UPC - NASA JPL

Bachelor Thesis

Mathematics and Telecommunications Engineering

Development and Implementation of an Adaptive-Sweep Algorithm for Carrier Acquisition and Tracking in Spacecraft Radios

Author
Tomàs Ortega

Supervisor
Dr. Marc Sánchez

Tutor
Dr. Olga Muñoz

Co-Supervisor
Dr. Kar-Ming Cheung

July 2020

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the JPL Visiting Student Research Program and the National Aeronautics and Space Administration

This page is intentionally left blank

Abstract

Spacecraft radios transmit a signal modulated at a carrier frequency, which is later demodulated and processed at baseband. However, this carrier frequency shifts over time due to the Doppler effect, which must be removed prior to demodulation by acquiring and tracking the carrier. An algorithm to do so is already implemented in spacecraft radios developed by NASA [5]. However, it uses a fixed-step sweep algorithm to acquire the carrier, a limitation that facilitates implementing it in flight hardware, but makes the acquisition process brittle. In this thesis, a novel carrier acquisition algorithm that uses an adaptive-step is presented and its performance is quantified in the presence of Additive White Gaussian Noise (AWGN). Additionally, a software implementation using GNU Radio is also presented and tested in simulation for a wide variety of Signal to Noise Ratio (SNR) conditions. Finally, results presented in this work demonstrate the ability of the algorithm to acquire and track the carrier of the Lunar Reconnaissance Orbiter (LRO).

Keywords Signal theory, carrier acquisition, sweeping algorithm
AMS code 94A99

Acknowledgements

First of all, I want to express my sincere gratitude to my supervisor at JPL, Dr. Marc Sánchez for the attention, patience, and help he has given me throughout this thesis.

I would also like to acknowledge Dr. Kar-Ming Cheung and Dr. Dariush Divsalar, who have followed my work during these six months, and given me their input and guidance.

To my tutor from UPC, Dr. Olga Muñoz, thank you for fueling my passion for telecommunications since the first college course.

I would like to take a moment to recognize JPL and the CFIS program, who have given me the opportunity to work on such an interesting problem.

Last but not least, I would like to thank my family and friends, both at JPL and abroad, for their tremendous support during these unconventional six months.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the JPL Visiting Student Research Program and the National Aeronautics and Space Administration (80NM0018D0004).

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

NASA operates a variety of spacecraft, including rovers, satellites, instruments, and other devices. For all interplanetary missions, and a few in Earth vicinity, NASA uses the Deep Space Network (DSN) to communicate with said spacecraft. The DSN also provides the means to command, track and monitor the health and safety of those missions.

According to NASA, the DSN is the largest and most sensitive scientific telecommunications system in the world [3]. It is equipped with an array of giant antennas, in three facilities strategically located around the globe to allow constant communication with spacecraft. The DSN is monitored and operated by NASA's Jet Propulsion Laboratory (JPL) and its current state can be consulted at eyes.nasa.gov/dsn.

When spacecraft communicate with Earth, they transmit at a predetermined carrier frequency. However, due to the Doppler effect, the received signal on Earth has suffered a frequency shift. For example, Figure 1.1 plots the received signal at the Canberra DSN station when the Lunar Reconnaissance Orbiter (LRO) transmits a carrier tone. Also, no signal is received during the periods of occultation, that is, when the Moon is between the spacecraft and Earth.

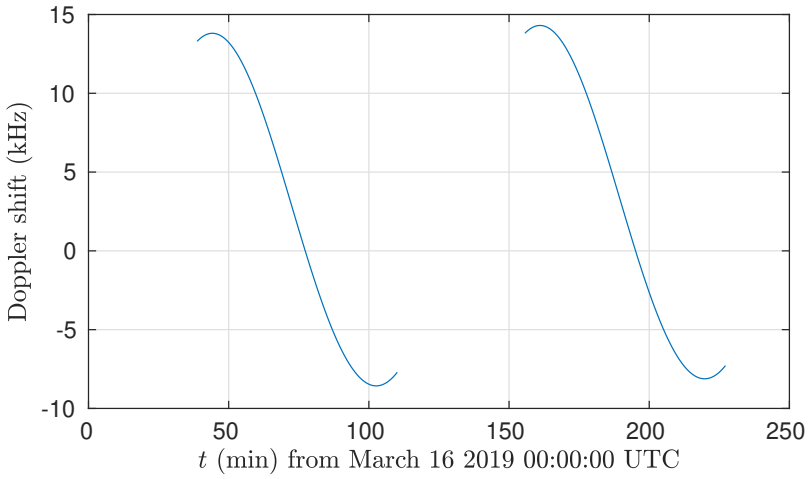

Figure 1.1: Doppler shift received in the Canberra DSN station

The frequency shift induced by the Doppler effect depends on the relative motion

of the spacecraft with respect to Earth. Each mission provides the DSN with Doppler shift predictions that allow the frequency shift to be corrected on Earth. When the ground station uplinks data to the spacecraft, the Doppler shift is compensated at the Earth node. This way, the spacecraft can transmit and receive at a given frequency ignoring the Doppler effect.

However, consider the case of a communications link between a rover and a satellite, also known as a proximity link. One of the two communications terminals must correct the Doppler shift. Since a precise location of the receiver is not necessarily known by the transmitter, the correction is done at the receiver.

Spacecraft radio that must support proximity links are already equipped with technology to perform this correction, using a fixed-step frequency sweep algorithm. This thesis explains how this technology works, and explores a possible improvement on this technology, in which the frequency sweep is enhanced with an adaptive step.

A spacecraft radio, like Iris or Electra, performs the carrier acquisition and tracking digitally, with the fixed-step algorithm implemented in an Field Programmable Gate Array (FPGA) [5]. This can be accurately simulated with a Software-Defined Radio (SDR), which is a radio communication system that is primarily implemented by means of software [9]. In a SDR, the input signal (which originates either from software or hardware) is processed by software run on a general purpose CPU or embedded system. This facilitates the development process of new technologies, such as the proposed adaptive-sweep algorithm, by allowing the implementation of components like mixers, filters or amplifiers directly in software.

1.2 Motivation

Current technology for carrier acquisition in spacecraft radios for proximity links consists in a fixed-step frequency sweep algorithm. This approach to solve the problem is intuitive, but lacks robustness. Theoretical research developed recently at JPL suggests that that a faster, more robust approach could be achieved using an adaptive-sweep algorithm [2].

There are at least three motivating applications for an improved carrier acquisition and tracking algorithm: Proximity links at Mars or the Moon, support of Multiple Uplinks per Antenna (MUPA) in the DSN, and generation of Doppler measurements for navigation purposes.

Proximity links at Mars or the Moon require a fast and robust method to acquire and track the carrier frequency in order to establish communications, since the Doppler shift cannot be predicted by the transmitter or the receiver because they do not know their relative motion. Instances of noise or high Doppler dynamics might trigger loss of lock at the receiver, and hence loss of communications. Carrier acquisition is therefore paramount in order to minimize the time without communications.

Having an advanced acquisition and tracking system would also facilitate the development of MUPA. Currently, a DSN antenna uplinks data to a spacecraft by pre-compensating the sent signal so that the spacecraft receiver operates at its Best-Lock Frequency (BLF). If multiple spacecraft with different trajectories share the same uplink, the DSN station cannot pre-compensate the Doppler for all of them at once. However, if spacecraft could acquire and track the frequency shift with sufficient precision, the correction of different Doppler shifts could be done at

reception. This way, the Earth antenna could uplink data to several spacecraft at once.

A third possible application of this system are navigation systems in planetary bodies where GPS or weak-GPS signals are not available. Since the Doppler shift is a function of the direction of motion and relative velocity between transmitter and receiver, knowing the evolution of the Doppler shift over time could allow the navigation system to infer its current position. Therefore, an advanced acquisition and tracking system such as the one proposed in this thesis could be used to produced the required observables for the navigation system.

The main purpose of this thesis is to refine and mature an adaptive sweep algorithm by studying its performance in the presence of Additive White Gaussian Noise (AWGN), and implementing it in a SDR environment. Successful demonstration of the algorithm in the SDR will make a strong case for hardware-based testing and, ultimately, implementation in flight radios such as IRIS or Electra.

1.3 Literature review

In order to understand the current system for carrier acquisition and tracking, it is necessary to introduce some core concepts and definitions. These include the definition of the Doppler shift, the model of the signal, the essentials of a Phase-Locked Loop (PLL) for carrier tracking and an introduction to a carrier acquisition sweeping algorithm.

The first concept to introduce is the Doppler shift. Spacecraft radios transmit a signal modulated at a carrier frequency, which is later demodulated and processed at baseband. However, the Doppler effect introduces a phase shift on the received signal that must be compensated in order to recover the modulated symbols.

The Doppler shift is defined as the difference between the received and the transmitted frequency,

$$f_d(t) = f_{Rx}(t) - f_{Tx}(t)$$

This shift, along with changes in frequency due to instabilities in oscillators, adds a time-varying distortion in the phase of the received signal. It is important to note that the frequency is the time derivative of the phase, that is, $\dot{\phi}(t) = \omega(t) = 2\pi f(t)$. The full signal model will be presented in subsection 1.3.1.

A Phase-Locked Loop (PLL) is a system designed to generate an output signal whose phase $\hat{\phi}(t)$ is as similar as possible to the phase $\phi(t)$ of the input signal. Therefore, variations of carrier frequency due to Doppler effects or instabilities of the oscillators can be tracked using a PLL, which enables proper demodulation of the received signal. The essentials of a PLL will be presented in subsection 1.3.2.

The drawback of using only a PLL is that it cannot operate when the difference between the received frequency and the transmitted one is too large. In particular, the first acquisition of the carrier frequency must be done by a separate system, this is where the Sweeping Algorithm comes into play. The way the Sweeping Algorithm works is explained in subsection 1.3.3.

1.3.1 Signal model

This thesis assumes that the transmitted signal is a Binary Phase Shift Keying (BPSK) signal with residual carrier, as is common in deep space applications [7].

The complex baseband equivalent of the received signal is [2]

$$r(t) = \sqrt{P_t} e^{j\left[\phi(t)+\beta m(t) g(t)\right]} + n(t), \tag{1.1}$$

$$\phi(t) = 2\pi \int_{-\infty}^{t} f_d(\tau)\, \mathrm{d}\tau + \theta_c \tag{1.2}$$

where P_t is the total received carrier and data power, $m(t)$ is the BPSK modulated signal, θ_c is the carrier phase and $f_d(t)$ is the Doppler frequency shift. Data is modulated onto a subcarrier using, for example, a square waveform $g(t) = \text{sign}(2\pi f_{sc} t)$, with a modulation index $\beta < \pi/2$. Finally, $n(t)$ is a complex Additive White Gaussian Noise (AWGN) with two-sided power spectral density $N_0/2$.

When acquiring and tracking the carrier, the sub-carrier components are filtered out. The remaining complex baseband equivalent of the carrier signal is

$$s(t) = \sqrt{P_c} e^{j\phi(t)} + n(t), \tag{1.3}$$

where $\phi(t)$ is the same as in Equation 1.2. Assuming a square wave subcarrier, $P_c = P_t \cos^2(\beta)$ is the power in the carrier (unmodulated) component. The rest of the power, $P_d = P_t \sin^2(\beta)$ corresponds to the data (modulated) component, that has been filtered out.

Assuming that the carrier power can be estimated, the input signal is normalized, and Equation 1.3 becomes

$$s(t) = e^{j\phi(t)} + n(t), \tag{1.4}$$

where $n(t)$ is now AWGN of variance $\frac{F_s}{P_c/N_0}$, and F_s is the sample frequency. The P_c/N_0 parameter, called carrier to noise ratio, is usually estimated by NASA missions using the link budget equation.

The final aspect of the signal model that must be explored is the Doppler shift. An extended study on the Doppler shift for the link between the LRO and the DSN can be found in Appendix A. For this work, the Doppler shift that will be considered is modelled as a ramp,

$$f_d(t) = f_0 + rt, \tag{1.5}$$

where f_0 is the Doppler offset, sometimes referred to simply as the Doppler of the signal, and r is the Doppler rate.

The Doppler shift model in Equation 1.5 is appropriate because the first order Taylor approximation in each point of Figure 1.1 is locally (within an interval of a minute) very accurate.

1.3.2 Phase-Locked Loop

This section introduces the fundamentals of a PLL. First, the PLL will be described in a block diagram, then the elements and signals present in the diagram will be specified. Finally, the PLL used in this work will be detailed, as well as the condition that relates analog and digital PLLs.

A basic PLL has the block structure detailed in Figure 1.2. To facilitate the analysis, an ideal scenario with no noise is initially considered; therefore the input signal can be written as $e^{j\phi(t)}$, as described in Equation 1.4. Similarly, the phase estimated by the PLL is denoted by $\hat{\phi}(t)$. The received and estimated phases are first subtracted using a mixer, and then fed into a phase extractor. Then, the phase

is passed through a filter that estimates the phase derivative, or angular frequency. This is then fed to a Numerically Controlled Oscillator (NCO) that outputs the conjugate of the estimated phase to start the loop again.

It is important to remember the definition of angular frequency, which is $\omega = 2\pi f$. Therefore, an estimate of the Doppler frequency can be obtained from the output of the PLL. Additionally, when the estimated frequency is the Doppler frequency, or the absolute value of their difference is below a given tolerance, the PLL is said to be in lock.

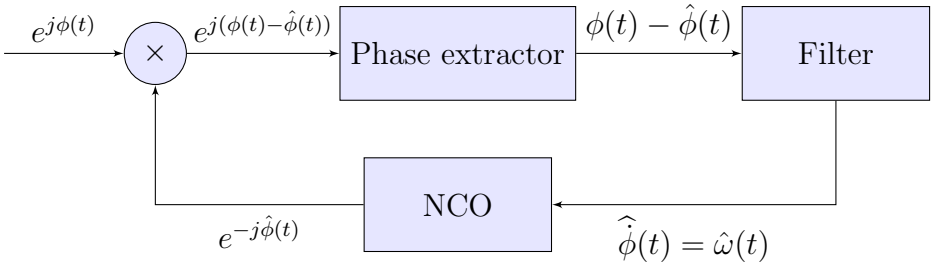

Figure 1.2: PLL block diagram

The first element after the mixer is the phase extractor. To extract the phase of the mixed signal, the imaginary operator is used. Assuming that the received and estimated phases are close, the small angle approximation of the sine is used to obtain

$$\Im\left\{e^{j(\phi(t)-\hat{\phi}(t))}\right\} = \sin\left(\phi(t) - \hat{\phi}(t)\right) \approx \phi(t) - \hat{\phi}(t) \tag{1.6}$$

This approximation allows the linear analysis of the circuit, and also illustrates the need for a good estimation of the initial frequency offset.

After the phase extractor, the next element in Figure 1.2 is the PLL filter, which is Infinite Impulse Response (IIR) filter. A PLL is said to be of order N when its filter is an IIR of order N. A filter of order N is designed to track dynamics up to the Nth order. In other words, a first order PLL will only be able to track dynamics up to the first order, i.e. $f_d(t) = f_0$ a constant offset. A second order PLL will track Doppler shifts of the form $f_d(t) = f_0 + rt$.

The PLL Filter is designed to be a lowpass filter with unitary gain and bandwidth

$$B_L = \frac{1}{2}\int_{-\infty}^{\infty} |H(j2\pi f)|^2 \,\mathrm{d}f \tag{1.7}$$

where $H(z)$ represents the closed-loop transfer function defined by

$$H(z) = \frac{\hat{\phi}(z)}{\phi(z)}$$

and for a second order PLL is [8]

$$H(z) = \frac{(K_1 + K_2)z - K_1}{z^2 + (K_1 + K_2 - 2)z + 1 - K_1} \tag{1.8}$$

However, there is a more intuitive way to see this. Given a $f_d(t)$ Doppler frequency shift in time, B_L is the threshold of the Fourier transform of $\phi(t)$ that the PLL

cancels. If $f_d(t) = \sin(2\pi\nu t)$, then signals with $\nu > B_L$ should be cancelled (experimental results on this topic can be found in subsection 3.2.1). The trade-off is that a lower B_L will offer better noise rejection, while a higher B_L will allow higher Doppler dynamics.

The filter used in the final system uses the coefficients described in [8] for a second order critically damped filter (current spacecraft radio like Electra also use a second order filter [5]). Both the under-damped and the critically damped filter responses can be found in Figure 1.3. As they are second order filters, the responses are far from an ideal rectangular-shaped filter.

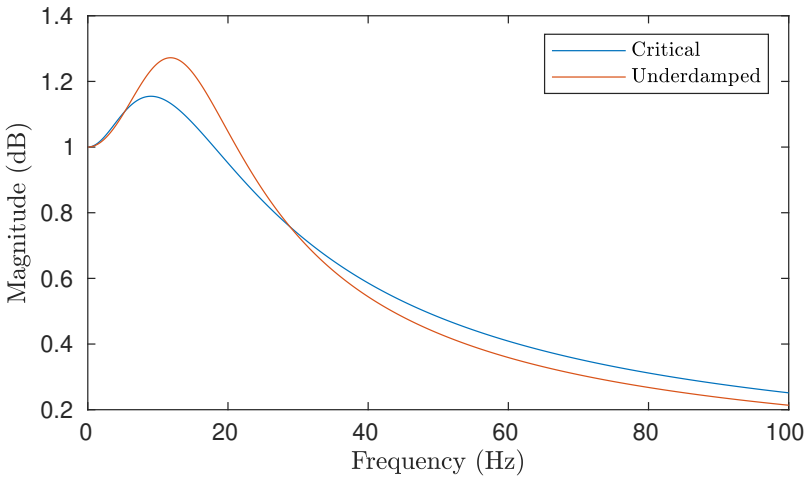

Figure 1.3: PLL filter response with $B_L = 50Hz$

In practice, PLL are implemented digitally. However, for a high sample rate, the behaviour is the same as an analog PLL [8]. The condition that must be satisfied for the digital PLL to behave like its analog counterpart is called the Continuous Update (CU) approximation, and is $B_L/F_{PLL} \leq 0.02$, where B_L is the PLL bandwidth and F_{PLL} is the PLL update rate [8].

The schematic in Figure 1.4 is a basic PLL Simulink model. An interesting element to point out is the integrate and dump block, which has a length of N samples and a $1/N$ gain block for averaging afterwards. This helps to reduce the noise; however, it also reduces the PLL update rate F_{PLL} by

$$F_{PLL} = \frac{F_s}{N} \tag{1.9}$$

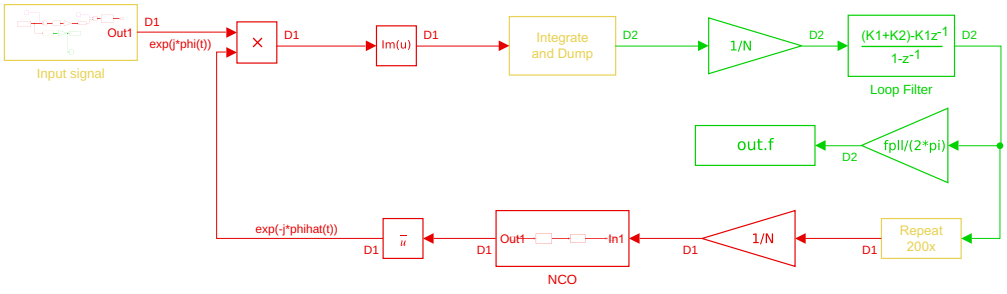

Figure 1.4: PLL Simulink design schematic

1.3.3 Sweeping Algorithm

The Sweeping Algorithm is in charge of correcting large variations in phase that cannot be tracked by the PLL because they exceed its bandwidth. In particular, it corrects the initial offset f_0 and thus help the PLL lock. It operates at a lower frequency than the PLL to allow the PLL to lock onto different candidate offsets. A block diagram of the complete system is illustrated in Figure 1.5.

