Ph.D. Thesis

Doctoral Program in Aerospace Science & Technology

Precise GPS-based position, velocity and acceleration determination: Algorithms and tools

Dagoberto José Salazar Hernández

Advisor:
Dr. Jaume Sanz Subirana

Research group of Astronomy and Geomatics (gAGE)
Depts. of Applied Mathematics IV and Applied Physics
Universitat Politècnica de Catalunya (UPC), Spain

March 8, 2010

Precise GPS-based position, velocity and acceleration determination: Algorithms and tools

Dagoberto Salazar, 2010.

Doctoral Program in Aerospace Science & Technology
Technical University of Catalonia

PhD. Thesis made with the support of:
Research group of Astronomy and Geomatics (gAGE/UPC)
Polytechnic School of Castelldefels (EPSC/UPC)

Dedicated to the memory of my father: A man who was patient enough to teach his 4-year-old son the difference between voltage and current...

...I believe I finally got it, Dad.

Acknowledgements

I want to start thanking my wife, Rocío. I've received from her much more support, comprehension and love than any husband deserves.

I also thank my children, Vanessa and Ricardo. They have enriched my life in so many ways.

My "compadre" Ignasi deserves a special mention here: He has been a "guardian angel" for us, and this thesis could well not have existed without his initial support. The same can be said from Jordi, whom I resort everytime I have a big problem, or when I just want to have a good laugh.

Mamá, Dangel, Darvin, Lorena, Érika, Daniela, Oscar... thank you very much for always been there.

I owe a huge debt to my mentors, gAGE senior researchers Jaume, Miguel and Manuel. I've been so lucky of been able to drink from their deep wells of wisdom and knowledge everyday during all these years.

Thanks to Angela, Alberto, Pere, Raúl, Pedro and Adriá, my gAGE comrades. You've made the hard work lighter, and you've been there to help me with my doubts and ideas. Pere and Raúl: I owe you a big one.

I also want to thank the GPSTk team: You've done a terrific job that I've taken advantage of. Special thanks go to Ben, who has been so supportive from the very beginning.

The EPSC school has provided a great support during these years. Thanks to Directors Miguel Valero and Jordi Berenguer for being so flexible. Also a big thank goes to the "Aeronautic bunch": Adeline, Jorge, Xavi, Luis, Pep, Santi and Consol, it is so fun to work with you guys.

My friends José, Oscar and Pinar: Thanks because although you are far away, you are also here with me.

I want to close this section thanking the GPSTk users: Everytime I receive your support requests, I have to learn new things and think in new ways about what I'm doing, making me feel that what I'm doing is worthwhile.

Contents

List of Figures

List of Tables

Acronyms list

ANTEX	Antenna Exchange Format
ANSI	American National Standards Institute
API	Application Programming Interface
ARL	Advanced Research Laboratory
AT&T	American Telephone and Telegraph
BRUS	Basic Research Utilities for SBAS
CODE	Center for Orbit Determination in Europe
DGPS	Differential GPS
DOP	Dilution Of Precision
ECEF	Earth-Centered, Earth-Fixed
EGNOS	European Geostationay Navigation Overlay System
EKF	Extended Kalman Filter
EPSC	Escuela Politécnica Superior de Castelldefels
ERP	Earth Rotation Parameters
ESA	European Space Agency
ESTB	EGNOS System Test Bed
EVA	Extended Velocity and Acceleration determination
EWL	Extra Wide Lane
FFMC	Full Function Miniature Computers
FIR	Finite Impulse Response

gAGE	Group of Astronomy and Geomatics
GDS	GNSS Data Structures
GLONASS	GLObal NAvigation Satellite System
GNSS	Global Navigation Satellite System
GNU	GNU's Not Unix
GPS	Global Positioning System
GPST	GPS Time
GPSTk	GPS Toolkit
IBM	International Business Machines Corporation
ICC	Institut Cartografic de Catalunya
IEC	International Electrotechnical Commission
IERS	International Earth Rotation and Reference Systems Service
IGS	International GNSS Service
IIR	Infinite Impulse Response
INS	Inertial Navigation System
IONEX	Ionosphere Map Exchange
ISO	International Organization for Standardization
LGPL	GNU Lesser General Public License
LMS	Least Mean Squares
MOPS	Minimum Operational Performance Standards
NAVSTAR	Navigation System with Time And Ranging
NEU	North-East-Up
NIMA	National Imagery and Mapping Agency
OOP	Object-Oriented Programming
PC	Personal Computer
PDA	Personal Digital Assistant
PLL	Phase Lock Loop

POP	Precise Orbits Positioning
PPP	Precise Point Positioning
PRN	Pseudo-Random Noise
PVT	Position, Velocity, Time
RAIM	Receiver Autonomous Integrity Monitoring
RINEX	Receiver INdependent EXchange format
RMS	Root Mean Square
RS-MMC	Reduced Size Multi Media Card
RTK	Real Time Kinematics
RX	Receiver
SBAS	Satellite-Based Augmentation System
SINEX	Solution Independent Exchange
SP3	Standard Product #3
SPS	Standard Positioning Service
STL	Standard Template Library
SV	Space Vehicle
TECU	Total Electron Content Unit
TGD	Total Group Delay
UEN	Up-East-North
UPC	Universitat Politecnica de Catalunya
UT	Universal Time
VFR	Visual Flight Rules
VRS	Virtual Reference Station
WMS	Weighted-Least Mean Squares

Abstract

This work is a Ph.D. Thesis for the Doctoral Program in Aerospace Science & Technology from the Universitat Politecnica de Catalunya (UPC), focusing on the development of algorithms and tools for precise Global Positioning System (GPS)-based position, velocity and acceleration determination very far from reference stations in post-process mode.

One of the goals of this thesis was to develop a set of state-of-the-art Global Navigation Satellite System (GNSS) data processing tools, and make them available for the research community. Therefore, the software development effort was done within the frame of a preexistent open source project called the GPS Toolkit (GPSTk). Validation of the GPSTk pseudorange-based processing capabilities was carried out comparing the results with a trusted GPS data processing tool, confirming the viability of the GPSTk as a source code base for developing reliable GNSS data processing software.

GNSS data management proved to be an important issue when trying to extend GPSTk capabilities to carrier phase-based data processing algorithms. In order to tackle this problem the GNSS Data Structures (GDS) and their associated *processing paradigm* were developed, preserving both the data and corresponding metadata. With this approach the GNSS data processing becomes like an "*assembly line*", where all the processing steps are performed sequentially, providing an easy and straightforward way to write clean, simple to read and use software that speeds up development and reduces errors.

The extension of GPSTk capabilities to carrier phase-based data processing algorithms was carried out with the help of the GDS, adding important accessory classes necessary for this kind of data processing and providing reference implementations. The performance comparison of these relatively simple GDS-based source code examples with other state-of-the art Precise Point Positioning (PPP) suites demonstrated that their results are among the best, confirming the validity of using the GPSTk combined with the GDS to get easy to write and maintain, yet powerful, GNSS data processing software. Furthermore, given that the GDS design is based on data abstraction, it allows a very flexible handling of concepts beyond mere data encapsulation, including programmable general solvers, among others.

The problem of post-process precise positioning of GPS receivers hundreds of kilometers away from nearest reference station at arbitrary data rates was dealt with, overcoming an important limitation of classical post-processing strategies like PPP. The advantages of GDS data abstraction regarding solvers, and in particular the possibility to set up a "general solver" object, were used to implement a kinematic PPP-like processing based on a network of stations. This procedure was named *Precise Orbits Positioning (POP)* because it is independent of precise clock information and it only needs precise orbits to work. The results from this approach were very similar (as expected) to the standard kinematic PPP processing strategy, but yielding a higher positioning rate. Also, the network-based processing of POP seems to provide additional robustness to the results, even for receivers outside the network area.

The last part of this thesis focused on implementing, improving and testing algorithms for the precise determination of velocity and acceleration hundreds of kilometers away from nearest reference station. Special emphasis was done on the Kennedy method because of its good performance. A reference implementation of Kennedy method was developed, and several experiments were carried out. Experiments done with very short baselines showed a flaw in the way satellite velocities were computed, introducing biases in the velocity solution. A relatively simple modification was proposed, and it reduced the RMS of 5-min average velocity 3D errors by a factor of over 35.

Then, borrowing ideas from Kennedy method and the POP method, a new velocity and acceleration determination procedure was developed and implemented that greatly extends the effective range. This method was named "Extended Velocity and Acceleration determination (EVA)".

An experiment using a light aircraft flying over the Pyrenees showed that both the modified-Kennedy and EVA methods were able to cope with the dynamics of this type of flight. EVA performance was a little behind RTK-derived velocity estimations, but modified-Kennedy and EVA outperformed RTK in acceleration estimations.

Finally, both modified-Kennedy and EVA method were applied to a very wide network on equatorial South America, near local noon, with baselines over 1770 km. Under this scenario, the EVA method showed a clear advantage in both averages and standard deviations for all components of velocity and acceleration. This confirms that EVA is an effective method to precisely compute velocities and accelerations when the distance to the nearest reference station is over one thousand kilometers.

Resumen

Este trabajo es una tesis doctoral para el Programa de Doctorado en Ciencia y Tecnología Aeroespacial de la Universidad Politècnica de Catalunya (UPC). Esta tesis llevó a cabo el estudio, desarrollo e implementación de algoritmos para la navegación con sistemas globales de navegación por satélite (GNSS), enfocándose en el desarrollo de algoritmos y herramientas para la determinación precisa de la posición, la velocidad y la aceleración usando el sistema GPS, en modo de post-procesado y muy lejos de estaciones de referencia.

Uno de los objetivos de esta tesis era el desarrollar herramientas avanzadas de procesado de datos GNSS, y hacerlas disponibles para la comunidad investigadora. Por ello, el esfuerzo de desarrollo del software se hizo dentro del marco de un proyecto preexistente de software libre llamado la GPS Toolkit (GPSTk). El Capítulo 1 presenta características generales de ese proyecto tales como su estructura, funcionalidades básicas y filosofía de desarrollo, mostrando además el gran nivel de portabilidad que presenta la GPSTk.

Una de las primeras tareas realizadas durante el curso de esta tesis fue la validación de las capacidades de la GPSTk para el procesado de datos de pseudorango. Los resultados de las comparaciones con una herramienta de procesamiento de datos probada (BRUS) mostraron un acuerdo excelente, tanto en el modelado como en la solución final de la posición, confirmando la viabilidad de la GPSTk como una base de código fuente confiable para el desarrollo de software de procesado de datos GNSS.

La gestión de datos GNSS demostró ser un asunto importante a tratar cuando se intentó extender las capacidades de la GPSTk al procesamiento de datos obtenidos de las fases de las ondas portadoras de la señal GPS. Esta tarea se desarrolló en el Capítulo 2, donde se presentaron las “Estructuras de Datos GNSS” (GDS por sus siglas en inglés). Se explicó allí la motivación para el desarrollo de las GDS, una visión general de su implementación, así como el *paradigma de procesamiento* asociado. En el Capítulo 2 también se incluyeron varias estrategias de procesado de datos basadas en el pseudorango con el fin de mostrar con claridad cómo pueden ser usadas las GDS.

La principal contribución de las GDS consiste en el hecho de que ellas preservan tanto los datos como las relaciones existentes entre ellos, indexando internamente toda la información relevante. Combinadas con su paradigma de procesamiento, el procesado de datos GNSS se convierte entonces en una especie de "*línea de ensamblado*", donde las fases de procesado son realizadas de manera secuencial en lugares específicos. Este enfoque proporciona una manera fácil y directa de encapsular y procesar los datos, permitiendo escribir software que es "limpio", fácil de leer y simple de usar, acelerando el proceso de desarrollo y reduciendo los errores de dicho proceso.

En el Capítulo 3 se trató la extensión de las capacidades de la GPSTk a los algoritmos de procesado de datos basados en la fase. Se presentaron allí algunas aplicaciones de las GDS a este tipo de procesamiento, así como importantes clases accesorias que facilitan el trabajo. También se proporcionaron implementaciones de referencia para su uso por parte de la comunidad GNSS, encontrándose éstas en el directorio `examples` del proyecto GPSTk.

Cuando se compara el rendimiento en el procesado de datos "Precise Point Positioning (PPP)" de estos ejemplos relativamente simples basados en las GDS con otras aplicaciones de reputación ya establecida, se encontró que sus resultados destacan entre los mejores. Esto confirma la validez de utilizar la GPSTk combinada con las GDS para obtener software de procesado de datos GNSS que es a la vez potente y fácil de escribir y mantener. Es más, dado que el diseño de las GDS está basado en la abstracción de datos, éstas permiten un manejo muy flexible de conceptos que están más allá de la simple encapsulación de datos, incluyendo, por ejemplo, resolvedores de ecuaciones genéricos y programables.

