SOFTWARE DEFINED RADIO
FOR GPS

Author: Marc Solé Gaset
Advisor: Carles Fernández Prades

Centre Tecnològic de Telecomunicacions de Catalunya (CTTC)
Escola Tècnica Superior Enginyeria Superior de Telecomunicacions de Barcelona (ETSETB).
Universitat Politècnica de Catalunya (UPC)

July 2009
Abstract

This project describes the implementation of a software defined radio (SDR) Global Navigation Satellite System (GNSS) receiver completely real-time. The receiver is based on Global Positioning System (GPS) L1 C/A receiver.

Also, the project describes the implementation and shows the improvement of a Double Delta Correlator (ΔΔDLL) respect a typical Delay Locked Loop (DLL) of a real-time software defined Global Positioning System (GPS) receiver on a multipath environment. A theoretical analysis of the DLL and ΔΔDLL has been done, and compared their performance over a simulated multipath scenario at radiofrequency. Then the same simulation but in a experimental mode is done with a channel emulator.

The double delta correlator exploits the Single Input Multiple Data (SIMD) instructions to achieve real-time performance. The GPS software receiver uses open source libraries including GNURadio, and is capable of running in real-time on standard PC.

In another hand, the receiver has been equipped with toolkit positioning system to compute the user position by different algorithms. This is done thanks to the open source GPS toolkit (GPStk). To be able to connect the receiver with the GPStk, a module with a Receiver Independent Exchange Format (RINEX) output has been added.
Resum

Aquest projecte descriu la implementació d’un receptor software definit per radio (SDR) per al Sistema de Navegació Global per Satèl·lit (GNSS) totalment en temps real. El receptor està basat en el sistema de posicionament GPS L1 C/A.

A més a més, descriu la implementació i mostra la millora d’un Correlador Doble Delta (ΔΔDLL) respecte a un correlador comú Delay Lock Loop (DLL) en un entorn amb multicamí. Primerament, es realitza un anàlisi teòric del correlador DLL i del correlador ΔΔDLL, i es comparen les seves prestacions en un escenari amb multicamí simulat en radiofreqüència. A continuació es duu a terme el mateix estudi però en un àmbit experimental amb el receptor i un emulador de canal.

El correlador doble delta explota les instruccions del Single Input Multiple Data (SIMD) per aconseguir una implementació en temps real. El receptor GPS software utilitza llibreries de distribució lliure, incloent GNURadio, i és capaç d’executar-se en temps real en un PC estàndard.

Per una altra banda, s’ha equipat el receptor amb un seguit d’eines per al càlcul de la posició de l’usuari amb diferents algorismes de posicionament, gràcies a les llibreries de programari lliure GPStk. Per tal de poder adaptar el receptor a les llibreries GPStk s’ha afegit al receptor un mòdul extractor de fitxers RINEX.
Acknowledgements

I would like to thank the CTTC (Centre Tecnològic de Comunicacions de Catalunya) for letting me make my final master thesis and for giving me all the facilities and resources necessaries for its realization. I would also like to thank Javier Arribas for their assistance throughout the project. And especially I would like to thank Carles Fernández a great mentor and best person for his help and teachings during this time and to let me work at CTTC.

Last but not least, thank you very much to all my family and the people who loves me. Thank you very much for allowing me to study and give me the courage enough to overcome the bad times.

Wholeheartedly thank you very much!
# Contents

Abstract ii 

Resum iii 

Acknowledgements iv 

Contents viii 

List of Figures xii 

List of Tables xiii 

Motivation 1 

1 Introduction and State-of-the-art 3 

1.1 Basic knowledges for GPS (*Global Positioning System*) 3 

1.1.1 An intuitive example of GPS positioning 3 

1.2 GNSS Signal Structure 6 

1.2.1 GPS Spreading Codes 7 

1.2.2 Navigation Data messages 10 

1.2.3 GPS Message Structure 13 

1.3 GNU Radio 19
3.5.1 SIMD ................................................................. 77
3.5.2 SIMD DLL correlator implementation in GPS-SDR ............... 79
3.5.3 SIMD ∆ΔDLL correlator implementation ........................... 91

4 The GPS Toolkit: GPSTk .............................. 95
  4.1 The GPSTk core library and applications .......................... 95
  4.2 The GPSTk in GPS-SDR ............................................. 97
    4.2.1 GPSTk–GPS-SDR Connection .................................. 97
    4.2.2 GPSTk utilities and GPS-SDR ................................ 108

5 Test and Simulations .................................. 113
  5.1 ∆ΔDLL Simulations ................................................. 113
    5.1.1 Testbed .......................................................... 113
    5.1.2 Simulations ...................................................... 115
    5.1.3 Results .......................................................... 117
  5.2 GPSTk Simulations .................................................. 119
    5.2.1 Testbed .......................................................... 119
    5.2.2 Simulations ...................................................... 121
    5.2.3 Static GPSTk Results ........................................... 122
    5.2.4 Dynamic GPSTk Results ....................................... 128
    5.2.5 GPSTk Algorithms Positioning Accuracy ....................... 132

6 Conclusions ................................................. 135

Bibliography ................................................. 137

Appendix A Software Installation and Execution ......................... 139
  A.1 GNU Radio Software v3.1.3 ...................................... 139
    A.1.1 GNU Radio Installation steps ................................ 140
List of Figures

1.1 Graphic 2D Positioning inspired in [MHP01] ........................................ 4
1.2 Clock error effect inspired in [MHP01] .................................................. 5
1.3 DOP effect in positioning inspired in [MHP01] ........................................ 6
1.4 GNSS Spectrum .............................................................................. 6
1.5 Structure GPS Message ..................................................................... 11
1.6 Universal Software Radio Peripheral Externally ................................. 21
1.7 Universal Software Radio Peripheral Internally ................................. 21

2.1 Thread Schedule ............................................................................ 28
2.2 Template for a threaded object ........................................................ 29
2.3 GPS-SDR Pipes Schedule ................................................................. 31
2.4 GPS-SDR Directory Structure .......................................................... 32
2.5 Content of the main directory ............................................................ 33
2.6 Acquisition thread main ................................................................. 36
2.7 Sample application of correction parameters .................................... 45

3.1 Basic code tracking loop block diagram .......................................... 58
3.2 Code tracking example .................................................................. 59
3.3 DLL block diagram with six correlators .......................................... 60
3.4 Comparison between the common DLL discriminator responses .... 62
3.5 void Correlator::Accum GPS-SDR implementation ......................... 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6 Early, Late and Prompt Power Calculation</td>
<td>63</td>
</tr>
<tr>
<td>3.7 DLL Code Correction</td>
<td>64</td>
</tr>
<tr>
<td>3.8 Direct Path and Multipath (Reflected Path) Signals</td>
<td>65</td>
</tr>
<tr>
<td>3.9 Direct Path, Multipath (in phase) and Resulting Correlation Functions with finite BW</td>
<td>66</td>
</tr>
<tr>
<td>3.10 Direct Path, Multipath (out of phase) and Resulting Correlation Functions with finite BW</td>
<td>67</td>
</tr>
<tr>
<td>3.11 Tracking Error Due to Multipath</td>
<td>68</td>
</tr>
<tr>
<td>3.12 Direct Path, Multipath (in phase) and Resulting Correlation Functions</td>
<td>69</td>
</tr>
<tr>
<td>3.13 Direct Path, Multipath (out of phase) and Resulting Correlation Functions</td>
<td>70</td>
</tr>
<tr>
<td>3.14 Early Late Technique. Ideal Case</td>
<td>71</td>
</tr>
<tr>
<td>3.15 Double Delta Correlator Layout</td>
<td>72</td>
</tr>
<tr>
<td>3.16 2MHz Bandwidth Correlator Filter used in theoretical simulation</td>
<td>75</td>
</tr>
<tr>
<td>3.17 Multipath error envelope. Theoretical comparison between DLL and ΔΔDLL correlators (δ = 1)</td>
<td>76</td>
</tr>
<tr>
<td>3.18 Single Instruction, Multiple Data concept</td>
<td>78</td>
</tr>
<tr>
<td>3.19 SIMD extensions, register layouts, and data types</td>
<td>80</td>
</tr>
<tr>
<td>3.20 General System and Application Programming Registers</td>
<td>81</td>
</tr>
<tr>
<td>3.21 SSE Execution Environment</td>
<td>83</td>
</tr>
<tr>
<td>4.1 GPS-SDR RINEX Observation File</td>
<td>98</td>
</tr>
<tr>
<td>4.2 GPS-SDR RINEX Navigation File</td>
<td>99</td>
</tr>
<tr>
<td>4.3 Representation of a GNSS propagation model (inspired in [Sal])</td>
<td>109</td>
</tr>
<tr>
<td>4.4 GPSTk Positioning Implementation</td>
<td>110</td>
</tr>
<tr>
<td>4.5 GPSTk DGPS Positioning Implementation</td>
<td>111</td>
</tr>
<tr>
<td>5.1 Diagram connection for multipath propagation simulation</td>
<td>114</td>
</tr>
<tr>
<td>5.2 Laboratory scenario for multipath propagation simulation</td>
<td>115</td>
</tr>
</tbody>
</table>
5.3 Multipath channel impulse response. ........................................... 116
5.4 Real pseudodistance error comparison (DLL vs ∆DLL). ............... 118
5.5 Theoretical pseudodistance error comparison (DLL vs ∆DLL). ........ 118
5.6 Hardware GPStk simulations components. ................................. 120
5.7 Hardware detailed diagram. ...................................................... 121
5.8 Simplified block diagram for the software GPS receiver. ............... 121
5.9 GPStk Positioning Implementation. .......................................... 122
5.10 GPS Receiver Real Position (yellow pin) vs LMS Position (red line). . 123
5.11 LMS GPS-SDR Position (red) versus GPStk WMS Position (blue). . 124
5.12 LMS GPS-SDR Position (red) versus GPStk kalman’s filter Position (white). . 125
5.13 LMS GPS-SDR Position (red) versus GPStk DGPS LMS Position (blue). . 126
5.14 Kalman GPStk Position (white) versus kalman GPStk DGPS Position (black). . 126
5.15 LMS GPS-SDR Position (red) versus kalman GPStk DGPS Position (black). . 127
5.16 GPS-SDR Car Assembling. ...................................................... 128
5.17 LMS GPS-SDR Position (red) versus Real GPS-SDR Receiver Route. . 129
5.18 LMS GPS-SDR Position (red) versus WMS GPStk Position (blue). . 129
5.19 LMS GPS-SDR Position (red) versus kalman GPStk Position (white). . 130
5.20 LMS GPS-SDR Position (red) versus LMS GPStk DGPS Position (blue). . 130
5.21 WMS GPStk Position (blue) versus WMS GPStk DGPS Position (purple). . 131
5.22 kalman GPStk Position (white) versus kalman GPStk DGPS Position (black). . 131
5.23 LMS GPS-SDR Position (red) versus kalman GPStk DGPS Position (black). . 132

B.1 Universal Software Radio Peripheral diagram: receiver path (above) and transmission path (below). ........................................... 152
B.2 Digital down-conversion and decimation .................................... 153
B.3 Digital up-conversion stage .................................................... 154
B.4 Each AD9862 has two ADCs, two DACs and their corresponding Programmable Gain Amplifiers (PGA) ..................................................... 156
B.5 AD9862 Transmit section block diagram .................................................. 157
B.6 AD9862 Receive section block diagram .................................................. 157
B.7 Physical aspect of the BasicRX board .................................................... 158
B.8 Front end schematic of the BasicRX board ............................................. 159
B.9 Physical aspect of the BasicTX board .................................................... 160
B.10 Physical aspect and schematic of the LFRX board ................................. 161
B.11 Front end schematic of the BasicTX board ............................................. 162
B.12 Physical aspect and schematic of the LFTX board ................................ 163
B.13 DBSRX Physical aspect (left) and diagram of the Direct Conversion tuner MAX2118 (right) .......................................................... 164
B.14 DBSRX Block diagram ........................................................................ 165

D.1 Probability function for WMS and LMS algorithms (Horizontal 2D) .......... 188
D.2 Probability function for WMS and LMS algorithms (Horizontal 2D) .......... 188
D.3 Probability function for WMS and LMS algorithms (Horizontal 2D) .......... 189
D.4 Probability function for WMS and LMS algorithms (Horizontal 2D) .......... 189
D.5 Probability function for WMS and LMS algorithms (Vertical) ................. 192
D.6 Probability function for WMS and LMS algorithms (Vertical) ................. 193
D.7 Probability function for WMS and LMS algorithms (Vertical) ................. 193
D.8 Probability function for WMS and LMS algorithms (Vertical) ................. 194
List of Tables

1.1 L2 signal configuration .................................................. 12
1.2 Subframe 1 Parameters .................................................. 13
1.3 Satellite Vehicle Health Information ................................. 14
1.4 Subframe 2 Information .................................................. 15
1.5 Subframe 3 Information .................................................. 16
1.6 GPS to UTC Parameters ................................................. 17
1.7 Earth equator and orbit plane parameters .......................... 18

3.1 Various types of delay lock loop discriminators and a description of them . . . 61
3.2 EPL sse_prn_new_accum storage ..................................... 90
3.3 EPL sse_prn_new_accum_dd storage .................................. 94

5.1 Simulated Signal Parameters ......................................... 115
5.2 Simulated Signal Parameters ......................................... 123
5.3 Simulated Signal Parameters ......................................... 128
5.4 Horizontal Position Accuracy (2D) ................................. 133
5.5 Vertical Position Accuracy ............................................ 134

D.1 Simulated Signal Parameters ......................................... 187
D.2 Horizontal Position Accuracy (2D) ................................. 190
D.3 Simulated Signal Parameters ......................................... 192
Motivation

The GPS (Global Positioning System) was invented by the US Army at the 70’s. As almost all the technologies it was developed for war purposes like guide missiles, search military targets and troops, etc. Through a satellite constellation, a GNSS receiver is able to set its position in four dimensions (longitude, latitude, altitude and time). This has caused the appearance of lots of civil and military applications.

In recent years, there is a clear trend that has emerged in designing GNSS receivers: the digitization process (implemented by an Analog-to-Digital Converter, ADC) is placed as close to the receivers antenna as possible, creating systems that work at increasingly higher frequencies and wider bandwidths. This is the so-called Software Defined Radio (SDR) paradigm, which tries to replace most of the hardware components of traditional RF front-ends by software-defined procedures running on programmable devices. This allows the full exploitation of powerful digital signal processing techniques, thus providing a high degree of flexibility, reconfigurability and robustness. It also constitutes a prime platform for research, as long as it is a rapid prototyping tool for assessing the overall system performance when a certain module or functionality is changed or added.

The application of SDR techniques represents an evolutionary step in the development of modern GNSS receivers. Software radios perform all digital signal processing via a programmable microprocessor such as FPGA, DSP or even a PC, as opposed to using an Application-Specific Integrated Circuit (ASIC). This separates analog signal conditioning in hardware from digital signal processing in software and results in significant advantages, such as soft transition to signal modernization (particularly important in the case of GNSS, which in the near future will have a number of additional signals that can be utilized for positioning, navigation, and timing), demand for weak signal processing (in-door navigation), multi-mode operation, and multi-sensor integration. Users of software receivers will only need a software upgrade to allow for the inclusion of the new signal processing.

Through this work we’re going to study about these Free Software GPS Receivers. For our purpose we are bound to use the GPS-SDR (Global Positioning System Software Defined
Radio) and essential hardware part as a receiver antenna and the GNURADIO USRP (Universal Software Radio Peripheral).

The main part of the project is to improve the GPS-SDR results by adding some applications. One of our a priori objectives is to obtain a better multipath response using a Double Delta Correlator in the tracking process. Another goal of our work is to improve the positioning algorithm, for it the GPStk will be used. One of the things used to attain these goal is the RINEX file format which is a useful tool to post process the GPS data.

Finally, to check our achievements we’re going to compare our results with the GPS-SDR original software ones. This are going to be done with real GPS data, varied environments and dynamic an static situations. Since it’s possible to put the GPS-SDR and the GNURADIO into the car.
Chapter 1

Introduction and State-of-the-art

1.1 Basic knowledges for GPS (Global Positioning System)

The GPS system is composed for about 30 satellites which at least 24 are always working. These satellites are orbiting more or less at 20,000 km over the Earth surface. Each satellite transmit in a continuous mode an electromagnetic signal which includes the PRN code and the navigation message. The navigation message indicates the satellite position and its time correction. The GPS signal uses for its modulation the spread spectrum technique. A GPS receiver receives this signal, process it, and get as a result the receiver position.

To get the position, the receiver has to solve a geometric problem. To get a well known position we need to know the three dimensional position and the time when we are at these coordinates. So in fact, to know the user position the receiver needs to know four variable, for these reason a GPS receiver needs at least 4 satellites to achieve the user position.

Nowadays exists loot of algorithms and techniques to compute the position, and with the adequate ones a GPS receiver is capable to compute the position with an accuracy of 10 meters or less [Par96], [EDK05].

1.1.1 An intuitive example of GPS positioning

As it has been said previously the GPS has to solve a geometric problem to solve the user position. To solve it the receiver uses what is called pseudorange. The pseudorange is the distance between the satellite and the GPS receiver. The way to know the pseudodistance is compute it through delay which suffers the electromagnetic wave transmitted by the satellite traveling from
the transmitter and the receiver. So the pseudodistance is computed multiplying this time by the propagation speed (velocity speed).

Following an example similar to the example of [MHP01], it will be seen more clearly the idea of the GPS positioning. The example is for a two-dimensional case, but the concept is the same for three-dimensional cases.

Let’s suppose we are in the middle of the desert and we want to know our position. Also, let’s suppose that at a certain distance there is a tower, which at a certain and known position. The tower is emitting an acoustic signal at a regular interval of 30 seconds starting at a certain time of the day. Let’s suppose that this acoustic signal is high enough to be heard from some kilometers of distance of the tower.

As it has said before, we are in the middle of the desert with a clock perfectly synchronized with the tower clock. As we have the same time as the tower transmitter, when we heard the acoustic signal we will be capable to measure the propagation time of the acoustic signal. If we assume for example, a propagation time of 50 seconds, we will compute the pseudodistance between us and the tower multiplying this time by the propagation speed (sound speed, 340 m/s). So in this case, \( d = 50 \text{ s} \times 340 \text{ m/s} = 17000 \text{ meters} \). So we could be at anywhere in the middle of the desert at a distance of 17 km of the tower. As it’s clear to see with a unique tower it’s no enough to know our position, so we could see at any point in a radius determined by the propagation time. If we add a second tower and we compute the pseudodistance and we draw it, we will see that there are two possible points for our position. So we have to add another tower.

As it could be seen at Fig. 1.1 with three towers we only could be at one point in a certain time; and we have been able to compute our position solving the geometric problem.

The idea of GPS positioning is the same that in the example but instead of using the sound speed the GPS system uses the velocity speed to compute the pseudodistance.

![Figure 1.1: Graphic 2D Positioning inspired in [MHP01]](image-url)
Chapter 1. Introduction and State-of-the-art

Sources of ambiguity

At the previous example it has been assumed that the tower coordinates are known in advance; and that the tower and the user clocks are perfectly synchronized. But in the GPS system these is not absolutely true; the satellites coordinates are transmitted inside the navigation message and the receiver computes it through the keplerian elements from the navigation message. Due to this some error could be produced in the calculation of the satellite coordinates. Other assumption done in the previous example, is that the receiver clock and the transmitter clock are perfectly aligned, but in GPS system it is not true. In fact, exist a some differences between both clocks. This synchronism error produces a mistaken error, and transform the final user point in a region of ambiguity of possible points for the user position. This effect could be seen in Figure 1.2.

![Figure 1.2: Clock error effect inspired in [MHP01]](image)

To solve the differences between group of clocks the GPS satellites are equipped with special clocks. These clocks are called atomic oscillators and they have stabilities of about $10^{-13}$ (For more information see [MHP01] chapter 2).

Other source error in GPS system is which is called DOP (Dilution of Precision). The DOP is produced depending on the position of the satellites and depending how the viewer sees them. DOP make that the possible positioning region changes their shape and size depending on the relative position between the satellites and the user. For more information about this effect (Dilution Of Precision –DOP–) see [MHP01] chapter 2. An example of the DOP effect is shown at Figure 1.3, where the area is smallest in case (b) for which the measurements are coming from orthogonal directions. This represents the optimal geometry for this two-dimensional space. The uncertainty in case (a) potentially has the greatest area, since it will increase as the angle between the satellites decreases.
Figure 1.3: DOP effect in positioning inspired in [MHP01]

1.2 GNSS Signal Structure

The GPS signals are transmitted on two radio frequencies in the UHF band. The UHF band covers the frequency band from 500 MHz to 3 GHz. These frequencies are referred to as L1 and L2 and are derived from a common frequency, $f_0=10.23$ MHz:
Also a third link L5 is envisaged. The availability of a third frequency for the general purpose user allows for higher precision and faster acquisition for kinematic applications than single or double band receivers allow. L1 and L2 which are two right-hand circularly polarized (RHCP) radio frequency links are the interface between the GPS Space Segment and the GPS navigation User Segment. The L-band links are modulated by one or more bit trains, each one formed by the Modulo-2 addition of a pseudo-random noise (PRN) ranging code and the downlink system data, usually referred to as navigation or NAV data.

The GPS signals are composed of the following three parts:

- The carrier which it has been explained above.
- Navigation data that contain information regarding satellite orbits.
- The spreading sequence.

### 1.2.1 GPS Spreading Codes

The pseudo-random noise (PRN) codes selected for GPS, referred to as Gold codes and firstly introduced in [Gol67], were chosen by their desirable properties of cross-correlation and easiness of implementation. A GPS receiver must perform a correlation operation in order to extract the signal timing and recover the data of a set of satellites. Thus, if the receiver has M satellites in view, it must separate each satellite signal from others and perform timing and data recovering. This separation is performed by a set of parallel correlators, each one devoted to a single satellite; for one of these correlators, a signal belonging to a certain satellite is the desired signal and all the other M-1 signals are seen as interferer’s. This needing of systems self-interference rejection demands spreading codes with low (and upper-bounded) cross-correlation. The Gold codes are linear combinations of two m-sequences (also called maximal length sequences) with different offset in a Galois field. All pairs of m-sequences do not yield Gold codes, and those which yield Gold codes are called preferred pairs. Gold codes have three-valued autocorrelation and cross-correlation function with values \{-1, t(m), t(m) - 2\}, where
The generation of Gold codes is very simple: they consist of the modulo-2 sum of two linear feedback shift registers (LFSR) outputs, each one defined by a feedback polynomial. The current and planned GPS spreading codes are:

- **C/A code.** The PRN C/A code for the Space Vehicle (SV) number \( i \) is a Gold code, \( C_{C/A,i(t)} \), of 1 millisecond in length at a chipping rate of 1023 Kbps. These sequences are linear patterns generated by the modulo-2 addition of two subsequences, \( G1 \) and \( G2i \), each of which being 1023 chip long linear patterns. The \( G2i \) sequence is formed by effectively delaying the \( G2 \) sequence by an integer number of chips ranging from 5 to 950. The \( G1 \) and \( G2 \) sequences are generated by 10-stage LFSRs having the following polynomials as referred to in the shift register input:

\[
G1 : 1 + x^3 + x^{10}, \quad \text{and} \\
G2 : 1 + x^2 + x^3 + x^6 + x^8 + x^9 + x^{10}
\]

- **P code.** The PRN P code for the SV number \( i \) is a sequence \( C_{P,i(t)} \) of 7 days in length at a chipping rate of 10.23 Mbps. Again, sequences are composed by the modulo-2 sum (also denoted as exclusive-or or \( \oplus \) operation) of two subsequences referred to as \( X1 \) and \( X2i \), with length 15,345,000 chips and 15,345,037 chips respectively. \( X1 \) itself is generated by the modulo-2 sum of the output of two 12-stage registers (\( X1A \) and \( X1B \)) short cycled to 4092 and 4093 chips respectively. When the \( X1A \) short cycles are counted to 3750, the \( X1 \) epoch is generated, occurring every 1.5 seconds. The \( X2i \) sequence is an \( X2 \) sequence (also generated by two shift registers, \( X2A \) and \( X2B \)) selectively delayed by 1 to 37 chips, thereby producing a set of 37 mutually exclusive sequences of 7 days in length. The polynomials for \( X1A \) and \( X1B \) codes, as referenced to the shift register input, are

\[
X1A : 1 + x^6 + x^8 + x^{11} + x^{12}, \quad \text{and} \\
X1B : 1 + x + x^2 + x^5 + x^8 + x^9 + x^{10} + x^{11} + x^{12}
\]

while the corresponding for the \( X2A \) and \( X2B \) codes are

\[
X2A : 1 + x + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12}, \quad \text{and} \\
X2B : 1 + x^2 + x^3 + x^4 + x^8 + x^9 + x^{12}.
\]
• **Y code.** Code P, although its length, is publicly available, and this includes potential jammers or spoofers. A spoofer generates a signal that mimics the GPS signal and attempts to cause the receiver to track a wrong signal. For this reason, GPS has the possibility to encrypt the public P code by a secure Y code, available only to authorized U.S. government users. Its main purpose is to assure that an eventual opponent cannot spoof the Y code signal generating a Y code replica.

• **L2 CM code.** The PRN L2 civil moderate (CM) code for SV number $i$ is a ranging code $C_{CM,i}(t)$ which is 20 milliseconds in length at a chipping rate of 511.5 Kbps. It constitutes a linear pattern which is short cycled every count of 10, 230 chips by resetting with a specified initial state. This sequences will be used from the deployment of IIR-M, IIF and subsequent blocks of SVs.

• **L2 CL code.** The PRN L2 civil long (CL) code for SV number $i$ is a ranging code $C_{CL,i}(t)$ which is 1.5 seconds in length at a chipping rate of 511.5 Kbps. The $C_{CL,i}(t)$ sequence is a linear pattern which is generated using the same code generator polynomial as the one used for $C_{M,i}(t)$, but short cycled every count of 767, 250 chips by resetting with a specified initial state.

• **L5 codes.** The PRN in-phase code (denoted as the I5-code) and the quadrature-phase code (denoted as the Q5-code) for SV number $i$ are ranging codes $C_{I5,i}(t)$ and $C_{Q5,i}(t)$ of 1 millisecond in length at a chipping rate of 10.23 Mbps. Each code is a modulo-2 sum of two subsequences $XA$ and $XBl_i$ or $XBo_i$, which lengths are 8,190 and 8,191 respectively, that restart to generate the 10,230 chip code. The $XBl_i$ is selectively delayed allowing the generation of 74 different code sequences (37 I5 and 37 Q5). Of these, 32 pairs are designated for use by SVs while the remaining 5 pairs are reserved. The polynomials for $XA$ and $XBl_i$ or $XBo_i$ codes, as referenced to the shift register input, are

$$XA : 1 + x^{9} + x^{10} + x^{12} + x^{13}, \text{ and}$$

$$XBl_i \text{ or } XBo_i : 1 + x + x^{3} + x^{4} + x^{6} + x^{7} + x^{8} + x^{12} + x^{13}.$$  

• **Non-standard codes.** The so-called NSC, NSCM, NSCL, NSI5, NSQ5 and NSY codes are intentionally incorrect versions of the codes intended to protect the users from receiving and utilizing anomalous NAV signals as a result of malfunction in the SV’s reference frequency generation system.

• **M code.** The M (military) code signal is designed to provide better jamming resistance than the Y code signal, enabling transmission at much higher power without interference with C/A code or Y code receivers [BCB00], and it is intended to progressively replace the military Y code providing better security in terms of exclusivity, authentication and confidentiality, besides more robust acquisition. It consists in a binary offset carrier signal with subcarrier frequency 10.23 MHz and spreading code rate of 5.115 Mbps, denoted as
BOC(10.23, 5.115) modulation, or BOC(10,5) for short. The spreading code transitions are aligned with transitions of the square wave subcarrier. Although the M-code signal will be implemented on some Block IIF satellites, that are planned for launch between 2005 and 2010, a fully operational constellation of 18 M-code satellites will not come into operation until 2016.

1.2.2 Navigation Data messages

As said above, these PRN are modulated by a BPSK signal which contains data in a much smaller rate than the codes. The navigation data provide the receiver with the information needed to perform the positioning computation. Usually referred to as NAV data or D(t), the message is structured in a 1500 bit long frame composed of five 300 bit long subframes. Subframes 4 and 5 are subcommutated 25 times each, giving the total of 25 full frames to transmit the complete data message, which is 375,000 bits long. The message is transmitted at 50 bps, and thus the subframes 1, 2 and 3 are repeated each 30 seconds, while the whole content of subframes 4 and 5 takes 12.5 minutes to be downloaded.

The first subframe contains, in addition to other system parameters, the information needed to perform clock correction: a third degree polynomial allows the determination of the SV PRN code phase offset referenced to the phase center of the antennas with respect to the GPS system time at data transmission. Other parameters of interest in this subframe are the group delay between L1 P(Y) and L2 P(Y) and ionospheric corrections.

Subframes 2 and 3 contains a set of parameters (referred as ephemeris) describing the orbit of the transmitting satellite. Among them, the Keplerian orbital elements, readily \( \sqrt{a} \) the square root of the ellipse’s semimajor axis; \( e \) its eccentricity, which is defined as \( e = \sqrt{\frac{a^2 - b^2}{a^2}} \), being \( b \) the semiminor axis; \( i_0 \) the inclination angle at reference time; \( \Omega_o \) the longitude of the ascending node of orbit plane at weekly epoch; \( \omega \) the argument of perigee and \( M_o \) the mean anomaly at reference time. Other parameters are intended to correct the eventual orbital perturbations: \( \Delta n \) the mean motion difference from computed value; \( \dot{\Omega} \) the rate of right ascension; and six more parameters indicating the sine and cosine harmonic correction terms to the argument of latitude, orbit radius and angle of inclination. The reference time ephemeris, \( t_{ref} \) is also provided. User algorithms for the computation of satellite position and clock correction can be found in [Cor03]. Both subframe 4 and 5 are subcommutated 25 times each; the 25 versions of these subframes are referred to as pages 1 through 25 of each subframe. The information contained herein is known as almanac, and consists in the ephemeris of other satellites, their clock correction factor, ionospheric and Universal Time Coordinated (UTC) data, flags about the health of the SVs, indication of the Anti-Spoofing (AS) status and other system information.
Recently, some modifications of the data message has been proposed. In Block IIR-M SVs, the NAV bit train $D(t)$ can be rate $\frac{1}{2}$ encoded with an encoding constraint length of 7, giving a 25 bps rate resulting in 50 sps. When this encoding is activated, the message is denoted as $D(t)$. Moreover, starting from Block IIF SVs (overview in [SCF99]), another message format has been defined: the GPS L2 CNAV data structure denoted as $D_c(t)$. This data stream includes the same data as the NAV data $D(t)$ but in an entirely different format: it utilizes 300 bit long messages, each one containing a Cyclic Redundancy Check (CRC) parity block of 24 bits. In addition to this protection, the bit train is Forward Error Correction (FEC) encoded by a rate $\frac{1}{2}$ convolutional code, thus offering a rate of 25 bps and 50 sps. The ultimate navigation data modulation is envisaged for its use in the L5 band: $D_5(t)$. While the message structure is basically the same as CNAV, in $D_5(t)$ the bit train is rate $\frac{1}{2}$ convolutional encoded with a FEC code with constraint length 7, at a symbol rate of 100 sps.

- **GPS L5**

The L5 link will be only available on Block IIF SVs, and the planned future Block III. Defined in [Cor02], it is a 24 MHz band centered at $f_{L5} = 115 \times 10.23 = 1176.45$ MHz consisting of two carrier components in phase quadrature. Neglecting multiplicative constants, the baseband equivalent referred to $f_{L5}$ is splitted into the Inphase and Quadrature components:

$$s_{L5}(t) = C_{L5}(t) \oplus D_5(t) \oplus nh_{10}(t) + jC_{Q5}(t) \oplus nh_{20}(t).$$
As can be observed, the I5 component contains a synchronization sequence \( nh_{10} = 0000110101 \), a 10 bit Neuman-Hoffman code that modules each 100 symbols of \( D_5(t) \), and the Q5 component has another synchronization sequence \( nh_{20} = 0000010011010101110 \) at 20 kbps. A description of the characteristics and potentialities of the L5 signal can be found in [AJVD00].

**GPS L2 Civil Signal (L2CS)**

The L2 link is centered at \( f_{L2} = 1227.6 \) MHz, frequency which is coherently derived from a common frequency source to L1 within the SV: notice that \( f_{L1} = 154 \times 10.23 \) MHz and \( f_{L2} = 120 \times 10.23 \) MHz. The Control Segment is able to select different signal configurations which are summarized in table 1.1.

<table>
<thead>
<tr>
<th>SV Blocks</th>
<th>L2 In-Phase</th>
<th>L2 Quadrature-Phase (L2CS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block II/IIA/IIR</td>
<td>( C_{PYY}(t) \oplus D(t) ), ( C_{PYY}(t) ), or ( C_{C/A}(t) \oplus D(t) )</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Block IIR-M</td>
<td>( C_{PYY}(t) \oplus D(t) ), or ( C_{PYY}(t) )</td>
<td>( C_{CM}(t) \oplus D(t) ) time multiplexed with ( C_{CL}(t) ), ( C_{CM}(t) \oplus D'(t) ) time multiplexed with ( C_{CL}(t) ), ( C_{C/A}(t) \oplus D(t) ), or ( C_{C/A}(t) )</td>
</tr>
<tr>
<td>Block IIF</td>
<td>( C_{PYY}(t) \oplus D_1(t) ), or ( C_{PYY}(t) )</td>
<td>( C_{CM}(t) \oplus D_1(t) ) time multiplexed with ( C_{CL}(t) ), ( C_{C/A}(t) \oplus D(t) ), or ( C_{C/A}(t) )</td>
</tr>
</tbody>
</table>

Table 1.1: L2 signal configuration

The preferred situation for the civil component, known as L2 Civil Signal or L2CS, is \( C_{CM}(t) \oplus D(t) \) chip-by-chip time multiplexed with \( C_{CL}(t) \) in case of Block IIR-M and \( C_{CM}(t) \oplus D_1(t) \) time multiplexed with \( C_{CL}(t) \) in case of Block IIF [Fon01]. This is because biphase data modulation forces the receiver to use a Costas (squaring) loop to create a second harmonic of the carrier, which can be tracked; the possibility to track \( C_{CL}(t) \) without data modulation allows the use of a simple Phase Locked Loop (PLL), thus improving 6 dB the tracking threshold with respect to the Costas loop. In addition, PLL makes whole-cycle phase measurements rather than having to use message content to resolve the radians phase ambiguity of the Costas loop.

**GPS L1**

The GPS L1 link, centered at \( f_{L1} = 1575.42 \) MHz, consists of two components in quadrature phase, and its baseband equivalent, referred to \( f_{L1} \) and neglecting multiplicative constants, can be expressed as

\[
s_{L1}(t) = C_{PYY}(t) \oplus D(t) + jC_{C/A}(t) \oplus D(t).
\]
This is the link used by the commercial receivers, in particular the quadrature component with the C/A code.

### 1.2.3 GPS Message Structure

The data contained in the first three subframes are the minimal parameters required to calculate the user position.

