NTNU
Innovation and Creativity

Software-Defined GNSS Receiver based on Free Software Components

Trond Danielsen

Master of Science in Electronics

Submission date:	July 2007
Supervisor:	Børje Forssell, IET
Co-supervisor:	Steinar Brede, Telenor R&D

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

This task is a continuation of a previous project consisting of an evaluation of existing free software GNSS solutions. None of these solutions were found suitable for future work, so specifications for a software-defined receiver built on top of the GNU Radio software-defined radio framework were developed.

The current task includes implementation, testing and evaluation of the OpenGNSS receiver described in the project mentioned, and also to evaluate, implement and propose changes to the GNU Radio framework that would make it more adaptable to GNSS applications.

Assignment given: 15. January 2007
Supervisor: Børje Forssell, IET

Software-Defined GNSS Receiver based on Free Software Components

Trond Danielsen

July 6, 2007

Preface

This is the Master Thesis for Trond Danielsen, presented to the Norwegian University of Science and Technology as partial fulfillment of the requirement for obtaining a Master's Degree in signal processing and communication. The thesis has been written under the supervision of Professor Börje Forssell at the Department of Electronics and Telecommunications, and Steinar Brede from Telenor R&I in Trondheim.

The assignment has been to implement, test and evaluate parts of a software-defined satellite navigation receiver based on free software. The specification for the receiver was written in a previous assignment by the author himself. Personally it has been a privilege for me to be able to work on this assignment. I find free software very exciting, and to be able to combine this with satellite navigation has been the perfect topic for a thesis for me. I am therefore grateful for the freedom given to me by my supervisors which enabled me to shape my assignment as I wished.

Acknowledgement

First of all I would also like to thank my supervisors — Professor Börje Forssell and Steinar Brede — for their help and guidance.

I would also like to thank the GNU Radio community for their valuable support. Without their insightful answers on the GNU Radio mailing list, this would not have been possible. I would also like to thank all of those who work on the GNU Radio framework and the Universal Software Radio Peripheral both professionally and on a voluntary on their free time.

I also would like to thank the thousands of people who spend their time working on free software. Isaac Newton has expressed what I feel better than what I am able to:

> If I have seen a little further, it is by standing on the shoulders of Giants.

And last, but not the least, I would like to thank my fiancé Cecilie for all her love and care.

Abstract

In this paper an acquisition module for the OpenGNSS software recevier is discussed. OpenGNSS is built on top of the GNU Radio SDR framework, which is a free software framework for building software-defined radio. The overall task is to create a complete software-defined GNSS receiver from free software components; the acquisition module is the first stage in this process.

The acquisition module supports variable length of non-coherent integration to improve signal-to-noise ratio, variable Doppler frequency search range, and an arbitrary number of satellites can be searched in parallel. Tests show that the module perform very well under normal to strong signal conditions, while it fails under weak signal conditions. This is expected as no techniques to improve the performance under such conditions have been applied.

A number of issues have been discovered during the development of the acquisition module:

- Limitations in the GNU Radio framework.
- Limitations in the appurtenant hardware: *The Universal Software Radio Peripheral*.

Both of these issues are addressed, and recommended modifications and topics for potential future work have are proposed.

Even though the complete OpenGNSS receiver is not yet complete, the results so far indicate that the current approach is a viable approach to create a software-defined GNSS receiver from free software components.

Contents

Abbreviations

ADC	Analog to digital converter
AGC	Automatic gain control
BSD	Berkeley Software Distribution
C/A	Clear/Acquisition
CAM	Coarse acquisition module
CDMA	Code-division multiple access
CIC	Cascaded Integrator Comb
CORDIC	COordinate Rotation DIgital Computer
COTS	Commersial of the shelf
DBS	Direct Broadcast Satellite
EMA	Exponentially weighted moving average
FFT	Fast Fourier Transform
FIFO	First in - First out
FIR	Finite impulse reponse
FPGA	Field Programmable Gate Array
GNSS	Global Navigation Satellite System
GNU	Recursive acronym for “GNU is not Unix”
GPS	Global Positioning System
IIR	Infinite impulse response
INS	Inertial navigation system
LE	Logical elements
LNA	Low-noise amplifier
PLL	Phase lock loop
PPM	Parts per million

PR	Pseudo-random
RF	Radio frequency
SDR	Software Defined Radio
SIMD	Single instruction, multiple data
SV	Space Vehicle or GPS satellite
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VCO	Voltage Controlled Oscillator

Chapter 1

Introduction

Satellite navigation systems have been around since the early 1960s, with TRANSIT as the first satellite navigation system of the world [12, Ch. 10.1]. But it was not until the 1980s and the creation of the NAVSTAR Global Positioning System — or more commonly known as GPS — that the use of satellite navigation became widespread. Satellite navigation systems were not the first radio-navigation systems to be created. Land-based systems has existed since the first world war, but land-based systems only provide partial coverage of the Earth. A satellite system is able to provide global coverage at any time with only a relatively small number of satellites.

The word *navigation* is derived from the Latin word *navigato*, which means voyage. However, satellite navigation systems are no longer exclusively used for traditional navigational purposes such as ship navigation, but also for precise positioning services in the offshore industry, personal navigation in urban areas with localized information about the area you are visiting, fleet management systems for people and goods transport companies as well as many other applications. Satellite navigation systems were originally created to serve military purposes, but the number of civil users has outnumbered military users by many times. Satellite navigation is now a part of our everyday life, and the number of applications grow every year.

Until now the only available system has been the American GPS. The Russian GLONASS system has also existed almost as long as GPS, but because of funding issues and a limited numbers of satellites, the system has not been able to provide complete coverage of the earth. The situation has changed in recent years, and the Russians are continuously deploying new satellites. With the increasing dependency of satellite navigation systems, the European Union (EU) decided during the early 2000s to create an alternative system that would be under civil control. This became known as the *Galileo positioning system*, and is scheduled to be operational at the end of this decade.

In the years to come, users will have several systems to choose from. As each of the satellite navigation systems evolve, they get increasingly more advanced. This present new possibilities to both researchers and users. But satellite navigation receiver technology is a very closed technology; manufacturers are reluctant to expose the inner workings of their systems, and commercially available receivers rarely provide access to the raw GNSS data. This makes it difficult to use conventional receiver for advanced concepts such as inertial navigation assisted systems, multi-system receivers and for developing more advanced algorithms. Receivers that are usable for such applications are both expensive and under restrictive licenses, which makes them inaccessible to many researchers, developers and for educational purposes.

In this paper a GPS acquisition module for the OpenGNSS software receiver is discussed. The OpenGNSS software receiver is a software GNSS receiver design described by Danielsen [10]. The purpose of the OpenGNSS software receiver is to create an expandable, flexible and modular platform for creating GNSS receivers for both research, development and education. It is built on top of the GNU Radio software-defined radio framework. The receiver is not yet complete, but the acquisition module is the first step towards creating a complete system. The present task can be divided into two distinct parts:

- Implementation, testing and verification of the functionality and performance of the GPS acquisition module.
- Evaluation of the existing GNU Radio framework, and implementation of required changes to make is more suitable for satellite navigation applications.

The acquisition module is the first stage after the analog processing in a GPS receiver. Initially there are several unknown parameters in the GPS signal which the receiver must estimate be for reception can begin. This is known as the acquisition procedure and involved searching for visible satellites and estimating the unknown parameters so that the receiver can start receiving data from the satellite. This is described in section 2.1 on page 5.

OpenGNSS is a receiver design entirely based on free software components. The term *free* is an ambiguous term in English. It is therefore important to emphasize that *free* here should be read as *freedom*, and not *gratis*. Many people think of free software as software that is available to the user at no cost, but in this context it is the freedom that is important.

The requirement for freedom applies both to the hardware and the software. This includes, but are not limited to, the following demands:

- One must be able to use the hardware and the software for whatever purpose.
- One must be able to study the hardware and the software, and also make changes to it if that is necessary to make it adaptable to ones needs. Access to sufficient documentation, specification and source code is a precondition for this.
- One must be able to share ones work with others. This includes both the original work, and any modifications that has been made. If any non-disclosure agreement is required to get access to documentation or specification, the result can not be considered *free*.

All though the intention is not to limit the OpenGNSS receiver to a particular type of satellite navigation system, the American Global Positioning System (GPS) is the only one that is fully operational and with widespread coverage. The rest of this document therefore focuses on the requirements of the Global Positioning System.

1.1 Previous work

Software-defined satellite navigation receivers are not a new idea, and several commercial receivers and related research projects exist. Real-time receiver have been implemented both on digital signal processors [17] and for personal computers [9], but both of these are implemented entirely in low level languages such as C, C++ and assembly. Many people have also written off-line receivers which process data in Matlab®, but such a receiver cannot operate in real-time, making its usefulness fairly limited.

The OpenGNSS design combines the low-level performance of C++, with the high-level flexibility of Python, creating an easily modifiable system that also include real-time performance. In addition to that, no other current receivers also includes a fully open source hardware and software stack, making it an interesting choice both for researchers and commercial users.

1.2 Software defined receivers

Traditionally satellite navigation receivers implemented the majority of the signal processing in hardware. But because of the growth in computational power in COTS computer equipment, many of these signal processing tasks can now be performed in an ordinary computer. This makes it possible to move much of the functionality of the receiver from static hardware implementations, to dynamic software. This is what is know as a software-defined radio (SDR).

The OpenGNSS design is a software-defined receiver design, since the majority of the signal processing is performed either in a FPGA or on an ordinary computer. This makes the design very flexible, since the receiver can be reconfigured dynamically, making it an ideal choice for multi-mode receivers, research and development, and for low-volume designs. An example of the advantage of the software-defined receiver compared to a traditional receiver is presented in [10, Ch. 1.2].

1.3 GNU Radio

GNU Radio is a framework for creating software-defined radios. GNU Radio has a large and active community of developers, and has been used to create many types of software-defined radios, such as HDTV receivers, passive radars, and experimental GSM receivers. GNU Radio runs under several operating systems such as Linux, Microsoft Windows, Mac OS X and NetBSD, but the primary development platform is Linux.

The GNU Radio project was started by Eric Blossom in 1998 as a fork of the PSpectra code that was developed by the *SpectrumWave* project at MIT. By 2004 all of the PSpectra code was replaced by GNU Radio code, but the design inherited from the PSpectra framework is still visible. The PSpectra SDR framework is described in [7][1].

[1]PSpetra is referred to as SPECTRA in this paper.

1.4 Outline

The rest of this paper contains the following chapters:

First several basic theoretical topics are presented. This includes the principle behind GPS signal acquisition, and important properties of the GPS signal. Then the GNU Radio framework is described in details, and finally a number of important topics regarding software radios is presented.

Furthermore the hardware and the software setup used during the development is described. This includes both the actual software and hardware used in the receiver, and also software and hardware used during the development, testing and verification.

After that the actual implementation of the acquisition module is presented. The various sub-components are described separately, as well as the complete system.

Finally the results of the various tests performed on the OpenGNSS acquisition module is presented and discussed.

Chapter 2

Theory

This chapter covers the basic theory of GPS signal acquisition, important properties of the GPS signal relevant to the acquisition procedure, and fundamental issues regarding software receivers. Basic knowledge of topics such as GPS position calculation and elementary radio and signal processing theory has intentionally been omitted from this paper as it is assumed known to the reader.

2.1 Basic Acquisition

In order to be able to track the GPS signal, an acquisition procedure must be used to detect the presence of the signal. In a regular radio receiver one would usually tune the receiver to the desired frequency and start receiving, but in a satellite navigation system there are a number of factors that must be taken into account:

1. The relative speed between the transmitter and the receiver causes a significant Doppler frequency shift on the GPS signal. The receiver have to calculate this frequency shift before the signal can be brought back to baseband.

2. The GPS signal is multiplied with a pseudo-random (PR) sequence. This done for two major reasons:

   - Provide simultaneous access to the same frequency band for all satellites. This is known as code-division multiple access (CDMA).
   - The PR sequence is used to measure the distance from the receiver to the satellite.

   The receiver has to remove the PR sequence before the data can be demodulated. The problem is that initially the start of the PR sequence is unknown.

The PR sequence is also known as a spreading code. Several different spreading codes are used in the GPS system, but only the *Clear/Acquisition* (C/A) code is available to the general public. The effect of the multiplication with the C/A code is that the signal is spread out over a wider frequency band.

The effect of multiplying a clean carrier at 1 MHz with the spreading code can be seen in figure 2.1.

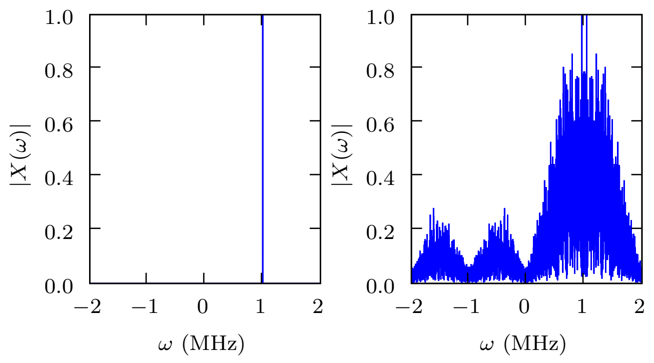


Figure 2.1: C/A code spectral properties.

The procedure for estimating the C/A code delay and Doppler frequency shift is described in [10], but the principle is to correlate the received signal with a locally generated signal that consists of the Doppler frequency and the C/A code of the desired satellite:

$$R = R(f_d, \delta) = (x \circledast \hat{x})(f_d, \delta) = \sum_{\delta=0}^{N-1} x[\delta]\hat{x}^*[n+\delta] \tag{2.1}$$

where $x = x[n]$ is the received signal, and $\hat{x} = \hat{x}(f_d, \delta)$ is the locally generated signal that is a function of both the Doppler frequency and the C/A code delay.[1]

The solution of R is therefore a surface, and the maximum value of $|R|$ corresponds to the Doppler frequency and C/A code delay for the received signal.

Two parameters determine the resolution of the estimate obtained from equation 2.1: The sampling frequency and the length of the sequence used in the calculation[24, Ch.7.4]. Given that one ms of the signal sampled at 5 MHz, the frequency resolution is 1 kHz. With the same sampling frequency, the time resolution will be ± 100 ns[24, Ch.8.10].

2.1.1 Frequency-domain correlation

In order to estimate the Doppler frequency and C/A code shift, the received signal is correlated with the locally generated code. The time-domain correlation is a computationally intensive operation, and is usually implemented in hardware. But the number of operations can be significantly reduced by performing the correlation in the frequency domain. This is known as frequency-domain correlation or fast correlation.

[1]Formally, the operation described in equation 2.1 is circular correlation, and not correlation. Mallat [18, Ch. 3.3.1] gives a deviation of circular convolution, which is very similar to the deviation of circular correlation. Strictly speaking, correlation is defined from $-\infty$ to ∞, but here only finite signals is used in the calculation.

Mallat [18] shows that if f and h have period N, then the discrete Fourier transform of the circular convolution of f and h, $g = f \circledast h$, is:

$$G[\omega] = F[\omega]H[\omega] \tag{2.2}$$

Since the only difference between correlation and convolution is that one of the signals have been reversed, it follows from the same deviation that circular correlation also can be calculated in the frequency domain. If $g = f \otimes h$, then:

$$G[\omega] = F[\omega]H[\omega]^* \tag{2.3}$$

since $\mathcal{F}\{h[-n]\} = H[\omega]^*$ [11, P.528]. The time-domain signal can now be obtained from the inverse Fourier transform of G. Since the Fourier transform is a linear operator, the time-domain representation of the signal can always be reconstruction from the frequency-domain coefficients. Therefore the results obtained from the time-domain and frequency-domain correlation are equal.

From this it is clear that the cross-correlation between the received signal and the locally generated code can be calculated using fast correlation in the frequency domain. While time-domain correlation requires $N(N+1)$ multiplications and additions, the frequency-domain correlation requires $3N\log_2(N) + 11N$ multiplications and additions. For signals where $N >= 32$ fast-correlation is faster than time-domain correlation[18, Ch. 3.3.4].

2.2 C/A code properties

As mentioned earlier, there are several reasons for using CDMA in GPS:

- Multiple access.
- Distance measurement.
- Makes the signal robust against interference and jamming.

The use of the C/A code for distance measurement is not discussed in this paper, as it is not directly relevant to the acquisition procedure, and neither is the jamming aspect. The multiple access features and the robustness against interference however important to the acquisition procedure, and is more throughly discussed.

The C/A code is a member of the family of pseudo noise sequences known as *Gold codes*. Important properties are given in table 2.1.

Property	**Value**
Chip rate	1.023 MHz
Code length	1023 chips or 1 ms

Table 2.1: C/A code properties

CDMA is an access method that permits access to the channel through the assignment of a unique spreading code[14, Ch.7.8]. This means that there at any given time is N interfering satellites visible to the user. The amount of interference between the individual satellites is dependent on the properties of the spreading code.

In order to be able to separate the different satellites from each other, the cross-correlation between the different spreading-codes codes should be zero everywhere:

$$R_{ij}(\tau) = \int_{-\infty}^{\infty} g_i(t) g_j(t+\tau) dt = 0 \tag{2.4}$$

for all τ where $i \neq j$. $g_i(t)$ is the C/A code from satellite i.

