex signal is then fed through a low pass filter to attenuate frequencies beyond the desired bandwidth. If we apply an LPF with a cutoff at 10 kHz, then the audio between 0 and 10 kHz will be allowed to “pass” while everything over 10 kHz is attenuated. Once the signal is mixed with a carrier, this will result in a channel occupied bandwidth of 10 kHz. Most broadcast medium wave AM has roughly 10 kHz of bandwidth [10], though we only use 1 kHz from the center of this in most of our models. The signal is then multiplied by a sine wave at the frequency of the signal we wish to generate (the carrier). This is the usual way to mix an input signal to the desired frequency – the heterodyning (or mixing) equation is shown in equation 4 later in the paper. The GNU Radio Companion (GRC) block that we designed for the project can optionally (by setting) output the modulated signal or the carrier wave instead of the modulated signal. We call this block the AM Modulator. A simplified version of the block is shown in figure 5 (the mixer is not pictured).

Most of our early models require receiving two (or more) signals and
comparing or multiplying the two signals together. Because of this, our simulator is able to simulate more than one signal at a time.

Generation of multi-signal output is the job of another block called *Two Signals*. This block feeds the output of two *AM Modulators* multiplied by an arbitrary complex rotation into a three-way adder along with some Gaussian noise (meant to look like a noise floor). The complex rotations can be changed while the program is running so users can simulate movement and observe while the system reacts to the phase changes.

The phase shift arguments to the block are in radians. The constant multiplier computes the rotation as \( \text{cmath.exp}(\theta \cdot 1j) \) (Python code for \( e^{\theta i} \)). In practice, when we build user interfaces in our simulator suite we employ slider for one or both of the phase settings. We use the sliders to simulate phase changes we would like to be able to detect.
Debugging and Output Log Files

The usual way to guess or infer that an experimental run of a GNU Radio program is functioning as intended is to try to peak into the operation of the blocks using various graphical elements built into the GNU Radio suite. In particular, one tries to use the scope output block (as in oscilloscope) to show signals, or some other scalar over time. This can be used to spot patterns and cycles (e.g., see figure 10), but it is imprecise. It is imprecise in the sense that one can only see the small slice of time plotted in the scope element. If one wondered about patterns over the course of millions of samples, the scope element would not be able to effectively show the pattern.

Initially, for debugging purpose, we installed many debugging log outputs. They reported on the most trivial aspects and decisions made by our
various models and blocks, but parsing the output for needed information was a distracting side project at best. At worst, it was consuming perhaps a third of development time. Another problem with this verbose debugging output was the size. A sample on disk is several bytes, but the debugging logs for a single sample could be on the order of 300 bytes – the space required for human readable output is surprisingly large. For these reasons, we developed a compact log format intended for parsing by machines. We present the details of the log format here and discuss some of the ways we condense the output in graphical format.

At every time slice, we print a line to the logfile. Each line represents a time slice and the units we use for time are samples; so that the impulse response of every sample can be tracked in detail in the log. Lines end with a 0x0a (\n) byte and consist of mainly numerical data in somewhat human readable format. Lines can also contain strings and booleans (stored as the word ‘true’ or the word ‘false’), but we intend this to be used mainly for annotations rather than information used in graphs and output.

Each datum of interest is stored as a tag-pair. A tag is a string, but rather than storing the string along with every datum, we shrink the tag to an integer. As each new tag presents while the program is running, we store a string/tag map header inline with the data. The Header format is stored similarly to a datum, though it is prefixed and postfixed with square
brackets and can contain a space. Each tag/pair is separated from the next by a 0x07 (\b) character. The tag/datum and tag-name/tag-number pairs are separated by a colon character.