#### Subframe 1

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Location</th>
<th># bits</th>
<th>LSB</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week number</td>
<td>61–70</td>
<td>10</td>
<td>1</td>
<td>week</td>
</tr>
<tr>
<td>Sat. accuracy</td>
<td>73–76</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sat. health</td>
<td>77–82</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IODC</td>
<td>83–84</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Issue of data, clock</td>
<td>211–218</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{GD}$ Sat. group delay differential</td>
<td>197–204</td>
<td>8</td>
<td>$2^{-31}$</td>
<td>seconds</td>
</tr>
<tr>
<td>$t_{ce}$ Sat. clock correction</td>
<td>219–234</td>
<td>16</td>
<td>$2^4$</td>
<td>seconds</td>
</tr>
<tr>
<td>$a_{f2}$ Sat. clock correction</td>
<td>241–248</td>
<td>8</td>
<td>$2^{-55}$</td>
<td>sec/sec^2</td>
</tr>
<tr>
<td>$a_{f1}$ Sat. clock correction</td>
<td>249–264</td>
<td>16</td>
<td>$2^{-43}$</td>
<td>sec/sec</td>
</tr>
<tr>
<td>$a_{f0}$ Sat. clock correction</td>
<td>271–292</td>
<td>22</td>
<td>$2^{-31}$</td>
<td>seconds</td>
</tr>
</tbody>
</table>

Table 1.2: Subframe 1 Parameters

- **Week number** (bits 61-70): These ten bits represent the MSB of the Z counts and indicates the number of weeks from midnight on the night of January 5, 1980/morning of January 6, 1980. Users must count the rollover if it is over 1,023 weeks.

- **User range accuracy** (bits 73-76): These four bits give the predicted user range accuracy and its value N ranges from 0 to 15. The accuracy value X is:
  - If N is 6 or less, $X = 21+N/2$ (rounded-of-values $N = 1$, $X = 2.8$; $N = 3$, $X = 5.7$; $N = 5$, $X = 11.3$).
  - If N is 6 or more, but less than 15, $X = 2N - 2$.
  - $N = 15$ will indicate the absence of an accuracy prediction and will advise the user to use that satellite at the user’s risk.
• **Satellite health** (bits 77-82): These six bits represent the health indication of the transmitting satellite. The MSB (bit 77) indicates a summary of the health of the navigation data, where bit 77 equals:
  
  – 0 All navigation data are OK.
  – 1 Some or all navigation data are bad.

The five LSBs indicate the health of the signal components:

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>All signals OK.</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>Satellite is temporarily out – do not use this satellite during current pass.</td>
</tr>
<tr>
<td>1 1 1 0 1</td>
<td>Satellite will be temporarily out – use with caution.</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
<td>Spare</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>More than one combination would be required to describe anomalies.</td>
</tr>
</tbody>
</table>

All other combinations: Satellite experiencing code modulation and/or signal level transmission problem – modulation navigation data valid; however, user may experience intermittent tracking problems if satellite is acquired.

Table 1.3: Satellite Vehicle Health Information

Additional satellite health data are given in subframes 4 and 5. The data given in subframe 1 may differ from that shown in subframes 4 and/or 5 of other satellites, since the latter may be updated at a different time.

• **Issue of data, clock** (IODC) (83-84 MSB, 211-218 LSB): These 10-bit IODC data indicate the issue number of the data set and thereby provide the user with a convenient means of detecting any change in the correction parameters. The transmitted IODC will be different from any value transmitted by the satellite during the preceding seven days. The relationship between IODC and IODE (in both subframes 2 and 3) will be discussed in the next section.

• **Estimated group delay differential** $t_{GD}$ (bits 197-204): This eight-bit information is a clock correction term to account for the effect of satellite group delay differential. This parameter is in two’s complement, with the sign bit (+ or -) occupying the MSB.

• **Satellite clock correction parameters**: This subframe also contains the four additional satellite clock correction parameters: $t_{oc}$ (219-234), $a_{f0}$ (271-292), $a_{f1}$ (249-264), and $a_{f2}$ (241-248).
In subframe 1 there are some reserved data fields and their locations are 71-72; 91-114; 121-144; 151-174; 181-196. All reserved data fields support valid parity within their respective words.

**Subframe 2**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Location</th>
<th># bits</th>
<th>LSB</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>IODE</td>
<td>61-68</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_r$ Amplitude of the sine harmonic correction terms to the orbit radius</td>
<td>69–84</td>
<td>16</td>
<td>$2^{-5}$</td>
<td>meters</td>
</tr>
<tr>
<td>$\Delta n$ Mean motion difference from computed value</td>
<td>91–106</td>
<td>16</td>
<td>$2^{-4}$</td>
<td>semicircles / s</td>
</tr>
<tr>
<td>$M_0$ Mean anomaly at reference time</td>
<td>107–114; 121–144</td>
<td>32</td>
<td>$2^{-3}$</td>
<td>semicircle</td>
</tr>
<tr>
<td>$C_{\omega}$ Amplitude of the cosine harmonic correction term to the argument of latitude of argument of latitude</td>
<td>151–166</td>
<td>16</td>
<td>$2^{-20}$</td>
<td>rad</td>
</tr>
<tr>
<td>$e_s$ Eccentricity</td>
<td>167–174; 181–204</td>
<td>32</td>
<td>$2^{-39}$</td>
<td></td>
</tr>
<tr>
<td>$C_{\omega s}$ Amplitude of the sine harmonic correction term to the argument of latitude</td>
<td>211–226</td>
<td>16</td>
<td>$2^{-20}$</td>
<td>rad</td>
</tr>
<tr>
<td>$\sqrt{a}$ Square root of the semimajor axis</td>
<td>227–234; 241–264</td>
<td>32</td>
<td>$2^{10}$</td>
<td>(a^{1/2})</td>
</tr>
<tr>
<td>$t_0$ Reference time ephemeris</td>
<td>277–286</td>
<td>16</td>
<td>$2^{4}$</td>
<td>seconds</td>
</tr>
</tbody>
</table>

Table 1.4: Subframe 2 Information

- **The issue of data, ephemeris** (IODE): This parameter has 8 bits and is in both subframes 2 (61–68) and 3 (271–278). The IODE equals the 8 LSB of the IODC, which has 10 bits. The IODE provides the user with a convenient means for detecting any change in the ephemeris representation parameters. The transmitted IODE will be different from any value transmitted by the satellite during the preceding six hours. Whenever these three terms, two IODEs from subframes 2, 3 and the 8 LSBs of the IODC, do not match, a data set cutover has occurred and new data must be collected. Any change in the subframe 2 and 3 data will be accomplished in concert with a change in both IODE words. Cutovers to new data will occur only on hour boundaries except for the first data set of a new
upload. The first data set may be cut in at any time during the hour and therefore may be transmitted by the satellite for less than one hour. Additionally, the toe value for at least the first data set transmitted by a satellite after an upload will be different from that transmitted prior to the cutover.

- The rest of the ephemeris data: These are listed in tables

- Spare and reserved data fields: In subframe 2 bit 287 is reserved and bits 288-292 are spared. All spare and reserved data fields support valid parity within their respective words. Contents of spare data fields are alternating ones and zeros until they are allocated for a new function. Users are cautioned that the contents of spare data fields can change without warning.

### Subframe 3

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Location</th>
<th># bits</th>
<th>LSB</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{1u}$: Amplitude of the cosine harmonic correction term to angle of inclination</td>
<td>61–76</td>
<td>16</td>
<td>$2^{-39}$</td>
<td>rad</td>
</tr>
<tr>
<td>$\Omega$: Longitude of ascending node of orbit plane at weekly epoch</td>
<td>77–84; 91–114</td>
<td>32</td>
<td>$2^{-31}$</td>
<td>semicircles</td>
</tr>
<tr>
<td>$C_{1v}$: Amplitude of the sine harmonic</td>
<td>212–136</td>
<td>16</td>
<td>$2^{-39}$</td>
<td>rad</td>
</tr>
<tr>
<td>$i_0$: Inclination angle at reference time</td>
<td>137–144; 151–174</td>
<td>32</td>
<td>$2^{-31}$</td>
<td>semicircles</td>
</tr>
<tr>
<td>$C_{2u}$: Amplitude of the sine harmonic correction term to the orbit radius</td>
<td>181–196</td>
<td>16</td>
<td>$2^{-5}$</td>
<td>meters</td>
</tr>
<tr>
<td>$\omega$: Argument of perigee</td>
<td>197–204; 211–234</td>
<td>32</td>
<td>$2^{-31}$</td>
<td>semicircles</td>
</tr>
<tr>
<td>$\Omega$: Rate of right ascension</td>
<td>241–264</td>
<td>24</td>
<td>$2^{-43}$</td>
<td>semicircles / s</td>
</tr>
<tr>
<td>IODE</td>
<td>271–278</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i$: Rate of inclination angle</td>
<td>279–292</td>
<td>14</td>
<td>$2^{-43}$</td>
<td>semicircles / s</td>
</tr>
</tbody>
</table>

### Table 1.5: Subframe 3 Information

### Subframe 4

- Pages 2, 3, 4, 5, 7, 8, 9, and 10 contain the almanac data for satellite 25 through 32. These pages may be designated for other functions. The satellite ID of that page defines the format and content.

- Page 17 contains special messages.
Chapter 1. Introduction and State-of-the-art

- Page 18 contains ionospheric and universal coordinated time (UTC).
- Page 25 contains antispoof flag, satellite configuration for 32 satellites, and satellite health for satellites 25.32.
- Pages 1, 6, 11, 12, 16, 19, 20, 21, 22, 23, and 24 are reserved.
- Pages 13, 14, and 15 are spares.

Subframe 5

- Pages 1 to 24 contain almanac data for satellites 1 through 24.
- Page 25 contains satellite health for satellites 1 through 24, the almanac reference time, and the almanac reference week number.

Almanac data. The almanac parameters provided in subframes 4 and 5 are: $e_s$, $t_{oa}$, $\Omega$, $\sqrt{\bar{a}}$, $\Omega_e$, $\omega$, $M_0$, $a_{f0}$, and $a_{f1}$. The almanac data are much less accurate than the detailed ephemeris data of subframes 2 and 3. However, the almanac data are valid for longer periods of time and do not require frequency updates.

Translation of GPS time to UTC time. In page 18 of subframe 4 the following parameters are included:

<table>
<thead>
<tr>
<th>Parameter</th>
<th># bits</th>
<th>LSB</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>32</td>
<td>$2^{-30}$</td>
<td>seconds</td>
</tr>
<tr>
<td>$A_1$</td>
<td>24</td>
<td>$2^{-50}$</td>
<td>sec/sec</td>
</tr>
<tr>
<td>$\Delta t_{LS}$</td>
<td>8</td>
<td>1</td>
<td>seconds</td>
</tr>
<tr>
<td>$t_{ot}$</td>
<td>8</td>
<td>$2^{12}$</td>
<td>seconds</td>
</tr>
<tr>
<td>$W N_r$</td>
<td>8</td>
<td>1</td>
<td>weeks</td>
</tr>
<tr>
<td>$W N_{LSF}$</td>
<td>8</td>
<td>1</td>
<td>weeks</td>
</tr>
<tr>
<td>$D N$</td>
<td>8</td>
<td>1</td>
<td>days</td>
</tr>
<tr>
<td>$\Delta t_{LSF}$</td>
<td>8</td>
<td>1</td>
<td>seconds</td>
</tr>
</tbody>
</table>

Table 1.6: GPS to UTC Parameters

The GPS/UTC time relationship is given by
\[ t_{UTC} = (t_E - \Delta t_{UTC}) \bmod 86400 \text{ seconds} \]

where \( t_{UTC} \) is in seconds and

\[ \Delta t_{UTC} = \Delta t_{LS} + A_0 + A_1 [t_E - t_{ot} + 604,800 (W_N - W_{N_t})] \text{ seconds} \]

- \( t_E \): GPS time as estimated by the user on the basis of correcting \( t_{SV} \) for factors given in the subframe 1 clock correction discussion as well as for ionospheric and satellite (dither) effects.
- \( t_{SV} \): effective satellite pseudorange code phase time at message of transmission time.
- \( \Delta t_{LS} \): delta time due to leap seconds.
- \( A_0, A_1 \): constant and first order terms of polynomial.
- \( t_{ot} \): reference time for UTC data.
- \( W_N \): current week number (derived from subframe 1).
- \( W_{N_t} \): UTC reference week number.

The estimated GPS time \( (t_E) \) is in seconds relative to end/start of week. The reference time for UTC data \( (t_{ot}) \) is referenced to the start of that week whose number \( (W_{N_t}) \) is given in bits (227-234) of page 18 subframe 4 representing the 8 LSB of the week. The user must account for the truncated nature of the week number.
Chapter 1. Introduction and State-of-the-art

1.3 GNU Radio

GNU Radio is a free software development toolkit that provides the signal processing runtime and processing blocks to implement software radios using readily-available, low-cost external RF hardware and commodity processors. It is widely used in hobbyist, academic and commercial environments to support wireless communications research as well as to implement real-world radio systems. GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions where available. Thus, the developer is able to implement real-time, high-throughput radio systems in a simple-to-use, rapid-application-development environment. While not primarily a simulation tool, GNU Radio does support development of signal processing algorithms using pre-recorded or generated data, avoiding the need for actual RF hardware. (See [Com])

GNU Radio is an open source Software Defined Radio (SDR) project that was started about ten years ago by Eric Blossom, an electrical engineer. The main idea which is behind this project, as its founder says, was to turn all the hardware problems into software problems, that is move the complexity of a radio equipment from the hardware level to the software one, and get the software as close to the antenna as possible. Advantages of SDR

Software defined radio has some advantages that were not been possible before:

- It can be reconfigured "on-the-fly"
- It can be easily and rapidly upgraded with new software versions or enhanced features
- It is possible to talk and listen to multiple channels at the same time

1.3.1 What is the story of GNU Radio?

Blossom initiated this project because he was disappointed by the SDR projects available at that time: all of them had a proprietary nature, and he wanted to bring the free-software philosophy into the SDR world. Richard Stallman, the GNU Project founder, liked Blossoms idea and agreed to take the project under the GNU aegis.

So far, the GNU Radio project has not disappointed its affiliates and supporters. Eric Blossom, together with his development colleague Matt Ettus, have realized a project which can turn an ordinary PC into a good quality radio receiver; the only additional hardware required are a low-cost RF tuner and an analog-to-digital converter to convert the received signal into digital samples. GNU Radio is a free software development toolkit which allows to develop a custom non commercial radio receiver just combining and interconnecting appropriate software
modules, as if they were functional blocks (the package include about 100 modules, but others can be added to the initial library). Each module is able to perform a specific signal processing function (for example a mixer, a phase lock loop, a filter), with a real-time behavior and with high-throughput; for this reason, a recent PC with enough processing capability and memory shall be used. With the GNU Radio approach, the designer is a software developer who builds the radio by creating a graph (in a similar way to what happens in the graph theory) where the vertices are signal processing blocks and the edges represent the data flow between them. The signal processing blocks are normally implemented in C++, whereas the graph structure is defined in Python. GNU Radio is well known and widely used especially in academic environments and among hobbyists and radio amateurs; it is used either to implement real and working radio equipments, or just as a research project in the area of wireless communication and transmission. GNU Radio software modules support various modulations (GMSK, PSK, QAM, OFDM), error corrections codes (Reed-Solomon, Viterbi, Turbo Codes), and signal processing capabilities (filters, FFTs, equalizers, timing recovery).

GNU Radio applications are mainly written in Python; however, the critical and low-level algorithms and signal processing modules are written using the C/C++ programming language, with wide usage of floating-point specific instructions for the relevant processor. Python is primarily used to setup the flow graph, after that most of the work is done in C/C++. GNU Radio is simple to use and a radio receiver can be created in a fast and straightforward manner; moreover, the development of a signal processing algorithm can be carried out using a pre-recorded or generated data set, thus allowing the development without the need for a real RF hardware. An example of minimal hardware required to work with GNU Radio is offered by the USRP, developed by Ettus Research LLC.

1.3.2 What is USRP?

USRP, which stands for Universal Software Radio Peripheral, is a general purpose motherboard which can host a wide selection of daughterboards, each of which implements a signal processing block found in the GNU Radio software package [USR]. The original USRP is a low cost software radio device which connects to the host computer through a USB 2.0 interface, and can send up to 16 MHz of RF bandwidth in both directions. It hosts an FPGA which can be reprogrammed, 4 high-speed Analog to Digital Converters (ADCs), 4 high-speed Digital to Analog Converters DACs), and many auxiliary analog and digital I/Os.

The following pictures 1.6 and 1.7 show how the USRP looks like, externally and internally, although the internally distribution and elements could be modified depending on the USRP application. (More information and specifications of USRP board see appendix B)
Chapter 1. Introduction and State-of-the-art

Figure 1.6: Universal Software Radio Peripheral Externally

Figure 1.7: Universal Software Radio Peripheral Internally
1.3.3 GNU Radio applications

The GNU Radio package is provided with a complete HDTV transmitter and receiver, a spectrum analyzer, an oscilloscope, a multichannel receiver and a wide collection of modulators and demodulators. Other advanced projects are still in the feasibility phase or in progress:

- A system able to recording multiple stations simultaneously
- Time Division Multiple Access (TDMA) waveforms
- A passive radar system that takes advantage of broadcast TV for its signal source
- Radio astronomy
- Digital Radio Mundial (DRM)
- Software GPS
- Amateur radio transceivers

1.4 RINEX Files

All the following information has been extracted from The Receiver Independent Exchange Format Version 2.10, for more information you can see [RIN].

1.4.1 The Philosophy of RINEX

The first proposal for the “Receiver Independent Exchange Format” RINEX has been developed by the Astronomical Institute of the University of Berne for the easy exchange of the GPS data to be collected during the large European GPS campaign EUREF 89, which involved more than 60 GPS receivers of 4 different manufacturers. The governing aspect during the development was the following fact:

Most geodetic processing software for GPS data use a well-defined set of observables:

- the carrier-phase measurement at one or both carriers (actually being a measurement on the beat frequency between the received carrier of the satellite signal and a receiver-generated reference frequency).
• the pseudorange (code) measurement, equivalent to the difference of the time of reception (expressed in the time frame of the receiver) and the time of transmission (expressed in the time frame of the satellite) of a distinct satellite signal.

• the observation time being the reading of the receiver clock at the instant of validity of the carrier-phase and/or the code measurements.

Usually the software assumes that the observation time is valid for both the phase AND the code measurements, AND for all satellites observed.

Consequently all these programs do not need most of the information that is usually stored by the receivers: They need phase, code, and time in the above mentioned definitions, and some station-related information like station name, antenna height, etc.

1.4.2 General Format Description

Currently the format consists of six *ASCII RINEX files*:\(^1\):

1. Observation Data File Navigation Message File
2. Meteorological Data File
3. GLONASS Navigation Message File
4. GEO Navigation Message File
5. Satellite and Receiver Clock Date File

The format definition of the clock files has been published in 1998 in a separate document by Jim Ray and Werner Gurtner, available at the IGS Central Bureau Information System:

Each file type consists of a header section and a data section. The header section contains global information for the entire file and is placed at the beginning of the file. The header section contains header labels in columns 61-80 for each line contained in the header section. These labels are mandatory and must appear exactly as given in these descriptions and examples.

The format has been optimized for minimum space requirements independent from the number of different observation types of a specific receiver by indicating in the header the types of observations to be stored. In computer systems allowing variable record lengths the observation

---

\(^1\)You can see the header and body format and some examples at [RIN]
records may be kept as short as possible. Trailing blanks can be removed from the records. The maximum record length is 80 bytes per record.

Each Observation file and each Meteorological Data file basically contain the data from one site and one session. RINEX Version 2 also allows to include observation data from more than one site subsequently occupied by a roving receiver in rapid static or kinematic applications. Although Version 2 allows to insert header records into the data field we do not recommend to concatenate data of more than one receiver (or antenna) into the same file, even if the data do not overlap in time.

If data from more than one receiver has to be exchanged it would not be economical to include the identical satellite messages collected by the different receivers several times. Therefore the Navigation Message File from one receiver may be exchanged or a composite Navigation Message File created containing non-redundant information from several receivers in order to make the most complete file.

The format of the data records of the RINEX Version 1 Navigation Message file is identical to the former NGS exchange format.

The actual format descriptions as well as examples are given in the Tables at the end of the paper.

1.4.3 Definition of the observables

GPS observables include three fundamental quantities that need to be defined: Time, Phase, and Range.

- **TIME:**

  The time of the measurement is the receiver time of the received signals. It is identical for the phase and range measurements and is identical for all satellites observed at that epoch. It is expressed in GPS time (not Universal Time).

- **PSEUDO-RANGE:**

  The pseudo-range (PR) is the distance from the receiver antenna to the satellite antenna including receiver and satellite clock offsets (and other biases, such as atmospheric delays):

  \[ \text{PR} = \text{distance} + c \cdot (\text{receiver clock offset} - \text{satellite clock offset} + \text{other biases}) \]
so that the pseudo-range reflects the actual behavior of the receiver and satellite clocks. The pseudo-range is stored in units of meters.

- **PHASE:**

  The phase is the carrier-phase measured in whole cycles at both L1 and L2. The half-cycles measured by squaring-type receivers must be converted to whole cycles and flagged by the wavelength factor in the header section.

  The phase changes in the same sense as the range (negative doppler). The phase observations between epochs must be connected by including the integer number of cycles. The phase observations will not contain any systematic drifts from intentional offsets of the reference oscillators.

  The observables are not corrected for external effects like atmospheric refraction, satellite clock offsets, etc.

  If the receiver or the converter software adjusts the measurements using the real-time-derived receiver clock offsets \( dT(r) \), the consistency of the 3 quantities phase / pseudo-range / epoch must be maintained, i.e. the receiver clock correction should be applied to all 3 observables:

\[
\begin{align*}
\text{Time(corr)} &= \text{Time}(r) - dT(r) \\
\text{PR(corr)} &= \text{PR}(r) - dT(r) \cdot c \\
\text{Phase(corr)} &= \text{phase}(r) - dT(r) \cdot \text{freq}
\end{align*}
\]

- **DOPPLER:**

  The sign of the doppler shift as additional observable is defined as usual: Positive for approaching satellites.
Chapter 1. Introduction and State-of-the-art
Chapter 2

GPS Software Defined Radio

2.1 What is the Software Defined Radio?

Software-Defined Radio (SDR) system is a radio communication system where components that have typically been implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented using software on a personal computer or other embedded computing devices. While the concept of SDR is not new, the rapidly evolving capabilities of digital electronics are making practical many processes that were once only theoretically possible.

The SDR try to substitute all the analogic functions for digital ones, and have as a final objective connect a totally digital machine to a antenna which do the interface with the air. This final objective has some technological limitations, mainly because the analogic to digital converters (ADC) don’t work in the radiofrequency bands and don’t have the required bandwidth. The typically implementations have a radiofrequency module, and the sampling is done in a lower frequency.

The SDR is supported over some conversion and process dispositives like Digital Signal Processing (DSP) or analogic to digital converters or vicevers (ADC/DAC). Another used dispositivo is the Field Programmable Gate Array (FPGA) which has some hardware functions (for example logic functions, memory blocks or microprocessors) whose internal connexion can be configurated by the user.

As it has been said previously in our case the hardware part is implemented by the Universal Software Receiver Peripheal (USRP) and the software role is done by the GPS-SDR [W.H].
2.2 Source Code analysis of GPS-SDR

2.2.1 Program Structure

The GPS-SDR is coded in C++ [W.H]. Some of the more abstract, and performance inhibiting, aspects of C++ are not used. These include polymorphism, operator overloading, and inheritance. In fact the entire source could easily be converted to straight C by changing the classes to structures.

The main point of the software is the idea of a threaded object. Each discrete task that the receiver must perform is given a plainly named object and an associated thread. The object can be treated like a black box with an input and output(s). The threaded object will be fed data through a blocking pipe, which will be read by the objects Import() method. The method will perform its duties on the data it received, and then send the resulting data to its destination via a Export() method. Note that the blocking pipes achieve two things at once: the pipes both allow data to flow to and from a process as well as timing the process. Since the process will block on its Import() call if the incoming pipe is empty no assignment of thread execution frequencies is required. Threads are executed as needed when data is made available to them.

The actual thread of the threaded object should look something like this:

```c
while(1) {
    pObject->Import();
    pObject->Parse();  \* more than one method can be here
    pObject->Export();
}
```

Figure 2.1: Thread Schedule

The template for a threaded object is the following:
typedef class OBJECT
{
    private:
        pthread_t thread; // For the thread
        int32 temp;
    public:
        OBJECT();         // Constructor
        ~OBJECT();        // Destructor
        void Import();
        void Parse();
        void Export();
        void Start();
        void Stop();
};

Figure 2.2: Template for a threaded object

Threaded Objects

- **Keyboard**: Handles user input from stdio.
- **PVT**: Computes the Position Velocity Time solution.
- **FIFO**: Gets data from either the USRP or disk.
- **Correlators**: Manage all the correlation process (one per channel)
- **Acquisition**: Performs acquisition (3 modes: strong, medium or weak signal).
- **Ephemeris**: Parses the ephemeris collected in the navigation message.
- **SV Select**: Selects the satellites to be searched based on the almanac
- **Telemetry**: Collects information of the rest of threads and interfaces with the Graphical User Interface.

Pipes

- Interplay between acquisition and tracking:
  - **Acq 2 Trak P**: Get an acquisition result
Chapter 2. GPS Software Defined Radio

- **Trak.2.Acq.P**: Request an acquisition because some of the channels are empty

- **Interplay between correlator and channels:**
  - **Corr.2.PVT.P[MAX CHANNELS]**: Output measurements to PVT
  - **PVT.2.Corr.P[MAX CHANNELS]**: Output measurements to PVT
  - **Trak.2.Corr.P[MAX CHANNELS]**: Have the tracking tell the correlator to start or stop a channel

- **Decode the ephemeris:**
  - **Chan.2.Ephem P**: Dump raw subframes to Ephemeris

- **User feedback:**
  - **FIFO.2.Telem.P**: Send FIFO status (nodes empty, agc value)
  - **FIFO.2.PVT.P**: Send number of incoming measurements to nav
  - **Acq.2.Telem.P**: Send latest acquisition attempts to GUI
  - **PVT.2.Telem.P**: Send latest nav solution to GUI
  - **Ephem.2.Telem.P**: Send latest ephemeris to GUI
  - **SV.Select.2.Telem.P**: PIPES Send predicted SV states to GUI
  - **PVT.2.SV.Select.P**: Output nav state to sat select
Figure 2.3: GPS-SDR Pipes Schedule


2.2.2 Analysis of the source code

Directory structure of the source code

As we can see above the GPS-SDR software has nine folders organized as following:

- **main**: initializes everything (hardware, pipes, objects and threads) and cleanly shutdown everything if something wrong happens.

- **includes**: contains header files.

- **usrp**: contains a program for USRP configuration and a DBSRX daughterboard driver.

- **simd**: SIMD Multimedia Extensions.

- **objects**: implementation of the threaded objects.

- **acquisition**: signal acquisition procedures.

- **accessories**: implementation of FFT, generation of GPS codes, generation of sine waves...

- **documentation**: doxygen documentation files.

At the next pages we’ll describe the most important folders. Explaining threaded objects separately.
Figure 2.5: Content of the main directory

**main/main.cpp**

This file contains the main() function. It initializes everything (hardware, pipes, objects and threads) and cleanly shutdown everything if something wrong happens.

1. Parse arguments: sets default values to parameters and override them accordingly to the arguments passed to the main program GPS-SDR.
2. Hardware initialization: detects hardware and initialize it for realtime mode. The detectable options are defined in simd/cpuid.cpp
3. Pipes initialization: defines pipes for communication between processes.
4. Object initialization: creates a keyboard object to handle user input, acquisition, ephemeris, input data (USRP or file), correlation channels, etc.
5. Thread initialization: set the global run flag (grun) to true, starts the keyboard thread, the PVT (Position-Velocity-Time), the FIFO, the correlators, the acquisition, the ephemeris, the satellite select thread and the telemetry.
6. Keep everything running while all the threads are active. Check it every 10 ms.
7. Shutdown everything.

The corresponding functions (Parse_Arguments(), Hardware_Init(), Pipes_Init() and so on) are defined in the file init.cpp, and their shutdown counterparts are located in shutdown.cpp.
main/init.cpp
This file initializes the hardware, the pipes and the threaded objects and starts up all the threads.

main/shutdown.cpp
Stop all threads, shutdowns all pipes and threaded objects and deletes the global variables.

INCLUDES

includes/includes.h
The includes/includes.h header file includes standard Operating System headers (like stdlib.h, stdio.h, ctype.h, math.h, and so on) and some other important files:

- #include "config.h": Receiver configuration
- #include "signaldef.h": Define attributes of input data
- #include "defines.h": Defines from SIS-GPS-200D and some other things
- #include "macros.h": Macros
- #include "structs.h": Structs used for interprocess communication
- #include "protos.h": Functions and thread prototypes
- #include "simd.h": Include the SIMD functionality

It also includes the threaded objects:

- #include "fft.h": Fixed point FFT object
- #include "fifo.h": Circular buffer for importing IF data
- #include "keyboard.h": Handle user input via keyboard
- #include "correlator.h": Correlator
- #include "channel.h": Tracking channels
- #include "acquisition.h": Acquisition
- #include "pvt.h": PVT solution
- #include "ephemeris.h": Ephemeris decode
- #include "telemetry.h": Ncurses telemetry
- #include "sv_select.h": Drives acquisition/reacquisition process
Chapter 2. GPS Software Defined Radio

- **#include ”post_process.h”**: Run the receiver from a file and, finally, the global variables:
  - **#include ”globals.h”**: Global variables and objects

**includes/globals.h**
In this file there are declared the software globals as EXTERN variables. The extern keyword is used to specify that the variable is declared in a different file. This is mostly used to declare variables of global scope in C++ projects. When the keyword extern is used, the compiler will not allocate memory for the variable.

**includes.defines.h**
This file defines some nominal system parameters. One of them, specially relevant is the sample frequency 2048000bps.

**includes/config.h**
This file defines the configuration parameters as: number of channels, number of CPU Cores, type of data and channel, correlator or acquisition defines.

**ACCESSORIES**

**accessories/misc.cpp**

- **void sine_gen**(CPX *dest, double f, double fs, int32 samps) generates a full scale sinusoid of frequency f with sampling frequency fs for samps samps and put it into dest.
- **void sine_gen**(CPX *dest, double f, double fs, int32 _samps, double p) the same as above but adding a phase constant p.

**2.2.3 Threaded Objects**

**ACQUISITION**

The structure of the acquisition object is the following:

- **Inclusion of headers**: includes.h (see section INCLUDES) and prn_codes.h, which contains the FFT of the PRN codes. These codes are generated with Matlab, see acquisition/prn-gen.m and acquisition/gen_fft_codes.m.
• **Main acquisition thread:**

```c
void *Acquisition_Thread(void *arg)
{
    Acquisition *aAcquisition = pAcquisition;
    while(grun)
    {
        aAcquisition->Inport();
        aAcquisition->Acquire();
        aAcquisition->Export(NULL);
    }
    pthread_exit(0);
}
```

Figure 2.6: Acquisition thread main

• **Acquisition::Start():** Actually starts the acquisition thread.

• **Acquisition::Stop():** Actually stops the acquisition thread.

• **Acquisition constructor:** `Acquisition::Acquisition(float _fsample, float _fif)` creates some variables, grabs some constants and the FFT codes, allocates some buffers, generates sinusoids for wiping off.

• **Acquisition deconstructor:** `Acquisition::Acquisition()` deletes everything and closes the acquisition pipe.

• **IF Processing:** `Acquisition::doPrepIF(int32 _type, CPX *buff)` completes all of the up-front IF processing: resampling the buffer, mixing to baseband, 25-Hz, 500 Hz, and 750 Hz, computing the forward FFT, then copying the FFTd data into a 2-D matrix. The matrix is created in such a way to allow the circular-rotation trick to be carried out without repeatedly calling doRotate.

• **Strong signal acquisition:** `Acq_Result_S Acquisition::doAcqStrong(int32 _sv, int32 _doppmin, int32 _doppmax)` acquires using a 1 ms coherent integration.

• **Mediumstrength signal acquisition:** `Acq_Result_S Acquisition::doAcqMedium(int32 sv int32 _doppmin, int32 _doppmax)` acquires using a 10 ms coherent integration.

• **Weak signal acquisition:** `Acq_Result_S Acquisition::doAcqWeak(int32 _sv, int32 _doppmin, int32 _doppmax)` acquires using a 10 ms coherent integration and 15 incoherent integrations.
Chapter 2. GPS Software Defined Radio

- **Acquire**: according to request.type, performs the IF processing and calls strong, medium or weak signal acquisition.

- **Inport**: This function waits the Trak_2.Acq_P pipe.

- **Export**: This function opens the log file (called Acq.txt by default) and writes on it the type of correlation (STRONG, MEDIUM or WEAK), the satellite number, delay, doppler, magnitude and success for each satellite. Then, closes the file. Finally, the function writes the result to the tracking and the telemetry pipes.

**CHANNEL**

The Channel class implements the channels, using correlations to close the loops. These are its public methods:

- Channel(int32 chan); Creates a FFT object. If specified, it also opens one chanXX.dat file.

- Channel(); Destroys the FFT object. If specified, it also closes the chanXX.dat file.

- void Start(int32 sv, Acq Result S result, int32 corr len);

- void Stop(); Stops Channel Thread.

- void Clear(); Clean Channel Thread Variables.