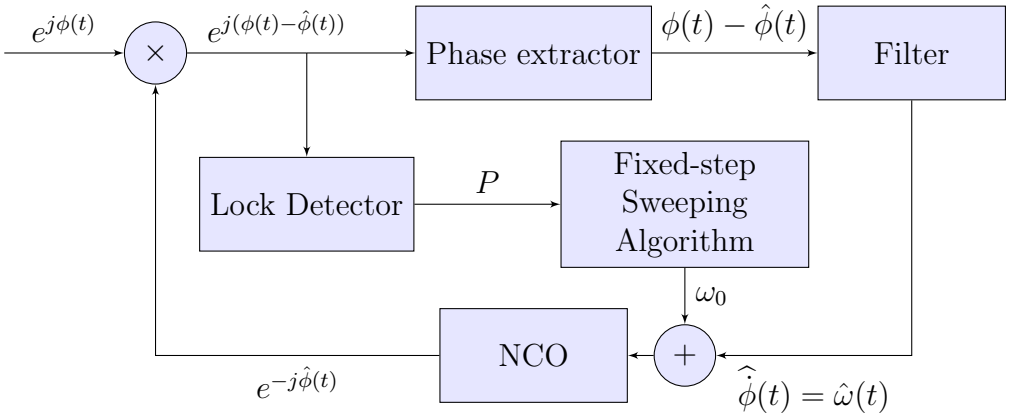

Figure 1.5: PLL and fixed-step frequency sweep block diagram

A crucial part of the Sweeping Algorithm is the Lock Detector. The Lock Detector is a system that, when fed a unitary amplitude input signal, outputs a function of the phase that is maximal when the phase is null. Ideally, this would be a binary block that outputs 1 when the phase is locked and 0 when it is not. In reality, it outputs a value between 1 and 0, and this value is later compared to a threshold to simulate the ideal behaviour.

The algorithm that is currently implemented in spacecraft radio is fairly straightforward [5], and is illustrated in Figure 1.6. The sweeping algorithm makes a sweep between an initial frequency and a final frequency with a fixed step jump. In each step, the conjugate of the estimated signal is fed into the mixer, along with the input signal, and the mixer result is fed into the Lock Detector. If the output of the Lock Detector (denoted P in Figure 1.2) is greater than a given threshold, the sweeping algorithm stops and lets the PLL continue tracking. If the final frequency is reached without having surpassed the threshold, the algorithm restarts.

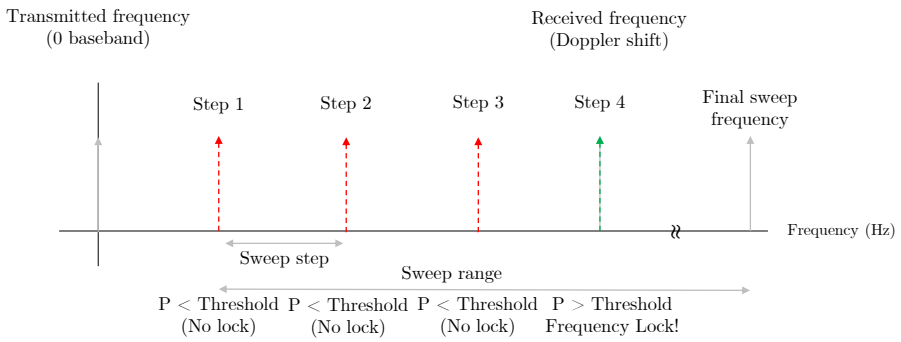

Figure 1.6: Fixed-step sweep algorithm scheme

The adaptive-sweep algorithm proposed in [2] is slightly more sophisticated. It

also sweeps between an initial and a final frequency. However, it has a Lock and Direction Detector that outputs a threshold and direction function, denoted P and D respectively. The sweep step and the direction of the step are determined by the value of P and D, instead of being fixed. A block diagram of the proposed system is illustrated in Figure 1.7.

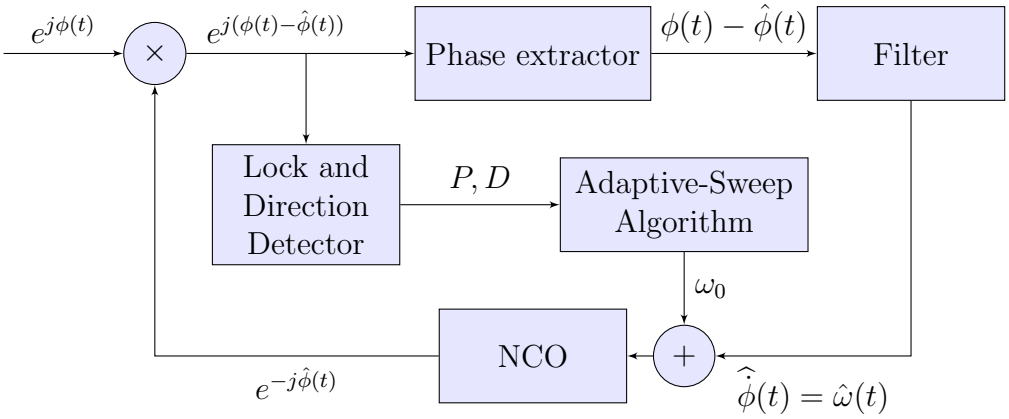

Figure 1.7: Block diagram of the proposed system

Ideally, the D function would be a ramp that depended on $\Delta_f = f - \hat{f}$, where f is the input signal offset and $\hat{f}$ is the current offset estimate. This would allow the system to simply add the value of D to the current estimate to achieve frequency lock in one step. In reality, the D function only approximates such a ramp in the vicinity of the frequency lock. Therefore, the P function is used to detect when the estimated frequency is near the frequency lock, thus enabling the use of D.

The previously mentioned high-level description of the algorithm is illustrated in a block scheme in Figure 1.8. There is a threshold θ, that dictates when to use the fixed step Δ_1 or the adaptive step $D\Delta_2$. The initial and final frequencies of the sweep are f_1 and f_2, respectively.

To understand the behaviour of the P and D functions the Lock and Direction Detector must be studied. A detailed analysis can be found in [2], so only a simplified version is provided in this thesis. Suppose that the frequency of the input signal has an offset f, but the PLL estimates a frequency $\hat{f}$ such that $f \neq \hat{f}$. In the absence of noise, the input of the Lock Detector is

$$x_i(t) = e^{j(2\pi\Delta f t + \Delta\phi)} \tag{1.10}$$

where $\Delta f = f - \hat{f}$ and $\Delta\phi = \phi - \hat{\phi}$. Note that both are approximated as constant in time. Then, the following operations are performed:

$$\begin{aligned} y_k &= \frac{1}{T}\int_{T(k-1)}^{Tk} x_i(t)dt = \frac{e^{j(2\pi\Delta fTk+\Delta\phi)}(1-e^{-j2\pi\Delta fT})}{j2\pi\Delta fT} \\ &= \frac{e^{j(2\pi\Delta fTk+\Delta\phi)}(e^{j\pi\Delta fT}-e^{-j\pi\Delta fT})e^{-j\pi\Delta fT}}{j2\pi\Delta fT} \\ &= e^{j(2\pi\Delta f(k-\frac{1}{2})T+\Delta\phi)}\,\mathrm{sinc}(\pi\Delta fT) \end{aligned} \tag{1.11}$$

$$y_{k-1}^* y_k = e^{j(2\pi\Delta fT)}\,\mathrm{sinc}^2(\pi\Delta fT) \tag{1.12}$$

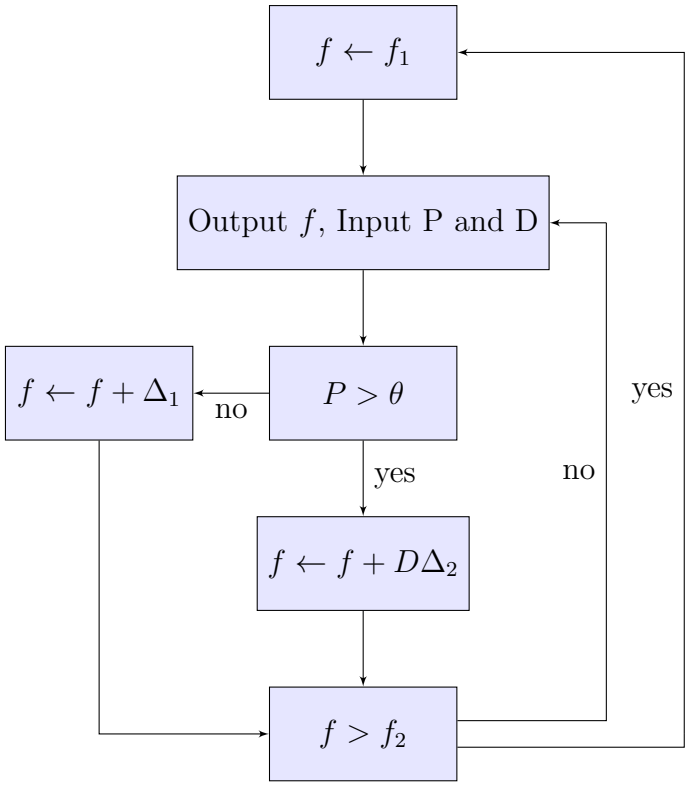

Figure 1.8: Block diagram of the adaptive-sweep algorithm

This results in:

$$P = \Re\left\{y_{k-1}^* y_k\right\} = \cos(2\pi\Delta f T)\,\mathrm{sinc}^2(\pi\Delta f T) \tag{1.13}$$

$$D = \Im\left\{y_{k-1}^* y_k\right\} = \sin(2\pi\Delta f T)\,\mathrm{sinc}^2(\pi\Delta f T) \tag{1.14}$$

These functions are plotted in Figure 1.9. Observe that the behaviour of P successfully approximates an ideal lock detector: when Δf tends to zero, the function has a maximum, which is equal to one. Around $\Delta f = 0$ the function D also behaves as desired, since it approximates a ramp that is a function of Δf. A version of Figure 1.9 with noise can be found in section 3.3. It is interesting to observe that for high noise levels, the D function might introduce a high variance into the estimated frequency. In this scenario, D can be fed to a possibly non-linear function $q(D)$, such as a clipper, to ensure the output of $q(D)$ is bounded between $[-1, 1]$. That being said, this will not be taken into consideration in this thesis.

In practice, the Lock and Direction Detector block is implemented in the discrete domain. Reference [2] shows nuances related to this change, but in this work the sampling frequency will be assumed high enough to behave as the analog version.

The Simulink block schematic of the discrete Lock and Direction Detector block can be seen in Figure 1.10. The integrator of length T to calculate y_k in Equation 1.11 has been replaced by its discrete counterpart, an integrate and dump block. Its sample length M is chosen such that, given an input sampling frequency F_s,

$$T = MT_s = \frac{M}{F_s} \tag{1.15}$$

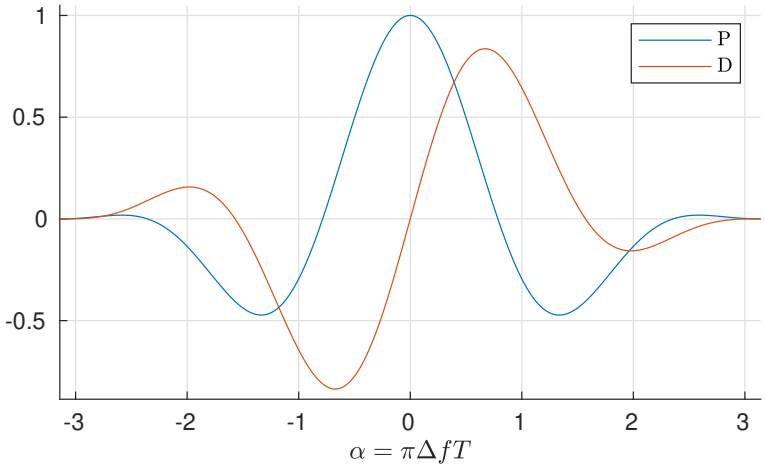

Figure 1.9: P and D as a function of $\pi\Delta fT$

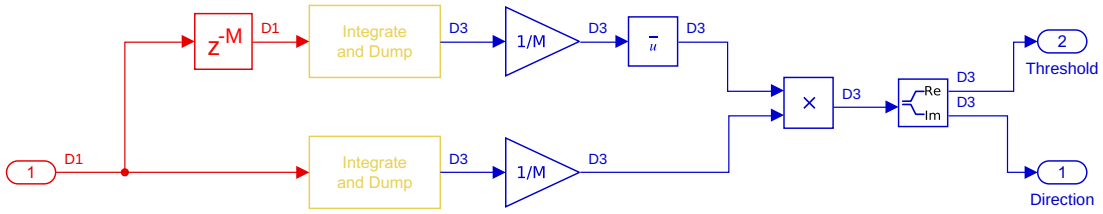

Figure 1.10: Lock and Direction Detector Simulink block

Several implementations of the adaptive-sweep algorithm have been developed in this thesis, see subsection 3.2.2 for a Simulink implementation and section 2.3 for a GNU Radio implementation.

1.4 Thesis statement

Given the adaptive sweep algorithm for carrier acquisition and tracking from [2], explained in the literature review, the goals of this thesis are:

1. Modify the adaptive-sweep algorithm from [2] to use an automatic adaptive step.

2. Characterize the performance of the system in the presence of AWGN.

3. Derive the design equations of a carrier acquisition and tracking system that combines a PLL with an Adaptive-Sweep Algorithm (ASA).

4. Implement the system in a SDR and port it to a stand-alone platform.

5. Test the algorithm in both a simulated environment and using received signals from LRO.

1.5 Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2 provides a system analysis and performance metrics as a function of noise and Doppler characteristics, which are then used to derive the design equations of a carrier acquisition and tracking system that combines a PLL with an Adaptive-Sweep Algorithm. It also describes the characteristics of the chosen SDR software and peripherals.

Chapter 3 contains the results obtained from the previously described work. First, a script that calculates the system parameters using the design equations is detailed. Next, results obtained with simulated signals running on Matlab Simulink and SDR models are compared. Finally, experimental results obtained with LRO signals running on a stand-alone SDR platform are presented.

Chapter 4 summarizes the thesis and its main contributions, and introduces the possible future opportunities for research.

Chapter 2

Analysis and implementation

In Chapter 1, most of the discussion assumed an ideal noiseless scenario. However, in order to quantify the performance of the system in representative flight conditions, the noise must be accurately modelled. In this chapter, the analysis of a combined PLL and adaptive-sweep system in the presence of AWGN is presented. First, section 2.1 defines several Signal to Noise Ratios that will be useful for the system analysis. Next, section 2.2 presents the system analysis and performance metrics as a function of noise and Doppler characteristics, which are then used to calculate system parameters such as the threshold for the P function, and the size of the fixed and adaptive steps. Finally, section 2.3 gives an overview of the SDR software that has been chosen for the proposed ASA, as well as the peripherals used for experimenting.

2.1 Signal to Noise Ratios

Signal to Noise Ratios can cause confusion when analyzing PLLs because multiple SNRs are of interest. Before starting the SNR definitions, it is important to recall the system model described in subsection 1.3.1. The input signal to the system, described in Equation 1.4, is

$$s(t) = e^{j\phi(t)} + n(t)$$

where $n(t)$ is AWGN of variance $\frac{F_s}{P_c/N_0}$, and F_s is the sample frequency.

Once the input signal is known, the carrier SNR can be defined. Given a sample rate F_s, and P_c/N_0, usually calculated using the link budget equation, the carrier SNR, or SNR_c, can be calculated as

$$SNR_c = \frac{P_c}{N_0 F_s}$$

The carrier SNR allows the sweeping branch SNR to be defined. This is the SNR that governs the behaviour of the sweeping algorithm, and is denoted by SNR_M. Recall here that M denotes the length of the integrate and dump block used to generate the Lock and Direction functions (see subsection 1.3.3 and Figure 1.10). It can be computed as

$$SNR_M = SNR_c \cdot M,$$

which clearly indicates that larger values of M help reduce the variance of the noise. The latter is of great importance, as a longer integrator will reduce the noise better,

but will also increase the delay between consecutive samples of P and D, therefore making the sweep algorithm slower. As an example, Figure 2.1 presents the P and D functions in the presence of noise with $SNR_M = 20$ dB, and has been calculated using the implementation detailed in section 2.3.

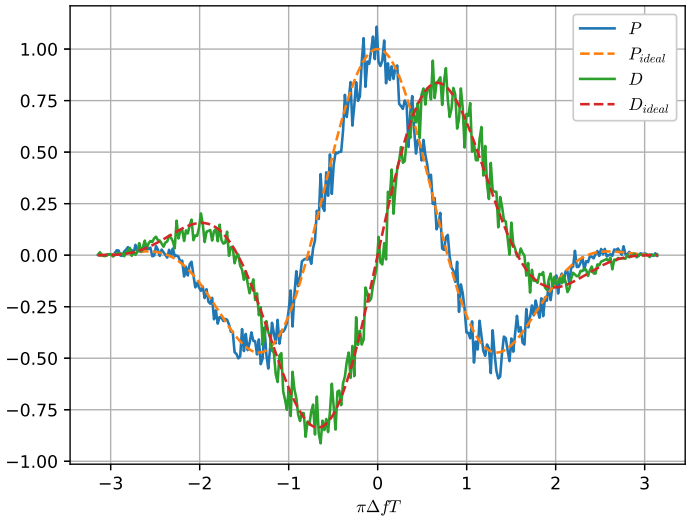

Figure 2.1: P and D with noise $SNR_M = 20dB$

Finally, another interesting SNR is the PLL SNR, usually denoted by ρ or SNR_{PLL}, that depends on the loop bandwidth B_L and is computed as

$$\rho = \frac{P_c}{N_0 B_L}$$

This parameter is used in the DSN Telecommunications Link Design Handbook, which recommends that $\rho > 10$ dB for the PLL to track a signal with data modulated in a residual carrier [4].

2.2 Parameter calculation

The SNR definitions from section 2.1 allow the system analysis that will be carried out in this section. The current system has the Degrees Of Freedom (DOF) presented in Table 2.1, which are currently chosen ad-hoc for every proximity link scenario. The goal of the system analysis is to provide the design equations that will make the current system completely determined, based on a few requirements: the range of expected Doppler offsets, the minimum expected P_c/N_0, and the maximum expected Doppler rate.

The first part of the system analysis will produce two equations that govern the PLL and Sweeping Algorithm update rates. Afterwards, the steady-state phase error will be deduced, giving a design equation for the PLL bandwidth. Subsequently, the noise distribution will be studied. This will provide a design equation for the sweep threshold, and the necessary background for calculating the adaptive and fixed step size. In particular, three design equations, one for each step (fixed and adaptive), and one for the minimum sweep branch integrator sample size will be

DOF	Description
F_s	Sampling frequency
B_L	PLL bandwidth
N	Length of PLL integrator
M	Length of the integrator to calculate the P and D functions
θ	Sweep threshold
Δ_1	Fixed sweep step
Δ_2	Adaptive sweep step

Table 2.1: Initial DOF with their description

derived. Finally, the expected acquisition time and the expected output frequency error standard deviation will be calculated. This will give an idea of the system performance without having to run simulations.

2.2.1 PLL and Sweeping Algorithm update rates

The first condition that governs the PLL update rate is the aforementioned Continuous Update approximation, which following the work in [8] is

$$B_L T_{PLL} = B_L \frac{N}{F_s} \leq 0.02 \tag{2.1}$$

where N is the size of the integrate and dump block in the PLL (see Figure 1.4).

Also, as was explained in subsection 1.3.3, the update rate of the PLL has to be faster than the update rate of the sweeping algorithm to allow the PLL to lock on to different candidate offsets. A 10 factor has been proven to work experimentally. Given a sampling frequency F_s, a PLL branch integrator size N, and a sweep branch integrator size M (see Figure 1.10), the following condition is derived.

$$10\frac{N}{F_s} \leq \frac{M}{F_s} \implies 10N \leq M \tag{2.2}$$

2.2.2 Steady-state phase error

To calculate the loop bandwidth, one must take into account the steady-state phase error, that is, the difference between the output phase and the input phase when the PLL is in lock.