Typical (although highly truncated) lines look something like this:

```
[time: 0]\b0:1\b[spr: 1]\b1:1.0000000\n0:2\b1:1.0000000\n0:3\b1:1.0000000\n```

For many months we were chasing a bug in our system where the frequency of the signal received from a radio station seemed to drift around. There is a degree of variability in this type of analysis, so analyzing the impulse response rarely tells the whole and complete story. It was felt that there was a general trend in the data and the local drifts (which were expected) were obfuscating the overall trend. To show the trend in a useful way, we turned to stock market data visualization techniques.

Stock market price data is sometimes studied as a 4-tuple, [open, high, low, close]. Technical analysts (as they are called) plot this data in various complicated ways and of particular interest to us is an informative time-value compression called candlestick plots. Each point shows the 4-tuple in one of two ways, a colored candlestick (which is confusing to read until one is acclimatized) or as a vertical stick with a tick on the left and a tick on the right. We use the clearer (though less common) candle stick for various plots. The results look like those in figure 7. The left and right ticks
indicate the starting and ending values for the stick and the vertical length of the stick shows the minimum and maximum for the data compressed into there.

Figure 7: Example theta graph using candlesticks

Carrier Signal Extraction

We have talked about using the phase of a signal to detect movement and position. If we are using the phase of a “signal” then clearly we are concerned only with the carrier itself. The data portion of the signal does not contain the information we desire – namely, the phase. Mathematically, we are looking for the $\arg(z)$ function of the complex sinusoid that carries
the signal.

The problem is that an ambient commercial AM radio signal is modulated. It is a very simple modulation, to be sure, but it is not a simple process to demodulate it and extract a carrier clean enough for what we wish to do with it. There are two possible answers when one solves for the phase, and the amplitude modulation causes the waveform to oscillate between them. The standard go-to answer for extracting the carrier, according to engineering textbooks, is a phase locked loop and notch filters [4]. The PLL model we show in figure 8 also contains a normalizer we call “Py Normalize,” which is a simple normalizer (written in python) that ensures every sample has a magnitude of 1 on the imaginary unit circle \((new = orig / |\sqrt{orig_{real}^2 + orig_{imag}^2}|)\). The normalizer does seem to help, but not enough to make this approach work.

Our problem is that the amplitude modulation flips and flops the \(\arg(z)\) function at the frequencies used by the data signal (10 kHz of frequencies in the usual case). This means the PLL has two phases to lock into and – typically it never picks one over the other. This actually makes perfect sense. One cannot lock onto the phase when there are two. Strictly speaking, this approach can probably be made to work (particularly in hardware), but we found the computational cost of the notched filtering was far too
high to continue with this approach. In SDR, a notched filter consumes an enormous amount of CPU time. In GNU Radio (at least) nearly all the frequency filters are built using some variation of FIR, and a narrow band-pass FIR filter requires a large number of taps – taps are the terms of the summation used to compute the FIR result. The narrower the passband, the more taps required. A notched filter requires so many taps the program cannot keep up with the USRP on even the fastest computers. Once the program falls behind, the USRP driver reports buffer overruns and begins dropping data. We found that the use of very narrow filters was impractical due to these CPU constraints. It may still be possible to extract the
carrier this way (in hardware at least), but we focused our efforts in areas that require practical amounts of CPU time.

Another option we considered and tried without any success is to simply demodulate the signal. We tried feeding the extracted amplitude modulations through a divider (a GNU Radio divider block) and this did little to help. We tried normalizing the signal, by this we mean forcing all samples onto the unit circle ($\bar{z} = 1$), and subtracting the amplitude modulations. We tried some automatic gain control blocks, forcing the signal into a square wave and then using notched filters. Nothing we tried seemed to get us any closer. Either the signal remained modulated or the resulting carrier was too distorted and/or erratic to use for phase analysis.

The real problem with demodulation is the lack of access to the original data signal and the original carrier signal. Demodulation without access to both is very challenging. It may be possible (indeed it seems it should be simple). Demodulation through division by a guess (like the magnitude of samples) tends to distort the carrier, or fails to demodulate completely.

Regardless of the particulars of the modulation and input data, a carrier multiplied by a data signal still multiplies to zero when one or both signals have an amplitude of 0. It is very simple to see where roots are when the signal is plotted and it is just as simple to compute them in a
software defined filter. GNU Radio has a visualization block that draws an oscilloscope-like output as a graphical element. It can be made to persist old values through a settings panel on the graphical element. Turning this on while looking at the modulated signal reveals the roots as small fuzzy circles or asterisks along the zero-amplitude line. These “root shadows” (as we call them) tend to be at very regular spacing. Figure 10 shows root shadows next to a reference sign wave at the same frequency of a received signal.
modulated signal. Mathematically speaking, there should be two roots per carrier cycle; however, there should also be more roots relating to the data signal(s). Notice that the root shadows shown in figure 10 are at a very clean spacing. While the roots produced by the data signal move (so to speak) relative to the carrier. The carrier roots, in contrast, always appear with that same regular spacing – most of the time. We encountered many problems using these roots, which is covered in section III.2.

Figure 10: On average, the roots build up at the carrier frequency

Exploiting the obviousness of the roots allows us to sidestep the problem of demodulation. We generate a reference carrier at the frequency of the input signal and align it to the source signal as best we can by trying to match up the roots of the signal and the reference. In figure 10 we
can see that the roots of the signal are the same fraction of $\pi$ away from the reference on each cycle. If we could simply move the $ref$ signal over a little, we would know the phase of the real carrier – well, one of two possible answers anyway.

Rootsync

Moving the reference over to match the signal is the technique (and block and process) we call WMU Rootsync. Rootsync allows us to extract the carrier or phase of a signal whether it is modulated or not. Basically, we generate a signal at the frequency of the input signal and align it to match, based on the roots of the reference and the input signal. This method works very well in the simulator, but turned out to be problematic when using real radios (covered in detail in section III.2).

The Rootsync block is written in C++ using the Boost C++ libraries – one of the two the standard languages of GNU Radio. The build process outputs a hodgepodge mixture of Python and dynamic library objects which are loaded at runtime by the code generated in the GRC. It is written as a general block (the four I/O modalities are described in section II.6.1), which allows it to consume variable numbers of input samples without producing output responses. This behavior can be utilized during a training phase, during which we may desire to not output any results. Most of the
time, the block operates as a sync block, meaning it outputs one sample for every input sample.

The Rootsync module aligns a reference signal to the input signal and then outputs either the rotations needed to align the reference ($\theta$) or outputs the rotated reference signal directly ($ref^*$). The alignment is computed by adjusting $\theta$ and multiplying the generated carrier to roughly the same phase as the source signal. We call the three signals, $sig$, $ref$ and $ref^*$. $ref^*$ is computed as $ref^* = ref \cdot e^{\theta i}$. There are, two ways to align the carrier (in phase and out of phase) and Rootsync makes no attempt to choose between them. With a modulated signal, the phase oscillates between the two answers and choosing one over the other is fairly meaningless. With unmodulated signals it is possible to select the correct phase, but our goal was to use ambient modulated signals, so we never considered choosing one phase over the other. When Rootsync outputs $\theta$, it outputs the two possible values of $\theta$ agnostically as the real and imaginary parts of a complex output signal.

For each sample, the main loop (the GNU Radio API calls this `general_work`) advances the reference signal: $ref$ (generated internally), computes $ref^* = ref \cdot e^{\theta i}$, and increments the samples-since-root variable (called `sroot[x]`). `general_work` then looks for roots using a function called `compute_sroot`. When roots are found, they rarely occur at a signal level zero (i.e., the
samples before and after the root are non-zero), so the root is computed
trigonometrically, and $s\text{root}[x]$ is set to the interpolated value of the root.

This interpolation is computed as shown in equation 3 and is based on the
digram shown figure 11. Of course, the more samples we have, the less
interpolation is necessary. Still, we found that the sample rate required to
obviate interpolation would be prohibitive.

$$t_{\text{root}} = h_1 \cdot \tan(\tan(t_{\text{last}} - t_{\text{now}}, h_1 + h_2))$$  \quad (2)$$

$$= h_1/(h_1 + h_2)$$  \quad (3)$$

Figure 11: The root is somewhere between the two root samples.
Next, \texttt{general\_work} calls a function named \texttt{compute\_difference}. This function computes two values, \texttt{droot} and \texttt{drad}. \texttt{droot} is simply the difference, in samples, between \texttt{sroot[idx]} and \texttt{sroot[(idx+1)\%2]}; in other words, the time difference between roots for the two signals. We wish for this to be as close to zero as possible (the alignment we call Root-sync) and we decrease \texttt{droot} by adjusting \texttt{\theta}, which in turn affects \texttt{ref^*}. To do this, we next compute \texttt{drad=\textit{wrap\_phase\_adj( (M\_PI / spr) * droot )}} the actual difference in radians, and then \texttt{drad = drad*(1-alpha) + alpha*\_drad}, the average difference in radians.

If \texttt{fabs(aadj = gamma*drad) > beta} we adjust \texttt{\theta} by \texttt{aadj} and wait for the next sample. This all works very well in the simulator and \texttt{ref^*} quickly (dozens or hundreds of samples, depending on the sample rate and the number of samples per root in particular) lines up \texttt{ref^*} to be either very closely in phase or very closely \texttt{\pi} out of phase with the input signal. Recall that since the input signal is modulated, \texttt{ref} is really both in-phase and anti-phase at the same time.