- void DumpAccum(); Dump the accumulation and do rest of processing

- void FrequencyLock() Use FFT to pull in the PLL

- void PLL W(float bw); Change the PLL bandwidth

- void DLL W(float bw); Change the DLL bandwidth

- void PLL(); Perform the phase lock loop

- void DLL(); Do the DLL

- void Epoch(); Increase 1ms epoch, 20ms epoch

- void Export() Return NCO command to correlator

- Chan Packet S getPacket();

- void Accum(Correlation S *corr, NCO Command S * feedback); Process an accumulation
Chapter 2. GPS Software Defined Radio

CORRELATOR

The Correlator class implements a bank of correlators. These are its public methods:

- Correlator(int32 chan); Correlator thread constructor.
- void Inport(); Get IF data, NCO commands, and acq results
- void Correlate(); Run the actual correlation
- void Export(); Dump results to channels and Navigation
- void Start(); Start the thread
- void Stop(); Stop the thread
- void TakeMeasurement(); Take some measurements
- void SamplePRN(); Sample all 32 PRN codes and put it into the code table
- void GetPRN(int32 sv); Get row pointers to specific PRN
- void InitCorrelator(); Initialize a correlator/channel with an acquisition result
- void DumpAccum(Correlation S *c); Dump accumulation to channel for processing
- void UpdateState(int32 samps); Update correlator state
- void ProcessFeedback(NCO Command S *f);
- void Accum(Correlation S *c, CPX *data, int32 samps); Do the actual accumulation
- void SineGen(int32 samps); Dynamic wipeoff generation

EPHEMERIS

The Ephemeris class extracts the ephimeris data from the navigation message. These are its public methods:

- Ephemeris(); Ephemeris thread constructor.
- void Parse(int32 sv); Parse data message into decimal values
- void ParsePage(int32 sv id); Parse almanac page
void Import(); Read data from channels
void Export(); Send stuff to the telemetry thread
void Start(); Start the thread
void Stop(); Stop the thread
void Lock(); Lock critical data
void Unlock(); Unlock critical data
void ClearSV(int32 sv); Dump an ephemeris (usually from a detected cross correlation)
void WriteEphemeris(); Write ephemerides to a txt file
void ReadEphemeris(); Read ephemerides from the same txt file
void WriteAlmanac(); Write almanacs to a YUMA file
void ReadAlmanac(); Write almanacs from a YUMA file
Ephemeris S getEphemeris(int32 sv) return(ephemerides[ sv]); Get this SV’s ephemeris values
Almanac S getAlmanac(int32 sv) return(almanacs[ sv]); Get this SV’s almanac values
int32 getIODE(int32 sv) return(iode master[sv]); Get the current IODE for this SV

PVT

The PVT class does the Position Velocity Time solution. These are its public methods:

PVT(int32 mode); PVT thread constructor.
void Inport(); Read from the corr 2 nav pipe
void Start(); Start the thread
void Stop(); Stop the thread
void Navigate(); main navigation task, call the following functions:
  – void ProjectState(); project state to current measurement epoch
  – void Update Time(); estimate GPS time at current tic
Chapter 2. GPS Software Defined Radio

- void Get Ephemerides(); get the current ephemeris from Ephemeris Thread
- void SV TransitTime(); Transit time to each SV
- void SV Positions(); calculate SV Positions (for each channel with valid ephemeris)
- void SV Elevations(); calculate SV Elevation angles for elevation mask
- void SV Correct(); correct SV positions for transit time
- void PseudoRange(); calculate the pseudo ranges
- void FormModel(); form direction cosine matrix and prediction residuals
- bool PreErrorCheck(); check all SV’s for bad measurements, etc
- void ErrorCheckCrossCorr(); check for cross-correlation problem via ephemeris match
- bool PostErrorCheck(); check all SV’s for bad measurements, etc
- bool Converged(); declare convergence
- void Residuals(); compute residuals
- void PVT Estimation(); estimate PVT. Uses Least Squares!!
- void ClockUpdate(); update the clock
- void LatLong(); convert ECEF coordinates to Lat,Long,Height
- void DOP(); calculate DOP terms
- void ClockInit(); initialize clock
- void Raim(); do a RAIM algorithm to look out for a bad SV

- void WritePVT(); Write the PVT to disk for a later warm start
- void ReadPVT(); Read the PVT from disk for a later warm start
- Nav Solution S getNav() return(master nav);
- Clock S getClock() return(master clock);
- void Export(); Export Navigator data to appropriate pipes
- void Reset(); Reset the Navigation solution to naught values
- void Reset(int32 chan); Reset individual channel
- double GPSTime(); Calculate GPS time from PC’s clock
TELEMETRY

The Telemetry class gathers all relevant receiver data and pipes it to the separate GUI application. These are its public methods:

- Telemetry(int32 ncurses); Telemetry thread constructor.
- void Inport(); Import data from other threads.
- void Export(); Export data to other threads.
- void LogNav(); Write Navigation data to a file.
- void LogPseudo(); Write pseudodistance to a file.
- void LogTracking(); Write tracked SV’s to a file.
- void LogSV(); Write SV’s to a file.
- void LogGoogleEarth(); Write Google Earth file body.
- void GoogleEarthFooter(); Write Google Earth file footer.
- void GoogleEarthHeader(); Write Google Earth file header.

FIFO

The FIFO class gets data and passes it into the receiver. These are its public methods:

- FIFO(); Create circular FIFO
- FIFO(); Destroy circular FIFO
- void Inport(); Get data from USRP Uno
- void Start(); Start up the thread
- void Stop(); End the thread
- void Enqueue(); Enqueue data to the FIFO.
- void Dequeue(int32 resource, ms packet *p);
- void SetScale(int32 age scale);
- void Wait(int32 resource);
SATELLITE SELECTION

The SV Select class contains the channels and drives the channel objects. These are its public methods:

- SV Select(); SV Select thread constructor.
- void Import(); Import data to SV Select.
- void Export(); Export something
- void Start(); Start the thread
- void Stop(); Stop the thread
- void UpdateState(); Update acq type
- void Acquire(); Run the acquisition
- void GetAlmanac(int32 sv); Get the most uptodate almanacs from the ephemeris
- void SV Predict(int32 sv); Predict states of SV’s
- void SV Position(int32 sv); Compute SV positions from almanac
- void SV LatLong(int32 sv); Compute SV’s lat and long
- bool SetupRequest(); Setup the acq request
- void ProcessResult(); Take the result and do something with it!
- void MaskAngle(); Calculate elevation mask angle

KEYBOARD

This threaded object handles user input from stdio (the keyboard):

- Q quits the program (sets the global run variable grun to false, grun=0x0)
- n shows the navigation display.
- a shows the almanac and satellite prediction.
- h shows the acquisition history.
2.3 Computing the position in GPS-SDR

2.3.1 Satellite Position

In GPS-SDR, the computation of the satellite position, given the ephemeris read from the navigation message, is performed in the method SV Positions() of PVT. The official reference for this computation is [Ser04]. However, GPS-SDR implements the algorithm computing the derivatives of some parameters, which are not included in the algorithms that appear in [Ser04].

- Constant: Earth’s universal gravitational parameter \( \mu = 3.986005 \times 10^{14} \text{m}^3/\text{s}^2 \)
- Calculate the mean motion as
  \[
  n = \sqrt{\frac{\mu}{a_s}} + \Delta n
  \]
  where \( a_s \) is the semi-major axis of the satellite orbit obtained from subframe 2 in bits 227-234 and 241-264, and \( \Delta n \) is the mean motion difference obtained from subframe 2 in bits 91-106.
- The transmit time \( t_c \) can be computed as:
  \[
  t_c = \text{TOW} - \text{relative transit time} = \text{TOW} - \text{diff of dat} \times T_s
  \]
  where TOW is Time Of the Week, and the relative transit time is the same as the difference in digitized data points (“diff of dat”). The factor \( T_s \) is the time between digitized points.
- Furthermore, \( t_k \) shall be the actual total time difference between the time \( t_c \) and the epoch time \( t_{oe} \), and must account for beginning or end of week crossovers. That is, if \( t_k \) is greater than 302, 400 seconds, subtract 604, 800 seconds from \( t_k \). If \( t_k \) is less than -302, 400 seconds, add 604, 800 seconds to \( t_k \)
  \[
  \begin{align*}
  &\text{if } t_k = t_c - t_{oe} > 302400, \text{ then } t_k = t_k - 604800 \\
  &\text{if } t_k = t_c - t_{oe} < -302400, \text{ then } t_k = t_k + 604800
  \end{align*}
  \]
  where \( t_{oe} \) is the reference time ephemeris obtained from navigation data (subframe 2, bits 271-286).
- Once the GPS system time at time of transmission \( (t_k) \) is found, the mean anomaly can be found as
\[ M = M_0 + n(t_c - t_{oe}) = M_0 + n t_k \]

where \( M_0 \) is the mean anomaly at reference time obtained from subframe 2 bits 107-114, 121-144.

- The eccentric anomaly \( E \) can be found as

\[ E = M + e_s \sin(E) \]

where \( e_s \) is the eccentricity of satellite orbit obtained from subframe 2 (bits 167-174 and 181-204). Since this equation is nonlinear, the iteration method will be used to obtain \( E \):

```c
// Obtain eccentric anomaly E by solving Kepler’s equation.
ecc = ephem->ecc;
E = M;
for (iter = 0; iter < 20; iter++)
{
    sE = sin(E); cE = cos(E);
    dEdM = 1.0 / (1.0 - ecc * cE);
    if (fabs(dtemp=(M-E+ecc*sE)*dEdM) < 1.0E-14
        break;
    E += dtemp;
}
```
We can also compute its rate:

\[
\dot{E} = \frac{n}{1 - e_s \sin(E)}
\]

- Compute the relativistic correction term

\[
\Delta t_r = F e_s \sqrt{\alpha_s \sin(E)}
\]
where $F = -4.442807633 \times 10^{-10} s/m^{1/2}$ is a constant.

• The overall time correction term is

$$\Delta t = a_{f_0} + a_{f_1}(t_c - t_{oc}) + a_{f_2}(t_c - t_{oc}) + \Delta t_r - t_{GD}$$

where $a_{f_0}$ (bits 271-292), $a_{f_1}$ (bits 249-264), $a_{f_2}$ (bits 241-248), $t_{oc}$ (bits 219-234) are satellite clock corrections, $t_{GD}$ is the estimated group delay differential (bits 197-204), and all are obtained from subframe 1.

• Computation of the true anomaly $\nu$ according to [Ser04]:

$$\nu = \tan^{-1}\left(\frac{\sqrt{1 - e^2} \sin(E)}{\cos(E) - e_s}\right)$$

and the argument of latitude $\phi$:

$$\phi = \nu + \omega$$

where $\omega$ is the argument of perigee (subframe 3, bits 197-204 and 211-234) obtained from navigation data. In GPS-SDR, this is computed as:

```c
sqrtlmeec = sqrt (1.0 - ecc * ecc);
sE = sin(E); cE = cos(E);
P = atan2 (sqrtlmeec + sE, cE - ecc) + ephem->argp;
```

• Compute the second harmonic perturbation terms:

$$\delta \phi = C_{us} \sin 2\phi + C_{uc} \cos 2\phi$$

$$\delta r = C_{rs} \sin 2\phi + C_{rc} \cos 2\phi$$

$$\delta i = C_{is} \sin 2\phi + C_{ic} \cos 2\phi$$

where $C_{us}$ (subframe 2, bits 211-226), $C_{uc}$ (subframe 2, bits 151-166), $C_{rs}$ (subframe 2, bits 69-84), $C_{rc}$ (subframe 3, bits 181-196), $C_{is}$ (subframe 3, bits 121-126), and $C_{ic}$ (subframe 3, bits 61-76) are obtained from navigation data. These three terms are used to correct the following terms:

$$\phi \implies \phi + \delta \phi \implies u = \omega + v + C_{uc} \cos (2(\omega + v)) + C_{us} \sin (2(\omega + v))$$

$$r \implies r + \delta r \implies r = a(1 - e \cos(E)) + C_{rc} \cos(2(\omega + v)) + C_{rs} \sin(2(\omega + v))$$

$$i \implies i + \delta i \implies i = i_0 + \dot{t}_i + C_{ic} \cos(2(\omega + v)) + C_{is} \sin(2(\omega + v))$$
where \( i_0 \) is the inclination angle at reference time (these data are divided into two parts, the 8-bit MSB from 137-144 bits and 24-bit LSB from 151-174 bits of subframe 3), and \( i \) is the rate of inclination angle (subframe 3, bits 279-292). The source code:

\[
\begin{align*}
\dot{u} &= \dot{\phi}(1 + 2(C_{us} \cos(2\phi) - C_{uc} \sin(2\phi))) \\
\dot{\phi} &= \sqrt{1 - e_s^2} \frac{\dot{E}}{1 - e_s \sin(E)} \\
\dot{i} &= \dot{i} + 2\dot{\phi}(C_{is} \cos(2\phi) - C_{ic} \sin(2\phi))
\end{align*}
\]

and their derivatives:

\[
\begin{align*}
\dot{X}_p &= r \cos(u) \\
\dot{Y}_p &= r \sin(u) \\
\dot{X}_p &= \dot{r} \cos(u) + r \cos(\dot{u}) \\
\dot{Y}_p &= \dot{r} \sin(u) + r \sin(\dot{u})
\end{align*}
\]

- Compute the satellite’s position in its orbital plane, \((X_p, Y_p)\).
- Compute the corrected longitude of the ascending node, \(\Omega\).
\[ \Omega = \Omega_0 + t_k(\dot{\Omega} - \Omega_e^{WGS84}) - \Omega_e^{WGS84}t_{oe} \]

\[ \dot{\Omega} = \dot{\Omega} - \Omega_e^{WGS84} \]

where \( \Omega_0 \) is the longitude of the ascending node of the orbit plane at weekly epoch (these data are divided into two parts, the 8-bit MSB from 77-84 and 24-bit LSB from 91-114 of subframe 3), \( \dot{\Omega} \) is the rate of the right ascension (subframe 5, bits 121-136) and \( \dot{\Omega}_e^{WGS84} = 7.2921151467 \times 10^{-5} \) rad/s is the WGS84 value of the earth’s rotation rate.

- Compute the satellite’s position in space, \((x, y, z)\).
  \[
  x = X_p \cos(\Omega) - Y_p \cos(i) \sin(\Omega) \\
  y = X_p \sin(\Omega) + Y_p \cos(i) \cos(\Omega) \\
  z = Y_p \sin(i)
  \]

- Compute SV clock correction
  \[
  t_s = t_k \\
  \text{clock bias} = af_0 + af_1(t_c - t_{oc}) + af_2(t_c - t_{oc})2 + \Delta t_r - t_{GD} \\
  \text{Frequency bias} = af_1 + 2af_2(t_c - t_{oc})
  \]

- Satellite’s velocity, \((v_x, v_y, v_z)\).
  \[
  v_x = -\dot{\Omega}y + \dot{X}_p \cos(\Omega) - \dot{Y}_p \cos(i) \sin(\Omega) + Y_p \sin(i) \dot{I} \sin(\Omega) \\
  v_y = \dot{\Omega}x + \dot{X}_p \sin(\Omega) + \dot{Y}_p \cos(i) \cos(\Omega) - Y_p \sin(i) \dot{I} \cos(\Omega) \\
  v_z = Y_p \cos(i) \dot{I} + \dot{Y}_p \sin(i)
  \]

### 2.3.2 User Position

The pseudorange for a given satellite can be expressed as [Fer00]:

\[
\rho = c(t_{Rxuser} - t_{Txsat})
\]

where \( c = 299.792.458 \) m/s is the speed of light, \( t_{Rxuser} \) is the measured arrival time, including the user’s clock bias estimate \( b_u \) and the error produced by receiver noise, multipath and other nuisance effects:

\[
t_{Rxuser} = t_{Rx} + b_u + \nu = t_{Tx} + \frac{D}{c} + d_{trop} + d_{ion} + b_u + \nu
\]
Where \( t_{Rx} \) is the true arrival time, and it is splitted in terms of true transmission time, \( t_{Tx} \), true distance from receiver to satellite, \( D \), and the true delays caused by the troposphere, \( d_{trop} \), and the ionosphere, \( d_{ion} \). By the other way, the term \( t_{Txsat} \) in the previous equation refers to the satellite transmission time, which can be expressed in function of the true transmission time, \( t_{Tx} \), and the true bias in satellite’s transmission time, \( b_s \):

\[
t_{Txsat} = t_{Tx} + b_s
\]

Since the position of the satellite, \( r_s \), can be computed from the navigation message and the position of the user \( r_u \) is the desired output (along with the user clock bias estimate), it seems natural to express the true distance from receiver to satellite by means of \( D = u_s \cdot [r_s - r_u] \), where \( u_s \) is the true unit vector from user to satellite. Gathering this definition with the last three equations leads to:

\[
\rho = u_s \cdot [r_s - r_u] + c(b_u - b_s) + c(d_{trop} + d_{ion} + \nu)
\]

In this last equation, the satellite’s position \( r_s \) is known because it has been computed from the navigation message. Estimations of the satellite clock bias \( b_s \), the delay caused by the troposphere \( d_{trop} \) and the delay caused by the ionosphere \( d_{ion} \) are also contained in the navigation message. The desired information —user’s position \( r_u \) and the receiver clock bias \( b_u \) — remains unknown. Noise term \( \nu \) and the misadjustments between the estimation of the commented parameters from the navigation data and the true value will contribute as sources of error in the navigation solution. If we are acquiescent in the degradation provoked by these deviations, we are only interested in four parameters: three for the user position and one for the clock bias. Indeed, four equations (i.e. four tracked satellites) will be the minimum number to work out the value of the unknowns. Actually, pseudorange equations are non-linear. For a given satellite \( i \), if we make explicit reference to the three spatial unknowns, equations turns into

\[
\rho_i = \sqrt{(x_{si} - x_u)^2 + (y_{si} - y_u)^2 + (z_{si} - z_u)^2} + c(b_u - b_{si}) + \text{noise terms}
\]

Next step is the linearization of the problem, performed by means of taking the first order term of the Taylor expansion around the point of interest, \( r_0 \). This point of interest could be a very rough estimation of the user’s position (for example, the country or the continent), and subsequently be refined in latter iterations. The initialization for the satellite clock offset is set to \( b_{s,0} = 0 \). Namely, the truncated Taylor expansion can be written as

\[
\rho_i(x, y, z, b_s) \simeq \rho_i(x_0, y_0, z_0, 0) + \nabla \rho_i|_{(x=x_0, y=y_0, z=z_0, b_s=0)} (r_u - r_0)
\]

Computing the gradient of \( \rho_i(x, y, z, b_s) \):
\[
\frac{\partial \rho_i}{\partial x} \bigg|_{(x=x_0, y=y_0, z=z_0, b_s=0)} \triangleq \alpha_i = \frac{x_0 - x_{s_i}}{D_i} \\
\frac{\partial \rho_i}{\partial y} \bigg|_{(x=x_0, y=y_0, z=z_0, b_s=0)} \triangleq \beta_i = \frac{y_0 - y_{s_i}}{D_i} \\
\frac{\partial \rho_i}{\partial z} \bigg|_{(x=x_0, y=y_0, z=z_0, b_s=0)} \triangleq \gamma_i = \frac{z_0 - z_{s_i}}{D_i} \\
\frac{\partial \rho_i}{\partial b_s} \bigg|_{(x=x_0, y=y_0, z=z_0, b_s=0)} = c
\]

where \( D_i = \sqrt{(x_{s_i} - x_u)^2 + (y_{s_i} - y_u)^2 + (z_{s_i} - z_u)^2} \) is the geometric distance. Gathering the data, the model and the unknowns in a matrix notation, the equation system when \( N \geq 4 \) satellites are being tracked is:

\[
\begin{pmatrix}
\rho_1 - D_1 + b_{s,1} \\
\vdots \\
\rho_N - D_N + b_{s,N}
\end{pmatrix} =
\begin{pmatrix}
\alpha_1 & \beta_1 & \gamma_1 & 1 \\
\vdots & \vdots & \vdots & \vdots \\
\alpha_N & \beta_N & \gamma_N & 1
\end{pmatrix}
\begin{pmatrix}
\Delta x \\
\Delta y \\
\Delta z \\
\Delta c
\end{pmatrix}
\]

or, more comfortably, \( \mathbf{y} = \mathbf{A}\mathbf{x} \). In the design of hybrid GPS/Galileo receivers, the vector of unknowns \( \mathbf{x} \) can also include the receiver clock offset with respect to the Galileo System Time (GST) scale in order to take advantage of both constellation of satellites. Once the pseudorange equations have been linearized, a standard least squares procedure provides an estimation of the difference between user position and clock and the linearization point [Fer00]:

\[
\hat{x}_{LMS} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y} 
\]

Assuming \( \mathbf{A}^T = [a_1^T ... a_m^T] \) where \( a_j \) are \( n \) vectors. Then

\[
\mathbf{A}^T \mathbf{A} = \sum_{j=1}^{m} a_j a_j^T
\]

The obtained position and receiver clock bias can be used as the new linearization point and the process can be iterated to obtain a refined estimation. This solution can be improved in many ways; there are a number of techniques described in GPS literature. Some of this techniques can be the weighted least squares estimation, the Kalman filter solution or the DGPS technique.
### 2.3.3 Other Positioning Algorithms

#### Weighted Least Mean Squares

In fact, not all the observations will be as good as others, in terms of less noise. For example, the satellite observations with very lower elevation will have more ionospheric and tropospheric delay than the zenital ones, in addition the first ones will cause a multipath effect more difficult to mitigate.

Assuming these aspects, a weight will be assigned to each observation. It will be assumed that the aleatory variable $P^j$ has a Gaussian distribution, whose average is $\hat{P}^j = E[P^j]$ and his typical deviation $\sigma_{P^j}$. It could be proved that divide the equation 2.1 by the typical deviation produces a lower residue[ Fer00 ].

$$\frac{y_i}{\sigma_i} = \sum_{j=1}^{n} a_{ij} \frac{x_j}{\tau_j} = \sum_{j=1}^{n} a_{ij} \frac{x_j}{\tau_j}$$  \tag{2.3}

Expressing it in matrix notation (since now we will call $A$ the weight matrix divided for the typical deviation):

$$A = \begin{pmatrix} \frac{1}{\tau_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\tau_m} \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} \frac{a_{11}}{\sigma_1} & \cdots & \frac{a_{1n}}{\sigma_1} \\ \vdots & \ddots & \vdots \\ \frac{a_{m1}}{\sigma_m} & \cdots & \frac{a_{mn}}{\sigma_m} \end{pmatrix}$$  \tag{2.4}

It is defined

$$B = \begin{pmatrix} \frac{1}{\sigma_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma_m} \end{pmatrix}$$  \tag{2.5}

a diagonal and squared matrix $m \times m$.

The matrix $B$ is assumed as diagonal which involve that the satellites equations will be lineally independents, thing which not always occurs. Now we have to substitute at the LMS solution 2.1 the matrix $A$ by $BA$ and the vector $y$ by $By$:

$$x = [(BA)^T(BA)]^{-1}(BA)^TBy = [A^T B^T BA]^{-1} A^T B^T By$$  \tag{2.6}

In this step the weight matrix $P$ is defined as:
\[ P = B^TB = \begin{pmatrix} \frac{1}{\sigma_1^2} & \cdots & 0 \\ \cdots \\ 0 & \cdots & \frac{1}{\sigma_m^2} \end{pmatrix} \]  

(2.7)

and the covariance matrix \( C_y \) as \( P \)'s inverse:

\[ C_y = P^{-1} = \begin{pmatrix} \sigma_1^2 & \cdots & 0 \\ \cdots \\ 0 & \cdots & \sigma_m^2 \end{pmatrix} \]  

(2.8)

So, finally the position solution through the weighted least mean squares (WMS) is:

\[
\hat{x}_{WMS} = [A^T C_y^{-1} A]^{-1} A^T C_y^{-1} y
\]  

(2.9)

Kalman Filter solution

The kalman filter is one of the most used tools to study the data obtained from observations when we have an idea of the results which we would have to get [Fer00]. It works comparing the obtained results in the observations with the samples of a predetermined model. The filter will give a combination of both samples having into account their fiability which is quantified with their variances. Now we will show the Extended kalman filter derivation seen as the least mean squares solution with aprioristic information.

With the aprioristic information we will have the position priori estimation and his a priori information matrix. This information must be included in the equations.

Beginning with the previous lineal system, \( y = Ax + v \), with a covariance matrix \( C_y \). Now equations as \( x = x_0 \) with a covariance matrix \( C_{x_0} \) are added. The \( x_0 \) values is the value which we associate to the unknown value, with a certain variance \( \sigma_x^2 \). Generalizing the previous development:

\[
\begin{pmatrix} y \\ x_0 \end{pmatrix} = \begin{pmatrix} A \\ I_n \end{pmatrix} x
\]  

(2.10)

where \( I_q \) is the identity matrix with a \( q \times q \) size where \( q \) is the number of ligatures.

The new covariance matrix will be:
\[ C^{-1} = \begin{pmatrix} \frac{1}{\sigma_1^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma_m^2} \end{pmatrix} = \begin{pmatrix} C_{y}^{-1} & 0 \\ 0 & C_{x_0}^{-1} \end{pmatrix} \quad (2.11) \]

Previously we have seen that the least mean squares solution with weights is \( \hat{x}_{WMS} = [A^T C_{y}^{-1} A]^{-1} A^T C_{y}^{-1} y \). Adding the a prioris to this equation and working with the matrix by blocks:

\[
\hat{x} = \left[ \begin{pmatrix} A \\ I_n \end{pmatrix}^T \begin{pmatrix} C_{y}^{-1} & 0 \\ 0 & C_{x_0}^{-1} \end{pmatrix} \begin{pmatrix} A \\ I_n \end{pmatrix} \right]^{-1} \begin{pmatrix} A \\ I_n \end{pmatrix}^T \begin{pmatrix} C_{y}^{-1} & 0 \\ 0 & C_{x_0}^{-1} \end{pmatrix} \begin{pmatrix} y \\ x_0 \end{pmatrix} \\
= \left[ (A^T C_{y}^{-1} C_{x_0}^{-1}) \right] \begin{pmatrix} A \\ I_n \end{pmatrix}^T \begin{pmatrix} C_{y}^{-1} C_{x_0}^{-1} \end{pmatrix} \begin{pmatrix} y \\ x_0 \end{pmatrix} \\
= [A^T C_{y}^{-1} A + C_{x_0}^{-1}]^{-1} (A^T C_{y}^{-1} y + C_{x_0}^{-1} x_0) \\
\quad (2.12) 
\]

The kalman filter formulation starts from the last result 2.12. Using the inverse matrix formula and matricial algebra, we can get the following expression:

\[
\hat{x}_{kalman} = x_0 + K (y - Ax_0) \\
\quad (2.13) 
\]

\[
K = C_{x_0} A^T (C_{y} + AC_{x_0} A^T)^{-1} \\
\quad (2.14) 
\]

This new formulation has very interesting aspects. Whereas in the expression 2.12 we need to invert a \( q \times q \) matrix (where \( q \) is the number of unknown) now we only need to invert a \( m \times m \) matrix (where \( m \) is the number of observations). To get a solution we must have \( q \geq m \), in the case which we want to estimate a lot of unknown values to model the noise we will have \( q \gg m \). So, the new formulation is more efficient in his computation. We have to warn that we pay it with a higher numerical instability.
Differential DGPS

The DGPS (Differential GPS) is a system that provides GPS receivers correction of data received from GPS satellites in order to provide greater accuracy in the calculated position. It was conceived primarily because the introduction of the selective availability (SA).

The DGPS is based in the fact that the errors produced by the GPS system by affecting the same (or very similar) to the receivers are located close together. The errors are strongly correlated in closer receivers.

A GPS receiver fixed on the ground (reference) that you know his exact position on the basis of other techniques, receives the position given by GPS, and can calculate the errors produced by the GPS system, comparing it with its own, known in advance. The receiver transmits the correction of errors at the receivers next to him, and with this information the others receivers, at the same time, also can correct the errors produced by the system within the coverage area of transmission of signals from GPS reference.

The DGPS structure will be:

- Monitorized Reference Station, which knows his position with a very high precision. It’s composed by:
  - A GPS receiver.
  - A microprocessor, to compute the GPS system errors and to generate the message structure provided to the receivers.
  - A transmitter, to establish a communication to the final user’s receivers.
- Final user, which is composed by a DGPS receiver (GPS + receiver to communicate with the reference station).

In fact, there are some modes to get the DGPS corrections. The most used ones are:

1. Received by radio, through a channel prepared for this purpose, like the RDS in a FM radio station.
2. Downloaded from Internet.
3. Provided by someone satellite system designed for this purpose. In the USA are the WAAS, in Europe the EGNOS and in Japan the MSAS, all of them compatible with each other.

The messages which are reported to the closer receivers we can include two types of corrections:
• A correction directly applied to the position. This has as problem that the reference station and the user receiver have to use the same satellites to compute their position, since the corrections are based in these satellites.

• A correction applied at the pseudodistance of each visible satellite. In this case the user will be able to do the correction with the four satellites with the best signal to noise ratio (SNR). This is a more flexible correction.

The error caused by selective availability (SA) varies even faster than the speed of data transmission. Therefore, along with the message that is sent to correction, it also sends the time of validity of the corrections and his trends. Therefore, the receiver must do some kind of interpolation to correct the errors.

If you want to increase the coverage area of DGPS corrections, and at the same time, minimize the number of fixed reference receivers will require modeling the spatial and temporal variations of the errors. In this case we would be talking about the wide area differential GPS.

With the DGPS we can correct in part the errors caused by:

• Ionosphere and troposphere propagation.

• Errors in the satellites positions (ephemeris).

• Errors produced by problems in the satellite clocks.

For DGPS corrections are valid, the receiver must be relatively close to a DGPS station, generally, less than 1000 km. The accuracy can be achieved by some two meters in latitude and longitude, and about 3 m in altitude.

For additionally information about the section 2.3 see [Fer00]

Chapter 4, analyses and compares all the previous explained techniques like LMS, WMS, Kalman filter and DGPS.
Chapter 3

Double Delta Correlation

During this section it will be seen how a GPS receiver tracks the received signal. Also it will be explained the multipath problem in a DLL tracking correlator receiver. Also the DLL GPS-SDR implementation will be explained. Finally, an improvement to the multipath problem and its software implementation will be presented. Both implementations will be done in the assembler language.

3.1 DLL (Delay Lock Loop)

One of the firsts steps in a GPS receiver is to track the GPS signal. To achieve this the receiver uses a technique based on a code tracking loop which keep tracked the code phase of the signal. The output of such a code tracking loop is a perfectly aligned replica of the code. The code tracking loop in a GPS receiver is done with a DLL (Delay Lock Loop) also called early-late tracking loop. The main idea of the DLL is to correlate the received signal with three internal generated replica’s of the code to know the delay of the tracked signal. The following figure shows the DLL implementation.
Figure 3.1: Basic code tracking loop block diagram.

As we can see at the previous picture the code tracking has basically three important steps:

1. The first step is downconvert the input signal (C/A code) to baseband by multiplying it by a local replica of the carrier wave.

2. The next step is multiply the incoming signal for the three local replicas of the code. These three replicas are usually generated with a spacing of $\pm \frac{1}{2}$ chip.

3. The last step is integrate and dump separately the three branches. The branches output’s is a numerical value which indicates how much the local replica of the code correlates with the incoming signal so it shows which local replica is more similar to the incoming code.

To understand the process better we’ll see an example extracted from [KB07] in the Figure 3.2:
The three correlation outputs $I_E$, $I_P$, and $I_L$ are compared to see which one provides the highest correlation. In figure (a) the late code has the highest correlation, so the code phase must be decreased. In the figure (b) the highest peak is located at the prompt replica, and the early and late replicas have equal correlation. In this case, the code phase is properly tracked.

When the local carrier wave is locked in phase and frequency the previous DLL (Figure 3.1) is the optimal one. But in the case of the GPS signals where there is a phase error on the local carrier wave the signal becomes more noisy and it makes more difficult to the DLL to keep lock on the code. So instead a three branches DLL, normally the GPS receiver uses a six branches early-late lock loop.
The design in Figure 3.3 has the advantage that it is independent of the phase on the local carrier wave. If the local carrier wave is in phase with the input signal, all the energy will be in the in-phase arm. But if the local carrier phase drifts compared to the input signal, the energy will switch between the in-phase and the quadrature arm.

If the code tracking loop performance has to be independent of the performance of the phase lock loop, the tracking loop has to use both the in-phase and quadrature arms to track the code. So now the DLL needs a feedback to the PRN code generator to adjust the code phase. At the following tables are showed some of the common DLL discriminators [KB07]:

![Figure 3.3: DLL block diagram with six correlators](image)
Table 3.1: Various types of delay lock loop discriminators and a description of them

<table>
<thead>
<tr>
<th>Type</th>
<th>Discriminator $D$</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coherent</td>
<td>$I_E - I_L$</td>
<td>Simplest of all discriminators. Does not require the $Q$ branch but requires a good carrier tracking loop for optimal functionality.</td>
</tr>
<tr>
<td>Noncoherent</td>
<td>$(I_E^2 + Q_E^2) - (I_L^2 + Q_L^2)$</td>
<td>Early minus late power. The discriminator response is nearly the same as the coherent discriminator inside $\pm \frac{1}{2}$ chip.</td>
</tr>
<tr>
<td></td>
<td>$(I_E^2 + Q_E^2) - (I_L^2 + Q_L^2)$</td>
<td>Normalized early minus late power. The discriminator has a great property when the chip error is larger than a $\frac{1}{2}$ chip; this will help the DLL to keep track in noisy signals.</td>
</tr>
<tr>
<td></td>
<td>$I_P(I_E - I_L) + Q_P(Q_E - Q_L)$</td>
<td>Dot product. This is the only DLL discriminator that uses all six correlator outputs.</td>
</tr>
</tbody>
</table>

The Figure 3.4 shows the coherent discriminator and three non-coherent discriminators using a standard correlator with an early, prompt and late spacing of $\pm \frac{1}{2}$ chip. The spacing between them determines the noise bandwidth in the DLL. So if the spacing is larger than $\pm \frac{1}{2}$ chip the DLL would be more noise robust, but in the opposite case a DLL with a smaller spacing would be more precise.

The implemented tracking loop discriminator in GPS-SDR is a variation of the normalized early minus late power. Thus, this discriminator is described as:

$$D = \frac{P_E - P_L}{P_E + P_L} = \frac{\sqrt{I_E^2 + Q_E^2} - \sqrt{I_L^2 + Q_L^2}}{\sqrt{I_E^2 + Q_E^2} + \sqrt{I_L^2 + Q_L^2}}$$

Where $I_E, Q_E, I_L, Q_L$ are output from four of the six correlators shown in Figure 3.3. The normalized early minus late power discriminator is chosen because thanks that it uses both inphase and quadrature arms it is independent of the performance of the PLL. Also, with the
normalization of the discriminator we achieve that it can be used with signals with different signal-to-noise ratios and strength.

![Figure 3.4: Comparison between the common DLL discriminator responses](image)

### 3.1.1 DLL in GPS-SDR

The implemented DLL in GPS-SDR is like the shown in the Figure 3.3. As we will explain later some of the function, specially the functions which requires a lot of calculations in a short period of time are implemented in assembler language to assure that we have enough time to do the calculation. The steps are the following ones:

1. Firstly we obtain the inphase and quadrature parts of the data. This step is implemented in `correlator.cpp` file and concretely in the function `void Correlator::Accum` and inside it in `sse-cmulsc`. The function’s code is showed in the file `simd/sse-new.cpp` inside the GPS-SDR’s source code.

2. Secondly step is multiply by the E,P,L and accumulate and dump the results which is done in `sse_pn_acum_new` function, also in `void Correlator::Accum` in `correlate.cpp`. The `void Correlator::Accum` function is showed in Figure 3.5.
### Chapter 3. Double Delta Correlation

```cpp
void Correlator::Accum(Correlation_S *c, CPX *data, int32 samps)
{
    CPX_ACCUM EPL[5];
    /* First do the wipeoff */
    int32 a=data->a;
    int32 b=data->b;
    sse_cmulsc(data, state.psine, scratch, samps, 14);
    /* Now do the accumulation */
    if(CDCORRELATOR==0){
        sse_prn_accum_new(scratch, state.pcodes[0], state.pcodes[1], state.pcodes[2], samps, &EPL[0]);
        c->I[0] += (int32) EPL[0].i;
        c->I[1] += (int32) EPL[1].i;
        c->Q[0] += (int32) EPL[0].q;
        c->Q[1] += (int32) EPL[1].q;
        c->Q[2] += (int32) EPL[2].q;
    }
}
```

**Figure 3.5:** `void Correlator::Accum GPS-SDR` implementation

3. The following step is calculate the P[0], P[1], P[2] which is done in file `/objects/channel.cpp` in function `void Channel::DumpAccum()`. The power calculation is illustrated in Figure 3.6.

```cpp
void Channel::DumpAccum()
{
    int32 lcv, bwrote;
    float NBP;
    float WBP;
    float cerr;
    float serr;

    /* Compute the powers */
    P[0] = I[0]*I[0]+Q[0]*Q[0];
    P[1] = I[1]*I[1]+Q[1]*Q[1];
}
```

**Figure 3.6:** Early, Late and Prompt Power Calculation
4. Finally, the last step is to calculate the DLL discriminator which is calculated in file /objects/channel.cpp and concretely in function void Channel::DLL(). The implementation is showed on Figure 3.7.

```c
void Channel::DLL()
{
    float code_err;
    float ep1, ep2, lp1, lp2, prompt;

    ep1 = sqrt(float(P[0]));
    lp1 = sqrt(float(P[2]));
    prompt= sqrt(float(P[1]));
    // DLL code correction
    code_err = (ep1 - lp1)/(ep1 + lp1);
}
```

Figure 3.7: DLL Code Correction

3.2 The multipath problem

GNSS signal processing has multiple error sources, but one of the more important is the multipath which impact directly to the code tracking performance. So as previously we’ve explained how the code tracking is implemented, it makes sense to investigate how multipath impacts the code tracking loop.