Given an input phase ϕ, the results in [8] show that the steady-state phase error $\tilde{\phi}_{ss} = \phi_n - \hat{\phi}_{n-1}$ can be assumed constant. Also, for a second order filter of constants K_1 and K_2, and an inverse PLL update rate T_{PLL}, the formula for the steady-state phase error is

$$\tilde{\phi}_{ss} = \frac{T_{PLL}^2}{K_2}\phi''$$

Since a filter with a critically damped response is used,

$$K_2 = \frac{1}{4}K_1^2 = \frac{1}{4}\left(\frac{16}{5}B_L T_{PLL}\right)^2 \implies \phi'' = B_L^2 \frac{\tilde{\phi}_{ss} 4^3}{5^2}$$

Knowing that the input signal has a phase $\phi(t) = 2\pi \int_{-\infty}^{t} f_d(\tau)\,\mathrm{d}\tau + \theta_c$ (see Equation 1.2), and that the Doppler shift is modelled as $f_d(t) = f_0 + rt$ (see Equation 1.5), then

$$\phi'' = 2\pi f_d'(t) = 2\pi r \implies r = B_L^2 \frac{\tilde{\phi}_{ss} 4^3}{2\pi 5^2}$$

Therefore, if the desired spacecraft radio is designed for $\tilde{\phi}_{ss} \leq 0.1$ rad, typical in spacecraft applications [4], then the previous formula yields

$$r \leq B_L^2 \frac{0.1 \cdot 4^3}{2\pi 5^2}$$

In other words, there is direct quadratic relationship between maximum supportable Doppler rate and PLL loop bandwidth (see Figure 2.2). Analogously, this means that given a maximum expected Doppler rate, one can obtain the required loop bandwidth as

$$\sqrt{\frac{2\pi 5^2 r}{4^3 \tilde{\phi}_{ss}}} \leq B_L \tag{2.3}$$

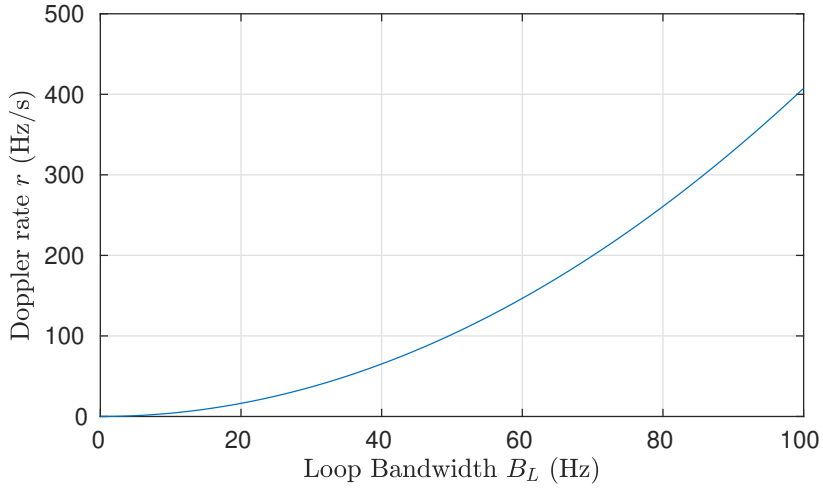

Figure 2.2: Maximum supported r depending on B_L

2.2.3 Noise distribution

After obtaining a design equation for the PLL bandwidth, the noise distribution will be studied in order to obtain a sweep threshold θ, and a sweep branch integrator of size M.

To begin the analysis, the noise variance of the P and D functions is computed as a function of the SNR conditions. To gain some insight, the noise PDF will first be numerically calculated for several values of SNR_M. This will later be used to compute error bounds for the P and D functions, which will yield the lower bound on M. Finally, the analysis will be validated using an improved Chebyshev's inequality.

To compute the noise variance for the P and D functions, the system's input noise $n_i(t)$ is modelled as complex AWGN of variance σ_i^2. After an analogous procedure

to Equation 1.11, the following equations are obtained:

$$\begin{aligned} y_k &= x_k + n_k, \\ x_k &= \frac{1}{T}\int_{(k-1)T}^{kT} x_i(t) = e^{j(2\pi\Delta f(k-\frac{1}{2})T+\Delta\phi)}\,\mathrm{sinc}(\pi\Delta fT), \\ n_k &= \frac{1}{T}\int_{(k-1)T}^{kT} n_i(t), \end{aligned}$$

where T is the lock detector integration period $T = MT_s = M/F_s$, n_k is complex AWGN with variance σ_i^2/T. Also, P and its expected value become

$$P = \Re\left\{y_{k-1}^* y_k\right\} = \Re\left\{x_{k-1}^* x_k + x_{k-1}^* n_k + n_{k-1}^* x_k + n_{k-1}^* n_k\right\} \tag{2.4}$$

$$\mathbb{E}[P] = \Re\left\{x_{k-1}^* x_k\right\} = \cos(2\pi\Delta fT)\,\mathrm{sinc}^2(\pi\Delta fT) = \cos(2\alpha)\,\mathrm{sinc}^2(\alpha) \tag{2.5}$$

where the variable $\alpha = \pi\Delta fT$ is defined in order to ease notation.

Using Equation 2.4, the variance of P can now be calculated. It is important to note that all noise terms have zero mean and zero pairwise covariance, and therefore, the variance of the sum of noise terms is the sum of the individual noise term variances. Using these arguments it follows that

$$\sigma_P^2 = \mathrm{Var}\left(\Re\left\{x_{k-1}^* n_k + n_{k-1}^* x_k + n_{k-1}^* n_k\right\}\right) = \frac{\sigma_i^2}{T}\,\mathrm{sinc}^2(\alpha) + \frac{1}{2}\left(\frac{\sigma_i^2}{T}\right)^2$$

This procedure can be repeated for D, which yields values for $\mathbb{E}[D]$ and σ_D^2 that are the same as for the P function. Therefore, the noise in D also has zero mean and $\sigma_D = \sigma_P$.

Now that the mean and the variance of the noise in P and D are explicitly known, the Probability Distribution Function (PDF) of the noise will be studied. To get an idea of the noise distribution behaviour for different SNR_M values, one can run a numerical simulation and obtain the plots from Figure 2.3. It is interesting to observe that for $SNR_M \geq 17$dB the behaviour is essentially Gaussian, a condition that we will assume valid for the rest of the discussion.

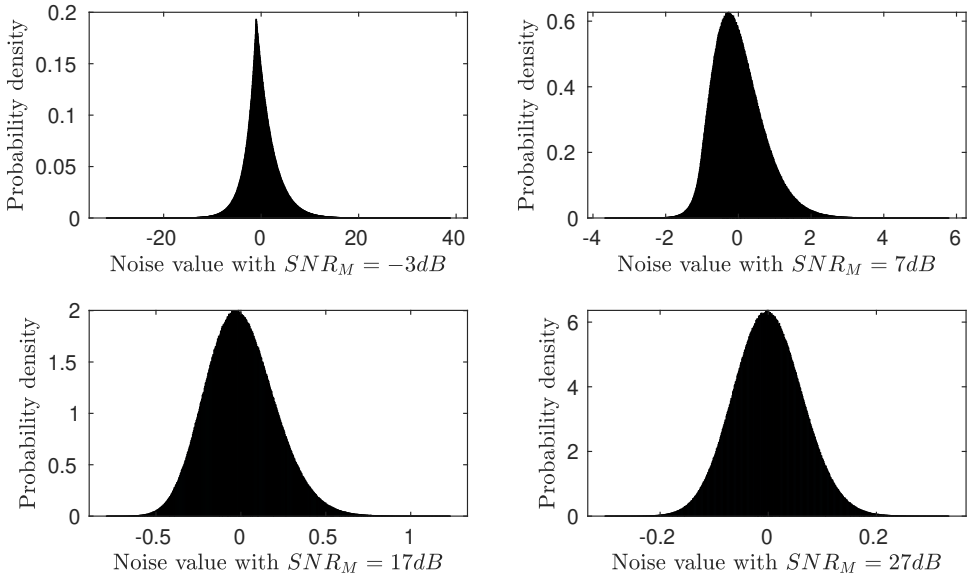

Figure 2.3: Noise PDF at $\alpha = 0$ with different SNR_M values

Several approaches can then be used to calculate error bounds of P and D due to noise at a given α. A numerical approach has been found to give more accurate bounds than an analytical approach using Chebyshev's inequality, or its improvement for unimodal distributions, the the Vysochanskij–Petunin inequality [6]. However, the analytical bound can be used to validate the numerical result.

Figure 2.4 shows the 99% error numerical bounds for P and D. As expected, the error is maximal at $\alpha = 0$, because $\text{sinc}^2(\alpha)$ is maximal at $\alpha = 0$. The spread of the error bounds is plotted for several values of SNR_M in Figure 2.5. One can observe that for very low values of SNR_M, the error at $\alpha = 0$ is close to 1, which is the maximum value of the ideal P function. In other words, the noise in the system is so large that the P function cannot accurately indicate whether the system is in lock or not. Therefore, a minimum value of SNR_M is required to ensure that the P and D functions are usable for carrier acquisition.

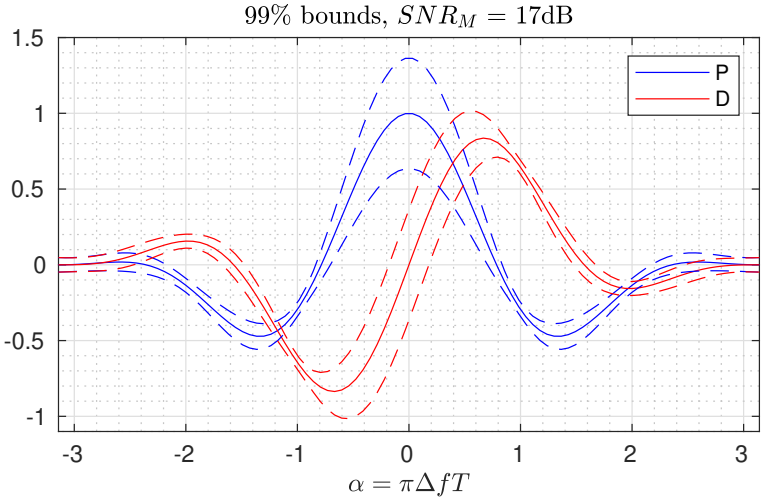

Figure 2.4: P and D with 99% error bounds

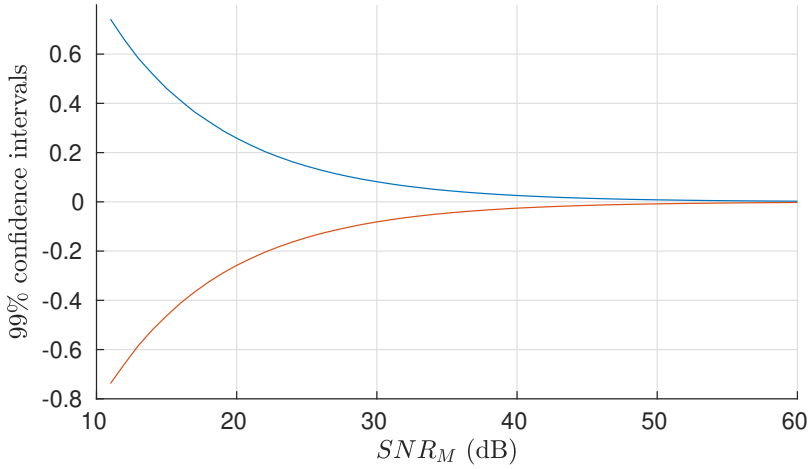

Figure 2.5: P and D error bounds at $\alpha = 0$

In order to allow the sweeping algorithm to lock for extended periods of time, a minimum SNR_M is required (or, equivalently, a minimum M). With the help

of the previous plots, one can find that for $SNR_M \geq 17$ dB, the noise bound at frequency lock is at most half the value required to reach the ideal threshold. Also, for $SNR_M \geq 17$ dB, the noise is known to be normally distributed. Experimentally, this has been tested to be a good minimum requirement for SNR_M, as thirty minute simulations have not reported losses of lock. This gives the condition

$$\frac{P_c}{N_0}\frac{M}{F_s} \geq 17dB \tag{2.6}$$

Now that a condition to calculate M is available, the ideal threshold θ for the P function has to be estimated. As explained in subsection 1.3.3, once θ is surpassed, a fully adaptive step will be used, and its value will be derived from the D function. This leads to the definition of a linear region for D, which can be obtained numerically by estimating its global minimum and maximum respectively. As can be seen in Figure 2.6, the linear region occurs when $\alpha \in [-0.672, 0.672]$. Note that this region is particularly interesting because D gives the direction and the size of the step that is required to jump to $\alpha = 0$, i.e., the lock condition.

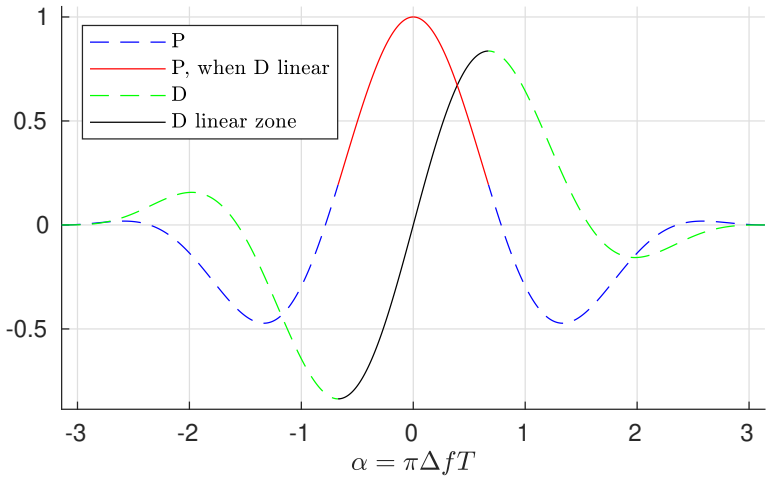

Figure 2.6: P and D without noise, with emphasis on the linear region

To calculate θ, the minimum and maximum values of α from the linear region are substituted into the P function from Equation 2.5, resulting in

$$\mathbb{E}[P(-0.672)] = \cos(2 \cdot -0.672)\,\mathrm{sinc}^2(-0.672) = \mathbb{E}[P(0.672)] = 0.193, \tag{2.7}$$

which defines the ideal value for θ when there is no noise in the system. However, as Figure 2.4 shows, for values of α not in the linear region, P with noise does not surpass the ideal threshold $\mathbb{E}[P(-0.672)] \approx 0.2 = \theta$. This holds for values of $SNR_M \geq 17$ dB, the suggested minimum from Equation 2.6. It has been experimentally verified for values of SNR_M as low as 12 dB. This means that the ideal threshold is very robust, because in a neighborhood of the linear zone, D will still point the next iteration of the algorithm towards $\alpha = 0$. In practice, the ideal threshold will be used.

Now that the ideal threshold and the minimum SNR_M conditions have been obtained, an analytical bound can validate the previous findings. The Vysochanskij–Petunin inequality will be used [6], which states that given X, a random variable with unimodal distribution, mean μ and finite, non-zero variance σ^2, for any

$\lambda > \sqrt{8/3}$ it holds that

$$\text{Prob}(|X - \mu| \geq \lambda\sigma) \leq \frac{4}{9\lambda^2} \tag{2.8}$$

Applying this to the random variable of noise in P, which has $\mu = 0$ and

$$\sigma_P^2 = \frac{\text{sinc}(\alpha)^2}{SNR_M} + \frac{1}{2SNR_M^2}$$

one can obtain a measure of how likely it is for θ to be surpassed for α not in the linear zone. Assuming a symmetrical distribution of noise, the bound from Equation 2.8 becomes

$$\text{Prob}(X - \mu \geq \lambda\sigma) \leq \frac{2}{9\lambda^2}$$

Using $\lambda\sigma_P = \theta - \cos(2\alpha)\,\text{sinc}^2(\alpha)$, from Equation 2.5 , the probability that $P(\alpha) > \theta$ when $\lambda > \sqrt{8/3}$ is

$$\text{Prob}(P(\alpha) \geq \theta) = \text{Prob}(X \geq \lambda\sigma_P) \leq \frac{2}{9\lambda^2}$$

$$\text{Prob}(P(\alpha) \geq \theta) \leq \frac{2\,\text{sinc}(\alpha)^2 + \frac{1}{SNR_M}}{9(\theta - \cos(2\alpha)\,\text{sinc}^2(\alpha))^2}\frac{1}{SNR_M}$$

Asymptotically, this means that for $|\alpha| \to \infty$, the probability that $P > \theta$ is

$$\text{Prob}(P \geq \theta) \leq \frac{1}{9\theta^2 SNR_M^2}$$

For $\theta = 0.2$ and $SNR_M = 17$dB, the probability that $P > \theta$ for $|\alpha| \to \infty$ is $\text{Prob}(P \geq \theta) \leq 0.114\%$. This confirms the expectations that for all working SNR_M the ideal threshold will not be surpassed when $|\alpha| \to \infty$.

2.2.4 Step calculation

The threshold θ has been calculated, and the noise in the sweep branch has been analyzed. This allows the adaptive step to apply once θ is surpassed to be calculated. After, it will be estimated under the assumption that the noise is normally distributed. Finally, the fixed step will also be calculated.

First, the adaptive step will be obtained. Let $\hat{\Delta f}$ denote the estimated $\Delta f = f - \hat{f}$, where f is the input frequency and $\hat{f}$ is the estimated frequency. The objective is to find a good Δ_2 to multiply the direction function D, such that $\Delta_2 D = \hat{\Delta f}$. In absence of noise, D is

$$\mathbb{E}[D] = \Im\left\{x_{k-1}^* x_k\right\} = \sin(2\alpha)\,\text{sinc}^2(\alpha)$$

Near $\alpha = 0$, $\sin(2\alpha) \approx 2\alpha$ and $\text{sinc}(\alpha) \approx 1$, so in the linear region, an optimal estimator for Δf would be

$$\Delta_2 D = \hat{\Delta f} = \frac{1}{2\pi T} D \tag{2.9}$$

Given that for $SNR_M \geq 17$ dB the noise has been proven to be Gaussian distributed, it is trivial to see that the previous estimator for Δf is a Maximum Likelihood Estimator (MLE). For $SNR_M < 17$ dB, when the distribution of the noise is

non-Gaussian, this might not be the case. Given a noise that is centered around zero, with a finite variance and unimodal, the sample mean is not necessarily the MLE (e.g., a log-normal distribution). However, since in this dissertation it is assumed that $SNR_M \geq 17$, the non-Gaussian case is only of concern in future work.

The estimator from Equation 2.9 is a MLE, but it is not ideal. When the sweep algorithm is in lock, a large adaptive step introduces noise in the output frequency, which is not desirable. In order to take into account this noise, the following family of functions will be considered

$$\Delta_2^C D = \hat{\Delta f} = \frac{C}{2\pi T} D$$

where C is a correction factor that will force more conservative steps at low SNR conditions.

To calculate C, the following design criteria is selected: C will be chosen to be the largest value that keeps the step due to noise at $\alpha = 0$ below 1Hz with 99% probability, which is equivalent to

$$\text{Prob}\left(\frac{C}{2\pi T}|D_{(\alpha=0)}| \geq 1\right) = 1\% \tag{2.10}$$

This way Δ_2 is a step large enough to arrive at $\alpha = 0$, or frequency lock, quickly; and keep the fluctuations due to noise low.

The definition from Equation 2.10 allows a numerical computation of Δ_2. However, assuming Gaussian distribution of the noise, one can use the Q function to deduce a formula to calculate the adaptive step for each value of SNR_M:

$$\begin{aligned} \text{Prob}(|\Delta_2 D| > 1Hz) < 1\% \iff Q\left(\frac{1}{\Delta_2 \sigma}\right) < \frac{0.01}{2} \implies \\ \Delta_2 = \frac{1}{Q^{-1}(0.01/2)}\sqrt{SNR_M} = 0.388\sqrt{SNR_M} \end{aligned} \tag{2.11}$$

Now that the adaptive step has been calculated, the fixed step Δ_1 is estimated. A larger step will allow a faster lock, but may skip the linear zone all together. Since it takes two iterations to calculate one output of P and D, the algorithm must ensure that at least two iterations of the algorithm fall inside of the linear zone. Therefore, the maximum Δ_1 is

$$2\pi\Delta_1 M T_s \leq 2 \cdot 0.672 \implies \Delta_{1_{\max}} = \frac{F_s \cdot 0.672}{\pi M} \tag{2.12}$$

Using a fixed step of $\Delta_{1_{\max}}$, the adaptive-sweep algorithm has been tested to work for $SNR_M \geq 17$dB.