In reality this is generally not possible, and it is one usually restricts the demand to say that the cross-correlation should be less than ρ for all τ:

$$|R_{ij}| < \rho \tag{2.5}$$

where $0 < \rho \ll |R_{ii}(0)|$ and $i \neq j$.

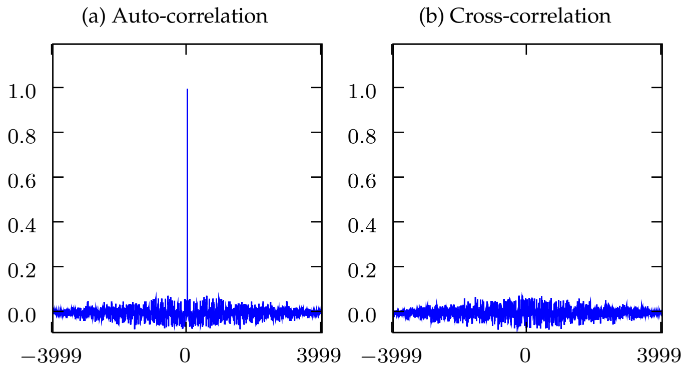


Figure 2.2: C/A code correlation properties.

Figure 2.2 shows the cross-correlation between the C/A code from satellite 1 and 2, and the auto-correlation of the C/A code from satellite 1.

Since the received signal is the sum of the signals from each of the satellites, the correlation for a given code is the sum of the auto-correlation function for the desired satellite, plus the cross-correlation from interfering satellites. If the number of visible satellites be N, then resulting correlation function would be:

$$R_{ii} = R_{ii} + \sum_{j=1}^{N-1} \alpha_j R_{ij}(f_{dj}, \tau_j) \tag{2.6}$$

Note that the both the C/A code delay, Doppler frequency shift and received power is different for each of the satellites. Spilker [21, p.105] shows that the worst-case peak value of the cross-correlation side-lobe is 21.6 dB below the auto-correlation peak of the desired signal. However, due to the time-varying Doppler frequency, the cross-correlation peaks are not stationary compared to the auto-correlation peak. Also, the probability for that all of the interfering satellites with produce cross-correlation peaks with the same delay is very small. As an approximation the undesired cross-correlation terms can therefore be expressed as additive white noise 30 dB below the auto-correlation peak of the desired signal.

2.3 Multipath effects

Multipath is the phenomenon whereby a signal arrives at a receiver via multiple paths attributable to reflection and diffraction[8, p.547]. Multipath effects is a dominant source of errors in navigation systems. Reflections distorts the signal modulation, which broadens the correlation peak of the C/A code and thereby reduced the accuracy of the pseudo-range measurement and decreases the signal to noise ratio and makes acquisition more difficult. It also distorts the carrier phase, and hence degrades the accuracy for interferometric use.

For a stationary receiver the effect can be significantly reduced by antenna design, but for mobile applications, in urban areas and in highly dynamic applications such as aircrafts, the effect can not reduced by the effect of the antenna pattern.

It is important to notice that the errors introduced by reflections generally have non-zero mean values[8, p.548], which means that the error can not be remove by averaging the signal over a given period of time.

The direct signal received from a single satellite can be described by the following equation:

$$s_d(t) = \frac{\sqrt{A}}{2} C_i sin(2\pi(f_c + f_d)t + \theta_0) \tag{2.7}$$

where A is the power of the received signal, C_i is the C/A code of SV #1, f_d is the initially unknown Doppler frequency shift, and θ_0 is the initial carrier phase.

If a single-ray reflection is introduced, the equations can be rewritten as follows:

$$s_m = s_d(t) + \alpha s_d(t + \delta) \tag{2.8}$$

where α is the attenuation factor for the reflected signal and δ is the delay of the reflected signal introduced by the longer path compared to the direct signal.

Simulations for a single-ray reflection that is attenuated 20 dB compared to the direct ray show that the peak pseudo-range error is 15 m[8, p. 554]. The worst case reflection scenario is where the reflected ray is stronger than the direct ray ($\alpha > 1$). This can occur in urban areas and indoor, but this special case is beyond the scoop of this paper.

2.4 Effect of sampling frequency inaccuracy

Most discussions regarding digital signals assume that the sampling frequency is exact, but if an offset in the sampling frequency is present, a number of errors are introduced. Tsui [24, Ch. 6.15] discusses the effect this has on the center frequency of the down-converted signal. This error is however so small that it is insignificant compared to the Doppler shift on the signal. The most important effect is on the pseudo range measurement, and is related to the drift of the correlation peak caused by the offset.

If an offset between the assumed and the actual sampling frequency is present, the estimated C/A code delay would change at a rate that is proportional to the sampling frequency offset Δf_s and the length of the C/A code, which is 1 ms.

If the received signal is sampled at 4 MHz, the locally generated C/A code also has to be digitized at this frequency, which means that the C/A code is 4000 samples long. But if there is an error Δf_s in the sampling frequency, the locally generated code would either be longer or shorter than the received code. This mismatch makes the cross-correlation peak drift with a rate that is given by:

$$\frac{d}{dm}\delta_{C/A}(m) = \frac{d}{dm}\left[\underset{n\in[0,N)}{\operatorname{argmax}}\{R(n)\}\right] = t_c\Delta f_s \tag{2.9}$$

where t_c is the length of the spreading code, $\delta_{C/A}(m)$ is the C/A code delay, and $R(n)$ is the cross-correlation function for a fixed f_d. Notice that the rate of $\delta_{C/A}$ is different from that of $R(n)$. The period of $\delta_{C/A}(m)$ is always equal that of the C/A code period, 1 ms, while the rate of $R(n)$ depends on the sampling frequency. The code tracking loop in the receiver must be able to follow this change in code delay.

In figure 2.3 the effect can be seen for a simulation with a sampling clock offset of -1.57 kHz. This corresponds to a different in length between the two codes of 2 samples.

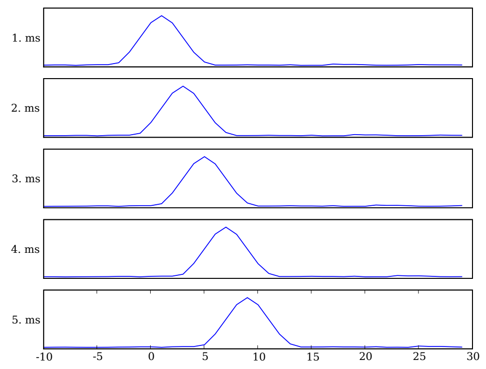


Figure 2.3: Effect of sampling frequency offset

In addition to the shifting the correlation peak is also broaden because the generated and received code no longer match perfectly. According to Tsui [24, Ch. 7.3], if the C/A code is off by half a chip, the correlation peak is reduced by 6 dB. Simulations show that an offset in sampling frequency of -1.57 kHz, attenuates the correlation peak by 0.6 dB. The reduces the signal to noise ratio, but the loss is small enough to be neglected.

2.5 Effect of FFT window on estimate accuracy.

When using frequency domain correlators for calculating the cross-correlation, the choice of Fast Fourier Transform (FFT) window function is important with regard to the accuracy of the estimate obtained from the acquisition module. A window function is any given finite sequence $\{z_n\}$ of length N equal to the length of the FFT operation that is multiplied with the signal before the FFT is performed. The most obvious window is the rectangular window, where $z_i = 1$ for $i \in [0, N)$, which is implicitly applied since sequences of finite length is used. Figure 2.4 shows the effect of multiplication with a Hann(ing) window.

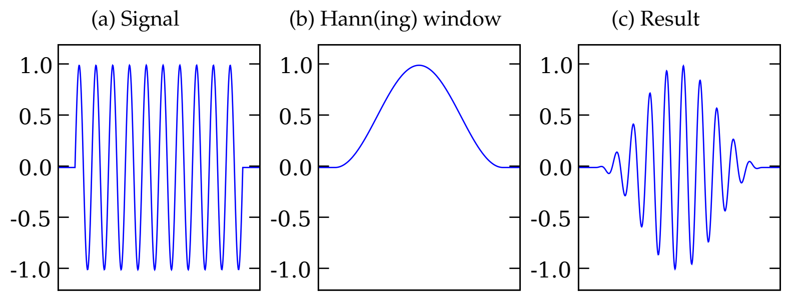


Figure 2.4: Signal before and after multiplication with a Hann(ing) window.

There are two important factors that must be considered when selecting the window function for the FFT:

- The effect on the C/A code correlation peak.
- The spectral leakage to adjacent frequency bins.

Both these factors are discussed in the following paragraphs.

Correlation peak loss The effect of the window function on the correlation peak magnitude has been simulated, and table 2.2 show the correlation peak loss relative to the ideal correlation peak for several well-known window functions. Both the received signal and the local code was multiplied with the window function before the cross-correlation was calculated.

Window function:	**Loss [dB]**
Flattop	-7
Blackman-Harris	-5
Blackman	-4
Hann	-4
Hamming	-4
Triangular	-4

Table 2.2: Correlation peak loss.

Spectral leakage A signal consisting of a single frequency corresponds to a Dirac in the frequency domain, but for this relation to be valid, the signal must have infinite duration. The effect of using finite length sequences for computing the frequency content of a signal is that the Dirac is convoluted with the Fourier transform of the window function, which leads to spectral leakage to adjacent frequency bands. When estimating the Doppler frequency of the received GPS signal, this means that the original carrier frequency might be smeared into neighbouring bands. If enough energy is leaked into these sidebands, the magnitude of the correlation peak in the sidebands might be almost equal to the correct one, and it would not be possible to determine which peak is the correct one.

The most important properties of the window function is the width of the main lobe, and the attenuation of the side lobes. Figure 2.5 on the next page shows the Fourier transform of a single frequency signal windowed by three different window functions: Rectangular window, Hann(ing) window and Blackman-Harris window (from left to right).

It is obvious from the figure that the level of the side lobes are much lower for both the Hann(ing) and the Blackman-Harris window. However what is not visible from the figure, is the width of the main lobe. If the main lobe is very wide, it will stretch into adjacent bins, and have large impact on the acquisition procedure.

Window function	**3 dB bandwidth (bins)**	**Highest side-lobe level (dB)**
Rectangular	0.89	-13
Hann(ing)	1.44	-32
Blackman-Harris	1.9	-58

Table 2.3: Properties of selected window functions[13]

Table 2.3 shows the 3dB bandwidth of three selected window functions.

From this it is clear that the relation between the side lobe level and the width of the main lobe are contradicting terms: One can not get a narrow main lobe while maintaining low side lobe levels.

2.6 Moving average filter

Because of the low signal to noise ratio of the GPS signal, it is usually not possible to estimate the Doppler frequency and C/A code delay from a single C/A code period[24, Ch. 7.11]. The correlator outputs are therefore non-coherently integrated over several C/A code periods to produce acceptable estimates. There are several ways to perform this non-coherent integration; here two different *moving average filters* are described.

A moving average filter is used to reduce random noise while retaining a sharp step response. This section describes the difference between the exponentially weighted moving average (EMA) filter and the simple moving average (SMA) filter[20, Ch. 15]. The EMA filter is also known as a *single pole recursive filter*[20, Ch. 19].

The EMA filter can be described by the following equation:

$$y[n] = (1 - \alpha)y[n-1] + \alpha x[n] \tag{2.10}$$

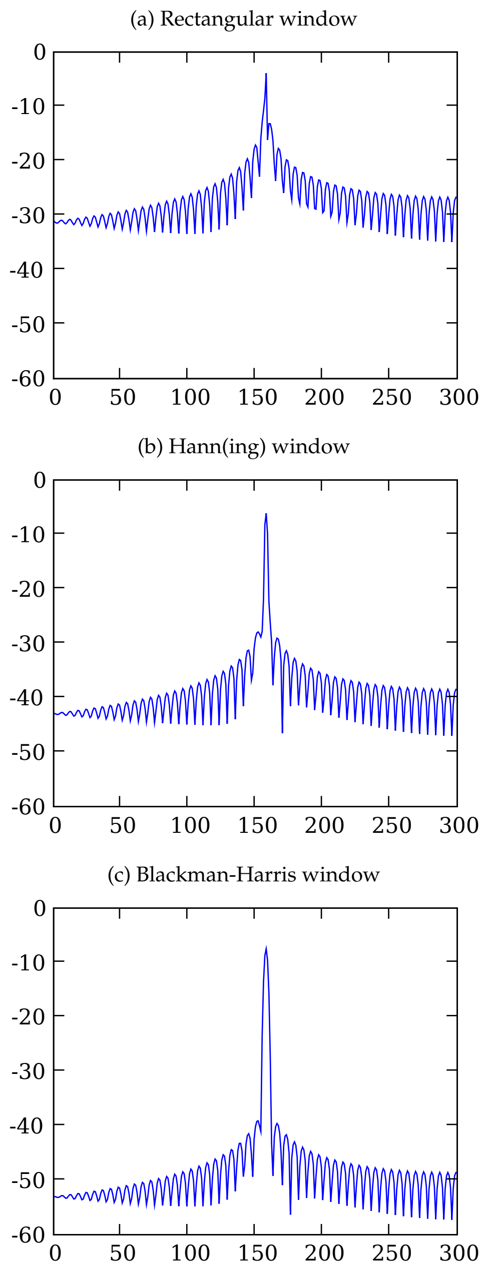


Figure 2.5: Fourier transform of window functions: Rectangular, Hann(ing) and Blackman-Harris (left to right)

where $x(n)$ is the input signal, and $y(n)$ is the output.

The SMA filter can be described by the following difference equation:

$$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i] \tag{2.11}$$

The EMA and the SMA filters are similar in the way that they both output a weighted sum of N previous input items. This is different is that in a simple moving average filter N previous values are weighted equally, whereas in a EMA filter higher confidence is placed in recent observations.

From this it is obvious that to compute a single output from the SMA filter N previous observations are required, while the EMA filter only depends on the previous output and the current input. The memory requirement of the EMA filter is therefore independent of the filter coefficients, while the requirement for the SMA increases linearly with the length of the filter.

The difference in memory requirement becomes significant when operating on data sets of the size of the correlation matrix found in the coarse acquisition procedure.

2.6 shows the step response of the EMA filter for $\alpha = 0.05$. As can be seen from the figure, the step response reaches 70 % of the maximum value after 20 samples, which is approximately $\frac{1}{\alpha}$, and 100 % after 100 samples ($\frac{5}{\alpha}$). This relationship between α and the step response can be used to find the desired filter coefficient.

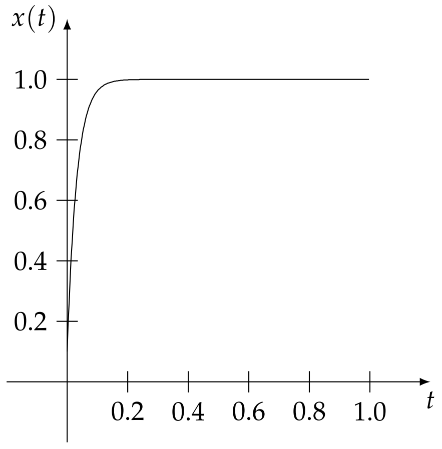


Figure 2.6: Exponential moving average filter step response.

2.7 The GNU Radio software radio framework

This section gives a brief introduction to the GNU Radio software radio framework. The purpose of this section is to illuminate the possibilities and limitations of the system.

GNU Radio uses a block-based architecture that utilizes a hybrid Python/C++ programming model. This enables programmers to write computationally intensive tasks in C++ to get high performance, while retaining the convenience of the dynamic type system, automatic memory management and interactivity of the high-level language Python. Python was created by Guido van Rossum in 1991 and was designed to be a multi-paradigm interpreted programming language[25, Ch. C1]. Python is a very popular programming language due to its versatility and clean syntax. It supports a number of programming paradigms, such as object-oriented programming, functional programming and imperative programming.

The principle behind the framework is to write elementary signal processing blocks such as FIR filters and mixers in C++, and then connect these together to create a chain of processing blocks. This is named a *flow graph* in GNU Radio terminology. The term is borrowed from graph theory. The advantage of this design is that the topology of the signal processing chain is defined in Python scripts, and the topology can therefore be changed without recompiling the program. The inner details of the GNU Radio framework is described more elaborately in 2.7 on the facing page.

GNU Radio is accompanied by a custom made hardware component named the *Universal Software Radio Peripheral* (USRP). The USRP is a general purpose software radio device that consists of a motherboard with AD- and DA-converters, a FPGA for high-speed signal processing, and an USB interface, and a variety of daughterboard for a broad range of frequency bands and applications. The USRP is described in section 3.3 on page 24.

The GNU Radio framework was evaluated and compared to the alternatives in [10], and found to be the most promising free framework for building software radios, and was therefore chosen as the foundation for further work. Other alternatives exist, but non provided the necessary flexibility and freedom [10, Ch. 3].

GNU Radio can be described as to separate parts:

Signal processing blocks: Elementary signal processing functions such as automatic gain control (AGC), phase lock loops (PLL), and simple multipliers are where the actual signal processing takes place.

Run-time support system: Memory buffers for communication between blocks, and scheduling of the data flow in the entire system.

Each of these parts are described separately in the following sections.