El Capítulo 4 se enfocó en el problema de obtener la posición precisa, en post-proceso, de un receptor GPS que se encuentra a cientos de kilómetros de la estación de referencia más cercana. Un requisito adicional era el uso de tasas de datos arbitrarias, resolviendo una importante limitación del método PPP clásico. Las ventajas aportadas por la abstracción de datos de las GDS a los resolvedores de ecuaciones, y en particular la posibilidad de utilizar un "resolvedor genérico", fueron una pieza clave en la implementación de un procesado semejante a un PPP cinemático basado en una red de estaciones de referencia. Esta estrategia fue bautizada como "*Precise Orbits Positioning (POP)*" porque sólo necesita órbitas precisas para trabajar y es independiente de la información de los relojes de los satélites GPS.

La estrategia POP involucra múltiples estaciones separadas cientos de kilómetros, y presenta un gran número de incógnitas de diferentes tipos. Algunas incógnitas están *indexadas por receptor* (es decir, son específicas de un receptor dado, como las coordenadas o el retraso troposférico), otras incógnitas están *indexadas por satélite* (como el desfase de los relojes atómicos de a bordo), y otras están indexadas tanto por receptor como por satélite (las ambigüedades de fase,

por ejemplo). Por tanto, el número de incógnitas en un instante dado presenta una gran variación, dependiendo ésta de las estaciones de referencia disponibles y del número de satélites visibles. Durante esta tesis se desarrolló la clase de la GPSTk llamada `SolverGeneral` que ayuda a implementar esta clase de sistemas *describiendo* (en vez de escribiendo en el código fuente del software), las ecuaciones, las variables, sus relaciones y los modelos estocásticos asociados a cada una. El programa `example14.cpp` se proporciona como una implementación de referencia de este método de procesado de datos.

Los resultados de este enfoque fueron muy similares (como era de esperar) a los del método PPP cinemático estándar, pero proporcionando soluciones de posición con una tasa mayor. Asimismo, la estrategia POP parece proporcionar una mayor robustez a los resultados, incluso para aquellos receptores que se encuentran fuera del área de la red. La distancia desde el receptor móvil ("*rover*") a la estación de referencia más cercana no parece ser un factor crítico, dado que en las pruebas realizadas los resultados no se degradaron de manera significativa cuando esta distancia se duplicó.

Por otra parte, el tiempo de convergencia con POP disminuye conforme el número de estaciones de la red se incrementa, pero hasta cierto punto. Este asunto representa un problema si se desea aplicar el método POP a vehículos, especialmente si los arcos de datos son cortos.

La última parte de esta tesis se enfocó en la implementación, mejora y prueba de algoritmos para determinar con precisión la velocidad y aceleración de un receptor GPS a cientos de kilómetros de la estación de referencia más cercana. El Capítulo 5 revisó varios métodos para calcular la velocidad y aceleración, haciendo énfasis en el método de las fases de Kennedy debido a su buen rendimiento. Dicho método fue explicado con detalle.

Se desarrolló una implementación de referencia del método Kennedy y se llevaron a cabo varias pruebas. Los experimentos hechos con líneas de base muy cortas mostraron que había una falla en el procedimiento propuesto originalmente por Kennedy para el cálculo de las velocidades de los satélites, introduciendo sesgos en la solución de velocidad. Se propuso entonces una modificación relativamente sencilla, y ésta redujo en un factor mayor que 35 el RMS de los errores 3D en velocidad (promedios a 5 minutos), conduciendo a una versión modificada de dicho método. Adicionalmente, resultados preliminares obtenidos experimentando con los modelos de covarianzas de errores sugieren que versiones más sencillas y rápidas pueden proporcionar resultados equivalentes al del modelo completo propuesto por Kennedy.

Entonces, y tomando ideas del método de Kennedy modificado y del método POP presentado en el Capítulo 4, se desarrolló e implementó un nuevo procedimiento de determinación de velocidad y aceleración que extiende en gran

medida el rango efectivo. Este método fue llamado "Extended Velocity and Acceleration determination (EVA)".

Un experimento usando una aeronave ligera volando sobre los Pirineos mostró que tanto el método de Kennedy modificado como el método EVA son capaces de responder ante la dinámica de este tipo de vuelos. Cuando se compararon los resultados de estos métodos con una zona de velocidad cero los resultados fueron muy similares, mostrando el método de Kennedy modificado una ligera ventaja. El rendimiento del método EVA estuvo un poco por detrás de las estimaciones de velocidad derivadas de posiciones RTK, pero tanto Kennedy modificado como EVA superaron ampliamente a RTK en lo que a estimaciones de aceleración se refiere.

Finalmente, tanto el método de Kennedy modificado como el método EVA fueron aplicados a una red muy amplia en la zona ecuatorial de Sur América, alrededor del mediodía local y con líneas de base mayores a 1770 km. En este escenario el método EVA mostró una clara ventaja tanto en los promedios como en las desviaciones estándar para todas las componentes de la velocidad y la aceleración. Esto confirma que EVA es un método efectivo para calcular con precisión las velocidades y aceleraciones cuando la distancia a la estación de referencia más cercana supera los mil kilómetros.

Contribuciones

El desarrollo de las GNSS Data Structures (GDS) y su paradigma de procesamiento es una de las contribuciones de esta tesis. Las GDS solucionan algunos aspectos de la gestión de datos GNSS preservando tanto los datos como los metadatos, y proporcionando una manera de escribir software que acelera el desarrollo y reduce los errores.

El procedimiento POP se considera otra contribución. Aunque no es una estrategia original (métodos similares han sido reportados previamente en la literatura) su implementación representa una manera nueva de resolver este tipo de problemas. Es particularmente notable el uso de un resolvedor de ecuaciones programable en tiempo de ejecución (`SolverGeneral`) donde las ecuaciones, las variables, sus relaciones y los modelos estocásticos asociados a cada una son *descritos* en vez de *escritos* en el software. Asimismo, este enfoque es lo suficientemente flexible como para ser utilizado en otros tipos de problemas complejos, como se demostró en el Capítulo 5.

El estudio de métodos de determinación de la velocidad y la aceleración basados en la fase representa otra contribución: En particular, la modificación de la manera como el método Kennedy calcula las velocidades de los satélites condujo a mejoras de un orden de magnitud en los sesgos de las estimaciones de velocidad.

Adicionalmente, el desarrollo del nuevo método "Extended Velocity and Acceleration determination (EVA)" soluciona el problema de la determinación precisa en post-proceso de la velocidad y la aceleración a miles de kilómetros de la estación de referencia más cercana. Ésta se considera una contribución original e importante que pudiera tener un impacto en campos tales como la aerogravimetría, donde se aplicaba el método de Kennedy original.

Otras contribuciones relativamente menores fueron la validación inicial del código básico de la GPSTk, la demostración de su proceso de adaptación a una plataforma de cálculo embebida, la extensión de las capacidades de la GPSTk para procesar datos basados en la fase, y en particular el procesado PPP. Las implementaciones de referencia de varias estrategias de procesado de datos deben ser muy útiles para investigadores y estudiantes en el área GNSS.

Como comentario de cierre, el autor de esta tesis quiere enfatizar que este trabajo no solamente proporcionó contribuciones *científicas*, sino que también dio fruto a contribuciones *logísticas* para la comunidad GNSS en su conjunto, esforzándose en proporcionar herramientas que incrementen la productividad de los investigadores GNSS.

Publicaciones

Este trabajo de tesis resultó en una publicación en una revista arbitrada:

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "GNSS data management and processing with the GPSTk". GPS Solutions, DOI: 10.1007/s10291-009-0149-9, 2009.

También estuvieron relacionadas con esta tesis un cierto número de publicaciones en *proceedings* de congresos:

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "Rapid Open Source GPS software development for modern embedded systems: Using the GPSTk with the Gumstix". Proceedings of the 3rd ESA Workshop on Satellite Navigation User Equipment Technologies NAVITEC '2006. Noordwijk. The Netherlands. December 2006.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "The GPS Toolkit: World class open source software tools for the GNSS research

community". Proceedings of the 7th Geomatic Week. Barcelona. Spain. February 2007.

Harris, R.B., Conn, T., Gaussiran, T.L., Kieschnick, C., Little, J.C., Mach, R.G., Munton, D.C., Renfro, B.A., Nelsen, S.L., Tolman, B.W., Vorce, J. and D. Salazar. "The GPSTk: New Features, Applications and Changes". Proceedings of the 20th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2007). Fort Worth, Texas. September 2007.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "Open source Precise Point Positioning with GNSS Data Structures and the GPSTk". Geophysical Research Abstracts, Vol 10, EGU2008-A-03925, 2008.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "High accuracy positioning using carrier-phases with the open source GPSTk software". Proceedings of the 4th ESA Workshop on Satellite Navigation User Equipment Technologies NAVITEC 2008. Noordwijk. The Netherlands. December 2008.

Salazar, D., Sanz-Subirana, J. and M. Hernandez-Pajares. "Phase-based GNSS data processing (PPP) with the GPSTk". Proceedings of the 8th Geomatic Week. Barcelona. Spain. February 2009.

Gaussiran, T.L., Hagen, E., Harris, R.B., Kieschnick, C., Little, J.C., Mach, R.G., Munton, D.C., Nelsen, S.L., Petersen, C.P., Rainwater, D.L., Renfro, B.A., Tolman, B.W., and D. Salazar. "The GPSTk: GLONASS, RINEX Version 3.00 and More". Proceedings of the 22nd International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2009). Savannah, Georgia. September 2009.

Finalmente, un artículo de investigación acerca del método EVA está siendo preparado actualmente, y será enviado a una revista arbitrada en un futuro cercano.

Futuras líneas de investigación

Durante el desarrollo de esta tesis surgieron varias lineas adicionales de investigación. A continuación se presenta una lista con aquéllas que se consideraron más prometedoras.

- El tiempo de convergencia del método POP se acelera conforme el número de estaciones aumenta, pero hasta cierto punto, y lo mismo puede decirse de las mejoras en los valores del error 3D-RMS. Debería investigarse la topología óptima de las redes de referencia para garantizar un nivel dado de rendimiento con el mínimo uso de recursos computacionales.

- El tiempo de convergencia es un problema cuando se aplica el método POP a vehículos en movimiento, especialmente si los arcos de datos son cortos. Un futuro tópico de investigación debería ser el encontrar estrategias para reducir el tiempo de convergencia, con el fin de aumentar la utilidad de esta estrategia de procesado de datos.

- La exactitud y el tiempo de convergencia del método POP se podrían mejorar considerablemente si se le pudieran aplicar estrategias de fijación de ambigüedades. Trabajos recientes sobre fijación de ambigüedades en PPP hechos por [Wang and Gao, 2006] y [Ge et al., 2008], entre otros, proporcionan una base que pudiera ser aplicada también a POP.

- Resultados preliminares obtenidos cuando se hacían pruebas con el método de Kennedy sugieren que se podrían utilizar modelos de covarianzas de errores simples y más rápidos, obteniendo no obstante resultados equivalentes en la determinación de la velocidad y la aceleración. Este aspecto debería ser explorado para proporcionar mejores modelos de covarianzas.

- Trabajos previos tales como [Serrano et al., 2004] han intentado extender el método de Kennedy a las aplicaciones en tiempo real, usando un único receptor, efemérides "*broadcast*" y un filtro diferenciador simple de primer orden. El autor de esta tesis considera que se podrían obtener mejores resultados usando un filtro diferenciador más sofisticado, de tipo Infinite Impulse Response (IIR), e incluyendo en el algoritmo correcciones proporcionadas por sistemas SBAS.

- Un trabajo hecho por [Kubo, 2009] mostró cómo se puede usar información de velocidad para mejorar el rendimiento del proceso de fijación de ambigüedades en RTK. El autor de esta tesis considera que sería interesante intentar fusionar los métodos POP y EVA con las estrategias de fijación de ambigüedades en PPP previamente mencionadas, para así crear un sistema robusto y preciso de posicionado en post-proceso capaz de operar a miles de kilómetros de la estación de referencia más cercana.

Introduction

This work is a Ph.D. Thesis for the Doctoral Program in Aerospace Science & Technology from the UPC, focusing on the development of algorithms and tools for precise GPS-based position, velocity and acceleration determination in post-process.

Also, one of the goals of this thesis is to develop a set of state-of-the-art GNSS data processing tools and make them available for the research community. In order to maximize the usefulness, ease of reuse, modification, maintenance and distribution among researchers of these tools, it was decided that the software development effort would be done within the frame of the open source GPSTk project, being this thesis work strongly related to the development of the GPSTk.

Motivation

The last decade has witnessed an explosive growth of GNSS, term coined to collectively refer to several operational or planned satellite-based navigation systems such as the American Navigation System with Time And Ranging (NAVSTAR) GPS, the Russian GLObal NAvigation Satellite System (GLONASS) and the future European Galileo.

The current pace of adoption of GNSS products in multiple scientific, commercial and daily life applications is nothing but accelerating. Beyond the original goal of positioning for military use, further uses have evolved along time such as geodesy, timing, ionosphere and troposphere research.

This acute and ever growing demand of GNSS-related techniques in multiple areas has spurred a wealth of research. Several different data processing techniques have appeared along time, and these techniques often must meet tight and conflicting demands of space, processing speed, power consumption, robustness and weight.