2.2.5 Expected acquisition time

Given M, F_s and P_c/N_0, the maximum and minimum frequencies to acquire, $f_{\max} - f_{\min} = 2f_0$, and the steps Δ_1 and Δ_2 selected with the previously identified criteria, the goal is now to obtain an estimate of the expected acquisition time. To do so, first the adaptive step will be analyzed in order to obtain a formula that approximates the calculation from subsection 2.2.4. With it, an expected acquisition time formula

will be deduced, which will be later minimized in order to obtain a second condition for the sweep branch integrator size M.

Given the adaptive step formula from Equation 2.11, and the fixed step formula from Equation 2.12, the expected acquisition time is deduced with the following assumptions: the frequency offset will be assumed uniform within the sweeping range $f_{\max} - f_{\min} = 2f_0$, and the initial point in the linear zone where the ASA starts using the adaptive step will also be assumed uniform. Subsequently,

$$\begin{aligned}
\mathbb{E}[T_{acq}] &= \mathbb{E}[T_{fixed} + T_{adaptive}] = \mathbb{E}[T_{fixed}] + \mathbb{E}[T_{adaptive}] \\
\mathbb{E}[T_{fixed}] &= \int_{f_{\min}}^{f_{\max}} \frac{f - f_{\min}}{\Delta_1} MT_s \frac{1}{2f_0} df = \frac{f_0}{\Delta_1} MT_s \\
\mathbb{E}[T_{adaptive}] &= \int_{-1}^{1} |x| \frac{\Delta_1}{\Delta_2} MT_s \frac{1}{2} dx = \frac{0.672}{2\pi 0.387\sqrt{SNR}\sqrt{M}} MT_s \\
\mathbb{E}[T_{acq}] &= \frac{f_0 \pi M^2 T_s^2}{0.672} + \frac{0.672}{2\pi 0.387\sqrt{SNR}\sqrt{M}} \\
&= 4.68 f_0 \left(\frac{M}{F_s}\right)^2 + 0.276 \frac{1}{\sqrt{\frac{P_c}{N_0}}\sqrt{\frac{M}{F_s}}}
\end{aligned} \tag{2.13}$$

This indicates that, on average, having a higher sampling rate has the same effect as having a smaller integrator.

The upside of having a higher sampling frequency is that the continuous time approximation will be better. Also, the range of frequencies that the ASA will be able to detect will be higher. On the downside, the necessary hardware will be more sophisticated.

With the given expected acquisition time formula, it is interesting to find the minimum expected acquisition time. Knowing that Equation 2.13 is a function of M/F_s, its derivative can be calculated by letting $a = 4.68 f_0$, $b = 0.276/\sqrt{P_c/N_0}$, and substituting these values in the expression

$$f(x) = ax^2 + \frac{b}{\sqrt{x}} \implies f'(x) = 2ax - \frac{b}{2x^{3/2}}$$

Since all coefficients are positive, one can obtain the global optimum by simply taking the derivative and equating it to zero. This results in

$$\frac{M}{F_s} = \left(\frac{0.276}{4\sqrt{P_c/N_0}\, 4.68 f_0}\right)^{2/5} \tag{2.14}$$

Note that this derivation is valid under the assumed condition of $SNR_M \geq 17$ dB. Using this criteria, one obtains the performance plotted in Figure 2.7. An interesting observation is that for $P_c/N_0 \leq 52$ dB the M value obtained from Equation 2.14 is smaller than the necessary one to satisfy the minimum SNR_M condition. This means that the value of M obtained from Equation 2.14 will only be used when $P_c/N_0 > 52$ dB.

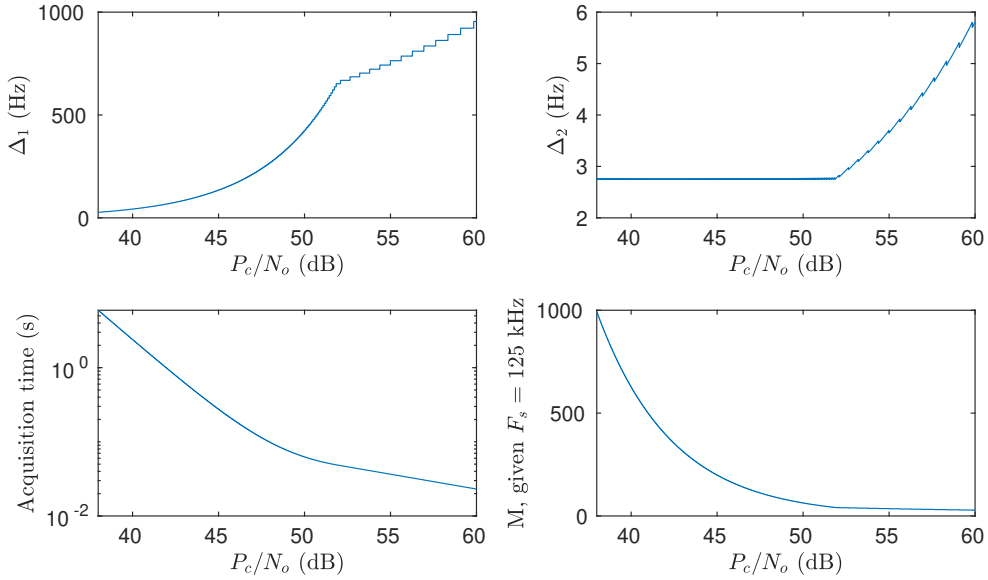

Figure 2.7: Optimal expected acquisition time performance plots

2.2.6 Standard Deviation of Doppler residuals

Having calculated the expected acquisition time, the next performance indicator to be calculated is the standard deviation of the difference between input and output frequencies, or Doppler residuals. This will be done in two parts. First, perfect lock will be assumed with a first order PLL. This will give an intuition of the behaviour of the system. Next, the only assumption that will be made is that the output noise tends to a stationary process. The resulting formula will be compared to the first approach.

Assuming perfect lock

Consider the multiplication of $s(t)$, the input signal from the antenna which carries complex AWGN (see Equation 1.4), and $s_{PLL}(t)$, the output signal of the NCO of the PLL, which can be modelled as

$$\begin{aligned} s &= e^{j\phi(t)} + n(t) \\ s_{PLL} &= e^{-\hat{\phi}(t)} \\ ss_{PLL} &= e^{j(\phi(t)-\hat{\phi}(t))} + e^{-j\hat{\phi}(t)}n(t) = e^{j(\phi(t)-\hat{\phi}(t))} + \tilde{n}(t) \end{aligned} \tag{2.15}$$

where $\tilde{n}(t)$ is also complex AWGN, with the same distribution as $n(t)$, because it has been multiplied by a complex number in the unit circle.

Assuming $\phi = \hat{\phi}$, which is a lock state with no error, then the product becomes

$$ss_{PLL} = \tilde{n}(t)$$

Extracting the imaginary part, $\tilde{n}(t)$ becomes real AWGN with variance $\sigma^2 = \frac{1}{2}\frac{1}{SNR_c} = \frac{1}{2}\frac{F_s}{P_c/N_0}$. This then goes through an integrate and dump block of size N, which reduces the variance by an N factor. Now, $\sigma^2 = \frac{1}{2}\frac{F_s}{P_c/N_0}\frac{1}{N}$.

After the integrate and dump comes the PLL filter, which will be assumed of first order with coefficient K_1. This means that the noise will be multiplied by K_1, and later transformed to frequency by multiplying by $\frac{F_s}{N}\frac{1}{2\pi}$.

Therefore, for a first order filter and assuming perfect lock, the expected output error standard deviation σ_f is

$$\sigma_f = \frac{F_s}{2\pi N} K_1 \sqrt{\frac{1}{2}\frac{1}{P_c/N_0}\frac{F_s}{N}}$$

Assuming the output noise tends to a stationary process

When in lock, as seen in Equation 1.6, the system can be linearized using the approximation $\sin(x) = x$ to extract the imaginary part of the output of the mixer. Due to the linear nature of the system, one can analyze the signal components and the noise components separately. Analyzing the noise component, the phase ϕ can be modelled as AWGN of variance σ^2. Using the transfer function for a second order loop presented in Equation 1.8, and knowing that the frequency is the derivative of the phase with respect to time, one obtains

$$\frac{\hat{f}(z)}{z-1} = \hat{\phi}(z) \implies \frac{\hat{f}(z)}{z-1} = \phi(z)H(z) \implies$$
$$\hat{f}(z)\left(z^2 + (K_1+K_2-2)z + 1 - K_1\right) = \phi(z)(z-1)\left((K_1+K_2)z - K_1\right)$$

Since the input phase ϕ has been modelled as AWGN, its samples will be denoted n_k. Using this notation, the previous equation implies the following recurrence

$$\begin{aligned}\hat{f}_{k+2} + (K_1+K_2-2)\hat{f}_{k+1} + (1-K_1)\hat{f}_k = \\ (K_1+K_2)n_{k+2} - (2K_1+K_2)n_{k+1} + K_1 n_k\end{aligned} \tag{2.16}$$

with which the three first outputs of the system can be computed

$$\hat{f}_0 = (K_1+K_2)n_0$$
$$\hat{f}_1 = (K_1+K_2)n_1 + (K_2 - (K_1+K_2)^2)n_0$$
$$\hat{f}_2 = (K_1+K_2)n_2 + (K_2 - (K_1+K_2)^2)n_1 + ((K_1+K_2)^3 - 2K_2(K_1+K_2) + K_2)n_0$$

and, by induction

$$\begin{aligned}\hat{f}_{k+2} =&(K_1+K_2)n_{k+2} + (K_2 - (K_1+K_2)^2)n_{k+1} \\ &+ ((K_1+K_2)^3 - 2K_2(K_1+K_2) + K_2)n_k + \sum_{i=0}^{k-1}\lambda_i n_i\end{aligned} \tag{2.17}$$

Equation 2.17 allows to compute some covariances, where σ^2 is the noise variance

$$C_0 = E[\hat{f}_k n_k] = (K_1+K_2)\sigma^2$$
$$C_1 = E[\hat{f}_k n_{k-1}] = (K_2 - (K_1+K_2)^2)\sigma^2$$
$$C_2 = E[\hat{f}_k n_{k-2}] = ((K_1+K_2)^3 - 2K_2(K_1+K_2) + K_2)\sigma^2$$

The variance of $\hat{f}_k$ can now be computed, assuming that it is a stationary process. In other words, $E[\hat{f}_k\hat{f}_{k-j}] = R_j$ is assumed to be independent of k. Taking the

expected value of multiplying Equation 2.16 by $\hat{f}_k$, $\hat{f}_{k+1}$ and $\hat{f}_{k+2}$, the following system is obtained

$$\begin{aligned}
R_2 + (K_1 + K_2 - 2)R_1 + (1 - K_1)R_0 &= K_1 C_0 \\
(2 - K_1)R_1 + (K_1 + K_2 - 2)R_0 &= -(2K_1 + K_2)C_0 + K_1 C_1 \\
(1 - K_1)R_2 + (K_1 + K_2 - 2)R_1 + R_0 &= (K_1 + K_2)C_0 - (2K_1 + K_2)C_1 + K_1 C_2
\end{aligned}$$

The closed form solution of this linear system can be obtained analytically, and results in

$$\begin{aligned}
R_0 &= \frac{-2(2K_1^3 + 3K_1^2 K_2 + K_1 K_2^2 + K_2^2)}{2K_1^2 - 4K_1 + K_1 K_2} \\
R_1 &= \frac{2K_1^4 + 5K_1^3 K_2 + 4K_1^2 K_2^2 - 2K_1^2 K_2 + K_1 K_2^3 + K_2^3 - 2K_2^2}{2K_1^2 - 4K_1 + K_1 K_2} \\
R_2 &= -\frac{2K_1^5 + 7K_1^4 K_2 - 2K_1^4 + 9K_1^3 K_2^2 - 9K_1^3 K_2 + 5K_1^2 K_2^3 - 9K_1^2 K_2^2}{2K_1^2 - 4K_1 + K_1 K_2} \\
&\quad - \frac{2K_1^2 K_2 + K_1 K_2^4 - K_1 K_2^3 - 2K_1 K_2^2 + K_2^4 - 4K_2^3 + 2K_2^2}{2K_1^2 - 4K_1 + K_1 K_2}
\end{aligned}$$

which gives the prediction for σ_f

$$\sigma_f = \frac{F_s}{2\pi N}\left(\sqrt{R_0}\right)\sqrt{\frac{1}{2}\frac{1}{P_c/N_0}\frac{F_s}{N}} \tag{2.18}$$

Finally, knowing that $1 << K_1 << K_2$, and specifically that $K_2 = \frac{K_1^2}{4}$, the autocorrelation's Taylor expansion can be calculated at $K_1 = 0$. This results in $R_0 = K_1^2 + O(K_1^3)$. In other words, the first-order approximation of σ_f for a 2nd order PLL matches the performance of a first order PLL.

$$\sigma_f = \frac{F_s}{2\pi N}\left(\sqrt{R_0}\right)\sqrt{\frac{1}{2}\frac{1}{P_c/N_0}\frac{F_s}{N}} \approx \frac{F_s}{2\pi N} K_1 \sqrt{\frac{1}{2}\frac{1}{P_c/N_0}\frac{F_s}{N}}$$

Figure 2.8 presents the goodness of the exact analytic prediction from Equation 2.18, as well as its first order approximation, compared to the measured σ_f from the GNU Radio implementation presented in section 2.3.

Higher order PLLs and phase error analysis

The previously outlined procedure can be generalized to higher order loops, which allow tracking higher Doppler dynamics but are more computationally intensive. The work done in this thesis uses a second order loop, but the analysis for a third order loop can be found in Appendix B, which also contains an analysis of the residual phase error.

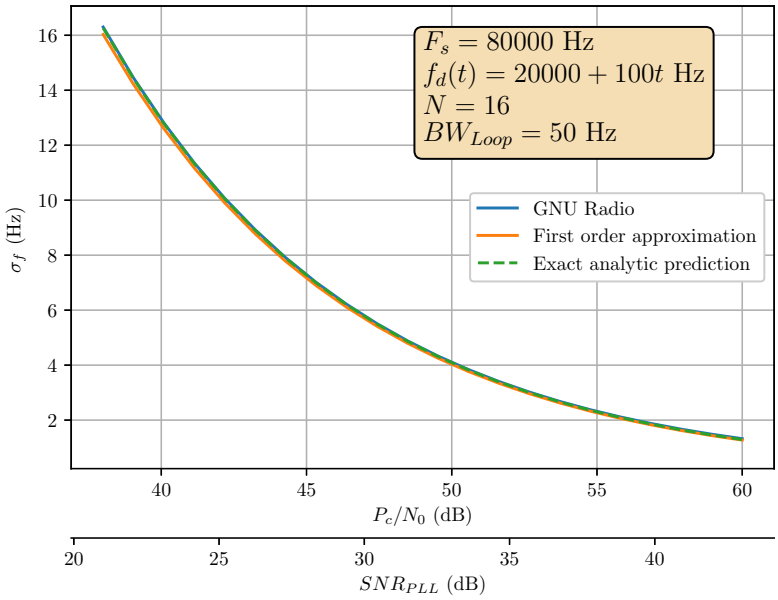

Figure 2.8: Experimental σ_f, with exact and first order predictions

2.3 SDR Implementation

The chosen tool to implement the algorithm is GNU Radio, which is a free and open-source software development toolkit that provides signal processing blocks to implement software defined radios. GNU Radio has a graphical tool to create signal flow graphs called GNU Radio Companion (GRC). Python scripts are generated from the flow graph. Visually, a GRC flow graph is very similar to a Simulink flow graph with blocks; an example can be found in Figure 2.9.

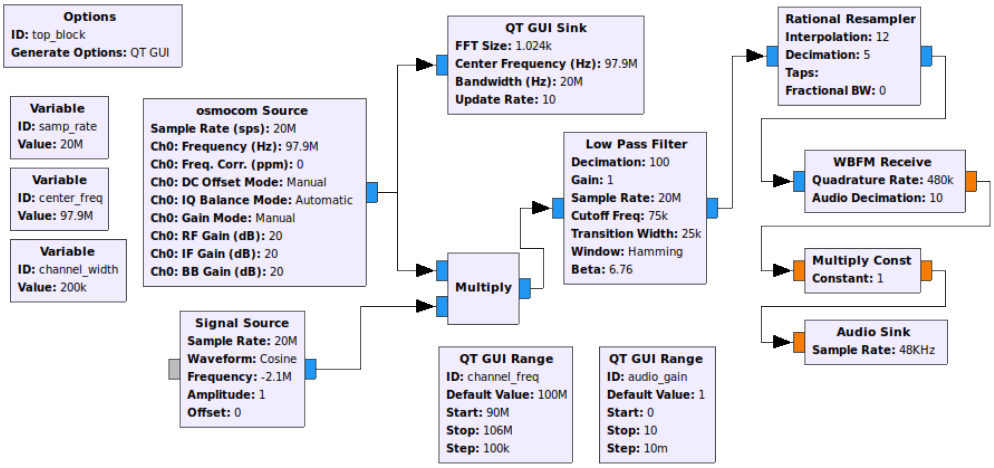

Figure 2.9: Flow graph of a FM Radio with GRC

The GNU Radio runtime is designed to stream large amounts of data in real-time between parallel computational nodes each operating completely independently. Each block has a completely independent scheduler running in its own execution thread. Blocks run as fast as the CPU, data flow, and buffer space allows [1].

GRC flow graphs process data by chunks, not sample by sample, to improve system performance. However, feedback loops like the one present in a PLL need to be processed sample by sample. Since this would produce significant data overhead, GNU radio developers decided to not support loops in flow graphs [10]. Despite that, one can still design loops as long as they are contained in an individual block, which can be programmed in either Python or C++. As the bulk of the computation is implemented in C++, the GNU Radio flow graph runs much faster than the Simulink one.

The flow graph of the system with a signal generator block, an Additive White Gaussian Noise (AWGN) channel block and the PLL block can be found in Figure 2.10. Both the signal generator and the PLL block have been implemented internally in C++, together with `gr_modtool`, which is a tool developed to facilitate this process.

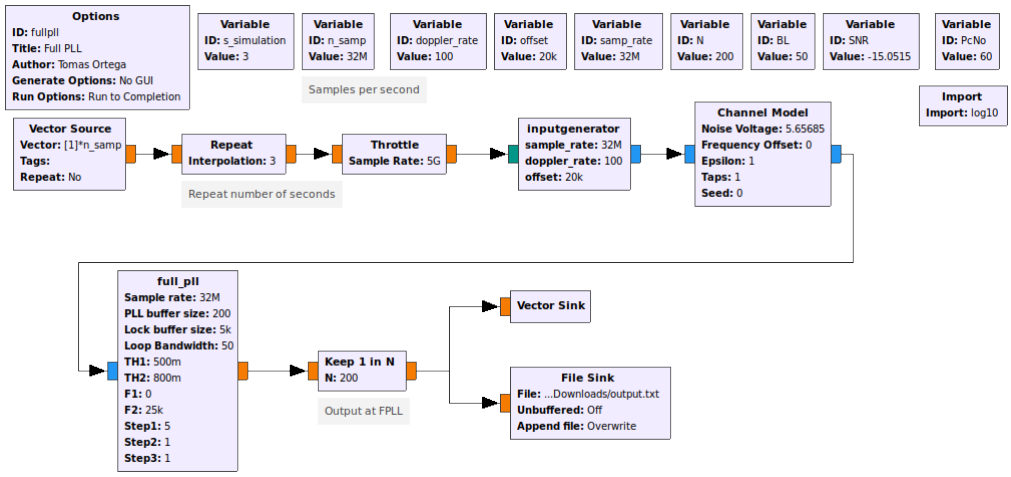

Figure 2.10: Flow graph with custom blocks in GRC, with a AWGN channel

Another nice feature of GNU radio is that it comes with pre-installed blocks that allow it to interface with commercially available SDR peripherals (see Figure 2.9, block `osmocom Source`). For this thesis, a HackRF One peripheral has used alongside the GNU Radio software for radio transmission. A Hack RF One is able to transmit or receive radio signals from 30 MHz to 6000 MHz with the power it receives from a USB port, with a maximum bandwidth of 20 MHz. This has been used to transmit a carrier frequency with a certain offset to simulate the received signal from a spacecraft.