The pseudocode for for key functions in Rootsync follows. Note that this pseudocode shows a function called \texttt{root\_relevant} which was not actually used in initial testing and that is described below under in section III.2.2.
WMURootsync::set_freq(f):
    freq = f
    spr = (sps/freq)/2.0
    drad = roots_since_reset[0] = roots_since_reset[1] = 0
    ref_gen.set_freq(2*M_PI * freq / sps)

    freq_wf = freq * freq_win
    min_spr = sps / (2 * (freq+freq_wf))
    max_spr = sps / (2 * (freq-freq_wf))

WMURootsync::set_theta(t):
    drad = aadj = roots_since_reset[0]
         = roots_since_reset[1] = 0
    theta = wrap_phase(t)
    phase_shift = cexp( theta * I )

WMURootsync::compute_sroot(idx, now):
    if last[idx]:
        sroot[idx] +=

        if (now>0 && last[idx]<=0) || (now<0 && last[idx]>=0):
            roots_since_reset[idx] ++
            rooted = true
        osr = sroot[idx]
        if fabs(now) == 0
            sroot[idx] = 0
        else:
            h1 = fabs(last[idx])
            h1_h2 = h1 + fabs(now)

            if h1_h2 == 0:
                sroot[idx] = 0
                throw_runtime_error( "h1_h2 is 0" )
            else:
                sroot[idx] = 1 - ( h1 / h1_h2 )

        if idx == 0:
            if (rooted = root_relevant()):
                relevant_roots_since_time_zero ++
                roots_since_time_zero ++
        last[idx] = now
    return rooted

WMURootsync::compute_difference(idx, dir):
    if roots_since_reset[0] > 1 && roots_since_reset[1] > 1:
        droot = sroot[idx + dir] - sroot[idx]
        _drad = wrap_phase_adj( dir * (M_PI / spr) * droot )
        drad = drad*(1-alpha) + alpha*_drad

36
WMURootsync::general_work(sample):
    ref_star = ref_gen.get() * phase_shift
    samples_since_time_zero ++
    signal_rooted = compute_sroot(0, sample.real())
    compute_sroot(1, ref_star.real())

    if signal_rooted:
        compute_difference(0, AHEAD_OF_REF)
        aadj = gamma*drad

        if fabs(aadj) > beta:
            theta_unwrapped -= aadj
            set_theta(theta - aadj)

    return theta

Extract Signal

Our models employ a block known called Extract Signal, which is simply a bandpass filter fed through an automatic gain control. The bandpass filter helps to filter out extraneous frequencies and the AGC helps mainly with visualizations. In an oscilloscope visualization, comparing two similar signals is more difficult if the amplitudes are very different.

Figure 12: Extract Signal
This research is meant to be a small piece of a larger localization project. The larger project requires the computation of the phase difference between two or more signals – where these differences are then used to compute the location or movement of interest.

Our favorite method for computing the difference lies in the formula 6. Multiplying two signals together gives an upper side band and a lower side band. The phase of the lower side band is the phase difference of the two signals. We must then only extract the carrier to compute our answer.

\[
\cos(f_1 \cdot t + \theta) \cos(f_2 \cdot t + \phi) = \frac{1}{2} [\cos((f_1 + f_2)t + (\theta + \phi)) + \\
\cos((f_1 - f_2)t + (\theta - \phi))] (6)
\]

Notice in the diagram (figure 13) that we employ a complex conjugate block when multiplying the signals. Because we filter more aggressively for the upper side band of the input signals, the result of the multiplication mainly gives only the upper side band. That is, when looking at an FFT,
there is little \(\cos((f_1 - f_2) + (\theta - \phi))\) in the result. Most of the result leans to the \(\cos((f_1 + f_2) + (\theta + \phi))\) component of the multiplication.

Since we are only interested in \((\theta - \phi)\), we apply a complex conjugate function to the lower frequency input signal before multiplication. This gives the desired result of \(\cos((f_1 - f_2) + (\theta - \phi))\).

Using this block, Rootsync would only have to examine one output wave to compute the difference between two others. This could save some computation and possibly help to provide greater accuracy. In order to test these ideas, we will later need to build a system with at least two signal generators.
Simulation Testing Results

Our early tests of Rootsync and EMEdf all showed great success. Once the basic models were figured out, everything seemed to work and we expected the next step would be a simple matter of tuning into some radio signals and using Rootsync to compute the phase. In fact, experiments showed further modifications are needed to get Rootsync to work with a USRP and an RF signal.

Non-Simulated Testing

In our empirical testing, we used an Ettus Research USRP2, a LFRX Daughterboard (which can tune DC – 30 MHz) and an AM radio antenna. The board can actually tune in and record multiple MHz bandwidth with no trouble at all, but our antenna tuner limits us to roughly 10 kHz and the nature of the project desires exactly 1 Hz in the ideal case. It is safe to say the USRP has quite a bit more capability than we really need. A sample rate of 250 ksp would be plenty, but the USRP produces sample rate errors at rates lower than 1 Msp. Figure 14 shows the USRP and antenna rig mounted to a board for use in a car during drive testing – since this is also a handy way to transport the unit between locations, it has not been removed from the board since the drive testing. The drive test results are not discussed since they failed to produce useful results, but they were
the impetus behind the Mobile Science Station.

![WMU Mobile Science Station](image1.png)

**Figure 14: WMU Mobile Science Station**

For most of our commercial AM testing we recorded ten minutes worth of samples to a file using a sample rate around 384 ksp (kilo samples per second). Again, the USRP does not cope well with such a low sample rate and actually produces a slightly higher rate, which makes frequencies in turn seem a bit lower than they should be – this is not such a terrible problem since the USRP does not tune accurately anyway. A diagram of the final GNU Radio Rootsync test configuration is shown far below in figure 23.

We tuned the antenna to 1360 kHz and tuned the UHD block (the USRP) to 1340 kHz. The resulting signal of interest (1360 kHz) appears as a 20 kHz signal in the testing file. These tests are therefore with respect to this 20 kHz signal. This gives roughly 20/384 samples per cycle, or 9.6 samples per root (SPR). Clearly, the interpolation shown in figure 11 is
necessary to lock onto the carrier properly at that rate. We had considered trying to move the signal closer to baseband (by tuning the USRP to, for example, 1350 kHz) but chose to operate well away from baseband. Commercial radio has a roughly 10 kHz bandwidth and we wished to have around 10 kHz channel spacing from baseband.

When we first began testing with this data file, we immediately noticed that none of our models worked in any meaningful or useful way. Our measure of the phase as output by the models ($\theta$) rather than being relatively constant – as it is in the simulator – would rapidly roll in one direction. By roll, we mean, the adjustments to keep the reference in phase were always in one direction, relatively constant, and kept going past an entire

Figure 15: 9.6 samples per root
cycle (π) without any end. Why? We employ a reference signal to measure theta. More precisely, we generate a perfect sine wave at the frequency of the signal and align the roots (indeed, this is the Rootsync method). If the reference frequency is set lower than the real signal or higher than the real signal, then this is actually the expected behavior. In point of fact, if we intentionally set the frequency of the reference incorrectly in the simulator we get this same result.

![Figure 16: Phase roll output of Rootsync on live radio θ₁ and θ₂](image)

Obviously, the signal output by the USRP is not quite where we asked it to be. We found that the USRP has a known tuning problem. They are not quite on the target asked for by the UHD driver, so the center frequency desired is slightly off. The first thing we tried to compensate for
this was to simply manually adjust the frequency of the reference signal. This is also known as trial and error: adjust, re-run, adjust, re-run, etc. It was essentially impossible to determine the frequency this way.

One of the many things we tried to find the input signal eventually evolved into the LSF and then Lag model discussed below. At first we simply introduced a training phase. If the phase rolls too far during our training period, we adjust the frequency of the reference to compensate. This is surprisingly prone to error. Though $\theta$ tends to average out to a steady value (in simulation anyway), the local variance is high enough to confuse an algorithm of this kind. Still, using this method, it became clear that the frequency itself is not constant in the samples we collected.

Even though this method is prone to error, it should work well enough to not produce the strange result that we observed. We found that once we locked on the signal we still had a slow phase roll. This was expected, since it is difficult to get the frequency exactly correct. The surprising result was that the phase roll slows, stops, and reverses several times during each of our sample recordings.

We eventually determined there were at least two problems with this approach. First, the FCC does not mandate that AM radio broadcasters keep their frequency exceedingly precise – it may vary over time. Indeed, does it really matter if an AM signal is a few Hz out of tune? Does it
matter if drifts around over a dozen Hz every few seconds? No human could hear this. In fact, the differences we are seeing are only a few Hz in either direction; so this could easily be the case – broadcaster errors could be a real problem, but the USRP itself has frequency errors in the tuner, so we decided not to focus on this yet. Another problem we identified is that it is possible to receive AM signals along multiple paths. AM radio signals can easily bounce off the ionosphere and can therefore arrive at the antenna directly and from one or more angles after bouncing off the sky. The phase of these signals will not match up well, causing the traditional fading, and also confounding our attempts to detect the phase [10] [1].

Later, after building our own transmitters, we came to believe most of the frequency locking problems were related to frequency drift rather than multi-path reception, but the problem is solved the same way: build or operate a local transmitter.

Least Square Fit

The least square fit computation of theta was added to try to rule out some other artifact in our models. It is difficult to accept that the frequency of the signal source changes over time. We wished to gain more certainty that the error was not an algorithmic fallacy. We choose to do this by looking at the overall trend of the phase roll rather than trying to compute
the roll in place.

The idea is simple. When we encounter a root, we store the time and the current value of theta in a list. When the list fills, we compute the least square fit and from this we compute the apparent frequency difference. Here we show the pseudo code for `LSF::insert`, `LSF::recalc`, `x_given_y()`, `y_given_x` and the modified `general_work` function.