2.7.1 Signal processing blocks

All GNU Radio programs are made up of elementary signal processing block that are connected together in a `flow_graph`. This is where the actual signal processing in a GNU Radio application takes place. Signal processing blocks are either written as individual signal processing blocks in C++, or they can be aggregated into hierarchical block in Python to create super-blocks consisting of several smaller blocks.

All signal processing blocks have well defined interfaces that consists of the number of input and output streams, the data type of the input and output streams, and the number of items per stream. The properties are derived from the top level class `gr_block`, which all blocks inherit from. The `gr_block` class has a number of properties and methods, but the most important ones are listed below:

- `virtual int general_work():`
  This function is a virtual function that must be overridden by every block, and is the function that is called when a block is activated by the run-time support system. This is therefore the key function that defines the actual functionality of the block.
- `gr_io_signature {input, output}_signature():`
  The IO signatures defines the input and the output interface of the signal processing block. The `gr_io_signature` data structure contains the number of minimum and maximum number of connections, and the data type of a given port.
- `void set_history():`
  The history of a block is the number for previous items are required to generate a single output item. An example of this is a FIR filter of length N. To produce one output item $y[n]$, the previous values of x from $x[n]$ to $x[n-M-1]$ are required.
- `void set_relative_rate():`
  The relative rate of a block is the ratio between the number of inputs compared to the number of outputs.

By using the `set_relative_rate()` method, it is possible to create variable rate blocks, but most common blocks fall into one of the following subclasses:

Synchronous blocks: 1:1 relation between the number of input and output items.

Decimating blocks: $N > 1$ inputs items is required to produce a single output item. This effectively reduces the rate of the signal.

Interpolating blocks: $M > 1$ output items are produced per input item. This effectively increases the rate of the signal.

For these three common cases, predefined subclasses of the `gr_block` has been created: `gr_synch_block`, `gr_sync_decimator`, and `gr_sync_interpolator`. These classed simplifies the development of blocks that fits these standard cases.

2.7.2 Run-time support system

The individual signal processing blocks are separate, functional units that accepts data on one end and returns the processed data on the other. Each elementary block has no knowledge of the surrounding environment, and therefore a run-time support system is needed to connect blocks together and control the data flow through the entire system.

The run-time support system provides two basic functionalities:

FIFO buffers: These buffers are used to connect the individual blocks together.

Scheduling: Each block is activated when there is sufficient data available at the input, and sufficient space available at the output.

The FIFO buffers used to connect individual blocks together are allocated during the initialization of the GNU Radio flow graph. The run-time support system traverses the entire graph, allocates buffer depending on the size of the data and the length history of the blocks, and ensures that the types of the input and the output matches. The GNU Radio framework is a strongly typed system, which means that no implicit conversion of data types is performed. This means that input and output of different data types cannot be connected, and attempting to do so will result in an exception from the run-time system.

Once the initialization is complete, control is passed to the scheduler. The scheduler in GNU Radio is a very simple scheduler that sequentially polls each block to see if there is enough data available in the input buffer, and sufficient free storage space available in the output buffer. If these criteria are met, the `general_work()` method is invoked.

2.8 Digital down-conversion

The process of down-conversion, or more generally frequency conversion, is to move a signal from one frequency to another. Ideally one would just connect the ADC to the antenna or a low-noise amplifier (LNA) and sample the GPS signal directly at 1575.42 GHz, but with the current performance of analog-to-digital technology, it is easier to build a down-converted design. The signal is therefore translated from RF using analog components to a lower IF where it can be digitized.

Once the signal is at IF there are several alternatives:

Direct conversion. Also known as Zero-IF. The signal is brought directly from RF down to baseband, and can therefore be sampled directly after the analog down-conversion.

Additional analog mixing stage(s). By mixing down the signal over several stages, the specifications (resolution, dynamic range) of each stage is relaxed compared to the direct approach.

Non-zero IF digitization. The signal can be brought down to an intermediate frequency and digitized there, and then the rest of the down conversion can be performed digitally. This is the approach that will be discussed further.

By doing the down-conversion from IF in the digital domain, one retains the relaxed requirements on the initial RF stage, while maintaining the flexibility of the digital signal processing.

The following sections describes three essential algorithms used in the digital down-converter on the USRP: The CORDIC algorithm for generation the complex carrier used for the actual down-conversion, and two filters used to decimate the signal to a lower sampling rate, cascaded integrator-comb and half-band filters.

2.8.1 CORDIC

CORDIC is an algorithm for calculating trigonometric functions and vector rotation using only simple operations such as shift and add, making it suitable for hardware implementation. The CORDIC algorithm is used in the USRP to generate the carrier for mixing down the received signal. There are several ways to generate the carrier frequency in the FPGA. One way would be to store the complex carrier in a lookup table, and just iterate over it as desired. Although simple, the memory requirement for such an approach is much too large for the FPGA on the USRP. It is possible to avoid the large lookup table by using generating the sine and cosine samples with appropriate IIR filters[16, p. 162], but this still requires four multipliers that are not available in the FPGA on the USRP.

The CORDIC algorithm can be operated in two modes: Rotation and vectoring. A complete description of both modes is given by Vankka in [26, Ch. 6], but a brief introduction to the rotation mode is given here.

The complex carrier $e^{2j\pi f_c n/N}$ can be described as a two-dimensional vector $\vec{x} = [x, y]$ that is rotated by an angle θ per iteration:

$$\vec{x}_{n+1} = A\vec{x}_n = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \vec{x}_n \tag{2.12}$$

where A is the transformation matrix.

The transformation matrix A can be rearranged as follows:

$$A = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} = \cos(\theta) \begin{bmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{bmatrix} \tag{2.13}$$

If the rotation angle θ is restricted to $\tan(\theta) = \pm 2^{-i} = d_i 2^{-i}$, the multiplication by the tangent term is reduced to a simple shift operation. In order to produce arbitrary angles, a sequence of successively smaller elementary rotations as shown in figure 2.7 are performed. d_i determines the direction of rotation, and the angle of the composite rotation is uniquely defined by the sequence of the directions of the elementary rotations.

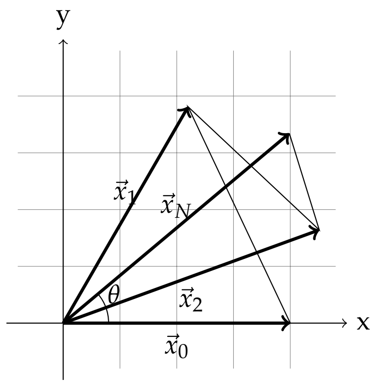


Figure 2.7: Vector rotation using CORDIC algorithm

Furthermore this restriction makes it possible to simplify the cosine term to:

$$\cos(\theta) = \cos(\arctan(2^{-i})) = K_i \tag{2.14}$$

The vector $\vec{K}$ is a constant vector which is independent of the rotational angle, and can thereby be computed in advanced and stored in ROM. This final modification allows us to write the transformation matrix on the following form:

$$A = K_i \begin{bmatrix} 1 & d_i 2^{-i} \\ -d_i 2^{-i} & 1 \end{bmatrix} \tag{2.15}$$

If the scaling term K_i is ignored, the equation 2.12 on the preceding page can be computed using only shifts and adds.

In rotation mode an additional equation is added to the CORDIC algorithm:

$$z_{i+1} = z_i - d_i \arctan(2^{-i}) \tag{2.16}$$

where

$$d_i = \begin{cases} 1 & \text{for } z >= 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.17}$$

$\vec{x}_0$ is the initial phase of the rotation vector, and z_0 is the desired rotation angle. When z_i approaches zero, the desired rotation is obtained. In order to generate a complex carrier, the rotation angle is chosen according to the carrier and the sampling frequency, and the $\vec{x}_n$ is used as the initial phase to calculate $\vec{x}_{n+1}$.

It can be shown that the product of the K's are [26, p.82]:

$$\lim_{N\to\infty} \prod_{i=0}^{N} K_i \approx 0.6073 \tag{2.18}$$

This means that the non-scaled CORDIC algorithm has a gain $G_N = \frac{1}{0.6073} = 1.647$. The effect of this scaling is not discussed here as it is not relevant for understanding the principle behind the CORDIC algorithm.

Errors introduced by the CORDIC algorithm

When the CORDIC algorithm is used to calculate trigonometric functions, a number of errors are introduced. This is partly because fixed-point numbers used for calculation, and because the number of iterations are always finite. The effect of using fixed-point numbers for calculation is discussed by Antelo et al. [5]. An error is introduced when the desired rotational angle gets smaller than the smallest number that can be represented by the z_i register. This will eventually happen when then number of iterations exceed a certain value.

The effect of using a finite number of iterations, is that there will be an error between the desired rotation and the actual rotation, i.e. $z_i \neq 0$. The largest error will occur if the vector after the second last iteration lands on the correct angle. The last rotation will then move the vector $\frac{\tan^{-1}(2^{N-1}360}{2\pi}$ away from the correct angle. For a 12 stage CORDIC, this error will then be:

$$e = \frac{\tan^{-1}(2^{11}360}{2\pi} = 0.03 \text{ Hz} \tag{2.19}$$

Frequency resolution

Since the largest angular rotation that can be performed by the CORDIC algorithm is $\pm\frac{\pi}{2}$, it takes for cycles to rotate the vector all the way round. The frequency resolution is given by the size of the phase register. This register is updated by a certain amount between each step to generate the complex carrier,

2.8.2 Cascaded integrator-comb filter

The cascaded integrator-comb (CIC) filter is an efficient way to perform interpolation and decimation without multipliers.

A CIC filter consist of two basic building blocks: The integrator and the comb. The integrator, which is also known as a accumulator, is similar to the exponential moving average filter described in section 2.6 on page 12, except that the integrator has a unity feedback coefficient:

$$y[n] = y[n-1] + x[n] \tag{2.20}$$

The power response of this is essentially a low-pass filter with -20 dB per decade roll-off, but with infinite DC gain. This means that the integrator by itself is unstable.

The comb is a odd-symmetric FIR filter filter described by:

$$y[n] = x[n] - x[n-RM] \tag{2.21}$$

Where R is the rate change parameter and M is the differential delay, usually 1 or 2[26, p.232]. For a decimating CIC, the integrators are connected before the change of sampling rate, and the comps are connected after the rate change as seen in figure 2.8.

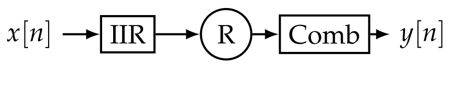


Figure 2.8: Cascaded integrator-comp filter

The power response of the filter at the output can be shown to be:

$$H(f) = \left[\frac{\sin(\pi M f)}{\sin(\pi f/R)}\right]^{2N} \tag{2.22}$$

where N is the order of the filter, namely the number of integrators and combs. Figure 2.8 shows a first order filter. From equation 2.22 it is evident that the frequency response of the CIC filter is fully described by the three parameters R, M and N.

2.8.3 Half-band filter

A half-band filter[6, Ch. 6.9.4] is a FIR filter where every alternate coefficient is zero, except for the center coefficient which is 0.5. To realize such a filter, the filter must be centered around $f_s/4$, and have an odd number of coefficients.

The advantage of the half-band filter over an arbitrary filter design is that the zero-valued coefficients reduces the number of multiplications and memory requirement by 50 %.

Half-band filters are usually used right after CIC filters in the digital down-converter because the CIC filter does not have sufficient stop band attenuation[26, Ch. 11.9].

Chapter 3

Hardware description

In this chapter the various hardware components used during development and testing of the OpenGNSS acquisition module is described.

3.1 Spirent satellite navigation simulator

A Spirent satellite navigation simulator had been used during the development of the OpenGNSS acquisition module. The simulator is a comprehensive facility for developing GNSS equipment. Both ionospheric, tropospheric, and multipath effects can be simulated. It is also possible to simulate a dynamic receiver platform. The simulator consist of two distinct parts:

- A multi-channel RF signal generator.
- A computer workstation for controlling simulation scenarios.

The simulator was used to generate GPS signals during development and evaluation. Initially the plan was to use real signals from an active antenna, but due to the lack of control of the signal properties, this was reject early in the process. The Spirent simulator can be operated in several modes, providing both single channel signals as well as complete scenarios with full satellite constellations.

In the single channel mode the simulator generate the signal from a single satellite. All parameters — such as Doppler frequency and signal strength — can be modified interactively at runtime.

In normal mode, the operator creates a scenario where parameters such as number of satellites, multipath properties and receiver dynamics are programmed in advance, and then the scenario is executed. In normal mode it is not possible to adjust parameters interactively at runtime.

3.2 Low noise amplifier

A ZHL-1217MLN low noise amplifier from Mini-Circuits was used between the Spirent simulator and the USRP. The LNA add an additional 30 dB gain and has a noise figure of 1.5 dB (max)[19].

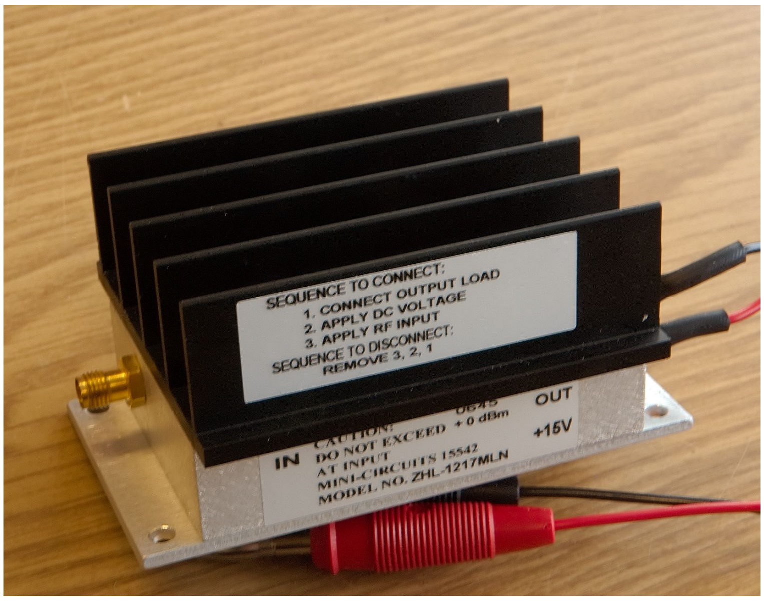


Figure 3.1: Low-noise amplifier

3.3 Universal Software Radio Peripheral

The USPR is a key component in the system, and its performance is crucial to the overall performance. The USRP is a modular system that consist of a motherboard with AD- and DA-converts, a FGPA for high-speed signal processing, and a USB controller for interfacing with a host computer. A variety of daughterboards are available which covers a large range of radio frequency bands and applications. The USRP motherboard can be seen in figure 3.2 on the facing page, where the four daughterboard connectors are visible in each of the corners.

The USRP has be especially made for use with the GNU Radio framework, and is under continuous development. Both the motherboard and daughterboards are available from `http://www.ettus.com`, but since the USRP is being developed under the same conditions as GNU Radio, both the schematics and Verilog source code[1] for the FPGA is available from the GNU Radio website.

3.3.1 RF front-end

The RF front-end used is the DBSRX daughterboard from `http://www.ettus.com`. The DBSRX is based on the Maxim MAX2118 tuner IC[4], which is a direct conversion tuner originally intended for digital direct broadcast satellite(DSB) television applications. It has integrated VCOs ranging from 925 to 2175 MHz, and tunable LP filters from 4 to 33 MHz, which makes it usable for receiving GPS L1 signals.

[1]see 3.3.4 on page 26

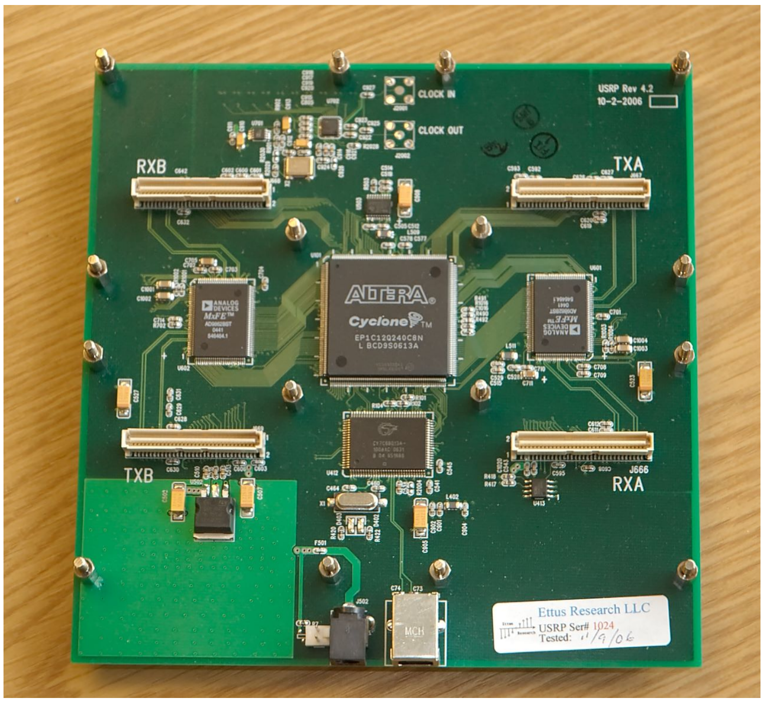


Figure 3.2: Universal Software Radio Peripheral

The MAX2118 can perform direct conversion of the signal from RF to complex baseband, but here non-zero IF digitization described in section 2.8 on page 17 is used, so the MAX2118 carrier frequency is set below the carrier frequency of the GPS signal.