Additionally, the aforementioned fast pace of adoption requires a short design-to-market cycle. Development, implementation and testing of robust, innovative and advanced GNSS data processing software in such a small time span is therefore a resource-consuming, error-prone daunting task. Hence, in order to ease this development process a solid code base to build upon is a must.

Although there exists a huge amount of GNSS data processing software available, most of it is closed source and/or they approach GNSS problems in an ad hoc fashion. Reusability of the source code is hard, and documentation is often scarce. Therefore, easy availability and reusability of the source code, as well as abundant documentation will be important aspects regarding the algorithms and tools developed in this work.

In this sense, the GPSTk is presented as a way to achieve these goals. The GPSTk project is an open source project initiated and maintained by the Applied Research Laboratories of the University of Texas, aiming to provide a world class GNSS library computing suite to the satellite navigation community.

An important advantage of working upon an open source framework is increased flexibility. For instance, part of the work done regarding this thesis work was to port the GPSTk to the Gumstix computing platform (see Appendix E for details). Such flexibility, going from a full fledged Pentium-DualCore 64-bits workstation platform to a tiny Gumstix board, is a huge advantage in a research environment. Adding to this advantage a set of solid, tested, powerful and freely available algorithms will be an important contribution to the GNSS research community.

As said in the opening lines, an important avenue of work is to research on the development of algorithms and tools for precise GPS-based position, velocity and acceleration determination in post-process. The original *pseudorange-based* positioning techniques, giving precisions in the order of meters, have long been replaced by *phase-based* techniques in the state-of-the-art research lines. Providing tools that help other researchers to apply the most modern techniques is another important goal of this thesis.

It must be noted that carrier phase ambiguity-solving methods have been traditionally used for precise positioning both in real time (for instance, Real Time Kinematics (RTK)) and in post-process techniques. In this regard, aircraft positioning with *phase-based* methods is a very challenging scenario because airplanes usually operate with long baselines from reference stations[1], and their dynamics are hard to predict and model.

This is a problem this Ph.D. thesis will approach to, providing a first set of

[1]While typical carrier phase ambiguity-solving methods only work a few tens of kilometers from nearest reference station.

tools to estimate, in post-process mode, the position, velocity and acceleration of GPS receivers located very far from reference stations.

Research objectives

The general objective of this thesis will be to study, develop and implement different algorithms for GNSS navigation, focusing on precise[2] position, velocity and acceleration determination in post-process.

Within the framework of this general goal, several specific objectives are also set:

- To extend GPSTk capabilities implementing carrier phase-based data processing algorithms. The classes providing such algorithms must be developed taking into account issues of documentation, maintainability, extensibility and ease of use, therefore maximizing their usefulness for the GNSS research community.
- To develop and implement GNSS data management strategies allowing the simplification of source code, in order to enhance the productivity of GNSS researchers and enable them to focus on more complex problems.
- To develop, implement and test algorithms allowing post-process precise positioning of GPS receivers hundreds of kilometers away from nearest reference station, improving positioning rate regarding classical post-processing strategies like Precise Point Positioning (PPP).
- To implement, improve and test algorithms allowing precise determination of velocity and acceleration, hundreds of kilometers away from nearest reference station, providing valuable tools for other areas such as aerogravimetry.

As it can be seen, this thesis work not only pursues the *scientific* contributions but also includes *logistic* contributions for the GNSS research community at large.

Methodology

The methodology to be used consists of the development of software applications and classes (mainly written in C++) in order to implement and assess different

[2]In this context, the term *precise* is used for positioning errors below the decimeter level.

navigation algorithms and methods.

The performance of these methods will then be validated comparing them with other GNSS data processing software suites. Also, the implementations will be tested using data from fixed receivers and rover receivers.

In order to maximize the usefulness of the results of this work to the GNSS research community, the implementation of these methods will be open source, and released in the context of the GPSTk.

Finally, modifications to current methods or new methods will be proposed when deemed applicable.

Structure of this thesis

This thesis consists of the following parts:

- *Introduction*. The current section, presenting the motivation, objectives, methodology and structure of this thesis.
- *Chapter 1: The GPS Toolkit*. In this part, the basic characteristics of the GPSTk will be introduced, as well as its state of development when this thesis work started and the initial work done.
- *Chapter 2: GNSS Data Structures*. Given the complexity of GNSS data processing software, a novel way to handle GNSS data management issues was developed, as well as an associated "processing paradigm". This chapter explains these developments and presents some simple pseudorange-based examples.
- *Chapter 3: Phase-based positioning*. It contains the author efforts to add phase-based capabilities to the GPSTk, including the full implementation of the PPP strategy. The results are compared with other state-of-the-art GNSS software suites.
- *Chapter 4: Precise Orbits Positioning*. The advantages of the data abstraction provided by the GNSS Data Structures are pushed forward and a PPP-like network method, that computes satellite clock offsets on the fly, is implemented. This method is tested and applied to aircraft positioning.
- *Chapter 5: Velocity and acceleration determination*. In this chapter, a known carrier phase-based method for precise velocity and acceleration determination is implemented and modified, improving its precision. Taking ideas from this method and the method explained in Chapter 4, a new

method called "Extended Velocity and Acceleration determination (EVA)" is proposed, that greatly extends the effective range.

- *Conclusions*. The final part presents overall conclusions from this thesis, as well as a list of further research lines that follow from the present work.

Also, in order to improve reader's awareness about the topics presented in the former chapters, several appendices are added:

- Appendix A, *GNSS fundamentals*. Introduces fundamental concepts related to this thesis. It delivers an introduction to the GNSS and their observables, plus the usual methods for building and solving the equation systems.
- Appendix B, *C++ basics*. This appendix presents a general review of American National Standards Institute (ANSI) C++ concepts.
- Appendix C, *GPSTk basics*. A very simple introduction to the GPSTk is provided in this appendix, including short programs and their results.
- Appendix D, *GPSTk documentation*. An example of the excellent quality of `Doxygen`-generated documentation is found in this appendix (for `SolverPPP` class).
- Appendix E, *Porting the GPSTk to the Gumstix*. Here the process to port part of the GPSTk to an embedded hardware platform (the Basix 200 Gumstix) is described.

Chapter 1

The GPS Toolkit

One of the goals of this thesis is to develop a set of state-of-the-art GNSS data processing tools and make them available for the research community. In order to maximize their usefulness, those tools should be designed and developed in such a way that it would ease their reuse, modification, maintenance and distribution among researchers.

Taking the former specifications into account, it was decided that the software development effort would be done within the frame of the open source GPSTk project. When the work of this thesis started, the capabilities of the GPSTk library were mainly limited to pseudorange-based GNSS data processing. From this starting point further capabilities were added, aiming to develop full carrier phase-based processing facilities.

This chapter explains the characteristics of this project, how it can be ported to some embedded hardware platforms, and the first contributions made during the development of this work (pseudorange-based only), setting the basis for the additional contributions explained in the following chapters.

1.1 GPSTk general description

The GPSTk project is an advanced open source GNSS data processing suite initiated and supported by the Applied Research Laboratories of the University of Texas (ARL:UT), aiming to provide a world class GNSS library computing suite to the satellite navigation community.

One of the main goals of the GPSTk is to free the research community from implementing common GNSS algorithms, providing a publicly accessible software repository, well documented and extensible, where those algorithms may be found and freely used.

The initial code of the GPSTk was released in summer 2004 and presented at the ION GNSS 2004 congress [Tolman et al., 2004], and its functionality has been continuously improving. A very brief list of the tools provided by the GPSTk includes:

- Handling of observation data and ephemeris in RINEX and SP3 formats.
- Mathematical, statistical, Matrix and Vector algorithms.
- Time handling and conversions.
- Ionospheric and tropospheric models.
- Cycle slip detection and correction.
- Least Mean Squares (LMS) solvers and extended Kalman filters, etc.

The website of the project may be found at http://www.gpstk.org.

1.2 GPSTk development philosophy

As an open source project, the GPSTk is released under the GNU Lesser General Public License (LGPL), allowing freedom to develop both commercial and non-commercial software based on it, and it is actively maintained by a dozen developers around the world using the Internet as communication medium.

The GPSTk Project is heavily based on object-oriented programming principles, ensuring a modular, extensible and maintainable source code. It also provides recommended coding standards for its developers, in order to foster easily legible code.

Although being an open source project, prior to July 2006 the source was provided only as compressed snapshots ([Harris et al., 2006]). The first contributions done in the framework of this PhD. thesis were provided as separated source code “patches”. Other developers external to ARL:UT sent their contributions via email, to be later merged with an internal repository.

Currently, the source code is managed using the development facilities provided by the popular SourceForge open source application repository. In particular, the repository is accessed using the `Subversion` source code management tool.

The advantage of this system is that currently the users have direct access to the last development version. It is enough to write the following line from the console of a Linux/Unix system[1]:

```
$ svn checkout https://svn.sourceforge.net/svnroot/gpstk
```

For users looking for the last stable version instead of the development one, it can be downloaded from the project website at http://www.gpstk.org.

1.3 GPSTk structure

The GPSTk software suite consists of a core library, some accessory libraries and extra applications.

The core library provides several functions that solve common processing problems associated with GNSS (for instance, proper parsing of Receiver INdependent EXchange format (RINEX) files) and it is the basis for more advanced applications distributed as part of the GPSTk suite.

On the other hand, the accessory libraries provide classes that, although useful in GNSS data processing, are very specialized or do not meet the portability standards of the core library, requiring libraries or system functions that are broadly available but not part of the C++ standard. Those libraries are found in the `gpstk/dev/lib` subdirectory.

Apart from the libraries, the GPSTk suite comes with a wealth of GNSS applications ready to run, explore and include in new developments. Full applications may be found in the `gpstk/dev/apps` subdirectory, and interesting and easy to follow examples are located at `gpstk/dev/examples`.

1.4 GPSTk documentation

A very important feature of a project of this nature is its documentation. In this regard, the GPSTk is profusely documented using the `Doxygen` documentation system, providing a very complete Application Programming Interface (API).

`Doxygen` allows the user to create a very complete set of documentation right from the GPSTk source code. Using special comment tags, the GPSTk developers write the documentation while they write the code.

[1]There are `Subversion` clients available for other operative systems.

In order to generate the documentation from a Linux or Unix system, the user must install the `Doxygen` tool, change to the `gpstk/dev/` subdirectory, and invoke the tool from the command line:

```
$ doxygen
```

For other platforms, the API from the last stable version is available in the project website at http://www.gpstk.org.

In Appendix D you will find an example of `Doxygen`-generated documentation for `SolverPPP` class.

1.5 GPSTk portability

The GPSTk provides a highly platform-independent software code base thanks to the use of the ANSI C++ programming language. It is reported to run on Microsoft Windows, as well as Linux, Solaris, Macintosh OS X, AIX, and other UNIX-based operating systems.

Also, it may be compiled using several versions of free and commercial compilers, such as `g++`, Microsoft Visual Studio C++ .NET 2003 (Version 7), Microsoft Visual C++ Express 2005 (Version 8), Forte Developer, International Business Machines Corporation (IBM) VisualAge, etc. Compilation can be carried out both in 32 bits and 64 bits platforms.

One of the first tasks carried out in this thesis was to test the capabilities of the GPSTk and, in particular, its portability. The results of porting part of the GPSTk to the Gumstix embedded boards are included in the proceedings of the 3rd. ESA Workshop on Satellite Navigation User Equipment Technologies (NAVITEC'2006) [Salazar et al., 2006].

In that work the lowest-end board was used: The Basix 200, running at 200 MHz with 64MB SDRAM, 4MB Strataflash and a Reduced Size Multi Media Card (RS-MMC) slot. This board has a power requirement of less than 250 mA at full load, and its price was about 80 Euros. Figure 1.1 shows a Basix 200 Gumstix board.

The process to port the GPSTk to the Basix 200 Gumstix can be found in Appendix E. Also, please consult [Salazar et al., 2006].

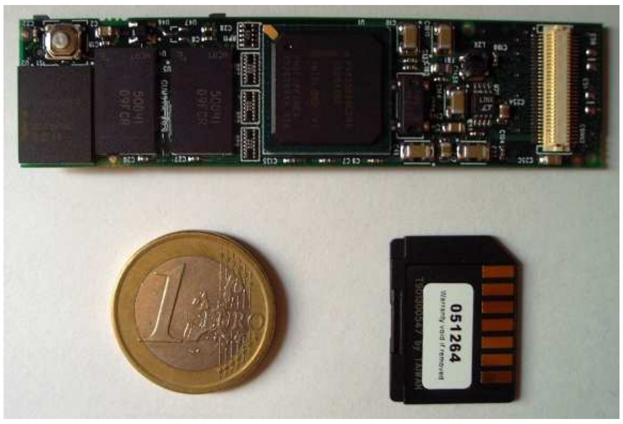


Figure 1.1: *Basix 200 Gumstix embedded board.*

1.6 Initial GPSTk functionality

The GPSTk library provides several different modules grouping the classes by common functions. It is important to emphasize that when this author started his work on the GPSTk, the facilities provided by the library were mainly limited to pseudorange data processing[2].