To receive the signal transmitted from the HackRF One, another more capable SDR peripheral has been used, namely the Ettus E310. This device is a stand-alone SDR platform designed for field deployment. It has its own processor running an OpenEmbedded Linux distribution, and uses a Field Programmable Gate Array (FPGA) to interface the font-end electronics with the embedded CPU and accelerate digital signal processing functionality. It provides up to 56 MHz of instantaneous bandwidth and can operate at carrier frequencies ranging from 70 MHz – 6 GHz. However, only 10 Msps can be transferred into the processor from the FPGA. The steps used to setup the stand-alone operation of the Ettus E310 can be found in Appendix C.

Chapter 3

Results

This chapter will present the results obtained from the implementation of the theoretical research in section 2.2 with the software tools described in section 2.3.

First, a Matlab design script will be presented, with its inputs and outputs, and a summary of the steps it follows. Next, the Simulink implementation will be presented, along with results that illustrate several effects described in previous chapters. The Simulink implementation will be used to validate the GNU Radio implementation, which has been used to produce the majority of figures from this thesis, and has been run with a recording of a real LRO signal, also included in this section. Finally, the stand-alone ASA implementation results will be presented, where two LRO signals are acquired and tracked successfully.

3.1 Design script

Using the theoretical analysis from section 2.2, a design script for the ASA has been written. Next, the inputs and outputs of the script will be presented. Finally, the procedure it follows will be explained.

First, the inputs of the script are the following:

1. Maximum steady-state phase error
2. Maximum absolute expected Doppler rate
3. Initial and final sweep frequencies
4. Minimum expected P_c/N_0
5. Maximum allowed error in adaptive step (set to 1Hz by default)
6. Probability that the maximum allowed error from item 5 is surpassed (set to 1% by default)
7. Boolean value: set to minimize frequency variation or acquisition time

And the following outputs:

1. Loop bandwidth B_L
2. Sampling frequency F_s

3. Length of PLL integrator N

4. Length of the sweep integrator M

5. Fixed step size Δ_1

6. Adaptive step size Δ_2

7. Sweep threshold θ

8. Expected acquisition time $\mathbb{E}[T_{acq}]$

9. Expected standard deviation of Doppler residuals σ_f

The outputs correspond to the degrees of freedom that were indicated in Table 2.1.

Now that the inputs and outputs have been specified, the overview of the script will be presented. The complete implemented design script can be found in Listing D.6. The outline of the procedure is the following:

1. The maximum steady-state phase error and the maximum expected Doppler rate inputs are used with Equation 2.3 to obtain the loop bandwidth B_L.

2. The sampling frequency is chosen to be twice the sweeping range to meet the Nyquist rate.

3. The sweep integrator size M is chosen to be the maximum of the values obtained from Equation 2.6 and Equation 2.14.

4. If the script is set to minimize frequency variation, the size of the PLL integrator N is chosen to be as large as possible while maintaining the CU approximation of Equation 2.1. The size of the sweep integrator M is then updated, if necessary, to maintain a sweep update rate lower than the PLL rate with Equation 2.2. On the other hand, if the script is set to minimize acquisition time, then M is constant and equal to the value from step 3. Thus, N is chosen to be the largest value that satisfies both Equation 2.1 and Equation 2.2 without changing the value of M.

5. The size of the steps is calculated based on the chosen sampling rate, the sweep integrator size, and the allowed error tolerances following Equation 2.12 for Δ_1 and Equation 2.11 for Δ_2.

6. The value of θ is constant and set to 0.193 based on subsection 2.2.3.

7. The expected acquisition time and the expected output frequency error are calculated with Equation 2.13 and Equation 2.18, respectively.

3.2 Simulink

This section presents the results obtained via simulation using a Simulink model of a combined PLL and adaptive-sweep algorithm. They are used to validate the theoretical analysis from Chapter 2.

3.2.1 PLL

The Simulink model of a PLL was presented in Figure 1.4. This model has been used to explore two questions: the effect of the PLL bandwidth on sinusoidal Doppler shifts, and the effect of an initial offset on the acquisition time of the PLL.

First, to illustrate the effect of the PLL bandwidth, the PLL response is recorded for different Doppler sinusoidal inputs. Given $f_d(t) = sin(2\pi\nu t)$ and $B_L = 50$ Hz, the expected norm of the difference between $f_d(t)$ and its estimate can be seen in Figure 3.1, which has been obtained with the code in Listing D.1. This plot

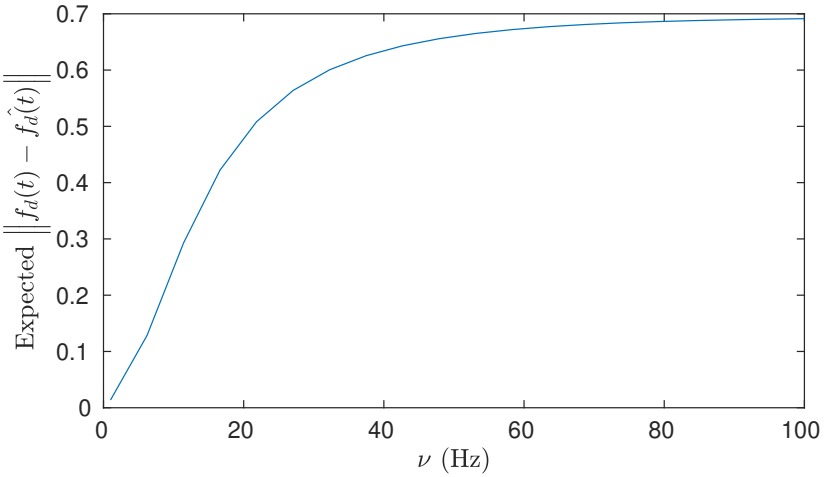

Figure 3.1: Difference between real and estimated $f_d(t)$ depending on ν

illustrates the expected behaviour, i.e., the PLL cannot track $f_d(t)$ for values of ν exceeding the PLL bandwidth. Note that if the PLL filter had an ideal (rectangular) frequency response, the expected norm of the error for $\nu < B_L$ would be zero. As one can see, the filter is not ideal.

The second phenomenon that has been studied is the effect of the initial frequency offset of $f_d(t)$, denoted f_0. To illustrate this effect, input signals with offsets of 1 Hz, 100 Hz, and 1 kHz have been input into the system, with the output plotted in Figure 3.2.

Given a 1 Hz initial offset, the PLL takes less than 0.1s to lock. For 100 Hz, the convergence happens around $t = 0.4$ s. On the other hand, the PLL converges very slowly (around $t = 2500$s) for $f_0 = 1$ kHz, and does not converge for $f_0 \geq 5$ kHz. This illustrates the need for a frequency sweep to correct the initial offset when it is higher than the PLL bandwidth.

3.2.2 Adaptive-Sweep Algorithm

Figure 3.3 presents a Simulink model that combines the previously analyzed PLL with the adaptive-sweep algorithm. The output of the sweeping algorithm has been added before the input of the NCO. This allows the sweeping algorithm to assist the PLL when the Doppler offset is large.

The behaviour of the system in Figure 3.3 with $F_s = 32$ MHz, $B_L = 50$ Hz, a PLL integrator of $N = 200$ samples, a sweep integrator of $M = 5000$ samples, $\theta = 0.5$, $\Delta_1 = 125$ Hz, $\Delta_2 = 25$ Hz and $f_d(t) = 100t + 20000$ Hz can be found

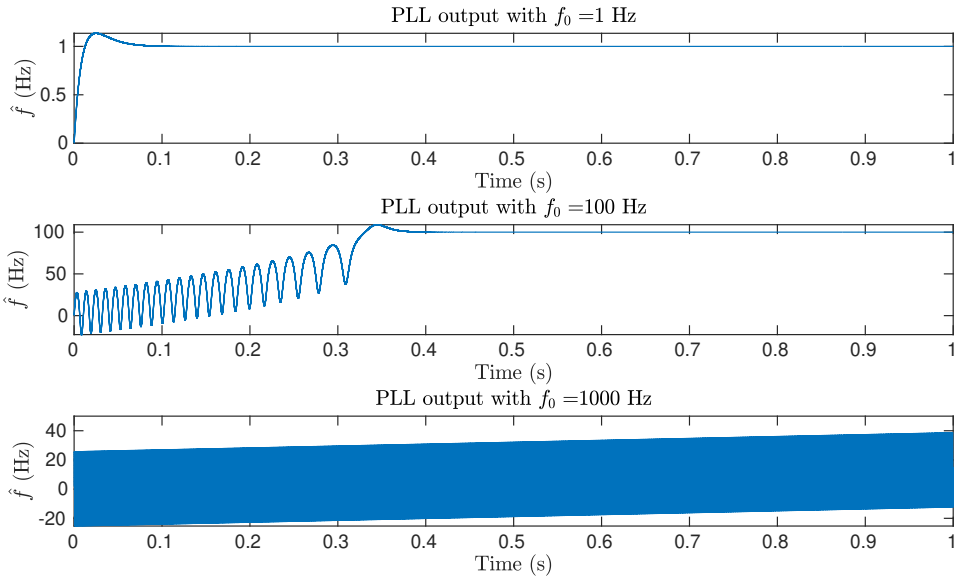

Figure 3.2: Estimated $f_d(t)$ with 1 Hz, 100 Hz and 1 kHz of offset

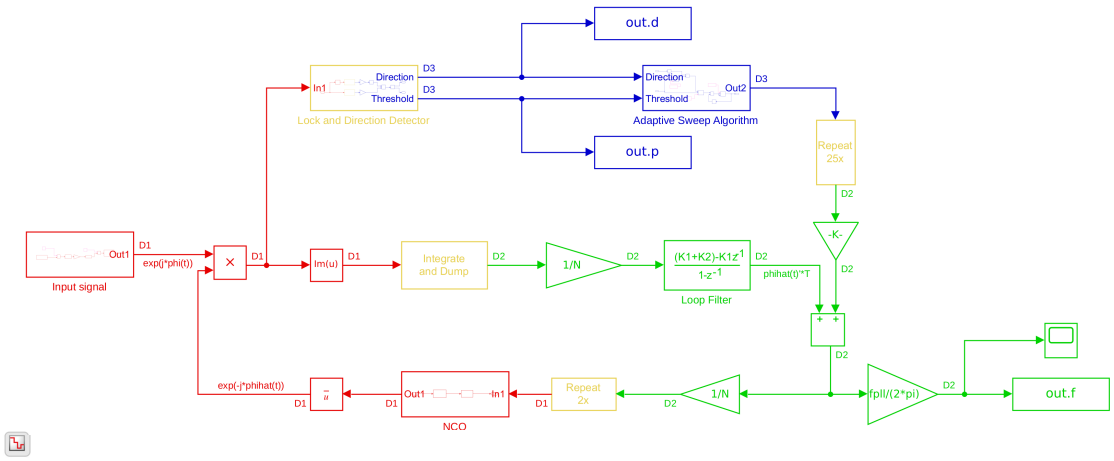

Figure 3.3: Complete Simulink model of the system

in Figure 3.4. This system configuration parameters are partially inherited from reference [2], but the sweep algorithm is allowed to take larger steps since no noise is included in the system. Three stages are clearly visible: first, when the fixed step is used, with a steep climb in frequency; second, when the adaptive step is used; and third, where the plot remains flat and the PLL tracks the frequency.

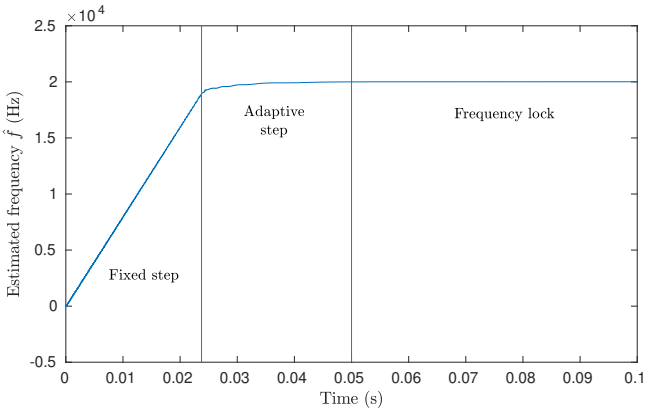

Figure 3.4: $\hat{f}(t)$ with the proposed sweep algorithm

3.3 Software-Defined Radio

This section presents the results obtained with the GNU Radio implementation, which is the implementation with which most of the figures of this thesis have been obtained. First, it will be validated using Simulink model shown in section 3.2. Next, the performance of both implementations will be compared. Finally, the results obtained from running the GNU Radio implementation with an LRO track as input will be presented.

The results of a simulation with the same parameters as in Figure 3.4, run with both implementations, are compared in Figure 3.5. As expected, their output is the same. Again, this simulation has been run without noise.

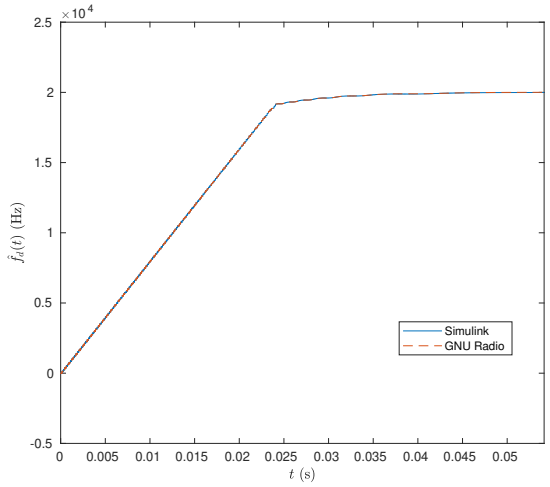

Figure 3.5: Comparison of results with Simulink and GNU Radio simulations

Once the SDR implementation has been validated, its performance is compared to the Simulink model. The 90s Simulink simulations in [2] took a whole day to run. As the model is linear, one can estimate the run time. To simulate 1s, the Simulink model takes 880.3s. This means it would take 22h to run the full simulation. In contrast, the full simulation with the SDR implementation takes 68.34s to run. The code for the GNU Radio Python program can be found in Listing D.4. The main C++ code used for the custom block that runs the ASA can be found in Listing D.5.

This difference in performance allows the GNU Radio implementation to be used to produce results like the one in Figure 3.6, where the standard deviation of the estimated frequency is plotted against the carrier SNR with different filter bandwidths. To obtain this plot multiple simulations were run at different B_L and P_c/N_0 levels. One can observe the behaviour predicted in subsection 2.2.6, which is that a higher B_L will make σ_f higher, which in turn will hurt the recovery of the modulated symbols. Using Equation 2.18, the curves should follow

$$\sigma_f \approx \frac{8}{5\pi} B_L \sqrt{\frac{1}{2N}\frac{1}{SNR_c}}, \tag{3.1}$$

which the simulations confirm. Since SNR_c is in logarithmic scale on the x-axis, the curves are not linear.

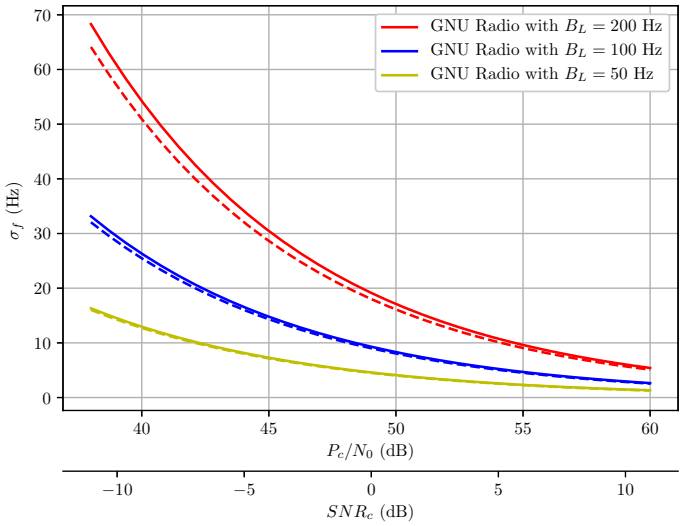

Figure 3.6: σ_f when PLL is locked, with $N = 16$ and Equation 3.1 (dashed line)

Finally, GNU Radio has a built-in block to read from file, which has been used to run the ASA with recordings of real signals. This can be used to obtain the results that a hardware implementation of the ASA would have obtained when recording the signal. To this purpose, an open-loop recording in the Goldstone DSN Antenna receiving a signal from the LRO has been used. The spectrogram of the signal, and the output that the ASA produced with the Short Time Fourier Transform (STFT) of the LRO recording plotted above, can be found in Figure 3.7. As can be seen in the spectrogram, the maximum Doppler shift is of 80 Hz, this is because the DSN already has done some Doppler compensation with the LRO position predicts.

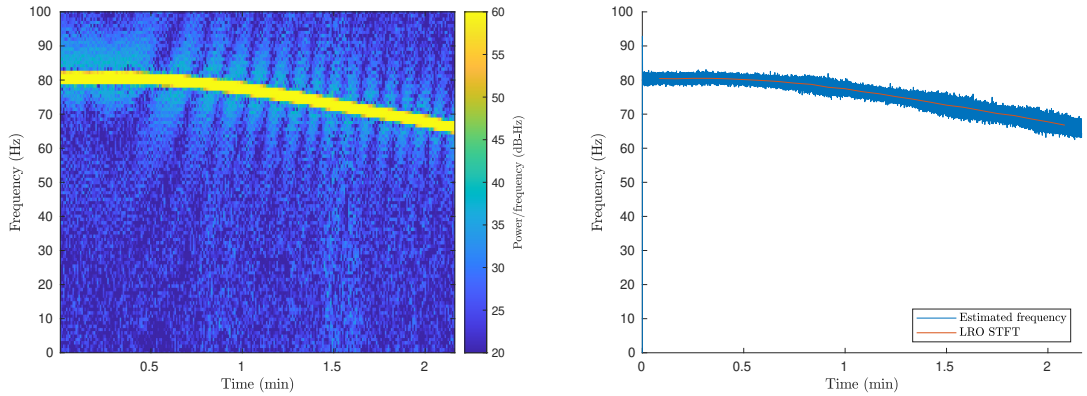

Figure 3.7: LRO signal spectrogram and system output

3.4 Stand-alone SDR platform

As explained in section 2.3, the SDR implementation was loaded into an Ettus E310 stand-alone SDR platform. The technical details on how this was done can be found in Appendix C. This section first outlines the experiment carried out with this implementation, where LRO recordings were transmitted using a HackRF One SDR peripheral, and received by an Ettus E310 stand-alone SDR platform that ran ASA in real time. Next, the LRO tracks that were used for this experiment will be described. Finally, the results obtained from the Ettus E310 will be presented.

The set-up for the experiment, seen in Figure 3.8, was the following: the LRO tracks were stored in a computer, connected to the HackRF One, which served as a transmitter that emulated the LRO. The Ettus E310 received the signal from the HackRF and ran ASA in real time. The results were stored in the Ettus and later transferred to a computer via ssh in order to be plotted.

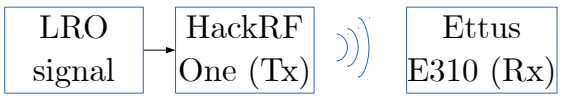

Figure 3.8: Scheme of the experimental set-up

The experimental set-up has been described, now the LRO tracks must be analyzed. These tracks, previously recorded at the DSN using its open loop receivers, are identified by an LRO orbit number (48247 and 48810). The spectrograms of both tracks can be found in Figure 3.9 and Figure 3.10.