```cpp
LSF::x_given_y(_y):
    if recalc_needed: recalc()
    return (_y-alpha)/beta

LSF::y_given_x(_x):
    if recalc_needed: recalc()
    return (beta*_x) + alpha

LSF::insert(_x,_y):
    recalc_needed = true
    x.insert(_x)
    y.insert(_y)

LSF::recalc():
    mnx = x.mean()
    mny = y.mean()
    vrx = x.variance()

    sum = 0
    for i := 0 .. x.size()-1:
        sum += (x[i]-mnx) * (y[i]-mny)

    cov = sum / x.size()
    beta = cov / vrx
    alpha = mny - (beta * mnx)
    recalc_needed = false
```
Using this system, we can be very confident that the frequency adjustments we make during training are generally the right adjustments to make and not a misinformed decision based on a peculiarity or local error. Indeed, it did make the output of the Rootsync much more stable, but then also showed that the phase rolls (which change direction over time) are not just a figment of some other systemic error.

Relevant Root Notched Filter

To get a clear understanding, if we were indeed trying to lock on the phase of a multi-path signal (which clearly will not work without additional
filtering), we first needed to obtain a better confidence in our frequency locking.

We devised a method for frequency locking that is less sloppy and error prone than the phase-roll training we employed above. The main problem is that there are more roots in the signal than those generated by the carrier. The data signal also produces roots, as do any other multi-path arrivals, and our new method attempts to assign a relevance score to the roots.

We define root relevance by

1. establishing an acceptable frequency window,
2. building a linked list of roots and their time of occurrence (time is measured in samples),
3. counting the number of roots in the list that compare favorably to the frequency relevance window,
4. and finally determining whether the root under scrutiny occurs within the frequency window for some threshold number of roots in the list.

This all sounds rather complicated, but it is a valuable operation to make sure the main algorithm is comparing roots from the carrier and not from the data signal or some other artifact.

The pseudo code for root relevance is as follows.
WMURootsync::root_relevant():

tmp = new root_info
tmp.time = samples_since_time_zero
tmp.root = samples_since_time_zero - sroot[0]

while root_list.size() > 50:
    root_list.shift()

i = 1
counter = 0

while i < root_list.size():
    sq = tmp.root - root_list[i].root
    sq_min = (1+counter)*min_spr
    sq_max = (1+counter)*min_spr

    if sq>=sq_min && sq<=sq_max:
        counter ++

    if sq > sq_max:
        break

relevant = counter >= 0.3*root_list.size()
root_list.push( tmp )

return relevant

As we iterate over the list, we simply compare the apparent symbol rate \( sq \) against the expected symbol rate \( \text{min}_\text{spr} \) and \( \text{max}_\text{spr} \) – if the symbol rate appears to be in the right range, then we increase the score. The numbers 50 and 30\% are completely arbitrary and should later be added as tunable options, but these numbers serve as well as any. The idea is to increase the trend toward computation on relevant roots and decrease computation based on irrelevant ones. As long as we are working with more relevance, the computation improves overall.

To make this all as clear as possible, we provide the signal diagram shown in figure 17. There, we see an 8kHz signal and an 8kHz signal mixed with some other sinusoid. The actual samples of the mixed signal
Figure 17: 8 kHz signal with mixed signal at 96 ksp

care shown as the smaller dots and the roots are shown as the bigger dots with labels. The labels indicate the time of the sample (in samples since time zero) minus the last $\text{root[0]}$ – which is shown as the third line in $\text{::root\_relevant()}$.

Suppose we just noticed a root at time 51.9. It is clear by inspection this is not a relevant root, but how would this look in the notched filter? The root list would contain 48.8, 42.6, 37, 36.9 and 30.7 – and also another 45 more roots in a real run. Our $\text{spr}$ (symbols per root) should be around $96/(2 \cdot 8) = 6$. We compare 51.9 against our list of roots $(51.9 - 48.8/1) = 3.1$, $(51.9 - 42.6/1) = 4.65$, $(51.9 - 37/1) = 4.97$, $(51.9 - 36.9/1) = 3.75$, $(51.9 - 30.7/1) = 4.24$, and conclude that 51.9 is roughly 6 symbols from zero elements in the list. By contrast, if we compare 54.7 in a similar fashion, $(54.7 - 51.9/1) = 2.8$, $(54.7 - 48.8/1) = 5.9$, $(54.7 - 42.6/2) = 6.05$, $(54.7 - 37/3) = 5.9$, $(54.7 - 36.9/3) = 4.45$, we may conclude that 54.7 is roughly 6 symbols from three elements in the list. If we use 50% as our
threshold score, and 0.1 as our frequency window (really SPR window), then 54.7 is a relevant root and 51.9 is not. Note that the divisor (called counter in the pseudo code) only advances if the root falls within the frequency window.

The combination of LSF frequency adjustments, root relevance filtering, and our debug log format allowed us to show that there is definitely a frequency drift problem. We did a careful plot of theta over the course of our minute-and-a-half long recording and the result is very convincing (figure 18). “Unwrapped theta” is (just like it sounds) theta as it is allowed to continually increase (or decrease in this case) beyond $\pi$ without rolling over. In our plot here, $\pi$ has drifted a hundred cycles in the negative direction during training. Once training is complete, theta is reasonably stable, but over the course of a minute-or-so it drifts unacceptably – a phase roll from $345 - 347$ radians over the course of approximately 50 seconds (assuming no drift problems of any kind) represents the radio moving $(347 - 345) \cdot 220 \text{ m} \cdot 50 \text{ s} = 8.8 \frac{\text{m}}{\text{s}} = 19.6 \text{ MPH}$, which is clearly spurious.

Laguerre Frequency Filter

Using long running theta changes (via LSF) to measure frequency drift and misalignment does allow us to learn and adapt to the frequency of a signal, but it is a slow and indirect route by which to acquire the informa-
The information we are actually seeking can be measured more directly by looking at the inter-root-spacing of the signal. By applying an LSF to the IRS, we were able to lock on the frequency more directly. However, smoothing of the desired rate required the LSF to have as many as 10,000 elements in the linked list. This extra CPU time caused the model to lag the USRP unacceptably (causing buffer overruns) and we began to look for another way to average the IRS.

In our very best model, we still use the frequency window for an adjustable notched filter and we then feed those inter-root-spacings to a Laguerre filter. A Laguerre filter is a construction meant to use Laguerre polynomials as a filter to adapt to a signal. The hardware constructions of
this are somewhat different from the software version we are using in this project. [2] offers up the schematic shown in figure 19.

Figure 19: Laguerre Filter Hardware Schematic

The main advantage of the Lag filter (as we call it for short) in our models is that it only requires 4 computations per root, rather than the tens of thousands the LSF computation requires – this is true, regardless of the length of the smoothing.

We also wished to look at the adaptive capabilities described in [2] but found that the non-adaptive filter works very well. Once we have chosen smoothing of the desired length (a manual process currently), changing the smoothing rate rapidly would only help for signals with suddenly changing frequencies. With those signals, its not at all clear one could achieve a positive result anyway.

The pseudo code for the Lag filter and the modified `general_work` function follows.
Lag(size)
1[4]
stage = 0
full = length
alpha = 2/(1+size)
F = -1

Lag::insert(x)
if full>0:
  full --
  // adapt_alpha() :- intentionally omitted

if stage:
  l[0] = alpha*x + (1-alpha)*o[0]
  for i=1; i<stage && i<4; i++:
    l[i] = (1-alpha)*o[i] - (1-alpha)*l[i-1] + o[i-1];
  if stage == 4:
    F = (l[0] + l[1]*2 + l[2]*2 + l[3]) / 6;
  for i=stage; i<4; i++:
    l[i] = o[i-1];
else:
  l[0] = x;
  stage = 1;
With this approach to frequency alignment, one can conduct frequency analysis while also conducting phase analysis. Phase shifts are actually complimentary to another affect called the Doppler effect. When a frequency source is moving towards a receiver the wave length appears shorter. In practice, minus a rubber-banding effect caused by lag from the smoothing effect, the phase shifts are indeed canceled out by the Doppler effect. The phase shift of the motion is immediately detectable and once the Laguerre filter catches up to the motion it reverses the phase change neatly. For this reason, frequency analysis needs stop before phase analysis can really begin. The WMU Rootsync block therefore has a setting to dis-
able the frequency analysis after some set number of roots have passed. Later builds might choose to turn frequency training on and off as needed (as determined by some as-yet unknown criteria), rather than after some arbitrary duration.

This set up works really well in the audio domain (which also works well with the theta roll training), but still fails to produce useful results using commercial AM. With the new frequency analysis the problem of frequency drift is much more obvious and apparent. Just a few mHz will cause a phase roll, which will may appear to be faster than a human can walk. Consider that a Hz is a cycle per second and that at 1 MHz RF, the wavelength is approximately 980 ft. Therefore, a single Hz difference in the real life signal compared to the ref will appear to be motion of approximately 980 ft/s. The effects are not as bad as it sounds though, since the USRP xlates the MHz signal down to a kHz. In our models then, the frequency differences and drift appear to be in the mHz range.

Audio Domain Tests

After having not resolved these issues after many months of trial and error and in light of plots similar to those shown in figure 18, we decided to briefly take a look at the audio domain. With audio, we can control for many of the most annoying variables. We can take steps to ensure that
our source is constant (not caring if the frequency generated is precisely accurate since we can easily adapt for that). We can take steps to minimize multipath, and we can then test the USRP and our models. Really, this is true for any signal generated locally, but audio is very easy to work with.