Table 1.1 summarizes some of the the most important and used classes available when this work started. This list is by no means exhaustive, but gives a very rough idea of the initial GPSTk capabilities.

It can be seen that the GPSTk already provided a very important set of facilities for researchers in the GNSS area. Worth of mention are class `DayTime` for time management, classes `Vector` and `Matrix`, RINEX files-related classes (`RinexObsStream`, `RinexNavStream`, etc.) and satellite ephemeris classes (`RinexEphemerisStore` and `SP3EphemerisStore`, among others).

The author of this work started from that point, developing several additional classes aiming to enhance and ease the pseudorange-based data processing tasks. A few of those additional initial classes are presented in Table 1.2.

1.7 Validation of the GPSTk with BRUS

After the first additional classes were added, it was considered that GPSTk validation was a priority in order to confirm that the initial classes provided a

[2]At the time, some carrier phase-based data processing support was provided as part of *vecsol* and *DDBase* applications, but not as separated, easy to use classes.

FUNCTION	CLASS NAME	REMARKS
Time handling	ANSITime	"Seconds since Unix epoch" representation
	CivilTime	Common year/month/day/hour/min/sec time
	DayTime	Time representation for all common formats
Formatted I/O	FFStream	Formatted File Stream
	FFData	Formatted File Data
	RinexObsStream	I/O on RINEX Observation files
	RinexNavStream	I/O on RINEX Navigation files
	RinexMetStream	I/O on RINEX Meteorological files
	SP3Stream	I/O on SP3 files
Atmospheric models	IonoModel	Klobuchar ionospheric model
	SimpleTropModel	Simple Black tropospheric model
	SaasTropModel	Saastamoinen tropospheric model
	NBTropModel	New Brunswick tropospheric model
	GGTropModel	Goad and Goodman tropospheric model
Ephemeris	EngAlmanac	Almanac information for the GPS constellation
	EngEphemeris	Ephemeris information for a single satellite
	RinexEphemerisStore	Interface to read RINEX Navigation data
	SP3EphemerisStore	Interface to read SP3 Navigation data
Solution algorithms	PRSolution	Compute a position and time solution using RAIM
Math tools	Vector	Mathematical vector representation
	Matrix	Mathematical matrix representation
	SVD	Singular value decomposition of a matrix
	LUDecomp	Performs the lower/upper triangular decomposition
	PolyFit	Computes a polynomial fit
	RungeKutta4	Provides a collection of integration routines
	Stats	Conventional statistics for one sample
	TwoSampleStats	Conventional statistics for two samples
	Expression	Solves general mathematical expressions at run time
Coordinates	ECEF	Earth centered, Earth fixed geodetic coordinates
	Position	Common 3D geographic position format
	Triple	Three-dimensional vectors
	Xvt	Earth centered, Earth fixed position/velocity/clock
	GPSGeoid	Geodetic model defined in ICD-GPS-200
	WGS84Geoid	Geodetic model defined in NIMA TR8350.2
Miscellanea	CommandOption	Set of several command line options
	BasicFramework	Basic framework for programs in the GPS Toolkit
	Exception	Base class for all exception objects
	FileFilter	Sorts and filters file data
	FileHunter	Finds files matching specified criteria

Table 1.1: *Some basic GPSTk classes.*

FUNCTION	CLASS NAME	REMARKS
Atmospheric models	GCATTropModel	Tropospheric model for GCAT software
	MOPSTropModel	RTCA/DO-229D tropospheric model
	NiellTropModel	Tropospheric model with Niell mapping functions
Math tools	Cholesky	Computes Cholesky decomposition of a matrix
	CholeskyCrout	Implements Cholesky-Crout algorithm
	SimpleKalmanFilter	Implements basic Kalman filter algorithm
Solution algorithms	Bancroft	Algorithm to get initial guess of receiver position
	ModeledPR	Compute modeled pseudoranges
	SimpleIURAWeight	Assigns weights to satellites based on IURA
	MOPSWeight	Assigns weights to satellites based on DO-229D
	DOP	Computes Dilution Of Precision values
	SolverLMS	Computes the Least Mean Squares (LMS) solution
	SolverWMS	Computes the Weighted-Least Mean Squares (WMS) solution
	CodeKalmanSolver	Computes pseudorange-based EKF solution
Observable handling	CodeSmoother	Smoothes code observable with corresponding phase
	ExtractData	Eases data extraction from RinexObsData objects
	ExtractPC	Extracts and compute PC combination
Miscellanea	ConfDataReader	Parses and manages configuration data files
	SunPosition	Computes Sun position in ECEF
	MoonPosition	Computes Moon position in ECEF

Table 1.2: *Some additional GPSTk classes.*

solid base to work upon.

Therefore, a validation study of those initial GPSTk capabilities was carried out and the results were reported in the proceedings of the 7th. Geomatics Week ([Salazar et al., 2007]). The following sections present a brief summary of that work.

For validation purposes, in this section the data results from one of the aforementioned example programs (`example-b.cpp`) will be compared with the results yielded by the "Basic Research Utilities for SBAS (BRUS)" software package.

BRUS [Hernandez-Pajares et al., 2003b] is a software package developed by gAGE/UPC and designed to be compliant with [RTCA/SC-159., 2006] (Minimum Operational Performance Standards (MOPS)). BRUS has been in use since January 2002, first in the EGNOS System Test Bed (ESTB) Data Collection and Evaluation project of EUROCONTROL[3] to process and analyze weekly data sets, and currently to monitor the real EGNOS signal for its operational certification. Thence, it is a tested software suitable for comparison purposes.

BRUS in composed of three different parts: `B2AConv` (Binary to ASCII GPS measurements Converter), BNAV (BRUS NAVIGATOR) and BNAL (BRUS NAVIGATION ANALYZER). In particular, the part relevant to this comparison is the navigation module BNAV. Version 3.2.1 was used.

Given that the GPSTk has not yet implemented the modules regarding EGNOS messages, then BNAV was configured to ignore EGNOS messages and only implement the MOPS standards regarding modeling.

1.7.1 Validation at range domain

In order to compare the results, the first approach to validate the results has been to compare the modeling of some important parameters for a given station, time span and satellite. The work leading to [Salazar et al., 2007] included a thorough comparison of all the modeled parameters involved in GPS' Standard Positioning Service (SPS).

To be brief, in this section only the comparison of the "Prefilter Residuals"[4] for EBRE station and satellite PRN #14 at 2002/01/30 will be shown. For this type of validation, the comparison of "Prefilter Residuals" is the most important part because they combine the information of all modeling algorithms, and therefore

[3]EUROCONTROL is the European Organisation for the Safety of Air Navigation.

[4]Difference between observations and modeled estimations. Please consult Section A.4 for further information.

the *difference* of their values for a pair of GNSS data processing tools is an effective way of compare their performance.

In this case, Figure 1.2 shows that the agreement is remarkably good: Differences between Prefilter Residuals are within 1 mm (i.e., within quantization noise given that the output resolution was 1 mm), confirming that the applied algorithms are equivalent.

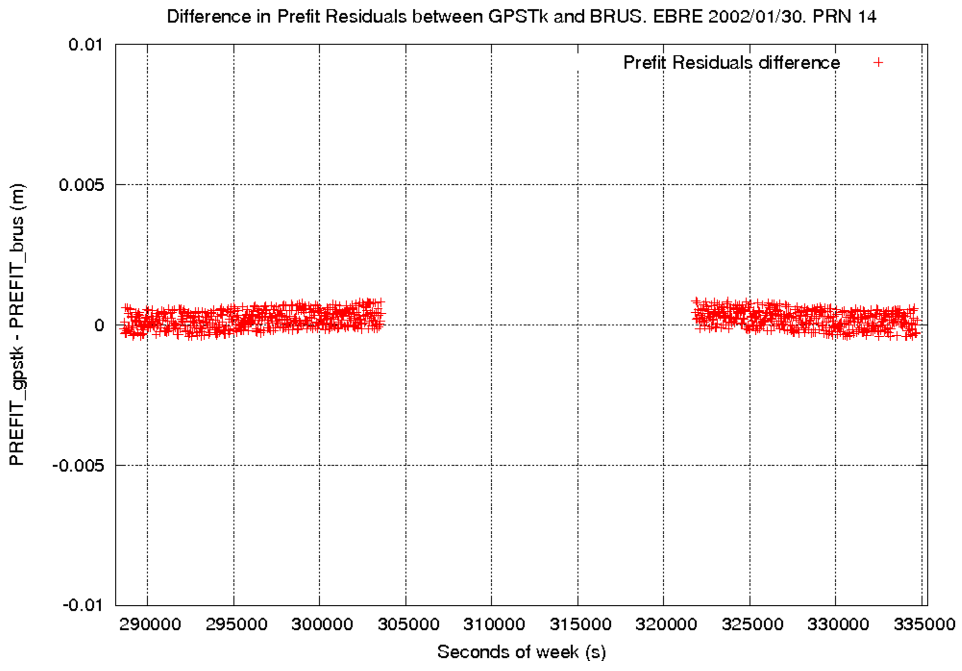


Figure 1.2: *Difference in Prefilter Residuals between* `example-b` *and BRUS.* EBRE *2002/01/30. PRN #14.*

1.7.2 Validation at position domain

A second approach to compare the results is to compute the vertical and horizontal positioning errors for a fixed receiver with known coordinates. In this section, `COCO` station (ecuatorial latitude) at 2000/07/26 was used.

It can be seen in Figure 1.3 how the GPSTk-based program match very well with BRUS, typically better than several centimeters, for the vertical error at `COCO`. The main differences, specially at the end of the data set, are due to satellites being dropped by `example-b.cpp` (i.e., the GPSTk) before being dropped by BRUS, and both tools must implement their equation solvers in slightly different ways.

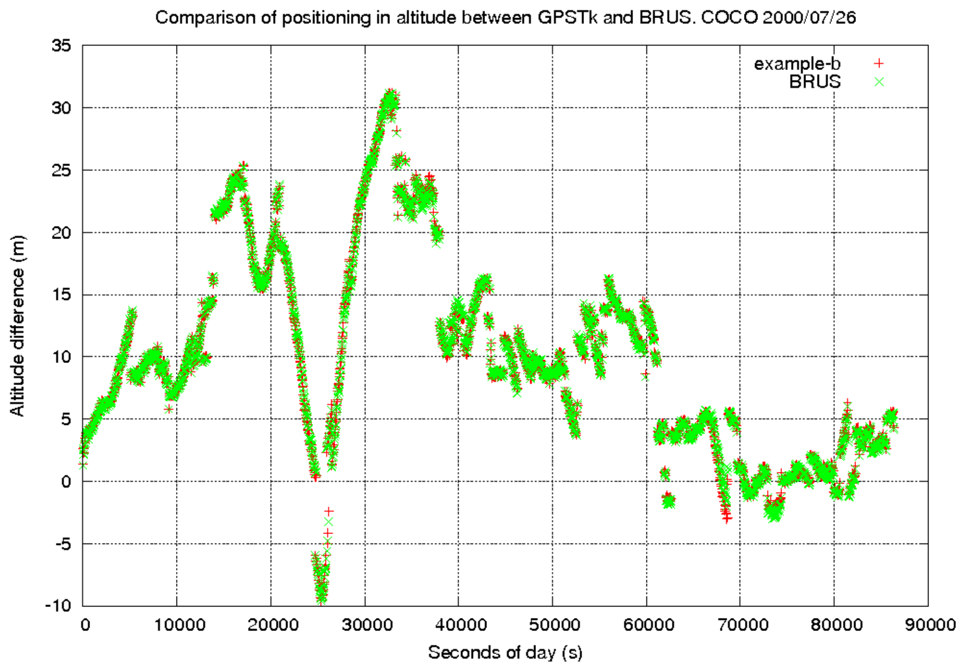


Figure 1.3: *Comparison of vertical error between* `example-b` *and BRUS.* `COCO` *2000/07/26.*

On the other hand, Figure 1.4 shows a similar agreement in horizontal error at `COCO`.

1.8 Summary

In this chapter the general characteristics of the GPSTk project were presented, such as structure, basic facilities and development philosophy

In particular, it was shown the high level of portability of the GPSTk, combining it with an advanced embedded system (Gumstix Basix 200) in an easy an effective way to develop applications able to process GNSS data.

The former work resulted in a publication in the congress proceedings of the 3rd. European Space Agency (ESA) Workshop on Satellite Navigation User Equipment Technologies (NAVITEC'2006): [Salazar et al., 2006].

Also, the validation of the pseudorange-based processing capabilities was carried out. For this, the BRUS software package developed by gAGE/UPC has been used, given its proven performance during the development of the ESTB project and the current monitoring of the EGNOS system.