The LRO track 48247 seen in Figure 3.9 is a scenario where the LRO is setting behind the lunar limb. The frequency jump around second 160 is the jump from coherent to non-coherent mode. In coherent mode, the LRO acquires and tracks the uplink frequency, generated with a DSN super stable hydrogen-maser oscillator, and transmits it back to the DSN multiplied by the turn-around ratio, a constant value that is pre-assigned to the spacecraft and kept constant for the duration of the mission. In non-coherent mode, the LRO transmits at its own ultra-stable oscillator frequency. It is also interesting to note that between second 350 and 400 there is a temporary loss of signal. At the end of the track the LRO turns the transmitter off.

The LRO track 48810 seen in Figure 3.10 is representative of a scenario in which LRO is egressing from the back-side of the Moon. Between seconds 50 and 100,

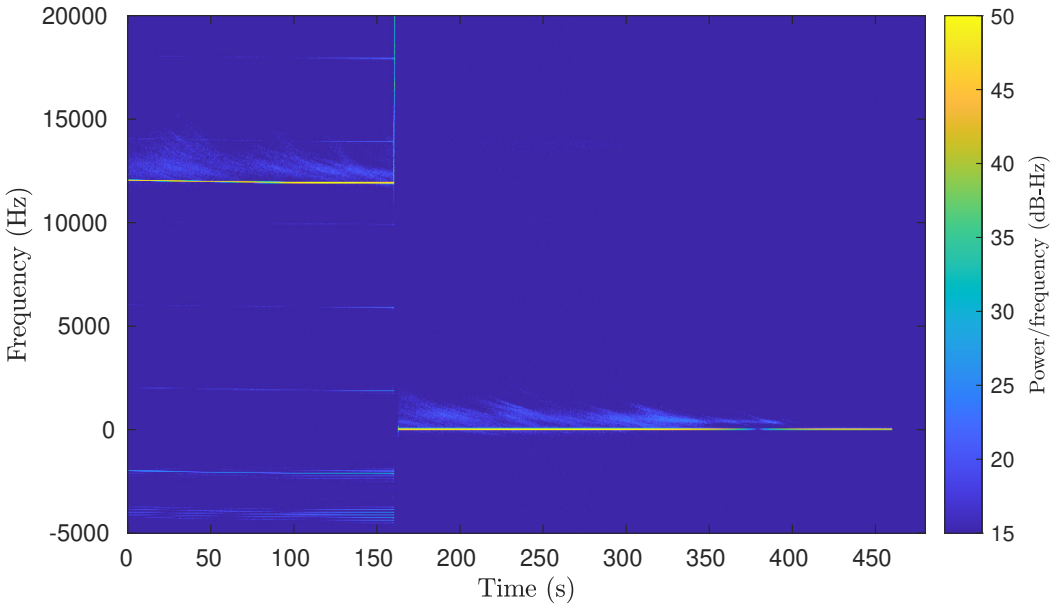

Figure 3.9: LRO track 48247 spectrogram

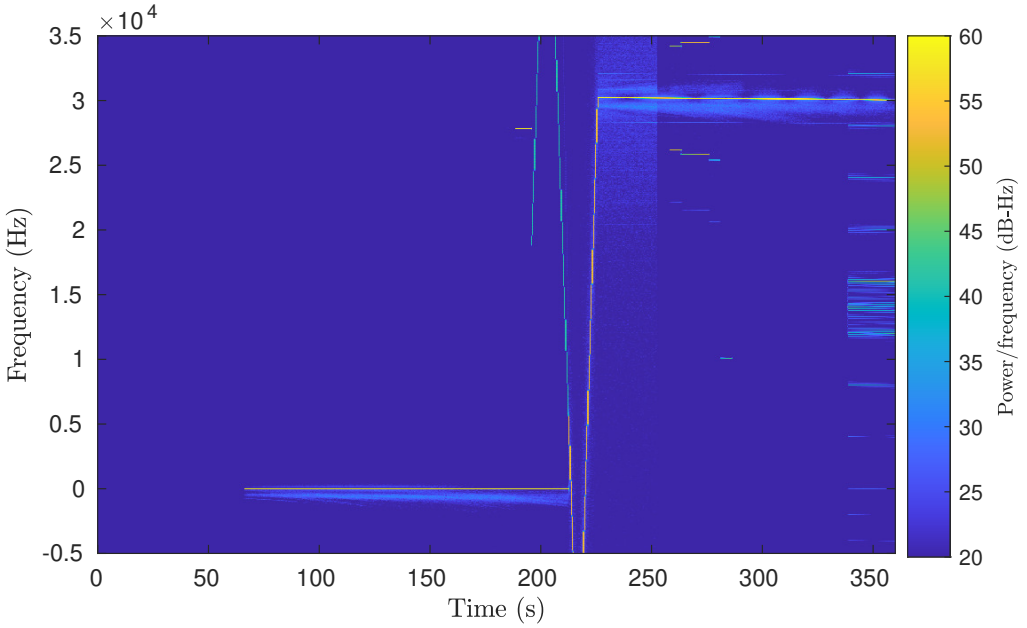

Figure 3.10: LRO track 48810 spectrogram

LRO turns the transmitter on and starts transmitting in non-coherent mode. Later, between seconds 200 and 250 there is a jump to coherent mode, as LRO's primary ground station, White Sands, starts transmitting an uplink signal.

Finally, the results of the experiment will be presented. Both tracks have been transmitted with a HackRF One, which has a very unstable oscillator, and have been acquired by the Ettus E310 stand-alone platform running the ASA. The results have been sent back to a computer via ssh to be plotted, and have produced Figure 3.11 and Figure 3.12.

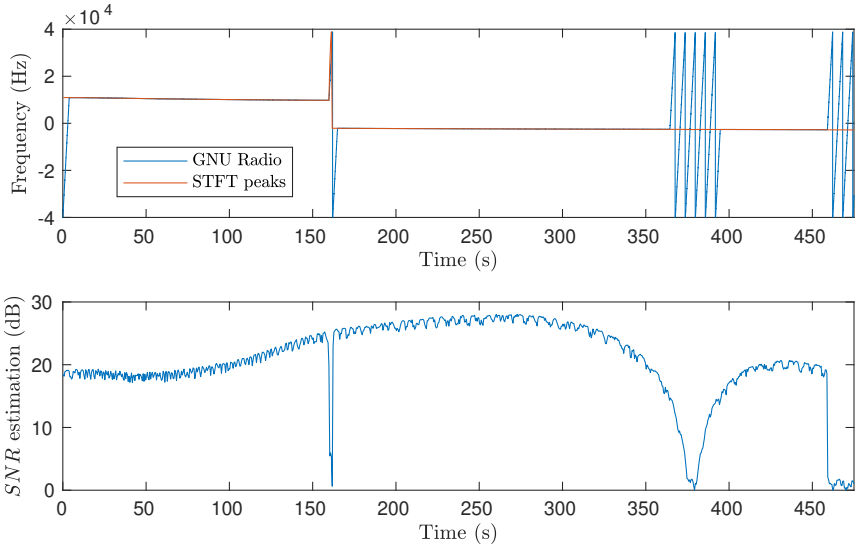

Figure 3.11: LRO track 48247 results

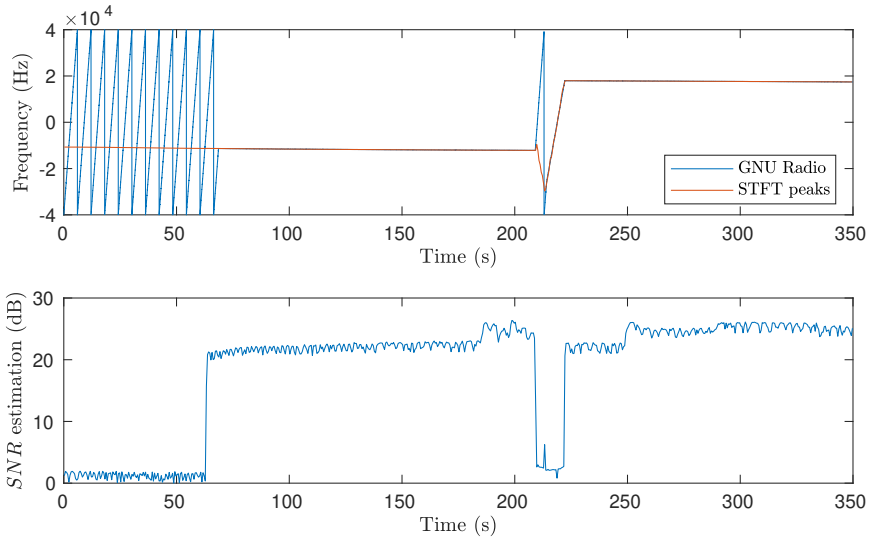

Figure 3.12: LRO track 48810 results

The SNR estimation and STFT of the received signal were calculated a posteriori from the recording of the received signal in the Ettus E310. The peaks of the STFT serve as a truth signal for the ASA output to be compared to, and the SNR

estimation is used to know when there is a carrier to be acquired and when there is just noise.

As can be observed, both Ettus E310 outputs resemble the original LRO recordings. They are not identical due to the oscillator in the HackRF, which introduces a large initial offset and a slight drop in frequency with time. However, both results follow the main events as expected: they acquire the carrier frequency when there is signal available and they track the change between coherent and non-coherent modes as desired.

In conclusion, the stand-alone SDR implementation of the ASA has successfully acquired and tracked two recordings from LRO signals in real time.

Chapter 4

Conclusions

4.1 Summary

The communications link between spacecraft is affected by the Doppler shift, which is a function of the relative trajectory between transmitter and receiver. When the relative trajectory is unknown, like in proximity links, the Doppler shift forces spacecraft radios to have a carrier acquisition and tracking system.

This thesis provides an overview and enhancement of the adaptive-sweep algorithm proposed in [2], a method to design the system parameters, a theoretical estimate of the system's performance, an SDR GNU Radio implementation, and experimental results that show the system working on real spacecraft signals.

Chapter 1 introduces the main concepts of this field, and examines present day algorithms used for carrier acquisition and tracking. In the literature review, the current efforts to improve these algorithms are explained. Finally, the thesis statement is presented.

Chapter 2 first gives some key definitions in order to understand the following theoretical analysis of the adaptive-sweep algorithm. The theoretical analysis allows the formulation of design equations for system parameters, and accurate theoretical predictions of the system's performance. Afterwards, a short description of the proposed SDR implementation is given, along with the benefits it will bring.

Chapter 3 describes the ASA design script, presents the obtained results with the GNU Radio implementation, gives examples of its possible applications, and presents the obtained results with the stand-alone SDR implementation and two LRO tracks.

Finally, this chapter summarizes the work, highlights the main findings and contributions, and lays out possible future endeavours on this subject.

4.2 Contributions

The main contributions of this thesis mirrors section 1.4. The adaptive sweep algorithm for carrier acquisition and tracking proposed in [2] has been enhanced to have an automatic adaptive step. In addition, the system's performance has been characterized in the presence of AWGN. The derived design equations have made the system fully determined. Also, the SDR implementation has reduced the simulation time by several orders of magnitude compared to the previous Simulink model, allowing faster than real-time simulations. Additionally, this implementation has

been cross-compiled to a stand-alone platform, which has allowed system testing without a computer, and yet prevented the added complexity of implementing in a FPGA.

The main theoretical contributions have been the noise analysis of the lock and direction functions, and the analytical prediction of acquisition time and frequency error variance. GNU Radio simulations have been run to validate the predicted frequency error variance.

The contributions towards technology maturity have been the design script for system parameters based on Doppler and noise characteristics, and the SDR implementation of the ASA. Together with theoretical contributions, this implementation has allowed the validation of the algorithm's expected performance over a wide-range of conditions based on current and expected proximity link conditions. Finally, the SDR implementation has allowed testing with real signals in a stand-alone platform, that have confirmed the computer simulations.

4.3 Future work

The possible avenues of future work can be grouped into two categories: further theoretical research and technology maturity.

From a theoretical standpoint, at very low signal-to-noise ratios remains should be studied further. Preliminary experiments suggest that adding a second integrator after the Lock and Direction Detector block (Figure 1.10) can help in these scenarios. This, combined with a 3rd order PLL that can track higher Doppler dynamics, would allow the system to work in more challenging communications scenarios than proximity links, such as in entry, descent, and landing.

Finally, in the technology maturity, the positive results obtained with ASA's software-based implementation running on a stand-alone SDR have opened to door to developing a more capable hardware-based implementation. In particular, the FPGA capabilities of the Ettus E310 have not been fully used, the whole system could be implemented on the FPGA, and taken to a DSN antenna for a real-time demonstration. Additionally, the FPGA-based implementation of ASA could be compared in performance with Electra's implementation of a fixed-size step algorithm.

Acronyms

ASA Adaptive-Sweep Algorithm. 10–12, 21, 27, 32, 33, 35–38, 52, 56, 59

AWGN Additive White Gaussian Noise. i, 3, 4, 10, 12, 15, 16, 23, 26, 37, 46

BLF Best-Lock Frequency. 2

BPSK Binary Phase Shift Keying. 3

CPU Central Processing Unit. 2

CU Continuous Update. 6, 14, 28

DOF Degrees Of Freedom. 13, 14

DSN Deep Space Network. 1, 2, 4, 13, 32, 33, 38, 41, 49

FPGA Field Programmable Gate Array. 2, 26, 38

GRC GNU Radio Companion. 25, 26

IIR Infinite Impulse Response. 5

JPL Jet Propulsion Laboratory. 1, 2, 41

LRO Lunar Reconnaissance Orbiter. i, 1, 4, 10, 11, 27, 31–33, 36, 37, 41

MLE Maximum Likelihood Estimator. 19, 20

MUPA Multiple Uplinks per Antenna. 2

NCO Numerically Controlled Oscillator. 5, 22, 29

OOT Out Of the Tree. 47

PDF Probability Distribution Function. 15, 16

PLL Phase-Locked Loop. 3–8, 11–15, 22, 26, 28, 29, 38

SDR Software-Defined Radio. 2, 3, 10–12, 32, 33, 36–38

SNR Signal to Noise Ratio. i, 12, 13, 15, 20, 32, 35

STFT Short Time Fourier Transform. 32, 35

Bibliography

[1] Johnathan Corgan. "GNU Radio Runtime Operation". In: *Proceedings of GR-CON15*. Aug. 2015.

[2] D. Divsalar, M. S. Net, and K. Cheung. "Adaptive Sweeping Carrier Acquisition and Tracking for Dynamic Links with High Uplink Doppler". In: *2020 IEEE Aerospace Conference*. Big Sky, Montana, Mar. 2020.

[3] NASA-Jet Propulsion Laboratory. *About the Deep Space Network*. URL: https://deepspace.jpl.nasa.gov/about (visited on 05/20/2020).

[4] Andrew O'Dea et al. "Doppler Tracking". In: *DSN Telecommunications Link Design Handbook*. Jet Propulsion Laboratory, 2019. URL: https://deepspace.jpl.nasa.gov/dsndocs/810-005/202/202C.pdf.

[5] Edgar Satorius et al. "The Electra Radio". In: *Autonomous Software-Defined Radio Receivers for Deep Space Applications*. John Wiley & Sons, Ltd, 2006. Chap. 2, pp. 19–43. ISBN: 9780470087800. DOI: 10.1002/9780470087800.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470087800.ch2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470087800.ch2.

[6] Thomas M. Sellke and Sarah H. Sellke. "Chebyshev Inequalities for Unimodal Distributions". In: *The American Statistician* 51.1 (1997), pp. 34–40. ISSN: 00031305. DOI: 10.2307/2684690.

[7] Marvin K. Simon and Jon Hamkins. "Carrier Synchronization". In: *Autonomous Software-Defined Radio Receivers for Deep Space Applications*. John Wiley & Sons, Ltd, 2006. Chap. 8, pp. 227–270. ISBN: 9780470087800. DOI: 10.1002/9780470087800.ch8. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470087800.ch8. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470087800.ch8.

[8] S.A. Stephens and J.B. Thomas. "Controlled-Root Formulation for Digital Phase-Locked Loops". In: *Aerospace and Electronic Systems, IEEE Transactions on* 31 (Feb. 1995), pp. 78–95. DOI: 10.1109/7.366295.

[9] Tore Ulversoy. "Software Defined Radio: Challenges and Opportunities". In: *IEEE Communications Surveys and Tutorials* 12 (Dec. 2010), pp. 531–550. DOI: 10.1109/SURV.2010.032910.00019.

[10] GNU Radio Wiki. *FAQ: Why can't we do loops?* May 13, 2020. URL: https://wiki.gnuradio.org/index.php/FAQ#Why_can.27t_we_do_loops.3F (visited on 05/20/2020).

Appendix A

Doppler shift

The Doppler shift is a well known phenomena, where the received frequency of a signal is modified by the relative position and velocity of the transmitter. For radio waves, the formula is

$$\frac{f_{Rx}}{f_{Tx}} = 1 - \frac{\vec{v} \cdot \vec{u}}{c}$$

where c is the speed of light, and

$$\vec{v} = \vec{v_{Rx}} - \vec{v_{Tx}}$$
$$\vec{u} = \frac{\vec{r}_{Rx} - \vec{r}_{Tx}}{||\vec{r}_{Rx} - \vec{r}_{Tx}||}$$

In the context of spacecraft radios, past research suggests that the frequency shift can be locally modelled using an offset and a Doppler rate parameter r [5]. In other words, one can model

$$f_d(t) = f_{Rx}(t) - f_{Tx}(t) = f_0 + rt$$

For example, the work done in [2] uses $f_0 = 20$ kHz and $r = 100$ Hz/s.

In order to justify these values and assert the goodness of this approximation, one can compute the trajectory of a spacecraft and calculate the Doppler shift in time with respect to a DSN station.

The trajectory of a spacecraft and its relative position and velocity with a DSN station can be computed using the SPICE Toolkit. This has been developed by JPL in order to help scientists obtain data from spacecraft.

Using the code found in Listing D.2, and the publicly available data from SPICE, the trajectory of the LRO has been computed. Knowing the trajectory, the Doppler shift read from the DSN Canberra Station has also been calculated. The calculations have been done without taking into account the periods of occultation. For a plot of the Doppler shift read in two lunar orbits (113 minutes each), see Figure 1.1. Since the adaptive algorithm attaches to a $f_d(t)$ modelled as a ramp in less than 10s, one can observe that modelling $f_d(t)$ as a ramp is reasonable.

Knowing that the LRO transmits at a highest carrier Frequency of $f_c = 2.2712$ GHz, one can compute the maximum Doppler shift and Doppler rate in a lunar month. The plots of the Doppler shift and its derivative (the Doppler rate) can be found in Figure A.1 and Figure A.2.