Our transmitter in this configuration is nothing more than an Android phone with a tone generator. We chose a frequency of 2.7 kHz which, although annoying to listen to, is ideal because of scale. The chosen frequency has a wavelength of \( \frac{343.36 \text{ m/s}}{2.7 \text{ kHz}} \approx 12.7 \text{ cm} \) – a distance that is easy to deal with on a desk. Since we make no attempt to determine which of two possible phases we might be locking onto, our resolution is half that distance or maybe 6 cm. We did not worry about finding the exact value of the wavelength, since we do not have a good number for the wave velocity – i.e., the speed of sound, which varies by air pressure and temperature.

The GNU Radio model is shown in figure 20. The results in this configuration are dramatic and instantly convincing The Rootsyc module locks on the frequency very quickly and the \( \theta \) output of the block (plotted on a GNU Wx Scope) produces straight lines that wiggle slightly, owing to slight movements of the operator’s hands holding the phone. When the phone is moved away and toward the computer, the lines indicating theta move up and down accordingly. Moving roughly 17 cm rolls \( \theta \) roughly 3 radians and
moving 1.7 m causes $\theta$ to roll roughly 10 times.

This works so well, in fact, that there can be little doubt that our models work, at least in principle, if not using ambient AM radio. Figure 21 shows the trace of theta as we move the phone back and forth. The drift shown is due to the fact that holding the phone (the signal generator) steady is difficult, however the computation is precise enough that it was simple to make sure theta moved exactly enough to span the wavelength for the plot without wrapping and making the diagram less attractive.

We were also able to produce less satisfying results by enclosing our experiment in a regular office cubicle. This created enough multipath to
confound the results somewhat, producing some of the same problems as seen with the ambient AM signals. This suggests that we may be fighting multipath in the real world. However, these phase/frequency differences are much more erratic than those seen with AM, suggesting frequency drift with AM. This difference may not matter since the problem is the same either way.

**WMU Radio Station**

Because commercial radio lacks the precision necessary to do our phase analysis, it became logical to simply generate our own signal like we did in the audio domain. Since we desire to consider only the carrier of a signal, it made sense for us to generate a signal that lacked any modulation at all. There exist inexpensive clock components that suited our purpose. We purchased a simple 1 MHz TTL oscillator component (Jameco #27861) and wired the output pin directly to a loop antenna. With this setup, we

![Figure 21: Audio domain 2.7 kHz run](image)
produced a signal with a range of around 30 ft with a very precise frequency.

![Image](image1.png)

**Figure 22: WMU 1,000,000 Hz Radio**

Our radio does not produce a signal that would be usable for anything besides this project, but it suits our purposes perfectly. The radio puts around 45 mA on the antenna at 5 V and oscillates at precisely 1 MHz. Using a simple coiled antenna, we receive approximately $10^{-100}$ mW of power at the USRP, which gives us approximately 30 ft of operating range. This is not enough range for a practical test of the system due to near field effects, but it is good enough to demonstrate the effect we are after.

Let us stress that point further. At a range of 30 ft, our transmitter is still well within the range of a single wavelength, which implies that there will be strong nearfield effects. Inside the neighborhood of the first wavelength, the waves take on a magnetic dipole shape; only outside a
wavelength or so, do signals become radiative the way we would normally expect\cite{5}.

Figure 23: The Final USRP Sync diagram

In our tests moving the antenna several inches moved theta a few deci-radians. That is, if we moved toward the receiver, theta would move one way, and if we moved the antenna the other way, theta would show this also. We consider this a positive result however, theta moves too far. If we only move the antenna a few feet, we should hardly notice the effects on the system. That would certainly be true if our transmitter was a good distance away, but because our waves wrap tightly around the dipole, the signal seems to move further than it should in the receiver. This is all shown in the run results of figure 24 Future transmitters will clearly have to broadcast with more power, with better antennae and probably with a license.
Figure 24: WMU 1,000,000.00 Hz Radio sync run

The peak marked “A” is the actual motion of the antenna. The long slope marked “B” is just frequency drift. After moving the antenna around, the capacitance/inductance of the operator (humans have a tiny amount of capacitance and inductance) seems to alter the output frequency of the antenna – we find that simply touching the antenna changes the frequency of the array by several Hz. Future radios, perhaps filtered properly or of a better design will be able to compensate for this effect. The important thing to note is that theta does not drift much until the antenna moves one way, then moves back when the antenna moves another. Once we have a transmitter with a little bit better design, this will all work as smoothly as the audio domain tests. [5]
CHAPTER IV

CONCLUSIONS AND FUTURE IMPROVEMENTS

It is important to note that this work was only prototyping and feasibility testing. Although we thought about how to use these ideas in usable systems or products, that was not our goal. Many of the details of this work may change when attempting to do real location measurements. We may need to choose higher frequencies (to shorten antenna requirements) and we may choose to have our receivers transmit as well as receive (see below).

There are significant challenges that still need to be overcome but they are clearly solvable. The frequency drift problem itself seems easiest to overcome (at least mostly) by designing a better signal generator, which can probably be done using very inexpensive off the shelf components. In the worst case, there exist expensive products that can do this job at much high power levels and better accuracy than what we have built here.

Such transmitters will likely require licensing or special permission by the FCC. There may exist a HF scientific band for such purposes, or one might be created. A system like this would need a channel occupied bandwidth of perhaps 10 – 100 Hz. There are, for example, some government use channels in 4 MHz range. The government might have an interest in
helping people navigate public structures and might free up a tiny amount of their own bandwidth for this purpose.

Frequency drift is enough of a problem that this effect probably cannot be used practically until we add additional strategies to detect it – small temperature changes in the transmitter or receiver could easily appear to be user motion. Since frequency changes can be either clock drift or Doppler effect and frequency changes are difficult to distinguish from phase changes, it may be best to look elsewhere for that information. We imagine a device that contains accelerometers, magnetometers or gyroscopes. The information from these devices could be used to enable and disable frequency training, or used to correlate motion with detected phase changes, or perhaps both.

We also imagine a self contained receiver unit, perhaps the size of a softball, that operators would carry with them. The antenna of the device might be wound around a core in a few orientations, giving it a spherical shape. The reason for this shape choice is that a loop antenna has significant blind spots along the plane of the winding.

With a configuration of several transmitters located around a campus, the receiver could then compute its location using simple triangulation. In this scenario, operators would quickly train the receivers at known starting points or pickup/drop-off stations. This hardly seems practical since the
receivers would need to remain always on and would become lost without resetting them to some known starting point.

Another idea uses a kind of challenge response for phase analysis (timing digram in figure 25). Receivers in this setup are called Tags. Rather than just being a simple receiver (as above), a Tag would have a wifi card for simple network communication and an AM Transceiver. Tags would send out a listing request to the AM Basestation Registry Service over wifi. The Registry Service would send back a list of participating AM Basestations, also over wifi.

The Tag would then send out wifi location service requests to several of the AM BS. They would in turn send out a start frequency to the Tag, still over wifi. The Tag would then generate a local carrier and synchronized transmit carrier over the AM frequency described by the AM BS. The BS would generate a local carrier in sync with the received signal and then send a stop instruction over wifi back to the Tag. When the Tag stops transmitting, the BS would send out a signal in sync with the local carrier and finally, the Tag could compare the initial local carrier with the received signal. Analyzing the phase of received signals from several BS would allow the Tag to compute a location.
Figure 25: Protocol for location by phase analysis
REFERENCES


REFERENCES–Continued


APPENDIX

dbg.h

#ifndef ___DEBUG
#define ___DEBUG

#include <wmu_api.h>
#include <stdio.h>
#include <queue>
#include <string>
#include <map>

#define BOOLSTR(x) ( x ? "true" : "false" )

using namespace std;

typedef struct _debug_msg debug_msg;
enum _type { ddouble, duint, dstr };

struct _debug_msg {
    string tag;
    double double_val;
    unsigned int uint_val;
    string str_val;
    _type type;
};

class debug_logger {
private:
    unsigned short tagno;
    map<string,unsigned short>tags;
    queue <debug_msg> mq;
    FILE *fh;

public:
    // use actual types
    void msg(string tag, double value);
    void msg(string tag, unsigned int value);
    void msg(string tag, string value);
    void print(unsigned int time);

    // really just type casts and things
    void msg(string tag, unsigned short value);
    void msg(string tag, unsigned char value);

    debug_logger();
    ~debug_logger();
};

#cmakedefine FREQ_TRAIN_DEBUG
#cmakedefine ROOT_RELEVENCE_DEBUG
#cmakedefine ROOT_INFO_DEBUG
#cmakedefine THETA_ADJ_DEBUG
#cmakedefine SIGNAL_DEBUG
```c
#define DEBUG_DECLARE() debug_logger DEBLOG
#define DEBUG_INIT() /* DEBLOG = new debug_logger() */
#define DEBUG_DEINIT() /* delete DEBLOG */
#define DEBUG_DONE(X) DEBLOG.print(X)
#define DEBUG(X,Y) DEBLOG.msg(X,Y)