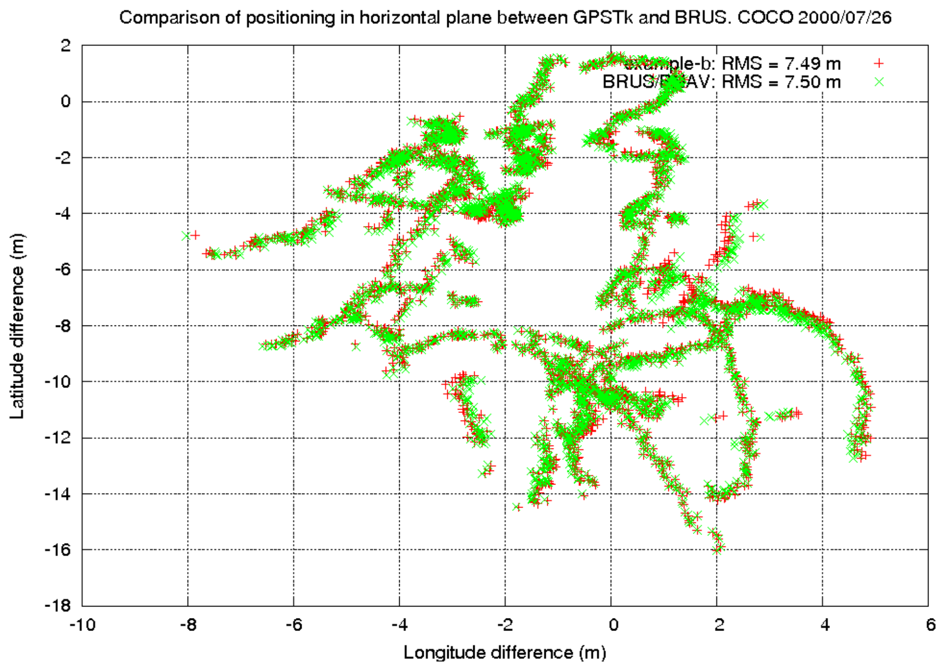


Figure 1.4: *Comparison of horizontal error between* `example-b` *and BRUS.* `COCO` *2000/07/26.*

The results of BRUS and the GPSTk show an excellent agreement both in the positioning domain (vertical and horizontal components of error for a couple stations at different latitudes and epochs) and in the modeling data. This confirms the viability of the GPSTk as a source code base for developing reliable GNSS data processing software. The GPSTk validation study resulted in a publication in the congress proceedings of the 7th. Geomatics Week ([Salazar et al., 2007]).

In summary, during the first part of this thesis work it was demonstrated that the GPSTk, although then mostly limited to pseudorange-based data processing only, already provided very interesting characteristics for the GNSS research community, and it represented a solid source code base to build upon.

Chapter 2

GNSS Data Structures

This chapter presents the GDS, a novel GNSS data management strategy that makes possible to organize complex problems in simple ways.

The GDS and their associated "processing paradigm" are considered an important contribution of this thesis, because the source code resulting from using them is remarkably compact and easy to follow, yielding better code maintainability and supporting the overall GPSTk design goal of "to free researchers to focus on research, not lower level coding" ([Harris et al., 2006]), resulting in an increased researcher productivity.

2.1 Motivation

After validating the code-based results from the GPSTk, and confirming its good portability characteristics, the author started to add carrier phase-based processing capabilities.

Shortly after starting this task, some project developers started to face, with increasing frequency, several data management issues that were difficult to deal with when using just vectors and matrices.

The task of writing source code supporting complex data processing, and being at the same time compact and easy to follow and maintain, was increasingly hard to achieve.

In order to illustrate this situation, let's consider the simple case of using C1 code pseudoranges from a RINEX observation file[1], smoothing them with corresponding L1 phases:

[1]This discussion is also valid for real-time settings, but it will be restricted to post-process for simplicity sake.

- The RINEX file is parsed and C1 and L1 observables are extracted. Each observation must be related to (or indexed by) the satellite it belongs to. Also, the receiver-generated cycle slip (CS) flags should also be parsed, indexed and stored for cycle slip detection purposes.

- A capable CS detector should implement algorithms taking into account data biases and variances. This extra information must also be indexed by source and satellite. Additionally, epoch-related information (for instance, to compute filter window length) is needed.

- With these extra data, and corresponding relationships among them, the code-smoothing routine may proceed.

A wide number of approaches are used by researchers to implement this kind of code. Vectors and matrices are commonly used to store observations and other intermediate data, and each researcher develops some type of ad hoc look up tables to store the relationships among data.

However, this common approach to the GNSS data management problem has a very important disadvantage: *it is difficult to scale*. For instance, relations among data will change when visible satellites change, prompting appropriate refreshing of the tables; also, using more than one source (for instance, in DGPS) adds more tables that must be linked with the previous ones. More complex processing requires different types of relationships, yielding yet more types of look up tables.

Therefore, the complexity of the software grows very fast when pursuing sophisticated GNSS processing strategies, dramatically increasing the possibility of errors. As a result, the GNSS researcher devotes an ever increasing amount of time looking for errors in his own code. Besides, each processing strategy generates software that is crafted in a very specific way (because of the very particular data relations it needs), which impairs code reuse. Therefore the researcher also devotes a lot of time readapting his own previous routines to solve an already solved problem (but in a different context), potentially introducing more errors in this process.

Facing this situation, this thesis introduces a novel approach to GNSS data processing software development. This approach is based in a hierarchy of data structures coping with data management issues in a consistent way. That is the origin of the GDS and the associated "GDS Processing Paradigm".

2.2 Explaining GNSS Data Structures

In order to solve the GNSS data management problem in a flexible, consistent and comprehensive way, the GDS were developed and added to the `procframe` auxiliary library of the GPSTk. First introduced in [Harris et al., 2007], the GDS and their associated "GDS Processing Paradigm" have been continuously evolving and improving since their inception. See for instance [Salazar et al., 2008a], [Salazar et al., 2008b] and [Salazar et al., 2009b].

The GDS hold several kinds of GNSS-related data, indexed by station, epoch, satellite and type. In this way, both the data and corresponding "*metadata*" (data relationships) are preserved, and data management issues are properly addressed. The indexing is done automatically (i.e., without researcher explicit intervention) because classes conforming to the "GDS Processing Paradigm" *must* fulfill some requirements including proper metadata handling and data indexing.

GDS take advantage of the observation that several types of GNSS-related data structures share some common characteristics, and thence they can be handled in an unified way. These structures index each data value with four different indexes:

- *Source*: The GNSS receiver the data is related to.
- *Epoch*: The time the data belongs to.
- *Satellite*: The satellite the data value is related to.
- *Type*: The type of data the value represents, for instance C1, P1, L2, cycle slip flag on L1, etc.

Those indexes are implemented in the GPSTk as C++ classes named `SourceID`, `SatID`, `TypeID`, and `DayTime`. Objects associated with these classes provide the researcher with a large set of methods to work with them in an easy way. Please refer to GPSTk's API document for details.

2.2.1 GDS examples

In the following sections some conceptual examples about how GDS are used to encapsulate GNSS-related data will be presented.

2.2.1.1 RINEX files

For a better understanding of how GDS work, the typical structure of a RINEX observation file will be reviewed. The data structure for any given RINEX epoch record may be modeled as an "*inverted tree*", as shown in Figure 2.1.

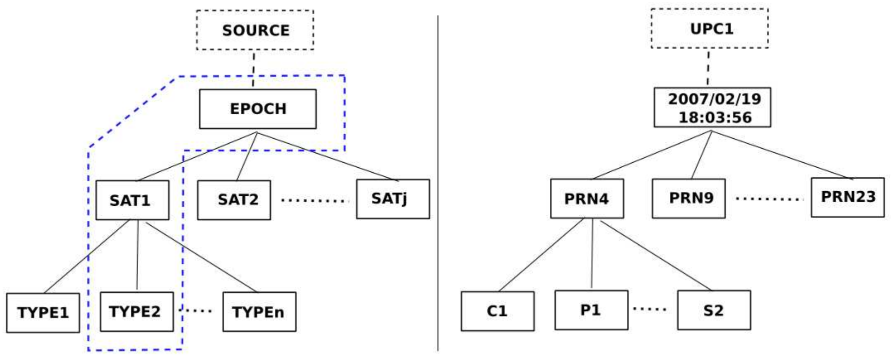


Figure 2.1: *Single-epoch RINEX data structure. General model at left, example with indexes at right.*

It can be seen that the data in a RINEX observation file is organized using a hierarchy of indexes providing access to any given value. In Figure 2.1 the data values themselves (i.e., the observations) are not shown: they are "*attached*" to the indexes at the bottom level of the inverted tree.

Traversing the tree in a top-down direction, the first level is the epoch (time), second level corresponds to satellite Pseudo-Random Noise (PRN) (one satellite per row in RINEX Version 2 observations files[2]), and then comes the type the observation belongs to (related to columns in the data file). The right part of Figure 2.1 shows an example of how this tree may look with some indexes set.

Please note that there is an implicit index on top of this tree: The source. Each RINEX observation file usually stores the data from one GNSS receiver only[3]. Also, note the fact that for this data structure, the source and epoch indexes are common for all values, whereas the satellite PRN and data type indexes are value-specific.

It is important to emphasize that only four indexes were needed to fully identify each RINEX data value: `SatID`, `TypeID`, `SourceID` and `DayTime`. Besides, with a careful implementation of these indexes (in particular `SatID`), several types of GNSS (GPS, Galileo, GLONASS, etc.) may be transparently handled.

[2]Support for RINEX Version 3 files is an ongoing work.

[3]From RINEX Version 2 onwards it is allowed to include observations from more than one site, but it is not recommended, [Gurtner, 2001]. For multiple antenna cases, each antenna must be handled as a different receiver.

The researcher then has to move along a given branch of the data tree in order to get an specific observable out of a RINEX data file. This is represented by the blue dashed shape at right side of Figure 2.1, and it implies that every RINEX GNSS observable value may be unequivocally identified by the aforementioned four indexes: source, epoch, satellite ID, and data type. Whether the use of all or part of these indexes is necessary for a given application is a matter of convenience, but nevertheless they are always explicitly or implicitly present.

Another common example is a RINEX data set, i.e., a set of observable values from a given source, epoch and observation type, but differing in satellite ID. It appears, for instance, when all C1 observations for a given epoch are extracted. Figure 2.2 shows a typical tree composed of the required branches.

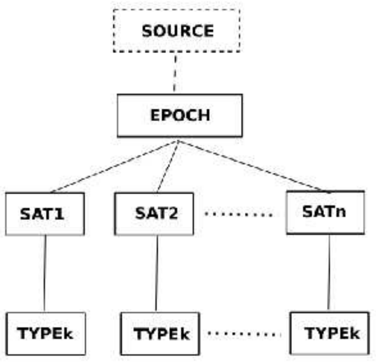


Figure 2.2: *RINEX data set.*

However, given that all values share the same type, a more efficient way to represent the former data structure is the one shown in Figure 2.3.

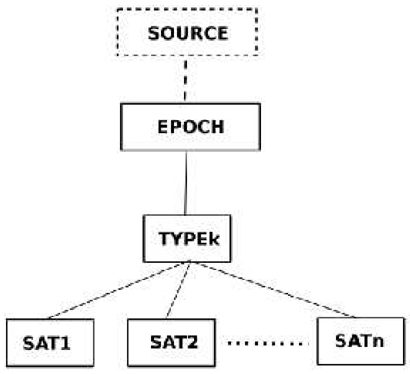


Figure 2.3: *More efficient RINEX data set.*

These structures will be revisited in the next sections.

2.2.1.2 Signal model

The GDS may be used to model several different GNSS-related data structures, for instance, the typical signal propagation model for pseudorange processing:

$$P_i^j = \rho_i^j + c(dt_i - dt^j) + rel_i^j + T_i^j + \alpha_f I_i^j + K_{f,i}^j + M_{P,i}^j + \epsilon_{P,i}^j \qquad (2.1)$$

Where:

- P_i^j : Pseudorange observation for satellite j (SV^j) from receiver i (RX_i).
- ρ_i^j : Geometric distance between SV^j and RX_i antenna phase centers.
- dt^j : Offset of SV^j clock with respect to GPS Time.
- dt_i : Offset of RX_i clock with respect to GPS Time.
- rel^j : Bias due to relativistic effects (linked to SV^j orbit eccentricity).
- T_i^j : Tropospheric delay.
- $\alpha_f I_i^j$: Ionospheric delay. This effect is frequency-dependent ($\alpha_f = 40.3 \cdot 10^{16}/f^2$ when I is expressed in TECU and f is in Hz).
- $K_{f,i}^j$: Frequency-dependent term due to the instrumental delays in SV^j and RX_i electronics.
- $M_{P,i}^j$: Multipath effect. It is environment-dependent, including frequency and code dependencies.
- $\epsilon_{P,i}^j$: Noise and unmodeled effects for code measurements. It is code-dependent.

Each term of Equation 2.1 is identified by its type (P, ρ, rel, etc.), receiver it belongs to (i), and satellite (j). Please note that in this case the epoch index is implicit: It is supposed that a given model is valid for a specific epoch.

Also, it is important to bear in mind that in this case data types are beyond the typical RINEX observables: Tropospheric, ionospheric and relativity delays are some examples. Therefore, it is important for `TypeID` to include a wide range of data types used in GNSS data processing and, if possible, it should be easily extensible.