All the GPS receivers observe a distorted version of the original transmitted GPS signal. As we’ve said above one of this distortion is the effect called multipath propagation. Normally the receiver can directly see the satellite, so a part of the received signal has propagated via the direct path from the satellite to the receiver. This signal has a time delay due to the distance between the satellite and the receiver. In normal conditions the signal propagated via the direct path is by far the strongest which the receiver sees.

However that, additionally the receiver may observe other signals propagating via other and longer paths. This is because the transmitted signal interacts with one or more objects/obstacles in the environment between the path from the satellite to the receiver. In fact exists different kind of interactions between the radio waves and objects, but the more common in the GNSS environment are reflection altering the direction of propagation, amplitude, polarity, and phase of the radio wave.
According to [KB07] the indirect signal received by the GPS can generate a destructive or a constructive interference. We’ll have a constructive interference when the reflected and the direct signals are in phase so the amplitude of the sum is larger than the amplitude of each of the components and on the other hand, if the direct and reflected signals are out of phase, the amplitude of the sum signal decreases and we talk about destructive interference. In the last situation the received signal will be fading. The Figures 3.9 and 3.10 shows the two possible interference in the GPS signal.
Figure 3.9: Direct Path, Multipath (in phase) and Resulting Correlation Functions with finite BW
Figure 3.10: Direct Path, Multipath (out of phase) and Resulting Correlation Functions with finite BW

Thus in conclusion the multipath propagation deforms the ideal correlation peak because the received signal is a sum of a direct and an indirect received signals. It happens because the multipath components arrive later at the receiver and it contributes with additional correlation peaks. Due to the multipath effect the early-late correlator samples may not be centered on the true arrival time of the direct path. The Figure 3.11 shows this effect.
3.2.1 The Early-Late DLL and multipath

In deriving the Early-Late technique it's convenient to considerer the ideal situation where the pre-correlation BW is infinite and the resulting correlation function is triangular. So the Figures 3.12 and 3.13 shows the resulting correlation for the same situation as in Figures 3.9 and 3.10 which are obtained with a finite bandwidth.
Figure 3.12: Direct Path, Multipath (in phase) and Resulting Correlation Functions
The Figures 3.12 and 3.13 share two important characteristics. The first one is that the desired point is the point with the highest power, and the other is that the slopes of the functions on either sides of the peak are not equal. The goal of the DLL is to calculate the point of maximum power to be able to obtain the received signal time delay. To calculate it the process in the infinite bandwidth case is shown in Figure 3.14 where $d$ is the distance between the early and late, $y_1$ and $y_2$ are the power of the early and late respectively. Applying geometrics it is possible to calculate the maximum power peak and consequently the received time delay. It’s important to note that in Figures 3.12 and 3.13 as we have used and infinite bandwidth the position of the maximum peak doesn’t change, but in Figures 3.9 and 3.10 the position of the maximum changes, due to the fact that we are using a finite bandwidth.
Chapter 3. Double Delta Correlation

3.3 ∆∆DLL (Double Delta Correlation)

As we’ve said before the multipath effects the early-late correlator work, and it does that the obtained arrival of time isn’t exactly true and consequently the observer position estimation will be wrong.

We know that the goal of a DLL is to derive the code tracking error to zero and that the multipath effects to this goal. So to improve the effects of the multipath on the receiver it is possible to implement the ∆∆DLL.

As it is intuitive with the name and as it is shown in the Figure 3.15 the ∆∆DLL uses a second set of correlators that are spaced at exactly half distance of the first correlator set. Thus, as it is easy to imagine it is possible to improve the calculation of the maximum power peak.
Chapter 3. Double Delta Correlation

Figure 3.15: Double Delta Correlator Layout

Applying geometrics in the Figure 3.15 the \( \Delta\Delta\) DLL can be written as following.

Assuming that:

\[
\begin{align*}
    a_1 &= \frac{(e_1 - e_2)}{(d_2)} \\
    a_2 &= \frac{(l_1 - l_2)}{(-d_2)} \\
    y_1 &= e_1 \\
    y_2 &= e_2
\end{align*}
\]

The discriminator is:

\[
(e_1 - l_1) + \left( \frac{d_2}{2} \right) \left[ \frac{(e_1 - e_2) + (l_2 - l_1)}{\frac{d_2}{2}} \right]
\]

Simplifying:

\[
2(e_1 - l_1) - (e_2 - l_2)
\]

Also it’s possible to normalize the discriminator to achieve that it can be used with signals with different signal-to-noise ratios and strength. In fact this is what we have done in the \( \Delta\Delta\) DLL GPS-SDR discriminator.


3.4 ΔΔDLL vs DLL. Theoretical improvement in a multipath scenario

Previously, it has been said that ΔΔDLL correlator has a better response in a multipath environment than DLL. So, in this section we will see his theoretical improvement [KB07].

In general, in a multipath scenario, the received signal $x(t)$ is composed of the direct signal and $M-1$ multipath components. Let $A_i(t)$ denote the amplitude of the $i$th multipath component, let $D$ denote the navigation message, let $C$ denote the code, let $\tau$ denote the multipath error, let the frequency change be $v_i$, the phase offset be $\phi_i$, and, finally, add a noise term $n(t)$; then the signal can be described as:

$$\begin{align*}
x(t) &= \sum_{i=1}^{n} A_i(t)D(t - \tau_i(t))C(t - \tau_i(t))\cos(2\pi(f_0 + v_i(t))t + \phi_i(t) + n(t). \quad (3.1)
\end{align*}$$

To simplify the compute we consider a two-path scenario ($M = 2$) and make the following assumptions:

$$\begin{align*}
A_1(t) &= A_1 & A_2(t) &= A_2 \\
\tau_1(1) &= \tau_1 & \tau_2(t) &= \tau_2 \\
v_1(1) &= v_1 & v_2(t) &= v_2 \\
n(t) &= 0
\end{align*}$$

In other words we have assumed that the parameters (amplitudes, delays, and Doppler shifts) are constant over the time period we consider. Consequently, in this case, (3.1) can be reduced to:

$$\begin{align*}
x(t) &= A_1D(t - \tau_1)C(t - \tau_1)\cos(2\pi(f_0 + v_1(t))t + \phi_1) + \\
&\quad + A_2D(t - \tau_2)C(t - \tau_2)\cos(2\pi(f_0 + v_2(t))t + \phi_2) \quad (3.2)
\end{align*}$$

Assuming $\cos(\phi) = \Re(\exp(j\phi))$ to rewrite (3.2) as:
Chapter 3. Double Delta Correlation

74

constructive and destructive interference: between the direct and the reflected signal components. We consider the two special cases of

The output of the envelope discriminator is

\[ x(t) = \Re \left\{ A_1 D(t - \tau_1) C(t - \tau_1) \exp \left( j2\pi(f_0 + v_1)t + \phi_1 \right) + \\ A_2 D(t - \tau_2) C(t - \tau_2) \exp \left( j2\pi(f_0 + v_2)t + \phi_2 \right) \right\} \]

\[ = \Re \left\{ A_1 D(t - \tau_1) C(t - \tau_1) \exp \left( j2\pi(f_0 + v_1)t + \phi_1 \right) + \\ A_2 D(t - \tau_2) C(t - \tau_2) \exp \left( j2\pi(v_2 - v_1)t + (\phi_2 - \phi_1) \right) \times \exp \left( j(2\pi(f_0 + v_1)t + \phi_1) \right) \right\} \]

\[ = \Re \left\{ \left( A_1 D(t - \tau_1) C(t - \tau_1) + \\ A_2 D(t - \tau_2) C(t - \tau_2) \exp \left( j2\pi(v_2 - v_1)t + (\phi_2 - \phi_1) \right) \times \exp \left( j(2\pi(f_0 + v_1)t + \phi_1) \right) \right) \right\} \] (3.3)

Defining the instantaneous phase difference between the two signal components,

\[ \psi(t) = 2\pi(v_2 - v_1)t + (\phi_2 - \phi_1) \] (3.4)

the output of the integrators in the DLL can be approximated as:

\[ y_\pm(t) \approx A_1 R_C(\tau_1 - \hat{\tau} \pm \delta) D(t - \tau_1) + A_2 R_C(\tau_2 - \hat{\tau} \pm \delta) \exp \left( j\psi(t) \right) D(t - \tau_2) \]

\[ \approx \left[ A_1 R_C(\tau_1 - \hat{\tau} \pm \delta) + A_2 \exp \left( j\psi(t) \right) R_C(\tau_2 - \hat{\tau} \pm \delta) \right] D(t - \tau_1). \] (3.5)

With \( \epsilon = \tau_1 - \hat{\tau} \), the envelope of \( y_\pm(t) \) is

\[ |y_\pm(t)| = \left| A_1 R_C(\epsilon \pm \delta) + A_2 \exp \left( j\psi(t) \right) R_C(\epsilon + (\tau_2 - \tau_1) \pm \delta) \right|. \] (3.6)

The output of the envelope discriminator is

\[ \epsilon(t) = |y_-(t)| - |y_+(t)|. \] (3.7)

As mentioned before, the amplitude of the sum signal depends on the relative phase \( \psi(t) \) between the direct and the reflected signal components. We consider the two special cases of constructive and destructive interference:

- **Constructive interference** In this case \( \exp(j\psi(t)) = 1 \) and

\[ |y_\pm(t)| = A_1 R_C(\epsilon \pm \delta) + A_2 R_C(\epsilon + (\tau_2 - \tau_1) \pm \delta). \] (3.8)
- **Destructive interference** Here, \( \exp(j \psi(t)) = -1 \) and, therefore,

\[
|y_{\pm}(t)| = A_1 R_C(\epsilon \pm \delta) - A_2 R_C(\epsilon + \tau_2 - \tau_1 \pm \delta).
\]  

(3.9)

Now we just know the algebraic equation for the received signal. So, the following step is to compare the multipath envelope error for the DLL correlator versus the \( \Delta \Delta \)DLL with different delayed multipath. We want to compute this envelope error with the same conditions which we have in the GPS-SDR to be able to compare it later with the real GPS-SDR response. So, we have used a 2MHz bandwidth (see Fig. 3.16), as USRP and GPS-SDR use, and the MATLAB tools. The single difference between this theoretical simulation in MATLAB and the real one, which will be done later, is that the first one is in the ideal case; this is without noise.

![2MHz Bandwidth Correlator Filter used in theoretical simulation](image)

The Figure 3.17 shows the multipath error envelope for an early/late detector (DLL) an for a \( \Delta \Delta \)DLL for C/A code both with \( \delta = 1 \). The line of sight signal has to times the power of the delayed received signal. In this case the channel response is:

\[
h_C(t) = A \delta(t) + \frac{A}{\sqrt{2}} \delta(t - t_i)
\]

(3.10)

where \( t_i \) goes from 0 to 2 times of chip(\( T_c \)).
Chapter 3. Double Delta Correlation

The positive multipath error corresponds to constructive interference while negative multipath error corresponds to destructive interference. However at the beginning both correlators have a similar response, it’s easy to see that the $\Delta\Delta$DLL gets an improvement of up 40 meters compared with the DLL correlator.

![Figure 3.17: Multipath error envelope. Theoretical comparison between DLL and $\Delta\Delta$DLL correlators ($\delta = 1$)](image)

### 3.5 $\Delta\Delta$DLL Software Implementation

As the GPS-SDR is a completely software GPS receiver, also the correlator is implemented by software. To execute the correlator algorithm it would be enough to program it in plain C. But as a real-time performance is required, and plain C is not able to achieve that requirement, the solution is to implement the software correlator with a technique which is called SIMD (Single Instruction, Multiple Data). The programming language is the assembler.

Thus, during this section, the SIMD technique and the DLL GPS-SDR implementation will be showed. Also our $\Delta\Delta$DLL multipath solution implementation will be explained.
3.5.1 SIMD

Single Instruction, Multiple Data (SIMD) is a technique employed to achieve data level parallelism, as in a vector processor. The basic unit of SIMD is the vector, which is why SIMD computing is also known as vector processing. A regular CPU operates on scalars, one at a time. (A superscalar CPU operates on multiple scalars at once, but it performs a different operation on each instruction.) A vector processor, on the other hand, lines up a whole row of these scalars, all of the same type, and operates on them as a unit.

A SIMD machine exploits a property of the data stream called data parallelism. You get data parallelism when you have a large mass of data of a uniform type that needs the same instruction performed on it, so you get a higher speed code execution.

Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD’s 3DNOW! (which had debuted a year earlier). SSE contains 70 new instructions.

Intel Notes about SIMD

Streaming SIMD Extensions 2 (SSE2) extends the MMX\textsuperscript{TM} technology and SSE technology with the addition of 144 instructions that deliver performance increases across a broad range of applications. The SIMD integer instructions introduced with MMX technology have been extended from 64 to 128 bits, doubling the effective execution rate of SIMD integer type operations. See the schedule concept at figure 3.18.

Double-precision floating point SIMD instructions allow for two floating-point operations to be simultaneously executed in the SIMD format, providing support for double-precision operations that help accelerate content creation, financial, engineering, and scientific applications.

In addition to the SSE2 instructions, the original SSE instructions have been enhanced to support a flexible and higher dynamic range of computational power by supporting arithmetic operations on multiple data types (e.g., double words and quad words). SSE2 instructions allow software developers to have maximum flexibility to implement algorithms and provide performance enhancements when running software such as MPEG-2, MP3, 3D graphics, etc.

The launch of 90 nm process-based Pentium 4 processor introduces the Streaming SIMD Extensions 3 (SSE3), which includes 13 additional SIMD instructions over SSE2. The 13 new instructions in SSE3 are primarily designed to improve thread synchronization and specific application areas such as media and gaming.

This applies to:
Figure 3.18: Single Instruction, Multiple Data concept

- Intel® Celeron® Processor Family
- Intel® Pentium® 4 Processors
- Intel® Pentium® D Processor
- Intel® Xeon® Processor
- Mobile Intel® Celeron® Processors
- Mobile Intel® Pentium® 4 Processors-M

Intel provides 6 documents describing the architecture and programming environment of the Intel 64 and 32 processors. Electronic versions of these documents are available at http://www.intel.com/products/processor/manuals/index.htm

3.5.2 SIMD DLL correlator implementation in GPS-SDR

The file simd/simd.h contains the interface definition to the x86 and SSE2 set of functions. This means that each function is coded using SSE2 instructions and also in a plain C counterpart. The appropriate function is executed depending on the capabilities of the processor.

SSE2 is an extension of assembly language which allows programs to execute one operation on multiple pieces of data at a time. SSE2 works just like any other set of assembly calls. There are registers in which data can be stored and operations that can execute on these registers. The 8 registers are named \texttt{xmm0} through \texttt{xmm7}.

In order to encapsulate the SIMD code as a C/C++ function, we will use inline assembly, which is a way of inserting assembly code into C++ source [GWH06]. The way to do that depends on the compiler. Here we will use the gcc format.
<table>
<thead>
<tr>
<th>SIMD Extension</th>
<th>Register Layout</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX Technology</td>
<td>MMX Registers</td>
<td>8 Packed Byte Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 Packed Word Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Packed Doubleword Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Quadword</td>
</tr>
<tr>
<td>SSE</td>
<td>MMX Registers</td>
<td>8 Packed Byte Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 Packed Word Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Packed Doubleword Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Quadword</td>
</tr>
<tr>
<td></td>
<td>XMM Registers</td>
<td>4 Packed Single-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Floating-Point Values</td>
</tr>
<tr>
<td>SSE2/SSE3/SSE3</td>
<td>MMX Registers</td>
<td>2 Packed Doubleword Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Quadword</td>
</tr>
<tr>
<td></td>
<td>XMM Registers</td>
<td>2 Packed Double-Precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Floating-Point Values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 Packed Byte Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 Packed Word Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 Packed Doubleword Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Packed Quadword Integers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Quadword</td>
</tr>
</tbody>
</table>

Figure 3.19: SIMD extensions, register layouts, and data types
The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an address space. The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called segments. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space.

Registers:

- **Basic program execution registers**: The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These
instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory.

- **General-purpose registers:** The eight general-purpose registers are used along with the existing IA-32 addressing modes to address operands in memory. (MMX and XMM registers cannot be used to address memory). The general-purpose registers are also used to hold operands for some SSE instructions and are referenced as EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.
  
  - EAX: Accumulator for operands and results data
  - EBX: Pointer to data in the DS segment
  - ECX: Counter for string and loop operations
  - EDX: I/O pointer
  - ESI: Pointer to data in the segment pointed to by the DS register; source pointer for string operations
  - EDI: Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
  - ESP: Stack pointer (in the SS segment)
  - EBP: Pointer to data on the stack (in the SS segment)

- **MMX registers:** The eight MMX registers are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between the MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.

- **XMM registers:** The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers. XMM registers are referenced by the names XMM0 through XMM7.

- **MXCSR register:** This 32-bit register provides status and control bits used in SIMD floating-point operations.

- **EFLAGS register:** This 32-bit register is used to record result of some compare operations.
Correlation algorithm

The correlation algorithm used by the DLL is as follows:

1. Generate the inphase carrier wipeoff vector, COS.
2. Generate the quadrature carrier wipeoff vector, SIN.
3. Generate the C/A code replica, PRN.
4. Multiply the IF by the PRN vector.
5. Multiply IF x PRN by COS, accumulate to generate the inphase correlation.
6. Multiply IF x PRN by SIN, accumulate to generate the quadrature correlation.

Assuming the PRN, COS, and SIN vectors are pregenerated, the algorithm further simplifies to:

1. Multiply PRN by IF.
2. Multiply (PRN x IF) by COS, accumulate

3. Multiply (PRN x IF) by SIN, accumulate.

**DLL Software functions**

The file simd.h contains the declaration of all the implemented functions. They are different versions of the same set, implemented in plain C (x86 instruction set) or using the SSE extensions, but as we have said previously in fact GPS-SDR uses the SSE ones.

Some of the most important functions based on SIMD are:

- **acquisition.cpp** uses
  - `sse_cmulsc(CPX*A, CPX*B, CPX*C, int32cnt, int32shift)` attribute((noinline)); Pointwise vector multiply with shift, dumps results into C
  - `sse_cmuls(CPX*A, CPX*B, int32cnt, int32shift)` attribute ((noin-line)); Pointwise vector multiply with shift
  - `sse_cacc(CPX*A, MIX*B, int32cnt, int32*iaccum, int32*baccum)` attribute ((noinline)); Compute dot product of complex and a mix vector

- **correlator.cpp** uses
  - `sse_cmulsc(CPX*A, CPX*B, CPX*C, int32cnt, int32shift)` attribute ((noin-line)); Pointwise vector multiply with shift, dumps results into C
  - `sse_prn_accum_new(CPX*A, MIX*E, MIX*P, MIX*L, int32cnt, CPXACCUM*accum)`

And the SIMD functions used by the DLL correlator are **sse_cmulsc** and **sse_prn_accum_new**.

The `sse_cmulsc()` function performs pointwise vector multiply with shift, and dumps result into vector C. It will multiply cnt elements of vector A by cnt elements of vector B, element by element, and the result will be stored in C. In fact, this function implements the multiplication of the IF x PRN by COS/SIN. Since we are using the SSE2 set of instructions, we will use the XMM registers (128 bits). The assembler `sse_cmulsc` code is as follows:

```c
void sse_cmulsc(CPX *A, CPX *B, CPX *C, int32 cnt, int32 shift) {
```
int32 cnt1;
int32 cnt2;
int32 round;

volatile int32 M[4] = {0xffff0001, 0x00010001,
0xffff0001, 0x00010001}; // {1, -1, 1, 1, -1, 1, 1, 1};

cnt1 = cnt / 4;
cnt2 = cnt - 4 * cnt1;

round = 1 << (shift - 1);

__asm
{
  " .intel_syntax noprefix \n" // Set up for loop
  " mov edi, [ebp+8] \n" // Address of A
  " mov esi, [ebp+12] \n" // Address of B
  " mov eax, [ebp+16] \n" // Address of C
  " mov ecx, [ebp-12] \n" // Counter 1
  " movupdx xmm7, [ebp-36] \n" // Move the multiply thingie
  " movss xmm6, [ebp+24] \n" // Move the round thingie
  " movss xmm5, [ebp-20] \n" // Move the round thingie
  " punpckldq xmm5, xmm5 \n" // punpckldq xmm5, xmm5
  " jecxz Z% = \n" // jecxz Z%=
  " L% =: \n"
  " movlpd xmm0, [edi] \n" // Copy from A
  " movlpd xmm1, [edi+8] \n" // Copy from A
  " movlpd xmm3, [esi] \n" // Copy from B
  " movlpd xmm4, [esi+8] \n" // Copy from B
  " punpckldq xmm0, xmm0 \n" // Copy low 32 bits to high 32 bits
  " punpckldq xmm1, xmm1 \n" // Copy low 32 bits to high 32 bits
  " punpckldq xmm3, xmm3 \n" // Copy low 32 bits to high 32 bits
  " punpckldq xmm4, xmm4 \n" // Copy low 32 bits to high 32 bits
  " pshufhw xmm3, xmm3, 0x14 \n" // [Re Im Im Re]
  " pshufhw xmm4, xmm4, 0x14 \n" // [Re Im Im Re]
  " pmullw xmm7, xmm3 \n" // Multiply to get [Re Im -Im Re]
  " pmullw xmm7, xmm4 \n" // Multiply to get [Re Im -Im Re]
  " pmaddwd xmm0, xmm3 \n" // Complex multiply and add
Chapter 3. Double Delta Correlation

```assembly
"pmaddwd xmm1, xmm4 \n" // Complex multiply and add
" paddd xmm0, xmm5 \n" // Add in 2^(shift-1)
" paddd xmm1, xmm5 \n" // Add in 2^(shift-1)
"psrad xmm0, xmm6 \n" // Shift by X bits
"psrad xmm1, xmm6 \n" // Shift by X bits
"packssdw xmm0, xmm0 \n" // Get into low 64 bits
"packssdw xmm1, xmm1 \n" // Get into low 64 bits
"movlpd [eax], xmm0 \n" // Move into A
"movlpd [eax+8], xmm1 \n" // Move into A
"add edi, 16 \n" // Move in array
"add esi, 16 \n" // Move in array
"add eax, 16 \n"

".att_syntax \n"
".intel_syntax noprefix \n"

"Z%:=; \n"
"mov ecx, [ebp-16] \n"

".att_syntax \n"
".intel_syntax noprefix \n"

"Z%:=; \n"
"mov ecx, [ebp-16] \n"
"jecxz ZZ%:= \n"

".att_syntax \n"
".intel_syntax noprefix \n"

"EMMS \n"
".att_syntax \n"
```

The equivalent code in plain C would be:

```c
void x86__cmulsc(CPX *A, CPX *B, CPX *C, int32 cnt, int32 shift) {
    int32 lcv;
    int32 ai, aq;
    int32 bi, bq;
    int32 ti, tq;
    int32 shift;
    int32 round;
    shift = _shift;
    round = 1 << (shift-1);
    for(lcv = 0; lcv < cnt; lcv++) {
        ai = A[lcv].i;
        aq = A[lcv].q;
        bi = B[lcv].i;
        bq = B[lcv].q;
        ti = ai*bi - aq*bq;
        tq = ai*bq + aq*bi;
        ti += round;
        tq += round;
        ti >>= shift;
        tq >>= shift;
        C[lcv].i = (int16)ti;
        C[lcv].q = (int16)tq;
    }
}
```

So, in fact, as we call the `sse_cmulsc` at `/objects/correlator.cpp` as:

```c
void sse_cmulsc(data, state.psine, scratch, samps, 14)
```
Chapter 3. Double Delta Correlation

What we are doing is multiply in complex mode the data by sin/cos and store the results at scratch, which will be used as data at the next function sse_prn_accum_new().

After multiplying the input samples by COS/SIN, now it’s time to apply the DLL algorithm. The function sse_prn_accum_new multiplies the samples given by the previous function (A) by the early (E), prompt (P) and late (L) samples and accumulate them separately in each buffer (ACCUM). The sse_prn_accum_new code is:

```c
void sse_prn_accum_new(CPX *A, MIX *E, MIX *P, MIX *L, int32 cnt, CPX_ACCUM *accum)
{
    __asm
    (.intel_syntax noprefix \n"
        // Set up for loop
        mov esi, [ebp+8] \n" // Address of A
        mov eax, [ebp+12] \n" // Address of E
        mov ebx, [ebp+16] \n" // Address of P
        mov edx, [ebp+20] \n" // Address of L
        mov ecx, [ebp+24] \n" // Value of cnt
        pxor mm5, mm5 \n" // Clear the running sum for E
        pxor mm6, mm6 \n" // Clear the running sum for P
        pxor mm7, mm7 \n" // Clear the running sum for L
        jecxz Z%= \n"
        // Clear the running sum for L
        movd mm0, [esi] \n" // load IF data
        movq mm1, [eax] \n" // load E data
        movq mm2, [ebx] \n" // load P data
        movq mm3, [edx] \n" // load L data
        punpckldq mm0, mm0 \n" // copy low 32 bits to high 32 pits
        pmaddwd mm1, mm0 \n" // complex multiply E by IF
        pmaddwd mm2, mm0 \n" // complex multiply P by IF
        pmaddwd mm3, mm0 \n" // complex multiply L by IF
        paddd mm5, mm1 \n" // add into E accumulator
        paddd mm6, mm2 \n" // add into E accumulator
        paddd mm7, mm3 \n" // add into E accumulator
        add esi, 4 \n" // move in baseband data by one sample (4 bytes)
        add eax, 8 \n" // move in PRN-E array by one sample (8 bytes)
        add ebx, 8 \n" // move in PRN-P array by one sample (8 bytes)
        add edx, 8 \n" // move in PRN-L array by one sample (8 bytes)
        .att_syntax \n"
        loop L%= \n"
    .intel_syntax noprefix \n"
    
```
Chapter 3. Double Delta Correlation

\[
Z^\# = \text{n} \text{\n} \text{t}
\]
\[
\text{mov}, \text{esi}, [ebp+28] \text{n} \text{\n} \text{t}
\]
\[
\text{movq}, [esi], mm5 \text{n} \text{\n} \text{t}
\]
\[
\text{add}, \text{esi}, 8 \text{n} \text{\n} \text{t}
\]
\[
\text{movq}, [esi], mm6 \text{n} \text{\n} \text{t}
\]
\[
\text{add}, \text{esi}, 8 \text{n} \text{\n} \text{t}
\]
\[
\text{movq}, [esi], mm7 \text{n} \text{\n} \text{t}
\]
\[
\text{EMMS} \text{n} \text{\n} \text{t}
\]
\[
\text{.att_syntax} \text{n} \text{\n} \text{t}
\]
\[
:\text{ "m" (A), "m" (E), "m" (P), "m" (L), "m" (cnt), "m" (accum)
\]
\[
:\text{ "%eax", "%ebx", "%ecx", "%edx", "%esi"
\]
\]
\[
}; \text{end__asm}
\]

His plain C implementation would be:

```c
void x86_prn_accum_new (CPX *A, MIX *E, MIX *P,
MIX *L, int32 cnt, CPX_ACCUM *accum)
{
CPX_ACCUM Ea, Pa, La;
int32 lcv;
Ea.i = 0; Ea.q = 0;
Pa.i = 0; Pa.q = 0;
La.i = 0; La.q = 0;
for (lcv = 0; lcv < cnt; lcv++){
Ea.i += A[lcv].i*E[lcv].i;
Ea.q += A[lcv].q*E[lcv].ni;
Pa.i += A[lcv].i*P[lcv].i;
Pa.q += A[lcv].q*P[lcv].ni;
La.i += A[lcv].i*L[lcv].i;
La.q += A[lcv].q*L[lcv].ni;
}
accum[0].i = Ea.i;
accum[0].q = Ea.q;
accum[1].i = Pa.i;
accum[1].q = Pa.q;
accum[2].i = La.i;
accum[2].q = La.q;
}```
As we call `sse_prn_new_accum` function at `/objects/correlator.cpp` as:

```c
sse_prn_accum_new(scratch, state.pcode[0], state.pcode[1], state.pcode[2], samps, &EPL[0]);
```

Where:
- `scratch`: is the data input taken from `sse_cmulsc`
- `state.pcode[0]`: is the early
- `state.pcode[1]`: is the prompt
- `state.pcode[2]`: is the late
- `EPL[]`: is the buffer here we store the accumulations of the early, late and prompt.

Therefore, after executing the function `sse_prn_new_accum` we have stored to EPL:

<table>
<thead>
<tr>
<th>EPL[]</th>
<th>Accumulation stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPL[0].i</td>
<td>Re{E}</td>
</tr>
<tr>
<td>EPL[1].i</td>
<td>Re{P}</td>
</tr>
<tr>
<td>EPL[2].i</td>
<td>Re{L}</td>
</tr>
<tr>
<td>EPL[0].q</td>
<td>Im{E}</td>
</tr>
<tr>
<td>EPL[1].q</td>
<td>Im{P}</td>
</tr>
<tr>
<td>EPL[2].q</td>
<td>Im{L}</td>
</tr>
</tbody>
</table>

Table 3.2: EPL `sse_prn_new_accum` storage

So now it is easy to compute the early, late and prompt power as:

- \( P[0]=I[0]^2+Q[0]^2 \); // P(E)

And the DLL code error is calculated as:

```c
ep1=sqrt(float(P[0]));
prompt=sqrt(float(P[1]));
lp1=sqrt(float(P[2]));
// Code error with Double Delta
code_err=(ep1-lp1)/(ep1+lp1);
```
3.5.3 SIMD ΔΔDLL correlator implementation

The correlation algorithm used for the ΔΔDLL correlator is the same than the used by the DLL. So, assuming the PRN, COS, and SIN vectors are pregenerated, the algorithm further simplifies to:

1. Multiply PRN by IF.
2. Multiply (PRN x IF) by COS, accumulate
3. Multiply (PRN x IF) by SIN, accumulate.

Where the IF×PRN by COS/SIN is the same operation than before, so function sse_cmulsc is not necessary to be modified for the ΔΔDLL implementation.

As we have seen at section 3.3, ΔΔDLL correlator uses two early’s, two late’s and one prompt. Accordingly to this specifications what we have done is to modify the function sse_prn_new_accum to be able to use more early’s and late’s. So we have created the function sse_prn_new_accum_dd. This function does the same as sse_prn_new_accum, but it multiplies the samples given by sse_cmulsc function (A) by the first and the second early (E1 and E2), prompt (P) and the first and the second late (L1 and L2) samples and accumulate them separately in each buffer (ACCUM). The function’s sse_prn_new_accum_dd code is as follows:

```c
void sse_prn_accum_new_dd(CPX *A, MIX *E1, MIX *E2, MIX *P, MIX *L1, MIX *L2, int32 cnt, CPX_ACCUM *accum)
{
    int32 mem_esp;
    __asm
    {
        "\t.intel_syntax noprefix\n\t //Set up for loop
        "\tmov[ebp-16], esp\n\t //Address of A
        "\tmov[ebp+8]\n\t //Address of E1
        "\tmov[ebp+12]\n\t //Address of E2
        "\tmov[ebp+20]\n\t //Address of P
        "\tmov[ebp+24]\n\t //Value of L1
        "\tmov[ebp+28]\n\t //Value of L2
        "\tmov[ebp+32]\n\t //Value of cnt
        /* clean registers */
        "\tpxor xmm5, xmm5\n\t //Clear the running sum for E
        "\tpxor mm6, mm6\n\t //Clear the running sum for P
```
"pxor xmm6, xmm6" // Clear the running sum for L
"jecxz Z%=
"L%:"
"movd mm0, [esi]" // load IF data (A) mm0=[A]=[0 0 Aq Ai]
"movq mm1, [ebx]" // where A has 32 bits
"movq mm2, [eax]" // load E1 data mm1=[E1]
"movq mm3, [edx]" // mm2=[E2]
"movq mm4, [esp]" // load L1 data mm4=[L1]
"punpckldq mm0, mm0" // copy low 32 bits to
"movq2dq xmm1, mm1" // high 32 bits mm0=[Aq Ai Aq Ai]
"movq2dq xmm2, mm2" // xmm1=[0 E1]
"movq2dq xmm3, mm3" // xmm2=[0 E2]
"movq2dq xmm4, mm4" // xmm3=[0 L1]
"movq2dq xmm0, mm0" // xmm4=[0 mm0]
"pshufd xmm1, xmm1, 0x4E" // xmm1=[E1 0]
"movq2dq xmm4, mm5" // xmm2=[0 L2]
"punpckldq xmm0, xmm0" // xmm3=[A A]
"paddq xmm1, xmm2" // xmm3=[Aq Ai Aq Ai]
"paddq xmm1, xmm2" // xmm4=[E1 E2]=
"pshufd xmm3, xmm3, 0x4E" // xmm5=[L1 L2]
"pmaddwd xmm1, xmm0" // xmm3=[L1 L2]=
"pmaddwd xmm1, xmm0" // xmm3=[A A]
"add esi, 4" // move in baseband data by one sample(4B)
"add ebx, 8" // move in PRN-E1 array by one sample (8B)
"add eax, 8" // move in PRN-E2 array by one sample (8B)
"pmaddwd xmm3, xmm0" // MM3=[AIPA+AIQP AQQPQ+APIQ]
"paddq xmm5, xmm1" // xmm5=[sum(E1*A) sum(E2*A)]
"paddq xmm5, xmm1" // IDEM XMM1 WITH LATE
"add edx, 8" // move in PRN-P array by one sample (8B)
"add esp, 8" // move in PRN-L1 array by one sample (8B)
"add edi, 8" // move in PRN-L2 array by one sample (8B)
"paddq xmm5, xmm3" // xmm6=[sum(P*A)]
"paddq xmm5, xmm3" // xmm6=[sum(L1*A) sum(L2*A)]
As we call `sse_prn_new_accum_dd` function at `/objects/correlator.cpp` as:

```cpp
sse_prn_accum_new_dd(scratch, state.pcode[0], state.pcode[1], state.pcode[2], state.pcode[3], state.pcode[4], samps, &EPL[0]);
```

Where:

- `scratch`: is the data input taken from `sse_cmulsc`
- `state.pcode[0/1]`: are the early 1 and 2 respectively
- `state.pcode[2]`: is the prompt
- `state.pcode[3/4]`: are the late 1 and 2 respectively
- `EPL[]`: is the buffer here we store the accumulations of the early’s, late’s and prompt.

Therefore, after executing the function `sse_prn_new_accum_dd` we have stored to EPL:
So now it is easy to compute the early, late and prompt power as:

\[
\begin{align*}
\text{P}[0] &= I[0]*I[0] + Q[0]*Q[0]; \quad // (E1) \\
\text{P}[1] &= I[1]*I[1] + Q[1]*Q[1]; \quad // (E2) \\
\text{P}[3] &= I[3]*I[3] + Q[3]*Q[3]; \quad // (L1) \\
\end{align*}
\]

And the $\Delta\Delta$ DLL code error is calculated as:

\[
\text{ep1} = \sqrt{\text{float} (\text{P}[0])}; \\
\text{ep2} = \sqrt{\text{float} (\text{P}[1])}; \\
\text{prompt} = \sqrt{\text{float} (\text{P}[2])}; \\
\text{lp1} = \sqrt{\text{float} (\text{P}[3])}; \\
\text{lp2} = \sqrt{\text{float} (\text{P}[4])};
\]

\[
// \text{Code error with Double Delta} \\
\text{float a=2, b=1;}
\]

\[
\text{code\_err} = (((a*(\text{ep1}-\text{lp1}))/((\text{ep1}+\text{lp1})))-((b*(\text{ep2}-\text{lp2}))/((\text{ep2}+\text{lp2}))));
\]
Chapter 4

The GPS Toolkit: GPSTk

According to the founders of the toolkit, the goal of the GPSTk project is to provide an open source library and suite of applications to the satellite navigation community to free researchers to focus on research, not lower level coding. The GPSTk suite consists of a core library, auxiliary libraries, and a set of applications. The GPSTk provides a wide array of functions that solve processing problems associated with GPS such as processing or using standard formats such as RINEX. The libraries are the basis for the more advanced applications distributed as part of the GPSTk suite.

The GPSTk is sponsored by Space and Geophysics Laboratory, within the Applied Research Laboratories at the University of Texas at Austin (ARL:UT) [GPS].

4.1 The GPSTk core library and applications

The GPSTk core library provides the most robust, broadly useful, and platform independent code in the GPSTk. It provides a number of models and algorithms found in GPS textbook and classic papers, such as solving for the user position or estimating atmospheric refraction. Common formats are supported as well, such as RINEX or SP3. There are several categories of function that provide the base functionality for the GPSTk applications and for a number of other independent projects:

1. **GPS time.** Conversion among time representations such as MJD, GPS week and seconds of week, and many others.

2. **Ephemeris calculations.** Position and clock interpolation for both broadcast and precise ephemeris.
3. **Atmospheric delay models.** Includes ionosphere and troposphere models.

4. **Position solution.** Includes an implementation of a Receiver Autonomous Integrity Monitoring algorithm.

5. **Mathematics.** Includes Matrix and Vector implementations, as well as interpolation and numerical integration.

6. **GNSS data structure.** Data structures that contain observations mapped to satellites.

7. **Application framework.** Includes processing command lines options, providing interactive help and working with file systems.

The GPSTk Core Library and its associated test programs can be built independently of building the GPSTk Applications or Auxiliary Libraries. The GPSTk Core Library source code contains no dependencies outside of the GPSTk Core Library and Standard C++ and will build cleanly on all supported platforms.

The libraries are the foundation for applications within the GPSTk suite. The applications support greater depth of functionality to support research and development. The applications are almost entirely console based (i.e., without a graphical user interface). They can be grouped functionally into a number of categories.

1. **RINEX utilities:** The RINEX utilities provide a set of applications that can be used to examine, manipulate, and plot RINEX observation files.

2. **Positioning:** The positioning applications include two different applications that perform standard pseudorange-based positioning and two that implement differential phase-based solutions.

3. **Residual analysis:** A residual analysis application computes two types of measurement residuals using a single receiver or two receivers in a zero baseline configuration.

4. **Ionospheric modeling:** The ionospheric modeling applications utilize the two frequency TEC estimate from the RINEX utilities and compute a model of the ionosphere.

5. **Signal Tracking Simulation:** These utilities simulate the tracking of GPS C/A and P-code.

6. **Basic transformations:** Conversions of time and coordinate systems.

7. **Observation data collection and conversion:** Translating receiver specific data formats to RINEX.

8. **File comparison and validation:** Differing observations files against a truth source.
9. **Data editing**: Simple editing like systematic removal of observations by satellite, type or time and more advanced editing like cycle slip detection and correction.

10. **Autonomous and relative positioning**: Navigation and surveying applications.

### 4.2 The GPSTk in GPS-SDR

As we have seen previously the GPSTk has lots of applications, however this we have used basically two of them to improve the GPS-SDR’s features and aptitudes. These are the RINEX and the Positioning utilities.

#### 4.2.1 GPStk–GPS-SDR Connection

To be able to use the GPStk utilities with the GPS-SDR, first of all we have to design and implement a connection between both platforms. The connection between them is based in the RINEX files exchange (navigation and observation files). So we have added to GPS-SDR an output RINEX files module which are read by the GPStk module.

The simplified steps to achieve the navigation and observation files is explained at following the paragraphs. Also an example from each file is given.

Taking the advantage that, of course, the GPS-SDR receives the GPS navigation message and calculates the satellite-user pseudorange we have followed the next steps to get the navigation and observation RINEX file output (See 1.4 RINEX Files to know the RINEX files standard schedule).

#### Getting the observation file

In fact, we only need two important things to build a RINEX observation file: the pseudorange and the GPS time. So the steps are:

1. Load the satellite-user pseudorange from the GPS-SDR structure \( \text{Pseudorange } S \).
2. Get the GPS time from the GPS navigation message.
3. Write the observation file header.
4. Write to the observation file the epoch, pseudorange and satellite identification when the GPS could track more than four satellites fulfilling the RINEX standard.
The Figure 4.1 shows an observation RINEX file example obtained with the GPS-SDR.

<table>
<thead>
<tr>
<th>2.10</th>
<th>Observation</th>
<th>G (GPS)</th>
<th>RINEX VERSION / TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTTC</td>
<td>MARKER NAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPQ SOLE</td>
<td>CTTC</td>
<td>OBSERVER / AGENCY</td>
<td></td>
</tr>
<tr>
<td>GPS-SDR-DU</td>
<td>GNURADIO</td>
<td>2.0</td>
<td>REC # / TYPE / VERS</td>
</tr>
<tr>
<td>0000000000</td>
<td>CTTC000000.00</td>
<td>ANF # / TYPE</td>
<td></td>
</tr>
<tr>
<td>4707862.2700</td>
<td>168436.1500</td>
<td>41.635804.6979</td>
<td>APPROX POSITION XYZ</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>ANTENNAS: DELTA H/E/N</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000000000</td>
<td>WAVELENGTH FACT 1/2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>L1</td>
<td># / TYPES OF OBSERV</td>
</tr>
<tr>
<td>0.1000</td>
<td></td>
<td></td>
<td>INTERVAL</td>
</tr>
<tr>
<td>2007</td>
<td>05</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MARKER NUMBER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TIME OF FIRST OBS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>END OF HEADER</td>
</tr>
</tbody>
</table>

```
07 05 21 1 0 48.9248806 0 8202 86 20 4G13025025027
23042661.929 143416.988 7
21305944.425 302800.893 8
22555129.889 323333.478 7
22609296.032 234946.767 9
22684790.069 145156.259 9
22885028.963 205076.345 8
21732864.350 140190.593 7
20759916.338 227712.917 8
07 05 21 1 0 48.9248803 0 8202 86 20 4G13025025027
25042667.015 143712.717 8
21309513.464 303516.141 8
22561904.941 324053.852 7
22609394.479 235496.377 8
22856489.462 140703.073 9
22885907.019 205157.256 8
21726127.461 140505.990 7
20759326.273 220206.927 8
```

Figure 4.1: GPS-SDR RINEX Observation File

Getting the navigation file

The followed steps are:

1. Load the navigation ephemeris from each satellite to the structure *Ephemeris S*
2. Get the GPS time from the GPS navigation message.
3. Write the navigation file header.
4. Write every satellite ephemeris into the navigation RINEX file.

The Figure 4.2 shows a navigation RINEX file example obtained with the GPS-SDR.
GPS-SDR Software modifications

To achieve the connection between both platforms we have had to modify some parts of the original GPS-SDR code. Our changes are based on the file /objects/telemetry.cpp. Therefore we have created basically four functions which are: Rinex2NavHeader(), Rinex2ObsHeader(), logrinex2nav(), logrinex2obs(); whose functions are created the RINEX navigation and observation files.

Rinex2NavHeader() and Rinex2ObsHeader() functions have both the same function, they write the RINEX navigation and observation header to the navigation and observation RINEX files respectively. These functions code is as follows:

```c
void Telemetry::Rinex2NavHeader()
{

    if(fp_rin2 != NULL)
    {
        //calculate UTC_TIME
        time_t tiempo;
        char cad[80];
        struct tm *tmPtr;
        tiempo = time(NULL);
        tmPtr = localtime(&tiempo);
        strftime(cad, 20, "%d-%b-%y %H:%M", tmPtr);
    
```
Apart from the files headers, we need to write the RINEX files body. As both files have different functions, their bodies are not very similar. The navigation file body stores each satellites ephemeris. Thus, to write the navigation body we need the epoch, the satellite identification and finally the ephemeris. The navigation file code is as follows:

```c
void Telemetry::LogRinex2Nav(){
    Ephemeris_S *efemerides;
    Nav_Solution_S *pNav=tNav.master_nav; /* Navigation Solution */
    Clock_S *pClock=tNav.master_clock; /* Clock solution */
    char packet[80];
    int index=0;
    int lcv;
    char cad[2];
    //Get ephemeris
    for(lcv=0; lcv<NUM_CODES;lcv++){
        efemerides=(Ephemeris_S*) &tEphem.ephem2telem[lcv];
        //Compute UTC time
        if(fp_rin2 != NULL){
            double decimalday, daydecimalhour, decimalhour,decimalmin,
                        decimalsec;
            double day,hour,minutes,seconds,enterseconds,a;
            double past_second,b,c,enter_past_second=1000;
            double gpstime;
            //1-Compute date and gps time
            //I compute year, month an day from UTC time
            //calculate UTC_TIME
            time_t tiempo;
```
char cad1[80];
char cad2[80];
char cad3[80];
struct tm *tmPtr;
tmPtr = gmtime(&tiempo);
strftime(cad1,20, "%y%m%d", tmPtr );/*

//calculate date of gps time:
time_t temps;
double setmanes=efemerides->week_number+1024;
//Days & weeks between 00h 1 Jan 1970 and 00h 6 Jan 1980
//520 weeks and 12 days.
temps=(520+setmanes)*7*24*3600+gpstime+17*24*3600;
tmPtr = gmtime(&temps);
strftime(cad1,20, "%y%m%d", tmPtr );

//Compute hour, minut, second from pClock.time= hour GPS
gpstime=pClock->time;
decimalday=((gpstime)/(24*3600)); //Day of the week
daydecimalhour=modf(decimalday,&day); //day=#enter days,
daydecimalhour=daydecimalhour*24; //portion day in hours
decimalhour=modf(daydecimalhour,&hour); //hour=day hour;
decimalhour= Hour portion
decimalmin=decimalhour*60; //decimalmin=day minuts decimal
decimalsec=modf(decimalmin,&minutes);
//minutes=enter day minuts,decimalsec=minut portion
seconds=decimalsec*60; //seconds=day second decimal
a=modf(seconds,&enterseconds);
sprintf(cad2,"%2.0f%2.0f%3.1f",hour,minutes,enterseconds);
data.year= 2007;
data.month=5;
data.day= 21;
data.hour=hour;
data.minut=minutes;
data.second=seconds;
if(cad2[0]=='\n')cad2[0]='0';
if(cad2[3]=='\n')cad2[3]='0';
if(cad2[6]=='\n')cad2[6]='0';
// Write Ephemris
if (pNav->converged) {
    if (correccio_primera_obs == 1) {
        // Write time correction a0 & a1
        if (efemerides->valid) {
            fseek(fp_rin2, fp_rin_end2, SEEK_SET);
            char correction[256], correction2[256];
            fprintf(fp_rin2, "\%s%s\%9.0f\%9d
DELTA-UTC: A0,A1,T,W\n", correction, correction2, gpstime, (efemerides->week_number + 1024));
            fprintf(fp_rin2, "\%015
LEAP SECONDS\n");
            fprintf(fp_rin2, "\%015
END OF HEADER\n");
            fp_rin_end2 = ftell(fp_rin2);
            correccio_primera_obs = 0;
        }
    }
    if (efemerides->valid == true) {
        // preparacio lines de efemerides per imprimir!!!
        char linia0[256], linia1[256], linia2[256], linia3[256],
        linia4[256], linia5[256], linia6[256], linia7[256];
        char idef[256];
        sprintf(idef, "\%2.0d\", efemerides->sv + 1);

        sprintf(linia0, "\%19.12E\%19.12E\%19.12E", efemerides->af0, efemerides->af1, efemerides->af2);

                double(efemerides->iode), efemerides->crs,
                efemerides->deltan, efemerides->m0);

                efemerides->sqrta);

                efemerides->cis);

                efemerides->argp2);
Chapter 4. The GPS Toolkit: GPStk

efemerides->omd;

sprintf(linia5,"%19.12E%19.12E%19.12E%19.12E",
   double(efemerides->idot),double(efemerides->code_on_L2),
   double(efemerides->week_number+1024),
   double(efemerides->L2pdata));

   double(efemerides->ura), double(efemerides->subframe_1_health),
   efemerides->tgd, double(efemerides->iodc));

sprintf(linia7,"%19.12E%19.12E", double(efemerides->tow),
   double(efemerides->fti));

fseek(fp_rin2,fp_rin_end2,SEEK_SET);
fprintf(fp_rin2, "%s%s%s%s
", idef,cad1,cad2,linia0);
fprintf(fp_rin2, "%s
", linia1);
fprintf(fp_rin2, "%s
", linia2);
fprintf(fp_rin2, "%s
", linia3);
fprintf(fp_rin2, "%s
", linia4);
linia5[16]='
';
fprintf(fp_rin2, "%s
", linia5);
fprintf(fp_rin2, "%s
", linia6);
printf(fp_rin2, "%s
", linia7);
fp_rin_end2 = ftell(fp_rin2);
}}}
}

At the same time, when we are writing the navigation file, we have to write the observation file. The code proposed for the function Rinex2ObsHeader() and the function logrinex2obs() is as follows.

```c
void Telemetry::Rinex2ObsHeader()
{
    Clock_S *pClock = &tNav.master_clock; /* Clock solution */
    if(fp_rin != NULL)
    {
      //calculate UTC_TIME
      time_t tiempo;
      char cad[80];
      struct tm *tPtr;
```
tiempo = time(NULL);
tmPtr = gmtime(&tiempo);
strftime(cad, 24, "%d/%m/%Y\%H:%M:%S", tmPtr);
fseek(fp_rin, fp_rin_end, SEEK_SET);

fprintf(fp_rin,"\n2.10 Observation
  G\(GPS\) RINEX_VERSION_/\ TYPE\n");
fprintf(fp_rin,"GPS-SDR-DUO\ CTTC
  %s PGM_/\ RUN_BY_/\ DATE\n", cad);
fprintf(fp_rin,"\nCTTC MARKER_NAME\n");
fprintf(fp_rin,"MARC_SOLE\ CTTC
  OBSERVER_/\ AGENCY\n");
fprintf(fp_rin,"GPS-SDR-DUO\ GNURADIO
  2.0 REC_/\ TYPE_/\ VERS\n");
fprintf(fp_rin,"000000000000\ CTDC00000.00
  ANT_/\ TYPE\n");
fprintf(fp_rin,"4797642.2790 166436.1500
  APPROX_POSITION_XYZ\n");
fprintf(fp_rin,"0.0000 0.0000 0.0000\ ANTENNA:\ DELTA_H/E/N\n");
fprintf(fp_rin,"\n1 WAVELENGTH_FACT_L1/2\n");
fprintf(fp_rin,"\n2 C1 L1 #\ TYPES_OF_OBSERV\n");
fprintf(fp_rin,"\n0.1000 INTERVAL\n");

// annotate which is first observation
fp_rin_end = ftell(fp_rin);
temps_primera_obs = 1;
}

void Telemetry::LogRinex2Obs()
{
  Ephemeris_S * efemerides;
  Nav_Solution_S *pNav= &tNav.master_nav; /* Navigation Solution */
  Clock_S *pClock = &tNav.master_clock; /* Clock solution */
  Pseudorange_S *pPseudo;
  Chan_Packet_S *pChan;
  Measurement_S *pMeas;
float cn0;
int ss;
char sat_vis[36];
for(int i=0;i<36;i++) sat_vis[i] = '\n';
char packet[80];
int index=0;
int lcv;
char cad[2];
double setmanes;

// I need
// 1- Date & hour <--- from struct Clock_S
// 2- #sat visibles, sv iden from (Chan_Packet_S *)&tChan[lcv]->sv
// 3- sv pseudodistance as LogPseudo()

if(fp_rin != NULL)
{
    if(pNav->converged)
    {
        for(lcv=0; lcv<NUM_CODES; lcv++)
        {
            efemerides = (Ephemeris_S *) &tEphem.ephem2telem[lcv];
            if(efemerides->valid) setmanes = efemerides->week_number + 1024;
        }
        // 1- Compute date and hour GPS
        // Compute the year, month and day from UTC
        // calculate UTC_TIME
        time_t temps;
        char cad1[80];
        char cad2[80];
        char cad3[80];
        char cad4[80];
        struct tm *tmPtr;
        // Compute hour, minute, seconds from pClocK.time = hour GPS
double decimalday, daydecimalhour, decimalhour,
decimalmin, decimalsec;
double day, hour, minutes, seconds, enterseconds, a;
double past_second, b, c, enter_past_second = 1000;
double gpstime;
gpstime = pClock->time;
// calculate date of GPS time:
// Days & weeks between 00h 1 Jan 1970 and 00h 6 Jan 1980
// 520 weeks and 12 days.

temps = (520 + setmanes) * 7 * 24 * 3600 + gpstime + 17 * 24 * 3600;
tmPtr = gmtime(&temps);
strftime(cad1, 20, "%Y%m%d", tmPtr);
strftime(cad2, 20, "%Y%m%d", tmPtr);

decimalday = ((gpstime) / (24 * 3600));

// days in week
daydecimalhour = modf(decimalday, &day);
// day=#enter days, daydecimalhour=portion of day
daydecimalhour = daydecimalhour * 24;

// portion of day in hours

decimalhour = modf(daydecimalhour, &hour);
// hour=day hour; decimalhour=portion

decimalmin = decimalhour * 60;
// decimalmin=decimal minute days

decimalsec = modf(decimalmin, &minutes);
// minutes=enter minutes day; decimalsec=portion minutes

seconds = decimalsec * 60; // seconds=decimal day seconds

a = modf(seconds, &enterseconds);

sprintf(cad4, "%6.0f%6.0f%13.7f", hour, minutes, seconds);

double offset;
offset = pClock->bias;

if (temps_primera_obs == 1) {

// Write first observation time
fseek(fp_rin, fp_rin_end, SEEK_SET);
fprintf(fp_rin, "%s
GPS TIME OF FIRST OBS
", cad2, cad4);
fprintf(fp_rin, "00000CTTC MARKER NUMBER
");
fprintf(fp_rin, "END OF HEADER
");
fp_rin_end = ftell(fp_rin);
temps_primera_obs = 0;
primeravegada = 0;
}

//2-Num visible sat and identifier
int32 nsvs = 0;
for(lcv = 0; lcv < MAX_CHANNELS; lcv++)
{
    //FORMAT RINEX2
    pChan = (Chan_Packet_S *) &tChan[lcv];
    if((pNav->nsvs >> lcv) & 0x1){
        nsvs ++;
    }
}

//FORMAT RINEX2
sprintf(cad,"%2.0f",(pChan->sv)+1);
if((pChan->sv>=0) && (pChan->sv<36)){
    int k=3*index;
    sat_vis[k]='G';
    sat_vis[k+1]=cad[0];
    sat_vis[k+2]=cad[1];
    index ++;
}

// format RINEX2
sprintf(packet,"%s%s_0%3d%s",cad1,cad3,nsvs,sat_vis);
packet[69]=packet[68];
packet[68]='\';
fseek(fp_rin, fp_rin_end, SEEK_SET);
fprintf(fp_rin,"%s\n",packet);
fp_rin_end = ftell(fp_rin);

//3-Write pseudodistance
for(lcv = 0; lcv < MAX_CHANNELS; lcv++)
{
    pChan = (Chan_Packet_S *) &tChan[lcv];
pPseudo = (Pseudorange_S *) &tNav.pseudoranges[lcv];
pMeas = (Measurement_S *) &tNav.measurements[lcv];
cn0 = pChan->CN0 > pChan->CN0_old ? pChan->CN0 : pChan->CN0_old;
ss=signalstrength(cn0);
if (pPseudo->meters!>0.00 ){
After the implementation of the Rinex2NavHeader(), Rinex2ObsHeader(), logrinex2nav(), logrinex2obs() functions, we have provided to the GPS-SDR an extractor RINEX module, which allows us to connect it with GPStk and use their RINEX and positioning utilities.

4.2.2 GPSTk utilities and GPS-SDR

The first feature added to the GPS-SDR related with the GPS Toolkit is the RINEX file output. As we have explained before, these RINEX files allows us to connect both platforms.

The second feature added to the GPS-SDR related with GPSTk, and probably the most important, is the Positioning utilities. These set of tools allow to the receptor get the position using differents algorithms as LMS, WLMS, Kalman, etc. which will be defined later.

To achieve this purpose it is necessary to understand as the GPSTk can organize the data and as it process them as it is explained in [Sal].

GNSS Data Management in the GPSTk

GPSTk includes hierarchy based on one kind of data structures added in order to easily cope with frequent data management situations that were very difficult to deal with when using just vectors and matrices, as was often the case. This set of data structures are called GNSS Data Structures (GDS). The GDS structures is capable to keep different types of data related with GNSS. GDS store this data indexed by very useful indexes like: station, epoch, satellite an type. Thanks to this indexes the data organization is well known and data management issues are properly addressed.

GDS Implementation

In GPStk the GDS structures are always modeled in the same way. So GDS index each data with four different indexes: SourceID, SatID, TypeID, and DayTime. These indexes are im-
implemented as C++ classes in the GPStk libraries. Thanks to GDS and their indexing mode the objects which are associated with these classes provide the GNSS programmer with a large set of methods to work with them in an easy way.

![Diagram of a GNSS propagation model](image)

Figure 4.3: Representation of a GNSS propagation model (inspired in [Sal]).

On the other hand, as it is explained in [Sal], are structured as an inverted tree (see figure 4.3). As a tree GDS structures have two parts, the trunk and the branches. In a tree the trunk is common for all branches, so in GDS the trunk, also called *header*, compress all the common information to all the GNSS data values stored inside the GDS structure. Instead these, the branches (*body*) include the GNSS data values and the indexes needed to access to these values in GDS. All the GDS classes are implemented combining different types of common data and indexes (*trunk*) and different types of specific data an values (*branches*).

### GDS Paradigm

Apart of the GDS structures the GDS paradigm is composed also by several processing classes. The objects of these processing classes perform inside GDS and modify, add, delete or save
and extract information indexing them depending on the process. These processing objects are
designed to use sensible defaults in their parameters, but may be tuned to suit specific needs.

Thanks to the GDS paradigm the GNSS is like a box where the data flow through it from
one box to another box implementing different processing classes. So we can say that the data
processing in GDS paradigm is like an assembly line. This approach allows for a clean, simple-
to-read, simple-to-use software code that speeds up development and minimizes errors.

For example, a ModeledPR (Modeled Pseudorange) object may take as parameters observ-
able type, ephemeris, ionosphere and troposphere models, and will add to the incoming GDS
some extra data such as geometric range, satellite elevation and azimuth, prefit residuals, and
so on. It will also automatically remove those satellites missing critical data (as ephemeris, for
example). Other example processing class is CodeSmoother, whose objects will take as param-
eters a given code observable type and a maximum window size, reading the corresponding
code and phase (as well as the cycle slip flag) from the GDS. Then they will compute a new
smoothed observable (self adjusting the window size along the way), and will replace the original
observable with the new one.

The idea of the execution is based on the redefinition of c++ operator ”>>”. This operator
is implemented in such a way that allows us to concatenate several instructions, allowing to
the programmer concatenate the output of one instruction with the input of the other one. The idea
of ”>>” operator is as the pipes in the Unix shell.

The following lines are some examples using GDS in the GPS-SDR.

First, let’s go with a portion of code implementing the core of a program making a standard,
C1 code GPS data processing, solving with a plain Least-Mean-Squares (LMS) solver. Results
are given in longitude (degrees), latitude (degrees) and altitude (meters).

```c++
while(readFile >> gpData)  // Get data out of RINEX into GDS
    gpsData.keepOnlyTypeID(TypeID::C1) >> myFilter >> model >> solver;  // GNSS data processing line
    Position solPos({ model.rxPos.X() + solver.getXSolution(),
                     model.rxPos.Y() + solver.getYSolution(),
                     model.rxPos.Z() + solver.getZSolution() });  // Get new position
    cout << solPos.longitude() << ',', '
    cout << solPos.latitude() << ', ',
    cout << solPos.height() << ' ' << endl;  // Print results
```

Figure 4.4: GPSTk Positioning Implementation.

The Figure 4.4 shows an example of the positioning implementation in the GPSTk. Line 1
is a while loop that each time will take one full epoch of data out of the RINEX observations
file (handled by RinexObsStream object rinexFile and will insert such data into the GDS called gpsData (provided by class gnssRinex). This will be done as long as there are epochs to process in the RINEX file, and line 12 closes the loop. The data processing itself is done in line 3: gpsData which only C1 data is taken is fed into limitsFilter (an object from SimpleFilter class) that checks if C1 is within reasonable limits. C1 values that don’t meet the limits are taken out from gpsData, and the result is then fed into model (an object from aforementioned ModeledPR class). Finally, the expanded gpsData (including model-related data along the original observations) is taken by the processing line into lmsSolver, an object that belongs to SolverLMS class and solves the equation system. That finishes data processing. Please note that the solver could be also a WLMS or another one.

Lines 5, 6 and 7 are just used to update the actual position. Lines 9, 10, 11 are just used to print the latitude, longitude, and altitude solution to screen. Please note that each solution is referred to using its corresponding TypeID indexing object. This is not strictly necessary (there are several ways to refer to the solution, according to user preferences), but this way is preferred because of ease of use and consistence with GDS paradigm.

The Figure 4.4 is only an example of the schedule to get the position in GPSTk, where all the initializations aren’t present, in the normal mode. The Figure 4.5 shows and example for the DGPS (Differential GPS) in GPSTk.

```cpp
while(rinexFile >> gpsData) { // Get data out of RINEX into GDS
    gRef >> synchro >> myFilter >> modelRef; // First, let’s synchronize
    delta.setRefData(gRef.body); //and process reference station data
    gFrom12 >> myFilter >> model >> delta >> mpwS >> solverKalman; //ONSS Data Processing Line
    Position solPos12((model.rxPos.X() + solverK12.getSolution(TypeID::dx)), //Get new Position
                     (model.rxPos.Y() + solverK12.getSolution(TypeID::dy)),
                     (model.rxPos.Z() + solverK12.getSolution(TypeID::dz)));
    cout << setprecision(10) << solPos12.longitude() << ' ', ';
    cout << solPos12.geodeticLatitude() << ' ', ';
    cout << solPos12.height() << ' ', ';
}
```

Figure 4.5: GPSTk DGPS Positioning Implementation.

The idea in the Figure 4.5 is the same than in the Figure 4.4 but now let’s apply the DGPS using a reference station (see Line 4 and 5 Figure 4.5) and then computing the difference in the data processing line (Line 7 Figure 4.5) with the `delta` statement.

More examples of the using of the GPSTk and GDS in positioning will be showed in appendix C.
Chapter 5

Test and Simulations

5.1 ∆∆DLL Simulations

Theoretically the ∆∆DLL has a better performance than the DLL correlator working in a multi-path scenario. Thus, this section’s goal is to show the improvements of the implemented ∆∆DLL versus the original DLL in GPS-SDR. To achieve it we need an specific hardware environment to create the correct simulation scenario.

5.1.1 Testbed

The ∆∆DLL mitigate the multipath in a better mode than DLL, so to demonstrate his improvements we need to create a multipath environment with the same characteristics for both correlators. So, also we have to be sure that we have the same signal in all simulations. To assume that we are fulfilling all this requirements we will use the next hardware.

USRP and PC

The main important hardware to execute the GPS-SDR software is the USRP (see section 1.3) and a standard PC. To do all ours simulations we will use a Intel QUAD-Core PC with the gnuradio libraries and GPS-SDR software installed. During the simulations we will assume that the software requirements will be correctly installed and wellworking. To know how to install the software see appendix A.
Signal Generator

Agilent E4438C ESG Vector Signal Generator with the GPS Personality, which allows us to get a real GPS signal and the telemetry for 8 satellites simultaneously, will be used during this simulation [Agi]. The signal generator will allow us to be sure that the DLL and the $\Delta \Delta$DLL correlators receive always the same signal and with the same conditions. Also it will allow us to know previously the correct receiver’s position, so to get the real pseudodistance between the satellite and the receiver.

Channel Emulator

As we have said previously theoretically the $\Delta \Delta$DLL correlator has a better performance than the DLL in a multipath scenario. However to demonstrate these improvements we need to create and control the multipath channel. So thanks to the Propsim C8 Channel Emulator we will be able to make a controllable and well-known multipath channel [Pro]. It will allow us to change the multipath delay, power and other characteristics.

Hardware Assembly

The simulation hardware assembly will be as following. First of all we have to connect signal’s generator output to the channel’s emulator input, then the channel’s emulator output to the USRP’s input. Finally we have to connect the USB2.0 USRP’s output to the USB2.0 PC’s input. Now we are ready to execute the simulations. The final hardware assembly diagram is showed at figure 5.1.

Figure 5.1: Diagram connection for multipath propagation simulation
5.1.2 Simulations

The following simulations have as achievement compare the error in the pseudodistance computed between the doble delta correlator and the simple one, both with a $\delta$ spacing of $1T_c$ (Chip Time).

![Laboratory scenario for multipath propagation simulation](image)

The first step is to assemble all the hardware to create a multipath propagation scenario at the laboratory. The Figure 5.2 shows the laboratory testbed used at these simulations. Once assembled the hardware the following steps are:

1. **Get GPS Signal**[Agi]. Thanks to the Signal Generator we can get always the same signal. During these simulations we have used the internal simulation file *Hawaii.gps*. The following table shows the signal parameters.

<table>
<thead>
<tr>
<th>Position</th>
<th>Latitude (degrees)</th>
<th>Longitude (degrees)</th>
<th>Altitude (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>19.81888</td>
<td>-155.988055</td>
<td>100</td>
</tr>
<tr>
<td>Visible Satellites (SV Identifier)</td>
<td>2, 4, 8, 13, 20, 23, 25, 27</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Simulated Signal Parameters
Chapter 5. Test and Simulations

2. **Generate the multipath channel.** The multipath channel model used during all the simulations is a channel with single direct signal and a powerless reflected signal. Thus we can represent mathematically the channel impulse response as:

\[ h_m(t) = A\delta(t) + \frac{A}{\sqrt{2}}\delta(t - \tau_j) \]

where:

- \( h_m(t) \) is the channel impulse response at every time \( t \).
- \( A \) is the scale factor introduced by the channel.
- And \( \tau_j \) is the delay introduced in the reflected signal by the channel. The \( \tau_j \) value changes between 0.1 to 1.4 times of chip according to the multipath delay. So:

\[ \tau_j = \frac{j}{10}\frac{1}{1023000\text{ chips/second}} \text{; and } j = 0, 1, 2, 3, 4...14. \]

The Fig.5.3 is the graphical multipath channel representation.

![Figure 5.3: Multipath channel impulse response.](image)

3. **Get the instantaneous pseudodistance error.** For each satellite the instantaneous pseudodistance error is obtained computing the difference between the real pseudodistance and the GPS-SDR computed pseudodistance. The real pseudodistance can be calculated thanks that we know the real position (see D.3) and it is fixed. The GPS-SDR pseudodistance is computed by the GPS receptor using the Least Mean Squares method (LMS) [W.H]. So mathematically the difference between them can be expressed as:

\[ \text{error}_i(t) = \rho_{GPS_i}(t) - \rho_{REAL_i}(t) \]

with

\[ \rho_{REAL_i}(t) = ||r_{s_i} - r_o|| - b_{s_i}(t)c + n(t) \]
where:

- \( \mathbf{r}_i \) and \( \mathbf{r}_o \) are the satellite \( i \) vectors position and the receptor/observer vector position both in ECEF (Earth Fixed Earth Centered) coordinates.
- \( b_s \) is the satellite \( i \) internal clock bias.
- \( c \) is the light speed.
- \( \rho_{GPS}(t) \) is the pseudodistance computed by the GPS-SDR receptor using the Least Mean Squares method (LMS).
- \( n(t) \) are the noise terms. Any noise estimation had been done, so it will affect at the real pseudorange computation.

4. **Compute the average pseudodistance error** for a preestablished multipath delay \( \tau_j \). A 10 minutes length signal has been recorded to compute the average error. As the GPS-SDR receptor can compute the pseudodistance 10 times per second we have 6000 samples to do the average. So the average pseudodistance error for the satellite \( i \) is:

\[
\text{error}_{\text{average}}|_{\tau=\tau_j} = \frac{\sum_{i=0}^{t_f} (\text{error}_i(t))}{\#\text{samples}}
\]

5. **Repeat** steps from 1 to 4 changing the channel multipath delay in the step 2 and compute the new average pseudorange error (\( \tau_j \)).

5.1.3 **Results**

Once the pseudorange error is calculated for each multipath delay (\( \tau_j \)) is time to show graphically his shape. The figure 5.4 shows it. The real pseudodistance error has more peaks in his graph because samples are separated 0.1 times of chip between them, so it is a discrete graph. Inversely the theoretically pseudodistance error graph is a continuous one.
Figure 5.4: Real pseudodistance error comparison (DLL vs $\Delta\Delta$DLL).

Figure 5.5: Theoretical pseudodistance error comparison (DLL vs $\Delta\Delta$DLL).
As we can see in the picture the \( \Delta\Delta \text{DLL} \) gets an improvement which arrives until 10 meters. If we remember the theoretical improvement of \( \Delta\Delta \text{DLL} \) versus DLL (figure 5.5) computed in section 3.5 it arrived until 40 meters. The difference between both improvements is that in section 3.5 we did the simulations without noise, it is in an ideal situation. But in the real simulation with the GPS-SDR, the signal generator and the channel emulator we have a noise performance. In spite of the noise effect, observing the results we can conclude that the \( \Delta\Delta \text{DLL} \) adds a real improvement in the user position calculation.

5.2 GPStk Simulations

In this section will be studied and compared the position calculation with the GPS-SDR which uses LMS method versus other methods. This comparison is possible thanks to the GPStk libraries which have implemented some other position algorithms. LMS algorithm will be compared versus Weight Least Mean Squares (WMS) and kalman filter, all of them in their normal use and with the DGPS technique. Finally the final positions estimations will be compared graphically thanks to the Google Earth tool. These procedures will be done for a static receiver position and for a dynamic one.

5.2.1 Testbed

The experimental testbed in this section is very simple. In fact we need only three components: a PC, USRP and an RF antenna.

The main important hardware to execute the GPS-SDR software is the USRP (see section 1.3) and a standard PC. To do all ours simulations we will use a Intel QUAD-Core PC with the gnuradio libraries and GPS-SDR software installed. During the simulations we will assume that the software requirements will be correctly installed and wellworking. To know how to install the software see appendix A. In addition to this, previously we have been installed the Google Earth tool in our PC. Also, to be able to get GPS signal in the receiver we will use an RF antenna. A car will be needed to be able to do the dynamic experimental simulations.

Hardware assembly

The three hardware components will be assembled as following. First of all we have to connect the USRP USB2.0 output to the PC USB2.0 input. And then we have to connect the RF antenna connector to the RF1 USRP’s input connector. Now we are ready to execute the GPS-SDR in real time. Figure 5.6 shows the hardware used for these simulations and figure 5.7 the RF Front
end hardware block diagram. In the dynamic simulation the USRP will be plugged in the car battery through a car AC/DC adapter, and the PC will be changed by a laptop computer.

Figure 5.6: Hardware GPStk simulations components.
5.2.2 Simulations

As it has been said previously the following simulations has as achievement add some more positioning algorithms to the GPS-SDR original software. Once we have been added the additional positioning algorithms the final block diagram for the software GPS receiver is showed in figure 5.8. Where the green part on the left is the original GPS-SDR plus the Rinex (v2.1) file extractor added by us. The orange part on the right is the GPStk part of the software receiver which implement the LMS, WMS and Kalman filter positioning algorithms. This position is given in the Google Earth compatible file (’.kml’) and it is showed in the screen maps.

Figure 5.7: Hardware detailed diagram.

Figure 5.8: Simplified block diagram for the software GPS receiver.
Chapter 5. Test and Simulations

The compare graphically all the positioning algorithms versus the LMS algorithm of the original GPS-SDR are the following ones.

1. **Execute GPS-SDR software with the Rinex extractor option enabled.** Get the file ‘GPS-SDR.kml’, GPS-SDR Google Earth navigation file, and the Rinex navigation and observation files, ‘cttcddd1.09n’ and ‘cttcddd1.09o’. It is done thanks to the functions Rinex2ObsHeader(), Rinex2NavHeader(), LogRinex2Obs() and LogRinex2Nav() added to the original source code at /objects/telemetry.cpp.

2. **Load ‘cttcddd1.09n’ and ‘cttcddd1.09o’ at the GPStk software.**

3. **Compute the position.** To compute the position apply the algorithms (LMS, WMS, kalman) to obtained Rinex files. If we want to apply the DGPS technique we also need to load the reference station Rinex observation file. The positioning results will be stored in the Google Earth extension files: ‘GPStk-LMS.kml’, ‘GPStk-LMS-DGPS.kml’, ‘GPStk-WMS.kml’, ‘GPStk-WMS-DGPS.kml’, ‘GPStk-Kalman.kml’ and ‘GPStk-Kalman-DGPS.kml’.

4. **Print results.** Once we have computed the position for each algorithms we have to load the previous’.kml’ files in the Google Earth tool to evaluate each improvements.

The following figure shows how we implement steps 2 and 3 for a positioning algorithm without applying DGPS technique, for more information see chapter 4. The code used for all the GPStk simulations is contained in the file gpstkposition.cpp, see appendix C.