Figure 3.3 on the next page shows the DBSRX daughterboard. The MAX2118 IC is visible just to the right of the coaxial connector.

3.3.2 Analog to digital converter

The analog to digital (ADC) converters are Analog Devices AD9862 [1], which are mixed-signal front end processors. The AD9862 consists of both a receiver and transmitter path, but only the receiver path is described here.

The AD9862 has differential input, programmable gain before sampling, and optional digital Hilbert filters. The Hilbert filter can be used to generate complex signals from real input, but this is not necessary, since the MAX2118 outputs both I and Q channels. Each of the I and Q channel are sampled at 12 bit resolution.

The USRP has two AD9862; one for each pair of RX and TX. These can be seen on each side of the FPGA in the middle of the picture in figure 3.2.

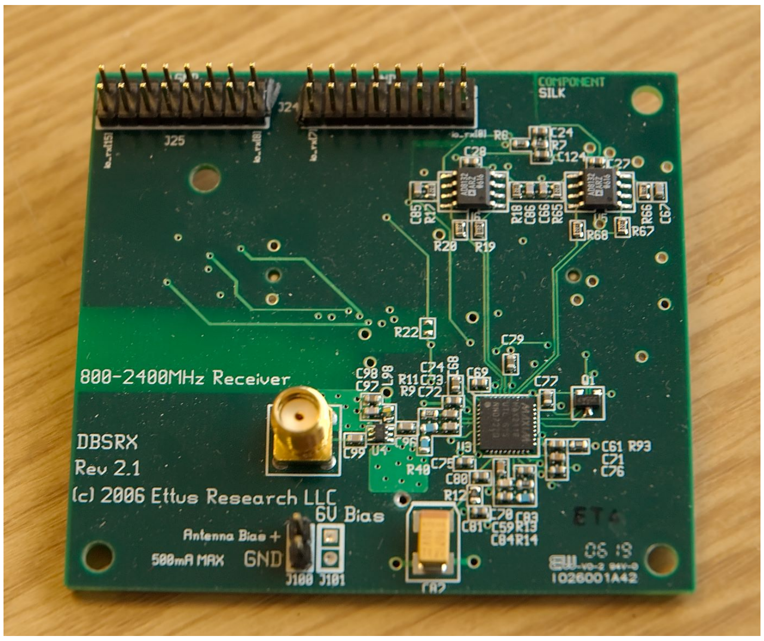


Figure 3.3: DBSRX daughterboard

3.3.3 Oscillator

A 64 MHz crystal oscillator is used to generate the clock signals for everything on the USRP. The oscillator has an tolerance of 50 ppm, which means that is should be within 64 MHz $\pm$ 50 Hz. This clock is used to generate all of the signals used in both the FPGA, the AD/DA-converters and on the daughter-boards.

3.3.4 Field Programmable Gate Array

FPGA is a device containing programmable logic components and inter-connectors. FPGAs are programmed in a hardware description language such as Verilog or VHDL, and the source code is then synthesised to an image that is loaded into the FPGA.

The FPGA on the USRP is a Altera Cyclone I[3], and can be seen as the large integrated circuit in the middle of the board on figure 3.2 on the previous page. Key parameters are given in table 3.1, and the complete reference is given in [3].

Parameter	**Value**
Logical elements(LE)	12060
RAM	Up to 36,864 bytes
Clock frequency	64 MHz

Table 3.1: Altera Cyclone I specifications

The FPGA is used to implement the digital up- and down-converters. Since the Cyclone I does not have any hardware multipliers, the mixing operation cannot be performed directly by multiplying the received signal with a locally generated complex carrier. Therefore the CORDIC algorithm described in section 2.8 on page 17 is used to perform this operation.

The USRP motherboard has room for four daughterboards, which means that four independent RX and TX paths must be available in the FPGA. The standard FPGA image for the USRP has two RX and two TX paths, each with a CORDIC, a CIC filter and a half-band filter. An alternative image is also available which contains 4 RX paths, but no half-band filter.

3.3.5 USB controller

A Universal Serial Bus (USB) is used to interface the USRP with the host computer. The theoretical maximum speed of the USB 2.0 interface is 480 Mbits/sec or 60 MB/sec, but the overhead of the USB protocol is taken into account, the effective throughput is approximately 40 MB/sec. Each channel (I and Q) are sampled at 12 bit, but are either be padded to 16 bit or truncated to 8 bit before transmitting. This makes each sample from the USRP wither 2 or 4 bytes in total. The effective bandwidth over the USB interface of 10 MHz at 12 bit resolution, and 20 MHz at 8 bit resolution.

The USB controller can be seen below the FPGA in the picture in figure 3.2 on page 25.

3.4 Host computer

All of the software was developed and executed on a COTS workstation with an AMD Athlon X2 64 CPU. The GNU Radio framework has be written to take advantage of SIMD instructions available in modern processors such as 3DNow!, SSE and MMX. SIMD instructions are special instructions provided by the CPU that operates on blocks of data instead of single registers, and thereby increasing the performance of the calculation by taking advantage of the pipelining in the CPU. As described by Heckler and Garrison [15], this significantly speeds up the performance of functions such as FIR filters, FFT and other common signal processing tasks.

The complete specification of the computer is given in table 3.2.

Component:	**Specification:**
CPU:	AMD Athlon X2 64 3200+ or faster
RAM:	1 GB or more
Motherboard:	ASUS
USB controller:	Nvidia NForce
Hard drive:	Ordinary SATA drive

Table 3.2: Host computer specifications

Chapter 4

Software description

In this chapter the software used during development and testing is described. This includes development tools, operating system and runtime support systems.

4.1 Operating system

Fedora 6 and 7 was used both for development and as host platform for the OpenGNSS. Fedora is a free operating system based on the Linux kernel and user space software from many different sources. It is sponsored by Red Hat, which is one of the biggest commercial Linux vendors, but people from all over the world contribute to Fedora. Fedora is a completely free operating system that anyone can download, install, use, modify and redistribute as it suits their needs.

Fedora was chosen because of its large user base, up to date selection of software included in the operating system, and the authors existing knowledge of the system. It has proven to be a reliable and stable platform.

4.2 Development tools

All of the required tools such as compiler, assembler, linker, text editor and other required tools to build GNU Radio and OpenGNSS are available as free software and included in the Fedora distribution. The compiler required to compiler the firmware for the USB controller on the USRP was initially not available in Fedora, but was packages and submitted for inclusion, and is now a part of Fedora and maintained by the author of this paper.

Other important things to notice is that the compiler that has been used is GCC 4.1.2 and Python 2.5. GCC is the GNU Compiler Collection and is a collection of compilers for a large range of programming languages such as C, C++, Java and Fortran.

4.3 GNU Radio

The development version of GNU Radio was used for all of the development of the acquisition module. The development version is the unstable version of GNU Radio that is only available directly from the Subversion repository. Subversion is an open-source revision control system that is used to track changes to a code base.

Usually it is desirable to use the stable version of a piece of software, but unfortunately OpenGNSS depends on some modules which are only available from the Subversion repository. These modules are the `gr.max` and `gr.argmax` modules which were developed for OpenGNSS. They have been committed to the GNU Radio tree, and should therefore be available in the next stable release.[1]

For users who which to use the current stable release, it should not be very hard to download the modules from the Subversion repository, and patch the stable release. However, the unstable version of GNU Radio has not yielded any problems during the development of OpenGNSS

[1]The commit message can be found here: `http://lists.gnu.org/archive/html/commit-gnuradio/2007-05/msg00184.html`

Chapter 5

System design and implementation

In this chapter the design and implementation of the acquisition module is presented. The acquisition module is based overall description of the OpenGNSS receiver is given in [10]. Three important aspects are emphasized in the design of the acquisition module:

Modularity and flexibility: The most important aspect of the acquisition module design is the modularity and flexibility of the system. Each part should have a well defined interface; making it possible to replace individual components without breaking other parts of the system.

Parallel acquisition: It should be possible to perform acquisition on an arbitrary number of satellites and variable Doppler range. The design should not limit any of these properties.

Performance: Although performance is not the number one goal for this design, one should strive for high performance where this does not reduce the modularity and flexibility of the module.

The basic principle behind the acquisition procedure is described in 2.1 on page 5, and is essentially to compare or correlate the received signal with a locally generated copy of the signal. The correlation will have a peak when the locally generated signal matches the received signal perfectly. There are a number of ways to speed up the acquisition procedure, and in the OpenGNSS acquisition module two principles are used:

Fast Correlation: Also known as *frequency domain correlation*, fast correlation is a way to speed up the correlation procedure by performing it in the frequency domain as described in section 2.1.1 on page 6. Fast correlation is faster than ordinary time-domain correlation when the length of the sequence used in calculation is larger than 32 2.1.1 on page 6. If the GPS signal is sampled at 4 MHz, the C/A code will be 4000 samples long, so in this case it should be safe to say that fast correlation is indeed faster than the time-domain equivalent.

Parallel Doppler frequency search The acquisition module is capable of searching N Doppler frequencies in parallel, where N is determined by the Doppler frequency search range. The maximum number of Doppler frequencies that can be processed in parallel is only limited by the computational power of the host computer. Since the GNU Radio framework runs each block in its own thread, this comes automatically from the GNU Radio framework.

Figure 5.1 presents a graphical representation of the complete acquisition module which is given in listing A.2 on page 62. The block SCC is the single channel correlator described in section 5.1. It is important to note that the figure only shows three parallel channels, but as mentioned, an arbitrary number of channels can be connected to the FFT output. All of the elementary blocks will be described more elaborately in the following sections.

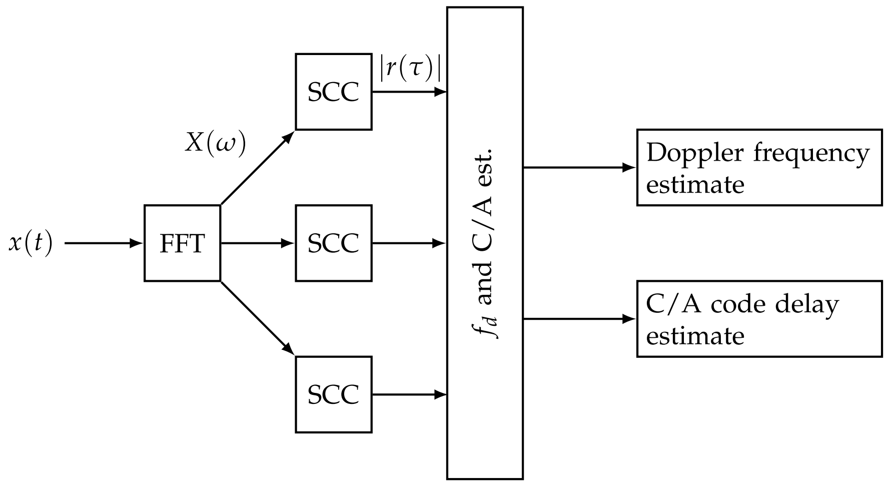


Figure 5.1: Complete acquisition module

5.1 Single channel correlator

The *single channel correlator* (SCC) correlates the received signal with the locally generated code, using the fast correlation procedure described in 2.1.1 on page 6. Input parameters at initialization are Doppler frequency, frequency range and Space Vehicle Number (SVN). These remain constant for the entire lifetime of the acquisition object. A graphical representation of the module is given in figure 5.2 on the next page, and the source code can be found in listing A.3 on page 64.

5.1.1 Exponential moving average filter

The single channel correlator module contains an IIR filter that is used to integrate the signal. The filter is an exponential moving average filter, and its characteristics is described in 2.6 on page 12.

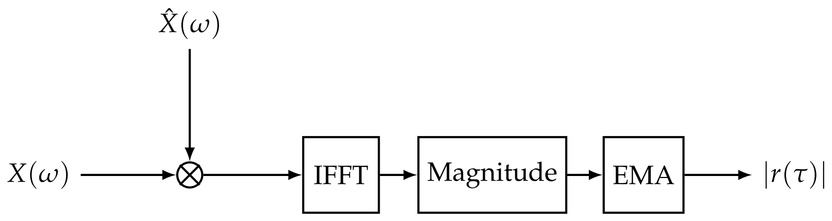


Figure 5.2: Single channel frequency domain correlator

The EMA filter integration constant can be adjusted by using the funtion `set_alpha()` on the acquisition object. This makes it possible to dynamically adjust the length of integration during runtime, and thereby making it possible to change from high sensitivity and slow response to lower sensitivity and rapid response to changes in signal properties.

5.1.2 Local code generator

The *local code generator* is a part of the signal channel correlator, and is used to generate the local code. The code is generated by multiplying the C/A code with a complex carrier, and the result is in turn transformed in the frequency domain and complex conjugated:

$$\hat{X} = \mathcal{F}\{C_i e^{j2\pi f_d t}\}^* \tag{5.1}$$

where $\mathcal{F}\{\cdot\}$ denotes the Fourier transform, and C_i is the C/A code from satellite i.

To reduce the number of calculations required at run-time, the local code is calculated during initialization of the block, and stored in a look-up table. The samples can then be fetched from memory during run-time, and no additional calculations are required.

The local code generator is implemented as an independent signal processing block, but is only used internally in the acquisition module. The source code for the block is given in listing A.1 on page 61.

5.2 Frequency and delay estimator

The frequency and delay estimator compares the signals from each of the single channel correlators, and returns the argument of the maximum value of the two dimensional cross-correlation function between the received and the locally generated signal. This module is essentially the `gr.argmax` block given in listing A.4 on page 66. The `gr.argmax` block accepts an arbitrary number of inputs, where each input is a vector of length N. The input to the `gr.argmax` block is therefore a matrix $|R|$ of dimension $N \times M$, where M is the number for Doppler frequency search bins.

The outputs from the frequency and delay estimator are two individual streams, where one is the index $n \in N$ that corresponds to the estimated C/A code delay, and the other is the index $m \in M$ that corresponds to the estimated Doppler frequency.

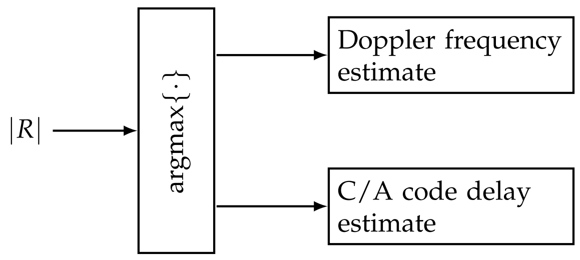


Figure 5.3: Doppler frequency and C/A code delay estimator.

Chapter 6

Results

This chapter the results from the tests of the OpenGNSS acquisition module is presented. Both simulated signals and signals generated with the Spirent GPS simulator described in section 3.1 on page 23 has been used to test the module.

6.1 Simulated signals

Figure 6.1 on the next page, 6.2 on the following page, and 6.3 on page 37 show the result of simulations where the acquisition module was fed a simulated signal from a single satellite with various signal to noise levels. This results are included to verify the basic functionality of the acquisition module.

The power level of the signals for the different SNR levels are similar, so the correlation peak magnitude for the different signals are comparable.

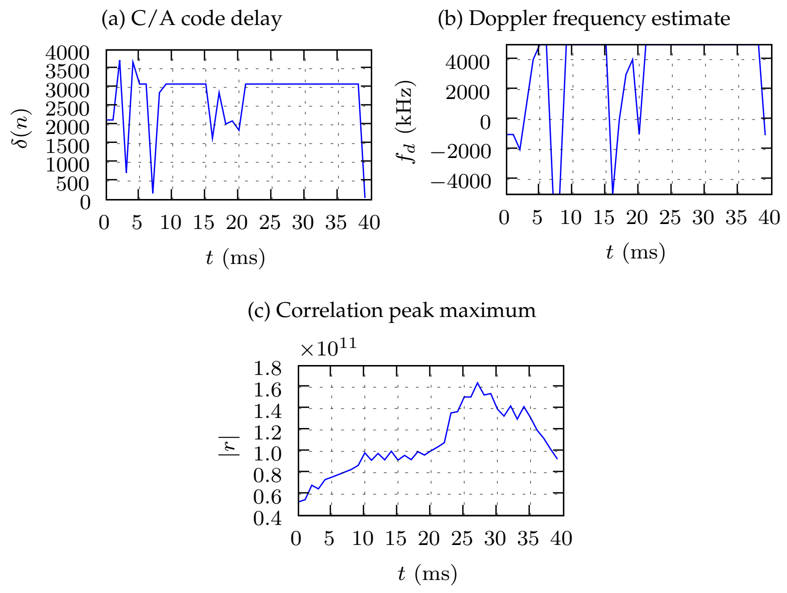


Figure 6.1: Simulated result: $f_d = 5123.00$kHz, SNR -20dB, C/A delay is 3090.0 samples.