The ability to grow beyond the original RINEX data types is paramount: *A very important data abstraction level is achieved in this way.*

Figure 2.4 represents this kind of "data structure" applied to the GNSS signal propagation model. Comparing Figure 2.4 with Figure 2.3 it can be confirmed that, although the data structures are different, the same four basic indexes are used and the major difference lies on which indexes are common to the values.

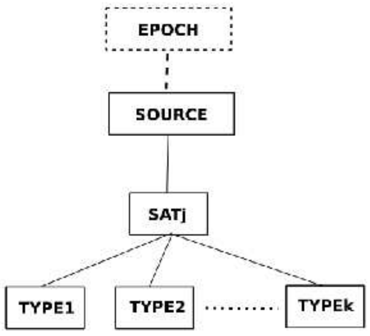


Figure 2.4: *Representation of a GNSS signal propagation model.*

If several structures such as Figure 2.4 were put together (holding data of several satellites), the resulting structure would look just like Figure 2.1.

2.2.1.3 Equation systems

Following the former data abstraction methodology, GNSS equation systems are also good candidates to have generic data structures. This could be useful when using solving methods where data is added and removed in a dynamic way.

When solving code-based, one-receiver GNSS data, it is common to build an equation system composed of equations like Equation 2.2:

$$Prefit_i^j = \Big(\frac{x_{i0} - x^j}{\rho_{i0}^j}\Big)dx_i + \Big(\frac{y_{i0} - y^j}{\rho_{i0}^j}\Big)dy_i + \Big(\frac{z_{i0} - z^j}{\rho_{i0}^j}\Big)dz_i + c.dt_i \tag{2.2}$$

Where:

- $Prefit_i^j$: Prefilter residual, i.e., difference between observation and modeled effects for satellite SV^j as seen from RX_i.
- (x_{i0}, y_{i0}, z_{i0}) : A priori position of receiver i.
- ρ_{i0}^j : A priori geometric distance between receiver i and satellite j antenna phase centers.

- (x^j, y^j, z^j) : Position of satellite SV^j.
- (dx_i, dy_i, dz_i) : Corrections to (x_{i0}, y_{i0}, z_{i0}). (Parameters to be estimated by the solver).
- dt_i : Offset of receiver clock with respect to GPS Time.
- c : Speed of light.

Equation 2.2 may also be modeled with a data structure like Figure 2.4 (just as Equation 2.1 was). Putting several of these equations together in an equation system, it is evident that each *row* in the equation system corresponds to the data structure shown in Figure 2.4, whereas each *column* is represented by the data structure in Figure 2.3 (different satellites, same type). Moreover, the whole equation system may also be represented by something akin to the RINEX data tree in Figure 2.1.

In summary, some structures are "*rows*" (Figure 2.4), while others are "*columns*" (Figure 2.2 and Figure 2.3), the full set is a "*matrix*" (Figure 2.1), and a specific observable, combination, coefficient or correction is just an element of a matrix.

Therefore, a properly implemented set of GNSS data structures may effectively and comprehensively encompass a wide range of data relationships, well beyond the rather simple relations stated in a RINEX file. The former simplifies data management and code reuse issues.

2.2.1.4 Other data sources

Often, the GNSS receiver is a subsystem of a more complex positioning system. Therefore, the GDS should be able to handle data from sources not directly related to GNSS systems.

For instance, with the emergence of hybrid GNSS-INS receivers it may be necessary to take into account other data streams. Imagine, for example, handling "the angular velocity read at epoch 3201.3 s by the second gyroscope of the aircraft inertial system number 1".

The data structures presented so far also fit this kind of data streams. For instance, if in the former example data stream the INS poses as `SourceID`, the gyroscope as `SatID` (this class is extensible and non-standard "satellites" may be easily added), and the angular velocity is classified as a given `TypeID` (this class is also extensible), then each INS data value can also be represented, and therefore it can be included in our unified data processing chain.

A similar approach may be used to handle other data sources like differential corrections.

2.3 GDS implementation

An efficient implementation of GDS should store just once those indexes that are common to all the GNSS data values (for instance, look at the structure in Figure 2.3). This approach cuts off as much overhead as possible, preserving at the same time the full set of information to completely identify each of the GNSS data values.

Therefore, the implementation of these structures should have something like a *header*, holding all the common indexes, and a *body*, storing the variable indexes and the data values themselves.

The "inverted tree" data structures presented in the former figures may be thought of as composed in this way: The *trunk* of the tree corresponds to the *header*, and the *branches* will form the *body* of the GDS.

This *header*/*body* approach to GDS eases implementation and improves efficiency. Moreover, the GPSTk extensively uses the C++ Standard Template Library (STL), a set of data structures and associated algorithms that efficiently implement and support this approach.

It is very important to emphasize, however, that the encapsulation provided by GDS makes unnecessary for the researcher to know the implementation details in order to effectively use the GDS.

2.4 GDS Processing Paradigm

Apart from the GDS themselves, a "*GDS Processing Paradigm*" was also developed in this thesis, where GNSS Data Structures are complemented with several associated processing classes.

With the GDS paradigm the GNSS data processing becomes like an "*assembly line*", where all the processing steps are performed sequentially. The GDS are treated like white boxes that "*flow*" from one "*workstation*" (processing step) to the next in such assembly line.

Thence, the GDS are always used as *both the input and output* of each processing step, providing an easy and straightforward way to encapsulate and process data. This paradigm allows developing clean, simple to read and use software that speeds up development and reduces errors.

The objects from these processing classes reach into the GDS and add, delete and/or modify what is needed (according to their function), and leave the results in the same GDS, appropriately indexed. These processing objects are designed to use sensible defaults in their parameters, but may be tuned to suit specific needs.

For instance, a `ModeledPR` (*Modeled Pseudorange*) object may take as parameters observable type, ephemeris, ionosphere and troposphere models, and will add to the incoming GDS some extra data such as geometric range, satellite elevation and azimuth, prefilter residuals, and so on, properly indexed by receiver-satellite pair. It will also automatically remove satellites missing critical data (as ephemeris, for example). Once the object is properly configured for a given task, the data processing is carried out without needing further adjustments.

Thanks to C++ object-oriented capabilities, all processing classes "*inherit*" from a single class: `ProcessingClass`. This is a "*pure virtual*" class that sets a common behavior to which all processing classes must adhere. As shown later, this approach furthers the data abstraction and code reuse of software using the GDS paradigm.

The former ideas are coupled with a redefinition of C++ operator ">>", implemented in such a way that several operators may be concatenated. It allows a programming style that clearly shows how the data is flowing along the processing steps (resembling the "pipes" concept used in UNIX-based systems).

In order to show the flexibility of this approach, some simple examples are presented. In the first one, a single epoch worth of data will be extracted out of a RINEX observation file, and that data will be put into a GDS:

```
1  RinexObsStream rinexFile("ebre0300.02o");
2  gnssRinex gpsData;

3  rinexFile >> gpsData;
```

Line #1 declares an object of class `RinexObsStream`, which is used to handle RINEX observation files. That object receives (in this case) the name of `rinexFile`, and it will take care of "`ebre0300.02o`" RINEX observations file. On the other hand, line #2 declares an object of class `gnssRinex`. This is a GDS and data will be stored in this object, which will be called `gpsData`.

Finally, line #3 takes one epoch of data out of `rinexFile` and will put it into `gpsData`. No more code is needed for this action, and line #3 is referred to as the “*processing line*”. Please note how the C++ operator “>>” is used to convey the idea that data “*flows*” out of the RINEX file into the GDS “*box*” that will carry GNSS data around.

It is important to emphasize that the statement `rinexFile >> gpsData` implies that a *full* epoch worth of data is taken out of the RINEX observation file and “poured” into `gpsData`, filling in a structure just as the one showed in Figure 2.1 with several different satellites and their corresponding observations, everything appropriately organized.

Also, the statement `rinexFile >> gpsData` has the additional property that it is evaluated as TRUE if operation completes successfully, and as FALSE otherwise (for instance, when the end of RINEX file is reached). This property allows us to modify the former example to get a much more useful behavior:

```
RinexObsStream rinexFile("ebre0300.02o");
gnssRinex gpsData;

while( rinexFile >> gpsData ) {
   // . . . put your GNSS data processing code here . . .
}
```

In this case the while loop will repeat itself until the end of RINEX file is reached, and in each repetition a single epoch data set of RINEX observations is automatically encapsulated into `gpsData`, fully available for processing.

2.5 Examples of code-based data processing

In the following sections a set of short examples providing pseudorange-based data processing with the GDS will be presented, in order to illustrate how different classes and structures may be combined to implement several data processing strategies.

2.5.1 GPS Standard Positioning Service (SPS)

In this example the GPS SPS ([DoD, USA, 2008]) will be presented. For space reasons most of the initialization phase is skipped. For further details, please consult the GPSTk API).

Also, the GPSTk provides carefully explained examples: Look at `example6.cpp` and `example7.cpp` in the `examples` directory[4]. Most of the examples pre-

[4]A detailed list of GPSTk examples may be found online at:

sented here are modified versions of the aforementioned programs.

This example starts with the lines handling broadcast ephemeris data:

```
 1  RinexNavStream rnavin("bahr1620.04n");
 2  RinexNavHeader rNavHeader;
 3  rnavin >> rNavHeader;

 4  IonoModel ioModel;
 5  ioModel.setModel( rNavHeader.ionAlpha, rNavHeader.ionBeta );
 6  IonoModelStore ionoStore;
 7  ionoStore.addIonoModel( DayTime::BEGINNING_OF_TIME, ioModel );

 8  RinexNavData rNavData;
 9  GPSEphemerisStore bceStore;
10  while (rnavin >> rNavData) {
11     bceStore.addEphemeris(rNavData);
12  }
```

Lines #1 and #2 declare objects to take care of a broadcast ephemeris RINEX file and its associated header. Line #3 reads the header and stores it. Klobuchar ionospheric coefficients are stored in the ephemeris RINEX header, so this step is important for lines #4 to #7. The first two of them declare an "ionospheric model" object (`ioModel`) and fill it with Klobuchar coefficients, and then they declare an ionospheric model store (`ionoStore`) and push the previously defined model into it.

After that, the ephemeris data is read and stored in a proper object (`bceStore`), which is filled with all available ephemeris data, one epoch at a time, in a similar way as what was already explained for observation data.

Afterwards, some model initialization is necessary:

http://www.gpstk.org/doxygen/examples.html

```
13  Position nominalPos( 3633909.1016, 4425275.5033, 2799861.2736 );

14  MOPSTropModel mopsTM( nominalPos.getAltitude(),
           nominalPos.getGeodeticLatitude(),
           162 );

15  ModelObs gpsModel( nominalPos,
                       ionoStore,
                       mopsTM,
                       bceStore,
                       TypeID::C1 );

16  SolverLMS solver;

17  RinexObsStream rinexFile("ebre0300.02o");
18  gnssRinex gpsData;
```

Line #13 declares the nominal position of receiver in Earth-Centered, Earth-Fixed (ECEF) coordinates, line #14 setups a tropospheric model (there are several types available), and line #15 creates a `ModelObs` object (`gpsModel`).

This "*modeler*" object (`gpsModel`) takes as input the nominal receiver position, ionospheric and tropospheric models, ephemeris data and type of observable it will work with, and then it carries up the tasks related with SPS-GPS data modeling. Once it is fed with `gpsData`, `gpsModel` computes all the delays defined by the standards and uses them to get the prefilter residuals and geometric coefficients that will later be used by a "*solver*" object. All these data is automatically inserted and indexed in the GDS.

Then, line #16 declares the "*solver*", the object in charge of building and solving the equation system. In this case it uses a simple LMS solving algorithm (there are several algorithms available). The function of lines #17 and #18 was already described.

Finally, it comes the final while loop that extracts RINEX data, runs the GPS model, solves the navigation equations and prints the results:

```
19  while( rinexFile >> gpsData )
    {
20     gpsData >> gpsModel >> solver;

21     cout << solver.getSolution(TypeID::dx) << ’’ ’’;
22     cout << solver.getSolution(TypeID::dy) << ’’ ’’;
23     cout << solver.getSolution(TypeID::dz) << endl;
    }
```

Please remember from Section 2.4 that lines #20 to #23 will be carried out for each epoch of data while line #19 evaluates as `TRUE`, i.e., while there are epochs to process still available in `rinexFile`. The GNSS processing is done in line #20 (the "*processing line*"): The epoch-worth of data that was just taken out from `rinexFile` and put into `gpsData`, is then pushed through `gpsModel` (to generate the values associated with SPS-GPS signal modeling) and `solver` (to build and solve the navigation equation system).

Some important remarks are in order: It can be seen that the first part of line #20 is `gpsData >> gpsModel`, which generates the model. During that phase, all new data generated by the model is stored and indexed in `gpsData`. For instance, the relativistic delay between receiver and satellite, let's say, PRN17, is computed and stored with all metadata needed to tell it apart from relativistic delays from other satellites. This is the reason it has been previously emphasized that although GNSS data structures and `TypeID`'s may initially be seen as associated with RINEX data, they represent much more than that.

Also, if a given satellite is missing a critical piece of data (like ephemeris data, for example), it will be deleted from the `gpsData` GDS to avoid problems further down in the data processing chain.