It is interesting to observe the periodic nature of this Doppler shift, which has a maximum offset of 15.455 kHz and a maximum rate of 11.69 Hz/s. The maximum

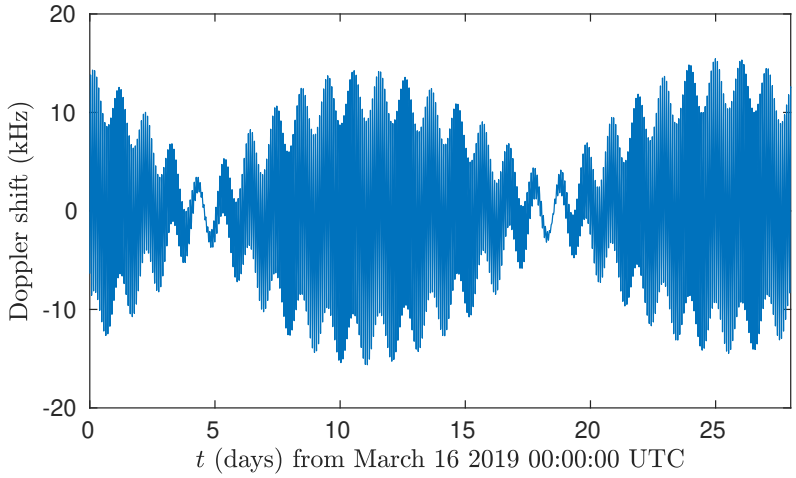

Figure A.1: LRO Doppler shift from Canberra during a whole lunar month

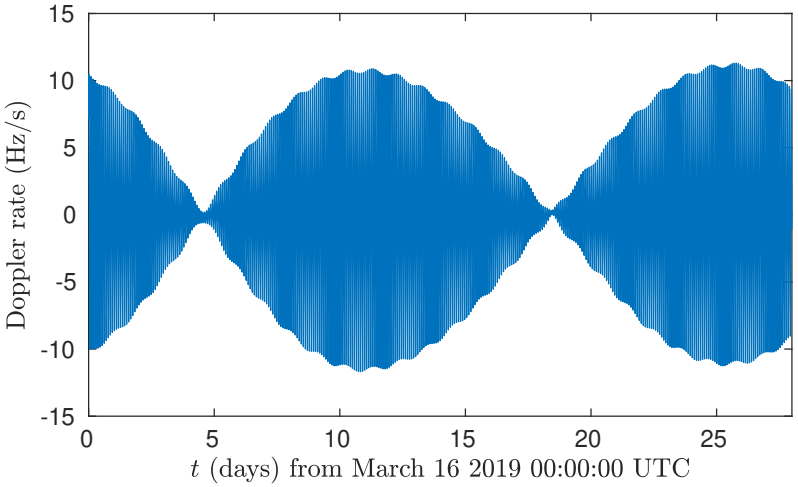

Figure A.2: LRO Doppler rate from Canberra during a whole lunar month

Doppler acceleration (second derivative of the shift) is 10.765 mHz/s^2, further confirming that modelling $f_d(t)$ as a ramp is a good local approximation for this work's study. These values also confirm that the values used in [2] accurately portray a possible Doppler shift.

If one computes the transform of the Doppler shift $f_d(t)$, in Figure A.3, then the periodicity of the signal can be observed even better. As one might expect, there are only two spectral components: the one corresponding to the orbit of the LRO around the Moon, and the one corresponding to the orbit of the Moon around the Earth.

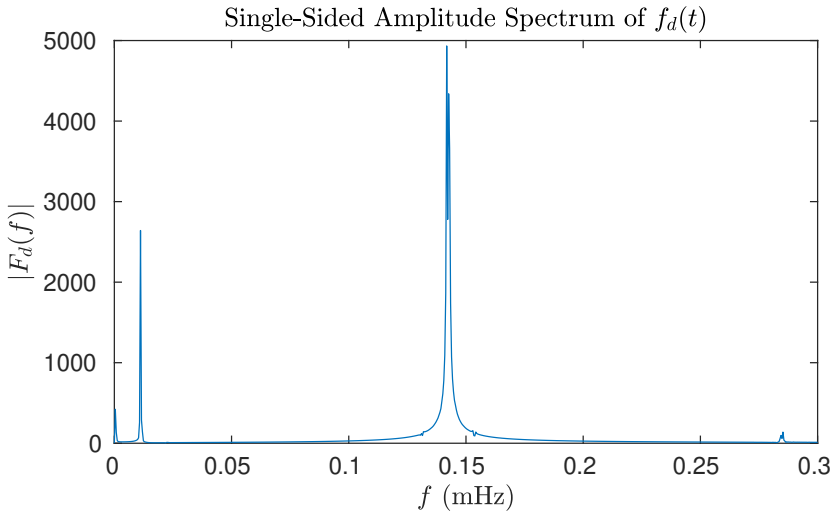

Figure A.3: Zoomed in Fourier transform of LRO Doppler shift from Canberra

Appendix B

Extended PLL residual error analysis

The PLL residual error can be measured in phase or in frequency. The analysis for frequency residuals from subsection 2.2.6 can be extended for higher order PLLs, as is done in section B.1. An analogous procedure will give the residual phase error, which is deduced in section B.2.

B.1 Residual frequency error

The procedure outlined for second order loops is analogous for third order loops. First, one obtains the transfer function

$$H(z) = \frac{K_3 z^2 + K_1(z-1)^2 + K_2 z(z-1)}{(z-1)^3 + K_3 z^2 + K_1(z-1)^2 + K_2 z(z-1)}$$

Then, the recurrence equation

$$\hat{f}_{k+3} + \hat{f}_{k+2}(-3 + K_1 + K_2 + K_3) + \hat{f}_{k+1}(3 - 2K_1 - K_2) + \hat{f}_k(K_1 - 1) =$$
$$n_{k+3}(K_1 + K_2 + K_3) + n_{k+2}(-K_3 - 3K_1 - 2K_2) + n_{k+1}(3K_1 + K_2) + n_k(-K_1) + \sum_{i=0}^{k-1} \lambda_i n_i$$

After, the covariance coefficients

$$\begin{aligned}
C_0 &= K_1 + K_2 + K_3 \\
C_1 &= -3K_1 - 2K_2 - K_3 - (K_1 + K_2 + K_3)(K_1 + K_2 + K_3 - 3) \\
C_2 &= 3K_1 + K_2 + (K_1 + K_2 + K_3 - 3)(3K_1 + 2K_2 + K_3 \\
&\quad + (K_1 + K_2 + K_3)(K_1 + K_2 + K_3 - 3)) + (K_1 + K_2 + K_3)(2K_1 + K_2 - 3) \\
C_3 &= -K_1 - (K_1 - 1)(K_1 + K_2 + K_3) - (K_1 + K_2 + K_3 - 3)(3K_1 + K_2 + (K_1 + K_2 + K_3 - 3) \\
&\quad (3K_1 + 2K_2 + K_3 + (K_1 + K_2 + K_3)(K_1 + K_2 + K_3 - 3)) + (K_1 + K_2 + K_3)(2K_1 + K_2 - 3)) \\
&\quad - (2K_1 + K_2 - 3)(3K_1 + 2K_2 + K_3 + (K_1 + K_2 + K_3)(K_1 + K_2 + K_3 - 3))
\end{aligned}$$

And finally, one solves the system and obtains R_0

$$R_0 = -2\frac{4K_1^3K_2 + 4K_1^3K_3 + 6K_1^2K_2^2 + 11K_1^2K_2K_3 + 5K_1^2K_3^2 - 4K_1^2K_3 + 2K_1K_2^3}{(4K_1^2K_2 + 4K_1^2K_3 + 2K_1K_2^2 + 3K_1K_2K_3 - 8K_1K_2 + K_1K_3^2 - 12K_1K_3 - 2K_2K_3 - K_3^2 + 8K_3)} +$$
$$-2\frac{5K_1K_2^2K_3 + 4K_1K_2K_3^2 - 8K_1K_2K_3 + K_1K_3^3 - 7K_1K_3^2 + 2K_2^3 + 3K_2^2K_3 + K_2K_3^2 + 2K_3^2}{(4K_1^2K_2 + 4K_1^2K_3 + 2K_1K_2^2 + 3K_1K_2K_3 - 8K_1K_2 + K_1K_3^2 - 12K_1K_3 - 2K_2K_3 - K_3^2 + 8K_3)}$$

To check if the result is correct, one can substitute $K_3 = 0$, and obtain

$$R_0 = -\frac{4K_1^3 + 6K_1^2K_2 + 2K_1K_2^2 + 2K_2^2}{K_1(2K_1 + K_2 - 4)}$$

Which is the same result that was obtained for a second order loop.

B.1.1 Taylor expansion with critically damped coefficients

The Taylor expansion will be calculated at $K_1 = 0$, and the rest of the filter coefficients will be taken for a critically damped response, as explained in [8].

For first order loops, the formula for R_0 is

$$R_0 = \frac{2K_1}{2 - K_1}$$
$$R_0 = K_1 + \frac{K_1^2}{2} + O(K_1^4)$$

For second order loops, $K_2 = \frac{1}{4}K_1^2$

$$R_0 = -\frac{K_1^2(K_1^2 + 13K_1 + 32)}{2K_1^2 + 16K_1 - 32}$$
$$R_0 = K_1^2 + \frac{29K_1^3}{32} + O(K_1^3)$$

For third order loops, $K_2 = \frac{1}{3}K_1^2$, $K_3 = \frac{1}{27}K_1^3$

$$R_0 = -\frac{2K_1^2(K_1^5 + 36K_1^4 + 549K_1^3 + 4185K_1^2 + 15606K_1 + 23328)}{27(K_1^4 + 26K_1^3 + 252K_1^2 + 648K_1 - 1728)}$$
$$R_0 = K_1^2 + \frac{451K_1^3}{432} + O(K_1^4)$$

B.2 Residual phase error

The signal after the mixer, modelled in Equation 2.15, is

$$e^{j(\phi(t)-\hat{\phi}(t))} + \tilde{n}(t)$$

Following the same procedure as in subsection 2.2.6, the signal after the phase extractor is

$$\phi_k - \hat{\phi}_k + n_k,$$

where n_k is AWGN with variance $\sigma^2 = \frac{1}{2SNR_c}$. Assuming a perfect estimation of the phase, the only error term left would be n_k. This estimation is compared to the measured phase error in Figure B.1, and as one can observe, it is very accurate.

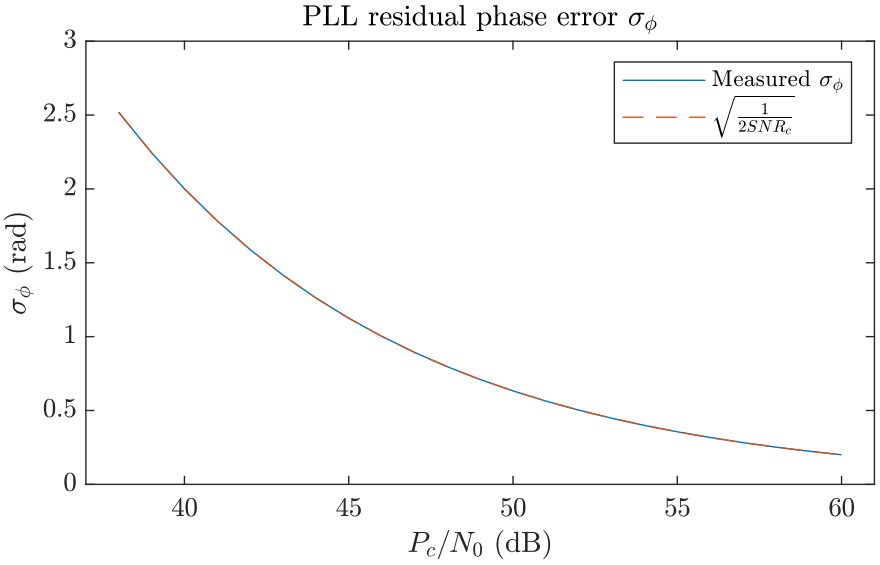

Figure B.1: Residual phase error and its estimation

Appendix C

Stand-alone SDR setup

To run GNU Radio with OOT blocks in Ettus E310, one must first follow the general setup instructions provided by Ettus, which can be found *here*.

To install an out of the tree block, an analogous procedure is followed. First, on the E310, a new install folder is created, and it is mounted via ssh to the user install folder on the PC. It is important to source the setup environment. To do that, type the following commands on the E310

```
mkdir -p ~/newinstall
sshfs username@192.168.10.1:/home/user/rfnoc/e300 newinstall/
cd ~/newinstall
source ./setup.env
```

Then, on the computer, the block is cross compiled in the previously mounted folder on the PC with the following commands

```
cp -r gr-your_block ~\rfnoc\src\
cd ~\rfnoc\src\gr-your_block
mkdir build-arm
cd build-arm
cmake -DCMAKE_TOOLCHAIN_FILE=~/rfnoc/src/gnuradio/cmake/Toolchains/oe-sdk_cross.cmake -DCMAKE_INSTALL_PREFIX=/usr ..
make -j4
make install DESTDIR=~/rfnoc/e300/
make install DESTDIR=~/rfnoc/oe/sysroots/armv7ahf-vfp-neon-oe-linux-gnueabi/
```

Then, on the E310, the contents of the mounted folder are copied into the local installation. The mounted folder contained the cross-compiled OOT module. The necessary commands on the E310 are the following

```
cd ~/localinstall
cp -v ~/newinstall/setup.env .
cp -Rv ~/newinstall/etc .
cp -Rv ~/newinstall/usr .
sed -i 's/newinstall/localinstall/g' setup.env
source ./setup.env
cd ~/
umount ~/newinstall
rm -rf ~/newinstall
```

Remember to run `source ~/localinstall/setup.env` before running a GNU Radio Python script on the E310.

Appendix D

Code listings

This appendix contains part of the code that was used to obtain the plots for this thesis. Every figure that is presented in this work can be obtained with a fragment of code found below or with a small variation. Every listing is accompanied by a small description and a reference to the content it has generated.

Listing D.1 runs a PLL simulink model, not shown, and collects the output frequency to obtain the residual error. In order to assess the response of the system, this is done for a variety of input signals. The resulting plot can be seen in Figure 3.1.

```
clear; clc; close all;
N = 2; % PLL integrator size
M = 25*N; % Sweep branch integrator size
inmode = 1; % 0 fd(t) = rt, 1 fd(t) = sin(2*pi*nu*t)
nu = 1;
r = 0; % Doppler rate
fo = 0; % Frequency offset
BL = 50;
fs = 32e4;
fpll = fs/N;
Ts = 1/fs;
T = N/fs;
assert(BL*T <= 0.02, "CU approximation failed")
%values for supercritically damped response
K1 = (16/5)*BL*T;
K2 = 1/4*K1*K1;
PcNo = 6500; % No noise
SNRc = PcNo - 10*log10(fs);
nus = linspace(1,100,20);
derrs = 1:length(nus);
timesim = 0.1;
for i = 1:length(nus)
    disp(i)
    nu = nus(i); %Doppler fd(t) = sin(2*pi*nu*t)
    simi = sim('justpllsim',timesim);
    fdt = simi.fdt.Data;
    est = simi.f.Data(1,:)';
    derrs(i)=norm(fdt-est)^2;
end
nsamp = timesim*fs/N;
plot(nus, sqrt(derrs/nsamp))
xlabel('$\nu$ (Hz)', 'Interpreter', 'latex')
ylabel('Expected $\left \| f_d(t) - \hat{f_d(t)} \right \|$', '
    Interpreter', 'latex')
```

Listing D.1: Code used for Figure 3.1

Listing D.2 first loads some of NASA's NAIF kernels, which are publicly available at `https://naif.jpl.nasa.gov/pub/naif/`. Based on this data, the Doppler shift between the LRO and the Canberra DSN station is calculated, along with the periods of lunar occultation. The resulting plot can be seen in Figure 1.1. The rest of the plots in Appendix A have been obtained with variations of this code.

```
% Compute trajectory of LRO with respect to ground station
% From there, compute Doppler shift in time
cspice_kclear; clear; clc; close all;

kernel_folder = '/home/tomas/Documents/JPL/mice/data/my_kernels/';
leap_seconds = strcat(kernel_folder,'latest_leapseconds.tls');
lro_position = strcat(kernel_folder,'lrorg_2019074_2019166_v01.bsp');
planets_position = strcat(kernel_folder,'de430.bsp'); %to use Earth as
    observer
earth_pck_for_dsn = strcat(kernel_folder, 'earth_latest_high_prec.bpc')
    ;
dsn_spk = strcat(kernel_folder, 'earthstns_itrf93_050714.bsp');
general_pck = strcat(kernel_folder, 'pck00010.tpc');

cspice_furnsh(leap_seconds);
cspice_furnsh(planets_position);
cspice_furnsh(lro_position);
cspice_furnsh(earth_pck_for_dsn);
cspice_furnsh(dsn_spk);
cspice_furnsh(general_pck);

fc = 2271.2e6; % from https://directory.eoportal.org/web/eoportal/
    satellite-missions/l/lro
% Array of ephemeris, starting x
start = cspice_str2et('March 16 2019 UTC');
h = 1; %space between samples
lunar_orbit_sec = 2*114*60;
t = 0:(lunar_orbit_sec-1);
et = t*h + start;
reference_frame = 'J2000';
etend = cspice_str2et('2019-03-16T03:00:00.00');
target = 'LUNAR RECONNAISSANCE ORBITER';
observer = 'DSS-43';
occulted = cspice_occult (target, 'point', ' ', 'Moon', 'ellipsoid', '
    iau_moon', 'LT+S', observer, et) < 0;
[state, ~] = cspice_spkezr(target, et, reference_frame, 'LT+S',
    observer);
%LRO tx, DSN-43 rx (Canberra)
%vectors are defined from Earth to LRO
v = state(4:6, :) * 1e3;
u = state(1:3, :);
m = length(et);
f = ones(1,m);
c = 299792458.0;
for j = 1:m
    unorm = u(1:3,j) / norm(u(1:3,j));
    f(j) = 1 - (-v(1:3, j)')*(-unorm(1:3))/c;
    if (occulted(j))
        f(j) = nan;
```

```
    end
end
plot(t/60, (f-1)*fc/1000) % plot time in minutes and frequency in kHz
grid on
ylabel('Doppler shift (kHz)','Interpreter', 'latex');
xlabel('$t$ (min) from March 16 2019 00:00:00 UTC','Interpreter', '
    latex');
```

Listing D.2: Code used for LRO trajectory plot with occultation in Figure 1.1

Listing D.3 runs a GNU Radio flow graph that is used to obtain the noise P and D plots in Figure 2.1. The P and D block has been programmed internally in C++, its behaviour is self-explanatory based on the theory discussed in subsection 1.3.3.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: P and D simulation
# Author: Tomas Ortega
# GNU Radio version: 3.7.14.0
##################################################

from gnuradio import blocks
from gnuradio import channels
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from math import pi
from optparse import OptionParser
import inputgenerator
import pdsim
import numpy as np


class PandD(gr.top_block):

    def __init__(self, M, samp_rate, df, SNR):
        gr.top_block.__init__(self, "P and D simulation")

        ##################################################
        # Variables
        ##################################################
        self.M = M
        self.samp_rate = samp_rate
        self.n_samp = n_samp = 3*int(M)
        self.df = df
        self.SNR = SNR

        ##################################################
        # Blocks
        ##################################################
        self.vec_source = blocks.vector_source_i([1]*n_samp, False, 1,
    [])
        self.real = blocks.complex_to_real(1)
        self.p_and_d = pdsim.pandd(M, int(samp_rate))
        self.one_in_M = blocks.keep_one_in_n(gr.sizeof_gr_complex*1, M)
        self.noise_channel = channels.channel_model(
```

```
45              noise_voltage=10**(-SNR/20),
46              frequency_offset=0.0,
47              epsilon=1.0,
48              taps=(1.0 , ),
49              noise_seed=int(df*1000),
50              block_tags=False
51          )
52          self.ingenerator = inputgenerator.DopplerGenerator(int(
    samp_rate), 0, df)
53          self.imag = blocks.complex_to_imag(1)
54          self.blocks_vector_sink_P = blocks.vector_sink_f(1, 1024)
55          self.blocks_vector_sink_D = blocks.vector_sink_f(1, 1024)
56
57          ##################################################
58          # Connections
59          ##################################################
60          self.connect((self.imag, 0), (self.blocks_vector_sink_D, 0))
61          self.connect((self.ingenerator, 0), (self.noise_channel, 0))
62          self.connect((self.noise_channel, 0), (self.p_and_d, 0))
63          self.connect((self.one_in_M, 0), (self.imag, 0))
64          self.connect((self.one_in_M, 0), (self.real, 0))
65          self.connect((self.p_and_d, 0), (self.one_in_M, 0))
66          self.connect((self.real, 0), (self.blocks_vector_sink_P, 0))
67          self.connect((self.vec_source, 0), (self.ingenerator, 0))
68
69      def get_M(self):
70          return self.M
71
72      def set_M(self, M):
73          self.M = M
74          self.set_n_samp(3*int(self.M))
75          self.one_in_M.set_n(self.M)
76
77      def get_samp_rate(self):
78          return self.samp_rate
79
80      def set_samp_rate(self, samp_rate):
81          self.samp_rate = samp_rate
82
83      def get_n_samp(self):
84          return self.n_samp
85
86      def set_n_samp(self, n_samp):
87          self.n_samp = n_samp
88          self.vec_source.set_data([1]*self.n_samp, [])
89
90      def get_df(self):
91          return self.df
92
93      def set_df(self, df):
94          self.df = df
95
96      def get_SNR(self):
97          return self.SNR
98
99      def set_SNR(self, SNR):
100         self.SNR = SNR
101         self.noise_channel.set_noise_voltage(10**(-self.SNR/20))
```

```
102
103
104 def getpd(top_block_cls, df, M, samp_rate, SNR):
105     tb = top_block_cls(M, samp_rate, df, SNR)
106     tb.start()
107     tb.wait()
108     P = tb.blocks_vector_sink_P.data()[-1]
109     D = tb.blocks_vector_sink_D.data()[-1]
110     return [P, D]
111
112
113 def main(top_block_cls=PandD, options=None):
114
115     import matplotlib.pyplot as plt
116     import matplotlib as mpl
117     mpl.rcParams['mathtext.fontset'] = 'cm'
118
119     SNRm = 17
120     M = 993
121     samp_rate = 125e3
122     PcNo = SNRm + 10*np.log10(samp_rate/M)
123     SNR = PcNo - 10*np.log10(samp_rate)
124     df_min = -  1/1.0*samp_rate/M
125     df_max = -df_min
126     df_range = np.linspace(df_min, df_max, num=300)
127     [P,D] = np.array([getpd(top_block_cls, float(x), M, samp_rate, SNR)
        for x in df_range]).transpose()
128     Pt = np.array([np.cos(x*2*np.pi)*(np.sinc(x)**2) for x in (df_range
    *M/samp_rate)])
129     Dt = np.array([np.sin(x*2*np.pi)*(np.sinc(x)**2) for x in (df_range
    *M/samp_rate)])
130     fig, ax = plt.subplots()
131     ax.plot(df_range*M/samp_rate*pi, P)
132     ax.plot(df_range*M/samp_rate*pi, Pt, '--')
133     ax.plot(df_range*M/samp_rate*pi, D)
134     ax.plot(df_range*M/samp_rate*pi, Dt, '--')
135     ax.set(xlabel=r'$\alpha = \pi \Delta f T$')
136     ax.legend([r'$P$', r'$P_{ideal}$', r'$D$', r'$D_{ideal}$'])
137     ax.grid()
138     plt.xlim((-np.pi, np.pi))
139     print 'PcNo:', PcNo
140     fig.savefig("noisypd.eps")
141     plt.show()
142
143
144 if __name__ == '__main__':
145     main()
```

Listing D.3: Code used for noisy P and D plots, see Figure 2.1

Listing D.4 contains the GNU Radio flow graph of the full system. It has been used to obtain the majority of plots in this thesis, and is also the code that has been run in the Ettus E310. The code for the C++ code of the ASA block can be found in Listing D.5.