```cpp
while(rinexFile >> gpsData) { // Get data out of RINEX into ODS
    gpsData.keepOnlyTypeID(TypeID::C1) >> myFilter >> model >> solver; // ONSS data processing line
    Position solPos = model.rPos.x() + solver.getSolution(TypeID::dx),
               model.rPos.y() + solver.getSolution(TypeID::dy),
               model.rPos.z() + solver.getSolution(TypeID::dz));
    cout << solPos.longitude() << "", ";
    cout << solPos.geodeticLatitude() << "", ";
    cout << solPos.height() << "";
}
```

Figure 5.9: GPStk Positioning Implementation.

### 5.2.3 Static GPStk Results

In this section will be compared the results obtained by the different positioning algorithms mentioned before against LMS which is the original GPS-SDR algorithm. The following results
were obtained during a 120 seconds observation for a static receiver position at CTTC’s location (Castelldefels) on June 2, 2009.

<table>
<thead>
<tr>
<th>Position</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latitude (degrees)</td>
<td>Longitude (degrees)</td>
<td>Altitude (meters)</td>
</tr>
<tr>
<td>41.2747</td>
<td>1.9875</td>
<td>30</td>
</tr>
<tr>
<td>Visible Satellites (SV Identifier)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12, 17, 18, 22, 30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Simulated Signal Parameters

The reference observation file were obtained from the EPSC reference station, located in Castelldefels (Latitude:41° 16’ 31.33800” North, Longitude: 1° 59’ 12.71300” East, Ellipsoidal height: 74.790 meters), via the gAGE/UPC file transfer server ftp://gage19.upc.es/.incoming.

The figure 5.10 shows the original’s GPS-SDR position solution against the GPS-SDR receiver’s real position.

Figure 5.10: GPS Receiver Real Position (yellow pin) vs LMS Position (red line).

At the following sections the LMS positioning algorithm, which is the original one of GPS-SDR, will be compared against other algorithms.
Original LMS vs WMS and Kalman Filter

The first step will be to compare the LMS algorithm against the WMS one. Accordingly to the theoretical concept of both algorithms, the difference between them is that the WMS uses as each satellite received power to compute the position. So what we hope to see is more or less the same positioning response but with some variations. The figure 5.11 shows both responses. In this case it is a little bit difficult to appreciate the difference between both algorithms because all the satellites had the same power more or less. The conclusion is that the response shape, variance and average are the same, but with WMS we have a more realistic response because it has into account the received power.

![Figure 5.11: LMS GPS-SDR Position (red) versus GPStk WMS Position (blue).](image)

The next step is to compare LMS algorithm with kalman filter solution. As it has been explained at section 2.3.2, the main theoretical difference between them is that the kalman filter solution has into account the previous position and results to compute the actual position. In the other hand LMS algorithm doesn’t use the previous calculations. So, we can say that Kalman filter has some memory which hasn’t got LMS. The figure 5.12 shows both responses. It’s easy to see that kalman’s filter solution is less variable than LMS solution, this is as consequence of the kalman’s memory.

Assuming these results, we can say that LMS and WMS solutions are very similar, unless we have a lot of difference between each satellite received power. Also, it’s clear to think with
kalman’s filter solution as the better position solution when we are in a static position; because it reduces the variance and the noise effect to the position calculation.

Figure 5.12: LMS GPS-SDR Position (red) versus GPStk kalman’s filter Position (white).

**Algorithms without DGPS technique vs Algorithms with DGPS technique**

Until now we have compared different types of algorithms to compute the receiver position; so now it’s time to see the effect of applying the DGPS techniques. The figure 5.13 shows the solution obtained after applying DGPS technique on a LMS solver. It could be seen that the use of DGPS data on LMS solver has shown a small offset (bias) correction, but the variance of the position results is maintained at the same levels. This result indicates that the sources of the remaining error are not correlated to the atmospheric effects. Figure 5.14 shows the results obtained after applying the DGPS technique on a kalman’s filter solver. The effect of the DGPS on the kalman’s filter is the same that in the LMS solver, so DGPS gives an offset correction to the receiver position. In figure 5.14 we can see as the kalman filter with DGPS technique gets the exact receiver’s position.

So, in conclusion, as it have been explained at section 2.3.2, DGPS provide to the solution with an offset correction. They basically correct the atmospheric effects which produce an bias error to the receiver position.
Figure 5.13: LMS GPS-SDR Position (red) versus GPStk DGPS LMS Position (blue).

Figure 5.14: Kalman GPStk Position (white) versus kalman GPStk DGPS Position (black).
Chapter 5. Test and Simulations

Original LMS vs DGPS-Kalman Filter

To note our developments is necessary to compare the original GPS-SDR solution computation against our best solution calculation which is obtained thanks to kalman+DGPS solver. The figure 5.15 shows graphically this comparison. It is clear to see the great improvement obtained with kalman+DGPS solver, as we achieve to reduce the solution variance and eliminate the atmospheric error.

Figure 5.15: LMS GPS-SDR Position (red) versus kalman GPStk DGPS Position (black).

Static Simulation Conclusions

To conclude our static experimental simulations, we can say that with the GPStk’s module implementation we have got an important improvement to our static receiver position calculation. It has been done thanks to the two improvements added to the original GPS-SDR. The first one is the improvement to the positioning algorithm, achieved thanks to the use of the kalman filter solver. It gives us a better and less variable solution. The second improvement is the addition of the DGPS technique to the position calculation. This technique allows us to correct the receiver position, diverted from the real position due to the atmospheric error.
5.2.4 Dynamic GPStk Results

Again the results obtained by the GPS-SDR will be compared with other positioning algorithms but now the receiver will be in movement. The following results were obtained a dynamic receiver position at PMT (Parc Mediterrani de la Tecnologia) in Castelldefels on June 2, 2009. The figure 5.16 shows the GPS-SDR assembling in a car. The hardware needed it’s the same but substituting the computer by a laptop and using a car adapter to plug in the USRP board.

![Figure 5.16: GPS-SDR Car Assembling.](image)

<table>
<thead>
<tr>
<th>Initial Position</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latitude (degrees)</td>
<td>Longitude (degrees)</td>
<td>Altitude (meters)</td>
<td></td>
</tr>
<tr>
<td>41.2761</td>
<td>1.99027</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visible Satellites (SV Identifier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,12,29,30,31</td>
</tr>
</tbody>
</table>

Table 5.3: Simulated Signal Parameters

Once the GPS-SDR and the other components are assembled, we are prepared to use them. The figure 5.23 shows the real route described for the GPS-SDR against the original LMS algorithm position calculation in a dynamic user position.
Chapter 5. Test and Simulations

Figure 5.17: LMS GPS-SDR Position (red) versus Real GPS-SDR Receiver Route.

Original LMS vs WMS and Kalman Filter

As it has been done at the static GPStk simulations, the first step is to compare LMS original GPS-SDR solution against other algorithms, which we have added to the software. Figure 5.18 shows the comparison between LMS solver versus WMS solver. As it has been said before, the difference between them it’s not clear to see if each satellite received power are equal or similar. At figure 5.18 it could be seen a little difference between both algorithms at the beginning, but then both have the same positioning solution; as each satellite receiver power is the same.

Figure 5.18: LMS GPS-SDR Position (red) versus WMS GPStk Position (blue).

The next step is to compare the LMS original solver against the kalman filter solution. First of all, we have to note that to use the kalman filter in a dynamic receiver position we have to declare the samples as dynamic. In our case, this samples have been declared as white noise with a sigma value of 100. Figure 5.19 shows the difference between both algorithms. It is able
to see that Kalman’s filter solution is a little bit less variable than LMS solution. Also it’s easy to see as in the curve the Kalman’s solver gives a solution closer to the real than LMS solver, this occurs because Kalman’s filter has memory and uses the previously samples to compute the actual one.

Figure 5.19: LMS GPS-SDR Position (red) versus Kalman GPStk Position (white).

**Algorithms without DGPS technique vs Algorithms with DGPS technique**

Until now we have compared different types of algorithms to compute the receiver position. So now as in the static simulations, it’s time to see the effect of applying the DGPS techniques. The figure 5.20 shows the comparison between LMS and GPStk LMS DGPS, and the figure 5.21 shows the difference between GPStk WMS and GPStk WMS DGPS. It’s clear to see in both pictures that the DGPS technique introduce an offset (bias) correction to the solution without DGPS, but the variance of the position results is maintained at the same levels. This result indicates that the sources of the remaining error are not correlated to the atmospheric effects.

Figure 5.20: LMS GPS-SDR Position (red) versus LMS GPStk DGPS Position (blue).
Figure 5.21: WMS GPStk Position (blue) versus WMS GPStk DGPS Position (purple).

Figure 5.22 shows the results obtained after applying the DGPS technique on a kalman’s filter solver. The effect of the DGPS on the kalman’s filter is the same that in the LMS or WMS solvers, so DGPS gives an offset correction to the receiver position. In figure 5.22 we can see as the kalman filter with DGPS technique gets a more precise solution. The clearest DGPS perform can be seen at the curve, where the kalman+DGPS solution gets a solution practically equal at the real one.

Figure 5.22: kalman GPStk Position (white) versus kalman GPStk DGPS Position (black).

**Original LMS vs DGPS-Kalman Filter**

Again as previously, to note our developments is necessary to compare the original GPS-SDR solution computation against our best solution calculation in a dynamic user position. The best solution is obtained thanks to kalman+DGPS solver. Figure ?? shows the difference between the original LMS solver versus the programmed kalman DGPS solver. At the figure, it can be
seen as the DGPS+Kalman solution is the solution which is closer to the real one each time, also the solution variance is less higher in DGPS+kalman solver than at the original LMS solver.

Figure 5.23: LMS GPS-SDR Position (red) versus kalman GPStk DGPS Position (black).

**Dynamic Simulation Conclusions**

To conclude our dynamic experimental simulations, we can say that with the GPStk’s module implementation we have got an important improvement to our dynamic receiver position calculation. It has been done thanks to the two improvements added to the original GPS-SDR. The first one is the improvement to the positioning algorithm, achieved thanks to the use of the kalman filter solver. This solver provides us a better and less variable solution, although the solution is more variable than in the static simulations, because the receiver is in movement. The second improvement is the addition of the DGPS technique to the position calculation. This technique allows us to correct the offset error introduced to the receiver position by the atmospheric error.

**5.2.5 GPStk Algorithms Positioning Accuracy**

During this section it will be showed all the GPStk position accuracy analysis. This accuracy has been analyzed in two ways: horizontal accuracy (2D) and vertical accuracy. The appendix D shows all steps followed to achieve the next results. The position accuracy has been studied only for the GPStk static simulations because we could know the real position.
Chapter 5. Test and Simulations

GPStk Horizontal Positioning Accuracy

The following results (Table 5.4) has been obtained with a 120 second length simulation. Due to the fact that the GPS-SDR give the position ten times every second, it represents a group of 1200 samples. The WMS and the LMS cases, with DGPS and without it, give the same results, because when we did the simulations each satellite had more or less the same power.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CEP (meters)</th>
<th>RMS (meters)</th>
<th>95% (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS</td>
<td>23.5</td>
<td>28.3</td>
<td>49.9</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>13.0</td>
<td>15.6</td>
<td>27.0</td>
</tr>
<tr>
<td>WMS</td>
<td>23.5</td>
<td>28.3</td>
<td>49.9</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>13.0</td>
<td>15.6</td>
<td>27.0</td>
</tr>
<tr>
<td>Kalman</td>
<td>13.8</td>
<td>16.6</td>
<td>28.7</td>
</tr>
<tr>
<td>Kalman+DGPS</td>
<td>4.8</td>
<td>5.8</td>
<td>10.1</td>
</tr>
</tbody>
</table>

Table 5.4: Horizontal Position Accuracy (2D)

Analyzing the results, it’s clear to see that the best positioning algorithm in our GPS-SDR receiver is the Kalman filter plus the DGPS technique. It’s important to note that the distribution of the position samples have this particular shape (ellipsoidal) because during the observations we were tracking satellites from only one site of the receiver. Observing the accuracy for the commercial GPS receivers in [Wor09] which is included in Appendix E we can see that all of them have as the worst case $\approx 2.5 - 3$ meters of accuracy. So, although we haven’t been able to achieve the accuracy of a commercial receiver, adding the GPStk positioning algorithms we have been able to improve the position accuracy until 23 meters the RMS and about 30 meters the 95%, if we compare the GPS-SDR LMS original algorithm and the Kalman filter+DGPS.

GPStk Vertical Positioning Accuracy

The following results (Table 5.5) has been obtained with a 120 second length simulation. Due to the fact that the GPS-SDR give the position ten times every second, it represents a group of 1200 samples. The WMS and the LMS cases, with DGPS and without it, give the same results, because when we did the simulations each satellite had more or less the same power.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CEP (meters)</th>
<th>RMS (meters)</th>
<th>95% (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS</td>
<td>8.8</td>
<td>10.5</td>
<td>18.2</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>4.8</td>
<td>5.7</td>
<td>9.9</td>
</tr>
<tr>
<td>WMS</td>
<td>8.8</td>
<td>10.5</td>
<td>18.2</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>4.8</td>
<td>5.7</td>
<td>9.9</td>
</tr>
<tr>
<td>Kalman</td>
<td>4.8</td>
<td>5.7</td>
<td>9.8</td>
</tr>
<tr>
<td>Kalman+DGPS</td>
<td>1</td>
<td>1.2</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table 5.5: Vertical Position Accuracy

Analyzing the results, it’s clear to see that the best positioning algorithm in our GPS-SDR receiver is the Kalman filter plus the DGPS technique. Although we haven’t been able to achieve the accuracy of a comercial receiver, adding the GPStk positioning algorithms we have been able to improve the position accuracy until 9 meters the RMS and about 16 meters the 95%, if we compare the GPS-SDR LMS original algorithm and the Kalman filter+DGPS.
Chapter 6

Conclusions

Since in 70’s the US Army invented the Global Positioning System (GPS) lots of investigations have been done about GNSS receivers. The firsts steps in GNSS receivers were done with hardware receivers, but thanks to the computer’s powerful improvement, in the last years the digital GNSS receivers development have been possible. The software GPS receivers provide to the user a very useful and powerful tool adding too an interesting characteristic, the reprogrammability.

During all this research project we have been working with a GPS software receiver called GPS-SDR. As we had said in the introduction our purpose was improve the GPS-SDR positioning computation, and also explore the GNSS receivers possibilities. To achieve our goal we have attack the problem by to ways: the multipath mitigation and the positioning algorithm improvement.

The first step has been to solve the multipath problem. To do this we have added to the GPS-SDR what we have called Double Delta Correlator (ΔΔDLL). The multipath effect performs, making worse, the satellites tracking, so the ΔΔDLL purpose have been to get a better satellite tracking compared with the original GPS-SDR’s correlator (DLL). First of all the theoretically response of both correlations has been studied by the MATLAB tool in terms of pseudorange error. These simulations have been done assuming an ideal case, it is without noise. The theoretical pseudorange error has as a result an improvement about 40 meters between the ΔΔDLL and the DLL.

The next step has been to compute the pseudorange calculation improvement in the real or experimental case, it is with the receiver and real GPS signal. To get these results we have used the GPS-SDR and also a channel emulator, to create a multipath channel, and a GPS signal generator, to have the same GPS signal to whole simulations ([Pro] [Agi]). In the experimental simulations we have assumed the noise presence without estimating it, so it has affected our results. The difference in terms of pseudorange error between both correlators in the experimental
simulations have as a result and improvement of 10 meters for the ΔΔDLL.
The differences between theoretical and the experimental simulations are produced by the noise presence.

Once the multipath problem has been solved, the next step of our research has been to improve the positioning calculation. To achieve this goal we have used the GPS toolkit (GPStk) which has provided as a large set of positioning algorithms. To use the GPStk first we have had to add to the GPS-SDR a module to extract the Receiver Independent Exchange Format (RINEX) which has allowed us to connect the GPS-SDR with GPStk. During this part of the project the Least Means Squares (LMS) positioning method, which is the original in GPS-SDR, has been compared with others as: Weighthed Least Means Squares (WMS), Kalman filter both with normal positioning and with the Differential GPS (DGPS) technique; it also has been compared with DGPS-LMS. This comparison has been done graphically thanks to the Google Earth tool. The whole algorithms have been analyzed with a static and a dynamic receiver mode.

As it has been able to seen in the experimental results the WMS has a softer improvement versus the LMS algorithm thanks that WMS ponders each satellite result in agreement with it signal to noise ratio (SNR). The best positioning results has been achieved with the kalman filter algorithm thanks that it is an algorithm with memory, it is, it computes the actual position taking into account the previous position.

Also it’s easy to see in our results that the DGPS technique provides us an offset position correction which allows us to get a significant improvement. Thus, after doing all the simulations we can say that the kalman filter plus DGPS technique gives us the better position calculation either static or dynamic receiver.

After conclude our research we can say that we have achieved our goals. So we have been able to mitigate in some way the multipath problem and also we have been able to improve the position calculation.

It’s important to note that all the research and developments done during this project have produced two scientific publications. The GPS-SDR receiver plus the Rinex output’s and the GPStk’s module have been presented as ‘GNSS Open-Source Software Receiver’ demo at the NEWCOM++/ACoRN Joint Workshop congress (http://www.newcom-acorn.org/naw09). The GPS-SDR comparison between the ΔΔDLL and DLL correlators implementation has been presented and accepted at the URSI2009 Cantabria Congress with the tittle ’Double Delta Correlator for a real-time GPS’ (http://www.ursi2009.org/). See appendix F to see these publications.

Also, the project is involved in other future projects, so it’s the GPS software used to test another CTTC’s work which consists to implement a beamforming array for GPS.
Bibliography


[Sal] D. Salazar, *High accuracy positioning using carrier-phases with the open source GPSTk software*, gAGE-UPC.


Appendix A

Software Installation and Execution

A.1 GNU Radio Software v3.1.3

GNU Radio needs the following tools:

- Development Tools (need for compilation)
  - `g++` - GNU project C and C++ compiler
  - `subversion` - a version control system
  - `make` - determines automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.
  - `autoconf, automake, libtool` - Generate configuration scripts, automatically create Makefile.in’s from Makefile.am’s and generate static or libraries.
  - `sdcc` (from “universe”; 2.4 or newer) - Small Device C Compiler
  - `guile` (1.6 or newer) - Evaluates Scheme code, interactively or from a script

- Libraries (need for runtime and for compilation)
  - `python-dev` - For Python programming
  - `FFTW 3.X (fftw3, fftw3-dev)` - Libraries for fast FFT computation
  - `cppunit` (libcppunit and libcppunit-dev) - the C++ port of the JUnit framework for unit testing
  - `Boost` (libboost and libboost-dev) - free, peer-reviewed, portable C++ source libraries
– **libusb** and **libusb-dev** - USB handling
– **wxWidgets** (wx-common) and **wxPython** (python-wxgtk2.6) - Libraries for GUIs.
– **python-numpy** (via python-numpy-ext) (for SVN on or after 2007-May-28) - Numerical methods in Python
– **ALSA** (alsa-base, libasound2 and libasound2-dev) - Sound
– **Qt** (libqt3-mt-dev; version 4 does not seem to work) - Libraries for nice GUIs
– **SDL** (libsdl-dev) - the cross-platform Simple Directmedia Layer graphics library that allows programs portable low level access to a video framebuffer, audio output, mouse, and keyboard.

- **SWIG** (1.3.31 or newer required) - Simplified Wrapper and Interface Generator, creates wrapper code to connect C and C++ code to scripting languages like Python.
- **QWT** (optional) (5.0.0 or newer required) - Qt GUI Components and utility classes
  – Must be installed from source

- Other useful packages
  – **doxygen** - For creating documentation from source code
  – **usbview** - A small GTK application to show what the device tree of the USB bus looks like
  – **octave** - Similar to MATLAB

### A.1.1 GNU Radio Installation steps

You can install the software to some operative system. The following information gives the steps followed for the software installation over Ubuntu 8.04 LTS.

**Installation of required packages:**

```
$ sudo apt-get -y install g++ automake1.9 libtool python-dev fftw3-dev
   libcppunit-dev libboost-dev sdcc libusb-dev libasound2-dev \
   libsdll1.2-dev python-wxgtk2.8 subversion guile-1.6 libqt3-mt-dev\ 
   python-numpy-ext
```

**Installation of optional packages:**

```
$ sudo apt-get -y install gkrellm wx-common libwxgtk2.8-dev alsa-base\ 
   autoconfxorg-dev g77 gawk bison openssh-server emacs cvs\ 
   usbview octave python-cheetah python-lxml python-gtk2 doxygen
```
Appendix A. Software Installation and Execution

Get, compile and install SWIG:

$ wget http://prdownloads.sourceforge.net/swig/swig-1.3.33.tar.gz
$ tar zxf swig-1.3.33.tar.gz
$ cd swig-1.3.33
$ ./configure
$ make
$ sudo make install

Get, compile and install the latest version of the Boost library. Since August 20, 2008, GNU Radio has a scheduler for SMP (Symmetrical Multi Processing) machines that needs a Boost version higher than 1.35. This version has not been yet released as an Ubuntu package, so it has to be installed manually:

1. Download the latest version of boost from boost.sourceforge.net. (boost_1_36_0.tar.bz2 was the latest when this was written)

2. Unpack it somewhere

   $ tar -xvf boost_1_36_0.tar.bz2

3. cd into the resulting directory

   $ cd boost_1_36_0

4. Pick a prefix to install it into. I used /opt/boost_1_36_0

   $ BOOST_PREFIX=/opt/boost_1_36_0
   $ ./configure --prefix=$BOOST_PREFIX --with-libraries=thread, date_time
   $ make
   $ sudo make install

5. Now, tell GNU Radio where to find it:

   $ export LD_LIBRARY_PATH=$BOOST_PREFIX/lib

The environmental variables are only available for the current terminal session and are not available after the session is terminated. If you plan to manipulate the configuration many times, you should consider adding the following line to your .bashrc file:

   $ export LD_LIBRARY_PATH=/opt/boost_1_36_0/lib

Download, bootstrap, configure, and compile GNU Radio with all components working on Linux:
$ svn co http://gnuradio.org/svn/gnuradio/trunk gnuradio
$ cd gnuradio
$ ./bootstrap
$ ./configure --with-boost=$BOOST_PREFIX --enable-all-components
  --disable-gr-gcell --disable-gr-audio-osx \
  --disable-gr-audio-windows --disable-gr-audio-oss \
  --disable-gcell --disable-gr-audio-jack \
  --disable-gr-audio-portaudio --disable-gr-comedi
$ make
$ sudo make install

Instructions about fine tuning of the installation process can be found at [http://gnuradio.org/trac/wiki/BuildConfiguration](http://gnuradio.org/trac/wiki/BuildConfiguration).

Ubuntu uses udev for handling hotplug devices, and does not by default provide non-root access to the USRP. The following script sets up groups to handle USRP via USB, either live or hot-plug:

$ sudo addgroup usrp
$ sudo addgroup <YOUR_USERNAME> usrp
$ echo 'ACTION="add", BUS="usb", SYSFS{idVendor}="fffe",\n  SYSFS{idProduct}="0002", GROUP="usrp", \n  MODE="0660"' > tmpfile
$ sudo chown root.root tmpfile
$ sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules

Building the documentation:

$ cd gnuradio-core/doc
$ doxygen

This creates HTML files with the documentation generated by the comments on the source code. The main web page is located at gnuradio-core/doc/html/index.html

Building the USRP documentation:

$ cd usrp/doc
$ doxygen


Edit the file .bashrc and add the following lines:

```bash
# set pythonpath for standard python libraries and GNU Radio
export PYTHONPATH=/usr/local/lib/python2.5/site-packages
```
A.2 GPS-SDR Software v2.0

A.2.1 GPS-SDR Installation steps

The GPS-SDR software installation is very simple. In fact we only need to download the source code and execute it. It’s important to note that previously we need to install GNURadio packages.

The source code can be downloaded from the GPS-SDR web [http://www.gps-sdr.com/](http://www.gps-sdr.com/). For our case we will store it at the PC Desktop inside the GPS-SDr folder.

Once we had been downloaded the source code, it’s time to know how to execute them. The steps are as following:

1. Compile GPS-SDR: open a terminal and execute the following commands.

   ```bash
   $ cd Desktop/gps-sdr
   $ make
   ```

2. Compile the GPS-SDR USRP driver: open a terminal and execute the following commands.

   ```bash
   $ cd Desktop/gps-sdr/usrp
   $ sudo su
   $ ********* (Introduce your Ubuntu password)
   $ make
   ```

3. Connect USRP and GPS-SDR: at the second opened terminal execute the following commands.

   ```bash
   $ cd ..
   $ ./gps-usrp
   ```

4. Execute GPS-SDR software: at the first opened terminal execute the following commands for the original GPS-SDR software.

   ```bash
   $ ./gps-sdr
   ```

A.2.2 GPS-SDR Software Execution

The GPS-SDR has some execution options, some of them were implemented and some of them have been implemented during this research project. For example if we want to extract a Google Earth compatible file we have to execute the program as:
$ ./gps-sdr -g

If we want to get RINEX files we have to execute:

$ ./gps-sdr -l

And if we want both outputs we execute:

$ ./gps-sdr -g -l

Also we can execute the program with the normal correlator (DLL) or the double delta one (\(\Delta\Delta\) DLL). With the previous instructions we were executing the program with the normal DLL, to execute it as \(\Delta\Delta\) DLL we need to execute:

$ ./gps-sdr -dd

And if we want to add more options, we only need to add \(-l\) \(-g\) and so on. All the possible options are:

\[-p\] <filename> use prerecorded data
\[-o\] <filename1> <filename2> do ocean reflection
\[-dd\] run receiver with double delta correlation acquisition
\[-c\] log high rate channel data
\[-l\] log navigation data and navigation/observation RINEX files
\[-d\] <N> decimate logged nav data by this N factor
\[-g\] log google earth data
\[-v\] be \texttt{verbose}
\[-n\] ncurses OFF
\[-w\] start receiver in warm start, using almanac
      and last good position
\[-u\] run receiver with usrp-gps as \texttt{child} process
A.3 GPStk Software

To use the GPSTk you must have a modern, ANSI compliant C++ compiler, such as the one provided with gcc. The GPSTk has been built successfully using the following compilers:

- GCC version 3.3, 3.4, 4.0, 4.1 under Cygwin, Linux, Solaris and AIX (as of GPSTk 1.1 or later)
- Microsoft Visual Studio .NET 2003, Visual C++ 2005 Express (as of GPSTK 1.3 or later)
- Sun Workshop 6 and Sun ONE Studio 8 (as of GPSTk 1.0 or later)
- OS X Xtools (as of GPSTK 1.5 or later)

A.3.1 GPStk Installation steps

The following procedure will build and install the GPSTk using recent, standard GNU build tools. The build has been successfully tested using the following tools and versions:

```
autoconf (GNU Autoconf) 2.59
GNU m4 1.4.2
automake (GNU automake) 1.8.5
autoconf (GNU Autoconf) 2.60
openbsd 4.0 m4
automake (GNU automake) 1.9.6
```

To check the version of the tool you are using, use the GNU standard version option. For example

```
autoconf --version
```

If you successfully build the GPSTk using other versions of the tools, or if you successfully build in a new environment, please let us know.

The following procedure will build and install the GPSTK.

1. Ensure that prerequisites, except jam, have been installed.
2. Download the GPSTk distribution.
3. Extract the GPSTk tarball. For example, using GNU tar
Appendix A. Software Installation and Execution

4. Change into the gpstk/dev directory (if using Subversion) or the gpstk/ directory (if using the tarball) and execute these commands:

```
./autogen.sh
./configure
make
```

5. To build the library documentation using doxygen:

```
doxygen
```

6. To install GPSTk as a system library in /usr/local, assume root privileges then execute

```
make install
```

### A.3.2 GPStk Software Execution

First of all to execute a GPStk program we need to compile it. To compile the program and execute it in Linux with gcc would look like the following.