Additionally, the output of expression `gpsData >> gpsModel` is the modified `gpsData` structure, including the data generated by the `gpsModel` object. In this way, `gpsData >> gpsModel` becomes again `gpsData`, where the new `gpsData` is a *superset* of the original.

Therefore, the second part of line #20 then becomes `gpsData >> solver`, and `solver` object will find inside `gpsData` all the information it needs to build and solve the navigation equation system.

This process is efficiently implemented using the aforementioned C++ STL. A 24-hours RINEX observation file (at a 30 s data rate) processed in this way takes less than 0.2 seconds in an average laptop PC with Linux.

The final lines #21 to #23 take care of printing the solution to the screen using the standard C++ `cout` printing object. Take note of the way to get solution values out of *solver* objects, which represents the consistent way to refer to data types along all the GDS processing paradigm.

Take note that in the examples presented in this chapter only the coordinates and the receiver clock will be estimated by the solvers, while the other parameters are either modeled or taken from broadcast values. More complex setups will be shown in later chapters.

2.5.2 C1 smoothed pseudorange with WMS

The very basic processing presented in Section 2.5.1 will be extended to do something a little more complex: The next example will use C1 observables smoothed with corresponding L1 phases, detect cycle slips, and include weighting into the solver. The solution will be presented in a North-East-Up (NEU) reference frame instead of ECEF.

The initialization phase for this processing chain is almost the same as the former example, so it will be skipped. The main change is in line #2, which in spite of spanning several physical lines, it is performed in a single processing line. New processing objects were added: Object `markCSC1` (from `OneFreqCSDetector` class) takes care of detecting and marking cycle slips using a one-frequency-only algorithm. Then, `smoothC1` object (belonging to `CodeSmoother` class) applies a C1/L1 smoothing filter.

```
1  while( rinexFile >> gpsData )
   {
2     gpsData >> markCSC1 >> smoothC1 >> gpsModel
              >> mopsW >> baseChange >> wSolver;

3     cout << wSolver.getSolution(TypeID::dLat) << '' '';
4     cout << wSolver.getSolution(TypeID::dLon) << '' '';
5     cout << wSolver.getSolution(TypeID::dH) << endl;
   }
```

The `gpsModel` object is the same as in the previous section, while `mopsW` object computes the relative weights to be applied to the satellites. There are several ways to achieve this, and `mopsW` (belonging to `ComputeMOPSWeights` class) applies the algorithms described in [RTCA/SC-159., 2006] to compute weights. It is important to emphasize that all this objects are highly configurable and may be easily extended (please consult GPSTk's API).

Then, the `baseChange` object (from `XYZ2NEU` class) reaches into the GDS-encapsulated information and computes the geometry matrix coefficients corresponding to a NEU reference system. After that, the `wSolver` object (which belongs to `SolverWMS` class) solves the equation system using the appropriate geometry matrix parameters and weights.

Please take note that the reference system to be used by `wSolver` was configured in the (skipped) initialization part.

Finally, the `wSolver` object contains the NEU solution. `TypeID`'s `dLat`, `dLon` and `dH` are used to obtain the results.

2.5.3 Ionosphere-free smoothed pseudorange (PC) with WMS

The following example deals with ionosphere-free pseudorange processing. In this case several additional tasks must be carried out:

- Compute the ionosphere-free combinations: The GPSTk provides several means to compute observable combinations. In this case, an object from class `ComputePC`, named `getPC`, will be used.

  Given that the PC combination will be smoothed, the ionosphere-free carrier phase combination must also be computed. For that, object `getLC` from class `ComputeLC` will be used.

- Compute cycle slips: Cycle slip detection is a previous necessary step to make the smoother object `smoothPC` (from class `PCSmoother`) properly work.

  Therefore, object `markCSLI` (from class `LICSDetector`) and object `markCSMW` (`MWCSDetector`) are declared.

  These objects implement to different but complementary cycle slip detection algorithms based on ionospheric and Melbourne-Wubbena combinations.

  Then, additional objects to compute those combinations are also added: `getLI` (`ComputeLI`) and `getMW` (`ComputeMelbourneWubbena`).

- Sometimes, a missing observable at the time of combination computation may cause wild variations in solver input data. Therefore, an object is inserted in the *processing line* (`pcFilter` from class `SimpleFilter`) that simply removes from the GDS those satellites whose PC combinations are outside some reasonable range[5].

- Given that the Total Group Delay (TGD) is not applicable when using PC observables, the object in charge of the modeling (`gpsModel`, from `ModelObs`) is configured to ignore the TGD[6].

The rest of the processing code follows the same pattern set at previous sections:

[5]`SimpleFilter` objects can be configured to act on other parameters, and they have adjustable limits.

[6]This criteria is only valid when PC combination is computed using P1 and P2 observables.

```
while( rinexFile >> gpsData )
{
    gpsData >> getPC >> getLC >> getLI >> getMW
            >> markCSLI >> markCSMW
            >> smoothPC >> pcFilter >> gpsModel
            >> mopsW >> baseChange >> wSolver;

    cout << wSolver.getSolution(TypeID::dLat) << '' '';
    cout << wSolver.getSolution(TypeID::dLon) << '' '';
    cout << wSolver.getSolution(TypeID::dH) << endl;
}
```

The results from this processing, as well as the results from Sections 2.5.1 and 2.5.2, are presented in Figure 2.5, which shows the northing and easting error from nominal position[7] obtained with the former processing strategies for station EBRE, January 30th, 2002.

The results are well within what it is expected from these data processing strategies.

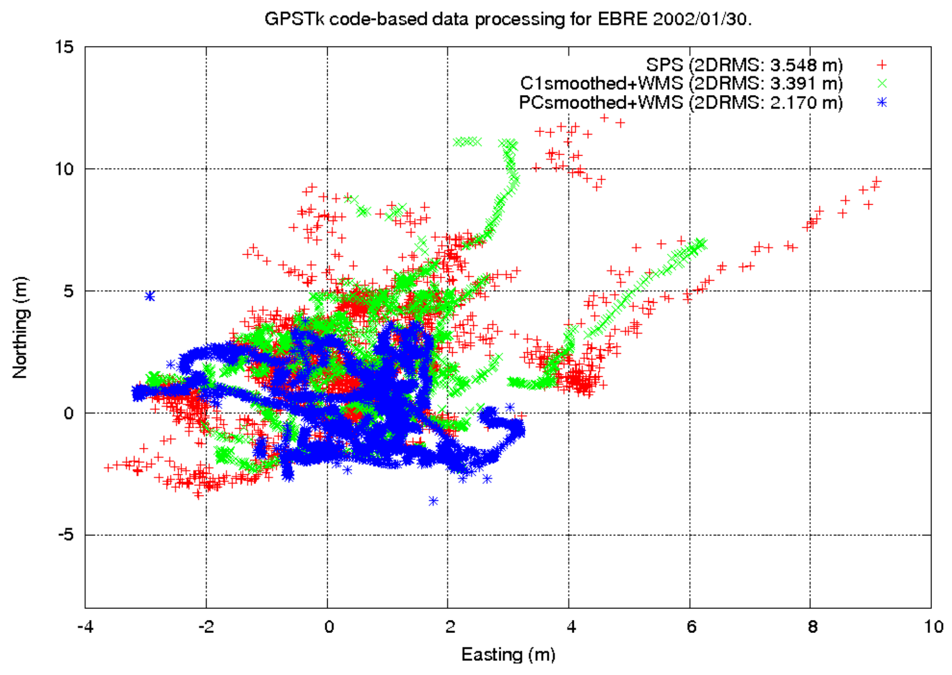


Figure 2.5: *Pseudorange-based data processing. EBRE 2002/01/30.*

[7]For these examples, the nominal position was taken from RINEX observation file header.

2.5.4 PC and WMS with additional information

In this case, a similar processing as of Section 2.5.3 will be carried out, but it will shown how additional information could be added to the solver.

The core of this example is to add a new equation to the equation system[8]. In this particular case such equation states that there are *NO changes in height* for the receiver, i.e.:

$$dH = 0 \tag{2.3}$$

This can be accomplished adding to the GDS the information corresponding to a "*fake*" satellite. From the solver's point of view, an additional satellite means an additional equation, and the following source code (inserted in the initialization phase) sets the extra information:

```
1  SatID satEq(1,SatID::systemUserDefined);

2  typeValueMap equTVMap;

3  equTVMap[TypeID::prefitC] = 0.0;
4  equTVMap[TypeID::dLat]    = 0.0;
5  equTVMap[TypeID::dH]      = 1.0;
6  equTVMap[TypeID::cdt]     = 0.0;

7  equTVMap[TypeID::weight]  = 4.0;
```

The former code lines start declaring a new "user defined" satellite called `satEq` from class `SatID` (line #1). Afterwards, a GDS called `equTVMap` is created. This GDS will contain the geometry matrix equation coefficients corresponding to Equation 2.3. Those coefficients are set in lines #3 to #6. Please note that the only coefficient that is not zero is the one for `TypeID::dH`.

Finally, line #7 assigns a relative weight to this information. Given that weights are indeed the inverse of variances, if we assign to our new equation a confidence of 0.5 m of sigma, it means that we should use a weight of $1/(0.5^2) = 4\ m^{-2}$. This is the value set in line #7.

After the data structure is fed with the additional information, the data processing is similar to Section 2.5.3, but the extra "*satellite*" is added just before the solver.

In the following source code, line #10 achieves just that: It takes the extra

[8]More complex examples will be shown in next chapters

data in equTVMap and inserts it into the *body* of the main GDS (gpsData), indexing (or *linking*) the information to our user-defined satellite satEq:

```
 8  while( rinexFile >> gpsData )
    {
 9     gpsData >> getPC >> getLC >> getLI >> getMW
               >> markCSLI >> markCSMW
               >> smoothPC >> pcFilter >> gpsModel
               >> mopsW >> baseChange;

10     gpsData.body[satEq] = equTVMap;


11     gpsData >> wSolver;

12     cout << wSolver.getSolution(TypeID::dLat) << '' '';
13     cout << wSolver.getSolution(TypeID::dLon) << '' '';
14     cout << wSolver.getSolution(TypeID::dH) << endl;
    }
```

After inserting the extra information, line #11 calls the solver (wSolver) and lines #12 to #14 print the solution. Figures 2.6 and 2.7 show the results (as error regarding nominal position) in both horizontal and vertical coordinates. The major changes are in the vertical coordinates, as expected according to Equation 2.3. Figure 2.7 also includes results when extra equation sigma is set to 5 and 10 m.

Please note that in this example the data abstraction provided by the GDS allowed to implement a different processing strategy with minor changes in code with respect to the previous processing (Section 2.5.3).

2.5.5 Differential GPS (DGPS) with WMS

This example applies pseudorange-based differential GPS techniques mixed with code smoothing. Several details are left out, but you will find more information, including several full DGPS implementations, in the GPSTk examples. The advantage of DGPS techniques is that the errors in satellite clocks are cancelled out, and strongly spatially correlated errors such as orbital and ionospheric errors are greatly attenuated.

The example starts partially processing data from a reference station (gpsDataRef), and assigning the resulting GNSS data structure as the reference data of a DeltaOp object named delta:

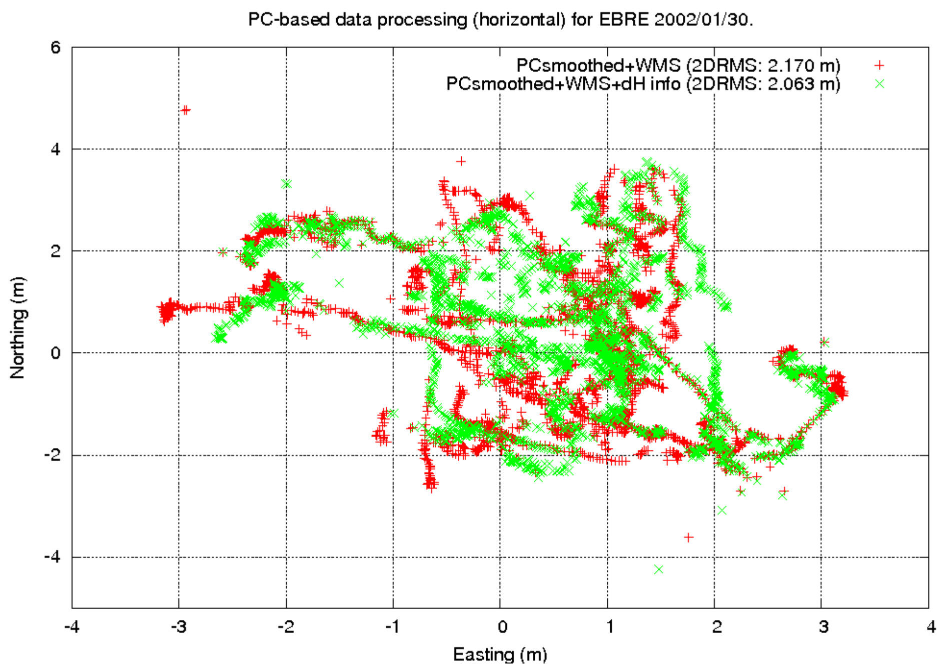


Figure 2.6: *PC processing with extra information. Horizontal coordinates. EBRE 2002/01/30.*