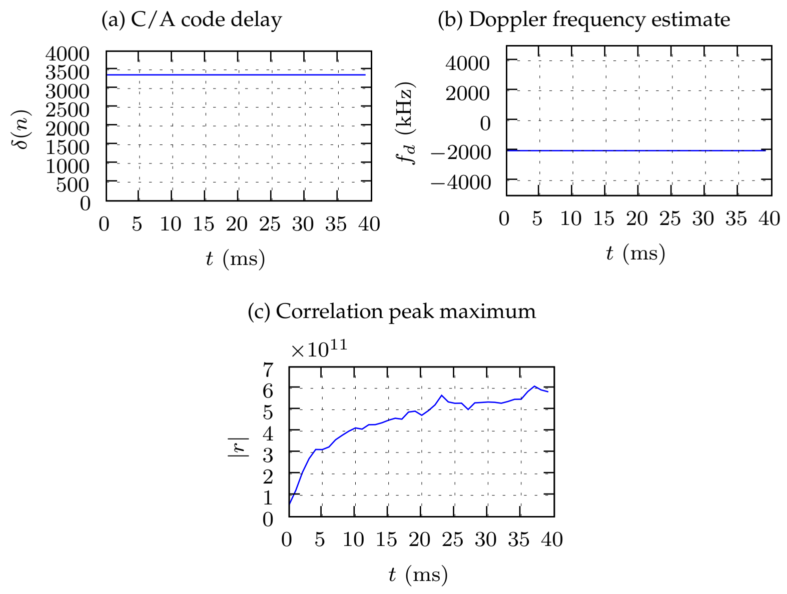


Figure 6.2: Simulated result: $f_d = -2314.00$kHz, SNR -10dB, C/A delay is 3366.0 samples.

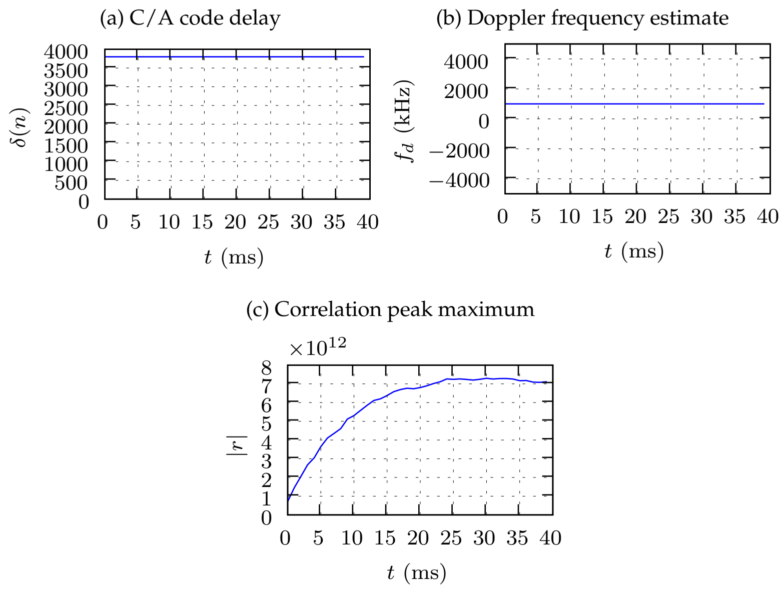


Figure 6.3: Simulated result: f_d = 1123.00kHz, SNR 0dB, C/A delay is 3800.0 samples.

6.2 Single satellite

This section contains the output from the acquisition module when processing signals from the Spirent GPS simulator described in 3.1 on page 23. The simulator was operated in single channel mode, with only the L1 frequency enabled and the P-code disabled.

The USRP settings given in table 6.1 on the next page where used under all tests. Due to the tuning-inaccuracies caused by the oscillator instability described in section 3.3.3 on page 26, the carrier frequency had to be adjusted manually to place the GPS signal as close to baseband as possible. The oscillator has a tolerance of 50 PPM, and when this error is mixed up to 1575.42 GHz, the carrier could be at a maximum 78 kHz away from the correct carrier.

The signal was tuned close to baseband by sending a signal consisting of only the GPS L1 frequency from the simulator and using the `usrp_fft.py` program that is part of GNU Radio to find the correct carrier frequency. The `usrp_fft.py` is a software spectrum analyzer that uses the USRP as the frontend.

This correction could be avoided by increasing the Doppler frequency search range, but that would significantly increase the computational requirements.

Settings	Value
Sampling frequency	4 MHz
USRP Gain:	50 dB
Carrier frequency	1575.436 MHz

Table 6.1: USRP settings.

6.2.1 Length of non-coherent integration

This section contains the output from the acquisition module for different length of non-coherent integration. The length of the non-coherent integration is determined by the parameter α in the exponential moving average filter described in 2.6 on page 12.

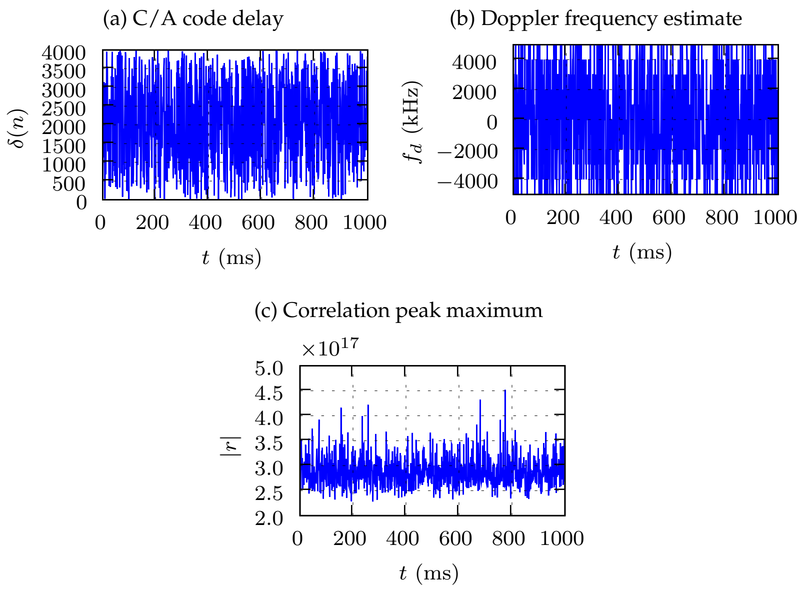


Figure 6.4: Single satellite. SNR -10 dB. $\alpha = 0.500$

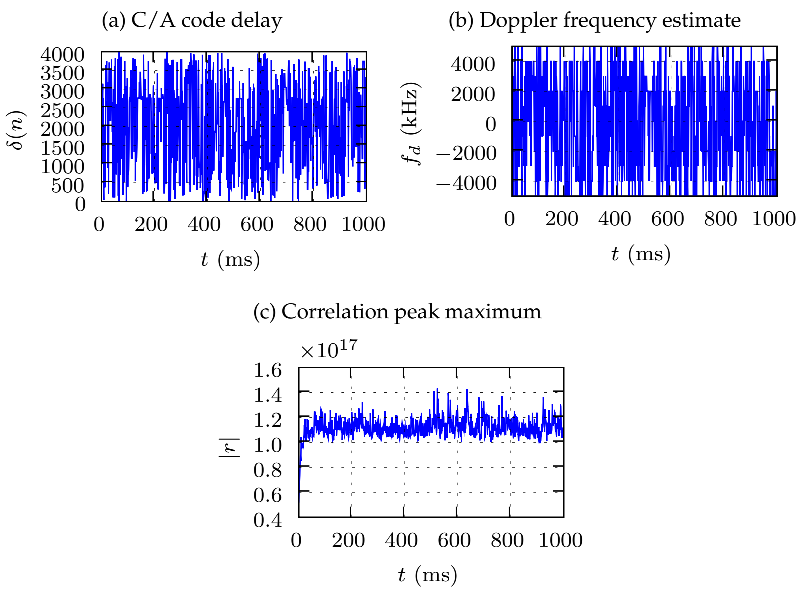


Figure 6.5: Single satellite. SNR -10 dB. $\alpha = 0.100$

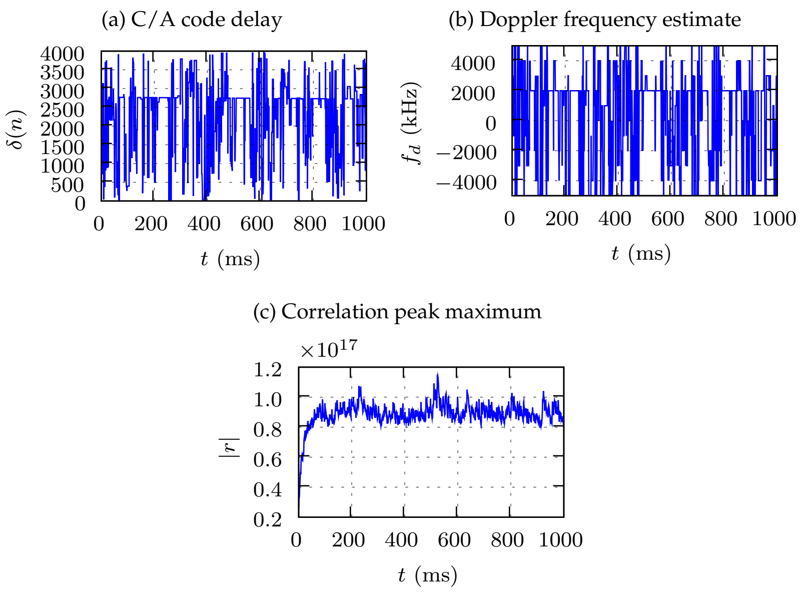


Figure 6.6: Single satellite. SNR -10 dB. $\alpha = 0.050$

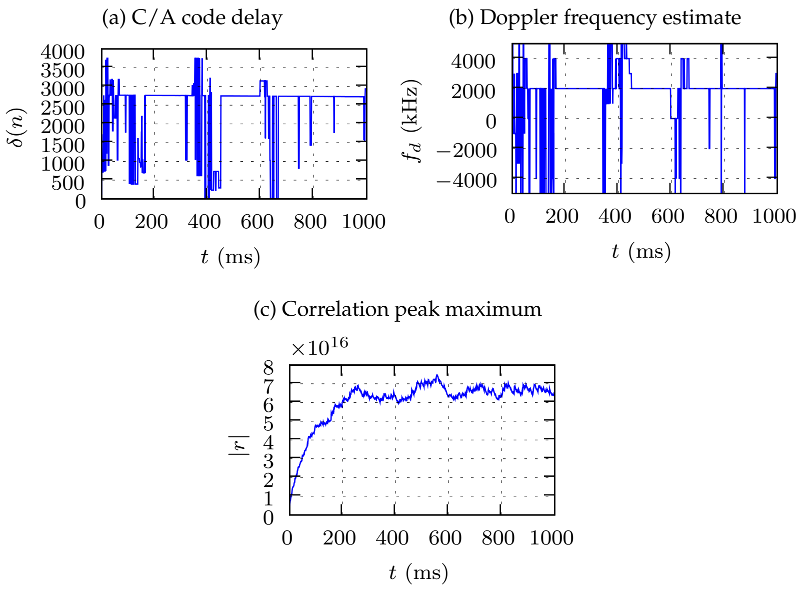


Figure 6.7: Single satellite. SNR -10 dB. $\alpha = 0.010$

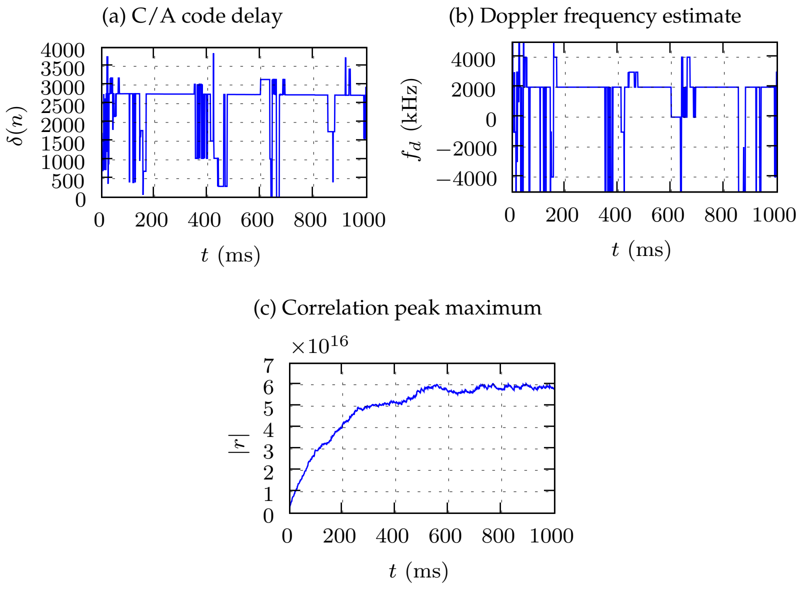


Figure 6.8: Single satellite. SNR -10 dB. $\alpha = 0.005$

6.2.2 Signal to noise ratio

This section shows the results for various signal to noise ratios of the GPS signal. As described in section 3.1 on page 23, the SNR of the GPS signal generated by the Spirent simulator can be adjusted from -20 to 20 dB with respect to the STANAG minimum level of -130 dBm[22]. The integration constant α were set to 0.01 during all of these tests.