#define DEBUG_DECLARE() /* DEBUG */
#define DEBUG_INIT() /* DEBUG */
#define DEBUG_DEINIT() /* DEBUG */
#define DEBUG_DONE(...) /* DEBUG */
#define DEBUG(...) /* DEBUG */
#endif

// console debugging (never disables)
#define CDEBUG(...) fprintf(stderr, __VA_ARGS__)
#endif

wmu_api.h

#ifndef INCLUDED_WMU_API_H
#define INCLUDED_WMU_API_H
#include <gruel/attributes.h>
#ifdef gnuradio_wmu_EXPORTS
#define WMU_API __GR_ATTR_EXPORT
#else
#define WMU_API __GR_ATTR_IMPORT
#endif
#define too_damn_close 0.00000001
#define non_zero_divisor_ize(X) \n  (fabs(X)<too_close ? (X<0 ? -too_close : too_close) : X)
#define LAST_NOT_DEFINED -77.77077
#include <debug.h>
#endif /* INCLUDED_WMU_API_H */

wmu_root_sync.h

#ifndef INCLUDED_ROOTSYNC_CCC_H
#define INCLUDED_ROOTSYNC_CCC_H
#endif
```

70
#include <wmu_api.h>
#include <gr_block.h>
#include <gr_complex.h>
#include <gr_fxpt_nco.h>

#include <lsf.h>
#include <laguerre.h>

#include <list>

#define uint unsigned int
#define uchr unsigned char
#define ushr unsigned short

class root_info {
public:
    uint time;  // time of root
    double root; // time of root
    double sq; // symbols using start of queue / applicable root number
    double fq; // frequency using sq (sps / 2sq)
};

class wmu_root_sync_ccc;

typedef boost::shared_ptr<wmu_root_sync_ccc> wmu_root_sync_ccc_sptr;

WMU_API wmu_root_sync_ccc_sptr wmu_make_root_sync_ccc(double alpha=0.7,
    double beta=0.001, double gamma=0.9, double theta=0,
    double rfreqw=0.01, uint laglen=500, uint lagtlen=0,
    uint samp_rate=1000, double ref_freq=1000);

class wmu_root_sync_ccc : public gr_block {
private:
    friend WMU_API wmu_root_sync_ccc_sptr wmu_make_root_sync_ccc(
        double alpha, double beta, double gamma, double theta,
        double rfreqw, uint laglen, uint lagtlen, uint samp_rate,
        double ref_freq);

    wmu_root_sync_ccc(double alpha, double beta, double gamma, double theta,
        double rfreqw, uint laglen, uint lagtlen, uint samp_rate,
        double ref_freq);

    double freq;
    uint sps;
    gr_fxpt_nco ref_gen; // ref signal generator
    uint roots_since_time_zero, samples_since_time_zero;
    double alpha, beta, gamma, theta, theta_unwrapped;
    // alpha used in drad=drad*(1-alpha)+drad*alpha
    // beta do something if fabs(gamma*aadj)<beta
    // gamma ref* = ref + e^[(theta + drad*gamma)*i])
    // theta ref* = ref * (phase_shift=e^theta*i))
    double last[2]; // [0] the very last real() reading [1] of ref
double sroot[2]; // [0] samples since the last root [1] of ref
double spr;       // samples per root (computed from freq and rate
double drad;      // current offset between ref and signal
double aadj;      // adjusted adjustment, adj=gamma*drad

int roots_since_reset[2];
// there are various computations that depend on the roots being
// accurately counted we use this to count the roots since the last
// time we reset this variable (like it's named) typically we avoid
// computations on spr until the reset rsr reaches a certain point

gr_complex phase_shift; // the current phase shift

void compute_phase_shift(); // sets phase_shift from theta
char compute_sroot(char idx, double now);
void compute_difference(char idx, char direction);
double wrap_phase(double x);
double wrap_phase_adj(double x);

double relevant_frequency_window_factor;
double min_spr, max_spr;
list <root_info> root_list; // a list of root information
uint relevant_roots_since_time_zero;

Laguerre freq_lag;
bool freq_lag_is_enabled;
uint lag_training_roots;

bool root_relevant();

DEBUG_DECLARE();

public:

"wmu_root_sync_ccc();

int general_work (int noutput_items,
    gr_vector_int &ninput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items);

// NOTE: WMU_API means the symbol should be set exported in the shared
// library public members that lack WMU_API will be in the shared
// library, but will not be accessible to dynamic loaders.

WMU_API void set_alpha(double a);
WMU_API void set_beta( double b);
WMU_API void set_gamma(double g);
WMU_API void set_theta(double t);
WMU_API void set_freq( double f);
WMU_API void set_spr( double r);
WMU_API void set_rate( uint  r);

WMU_API double get_alpha();
WMU_API double get_beta();
WMU_API double get_gamma();
WMU_API double get_theta();
WMU_API double get_freq();
WMU_API double get_spr();
WMU_API uint  get_rate();

};

#endif
#ifndef ___LAGUERRE__
#define ___LAGUERRE__

#include <stdio.h>
#include <math.h>
#include "slist.h"

class Laguerre {
  private:
    double l[4];
    int stage; // counter to see where we are in the polynomial filter
    int full; // counter to see if F is ready
    double alpha;

  public:
    double F; // filter output
    Laguerre(size_t length) {
        reset(length);
    }

    bool is_full() { return full<1; }

    void reset(size_t length) {
      alpha = 2.0/(1.0+length);
      stage = 0;
      full = length;
      F = -1;
    }

    void insert(double x) {
      if( full > 0 )
        full --;
      if( stage ) {
        double o[4];
        int i;
        for(i=0; i<4; i++)
          o[i] = l[i];
        l[0] = alpha*x + (1-alpha)*o[0];
        for(i=1; i<stage & i<4; i++)
          l[i] = (1-alpha)*o[i] - (1-alpha)*l[i-1] + o[i-1];
        if( stage == 4 )
          F = (l[0] + l[1]*2 + l[2]*2 + l[3]) / 6.0;
        for(i=stage; i<4; i++)
          l[i] = o[i-1];
      }
    }
}
if( stage < 4 )
    stage ++;
} else {
    l[0] = x;
    stage = 1;
}

}

#endif

lsf.h

#ifndef ___LSF__
#define ___LSF__

#include <stdio.h>
#include <math.h>

#include "slist.h"

class LSF {
    private:
        size_t _size;
        bool recalc_needed;
        double alpha,beta;
        slist x, y;
    
        void recalc() {
            double mnx = x.mean();
            double mny = y.mean();
            double vrx = x.variance();
            
            siter i = x.begin();
            siter j = y.begin();
            siter e = x.end();  // same size or sagfault ...
            double sum = 0;
            while( i!=e ) {
                sum += (*i-mnx) * (*j-mny);
                ++i;
                ++j;
            }
            double cov = sum / x.size();
            beta = cov / vrx;
            alpha = mny - (beta * mnx);
            recalc_needed = false;
        }
    
    public:
        LSF(size_t s) : x(s), y(s) {
private:
    list<double> members;
    size_t _size;
    double _mean, _variance;
    bool recalc_needed;

    void recalc() {
        siter i = members.begin();
        siter e = members.end();
        double sum = 0;
        while(i != e) { sum += *i; ++i; }
    }
_mean = sum / members.size();

i = members.begin();
e = members.end();
sum = 0;
while(i != e) { sum += pow(*i - _mean, 2); ++i; }

_variance = sum / members.size();
recalc_needed = false;
}

public:
  slist(size_t s) { recalc_needed = false; _size = s; }

  void insert(double x) { recalc_needed = true; members.push_back(x);
    while(members.size() > _size) members.erase(members.begin()); }

  siter begin() { return members.begin(); }
  siter end() { return members.end(); }
  size_t size() { return members.size(); }
  bool full() { return members.size() == _size; }

  double mean() { if(recalc_needed) recalc(); return _mean; }
  double variance() { if(recalc_needed) recalc(); return _variance; }

  void clear() { members.clear(); }
};

#endif

debug.cc

#include <debug.h>
#include <stdlib.h>

int debug_object_number = 0;

void debug_logger::msg(string tag, double value) {
  debug_msg m;
  m.tag = tag;
  m.double_val = value;
  m.type = ddouble;
  mq.push(m);
}

void debug_logger::msg(string tag, unsigned int value) {
  debug_msg m;
  m.tag = tag;
  m.uint_val = value;
  m.type = duint;
  mq.push(m);
}

void debug_logger::msg(string tag, unsigned short value) {
  msg(tag, (unsigned int) value);
}
void debug_logger::msg(string tag, unsigned char value) {
    msg(tag, (unsigned int) value);
}

void debug_logger::msg(string tag, string value) {
    debug_msg m;
    m.tag = tag;
    m.str_val = value;
    m.type = dstr;
    mq.push(m);
}

void debug_logger::print(unsigned int time) {
    debug_msg m;
    if( mq.size() ) {
        fprintf(fh, "\%d:\%d", 0, time);
        while( !mq.empty() ) {
            m = mq.front();
            if( !tags[m.tag] )
                fprintf(fh, "\[\%s: %d\]", m.tag.c_str(),
                        tags[m.tag] = ++ tagno);
            switch(m.type) {
                case ddouble:
                    fprintf(fh, "\%d:%.7f", tags[m.tag], m.double_val);
                    break;
                case duint:
                    fprintf(fh, "\%d:%d", tags[m.tag], m.uint_val);
                    break;
                case dstr:
                    fprintf(fh, "\%d:%s", tags[m.tag], m.str_val.c_str());
                    break;
            }
            mq.pop();
        }
        fprintf(fh, "\n");
        fflush(fh);
    }
}

definition of debug_logger() {
    char fname[255];
    char *d = getenv("DEBUG");
    snprintf(fname, 255, "/tmp/%s-root_sync-%d.log", 
            d ? d : "blarg", ++debug_object_number);
    fh = fopen(fname, "w");
    tagno = 0; // time
    fprintf(fh, "[time: 0]\n");
}

definition of debug_logger() {
    fclose(fh);
}
#include <wmu_root_sync_ccc.h>
#include <gr_io_signature.h>
#include <complex.h>
#include <time.h>

#define M_TWOPI (2*M_PI)
#define M_PIH (M_PI/2)
#define AHEAD_OF_REF 1
#define BEHIND_SIGNAL -1

wmu_root_sync_ccc_sptr wmu_make_root_sync_ccc(
    double _alpha, double _beta, double _gamma, double _theta,
    double rfreqw, uint laglen, uint lagtlen, uint _samp_rate,
    double _ref_freq) {
    return gnuradio::get_initial_sptr(
        new wmu_root_sync_ccc(_alpha, _beta, _gamma, _theta,
            rfreqw, laglen, lagtlen, _samp_rate, _ref_freq)
    );
}

wmu_root_sync_ccc::wmu_root_sync_ccc(
    double _alpha, double _beta, double _gamma, double _theta,
    double rfreqw, uint laglen, uint lagtlen, uint _samp_rate, double _ref_freq)
    : gr_block( "root_sync_ccc", gr_make_io_signature(1,1, sizeof(gr_complex)),
        gr_make_io_signature(3, 3, sizeof(gr_complex))),
    freq_lag(laglen)
    {
        DEBUG_INIT();
        theta_unwrapped = 0;
        freq_lag_is_enabled = laglen > 0;
        lag_training_roots = lagtlen;
        relevant_frequency_window_factor = rfreqw;
        set_alpha(_alpha);
        set_beta(_beta);
        set_gamma(_gamma);
        set_theta(_theta);
        set_rate(_samp_rate);
        set_freq(_ref_freq);
        last[0] = LAST_NOT_DEFINED;
        relevant_roots_since_time_zero = roots_since_time_zero = samples_since_time_zero = 0;
        sroot[0] = sroot[1] = drad = 0;