```
user@host:˜/gpstk/examples$ g++ -o example -I˜/gpstk/src
 -L˜/gpstk/src -lgpstk example1.cpp
```

Another simple option would be to add the program inside the folder `/gpstk/dev/examples`. Adding to the file `Makefile.am`:

```
bin_PROGRAMS = gpstkposition
gpstkposition_SOURCES = gpstkposition.cpp
```

Where `gpstkposition.cpp` is the program which we want to execute. So now to execute the program we only need to put:

```
user@host:˜/gpstk/examples$ make
user@host:˜/gpstk/examples$
```

### A.4 Getting RINEX files from Reference Stations

To be able to implement DGPS technique we need to download an observation RINEX file from a reference station. We suggest two options as reference station. The first one is download
Appendix A. Software Installation and Execution

the file from ICC (Institut CartogrÀ fic de Catalunya) http://www.icc.es/web/content/ca/prof/geodesia/catnet.html, we can get reference files for every place of Catalonia. The second source of observation reference file is the gAGE observatory, and we can get it from ftp://gage19.upc.es/.incoming. Whether you have downloaded the file from one or another site, it file will be compressed, so we need to unpack it. The following steps explain both processes.

ICC Reference Station

As we have said previously the reference observation file is compressed, so we need to unpack it. The ICC files are compressed with Hatanaka compressor. We can download Hatanaka decompressor from http://www.icc.es/web/content/ca/prof/geodesia/eines_geodesiques.html. Once we have downloaded the reference observation file and the decompressor software we can obtain the RINEX file executing the next command inside Hatanaka decompressor folder:

```
zcat AVEL0010.06E | ./CRX2RNX -> data
```

where AVEL0010.06E is the Hanataka file name, and data is the RINEX observation file.

gAGE Reference Station

The gAGE reference observation files are compressed with the ‘teqc’ software. We can download the decompressor from http://facility.unavco.org/software/teqc/teqc.html. Once we have downloaded the reference observation file and the decompressor software we can obtain the RINEX file executing the next command inside teqc decompressor folder:

```
./teqc -leica mdb +nav 5003040k.09n 5003040k.m00 > 5003040k.09o
```

Where 5003040k.m00 is the downloaded file and 5003040k.09o is the obtained reference observation file.

A.5 A completely execution, step by step.

At this section it will be explained step by step the instructions needed to execute all the GPS software. To these we will assume that we have two folders /Desktop/gps-sdr and /Desktop/gpstk which contain the GPS-SDR software and GPStk libraries respectively. First of all we need to execute GPS-SDR software, and then charge its results to the GPStk developed software. Thus the steps are:
Appendix A. Software Installation and Execution

1. Compile and execute GPS-SDR. We need to open two execution terminals.

(a) First terminal:

```plaintext
$ cd Desktop/gps-sdr/usrp
$ make
$ sudo su
$ ********** (Introduce Password)
$ cd ..
$ ./gps-usrp
```

(b) Second terminal:

```plaintext
$ cd Desktop/gps-sdr
$ make
$ ./gps-sdr -dd -g -l //These options execute double delta, google an rinex module outputs
```

With these previous instructions we have got three output files, the navigation.kml which is the Gogle Earth compatible navigation file, cttcddd1.09n and cttcddd1.09o the navigation and observation RINEX files respectively.

2. Charge RINEX files to GPStk. We only need to copy cttcddd1.09n and cttcddd1.09o to /Desktop/gpstk/dev/examples

3. Charge Reference Station RINEX observation file. Copy the observation file obtained from the reference station to /Desktop/gpstk/dev/examples. See A.4 to know how to get the file.

4. Execute gpstkposition.cpp. After adding gpstkposition.cpp to Makefile.am (see A.3.2), open a new terminal and insert:

```plaintext
$ cd Desktop/gpstk/dev/examples
$ make
$ ./gpstkposition
```

After these we will have six files: LMS, WMS and Kalman users position computed with normal and with DGPS technique. These files are compatibles with Google Earth tool.

A.6  Software Providing

As the installation of GNURadio is a little bit complex and large, we have built a Ubuntu Live DVD. Thanks to these live DVD we don’t need to have installed Ubuntu to our PC. We can
execute all GNURadio software just putting the DVD to our PC and start PC from DVD. We provide this DVD added to the project report. Also attached to the project report as a CD, we have added the developed GPS-SDR software. The `gpstkposition.cpp` file is also attached in GPS-SDR software CD.
Appendix B

USRP Hardware description

B.1 Hardware description

All the schematics and layouts are available in the repository. They can be retrieved by

svn co http://gnuradio.org/svn/usrp-hw/trunkusrp-hw

B.1.1 The Universal Software Radio Peripheral (USRP)

USRP Features:

- Four 64 Msp 12-bit Analog to Digital Converters (ADC: Analog Devices AD9862). Datasheet:
- Four 128 Msp 14-bit Digital to Analog Converters (DAC: also integrated in AD9862)
- Four digital downconverters (DDC) with programmable decimation rates
- Two digital upconverters (DUC) with programmable interpolation rates
- High-speed USB 2.0 interface (480 Mbps)
- Capable of processing signals up to 16 MHz wide
- Auxiliary analog and digital I/O support complex radio controls such as RSSI and AGC
The USRP can simultaneously receive and transmit on two antennas in real time. All sampling clocks and local oscillators are fully coherent, thus allowing you to create MIMO systems.

- Input:
  - Number of input channels: 4 (or 2 I-Q pairs)
  - Sample rate: 64 Msps
  - Resolution: 12 bits
  - SFDR: 85 dB. In ADCs, Spurious-Free Dynamic Range (SFDR) is the ratio of the RMS amplitude of the carrier frequency (maximum signal component) to the RMS value of the next largest noise or harmonic distortion component.

- Output:
Appendix B. USRP Hardware description

- Number of output channels: 4 (or 2 I-Q pairs)
- Sample rate: 128 Msps
- Resolution: 14 bits
- SFDR: 83 dB. In DACs, Spurious-Free Dynamic Range (SFDR) is the ratio of the RMS amplitude of the carrier frequency (maximum signal components) to the RMS value of their next largest distortion component.

- Auxiliary I/O:
  - High-speed digital I/O: 64 bits
  - Analog input: 8 channels
  - Analog output: 8 channels
  - Spurious Free Dynamic Range (SFDR): 83 dB

- Digital down-conversion and decimation stage AD9862 → CORDIC → Decimating CIC Filter (4 stage, Programmable decimation rate [4,128]) → Halfband Decimation Filter (Fixed decimation by 2) → RX FIFO. This yields possible decimation rates of all even numbers between [8,256].

![Digital down-conversion and decimation diagram]

Figure B.2: Digital down-conversion and decimation

The CORDIC (Coordinate Rotation Digital Computer) algorithms are designed to calculate transcendental functions, such as cosine and sine magnitude and phase (arctangent) to
any desired precision, by using only shift and add operations. CORDIC revolves around the idea of rotating the phase of a complex number, by multiplying it by a succession of constant values. However, the multiplies can all be powers of 2, so in binary arithmetic they can be done using just shifts and adds; no actual multiplier is needed. It is suited for FPGA implementations, because it does not use multipliers and need small look-up tables. On the other hand, when a hardware multiplier is available, e.g. in a DSP microprocessor, table-lookup methods and good old-fashioned power series are generally faster than CORDIC.

- Block diagram of the digital up-conversion stage

![Block diagram of the digital up-conversion stage](image)

Figure B.3: Digital up-conversion stage


<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LEs</strong></td>
<td>12060</td>
</tr>
<tr>
<td><strong>M4k RAM blocks (128 x 36 bits)</strong></td>
<td>52</td>
</tr>
<tr>
<td><strong>Total RAM bits</strong></td>
<td>239,616</td>
</tr>
<tr>
<td><strong>PLLs</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Maximum user I/O pins</strong></td>
<td>173</td>
</tr>
</tbody>
</table>
Appendix B. USRP Hardware description

The hardware language used to describe the functionality within the FPGA is written in Verilog and synthesized using Altera’s free web tool Quartus II. The FPGA runs off a 64MHz clock with every internal component synchronous to that global clock.

- Analog-to-digital converter: Analog Devices AD9862. 14 bit resolution, 64 Msp. The full range of the ADCs is 2V peak to peak, and the input is 50 Ω differential. This is 40 mW, or 16 dBm. There is a programmable gain amplifier (PGA) before the ADCs to amplify the input signal to utilize the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20 dB. With gain set to zero, full scale inputs are 2 Volts peak-to-peak differential. When set to 20 dB, only 0.2 Vpp differential input signal is needed to reach full scale. This PGA is software programmable. On the Rx path it has steps of 1.0 dB before ADC. On the Tx path, the steps are 0.1 dB after the DAC.

The DLL is used to double the sample rate so that the DACs can sample at 128 MHz. The ADCs stay at 64 MHz. The Hilbert filters are never used. As can be seen in Figure 5, in the transmission path the complex multiplication is split into two stages: coarse and fine. The fine part (block D) runs at 1/4 of the sample rate. This means that it can only move the frequency 1/4th as far. The block is then followed by the 4x interpolation (Block C), and then the coarse modulation, Block B. Block B only moves the signal +/- fs/4 or fs/8. The auxiliary ADC/DACs are used for AGC/VCO setting in the daughterboards. Details depend on the specific daughterboard.

The AD9862 is controlled over the Serial Peripheral Interface (SPI) bus, a synchronous serial data link standard named by Motorola that operates in full duplex mode. The code that sets up the AD9862’s is contained in usrp/host/lib/legacy/usrp_basic.ccandusrp/host/lib/legacy/usrp_standard.cc

- Maximum USB throughput: 32 MBps (bytes per second), which means 8 MHz bandwidth with 16 bit resolution and 16 MHz bandwidth with 8 bit resolution.
Figure B.4: Each AD9862 has two ADCs, two DACs and their corresponding Programmable Gain Amplifiers (PGA)
B.1.2 BasicRX daughterboard

Designed for use with external RF front ends as an intermediate frequency (IF) interface, the BasicRX daughterboard allows the direct reception of signals from 1 MHz to 250 MHz, fed into the USRP 64 Msps ADC directly. It has two channels A and B to capture two signals using real sampling or one complex signal on only one of the two channels. The analog input consist of two SMA connectors with 50 $\Omega$ of impedance followed by an input transformer on each channel that acts as a balun (converts from balanced to unbalanced electric signals because the USRP ADCs need a differential input, see Figure 4). In addition, the BasicRX board gives direct access
to all of the signals on the daughterboard interface (including 16 bits of high-speed digital I/O, SPI and I2C buses, and the low-speed ADCs and DACs), and as such are useful for developing your own daughterboards or custom FPGA designs. Also has a small pre-programmed 2 Kbyte I2C EEPROM to identify the board.

![Figure B.7: Physical aspect of the BasicRX board](image)

**B.1.3 BasicTX daughterboard**

The BasicTX daughterboard allows the direct transmission of signals from 1 MHz to 250 MHz, interfacing directly the two channel 128 Msps DAC. It has two 50 Ω SMA unbalanced output channels. The board has two baluns to convert the differential output from the DAC to common mode output. In addition, the board BasicTX gives direct access to all of the signals on the
daughterboard interface (including 16 bits of high-speed digital I/O, SPI and I2C buses, and the low-speed ADCs and DACs), and as such are useful for developing your own daughterboards or custom FPGA designs. Also has a small pre-programmed 2 Kbyte I2C EEPROM to identify the board.

Figure B.8: Front end schematic of the BasicRX board
B.1.4 LFRX and LFTX daughterboard

The LFRX and LFTX daughterboards allow the direct transmission and reception of signals from DC to 30 MHz. LFRX uses a Analog Devices AD8132 high speed differential amplifier to convert from common mode to differential mode and vice-versa, instead of a transformer balun. LFTX uses a general purpose voltage feedback OP AMP AD8047. In both boards bandwidth is limited with a RC filter to 30 MHz, acting like an anti aliasing filter. In addition the board LFRX and LFTX give direct access to all of the signals on the daughterboard interface (including 16 bits of high-speed digital I/O, SPI and I2C buses, and the low-speed ADCs and DACs), and as
such are useful for developing your own daughterboards or custom FPGA designs. Also has a small pre-programmed 2 Kbyte I2C EEPROM to identify the board.

### B.1.5 DBSRX daughterboard

This daughterboard receives from 800 MHz to 2.4 GHz. It is based on the MAX2118 Direct Conversion tuner Integrated Circuit with monolithic VCOs of Maxim (Noise Figure: 10 dB). It uses complex sampling, so it requires two ADC. Application note: [http://www.maxim-ic.com/appnotes.cfm/appnote_number/2240](http://www.maxim-ic.com/appnotes.cfm/appnote_number/2240)

The DBSRX is a complete receiver system for 800 MHz to 2.4 GHz with a 3-5 dB noise figure. The DBSRX features a software controllable channel filter which can be made as narrow as 1 MHz, or as wide as 60 MHz. The DBSRX frequency range covers many bands of interest, including all GPS and Galileo bands, the 902-928 MHz ISM band, cellular and PCS, the Hydrogen and Hydroxyl radio astronomy bands, DECT, and many more. The DBSRX is MIMO capable, and can power an active antenna via the coax.

![Figure B.10: Physical aspect and schematic of the LFRX board](image)
As is shown in B.14, after the SMA input there is a 17 dB amplifier (Agilent Technologies MGA82563, 0.1-6 GHz 3V, 17 dBm Amplifier, GaAsFET, Noise Figure: 0.9 dB. Datasheet: http://www.datasheetcatalog.org/datasheet/hp/MGA82563.pdf). The GC1 amplifier (0-56 dB) is set by a DAC, while the GC2 (0-24 dB) is set by the serial bus. The algorithm that splits gain within the amplifiers is located in gnuradio/gr-usrp/src/db_dbs_rx.py.

This is the code:
def set_gain(self, gain):
    if not (gain >= 0 and gain < 105):
        raise ValueError, "gain out of range"
    gc1 = 0
    gc2 = 0
    dl = 0
    pga = 0
    if gain < 56:
        gc1 = int((-gain * 1.85 / 56.0 + 2.6) * 4096.0 / 3.3)
        gain = 0
    else:
        gc1 = 0
        gain = gain - 56
    if gain < 24:
        gc2 = int(round(31.0 * (1 - gain / 24.0)))
        gain = 0

Figure B.12: Physical aspect and schematic of the LFTX board
Figure B.13: DBSRX Physical aspect (left) and diagram of the Direct Conversion tuner MAX2118 (right)

```python
else:
    gc2 = 0
    gain = gain - 24
if gain >= 4.58:
    dl = 1
    gain = gain - 4.58
pga = gain
self._set_gc1(gc1)
selself._set_gc2(gc2)
seltself._set_dl(dl)
seltself._set_pga(pga)
```
Design criteria:

- The Friis’ formula computes the noise figure or noise temperature of a system composed of a number of cascaded stages:

\[
F_{\text{total}} = F_1 + \frac{F_2 - 1}{G_1} + \frac{F_3 - 1}{G_1 G_2} + \frac{F_4 - 1}{G_1 G_2 G_3} + \ldots
\]

Figure B.14: DBSRX Block diagram

where Fn and Gn are the noise factor and available power gain, respectively, of the n-th stage. Note that both magnitudes are expressed as ratios, not in decibels. When the noise factor F is given in dB it is usually referred to as noise factor NF. The conversion between noise figure and noise temperature is:
Noise temperature (K): $T_{\text{Noise (K)}} = 290 \left(10^{\frac{N_{\text{F(db)}}}{10}} - 1\right)$

Noise Figure (dB): $N_{\text{F(dB)}} = 10 \log_{10} \left(\frac{T_{\text{Noise (K)}}}{290} + 1\right)$
Appendix C

GPStk Code

The next code belongs to the gpstkposition.cpp file which is the file used during the GPStk simulations in Chapter 5

// Example program Nro 7 for GPStk
// This program shows several different ways to process GPS data
// using GNSS Data Structures (DataStructures.hpp).

#include <iostream>
#include <iomanip>

// Class for handling satellite observation parameters RINEX files
#include "RinexObsStream.hpp"

// Classes for handling RINEX Broadcast ephemeris files
#include "RinexNavStream.hpp"
#include "RinexNavHeader.hpp"
#include "RinexNavData.hpp"

// Class in charge of the GPS signal modelling
#include "ModelObs.hpp"

// Class to store satellite broadcast navigation data
#include "GPSEphemerisStore.hpp"

// Class to model the tropospheric delays
#include "TropModel.hpp"
// Classes to model and store ionospheric delays
#include "IonoModel.hpp"
#include "IonoModelStore.hpp"

// Class to solve the equation system using Least Mean Squares
#include "SolverLMS.hpp"

// Class to solve the equation system using Weighted-Least Mean Squares
#include "SolverWMS.hpp"

// Class to solve equations systems using a simple code-based Kalman filter
#include "CodeKalmanSolver.hpp"

// Class defining the GNSS data structures
#include "DataStructures.hpp"

// Class to filter out observables grossly out of limits
#include "SimpleFilter.hpp"

// Class for easily changing reference base from ECEF to NEU
#include "XYZ2NEU.hpp"

// Class to detect cycle slips using just one frequency
#include "OneFreqCSDetector.hpp"

// Class to detect cycle slips using LI combination
#include "LICSDetector.hpp"

// Class to detect cycle slips using the Melbourne-Wubbena combination
#include "MWCSDetector.hpp"

// Class to compute weights according to Appendix J of MOPS C (RTCA/DO-229C)
#include "ComputeMOPSWeights.hpp"

// Class to smooth code observables (by default, C1)
#include "CodeSmoother.hpp"

// Class to smooth the PC combination
#include "PCSmother.hpp"

// Classes to compute several combinations
#include "ComputePC.hpp"
#include "ComputeLC.hpp"
#include "ComputeLI.hpp"
#include "ComputeMelbourneWubbena.hpp"

// Class to compute single differences between receiver stations
#include "DeltaOp.hpp"

// Class to synchronize two GNSS Data Structures data streams.
#include "Synchronize.hpp"

#include "geometry.hpp"

using namespace std;
using namespace gpstk;

void GoogleEarthHeader(){
    ofstream out("gps_gpstk_solver_LMS.kml",ios_base::app| ios::binary);
    out<<"<?xml version="1.0" encoding="UTF-8"?>\n";
    out<<"<kml xmlns="http://earth.google.com/kml/2.1">\n";
    out<<"<Document >\n";
    out<<"<name >\n";
    out<<"GPS_GPSTK_SOLVER_LMS.kml\n";
    out<<"</name >\n";
    out<<"<Placemark id="CTTC">\n";
    out<<"<name >\n";
    out<<"GPS_GPSTK_SOLVER_LMS.kml\n";
    out<<"</name >\n";
    out<<"<visibility >\n";
}
Appendix C. GPStk Code

```cpp
void GoogleEarthFooter()
{
    fstream out("gps_gpstk_solver_LMS.kml", ios_base::app | ios::binary);
    out <<"1\n";
    out <<"</visibility>\n";
    out <<"</description>\n";
    out <<"<![CDATA[]]>\n";
    out <<"</description>\n";
    out <<"<Style>\n";
    out <<"<LineStyle>\n";
    out <<"<color>\n";
    out <<"#FF0000FF\n";
    out <<"</color>\n";
    out <<"<width>\n";
    out <<"1.00\n";
    out <<"</width>\n";
    out <<"</LineStyle>\n";
    out <<"<PolyStyle>\n";
    out <<"<color>\n";
    out <<"00 ffffff\n";
    out <<"</color>\n";
    out <<"</PolyStyle>\n";
    out <<"</Style>\n";
    out <<"<Polygon id="poly_plot3">
    out <<"<extrude>0</extrude>
    out <<"<altitudeMode>relativeToGround
    out <<"</altitudeMode>
    out <<"<outerBoundaryIs>
    out <<"<extrude>0</extrude>
    out <<"<LinearRing>
    out <<"<extrude>0</extrude>
    out <<"<tessellate>0</tessellate>
    out <<"<altitudeMode>absolute
    out <<"</altitudeMode>
    out <<"<coordinates>
```

int main(void) 
{

    ///////// Initialization phase ///////////

    ///////// COMMON OBJECTS /////////////

    cout << fixed << setprecision(8); // Set a proper output format

    RinexNavData rNavData; // Object to store Rinex navigation data
    GPSEphemerisStore bceStore; // Object to store satellites ephemeris
    RinexNavHeader rNavHeader; // Object to read the header of Rinex
    // navigation data files
    IonoModelStore ionoStore; // Object to store ionospheric models
    IonoModel ioModel; // Declare a Ionospheric Model object

    //Create the input observation file stream
    //This is a fixed station, but here it will play as "rover"
    //RinexObsStream rin("ebre030a.02o");
    //RinexObsStream rin("temp0681.09o");
    RinexObsStream rin("cttcddd1.09o");
    // Please note that data was collected in year 2002, when the Sun
    // was very active

    // Create the input navigation file stream
    // RinexNavStream rnavin("brdc0300.02n");
    RinexNavStream rnavin("cttcddd1.09n");

    //We need to read ionospheric parameters (Klobuchar model) from header
    rnavin >> rNavHeader;
// Let’s feed the ionospheric model (Klobuchar type) from data in the
// Navigation file header
ioModel.setModel(rNavHeader.ionAlpha, rNavHeader.ionBeta);
// Beware: In this case, the same model will be used for the
// full data span
ionoStore.addIonoModel(DayTime::BEGINNING_OF_TIME, ioModel);

// Storing the ephemeris in "bceStore"
while (rnavin >> rNavData)
{
bceStore.addEphemeris(rNavData);
}
bceStore.SearchPast();  // This is the default

// CTTC station nominal position
Position nominalPos( 4797680.48851895, 166491.38275179,
                     4185448.59959262 );
// ParkinG EPSC
// Position nominalPos( 4797552, 166714, 4185528 );
// PALAU
// Position nominalPos( 4772234, 73174, 4217134 );

// Position nominalPos( 4774382.847, 49690.077, 4215084.815);
// Declare a MOPSTropModel object, setting the defaults
/* MOPSTropModel mopsTM( nominalPos.getAltitude(),
                        nominalPos.getGeodeticLatitude(),
                        30 );*/
MOPSTropModel mopsTM( nominalPos.getAltitude(),
                      nominalPos.getGeodeticLatitude(),
                      153 );

// Declare the modeler object, setting all the parameters in one pass
// Given that in this example we are using a fixed GPS station with known
// coordinates, you could have used the "ModeledReferencePR" class, which
// is a little bit simpler.
// However, for a rover is more appropriate to use a "ModelObs" object
// because it allows to update the apriori position more easily (and it
// may automatically compute one, if needed, using Bancroft’s method
ModelObs model(nominalPos, ionoStore, mopsTM, bceStore, TypeID::C1);

// On the other hand, the usual way to use "ModelObs" is setting just the
// models in the constructor, and calling method "Prepare()" later, like
// in the following lines:
// ModelObs model(ionoStore, mopsTM, bceStore, TypeID::C1);
// model.Prepare(nominalPos);
// Set the reference position

// Declare a simple filter object. By default, it filters C1 with
// default limits
SimpleFilter myFilter;

// This is the GNSS data structure that will hold all the
// GNSS-related information
gnssRinex gOriginal;

/////////CASE #1 OBJECTS/////////

// Declare a SolverLMS object
SolverLMS solver;

/////////CASE #2 OBJECTS/////////

// Declare a base-changing object: From ECEF to North-East-Up (NEU)
XYZ2NEU baseChange(nominalPos);

// For some examples we need to reconfigure the solver in order
// to use a NEU system
TypeIDSet typeSet;
typeSet.insert(TypeID::dLat);
typeSet.insert(TypeID::dLon);
typeSet.insert(TypeID::dH);
typeSet.insert(TypeID::cdt);

// This is the proper equation structure to use with a NEU system
gnssEquationDefinition newEq(TypeID::prefitC, typeSet);
Appendix C. GPStk Code

// Declare another SolverLMS object, but configure it to use a
topocentric reference system (North-East-Up: NEU)
SolverLMS solverNEU;
solverNEU.setDefaultEqDefinition(newEq);
// NEU reconfiguration

/////////////////////////////////////
// ////////// CASE #3 OBJECTS //////////

// This object will compute the appropriate MOPS weights
ComputeMOPSWeights mopsW(nominalPos, bceStore);

// Declare a solver object using Weighted-Least-Mean-Squares and
// a topocentric reference system (NEU)
SolverWMS solverWMS;
// solverWMS.setDefaultEqDefinition(newEq);
// NEU reconfiguration

/////////////////////////////////////
// ////////// CASE #4 OBJECTS //////////

// Let's declare a cycle slip detector using just one frequency
OneFreqCSDetector markCSC1;

// Declare an object to smooth code (C1 by default)
CodeSmoother smoothC1;
smoothC1.setMaxWindowSize(8); // Configure smoother for 30 s sampling

/////////////////////////////////////
// ////////// CASE #6 OBJECTS //////////

// Object to compute the PC (ionosphere-free) combination.
ComputePC getPC;
// Use C1 instead of P1. P1 observables are declared in available RINEX
// files, but often they are indeed missing (like in this case). When
// that happens, this step is mandatory
getPC.useC1();

// Declare a simple filter object to screen PC
SimpleFilter pcFilter;
pcFilter.setFilteredType(TypeID::PC);
// Declare the modeler object for PC, setting all the parameters
// in one pass
ModelObs modelPC(nominalPos, mopsTM, bceStore, TypeID::PC, false);
// Take notice that PC combination doesn’t use ionosphere modelling, nor
// TGD computation.
// WARNING: When using C1 instead of P1 to compute PC combination, be
// aware that instrumental errors will NOT cancel, introducing a bias
// that must be taken into account by other means. This will not work out
// in this example.

CCCCCC-----------------------------

CCCCCC CASE #10 OBJECTS

This is the GNSS data structure that will hold the
reference station data
gnssRinex gRef;

// Create the input observation file stream for REFERENCE STATION
RinexObsStream rinRef("5003153t.09o");

// GAGE reference station nominal position
Position nominalPosRef(4797642.279, 166436.150, 4185504.637);

// Create the input observation file stream for REFERENCE STATION LLEIDA
RinexObsStream rinRef("data.09o");

// LLEIDA reference station nominal position
// Position nominalPosRef(4774382.847, 49690.077, 4215084.815);

// Declare a MOPSTropModel object for the reference station, setting
// the defaults
MOPSTropModel mopsTMRef( nominalPosRef.getAltitude(),
    nominalPosRef.getGeodeticLatitude(),
    153 );

// Declare the appropriate modeler object for a reference station
ModelObsFixedStation modelRef(nominalPosRef,
    ionoStore,
    mopsTMRef,
    bceStore,
    TypeID::C1 );
// Create an object to compute the single differences of prefit residuals
DeltaOp delta; // By default, it will work on code prefit residuals

// Create an object to synchronize rover and reference station
// data streams. This object will take data out from "rinRef" until
// it is synchronized with data in "gOriginal". Default synchronization
// tolerance is 1 s.
Synchronize synchro(rinRef, gOriginal);

// ///////////////////////////////////////////
// ////////// CASE #12 OBJECTS /////////////

// Declare a new Kalman solver, already reconfigured for NEU system
// CodeKalmanSolver solverK12(newEq);

// Declare a new Kalman solver, configured for ECEF system
// WhiteNoiseModel newCoordinatesModel(100.0);
CodeKalmanSolver solverK12;

// solverK12.setCoordinatesModel(&newCoordinatesModel);

// //////////////////////////////////////
// ////////// CASE #13 OBJECTS ///////////

// Declare a new Kalman solver, already reconfigured for NEU system
// CodeKalmanSolver solverK12(newEq);

// Declare a new Kalman solver, configured for ECEF system
// WhiteNoiseModel newCoordinatesModel(100.0);
CodeKalmanSolver solverK13;

// //////////////////////////////////////////////////////////////////////////

// ///// End of initialization phase /////

// OPEN THE OUTPUT FILE
ofstream out("gps_gpstk_solver_LMS.kml", ios_base::app | ios::binary);
if (!out) {
cout << "No es posible abrir fichero OUTPUT\n";
return 1;
}

GoogleEarthHeader();
/****** Processing phase /////////

// Loop over all data epochs
while (rin >> gOriginal)
{
    // Let’s output the time stamp (in seconds of day)
    // cout << gOriginal.header.epoch.DOYsecond() << " 
    ";//Output field #1
    ////////////////// CASE #1 /////////////////////

    // This case is a common C1 + Least Mean Squares solver
    // (LMS) processing
    // Let’s make a working copy
    gnssRinex gRin1(gOriginal);

    try
    {
        // This is the line that will process all the GPS data
        gRin1 >> myFilter >> model >> solver;
gRin1.keepOnlyTypeID(TypeID::C1) >> myFilter >> model >> solver;
        // - First, a basic filter to screen out very bad observables
        // - Second, apply a model to the observables (ionosphere,
        // troposphere, relativity, etc.)
        // - Third, solve the equations using a simple Least-Mean-Squares
        // solver
    }
    catch(...)
    {
        cerr << "Case Exception at epoch: " << gRin1.header.epoch << endl;
    }

    // Get your results out of the solver object. In ECEF system
    // by default
    Position solPos( (model.rxPos.X() + solver.getSolution(TypeID::dx)),
                    (model.rxPos.Y() + solver.getSolution(TypeID::dy)),
                    (model.rxPos.Z() + solver.getSolution(TypeID::dz)) );

    out << setprecision(10) << solPos.longitude() << " 
    ";//Output field #5
    out << solPos.geodeticLatitude() << " 
    ";"; //Output field #6
out << solPos.height() << "\n";
// Output field #7
out << endl;

/////////////////// END OF CASE #1
///////////////////

/////////////////// CASE #3 /////////////

In this case we process data using C1 + Weighted Least Mean Squares
// solver (WMS)

// Let’s make a working copy
gnssRinex gRin3(gOriginal);

try
{
gRin3 >> myFilter >> model >> mopsW >> solverWMS;
// The "mopsW" object computes weights based on MOPS algorithm
// The "solverWMS" object solves the system using Weighted Least
// Mean Squares. It is already configured to work with NEU system.
}
catch(...)
{
cerr << "Case 3. Exception at epoch: " << gRin3.header.epoch << endl;
}

Position solPos3( (model.rxPos.X() + solverWMS.getSolution(TypeID::dx)),
                 (model.rxPos.Y() + solverWMS.getSolution(TypeID::dy)),
                 (model.rxPos.Z() + solverWMS.getSolution(TypeID::dz)));

// An alternative way to report the solution is to access it
// using the TypeID’s defined in the "gnssEquationDefinition" object
// assigned to the solver.
// With this method we avoid the possibility of getting the wrong
// type of solution from the "solution" vector.

out3 << setprecision(10)<< solPos3.longitude() << "\n"; // Output field #5
out3 << solPos3.geodeticLatitude() << "\n"; // Output field #6
out3 << solPos3.height() << " \n"; // Output field #7
out3 << endl;

END OF CASE #3

CASE #10

This is like cases #1 and #2, but using DGPS techniques instead.
Let's make a working copy of rover data
gnssRinex gRin10(gOriginal);

// First, let's synchronize and process reference station data
try
{
gRef >> synchro >> myFilter >> modelRef;
// Please note that the FIRST STEP is to synchronize "gRef", the
// reference station data stream, with "gOriginal" (or with gRin10,
// which is the same), the rover receiver data stream.
// Also, remember that in simple DGPS the differences are computed
// on code prefit residuals, so "modelRef" object is mandatory.
// The "delta" object will take care of proper differencing.
// We must tell it which GNSS data structure will be used
// as reference
delta.setRefData(gRef.body);
}

catch(SynchronizeException& e) // THIS IS VERY IMPORTANT IN ORDER TO
// MANAGE A POSSIBLE DESYNCHRONIZATION
{
cout << endl;
continue;
}

cerr << "Case 10. Exception when processing reference station data at epoch: " << gRef.header.epoch << endl;
// Rover data processing is done here:
try
{
  gRin10 >> myFilter >> model >> delta >> solver;
  // This is very similar to cases #1 and #2, but we insert a "delta"
  // object that will adjust code prefit residuals BEFORE solving the
  // system of equations.
} 
catch(...)
{
cerr << "Case10. Exception at epoch: " << gRin10.header.epoch
  << endl;
}

Position solPos10( (model.rxPos.X() + solver.getSolution(TypeID::dx)),
  (model.rxPos.Y() + solver.getSolution(TypeID::dy)),
  (model.rxPos.Z() + solver.getSolution(TypeID::dz)));

// An alternative way to report the solution is to access it
// using the TypeID’s defined in the "gnssEquationDefinition" object
// assigned to the solver.
// With this method we avoid the possibility of getting the wrong
// type of solution from the "solution" vector.
out10 <<setprecision(10)<< solPos10.longitude() << "
  ";//Outputfield #5
out10 << solPos10.geodeticLatitude() << "
  ";//Output field #6
out10 << solPos10.height() << "
  ";//Output field #7
out10<<endl;

@end{case #10}

// This is like case #10 (DGPS), but now let’s apply a WMS solver
// on data
// Let's make a working copy
gnssRinex gRin11(gOriginal);

// Please note that data streams are already synchronized, and
// "delta" object may be reused with the same reference data
// obtained from Case #10.
// First, let's synchronize and process reference station data
try
{
    gRef >> synchro >> myFilter >> modelRef;
    // Please note that the FIRST STEP is to synchronize "gRef", the
    // reference station data stream, with "gOriginal" (or with gRin10,
    // which is the same), the rover receiver data stream.
    // Also, remember that in simple DGPS the differences are computed
    // on code prefit residuals, so "modelRef" object is mandatory.

    // The "delta" object will take care of proper differencing.
    // We must tell it which GNSS data structure will be used
    // as reference
    delta.setRefData(gRef.body);
}
catch(SynchronizeException& e) // THIS IS VERY IMPORTANT IN ORDER TO
    // MANAGE A POSSIBLE DESYNCHRONIZATION!!
{
    cout << endl;
    continue;
}
catch(...)
{
    cerr << "Case 11. Exception when processing reference station data at epoch:"
        << gRef.header.epoch << endl;
}

try
{
    gRin11 >> myFilter >> model >> delta >> mopsW >> solverWMS;
    // Like case #10, but now with "mopsW" and "solverWMS"
}
catch(...)
Appendix C. GPStk Code

```cpp
cerr << "Case11. Exception at epoch: " << gRin11.header.epoch << endl;

Position solPos11((model.rxPos.X() + solverWMS.getSolution(TypeID::dx)),
                   (model.rxPos.Y() + solverWMS.getSolution(TypeID::dy)));

out11 << setprecision(10) << solPos11.longitude() << "\n"; // Output field #5
out11 << solPos11.geodeticLatitude() << "\n"; // Output field #6
out11 << solPos11.height() << "\n"; // Output field #7
out11<<endl;

// END OF CASE #11

// CASE #12

try
{
gRin12 >> myFilter >> model >> mopsW
       >> solverK12;
}

catch(...)
{
cerr << "Case12. Exception at epoch: " << gRin12.header.epoch << endl;
}

Position solPos12((model.rxPos.X() + solverK12.getSolution(TypeID::dx)),
                   (model.rxPos.Y() + solverK12.getSolution(TypeID::dy)));
```
(model.rxPos.Z() + solverK12.getSolution(TypeID::dz))

out12 << setprecision(10) << solPos12.longitude() << " \textdegree"; // Output field #5
out12 << solPos12.geodeticLatitude() << " \textdegree"; // Output field #6
out12 << solPos12.height() << " \text{m}"; // Output field #7
out12 << endl;

////////// END OF CASE #12


/////////// CASE #13 /////////////

// This is like case #11 (DGPS), but now let's apply a simple Kalman filter on data with kinematic movement

// Let's make a working copy
gnssRinex gRin13(gOriginal);
try {
gRef >> synchro >> myFilter >> modelRef;
delta.setRefData(gRef.body);
}
catch(SynchronizeException& e) {
    // THIS IS VERY IMPORTANT IN ORDER TO
    // MANAGE A POSSIBLE DESYNCHRONIZATION!!
    cout << endl;
    continue;
}
catch(...) {
cerr << "Case_13 Exception when processing reference station data at epoch:" << gRef.header.epoch << endl;
}

try {

gRin13 >> myFilter >> model >> delta >> mopsW >> solverK13;
// Like case #11, but now with "solverK13"
// VERY IMPORTANT: Note that in this case the coordinates are handled as constants, whereas the receiver clock is modeled as white noise.
}
catch(...)
{
cerr << "Case_13 Exception at epoch: " << gRin13.header.epoch << endl;
}

Position solPos13((model.rxPos.X() + solverK13.getSolution(TypeID::dx)),
(model.rxPos.Y() + solverK13.getSolution(TypeID::dy)),
(model.rxPos.Z() + solverK13.getSolution(TypeID::dz)));

// An alternative way to report the solution is to access it using the TypeID's defined in the "gnssEquationDefinition" object assigned to the solver.
// With this method we avoid the possibility of getting the wrong type of solution from the "solution" vector.
out13 << setprecision(10) << solPos13.longitude() << "\n";// Output field #5
out13 << solPos13.geodeticLatitude() << "\n";// Output field #6
out13 << solPos13.height() << "\n";// Output field #7
out13 << endl;

// END OF CASE #13

// End of data processing for this epoch
}
GoogleEarthFooter();
exit(0);
Appendix D

GPStk algorithms positioning accuracy analysis

This appendix explains the steps followed to compute the GPStk position accuracy in horizontal mode (2D) and vertical mode. First of all to compute the accuracy we need to know some concepts and definitions. The accuracy of a GPS receiver is organized depending on the distribution of his position samples, and more precisely they are classified depending on the distance between the position samples and the real position.

To understand the accuracy features we need to define the RMS (Root-Mean-Squared) error which is defined as:

\[
\text{RMS}_{\text{Error}} = \sqrt{\frac{\sum_{i=1}^{N} \text{Error}^2_i}{N}}
\]

The distribution of GPS fixes of a position may be approximated by a bivariate normal distribution with no correlation between the two variables. Sometimes this distribution has been inaccurately called "Gaussian"; but only a "slice" in any direction will indeed be a normal (Gaussian) distribution. For simplicity, one might assume the same variance in each direction (measurements show this is not quite actually true). With those approximating assumptions, the error distribution can be described by a very simple equation, which is known as a Rayleigh distribution. The probability function which determinates an error less than the distance is:

\[
\text{Probability}(\text{Error} \leq \text{Distance}) = 1 - \exp\left(-\frac{\text{Distance}^2}{\text{RMS}_{\text{Error}}^2}\right)
\]

From this probability function plotting it could be defined some important and characteristic values for a GPS receiver:
Appendix D. GPS\textit{tk} algorithms positioning accuracy analysis

- RMS error: is the distance which determinates the probability percentage of points which are at a distance less or equal than the RMS error. This percentage is usually about 63%
- CEP (Circular Error Probable): indicates the median or 50% error distance
- 95 error %: the 95% error distance will always be referred as the distance which determinates that the 95% of the points are as a maximum at a distance equal to the 95% error distance.

Assuming these previous concepts and definitions, the calculations are as follow.

D.1 Computing the horizontal accuracy

To compute the horizontal accuracy the next MATLAB file has been used:

```matlab
%Load the file which contains the position sample in LLA
load position_lla.txt
A=position_lla;

%Convert LLA to ECEF
for i=1:length(A(:,1))
    [x(i),y(i),z(i)]=lla2ecef(A(i,2)*pi/180,A(i,1)*pi/180,71.05);
end
plot(y,x)

%Convert real position from LLA to ECEF
[xref,yref,zref]=lla2ecef(41.274814*pi/180,1.9876139*pi/180,71.05);