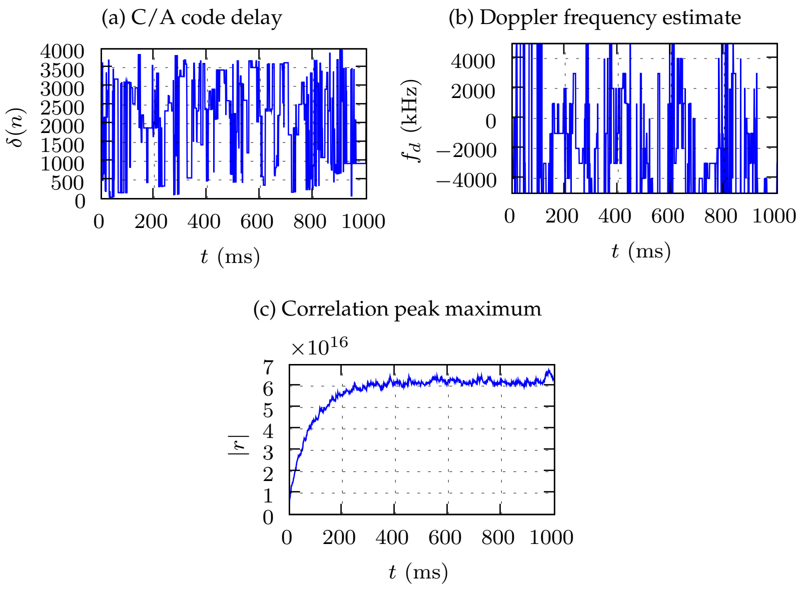


Figure 6.9: Single satellite. SNR -20 dB. $\alpha = 0.010$

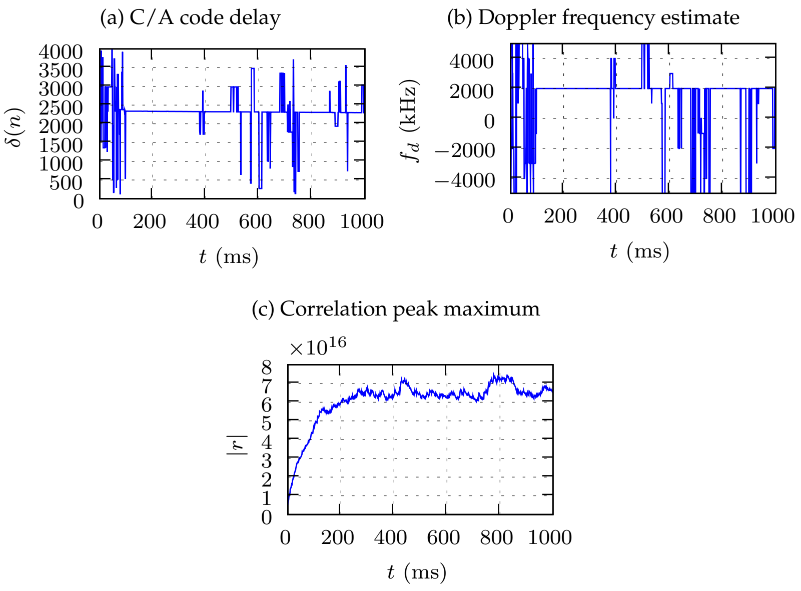


Figure 6.10: Single satellite. SNR -10 dB. $\alpha = 0.010$

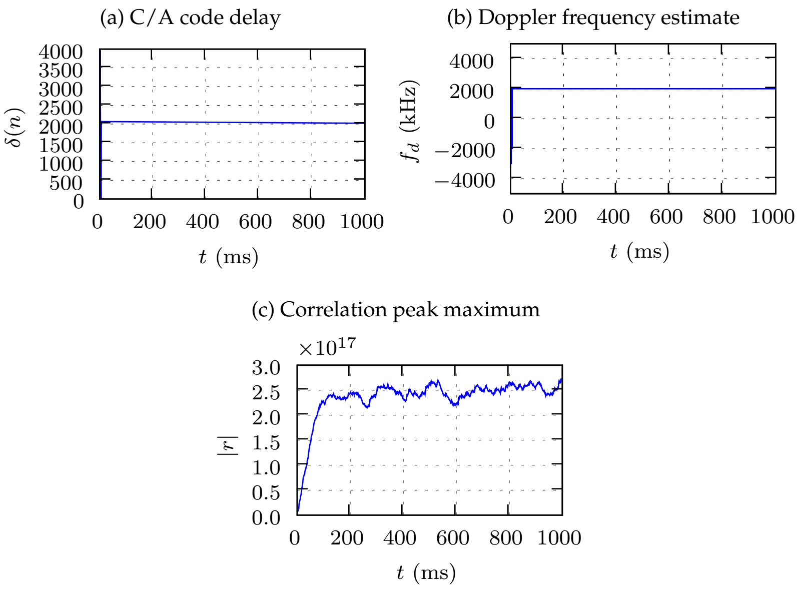


Figure 6.11: Single satellite. SNR 0 dB. $\alpha = 0.010$

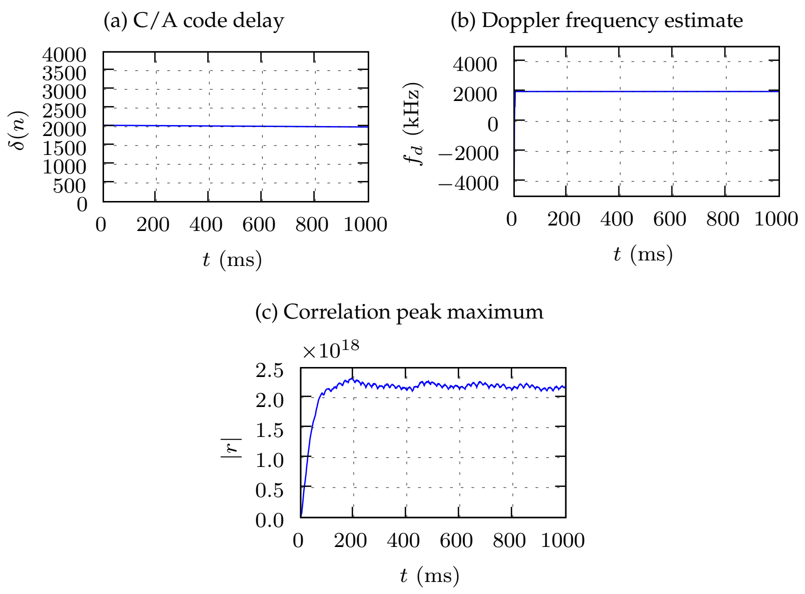


Figure 6.12: Single satellite. SNR 10 dB. $\alpha = 0.010$

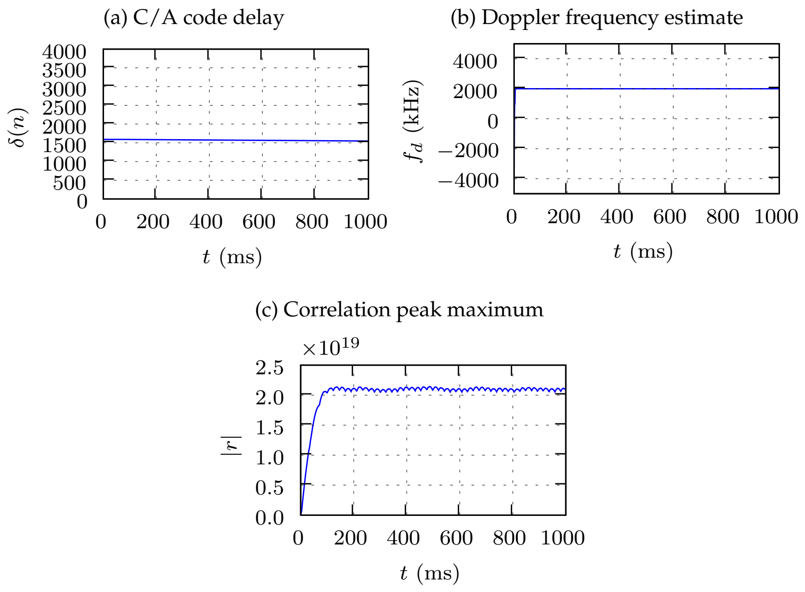


Figure 6.13: Single satellite. SNR 20 dB. $\alpha = 0.010$

6.3 Multiple satellites

Figure 6.14 on the next page, 6.15 on page 46, and 6.16 on page 47 shows the results from the acquisition module when a complete GPS scenario was generated by the Spirent simulator. A constellation of 24 GPS satellites with SVN 1 to 24 were present, and the receiver was stationary in a location with no reflections.

The two-dimension cross-correlation surfaces for each satellite is also shown. It is important to note that the values have been normalized, so that 1.0 corresponds to $|R|_{max}$. The absolute value of the correlation function can be found in sub-figure (c) in each of the figures.

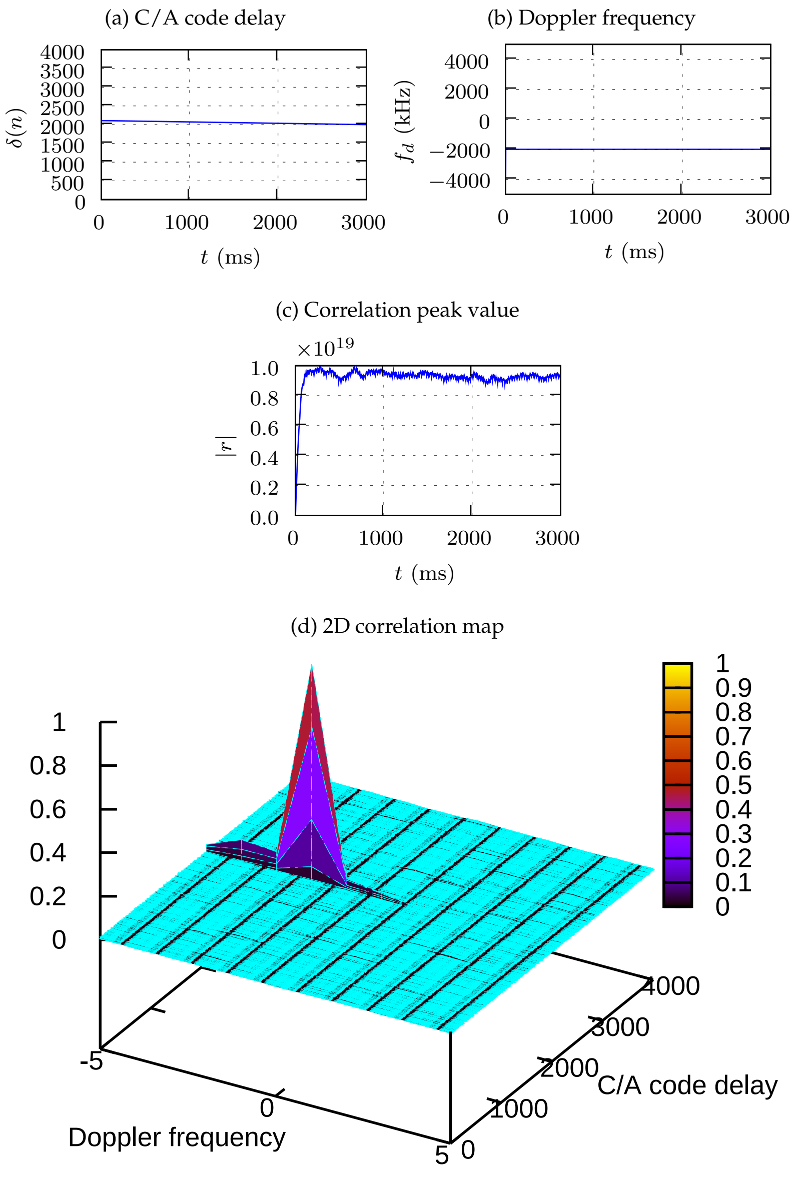


Figure 6.14: Acquisition result on signal with 24 satellites present. SVN 1

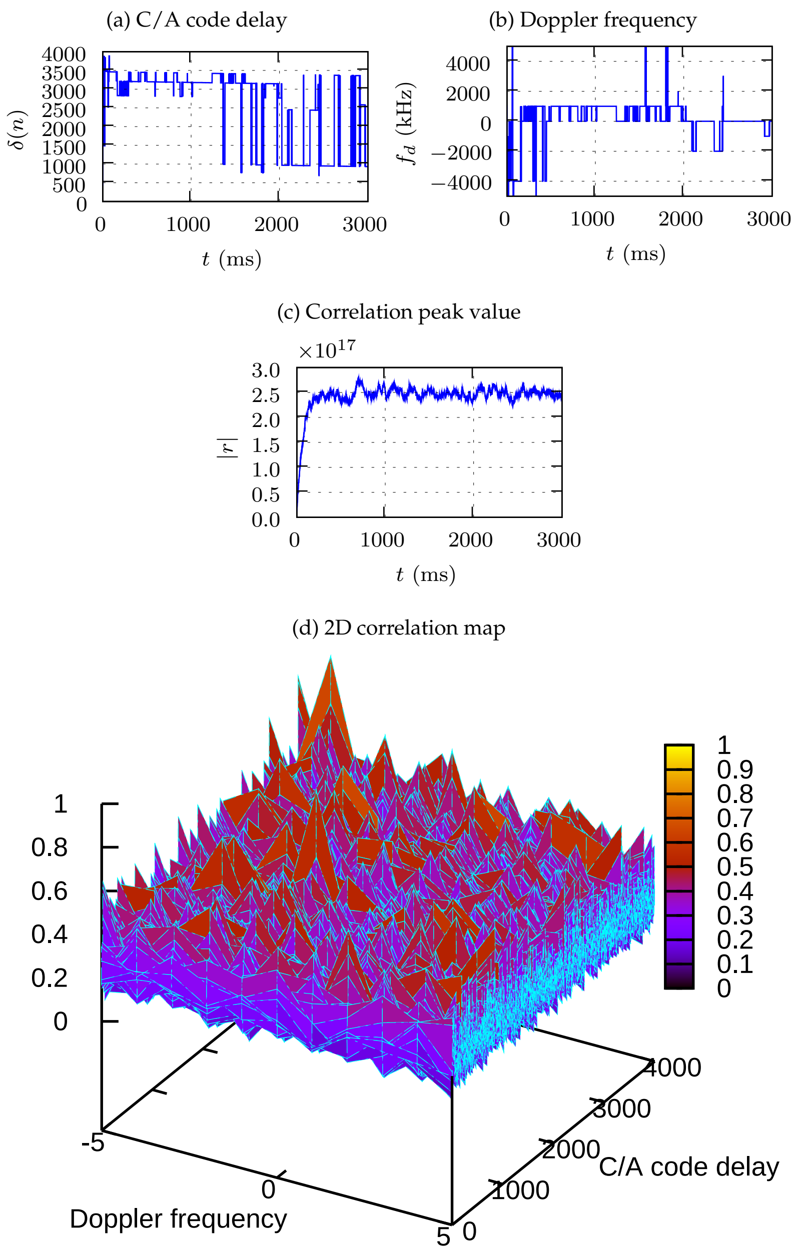


Figure 6.15: Acquisition result on signal with 24 satellites present. SVN 9

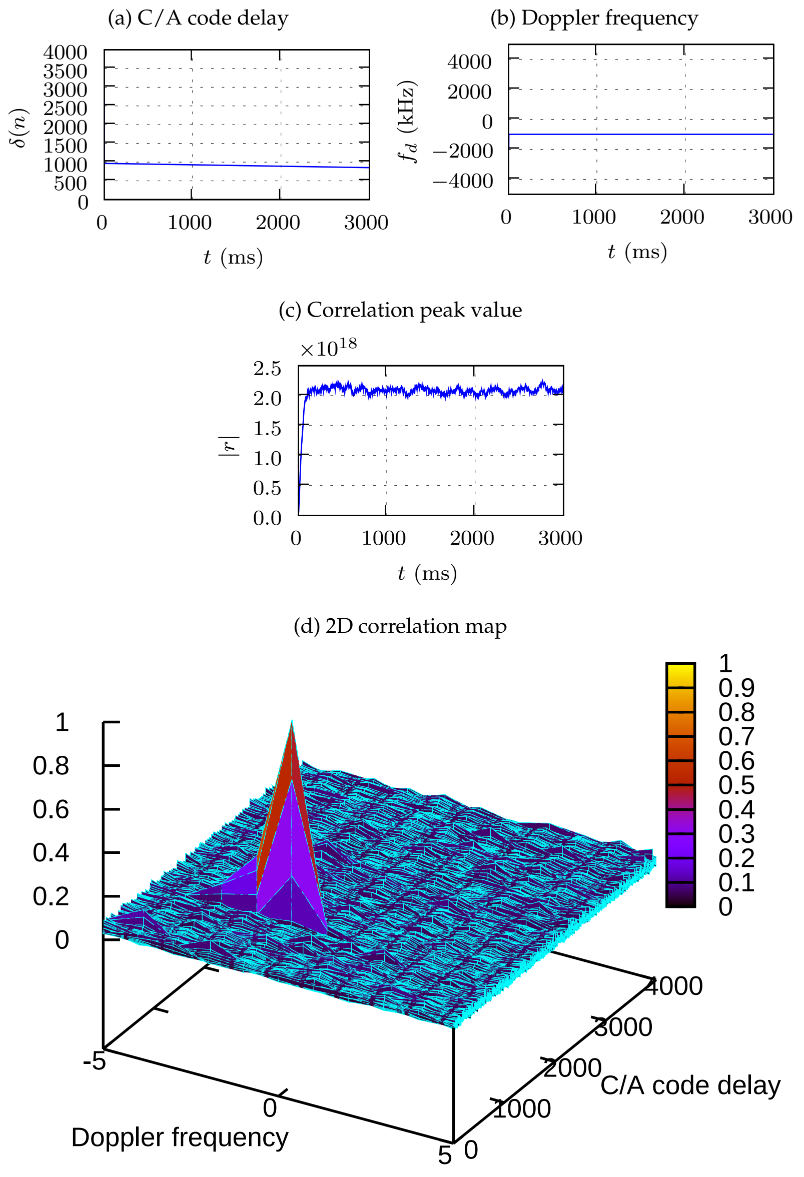


Figure 6.16: Acquisition result on signal with 24 satellites present. SVN 22

6.4 Effect of sampling frequency accuracy

Figure 6.17 shows a close-up version of the C/A code delay from figure 6.14 on page 45. From the figure it is evident that the C/A code delay drifts with approximately 100 samples in three seconds.

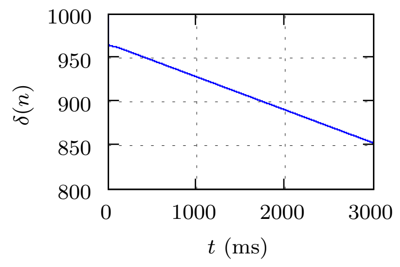


Figure 6.17: Effect of sampling frequency offset on the C/A code delay

6.5 Effect of FFT window function

Figure 6.18 on the facing page and 6.19 on page 50 shows the output from the acquisition module when the signal has been multiplies with different FFT window functions.

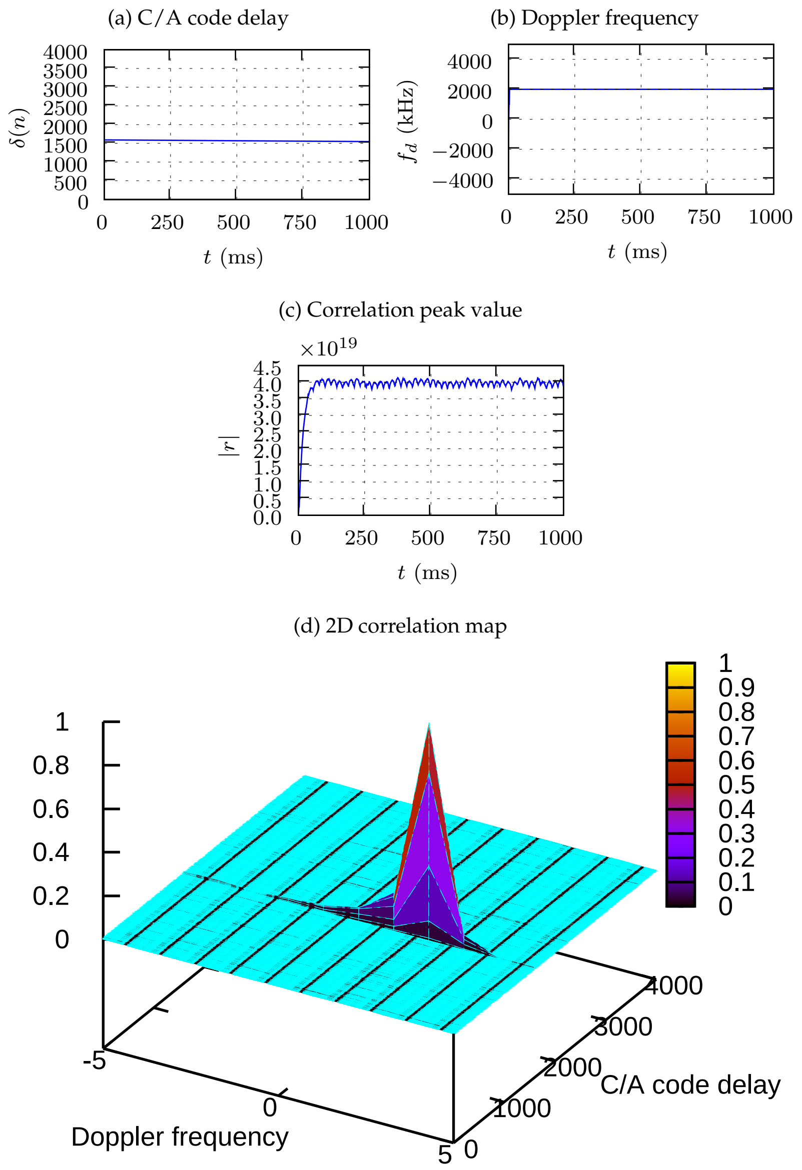
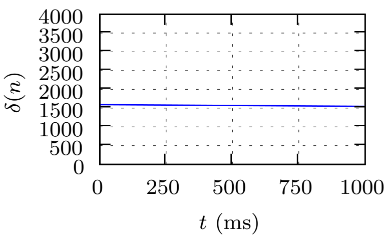
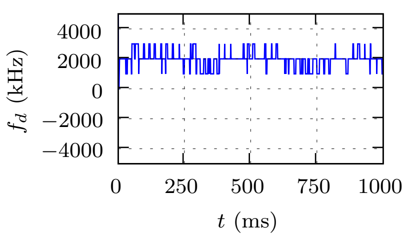


Figure 6.18: Acquisition module output when a rectangular window is used.