```cpp
wmu_root_sync_ccc::~wmu_root_sync_ccc() { DEBUG_DEINIT(); }

double wmu_root_sync_ccc::get_alpha() { return alpha; }
double wmu_root_sync_ccc::get_beta() { return beta; }
double wmu_root_sync_ccc::get_gamma() { return gamma; }
double wmu_root_sync_ccc::get_theta() { return theta; }
double wmu_root_sync_ccc::get_spr() { return spr; }
uint wmu_root_sync_ccc::get_rate() { return sps; }
double wmu_root_sync_ccc::get_freq() { return freq; }

void wmu_root_sync_ccc::set_alpha(double a) { alpha = a; }
void wmu_root_sync_ccc::set_beta( double b) { beta = b; }
void wmu_root_sync_ccc::set_gamma(double g) { gamma = g; }
void wmu_root_sync_ccc::set_theta(double t) {
    drad = aadj = roots_since_reset[0] = roots_since_reset[1] = 0;
    theta = wrap_phase(t);
    compute_phase_shift();
}
void wmu_root_sync_ccc::set_spr(double r) {
    freq = sps/(2.0*(spr=r));
    drad = aadj = roots_since_reset[0] = roots_since_reset[1] = 0;
    ref_gen.set_freq(2*M_PI * freq / sps);
    // from gr_sig_source_X:
    // d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);
    #ifdef FREQ_TRAIN_DEBUG
    DEBUG("rate", sps);
    DEBUG("spr", spr);
    DEBUG("freq", freq);
    #endif
    double frfwf = freq*relevant_frequency_window_factor;
    min_spr = sps / (2 * (freq+frfwf));
    max_spr = sps / (2 * (freq-frfwf));
}
void wmu_root_sync_ccc::set_freq(double f) {
    freq = f;
    spr = (sps/freq)/2.0;
    drad = aadj = roots_since_reset[0] = roots_since_reset[1] = 0;
    ref_gen.set_freq(2*M_PI * freq / sps);
    // from gr_sig_source_X:
    // d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);
    #ifdef FREQ_TRAIN_DEBUG
    DEBUG("rate", sps);
    DEBUG("spr", spr);
    DEBUG("freq", freq);
    #endif
    double frfwf = freq*relevant_frequency_window_factor;
    min_spr = sps / (2 * (freq+frfwf));
    max_spr = sps / (2 * (freq-frfwf));
}
```
void wmu_root_sync CCC::set_rate(uint r) {
    sps = r;
    spr = (sps/freq)/2.0;

    drad = aadj = roots_since_reset[0] = roots_since_reset[1] = 0;

    ref_gen.set_freq(2*M_PI * freq / sps);
    // from gr_sig_source_X:
    // d_nco.set_freq (2 * M_PI * d_frequency / d_sampling_freq);

    #ifdef FREQ_TRAIN_DEBUG
    DEBUG("rate", sps);
    DEBUG("spr", spr);
    DEBUG("freq", freq);
    #endif
}

void wmu_root_sync CCC::compute_phase_shift() {
    phase_shift = cexp( theta * I );
}

double wmu_root_sync CCC::wrap_phase(double x) {
    int max = 1000; // this may seem high,
    // but when spr=1.0 or so and sroot is like 100+ ... well ...

    while( x > M_PI || x < -M_PI ) {
        if(x > 0)
            x -= M_TWOPI;
        else
            x += M_TWOPI;

        if( (--max) < 0 )
            throw std::runtime_error("wrap_phase() died horribly [1]!");
    }

    return x;
}

double wmu_root_sync CCC::wrap_phase_adj(double x) {
    x = wrap_phase(x); // start with a well behaved phase difference

    if( x > M_PI ) // 2.7 -> -0.44
        return x - M_PI;

    if( x < -M_PI ) // -2.7 -> 0.44
        return x + M_PI;

    if( x > M_PIH || x < -M_PIH )
        throw std::runtime_error("wrap_phase_adj() died horribly [2]!");

    return x;
}

bool wmu_root_sync CCC::root_relevant() {
    root_info tmp;

    // XXX: there should probably be settings for 50 (relevance_list_size?
     // relevance_history_size? something like that) and for relevance counting
     // scores, 0.3*50, (relevance_winner_factor=0.3?)

    return false;
}
tmp.time = samples_since_time_zero;
tmp.root = samples_since_time_zero - sroot[0];

while( root_list.size() > 50 )
    root_list.erase( root_list.begin() );

list<root_info>::reverse_iterator i = root_list.rbegin();
list<root_info>::reverse_iterator e = root_list.rend();

double local_spr = tmp.root - i->root;

int counter = 0;
for(; i!=e; ++i) {
    double sq = tmp.root - i->root;
    double sq_min = (1+counter)*min_spr;
    double sq_max = (1+counter)*max_spr;

    if( sq>=sq_min && sq<=sq_max )
        counter ++;

    if( sq > sq_max )
        break;
}

bool relevant = counter >= 0.3*root_list.size() ? true : false;

// CDEBUG("count: %d of %d\n\n", counter, 20);

#ifdef ROOT_RELEVENCE_DEBUG
    DEBUG("spr", local_spr);
    DEBUG("min_spr", min_spr);
    DEBUG("max_spr", max_spr);
#endif

if( relevant ) {
    #ifdef ROOT_RELEVENCE_DEBUG
        DEBUG("relevant", "true");
        DEBUG("relevant-spr", local_spr);
        #endif

    if( freq_lag.is_enabled() ) {
        if( !--lag_training_roots )
            freq_lag_is_enabled = false;
    }

    freq_lag.insert(local_spr);

    if( freq_lag.is_full() ) {
        local_spr = spr = freq_lag.F;
        freq = sps/(2.0*local_spr);
        ref_gen.set_freq(2*M_PI * freq / sps);
        #ifdef ROOT_RELEVENCE_DEBUG
            DEBUG("freq_lag", freq);
            #endif
    }
} else {
    #ifdef ROOT_RELEVENCE_DEBUG

DEBUG("relevant", "false");
#endif

root_list.push_back( tmp );
return relevant;
}

char wmu_root_sync_ccc::compute_sroot(char idx, double now) {
    double h1, h1_h2, osr;
    bool rooted = false;

    if( last[idx] != LAST_NOT_DEFINED ) {
        sroot[idx] ++;
        if( (now>0 && last[idx]<=0) || (now<0 && last[idx]>=0) ) {
            roots_since_reset[idx] ++;
            rooted = true;
            osr = sroot[idx];
            if( fabs(now) == 0 ) {
                sroot[idx] = 0;
            } else {
                h1 = fabs(last[idx]);
                h1_h2 = h1 + fabs(now);
                if( h1_h2 == 0 ) {
                    sroot[idx] = 0;
                    throw std::runtime_error("h1h2 is 0, horror/death/demise");
                } else {
                    /*
                    time: 1 0 -1
                    x h1 = fabs(last[idx])
                    |\ 
                    | \ phi = atan2(1,h1+h2)
                    -------------------------------
                    | \ pos = h1 * tan(phi) = h1 / (h1+h2)
                    | \ 
                    +-----x h2 = fabs(now)
                    */
                    time: 1 0 -1
                    +-----x h2 = fabs(now)
                    | 
                    | / pos = h1 * tan(phi) = h1 / (h1+h2)
                    -------------------------------
                    | / phi = atan1(1,h1+h2)
                    |/
                    x h1 = fabs(last[idx])
                    
                    The value pos is the length of time since 1 until the root.
                    sroot[idx] should then be (1-pos)