%Compute RMS error
for i=1:length(A(:,1))
    error(i)= (x(i)-xref)^2+(y(i)-yref)^2;
end
numerror=sqrt(sum(error)/length(A(:,1)))
```
Appendix D. GPS tool algorithms positioning accuracy analysis

\[ d = 0:0.1:(3 \times \text{numerror}); \]

% Compute the Probability function
\[ \text{prob} = 1 - \exp\left(-\left(\frac{d}{\text{numerror}}\right)^2\right); \]

figure;

% Plot the probability function
\[ \text{plot}\left(\frac{d}{\text{numerror}}, \text{prob}\right); \]
\[ \text{title}'\text{Distance/RMS vs Probability}' \]
\[ \text{xlabel}'\text{Distance/RMS (Multiply by RMS to get distance)}' \]
\[ \text{ylabel}'\text{Probability that point is less than distance}' \]

So once time we have a file with the position samples it is directly to compute the probability function which determinates an error less than a distance. The probability functions obtained for a observation which were done during a 120 seconds observation for a static receiver position at CTTC’s location (Castelldefels) on June 2, 2009.

<table>
<thead>
<tr>
<th>Position</th>
<th>Latitude (degrees)</th>
<th>Longitude (degrees)</th>
<th>Altitude (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>41,2747</td>
<td>1.9875</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visible Satellites (SV Identifier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12, 17, 18, 22, 30</td>
</tr>
</tbody>
</table>

Table D.1: Simulated Signal Parameters

It is important to note that the results for the LMS and WMS algorithms are the same because when the samples were got each satellite power are similar, and consequently the weight ponderation didn’t affect the results. Also it’s important to note that the distribution of the position samples have this particular shape (ellipsoidal) because during the observations we were tracking satellites from only one site of the receiver.
Appendix D. GPStk algorithms positioning accuracy analysis

LMS and WMS Probability Function

![Figure D.1: Probability function for WMS and LMS algorithms (Horizontal 2D).](image1)

LMS and WMS + DGPS Probability Function

![Figure D.2: Probability function for WMS and LMS algorithms (Horizontal 2D).](image2)
Appendix D. GPS tk algorithms positioning accuracy analysis

Kalman Probability Function

Figure D.3: Probability function for WMS and LMS algorithms (Horizontal 2D).

Kalman + DGPS Probability Function

Figure D.4: Probability function for WMS and LMS algorithms (Horizontal 2D).
From the previous probability functions it is easy to compute the characteristic values which are showed in Table D.2.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CEP (meters)</th>
<th>RMS (meters)</th>
<th>95% (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS</td>
<td>23.4988</td>
<td>28.2608</td>
<td>49.8911</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>13.0368</td>
<td>15.6317</td>
<td>27.0584</td>
</tr>
<tr>
<td>WMS</td>
<td>23.4988</td>
<td>28.2608</td>
<td>49.8911</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>13.0368</td>
<td>15.6317</td>
<td>27.0584</td>
</tr>
<tr>
<td>Kalman</td>
<td>13.8136</td>
<td>16.5771</td>
<td>28.7115</td>
</tr>
<tr>
<td>Kalman+DGPS</td>
<td>4.8617</td>
<td>5.7981</td>
<td>10.1234</td>
</tr>
</tbody>
</table>

Table D.2: Horizontal Position Accuracy (2D)

## D.2 Computing the vertical accuracy

For the vertical accuracy, the steps are very similar. Now we will compute the vertical RMS error as:

\[
Vertical_{RMSError} = \sqrt{\frac{\sum_{i=1}^{N}(MeasH_t - TrueH_t)^2}{N}}
\]

Now the probability function is as as was a ”slice” of the horizontal probability function, so it could be considered as a Gaussian distribution. So the probability function can be defined as:

\[
Probability(HeightError \leq Error) = 1 - \exp\left(-\frac{Error}{Vertical_{RMSError}}\right)^2
\]

To compute the horizontal accuracy the next MATLAB file has been used:

```matlab
%Load the file which contains the position sample in LLA
load position_lla.txt
A=position_lla;

%Convert LLA to ECEF
for i=1:length(A(:,1))
```
Appendix D. GPStk algorithms positioning accuracy analysis

```matlab
[x(i),y(i),z(i)] = lla2ecef(A(i,2)*pi/180,A(i,1)*pi/180,71.05);
end
plot(y,x)

% Convert real position from LLA to ECEF
[xref,yref,zref] = lla2ecef(41.274814*pi/180,1.9876139*pi/180,71.05);

% Compute Vertical RMS error
for i=1:length(A(:,1))
    error(i) = (z(i) - zref)^2;
end

numerror = sqrt(sum(error)/length(A(:,1)))
d = 0:0.1:(3*numerror);

% Compute the Probability function
prob = 1 - exp(-(d/numerror).^2);

figure;
plot(d/numerror,prob);
title('Distance/RMS vs Probability')
xlabel('Distance/RMS (Multiply by RMS to get distance)')
ylabel('Probability that point is less than distance')
```

So once time we have a file with the position samples it is directly to compute the probability function which determinates an error less than a distance. The probability functions obtained for a observation which were done during a 120 seconds observation for a static receiver position at CTTC’s location (Castelldefels) on June 2, 2009.
Appendix D. GPStk algorithms positioning accuracy analysis

<table>
<thead>
<tr>
<th></th>
<th>Latitude (degrees)</th>
<th>Longitude (degrees)</th>
<th>Altitude (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>41.2747</td>
<td>1.9875</td>
<td>30</td>
</tr>
</tbody>
</table>

Visible Satellites (SV Identifier)

12, 17, 18, 22, 30

Table D.3: Simulated Signal Parameters

It is important to note that the results for the LMS and WMS algorithms are the same because when the samples were got each satellite power are similar, and consequently the weight ponderation didn’t affect the results.

LMS and WMS Probability Function

Figure D.5: Probability function for WMS and LMS algorithms (Vertical).
LMS and WMS + DGPS Probability Function

Figure D.6: Probability function for WMS and LMS algorithms (Vertical).

Kalman Probability Function

Figure D.7: Probability function for WMS and LMS algorithms (Vertical).
Appendix D. GPStk algorithms positioning accuracy analysis

Kalman + DGPS Probability Function

From the previous probability functions it is easy to compute the characteristic values which are showed in Table D.4.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CEP (meters)</th>
<th>RMS (meters)</th>
<th>95% (meters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS</td>
<td>8.8000</td>
<td>10.5013</td>
<td>18.1987</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>4.7988</td>
<td>5.7061</td>
<td>9.9000</td>
</tr>
<tr>
<td>WMS</td>
<td>8.8000</td>
<td>10.5013</td>
<td>18.1987</td>
</tr>
<tr>
<td>LMS+DGPS</td>
<td>4.7988</td>
<td>5.7061</td>
<td>9.9000</td>
</tr>
<tr>
<td>Kalman</td>
<td>4.7976</td>
<td>5.6844</td>
<td>9.7999</td>
</tr>
<tr>
<td>Kalman+DGPS</td>
<td>0.9999</td>
<td>1.1930</td>
<td>2.0877</td>
</tr>
</tbody>
</table>

Table D.4: Vertical Position Accuracy
Appendix E

Comercial GPS Receivers Features

This section of the appendix includes a part GPS World magazine publication of January 2009. This part shows the features of the great majority GPS receivers which are at present.
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Channels/Tracking Mode</th>
<th>Signal Tracking</th>
<th>Max No. Subsatellite Tracks</th>
<th>User Environment and Application</th>
<th>Size (L x W x D)</th>
<th>WH</th>
<th>Position Accuracy</th>
<th>Time (min)</th>
<th>Pos. Fit Update with Data (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DORO Technologies, LLC</td>
<td>DO-71</td>
<td>12 parallel</td>
<td>Channels</td>
<td>12</td>
<td>12 AGURV/1</td>
<td>118 x 114 x 45 mm</td>
<td>540 g</td>
<td>10 m / 70 cm / 2DRMS</td>
<td>10 Hz</td>
<td>10 Hz</td>
</tr>
<tr>
<td>Position Co. Ltd.</td>
<td>DO-70</td>
<td>20 par.</td>
<td></td>
<td></td>
<td></td>
<td>118 x 114 x 45 mm</td>
<td>540 g</td>
<td>10 m / 70 cm / 2DRMS</td>
<td>10 Hz</td>
<td>10 Hz</td>
</tr>
<tr>
<td>Precise Time and Frequency, Inc.</td>
<td>DO-50</td>
<td>12 par.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Quality Design | DO-50 | 12 par. | | | | | | | | }

**Accuracy:**

- **Horizontal Accuracy:**
  - ±0.3–0.6º per m
  - ±0.3–0.6º per m
  - ±0.3–0.6º per m

- **Position Fix:**
  - 20 Hz option
  - 20 Hz option
  - 20 Hz option

- **Time:**
  - 10 Hz
  - 20 Hz
  - 20 Hz

**Positioning:**

- **Static:**
  - 1 ppm / 5 mm + 1 ppm
  - 1 ppm / 5 mm + 1 ppm
  - 1 ppm / 5 mm + 1 ppm

- **Mobile:**
  - 1 ppm / 5 mm + 1 ppm
  - 1 ppm / 5 mm + 1 ppm
  - 1 ppm / 5 mm + 1 ppm
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Channels/ Tracking Mode</th>
<th>Signal Tracking</th>
<th>Max No. Satellites</th>
<th>Satisfactory</th>
<th>User Environment and Applications</th>
<th>Size (H x W x D)</th>
<th>Wt.</th>
<th>Position Accuracy</th>
<th>Time (min)</th>
<th>Pos. Fix</th>
<th>Update with Rate, Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td>SiGe Semiconductor</td>
<td>SiGe</td>
<td>NA</td>
<td>L1</td>
<td>4 x 4 mm</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>Model</td>
<td>Channels/Tracking Mode</td>
<td>Signal Tracking</td>
<td>Max No.</td>
<td>Satellite</td>
<td>Size</td>
<td>V.</td>
<td>Position Accuracy</td>
<td>Time (min)</td>
<td>Pos. Fix Update (with talk/sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>------------------------</td>
<td>----------------</td>
<td>---------</td>
<td>----------</td>
<td>------</td>
<td>---</td>
<td>------------------</td>
<td>------------</td>
<td>-------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>TRM-P/2104</td>
<td>12 par</td>
<td>L1, C/A -code</td>
<td>12</td>
<td>ADGL/NSM/PT2</td>
<td>3.175 x 0.497 x 3.50 in</td>
<td>5 in</td>
<td>&lt;5 in</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRM-5</td>
<td>16 par</td>
<td>L1, C/A -code</td>
<td>16</td>
<td>ADGL/NSM/PT2</td>
<td>5.1 x 1.0 x 0.8 in</td>
<td>0.5 in</td>
<td>2.5 in</td>
<td>2 min</td>
<td>CP</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Euro Star DS-2000</td>
<td>24/48</td>
<td>GPS &amp; GLO, SCA, L1</td>
<td>48 in view</td>
<td>ALMNTY</td>
<td>4.5 x 6 x 6 in</td>
<td>10 g</td>
<td>3 min</td>
<td>1 m</td>
<td>KEP</td>
<td>15/20</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>S molds</td>
<td>42 channels</td>
<td>GPS, L1, L2, C/A code</td>
<td>12</td>
<td>CHL/MTI</td>
<td>HCOLP: 0.6 x 2.4 x 0.8 mm, 54 ball, 0.44 mm pitch</td>
<td>TEBGA: 4 x 0.8 x 0.8 mm, 54 ball, 0.35 mm pitch</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Texas Instruments</td>
<td>T885B</td>
<td>24</td>
<td>GPS</td>
<td>20 WAAS/EGNOS capable</td>
<td>20</td>
<td>A</td>
<td>4.4 x 3.2 x 0.2 in</td>
<td>125 g</td>
<td>3 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Topcon</td>
<td>TO 2000</td>
<td>20</td>
<td>L1, C/A code</td>
<td>20</td>
<td>LTI</td>
<td>4 x 1.6 x 5 in</td>
<td>&lt;2 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TO 2000</td>
<td>12 par</td>
<td>L1, C/A -code</td>
<td>12</td>
<td>LTI</td>
<td>1.75 x 10 x 19 in</td>
<td>&lt;10 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TO 2000</td>
<td>12 par</td>
<td>GPS</td>
<td>12</td>
<td>LTI</td>
<td>1.75 x 10 x 19 in</td>
<td>&lt;10 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TO 2000</td>
<td>12 par</td>
<td>L1, C/A -code</td>
<td>12</td>
<td>LTI</td>
<td>1.75 x 10 x 19 in</td>
<td>&lt;10 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5 x 2 x 0.18 in</td>
<td>&lt;20 in</td>
<td>&lt;100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The table above represents a selection of GPS receivers surveyed in 2009, including details such as manufacturer, model, channels/tracking mode, signal tracking, max no. of satellites, user environment and application, size, V, position accuracy, time (min), and position fix update (with talk/sec). Each entry provides specific technical specifications and dimensions for various GPS receiver models.
<table>
<thead>
<tr>
<th>Manufacturer/Model</th>
<th>Character/Tracking Modes</th>
<th>Max No. Satellites</th>
<th>User Equipment &amp; Application</th>
<th>Size/Weight</th>
<th>Position Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trimble</td>
<td>LEA-5H GPS Module</td>
<td>1L, C/A, L1, P</td>
<td>50 par</td>
<td>12.2 x 16.0 x 2.4 mm</td>
<td>1.6 g</td>
</tr>
<tr>
<td></td>
<td>NEO-5M GPS Module</td>
<td>1L, C/A, L1, P</td>
<td>50 par</td>
<td>12.2 x 16.0 x 2.4 mm</td>
<td>1.6 g</td>
</tr>
<tr>
<td></td>
<td>NEO-5G GPS Module</td>
<td>1L, C/A, L1, P</td>
<td>50 par</td>
<td>12.2 x 16.0 x 2.4 mm</td>
<td>1.6 g</td>
</tr>
<tr>
<td></td>
<td>LEA-5G GPS Module</td>
<td>1L, C/A, L1, P</td>
<td>50 par</td>
<td>12.2 x 16.0 x 2.4 mm</td>
<td>1.6 g</td>
</tr>
<tr>
<td></td>
<td>LEA-5F GPS Module</td>
<td>1L, C/A, L1, P</td>
<td>50 par</td>
<td>12.2 x 16.0 x 2.4 mm</td>
<td>1.6 g</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| | Trimble | R7 GPS system receiver | 48 + 4 SBAS | 19.0 (Ø) x 11.5 cm | 1.35 kg | 1–5 m / 0.25 m + 1 ppm /
| | Trimble | R6 GPS system receiver | 48 + 4 SBAS | 19.0 (Ø) x 11.5 cm | 1.35 kg | 1–5 m / 0.25 m + 1 ppm /
| | Lassen | – SQ II | 8 par | 1.25 x 0.48 x 3.25 in | 0.7 oz | 1–5 m / 1–5 m |
| | | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | |

---

**Note:** The above table provides a summary of GPS receiver specifications. For detailed information, please refer to the original document or the manufacturer's official website.
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Channels/Tracking Mode</th>
<th>Signal(s)</th>
<th>Max No. Satellites Tracked</th>
<th>User Environment and Application</th>
<th>Size (W x H x D)</th>
<th>WA</th>
<th>Position Accuracy</th>
<th>Time (ns)</th>
<th>Pos. Fix Update with Rate (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA-9M</td>
<td>GPS Mod</td>
<td>50 par</td>
<td>L1, C/A code, L1 Galileo, WAAS / EGNOS / MSAS</td>
<td>16 (GPS, GALILEO)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.4 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-9Q</td>
<td>GPS Mod</td>
<td>50 par</td>
<td>L1, C/A code, L1 Galileo, WAAS / EGNOS / MSAS</td>
<td>16 (GPS, GALILEO)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.4 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-9S</td>
<td>GPS Mod</td>
<td>50 par</td>
<td>L1, C/A code, L1 Galileo, WAAS / EGNOS / MSAS</td>
<td>16 (GPS, GALILEO)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.4 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-ST</td>
<td>GPS Mod</td>
<td>50 par</td>
<td>L1, C/A code, L1 Galileo, WAAS / EGNOS / MSAS</td>
<td>16 (GPS, GALILEO)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-5H</td>
<td>GPS Mod</td>
<td>50 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-4A</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-4A</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-4H</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-4H</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-4P</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-4P</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-4S</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-4S</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-4T</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>TIM-4T</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3.0 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>NEO-4S</td>
<td>GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>12 x 16 x 2.8 mm</td>
<td>1.6 g</td>
<td>&lt;2.5 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>LEA-AR-Dual Reckoning GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>DUN/PV2</td>
<td>17 x 22 x 4.3 mm</td>
<td>2.1 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>TIM-AR-Dual Reckoning GPS Mod</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>DUN/PV2</td>
<td>25 x 25 x 4.3 mm</td>
<td>3 g</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ANTARIS-G 4 Chipset</td>
<td>16 par</td>
<td>L1, C/A code, DGPS, WAAS / EGNOS</td>
<td>16 (GPS or SBAS)</td>
<td>COHLMN/PV2</td>
<td>RF 4 x 4 mm, BB 3 x 3 x 9 mm</td>
<td>UNL 1.6 x 2 mm</td>
<td>n/a</td>
<td>&lt;2 m / &lt;2 m / na / na / E(PE)</td>
<td>50 (RMS)</td>
<td>4</td>
</tr>
<tr>
<td>SubIX</td>
<td>16 par</td>
<td>GPS, L1, C/A code, 16 GPS</td>
<td>16</td>
<td>GIS Mapping</td>
<td>12 x 6.5 x 4 cm</td>
<td>0.67 lb</td>
<td>WAAS 2 m / CEP, Kinematic PP, submeter, submeter, ILS / SPS / ILS / SPS</td>
<td>1 Hz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES**

2 User environment and applications:  
A = aviation  
C = commercial  
D = defense  
G = survey/GIS  
H = handheld  
L = land  
M = marine  
N = navigation  
O = other  
P = other position reporting  
R = real-time DGPS ref.  
S = space  
T = timing  
V = vehicle/vessel tracking  
1 = end-user product  
2 = board/chipset/module for OEM apps

3 Where three values appear, they refer to autonomous (code), real-time differential (code), and post-processed differential; where four values appear, they refer to autonomous (code), real-time differential (code), real-time kinematic, and post-processed differential.

4 Cold start: ephemeral, almanac, and initial position and time not known.

5 For a warm start, the receiver has a recent almanac, current time, and initial position, but no current ephemerals.

6 Reacquisition time is based on the loss of signal for at least one minute.

7 E = provision for an external antenna  
R = antenna is removable

**ABBREVIATIONS**

apps: applications  
ARINC: Aeronautical Radio, Inc. standard  
asynch: asynchronous  
bps: bits per second  
CP: carrier phase  
CEP: circular error probable  
diff: differential  
ext: external / int. = internal  
m, min: minutes  
a or NA: not applicable  
r: no response  
opt.: optional  
paz: parallel  
prog.: programmable  
ppm: parts per million  
RMS: root mean square  
s: seconds  
SBAS: Satellite-Based Augmentation System  
typ.: typical  
VRS: Virtual reference station  
WP: waterproof  
WR: water resistant
Appendix F

Publications

This section of the appendix includes the two publications obtained due to the research done during this project. The first one is an article presented at URSI 2009 Cantabria Congress http://www.ursi2009.org/. The second one is the poster showed in a demonstration in the NEWCOM++/ACoRN Joint Workshop congress http://www.newcom-acorn.org/naw09.
Correlador Doble Delta para GPS en tiempo real

M. Solé(1), J. Arribas(1), C. Fernández-Prades(1), D. Bernal(2), J. A. Fernández-Rubio(2)
{msole, javier.arribas, carles.fernandez}@cttc.es, {bernal, juan}@gps.tsc.upc.edu
(1) Centre Tecnològic de Telecomunicacions de Catalunya (CTTC) - Castelldefels (Spain).
(2) Universitat Politècnica de Catalunya (UPC), Dep. de Teoría de Señal y Comunicaciones - Barcelona (Spain).

Abstract—This paper describes the implementation and shows the performance improvement of a Double Delta Correlator (DD) respect a typical Delay Locked Loop (DLL) for a real-time software-defined Global Positioning System (GPS) receiver on a multipath environment. A theoretical analysis of wide and double delta correlator has been done, and their performance compared over a simulated multipath scenario at radiofrequency. The double delta correlator implementation exploits the Single Input Multiple Data (SIMD) instructions to achieve real-time performance. The GPS software receiver uses open source libraries including GNURadio and GPSTk, and is capable of running in real-time on a standard PC.

I. INTRODUCCIÓN

Uno de los principales objetivos en los sistemas de posicionamiento global basados en satélites (GNSS) es mejorar la precisión en el cálculo de la posición del receptor.

Para reducir el error en la posición es primordial estudiar una a una las fuentes del mismo. Errores provocados por la propagación atmosférica pueden ser corregidos de manera muy eficiente con técnicas basadas en correcciones diferenciales (DGPS) utilizando sistemas de referencia externos al receptor, no así los errores provenientes de reflexiones de la señal en obstáculos cercanos, ya que estos dependen de la posición del mismo y por lo tanto son imposibles de corregir mediante técnicas diferenciales.

Asimismo, los efectos de la propagación multicamino son la fuente más importante de error y existen numerosas técnicas para mitigar su efecto. Una de las técnicas para reducir el error debido al multicamino es el uso de distintos métodos para la adquisición y el seguimiento de la señal GPS.

En este documento se propone una solución al error producido por el multicamino mediante un correlador Doble Delta en tiempo real definido por software. Sus prestaciones han sido comprobadas en un escenario simulado un canal con propagación multicamino utilizando señal GPS real.

II. CARACTERÍSTICAS DE LA SEÑAL GPS C/A

El sistema GPS transmite la señal de posicionamiento civil denominada C/A en la frecuencia portadora L1. La portadora L1 está centrada en 1575.42 MHz y es modulada en BPSK por una secuencia pseudaleatoria (PRN) de 1023 valores (chips) única para cada satélite, con una frecuencia de chip de 1.023 Mcps. Sobre esta señal se superpone una nueva modulación BPSK con el mensaje de navegación con una frecuencia de bit de 50 bps. El resultado es una señal de espectro ensanchado, emitida desde los satélites con un filtro con 20 MHz de ancho de banda.

Cada satélite transmite con una secuencia PRN distinta y ortogonal al resto, haciendo uso de un generador tipo ‘gold code’ común a todos los satélites pero inicializado con parámetros diferentes según el identificador de satélite [1].

El receptor reproduce la secuencia PRN local para cada satélite y realiza la correlación entre la señal de entrada y dicha secuencia. Debido a las características de la señal GPS la autocorrelación de la señal sólo obtiene un máximo cuando las dos secuencias se encuentran alineadas. En este proceso se obtiene la pseudodistancia parcial del receptor al satélite.

Si analizamos la duración de cada chip y la convertimos a distancia, tenemos que equivale a 300 metros, la cual es la resolución de la propia señal C/A. Los receptores GPS han superado esta limitación haciendo uso de diversas técnicas para poder sincronizar con mayor exactitud las dos secuencias cuando la señal recibida por el receptor contiene réplicas procedentes de reflexiones, las cuales como se muestra en la próxima sección, distorsionan las características de autocorrelación.

III. TIPOS DE CORRELADORES

El objetivo del correlador en la estructura del GPS es calcular el retardo de la señal. Para ello se generan, además de la secuencia local (Prompt), réplicas avanzadas (Early) y retardadas (Late) del código PRN separadas un determinado tiempo (δ) que se correlan con la secuencia PRN recibida. Según la disposición y el número de réplicas existen distintos tipos de correladores.

A. Correlador Ancho (Wide Correlator)

El correlador ancho, más conocido como Wide, dispone de dos réplicas de la señal PRN, es decir, un Early y un Late separados por un tiempo δ = Tc, siendo Tc el periodo de chip. El error de la señal, también llamado discriminador, del correlador ancho es:

\[ D_W = \sqrt{(I_E^2 + Q_E^2) - \sqrt{(I_L^2 + Q_L^2)}} \]

De modo que cuando el discriminador es positivo el código PRN está avanzado una fracción determinada de chip, y si es negativo se encuentra retardado.

B. Correlador Estrecho (Narrow Correlator)

El correlador estrecho, también conocido como Narrow es una variación del correlador ancho. Aunque utiliza también una réplica retardada y otra adelantada del código PRN, la
separación entre ellas es menor, siendo de $\delta = 0.1T_c$. La ecuación del discriminador es la misma que en el caso anterior.

$$D_N = D_W$$

C. Correlador Doble Delta

Mientras que los dos correladores anteriores sólo utilizaban una réplica adelantada y otra retardada de la secuencia PRN, el correlador Doble Delta (DD) utiliza dos réplicas para cada caso[4]. La dos réplicas añadidas se intercalan entre las dos ya previstas anteriormente, considerando $\delta = 0.5T_c$. La Fig.1 muestra gráficamente la implementación del correlador DD, dónde $\delta$ es la separación entre las réplicas del código PRN generadas localmente y E1, E2, L1 y L2 sus respectivas potencias. El discriminador utilizado para este caso es:

$$D_{DD} = 2\sqrt{(I_{E1}^2 + Q_{E1}^2)} - \sqrt{(I_{L1}^2 + Q_{L1}^2)} - \sqrt{(I_{E2}^2 + Q_{E2}^2)} - \sqrt{(I_{L2}^2 + Q_{L2}^2)}$$

(1)

D. Efecto multicamino

Como ya hemos comentado anteriormente una de las fuentes principales de error para los receptores GPS es el producido por el efecto del multicamino. Este error se produce porque el multicamino provoca una deformación en la función de correlación, de manera que esta se vuelve asimétrica tal y como muestra la Fig.2, considerando el ancho de banda de señal entero. Así, cuando el ancho de banda es limitado, el punto de máxima amplitud en la correlación se suaviza y se desplaza según sea el retardo provocado por el multicamino. Los diferentes correladores, tienen como finalidad encontrar el punto de máxima amplitud de la correlación.

La Fig.3 muestra el error en el cálculo de la pseudodistancia obtenido de manera teórica para el correlador ancho y correlador DD en presencia de multicamino y ausencia de ruido. Puede verse que la ganancia en términos de error en la pseudodistancia mejora en unos 40 metros con el correlador DD.

IV. IMPLEMENTACIÓN

A. Receptor GPS definido por software

El receptor gps-sdr utilizado [7] parte de la base del desarrollo open source, realizándose una mejora en varios puntos: por un lado se ha pasado de usar el correlador ancho al correlador DD y por otro se han utilizado las librerías GPSTk [8] para obtener el cálculo de la posición. La comunicación entre el receptor gps-sdr y las librerías GPSTk así como los ficheros de salida tienen el formato estándar Receiver Independent Exchange Format (RINEX).

B. Estructura del correlador DD

El correlador DD sigue el siguiente proceso:

1) Generar los vectores en fase y en cuadratura que contienen las muestras de la señal destinada a eliminar la portadora (señal FI).
2) Generar la réplica local del código C/A para el satélite deseado.
3) Multiplicar la señal de entrada por la réplica local del código C/A generada anteriormente.
4) Multiplicar el resultado de la operación anterior por los vectores en fase y cuadratura, acumular y de esta forma obtener la correlación en fase y cuadratura.

Las operaciones de multiplicación de vectores son adecuadas para realizarse mediante instrucciones avanzadas del procesador como se verá en la siguiente sección, permitiendo así un procesado en tiempo real.

C. Instrucciones SIMD

Las operaciones de una sola instrucción para múltiples datos (SIMD) es una técnica utilizada para conseguir un paralelismo en la entrada de datos, como en un procesador por vectores [2]. Así, la unidad básica del SIMD es el vector.

Una unidad de procesamiento central regular (CPU) es capaz de tratar a la vez, como mucho, múltiples escalares pero actuando con una operación diferente para cada instrucción. Un procesador con la arquitectura SIMD explota la propiedad del paralelismo de datos, que permite actuar a la vez con un gran nombre de datos del mismo tipo que necesitan el mismo tratamiento o operación.

Para la implementación del correlador DD se ha utilizado la extensión SSE y SSE2 de SIMD. Estas extensiones proporcionan un conjunto de instrucciones y una serie de registros (XMM0−XMM7) de 128 bits para tratar los datos de entrada que permite incrementar la velocidad de ejecución respecto los procesadores regulares. Así gracias a la arquitectura SIMD podemos implementar el correlador en tiempo real.

V. ESCENARIO DE PRUEBAS

El escenario de pruebas utilizado para verificar la mejora teórica del correlador DD consiste en generar sintéticamente la señal mediante el generador en tiempo real Agilent E4438C ESG Vector Signal Generator con la ampliación GPS Personality, que convenientemente permite recrear un escenario completo GPS incluyendo las señales y la telemetría para 8 satélites [5].

La señal de RF generada a frecuencia L1 es inyectada al emulador de canal Propsim C8 [3] para generar un modelo de canal parametrizado con propagación multicanal.

La salida del emulador de canal conectada a la plataforma de adquisición de señal de radio USRP la cual realiza la conversión de RF a banda base y digitaliza la señal para ser enviada en tiempo real al ordenador mediante bus USB. Las muestras digitales son procesadas con el receptor definido por software y las medidas de pseudodistancia son registradas en archivos para posteriormente ser comparadas [6][7].

VI. SIMULACIÓN Y CONCLUSIONES

Las simulaciones realizadas tienen como finalidad comparar el error en la pseudodistancia obtenida producido por el efecto multicanal para cada satélite con dos correladores distintos. En este caso se comparan el correlador ancho y el correlador DD.

Así pues, basándose en el escenario de pruebas expuesto anteriormente y a partir de la señal GPS generada con el generador de funciones [5] los pasos seguidos para realizar la simulación han sido:

1) Generación de la respuesta impulsional del canal con multicanal. La figura 7 muestra la representación gráfica de esta respuesta, y su definición matemática es:

\[ h_m(t) = \frac{A}{10} \delta(t) + \frac{A}{\sqrt{2}} \delta(t - \tau) \]

con: \( \tau = \frac{i}{10} \frac{1}{1023000} \text{seg} \) donde \( i = 0, 1, 2, 3...14 \). donde se ha representado la recepción de la señal directa, más la señal indirecta con una potencia 3 dB inferior a la primera. Además el retardo \( \tau \) de la señal indirecta varia según se desee un retardo de chip mayor o menor debido al multicanal.
2) Obtención del error instantáneo para cada satélite cometido para el cálculo de la pseudodistancia en comparación con la pseudodistancia real.

\[ \rho_{REAL,i}(t) = ||\mathbf{r}_s - \mathbf{r}_o|| - b_i(t)c + \text{ruido}(t) \]

\[ \text{error}_i(t) = \rho_{GPS,i}(t) - \rho_{REAL,i}(t) \]

donde:
- \( \mathbf{r}_s \) y \( \mathbf{r}_o \) son los vectores de posición del satélite \( i \) y del receptor/observador en coordenadas ECEF (Earth Centered Earth Fixed)
- \( b_i \) es la desviación del reloj interno del satélite \( i \).
- \( c \) la velocidad de la luz.
- \( \rho_{GPS,i}(t) \) es la pseudodistancia calculada por el receptor gps-sdr mediante el método de mínimos cuadrados (LS) [7].

3) Obtención del error medio en términos de pseudodistancia para un retardo multicamino \( \tau \) preestablecido. Para ello se han tomado muestras de la señal durante aproximadamente unos 10 minutos (600 segundos), dado que el receptor GPS es capaz de calcular la pseudodistancia 10 veces por segundo esto hace un total de unas 6000 muestras. Así el error medio para el satélite \( i \) se calcula como:

\[ \text{error}_i(t) = \sum_{i=1}^{\#muestras} \text{error}_i(t) \]

4) Repetición de los pasos 2 y 3 variando el retardo \( \tau \) del multicamino en la respuesta impulsional del paso 1.

5) Representación gráfica de la evolución del error medio frente al retardo introducido por el multicamino en la señal recibida de forma indirecta.

La Fig. 8 muestra el error en la pseudodistancia para los dos correladores donde se ha utilizado un paso para el retardo del multicamino entre simulación y simulación de 0.1 chips. Este espacioado ha sido escogido debido al consumo de tiempo de la simulación que supone la obtención del error medio para cada retardo de chip. No obstante se está desarrollando un método de monitorización automatizada. A pesar de todo, en la Fig.8 se puede ver claramente como el error conseguido con el correlador DD propuesto es siempre menor que con el correlador ancho original. Existe un error sistemático causado por el retardo de propagación atmosférico, el cual no se ha corregido en el cálculo de la pseudodistancia estimada \( \rho_{GPS} \). Se observa como la mejora del cálculo de la pseudodistancia llega hasta los 10 metros con el correlador DD propuesto. Si comparamos la Fig. 3 con la Fig. 8, se observa una diferencia de rendimiento, pasando de una mejora de 40 metros en la simulación a una mejora real de 10 metros.

Este rendimiento se debe a que el receptor gps-sdr no utiliza una frecuencia de muestreo adecuada para el correlador DD. De esta forma, no es posible aprovechar toda la precisión que el discriminador DD ofrece, ya que la corrección del discriminador esta cuantizada.

AGRADECIMIENTOS

Este trabajo ha sido apoyado por el Ministerio de Educación y Ciencia a través de los proyectos TEC2008-02685/TEC (NARRA), CENIT2007-2002 (TIMI) y los fondos FEDER de la Comisión Europea a través del marco FP7 en la red de excelencia en comunicaciones inalámbricas NEWCOM++ (contrato N° 216715) así como la acción COST IC0803 (RFCSET).

REFERENCES

The CTTC’s software defined radio (SDR) Global Navigation Satellite System (GNSS) demonstrator is an implementation of a complete real-time software based Global Positioning System (GPS) L1 C/A receiver. All the hardware and software blocks are based on open source solutions and standards. CTTC has contributed to implement a real-time software based GPS receiver without modifying the hardware. The developed platform is modular, giving estimates of the code offset and Doppler shift of the received signal.

Acquisition: Performs a coarse synchronization process, giving estimates of the code offset and Doppler shift of the received signal.

Tracking: Shares with the rough approximations of the 200 best correlation parameters and consists of a software defined single delta Delay Locked Loop (DLL) for the fine code delay search and a Phase Locked Loop (PLL) for the fine carrier Doppler search. Both components are interchanging information. CTTC has implemented a double delta correlator in order to mitigate the multipath effect.

The atmospheric-dependent sources of accuracy degradation can be greatly mitigated by differential systems external to the receiver’s operation but the multipath effect is location-dependent and remains as the most important cause of accuracy degradation in time delay estimation, and consequently in position estimation.

The real-time double delta correlation on a standard PC has been implemented using Single Instruction, Multiple Data (SIMD) SSE2 assembly instructions. The basic unit of SIMD is the vector, which is suitable to perform the vector multiplications for the correlation algorithm exploiting data parallelism properties. The SIMD correlator algorithm is shown here:

1. Generate the C/A code replica, PRN.
2. Generate inphase (COS) and quadrature (SIN) carrier
3. Multiply IFxPRN vector (SIMD enabled function).
4. Multiply IFxPRN by COS/SIN, accumulate to generate the I/Q correlation (SIMD enabled function).
5. Real-time correlator implementation using SIMD Instructions.
6. The testing equipment consists of a GPS real-time signal generator Agilent GPS Personality for the E4438C ESG function generator.
7. The GPSTk solvers can improve the accuracy using the differential GPS (D-GPS) correction data. A specific RINEX file obtained from a reference station is used in the process. D-GPS is an enhancement to GPS that uses a network of fixed, ground-based reference stations to broadcast the difference between the pseudorange indicated by the satellite signal and the known fixed position.

The testing platform consists of a GPS real-time signal generator Agilent GPS Personality for the E4438C ESG function generator capable of generating a variety of pre-defined GPS scenarios with up to 8 satellites in view. The RF signal is fed into the USRP RF-downconversion board at GPS L1 frequency, providing accurate control to test the receiver sensitivity, accuracy and cold start-up time in various conditions.

The goal of the GPSTK project is to provide an open source software based navigation community to free researchers to focus on research, not lower level coding. The GPSTK suite consists of a core library, auxiliary libraries, and a set of applications. CTTC-GNSS receiver uses GPSTK classes to obtain the position from different approaches.

References