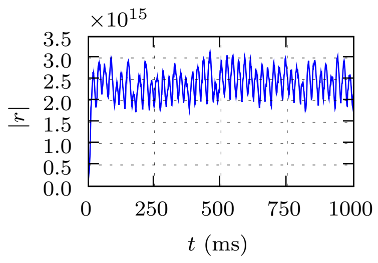
(a) C/A code delay



(b) Doppler frequency



(c) Correlation peak value



(d) 2D correlation map

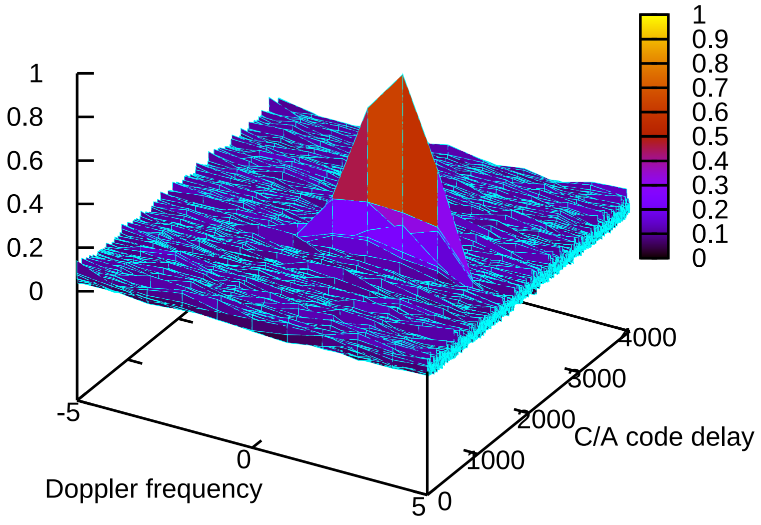


Figure 6.19: Acquisition module output when a Blackman-Harris window is used.

Chapter 7

Discussion

In this chapter the performance of the acquisition module is discussed. Issues regarding limitations in the hardware and in the software are not included in this chapter, but discussed separately in chapter 8 on page 55. This chapter therefore only deals with the current implementation of the acquisition module, and does not address potential improvements on both the design, hardware and software.

Tests has been performed with both simulated signals and signals from a single satellite for verification, and realistic scenarios with complete satellite constellations. The tests show that the acquisition module perform well under ordinary singal conditions. Under week signal conditions further signal processing has to be applied in order to obtain reliable results. A number of different issues such as length of integration, different signal to noise ratios, and cross-correlation properties has been evaluated and are discussed in the following sections.

7.1 Length of non-coherent integration

Given a SNR of -10 dB relative to the STANAG minimum level, it is clear from the results in section 6.2.1 on page 38 that for $\alpha >= 0.05$ it is not possible to estimate the desired parameters. For $\alpha < 0.01$, there are no obvious improvements, despite the increased integration time. $\alpha = 0.01$ has therefore been chosen as the integration constant.

Currently there is no functionality to detect phase transitions caused by the navigation data message. The rate of the navigation data is 50 Hz, which means that every 20th C/A code period there might be a phase transition. For $\alpha = 0.01$, the length of integration is much longer that 20 ms. According to 2.6 on page 12, the impulse response of the EMA filter will drop to 20 % of its maximum after 100 samples, which means that the EMA filter is almost equivalent to a simple sum of 100 consecutive results. Clearly within this time span, there is a high probability of a phase transition cause by the navigation data message. This does not cause any significant problems for strong signals, but might be one of the limiting factors when processing weak signals.

The OpenGNSS acquisition module uses an exponential moving average filter for integrating the signal. This may not be the optimal filter for this application, since it places more confidence in the most recent observations. Since the rate of change of the GPS signal parameters is very slow compared to the period of the C/A code, the GPS signal parameters are weak stationary for all reasonable length of integration. Therefore an ordinary moving average which places equal weight on all observations might have been more appropriate, but because of the computational requirement of a SMA filter compared to a EMA filter, this has not been implemented.

7.2 Signal to noise ratio

From simulated results in 6.1 on page 35, it is clear that the acquisition module should able to correctly estimate the parameters from the signal from a single satellite at least down to -10 dB. Below this, the parameters can not be obtained without further signal processing. Techniques for for acquisition of week signals have not been investigated in this paper.

When processing signals from the Spirent simulator, the acquisition module is not able to correctly estimate the parameters below the STANAG minimum level. There are a number of reasons why this result is different from the result with the simulated signals, but the most important factor is that the SNR loss caused by the LNA, the DBSRX and the digital down-converstion in the USRP has not been taken into account in the SNR level given in 6.2.2 on page 41. The actual SNR level is therefore lower than what is indicated.

However, the tests show that the OpenGNSS acquisition module is able to successfully acquire a signal at the STANAG minimum level.

7.3 Multiple satellites

The results from the tests with realistic signals provide similar results to ones found in the single satellite case. From figure 6.14d on page 45 and 6.16d on page 47 it is clear that satellite 1 and 22 which were visible from the receiver location could be acquired, while satellite 9 could not. From these tests it is clear that the acquisition module works well under normal signal conditions, and that the cross-correlation between the different satellites does not affect the acquisition procedure to a great extent.

7.4 Effect of sampling frequency accuracy

From figure 6.17 on page 48 it is found that the C/A code drifts by approximately 100 samples in three seconds. According to equation 2.9 on page 10, this corresponds to a sampling frequency offset of:

$$\Delta f_s = \frac{100}{3000 t_c} = 3.3\,\text{Hz} \tag{7.1}$$

This result is well within the oscillator tolerance of 50 PPM, which at 1.023 MHz gives a maximum error of approximately 50 Hz.

7.5 Effect of FFT window function

Section 6.5 on page 48 shows the results of the tests where the GPS signal has been multiplied with different window functions. From these tests it is evident that window functions other than the rectangular window has a major negative impact on the acquisition estimate. The choice of FFT window affects both the spectral leakage and the correlation peak magnitude, but from the results seen in figure 6.18d on page 49 it can be seen that the high side-lobe levels of the rectangular window does not affect the correlation peak to a great degree. The wide main-lobe and the reductions of the correlation peak magnitude caused by the Blackman-Harris window however, has a major impact on the estimate. From figure 6.19d on page 50 is is clear that the wide main-lobe of the Blackman-Harris causes a significant spectral leakage to adjacent side-bands which makes it very hard to determine the correct Doppler frequency. This can be seen in figure 6.19 on page 50, where the Doppler frequency estimate fails to lock onto the correct frequency bin, but jumps between the correct and the neighbouring bins. The C/A code estimate is not affected that much, but the reduced correlation peak magnitude increases the required SNR by approximately 5 dB as seen in 2.2 on page 11.

Chapter 8

Future work

In this chapter the various limitations of the hardware, the software, and the design of the OpenGNSS acquisition module is discussed. This chapter should be seen as a continuation of previous chapter, and address issues that were discovered during the development of the OpenGNSS acquisition module. Proposals for future work are therefore presented in this chapter. These modifications include changes to both the hardware, the software, and the software receiver design it self.

8.1 GNU Radio limitations

There are a number of factors that curtail the usefulness of the GNU Radio framework for building complete GNSS receivers. The current architecture of the framework is suitable for processing continuous streams of data such as broadcast radio and TV, but not very well suited for packet based radios. Although the GPS satellites are broadcasting their signals in a continuous manner, the internal processing is more packet based, and therefore a GNSS receiver would greatly benefit from changes that makes the GNU Radio framework more adaptable to such applications.

The ADROIT[1] project has proposed a number of architectural changes to the GNU Radio framework that makes it more suitable for packet radio applications. The complete set of changes are described in [2], and include:

- Message-based scheduling.
- Standardized time transfer mechanism and common time reference.
- Dynamic reconfigurability.
- Support for control interfaces.

The main extension that implements all of these features, is the *message block*, or *m-block*, extension. All of the items mentioned above are described briefly in the following sections:

[1]The ADROIT project [23] is a joint project funded by DARPA woes goal is to create open-source cognitive software radios.

Message-based scheduling Since the GNU Radio framework has been created to process continuous streams of data, a fixed number of input items is aggravated in the FIFO buffer between blocks before a block is put in the run queue of the scheduler. This is not very suitable for packet radios where packet have varying sizes, and is certainly not very desirable for sending control messages between blocks since it introduces unknown latencies into the system.

To circumvent this limitation, a new message-based scheduler has been proposed. The message-based scheduler is fundamentally different from the stream-based scheduler, since it allows arbitrarily sized block of data coupled with meta-data to be passed through the system. The structure of the messages used in the new architecture is shown in table 8.1.

Field	**Description**
Message handle	Unique identifier.
Priority	Messages can be assigned different priorities.
Signal	This is an event name that can be used to control finite-state machines (FSM) within processing blocks.
Timestamps	standardized timestamps that allows synchronization across different flow_graphs and between different threads.
Data	user-defined data to be processed by the blocks in an ordinary *flow_graph*.

Table 8.1: Sectional description of message fields.

Standardized time transfer mechanism and common time reference With the aid of a common time reference and a standard mechanism for this information, is it possible both to measure the latency through different parts of the flow graph, and to ensure synchronization between parallel flow graphs and between different threads and processes. This would be a greatly beneficial feature when calculating the time of arrival of the signal from the different satellites.

Dynamic reconfiguration at runtime Currently it is not possible to reconfigure a flow graph at runtime. In a GNSS receiver it would be desirable to be able to add additional channels dynamically at runtime. That way one could add additional channels if the current number of satellites does not provide the desired result, and to start searching for additional satellites when the signal quality of one of the current satellites is fading.

It would also make it possible to connect and disconnect different tracking loops depending on the signal quality. Usually a phase lock loop is used to track the incoming signal, but depending on signal quality, it can also be used together with a delay lock loop or a frequency lock loop which perform better than the PLL under certain situations.

By being able to connect and disconnect the individual tracking loops the flexibility of the system is increased without decreasing the overall runtime performance.

Support for control interfaces By the use of the signal flag, priority and metadata it is possible to send control message from both internal and external entities. Currently one of the biggest limitations of the framework is the lack of mechanisms to pass control information and parameters such as Doppler frequency and C/A code delay from one part of the system to another. This will be simplified with the new m-block architecture.

8.2 Hardware limitations

There are several limitations with the current hardware setup with regards to GNSS applications.

- The lack of a more suitable RF frontend.
- No hardware multiplier in the FPGA.
- Limited bandwidth over the USB.

These items reduce the current performance of the USRP for GNSS applications, and also limits the potential for future expansions.

8.2.1 RF front-end limitations

The MAX2118 tuner IC which the DBSRX daughterboard is based on the is not designed for GNSS applications, and is therefore not optimal for such applications. The noise figure of the MAX2118 has a typical value of 10.5 dB[4], which is a fairly high value compared to LNAs that are typically used. This means that an external low-noise preamp must be used as between the antenna and the daughterboard.

Initially tests were performed without the LNA, but due to the high noise figure of the MAX2118 (see 3.3.1 on page 24) it was not possible to obtain satisfying results without an external preamp.

Oscillator stability is also a source of many problems. As given in section 3.3.3 on page 26, the oscillator tolerance should be within ± 50 Hz. When this clock signal is up-converted in the daughterboard, the carrier frequency will have the same error, which means that the L1 carrier at 1575.42MHz will have a tolerance of:

$$f_c 50\text{ PPM } = f_c \frac{50}{10^6} \approx 78\text{ kHz} \tag{8.1}$$

A dedicated high performance oscillator for the daughterboard would improve the carrier frequency precision and also help reduce the phase noise generated by the PLL in the MAX2118. Furthermore an analog filter such as a SAW filter should be used to suppress out-of-band noise before the preamp and the mixer.

The analog filter and the preamp are not crucial modifications, since they can easily be attached between the antenna and the daughterboard. The oscillator on the USRP can also be replaced with an external, stabilized clock source. However, a dedicated GNSS daughterboard would be a very interesting project for future work.

8.2.2 USRP limitations

Since the FPGA lacks hardware multipliers, it is difficult to move functionality such as the correlators and the FFT from the host computer and to the USRP. In order to implement support from new and more advance modulations available from both Galileo and GPS which uses more bandwidth than the C/A code, several parts of the demodulation must be moved from the host computer to the FPGA because of the limited bandwidth of the USB. This limits the potential for expansion to more advanced receiver designs without major modifications to the hardware.

There is ongoing work to improve the USRP, and hopefully the next revision will upgrade the FPGA to a more powerful version. There has also been discussions on the GNU Radio mailing list about replacing the USB with a faster interface such as gigabit Ethernet, but nothing has been decided yet.

8.3 OpenGNSS design limitations

Although the current design has not yet revealed any major limitations or flaws, there are a couple of issues that should be mentioned. The first one the fact that although the current implementation is very modular and flexible, it places high requirements on the processing computer. Since each block is connected with a FIFO, the memory requirement increases with the Doppler frequency search range and with the number of channels that are tracked in parallel.

A monolithic design would have reduced the memory usage at the expence of the flexibility. It would potentially also increase the performance of the system, because a monolithic design would avoid much of the memory copying that happends when data is moved from one block to another.

The second point that should be mentioned is related to the removal of the Doppler shift from the signal. In the initial design the Doppler removal was performed in the host computer. However, the USRP comes with an alternative FPGA image which contains 4 RX paths. This makes it possible to perform down-conversion of the incoming signal on four different frequencies, and thereby perform Doppler removal on four satellites in parallel in hardware.

8.4 Effect of multipath reflections

Initially is was desired to measure the performance of the acquisition module in scenarios with both strong and week reflected signals. However, it proved very difficult to obtain quantifiable results on the performance of the module under such conditions. It is expected that the reflections would broaden the correlation peak, but since no techniques to suppress reflected signals have implemented in the OpenGNSS, no results on the performance under such conditions are available. It would however be a very interesting topic for future work to improve the acquisition module with regard to such conditions.

Chapter 9

Conclusion

In this paper an acquisition module for the OpenGNSS software-defined receiver has been discussed. OpenGNSS is a flexible and modular software receiver targeted at both researchers, developers and for educational purposes. Other similar receivers are either high-performance implementations in either C or assembly, or off-line analysis tools written in Matlab. The unique thing about the OpenGNSS is that it combines the flexibility of the scripted language Python with the performance of C++. This makes it usable for both real-time or off-line uses.

The entire topology and functionality of the receiver is defined in Python scripts, and it is also possible to tap the signal at all stages of the receiver. This makes it an ideal choice for students who want to study the inner workings of a GNSS receiver. It also makes it an interesting choice for developers who are integrating the receiver with other navigational aids such as inertial navigation systems, or for researchers who are investigating atmospheric effects on the GNSS signal since they have simultaneous access to raw signal and the processed data.

Although the complete receiver is not finished, the acquisition module which is the first stage after the analog processing shows promising results. The performance of the acquisition module is satisfactory under normal signal conditions, and a number of issues that would improve the system has been presented in chapter 8 on page 55.

GNU Radio limitations A number of issues regarding the GNU Radio framework has been addressed. The predominant limitations of the framework is the lack of suitable mechanisms for transfer of control parameters between different signal processing blocks. The GNU Radio framework were originally created to process continuous streams for data, and no packet-based radio signals. However, these issues has been addressed with the new message-block extension that has been proposed. It is considered unrealistic to build a complete receiver without the features provided by this new extension.

Hardware limitations A couple of issues were also mentioned regarding the radio front-end of the receiver. This is not an optimal receiver for GNSS applications, in particular with regard to noise figures and oscillator stability. Although sufficient under normal and strong signal conditions, this is the limiting factor if the receiver should be improved to support week signal conditions such as those experienced in-door and in urban areas. A custom GNSS RF board would therefore be an interesting topic for future work that would be a significant improvement for the performance and expandability of the system as a whole.

Commercial support Another issue that has not been mentioned earlier, is the lack of commercial support for GNU Radio. Although the support provided by GNU Radio community has been indispensable, it is a fact that software developers are more fond of writing software than documentation. Although GNU Radio has extensive application programming interface documentation, the lack up-to-date tutorials might be intimidating for new users. This has not been a major problem for the development of the OpenGNSS acquisition module, but it has clearly been a retarding factor for the speed of progress.