```
1 #!/usr/bin/env python2
2 # -*- coding: utf-8 -*-
3 ##################################################
4 # GNU Radio Python Flow Graph
```

```
# Title: Full ASA system
# Author: Tomas Ortega
# GNU Radio version: 3.7.14.0
##################################################

from gnuradio import blocks
from gnuradio import channels
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
import EASA
import inputgenerator
import numpy as np


class fullpll(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Full ASA system")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 10e6
        self.PcNo = PcNo = 38
        self.SNR = SNR = PcNo - 10*np.log10(samp_rate)
        self.M1 = M1 = 79433
        self.SNRm = SNRm = SNR + 10*np.log10(M1)
        self.M2 = M2 = 1
        self.BL = BL = 50
        self.varfinal = varfinal = 1.0/M2 *(1.0/(10**(SNRm/10)) +
    0.5*1.0/(10**(SNRm/10))**2 )
        self.s_simulation = s_simulation = 30
        self.offset = offset = 20e3
        self.doppler_rate = doppler_rate = 100
        self.N = N = 2018
        self.LoopSNR = LoopSNR = PcNo - 10*np.log10(BL)

        ##################################################
        # Blocks
        ##################################################
        self.vector_sample_source = blocks.vector_source_i([1]* (int(
    samp_rate)), False, 1, [])
        self.throttle_block = blocks.throttle(gr.sizeof_int*1, 5e9,True
    )
        self.sink_p = blocks.vector_sink_f(1, 1024)
        self.sink_f = blocks.vector_sink_f(1, 1024)
        self.repeat_number_of_seconds = blocks.repeat(gr.sizeof_int*1,
    s_simulation)
        self.noisy_channel = channels.channel_model(
          noise_voltage=10**(-SNR/20),
          frequency_offset=0.0,
          epsilon=1.0,
          taps=(1.0 , ),
          noise_seed=0,
          block_tags=False
```

```
59          )
60          self.keep_one_in_n_f = blocks.keep_one_in_n(gr.sizeof_float*1,
    N)
61          self.keep_one_in_m1m2_p = blocks.keep_one_in_n(gr.sizeof_float
    *1, M1*M2)
62          self.keep_one_in_m1m2_d = blocks.keep_one_in_n(gr.sizeof_float
    *1, M1*M2)
63          self.file_sink_p = blocks.file_sink(gr.sizeof_float*1, '/home/
    tomas/Downloads/outputP.bin', False)
64          self.file_sink_p.set_unbuffered(False)
65          self.file_sink_f = blocks.file_sink(gr.sizeof_float*1, '/home/
    tomas/Downloads/outputF.bin', False)
66          self.file_sink_f.set_unbuffered(False)
67          self.file_sink_d = blocks.file_sink(gr.sizeof_float*1, '/home/
    tomas/Downloads/outputD.bin', False)
68          self.file_sink_d.set_unbuffered(False)
69          self.doppler_generator = inputgenerator.DopplerGenerator(int(
    samp_rate), doppler_rate, offset)
70          self.blocks_null_sink_0 = blocks.null_sink(gr.sizeof_gr_complex
    *1)
71          self.ASA_block = EASA.EASA(int(samp_rate), N, M1, M2, BL, 0.2,
    0, 0, 40e3, samp_rate*0.6719/np.pi/(M1), min(0.3793*np.exp(0.1157*
    SNRm), samp_rate*0.6719/np.pi/(M1)), min(0.3793*np.exp(0.1157*SNRm)
    , samp_rate*0.6719/np.pi/(M1)))
72
73
74
75          ##################################################
76          # Connections
77          ##################################################
78          self.connect((self.ASA_block, 0), (self.blocks_null_sink_0, 0))
79          self.connect((self.ASA_block, 3), (self.keep_one_in_m1m2_d, 0))
80          self.connect((self.ASA_block, 2), (self.keep_one_in_m1m2_p, 0))
81          self.connect((self.ASA_block, 1), (self.keep_one_in_n_f, 0))
82          self.connect((self.doppler_generator, 0), (self.noisy_channel,
    0))
83          self.connect((self.keep_one_in_m1m2_d, 0), (self.file_sink_d,
    0))
84          self.connect((self.keep_one_in_m1m2_p, 0), (self.file_sink_p,
    0))
85          self.connect((self.keep_one_in_m1m2_p, 0), (self.sink_p, 0))
86          self.connect((self.keep_one_in_n_f, 0), (self.file_sink_f, 0))
87          self.connect((self.keep_one_in_n_f, 0), (self.sink_f, 0))
88          self.connect((self.noisy_channel, 0), (self.ASA_block, 0))
89          self.connect((self.repeat_number_of_seconds, 0), (self.
    throttle_block, 0))
90          self.connect((self.throttle_block, 0), (self.doppler_generator,
     0))
91          self.connect((self.vector_sample_source, 0), (self.
    repeat_number_of_seconds, 0))
92
93      def get_samp_rate(self):
94          return self.samp_rate
95
96      def set_samp_rate(self, samp_rate):
97          self.samp_rate = samp_rate
98          self.set_SNR(self.PcNo - 10*np.log10(self.samp_rate))
99          self.vector_sample_source.set_data([1]* (int(self.samp_rate)),
```

```
    [])

    def get_PcNo(self):
        return self.PcNo

    def set_PcNo(self, PcNo):
        self.PcNo = PcNo
        self.set_SNR(self.PcNo - 10*np.log10(self.samp_rate))
        self.set_LoopSNR(self.PcNo - 10*np.log10(self.BL))

    def get_SNR(self):
        return self.SNR

    def set_SNR(self, SNR):
        self.SNR = SNR
        self.set_SNRm(self.SNR + 10*np.log10(self.M1))
        self.noisy_channel.set_noise_voltage(10**(-self.SNR/20))

    def get_M1(self):
        return self.M1

    def set_M1(self, M1):
        self.M1 = M1
        self.set_SNRm(self.SNR + 10*np.log10(self.M1))
        self.keep_one_in_m1m2_p.set_n(self.M1*self.M2)
        self.keep_one_in_m1m2_d.set_n(self.M1*self.M2)

    def get_SNRm(self):
        return self.SNRm

    def set_SNRm(self, SNRm):
        self.SNRm = SNRm
        self.set_varfinal(1.0/self.M2 *(1.0/(10**(self.SNRm/10)) +
    0.5*1.0/(10**(self.SNRm/10))**2 ))

    def get_M2(self):
        return self.M2

    def set_M2(self, M2):
        self.M2 = M2
        self.set_varfinal(1.0/self.M2 *(1.0/(10**(self.SNRm/10)) +
    0.5*1.0/(10**(self.SNRm/10))**2 ))
        self.keep_one_in_m1m2_p.set_n(self.M1*self.M2)
        self.keep_one_in_m1m2_d.set_n(self.M1*self.M2)

    def get_BL(self):
        return self.BL

    def set_BL(self, BL):
        self.BL = BL
        self.set_LoopSNR(self.PcNo - 10*np.log10(self.BL))

    def get_varfinal(self):
        return self.varfinal

    def set_varfinal(self, varfinal):
        self.varfinal = varfinal

```

```
155     def get_s_simulation(self):
156         return self.s_simulation
157 
158     def set_s_simulation(self, s_simulation):
159         self.s_simulation = s_simulation
160         self.repeat_number_of_seconds.set_interpolation(self.
    s_simulation)
161 
162     def get_offset(self):
163         return self.offset
164 
165     def set_offset(self, offset):
166         self.offset = offset
167 
168     def get_doppler_rate(self):
169         return self.doppler_rate
170 
171     def set_doppler_rate(self, doppler_rate):
172         self.doppler_rate = doppler_rate
173 
174     def get_N(self):
175         return self.N
176 
177     def set_N(self, N):
178         self.N = N
179         self.keep_one_in_n_f.set_n(self.N)
180 
181     def get_LoopSNR(self):
182         return self.LoopSNR
183 
184     def set_LoopSNR(self, LoopSNR):
185         self.LoopSNR = LoopSNR
186 
187 
188 def main(top_block_cls=fullpll, options=None):
189 
190     tb = top_block_cls()
191     tb.start()
192     tb.wait()
193 
194 
195 if __name__ == '__main__':
196     main()
```

Listing D.4: Python GNU Radio program for ASA

```
1  /* -*- c++ -*- */
2  /*
3   * Copyright 2020 Tomas Ortega.
4   */
5 
6  #ifdef HAVE_CONFIG_H
7  #include "config.h"
8  #endif
9 
10 #include <gnuradio/io_signature.h>
11 #include "EASA_impl.h"
12 
```

```
13 #include <math.h>
14 #include <gnuradio/sincos.h>
15 #define M_TWOPI (2.0 * M_PI)
16
17 namespace gr {
18   namespace EASA {
19
20     EASA::sptr
21     EASA::make(int msamp_rate, int mN, int mM1, int mM2, float mBL,
    float mth1, float mth2, float mf1, float mf2, float mstep1, float
    mstep2, float mstep3) {
22       return gnuradio::get_initial_sptr (new EASA_impl(msamp_rate, mN,
    mM1, mM2, mBL, mth1, mth2, mf1, mf2, mstep1, mstep2, mstep3));
23     }
24
25     /*
26      * The private constructor
27      */
28     EASA_impl::EASA_impl(int msamp_rate, int mN, int mM1, int mM2,
    float mBL, float mth1, float mth2, float mf1, float mf2, float
    mstep1, float mstep2, float mstep3) : gr::sync_block("EASA",
29             gr::io_signature::make(1, 1, sizeof(gr_complex)),
30             gr::io_signature::make2(4, 4, sizeof(gr_complex), sizeof(
    float))) {
31         //Global
32         samp_rate = msamp_rate;
33         //PLL buffer
34         N = mN;
35         pll_buf = 0.0;
36         pll_buf_out = 0.0;
37         pll_buf_index = 0;
38         //PLL
39         damp = 1.0;
40         out_f = 0.0;
41         pll_out = gr_complexd(0,0);
42         K1 = 16.0 * mBL * N / samp_rate /(1/(damp*damp) + 4);
43         K2 = 0.25 * K1 * K1 *damp*damp;
44         filter_sum = 0.0;
45         filter_out = 0.0;
46         nco_sum = 0.0;
47         //Lock and Direction detector
48         M1 = mM1;
49         M2 = mM2;
50         lock_buf1_index = 0;
51         lock_buf2_index = 0;
52         lock_buf1 = gr_complexd(0,0);
53         lock_buf1_old = gr_complexd(0,0);
54         lock_buf2 = gr_complexd(0,0);
55         lock_D = 0.0;
56         lock_P = 0.0;
57         //Sweep
58         th1 = mth1;
59         th2 = mth2;
60         step1 = mstep1;
61         step2 = mstep2;
62         step3 = mstep3;
63         f1 = mf1;
64         f2 = mf2;
```

```
65          fout = mf1;
66      }
67
68      /*
69       * Our virtual destructor.
70       */
71      EASA_impl::~EASA_impl() {}
72
73      /*
74       * Our pll function
75       */
76      void EASA_impl::pll_run() {
77          //get output from pll buffer, pass through filter, get sweep if
        it exists, update NCO output.
78          filter_sum += pll_buf_out;
79          filter_out = pll_buf_out*K1 + filter_sum*K2 + fout*N*M_TWOPI/
    samp_rate; //we add the sweep output
80          out_f = filter_out*samp_rate/N/M_TWOPI;
81      }
82
83      /*
84       * Our sweeping algorithm
85       */
86      void EASA_impl::sweeping_algorithm() {
87          //run a step of the sweep
88          if (lock_P < th1) {
89              fout += step1;
90          } else {
91              if (lock_D > 1) lock_D = 1;
92              else if (lock_D < -1) lock_D = -1;
93              if (lock_P < th2) fout += lock_D*step2;
94              else fout += lock_D*step3;
95          }
96          if (fout > f2) fout = f1;
97      }
98
99      /*
100      * Our lock detector
101      */
102     void EASA_impl::lock_detector(gr_complexd inp) {
103         //put the input in the buffer, if it is the last sample,
    restart counter, update output and trigger sweep
104         lock_buf1 += inp;
105         lock_buf1_index++;
106         if (lock_buf1_index == M1) {
107             lock_buf1 /= M1;
108             lock_buf2 += lock_buf1_old*lock_buf1;
109             lock_buf2_index++;
110             lock_buf1_old = gr_complexd(lock_buf1.real(), -lock_buf1.
    imag()); //conjugate
111             lock_buf1 = gr_complexd(0,0);
112             lock_buf1_index = 0;
113             if (lock_buf2_index == M2) {
114                 lock_buf2 /= M2;
115                 lock_D = lock_buf2.imag(); //no clipper used
116                 lock_P = lock_buf2.real();
117                 lock_buf2 = gr_complexd(0,0);
118                 lock_buf2_index = 0;
```

```
119                     sweeping_algorithm();
120                 }
121             }
122         }
123
124         /*
125          * Our pll function
126          */
127         void EASA_impl::pll_buf_run(float inp) {
128             //put the input in the buffer, if it is the last sample,
    restart counter, update output, trigger filter
129             pll_buf += inp;
130             pll_buf_index++;
131             if (pll_buf_index == N) {
132                 pll_buf_out = pll_buf / N;
133                 pll_buf_index = 0;
134                 pll_buf = 0.0;
135                 pll_run();
136             }
137             // NCO
138             nco_sum += filter_out / N;
139             while (nco_sum > M_PI) nco_sum -= M_TWOPI; //maybe not
    necessary to avoid overflow
140             float o_real, o_imag;
141             gr::sincosf(nco_sum, &o_imag, &o_real);
142             pll_out = gr_complexd(o_real, -o_imag); //conjugate
143         }
144
145         int EASA_impl::work(int noutput_items, gr_vector_const_void_star &
    input_items, gr_vector_void_star &output_items) {
146           const gr_complex *in = (const gr_complex *) input_items[0];
147           gr_complex *out0 = (gr_complex *) output_items[0];
148           float *out1 = (float *) output_items[1];
149           float *out2 = (float *) output_items[2];
150           float *out3 = (float *) output_items[3];
151           for (int i = 0; i < noutput_items; ++i) {
152               gr_complexd ind = gr_complexd(in[i].real(), in[i].imag()); //
    complex float to complex double
153               gr_complexd mult = ind *pll_out;
154               lock_detector(mult);
155               pll_buf_run(mult.imag());
156               out0[i] = mult;
157               out1[i] = out_f;
158               out2[i] = lock_P;
159               out3[i] = lock_D;
160           }
161           return noutput_items;
162         }
163
164     } /* namespace EASA */
165 } /* namespace gr */
```

Listing D.5: Main C++ code for ASA

Listing D.6 contains the final design script of the ASA described in Chapter 3. It encapsulates the theoretical work done in Chapter 2.

```
1 %% System input
2 % The system uses a critically damped PLL filter
```

```
max_sspe = 0.1; % maximum steady-state phase error (radians)
max_doppler_rate = 100; % maximum absolute value of doppler rate (Hz/s)
offset_range = 40e3; % maximum offset - minimum offset (Hz)
PcNo = dB2lin(38); % PcNo value
prob_error_step2 = 0.01; % error probability in step2, of value
    max_error_step2
max_error_step2 = 1; % maximum allowed error with prob_error_step2
    probability in step2 (Hz)
minfvariation = 0|0; % True: minimizes f variation. False: minimizes
    acq time

%% Parameter calculation
BL = sqrt(max_doppler_rate*2*pi*(5^2)/(max_sspe*(4^3))); % from the
    steady-state phase error equation
Fs = offset_range*2; % from Nyquist bound
N = 1; % N is the PLL integrator size, left at 1 by default
Fs = max(Fs, ceil(N*BL/0.01)); % from the continuous-time approximation
     equation
SNRc = PcNo/Fs; % from SNRc definition
M = ceil(dB2lin(17)/SNRc); % from minimum SNRm formula
Dfun = @(x) sin(2*x)*(sinc(x/pi)^2); % from the D function formula
alpha = abs(fminbnd(Dfun,-0.7,-0.6)); % from the D function linear zone
step2param = max_error_step2/qfuncinv(prob_error_step2*0.5); % from the
     error distribution formula
a = offset_range*0.5*pi/alpha;
b = alpha/(2*pi*step2param*sqrt(PcNo));
M = max(M, ceil((b/(4*a))^(2/5) * Fs)); % from the optimal expected
    acquisition time formula
if minfvariation
    N = max(floor(Fs*0.01/BL), 1); % to minimize f variation at output
    M = max(M, 10*N); % from Fpll >> Fsweep condition
else % Priority is acquisition time
    N = min(ceil(M/10), floor(Fs*0.01/BL)); % to minimize f variation
    at output
end
step1 = alpha*Fs/(pi*M); % from step1 formula
step2 = step2param*sqrt(SNRc*M); % from step2 formula
Pfun = @(x) cos(2*x)*(sinc(x/pi)^2); % from P function definition
threshold = Pfun(alpha); % from optimal threshold analysis
expected_acquisition_time = a*(M/Fs)^2 + b/sqrt(M/Fs); % from expected
    acquisition time formula
K1 = 16/5*BL*N/Fs; % From PLL paper, critically damped
K2 = 1/4 *K1*K1; % From PLL paper, critically damped
output_deviation = sqrt(1./SNRc/2.0/N*((K1 + K2)*Fs/N/(2*pi))^2); %
    from output frequency deviation analysis
```

Listing D.6: Design script