                    h1 * tan( atan2(time=1,h1+h2) ) = h1/(h1+h2)
                    */
sroot[idx] = 1 - ( h1 / h1_h2 );

#ifdef ROOT_INFO_DEBUG
DEBUG("root-idx", (uint) idx);
DEBUG("sroot[0]", sroot[0]);
DEBUG("sroot[1]", sroot[1]);
#endif

if( !idx ) {
    if( rooted = root_relevant() )
        relevant_roots_since_time_zero ++;
    roots_since_time_zero ++;
}

// end rooted

// end very-first-one

last[idx] = now;

return rooted;

void wmu_root_sync_ccc::compute_difference(char idx, char direction) {
    double _drad, droot;
    if( roots_since_reset[0] > 1 && roots_since_reset[1] > 1 ) {
        droot = sroot[idx + direction] - sroot[idx];
        _drad = wrap_phase_adj( direction * (M_PI / spr) * droot );
        drad = drad*(1-alpha) + alpha*_drad;
    }
}

int wmu_root_sync_ccc::general_work (int noutput_items,
                                     gr_vector_int &ninput_items,
                                     gr_vector_const_void_star &input_items,
                                     gr_vector_void_star &output_items) {
    const gr_complex *signal;
    gr_complex *ref_star;
    int number = noutput_items<ninput_items[0]
    ? noutput_items : ninput_items[0];
    int i;
    char _s,_r;
    double s,r;
    gr_complex ref[1];
    signal = (gr_complex *) input_items[0];
    ref_star = (gr_complex *) output_items[0];
for(i=0; i<number; i++) {
    ref_gen.sincos(ref, 1); // from gr_sig_source_X
    ref_star[i] = ref[0] * phase_shift;
    s = signal[i].real();
    r = ref_star[i].real();

    #ifdef SIGNAL_DEBUG
    DEBUG("signal", s);
    DEBUG("ref", r);
    DEBUG("theta", theta);
    DEBUG("theta-wrap", wrap_phase( theta - M_PI ));
    #endif

    samples_since_time_zero ++;
    _s = compute_sroot(0, s);
    _r = compute_sroot(1, r);

    // /* if(_s) */ compute_difference(0, AHEAD_OF_REF);
    // /* if(_r) */ compute_difference(1, BEHIND_SIGNAL);
    // XXX: moved below compute_difference(); but not tested
    // aadj = gamma*drad;

    if( _s ) {
        compute_difference(0, AHEAD_OF_REF);
        aadj = gamma*drad;

        #ifdef THETA_ADJ_DEBUG
        DEBUG("aadj", aadj);
        #endif

        if( fabs(aadj) > beta ) {
            theta_unwrapped -= aadj;
            set_theta(theta - aadj);
        #ifdef SIGNAL_DEBUG
        DEBUG("set-theta", theta);
        DEBUG("theta-unwrapped", theta_unwrapped);
        #endif
        } // if aadj big enough
    } // if rooted

    ( (gr_complex *) output_items[1] )[i] = ref[0];
    ( (gr_complex *) output_items[2] )[i] = theta
        + wrap_phase( theta - M_PI )*1j;

    DEBUG_DONE(samples_since_time_zero);
}

static time_t spam_time = time(NULL);
time_t new_time = time(NULL);
if( spam_time < new_time ) {
    double seconds = (1.0*samples_since_time_zero) / sps;
    if( freq_lag_is_enabled ) {
        if( lag_training_roots ) {
            // NOTE: not valid syntax in C++, for appendix only
            // ("== ==" string concatenation is
            //  valid in opc/pike though)
CDEBUG("time=%4.1f (%06d); freq=%8.4f; spr=%12.8f; "
"theta=%8.4f; freqw=%0.5f; relevance=%0.1f%% "
"[lag-enabled lag-training-roots: %d]\n",
seconds, samples_since_time_zero, freq, spr, theta,
relevant_frequency_window_factor,
(100.0*relevant_roots_since_time_zero)
/ roots_since_time_zero,
lag_training_roots);

} else {
 CDEBUG("time=%4.1f (%06d); freq=%8.4f; spr=%12.8f; "
"theta=%8.4f; freqw=%0.5f; relevance=%0.1f%% "
"[lag-enabled] \n",
seconds, samples_since_time_zero, freq, spr, theta,
relevant_frequency_window_factor,
(100.0*relevant_roots_since_time_zero)
/ roots_since_time_zero);

} else {
 CDEBUG("time=%4.1f (%06d); freq=%8.4f; spr=%12.8f; "
"theta=%8.4f; freqw=%0.5f; relevance=%0.1f%% "
"[lag-enabled] \n",
seconds, samples_since_time_zero, freq, spr, theta,
relevant_frequency_window_factor,
(100.0*relevant_roots_since_time_zero)
/ roots_since_time_zero);

}

spam_time = new_time;

consume_each(number);

return number;

</wmu_root_sync.xml>

<?xml version="1.0"?>

<!-- this source defines the GNU Radio Companion interface
 that pulls WMU Rootsync into the cad software -->

<block>
  <name>WMU RootSync</name>
  <key>wmu_root_sync_ccc</key>
  <category>WMU</category>
  <import>import wmu</import>
  <make>wmu.root_sync_ccc(
    alpha=$alpha, beta=$beta, gamma=$gamma, theta=$theta,
    rfreqw=$rfreqw, laglen=$laglen, lagtlen=$lagtlen,
    samp_rate=$samp_rate, ref_freq=$ref_freq)
  </make>
  <callback>set_alpha($alpha)</callback>
  <callback>set_beta($beta)</callback>
  <callback>set_gamma($gamma)</callback>
  <callback>set_theta($theta)</callback>
  <callback>set_rate($samp_rate)</callback>
</block>
<callback>set_freq($ref_freq)</callback>

<param>
  <name></name>
  <key>alpha</key>
  <value>0.7</value>
  <type>raw</type>
</param>

<param>
  <name></name>
  <key>beta</key>
  <value>0.001</value>
  <type>raw</type>
</param>

<param>
  <name></name>
  <key>gamma</key>
  <value>0.1</value>
  <type>raw</type>
</param>

<param>
  <name></name>
  <key>theta</key>
  <value>0</value>
  <type>raw</type>
</param>

<param>
  <name>Sample Rate</name>
  <key>samp_rate</key>
  <value>samp_rate</value>
  <type>raw</type>
</param>

<param>
  <name>Ref Frequency</name>
  <key>ref_freq</key>
  <value>freq</value>
  <type>raw</type>
</param>

<param>
  <name>Frequency Laguerre Length</name>
  <key>laglen</key>
  <value>500</value>
  <type>raw</type>
</param>

<param>
  <name>Laguerre Training Roots</name>
  <key>lagtlen</key>
  <value>0</value>
  <type>raw</type>
</param>

<param>
  <name>Relevant Frequency Window</name>
  <key>rfreqw</key>
  <value>0.01</value>
  <type>raw</type>
</param>

<sink>
  <name>sig</name>
  <type>complex</type>
</sink>

<source>
  <name>ref*</name>
</source>
alpha is the weight of the moving average: \( \text{drad} = \text{drad} \times (1 - \text{alpha}) + \text{drad} \times \text{alpha} \). \( \text{drad} \) is literally the difference (in radians) between signal and ref*.

beta is the minimum adjustment. If \( \text{gamma} \times \text{drad} \geq \text{beta} \), then make an adjustment.

gamma is the adjustment attenuation. Doing a small portion of an adjustment can damp adjustment oscillation.

The initial phase angle theta should normally be 0.

Laguerre length refers to the number of "days" in the Ehler/Laguerre filter. If Lag Stages are greater than theta, then the lag filter will reset to longer "days" settings and finally disable frequency training.