But despite these issues, the OpenGNSS receiver has proven to be a promising concept. With the computational power available in modern FPGAs, DSPs and ordinary computers, SDR has become the dominant technology for many applications. The flexibility offered by the SDR approach outweighs the disadvantages when compared to a conventional hardware-based design:

- Easy access to the signal in all parts of the receiver.
- Dynamic reconfigurability.
- Reuse of existing source code and ease of transfer of the receiver to a new target.
- Modular design makes it easy to replace individual parts without affecting other parts of the system.

By taking advantage of existing source code provided by a number of free software project, the development time of the OpenGNSS acquisition module has been significantly reduced. It would not have been reasonable to implement the equivalent functionality from scratch within the same timespan. It is therefore clear that free software combined with suitable hardware can be a powerful tool for building GNSS receiver.

Appendix A

Source Code

A.1 OpenGNSS acquisition module

Listing A.1: OpenGNSS local code generator

```
#    Copyright 2007 Trond Danielsen <trond.\
   danielsen@gmail.com>
#
#    This file is part of OpenGNSS.
#
#    This program is free software; you can redistribute\
     it and/or modify
#    it under the terms of the GNU General Public \
   License as published by
#    the Free Software Foundation; either version 2 of \
   the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it \
   will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied \
   warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE\
   .  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General \
   Public License
#    along with this program; if not, write to the Free \
   Software
#    Foundation, Inc., 51 Franklin Street, Boston, MA  \
   02110-1301  USA

from gnuradio import gr,window
from numpy import *
from gps import ca_code
```

```
class local_code(gr.hier_block2):
    def __init__(self, svn, fs, fd):

        # Compute local code in advance to reduce the \
            number of runtime
        # calculations required.
        code = ca_code(svn=svn, fs=fs)
        n = arange(len(code))
        f = e**(2j*pi*fd*n/fs)
        lc = conj(fft.fft(code * f))

        gr.hier_block2.__init__(self,
            "local_code",
            gr.io_signature(0,0,0),
            gr.io_signature(1,1, len(lc)*gr.\
                sizeof_gr_complex))

        src = gr.vector_source_c(lc, True)
        s2v = gr.stream_to_vector(gr.sizeof_gr_complex, \
            len(lc))

        self.connect( src, s2v, self)

# vim: ai ts=4 sts=4 et sw=4
```

Listing A.2: OpenGNSS coarse acquisition module

```
#    Copyright 2007 Trond Danielsen <trond.\
    danielsen@gmail.com>
#
#    This file is part of OpenGNSS.
#
#    This program is free software; you can redistribute\
     it and/or modify
#    it under the terms of the GNU General Public \
    License as published by
#    the Free Software Foundation; either version 2 of \
    the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it \
    will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE\
    .  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General \
    Public License
#    along with this program; if not, write to the Free \
    Software
#    Foundation, Inc., 51 Franklin Street, Boston, MA  \
    02110-1301  USA

from gnuradio import gr
from single_channel_correlator import *


class acquisition(gr.hier_block2):
    # Output 0 is the C/A code delay.
    # Output 1 is the Doppler frequency estimate in Hz.
    # Output 2 is the correlation peak value.

    def __init__(self, fs, svn, alpha, fd_range, \
        dump_bins=False):
        gr.hier_block2.__init__(self,
            "acquisition",
            gr.io_signature(1,1, gr.sizeof_gr_complex),
            gr.io_signature(3,3, gr.sizeof_float))

        fft_size = int( 1e-3*fs)
        doppler_range = self.get_doppler_range(fd_range)

        agc = gr.agc_cc( 1.0/fs, 1.0, 1.0, 1.0)
        s2v = gr.stream_to_vector(gr.sizeof_gr_complex, \
           fft_size)
        fft = gr.fft_vcc(fft_size, True, [])

        argmax = gr.argmax_fs(fft_size)
        max = gr.max_ff(fft_size)

        self.connect( self, s2v, fft)
        self.connect( (argmax, 0),
                gr.short_to_float(),
                (self, 0))
        self.connect( (argmax,1),
                gr.short_to_float(),
                gr.add_const_ff(-fd_range),
                gr.multiply_const_ff(1e3),
                (self,1))
        self.connect( max, (self, 2))

        # Connect the individual channels to the input \
           and the output.
        self.correlators = [ single_channel_correlator( \
           fs, fd, svn, alpha, dump_bins) for fd in \
           doppler_range ]
        for (correlator, i) in zip( self.correlators, \
           range(len(self.correlators))):
            self.connect( fft, correlator )
            self.connect( correlator, (argmax, i) )
            self.connect( correlator, (max, i) )


    def set_alpha(self, alpha):
        for correlator in self.correlators:
            correlator.set_alpha(alpha)


    def get_doppler_range(self, fd_range):
        """Range␣is␣given␣in␣kHz.
␣␣␣␣␣␣␣␣Step␣length␣is␣currently␣hard␣coded␣to␣1kHz."""
        step = 1e3
        return range( int(-fd_range*1e3), int((fd_range\
           +1)*1e3), int(step))

# vim: ts=4 sts=4 sw=4 sta et ai
```

Listing A.3: OpenGNSS single channel correlator

```
#    Copyright 2007 Trond Danielsen <trond.\
    danielsen@gmail.com>
#
#    This file is part of OpenGNSS.
#
#    This program is free software; you can redistribute\
     it and/or modify
#    it under the terms of the GNU General Public \
    License as published by
#    the Free Software Foundation; either version 2 of \
    the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it \
    will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE\
    .  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General \
    Public License
#    along with this program; if not, write to the Free \
    Software
#    Foundation, Inc., 51 Franklin Street, Boston, MA  \
    02110-1301  USA
```

```

from gnuradio import gr
from local_code import local_code
import os


class single_channel_correlator(gr.hier_block2):
    def __init__(self, fs, fd, svn, alpha, dump_bins=\
        False):
        fft_size = int( 1e-3*fs)

        gr.hier_block2.__init__(self,
            "single_channel_correlator",
            gr.io_signature(1,1, gr.sizeof_gr_complex*\
                fft_size),
            gr.io_signature(1,1, gr.sizeof_float*\
                fft_size))

        lc = local_code(svn=svn, fs=fs, fd=fd)
        mult = gr.multiply_vcc(fft_size)
        ifft = gr.fft_vcc(fft_size, False, [])
        mag = gr.complex_to_mag_squared(fft_size)
        self.iir = gr.single_pole_iir_filter_ff( alpha, \
            fft_size)

        self.connect( self, (mult, 0))
        self.connect( lc,   (mult, 1))
        self.connect( mult, ifft, mag, self.iir, self)

        if dump_bins == True:
            self.connect_debug_sink(self.iir,fft_size,'/\
                home/trondd/opengnss_output', fd)


    def set_alpha(self, alpha):
        self.iir.set_taps(alpha)


    def connect_debug_sink(self, src, fft_size, \
        output_path, fd):
        filename = os.path.join(output_path, "fd_%d.dat"\
             % fd)
        file_sink = gr.file_sink(fft_size*gr.\
            sizeof_float, filename)
        self.connect( src, file_sink )


# vim: ai ts=4 sts=4 et sw=4
```

A.2 GNU Radio argmax extension

Listing A.4: /gnuradio-core/src/lib/gengen/gr_argmax_XX.cc.t

```
/* -*- c++ -*- */
/*
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
     See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <@NAME@.h>
#include <gr_io_signature.h>

@SPTR_NAME@
gr_make_@BASE_NAME@ ( size_t vlen )
{
  return @SPTR_NAME@ ( new @NAME@(vlen));
}

```

```
@NAME@::@NAME@( size_t vlen)
  : gr_sync_block ( "@BASE_NAME@",
                    gr_make_io_signature (1, -1, vlen*\
                       sizeof (@I_TYPE@)),
                    gr_make_io_signature (2, 2, sizeof (\
                       @O_TYPE@))),
  d_vlen(vlen)
{
}


int
@NAME@::work( int noutput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items)
{

  int ninputs = input_items.size ();

  @O_TYPE@ *x_optr = (@O_TYPE@ *) output_items[0];
  @O_TYPE@ *y_optr = (@O_TYPE@ *) output_items[1];

  for (int i=0; i<noutput_items; i++) {

    @I_TYPE@ max = 0;
    int x = 0;
    int y = 0;

    for (int j=0; j < (int) d_vlen; j++ ) {
      for (int k=0; k<ninputs; k++) {
        if ( ((@I_TYPE@ *) input_items[k])[i*d_vlen + j]\
            > max) {
          max = ((@I_TYPE@ *) input_items[k])[i*d_vlen +\
              j];
          x = j;
          y = k;
        }
      }
    }

    *x_optr++ = (@O_TYPE@) x;
    *y_optr++ = (@O_TYPE@) y;
  }
  return noutput_items;
}
```

Listing A.5: /gnuradio-core/src/lib/gengen/gr_argmax_XX.h.t

```
/* -*- c++ -*- */
/*
```

```
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
     See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifndef @GUARD_NAME@
#define @GUARD_NAME@

#include <gr_sync_block.h>

class @NAME@;
typedef boost::shared_ptr<@NAME@> @SPTR_NAME@;

@SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);


class @NAME@ : public gr_sync_block
{
  friend @SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);

  @NAME@ (size_t vlen);
  size_t d_vlen;

 public:
  int work (int noutput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items);
};


#endif
```

Listing A.6: /gnuradio-core/src/lib/gengen/gr_argmax_XX.i.t

```
/* -*- c++ -*- */
/*
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
     See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

GR_SWIG_BLOCK_MAGIC(gr,@BASE_NAME@)

@SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);

class @NAME@ : public gr_sync_block
{
 private:
  @NAME@ (size_t vlen);
  size_t d_vlen;
};
```

A.3 GNU Radio max extension

Listing A.7: /gnuradio-core/src/lib/gengen/gr_max_XX.cc.t

```
/* -*- c++ -*- */
/*
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
     See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <@NAME@.h>
#include <gr_io_signature.h>

@SPTR_NAME@
gr_make_@BASE_NAME@ ( size_t vlen )
{
  return @SPTR_NAME@ ( new @NAME@(vlen));
}
```

```
@NAME@::@NAME@( size_t vlen)
  : gr_sync_block ( "@BASE_NAME@",
                   gr_make_io_signature (1, -1, vlen*\
                      sizeof (@I_TYPE@)),
                   gr_make_io_signature (1, 1, sizeof (\
                      @O_TYPE@))),
       d_vlen(vlen)
{
}

int
@NAME@::work( int noutput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items)
{
  @O_TYPE@ *optr = (@O_TYPE@ *) output_items[0];

  int ninputs = input_items.size ();

  for (int i=0; i<noutput_items; i++) {

    @I_TYPE@ max = 0;

    for (int j=0; j < (int) d_vlen; j++ ) {
      for (int k=0; k<ninputs; k++) {
        if ( ((@I_TYPE@ *) input_items[k])[i*d_vlen + j]\
             > max) {
          max = ((@I_TYPE@*) input_items[k])[i*d_vlen + \
              j];
        }
      }
    }

    *optr++ = (@O_TYPE@) max;
  }
  return noutput_items;
}
```

Listing A.8: /gnuradio-core/src/lib/gengen/gr_max_XX.h.t

```
/* -*- c++ -*- */
/*
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
     See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

#ifndef @GUARD_NAME@
#define @GUARD_NAME@

#include <gr_sync_block.h>

class @NAME@;
typedef boost::shared_ptr<@NAME@> @SPTR_NAME@;

@SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);


class @NAME@ : public gr_sync_block
{
  friend @SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);

  @NAME@ (size_t vlen);
  size_t d_vlen;

 public:

  int work (int noutput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items);
};


#endif
```

Listing A.9: `/gnuradio-core/src/lib/gengen/gr_max_XX.i.t`

```
/* -*- c++ -*- */
/*
 * Copyright 2007 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it \
    and/or modify
 * it under the terms of the GNU General Public License \
    as published by
 * the Free Software Foundation; either version 2, or (\
    at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be \
    useful,
 * but WITHOUT ANY WARRANTY; without even the implied \
    warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. \
      See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General \
    Public License
 * along with GNU Radio; see the file COPYING.  If not, \
    write to
 * the Free Software Foundation, Inc., 51 Franklin \
    Street,
 * Boston, MA 02110-1301, USA.
 */

// @WARNING@

GR_SWIG_BLOCK_MAGIC(gr,@BASE_NAME@)

@SPTR_NAME@ gr_make_@BASE_NAME@ (size_t vlen);

class @NAME@ : public gr_sync_block
{
 private:
  @NAME@ (size_t vlen);
  size_t d_vlen;
};
```

Bibliography

[1] *AD9860/AD9862 Mixed-Signal Front-End (MxFE™) Processor for Broadband Communications.*
URL `http://www.analog.com/UploadedFiles/Data_Sheets/AD9860_9862.pdf`.

[2] GNU Radio Architectural Changes, 2006.
URL `http://acert.ir.bbn.com/downloads/adroit/gnuradio-architectural-enhanc%ements-3.pdf`.

[3] *Cyclone Device Handbook.*
URL `http://www.altera.com/literature/hb/cyc/cyc_c5v1.pdf`.

[4] *MAX2118 Complete DBS direct-conversion tuner ICs with monolithic VCOs.*
URL `http://datasheets.maxim-ic.com/en/ds/MAX2116-MAX2118.pdf`.

[5] Elisardo Antelo, Javier D. Bruguera, T. Lang, and Emilio L. Zapata. Error Analysis and Reduction for Angle Calculation Using the CORDIC Algorithm. *IEEE Transactions on Computers*, 46(11):1264–1271, 1997.
URL `citeseer.ist.psu.edu/antelo97error.html`.

[6] Andrew Bateman and Paterson-Stephens Iain. *The DSP Handbook – Algorithms, Applications and Design Techniques*. Prentice Hall, 2002. ISBN 0201398516.

[7] Vanu Bose. *Virtual Radios*. PhD thesis, 1999.
URL `http://citeseer.ist.psu.edu/bose98virtual.html`.

[8] Michael S. Braasch. Multipath Effects. In Bradford W. Parkinson and James J. Spilker, Jr., editors, *Global Positioning System: Theory and Applications*. American Institute of Aeronautics and Astronautics, 1996.

[9] Shahin Charkhandeh, Dr. Mark G. Petovello, and Dr. Gerard Lachapelle. Performance Testing of a Real-Time Software-Based GPS Receiver for x86 Processor. In *ION GNSS 2006 Proceedings*, pages 2313–2320. The Institute of Navigation, 2007.

[10] Trond Danielsen. Creating a software defined GNSS receiver from free software components. Technical report, 2006.

[11] Byron Edde. *Radar: Principles, Technology, Applications*. Prentice Hall, 1993.

[12] Börje Forssell. *Radionavigation Systems*. Kompendieforlaget, 2003.

[13] Fredric J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1979.

[14] Simon Haykin. *Communication Systems*. John Wiley & Sons, 2001.

[15] Gregory W. Heckler and James L. Garrison. SIMD correlator library for GNSS software receivers. *GPS Solutions*, 10(4):269, 2006.

[16] Tim Hentschel and Gerhard Gettweis. The Digital Front End - Bridge Between RF and Baseband Processing. In Walter Tuttlebee, editor, *Software defined radio: Enabling Technologies*. John Wiley & sons Ltd.

[17] Todd E. Humphreys, Mark L. Psiaki, Paul M. Kintner jr., and Brent M. Ledvina. GNSS Receiver Implementation on a DPS: Status, Challenges, and Prospects. In *ION GNSS 2006 Proceedings*, pages 2370–2382. The Institute of Navigation, 2007.

[18] Stèphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1999.

[19] *ZHL-1217MLN – Coaxial Low-Noise Amplifier*. Mini-Circuits.
URL `http://www.minicircuits.com/pdfs/ZHL-1217MLN.pdf`.

[20] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. 1997.
URL `http://www.dspguide.com/`.

[21] James J. Spilker, Jr. GPS Signal Structure and Theoretical Performance. In Bradford W. Parkinson and James J. Spilker, Jr., editors, *Global Positioning System: Theory and Applications*. American Institute of Aeronautics and Astronautics, 1996.

[22] *STR Series Multichannel Satellite Navigator Simulator Reference Manual*. Spirent Communications, 2001.

[23] G.D. Troxel, E. Blossom, S. Boswell, A. Caro, I. Castineyra, A. Colvin, T. Dreier, J.B. Evans, N. Goffee, K.Z. Haigh, T. Hussain, V. Kawadia, D. Lapsley, C. Livadas, A. Medina, J. Mikkelson, G.J. Minden, R. Morris, C. Partridge, V. Raghunathan, R. Ramanathan, C. Santivanez, T. Schmid, D. Sumorok, M. Srivastava, R.S. Vincent, D. Wiggins, A.M. Wyglinski, and S Zahedi. Adaptive Dynamic Radio Open-source Intelligent Team (ADROIT): Cognitively-controlled Collaboration among SDR Nodes. 2006.

[24] James Bao-yen Tsui. *Fundamentals of global positioning system receivers : a software approach*. John Wiley & Sons, 2006.

[25] Guido van Rossum. *Python Library Reference*, 2.5 edition.
URL `http://docs.python.org/lib/lib.html`.

[26] Jouko Vankka. *Digital Synthesizers and Transmitters for Software Radio*. Springer, 2005.