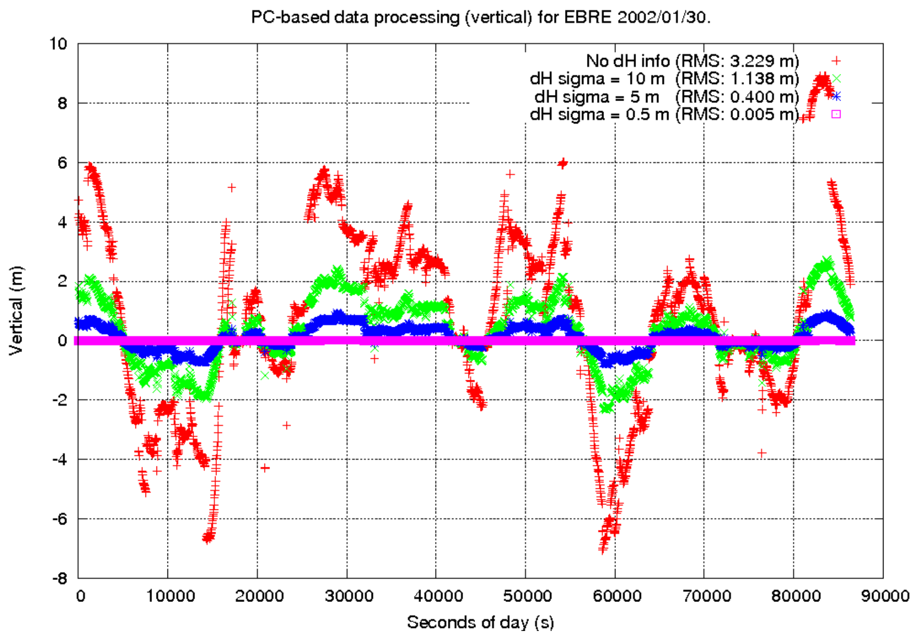


Figure 2.7: *PC processing with extra information. Vertical coordinates. EBRE 2002/01/30.*

```
1  Synchronize synchro( rinexRefFile, gpsData );

2  while( rinexFile >> gpsData )
   {

3     gpsDataRef >> synchro >> markCSC1Ref
                >> smoothC1Ref >> gpsModelReference;

4     delta.setRefData(gpsDataRef.body);
```

Note the new `synchro` object, belonging to `Synchronize` class. It is preconfigured to take care of epoch synchronization between the data structures for the receiver station and the rover receiver (`gpsDataRef` and `gpsData`, respectively).

In line #1, the `synchro` object is instructed to take data out of the reference receiver RINEX file (managed by object `rinexRefFile`) in synchronism with the `gpsData` data structure.

Therefore, initially the `gpsDataRef` data structure in line #3 is empty, but after calling the first part of the processing line (`gpsDataRef >> synchro`), `gpsDataRef` fills up with the appropriate synchronized data, and then the reference station data processing may proceed.

The processing in line #3 includes cycle slip detection, pseudorange observable smoothing with phase carrier observation, and standard modeling.

After processing the reference receiver data, the rover receiver data (`gpsData`) is processed in full:

```
5     gpsData >> markCSC1Rov >> smoothC1Rov
              >> gpsModel >> delta >> mopsW
              >> baseChange >> wSolver;

6     cout << wSolver.getSolution(TypeID::dLat) << '' '';
7     cout << wSolver.getSolution(TypeID::dLon) << '' '';
8     cout << wSolver.getSolution(TypeID::dH) << endl;
   }
```

The inserted `delta` object will substract `gpsDataRef` prefilter residuals from the corresponding `gpsData` residuals, deleting (by default) satellites not common to both receivers. The rest of the data processing is as shown in the previous examples.

2.5.6 Differential GPS (DGPS) with Kalman Filter

This case presents a DGPS strategy with a Kalman Filter[9] solver configured in *static* mode. This means that the coordinates are considered as constant, and the receiver clock offset will be handled as a white noise process with very high sigma.

The source code is essentially the same as Section 2.5.5, but the `SolverWMS` object (`wSolver`) is replaced by a `CodeKalmanSolver` object (`solverEKF`). This is an important difference with respect to the WMS solver used in the previous section, where all the variables where handled as white noise.

Another important change is that the `solverEKF` object must be configured to solve the equation systems in an NEU reference frame (instead of ECEF). The former is achieved with the following source code:

```
1  TypeIDSet typeSet;

2  typeSet.insert(TypeID::dLat);
3  typeSet.insert(TypeID::dLon);
4  typeSet.insert(TypeID::dH);
5  typeSet.insert(TypeID::cdt);

6  gnssEquationDefinition newEq(TypeID::prefitC, typeSet);

7  CodeKalmanSolver solverEKF(newEq);
```

The main idea is that object `solverEKF` must be fed with an appropriate description of the equation system it is going to solve. In this simple case, this can be achieved with a `gnssEquationDefinition` structure (`newEq`) that contains the `TypeID`'s of both the equation coefficients and the independent term.

Line #1 creates a `TypeIDSet` to hold the coefficient types, and lines #2 to #5 set those types for a NEU reference frame. Then, line #6 creates the equation description setting the independent term type (`TypeID::prefitC`) and the coefficient type set.

After the former is done, the new EKF solver is created in line #7, setting the new equation description in the initialization. No further changes are needed.

Again, GDS data abstraction allowed to modify a previous data processing strategy with minimum changes.

Results from both DGPS examples are presented in Figure 2.8, which shows

[9]Or, more properly, an Extended Kalman Filter (EKF) given that the system is linearized.

the northing and easting errors regarding nominal position for stations EBRE (Rover) and BELL (Reference), for January 30th, 2002, with a baseline of about 115 km. The results are well within what it is expected from these data processing strategies, and the improvements for the EKF case come from the fact that static positioning is used.

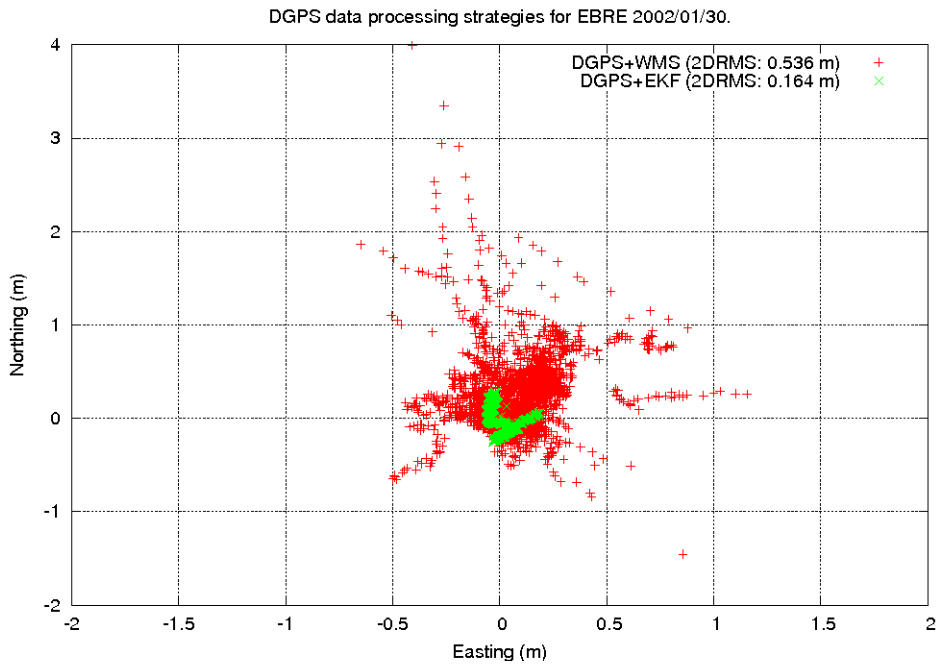


Figure 2.8: *DGPS data processing. EBRE 2002/01/30.*

2.6 Summary

In this chapter the GNSS Data Structures (GDS) were presented, including the motivation for their development, rationale, the implementation overview, and their associated *processing paradigm*. Also, several types of pseudorange-based data processing strategies were included, in order to better show how they can be used.

The key to understand the contribution of the GDS is to realize that they preserve both the data and corresponding "*metadata*" (data relationships), internally indexing all the GNSS-related information.

With the GDS paradigm the GNSS data processing becomes like an "*assembly line*", where all the processing steps are performed sequentially. The GDS are treated like white boxes that "*flow*" from one "*workstation*" (processing step) to the next in such assembly line, providing an easy and straightforward way to encapsulate and process data. This approach allows developing clean, simple to read and use software that speeds up development and reduces errors.

The GDS form a fundamental part of the work developed during this thesis. Their inclusion in the GPSTk code base prompted an invitation to participate as coauthor in a joint publication with the ARL:UT development team at ION GNSS 2007 congress proceedings ([Harris et al., 2007]), where the GDS were presented for the first time.

Also, the GDS have enabled the work leading to several other publications in congress proceedings, for instance [Salazar et al., 2008b], [Salazar et al., 2008a], and [Salazar et al., 2009b], as well as being an integral part of a paper at the GPS Solutions journal ([Salazar et al., 2009a]). More about this work in the following chapters.

Finally, reference implementations for most of the algorithms presented in this chapter have been provided in the GPSTk `examples` directory, so GNSS students and researchers may easily understand and implement the GNSS data processing strategies indicated here. Those reference implementations are files `example6.cpp` and `example7.cpp`.

Chapter 3

Phase-based positioning

The GNSS Data Structures (GDS) were introduced in the previous Chapter 2, and several examples of use for pseudorange-based GNSS data processing were presented.

However, the advantages provided by the GDS become more evident when dealing with more complex software, such as carrier phase-based GNSS data processing. Indeed, it was just when the phase-based capabilities were added to the GPSTk, that the need for advanced, consistent and easy to use GNSS data structures became evident.

In this chapter the application of the GDS to carrier phase-based GNSS data processing will be shown, highlighting the flexibility and power of the capabilities added to the GPSTk during the development of this thesis. Most of the work will be focused in the implementation and results of the Precise Point Positioning (PPP) data processing strategy.

3.1 Precise Point Positioning (PPP)

Precise Point Positioning (PPP) implementation ([Kouba and Heroux, 2001]), is an important example of carrier phase-based GNSS Data Processing. PPP is a complex task, and issues like wind-up effects, solid, oceanic and polar tides, antenna phase centers variations, etc. must be taken into account.

Also, International GNSS Service (IGS) SP3 final precise satellite orbits and clocks are used in PPP, but these products are typically provided each 900 s, while observations are usually provided each 30 s. Therefore, time management issues also arise.

The following sections will present some accessory classes that ease these complex issues. However, take into account that, again, most initialization details are skipped. For complete carrier phase-based processing implementations please read `example8.cpp`, `example9.cpp` and `example10.cpp` in the GPSTk development repository. The GPSTk API is also a mandatory read.

3.1.1 Handling configuration files

Given the potentially high number of PPP processing parameters involved, reading configuration files is an important ability in order to avoid recompilation of source code each time we want to change a given parameter.

The GPSTk provides `ConfDataReader`, a powerful class to parse and manage configuration data files. It supports multiple sections, variable descriptions and value descriptions (such as units), and a wide range of variable types.

Given a configuration file named `configuration-file.txt`, whose content is:

```
# Default section

   tolerance, allowed difference between epochs = 1.5, secs

[BELL]

   reference = TRUE
```

Then a typical way to use this class follows:

```
1  ConfDataReader confRead;
2  confRead.open("configuration-file.txt");

3  double tolerance = confRead.getValueAsDouble("tolerance");

4  cout << confRead.getVariableDescription("tolerance") << endl;
5  cout << confRead.getValueDescription("tolerance")    << endl;

6  bool bellRef = confRead.getValueAsBoolean( "reference",
                                              "BELL"      );
```

Lines #1 and #2 declare the `ConfDataReader` object and open the configuration file. Line #3 declares a double precision variable called `tolerance` and feeds it with the value read from configuration file.

Then, line #4 prints the *description* of variable `tolerance` (the phrase "allowed difference between time stamps") and line #5 prints the description of the corresponding *value* (in this case the word "secs").

3.1.2 Handling Antenna Exchange Format (ANTEX) files

Starting from GPS week #1400 (Nov 5th, 2006), the International GNSS Service (IGS) adopted the use of "*absolute*" antenna phase center values, dropping the "*relative*" values used so far ([Gendt, 2005]).

These values are now stored in ANTEX ([Rothacher and Schmid, 2006]) format files. The GPSTk provides the `AntexReader` class to parse these files, and the `Antenna` class to manage antenna data.

Then, these objects should be fed to others from processing classes that will take care of applying the corresponding corrections: `CorrectObservables` to manage receiver antenna corrections, and `ComputeSatPCenter` to handle satellite antenna corrections.

A typical way to use it follows:

```
AntexReader antexread;
antexread.open( "igs05.atx" );

ComputeSatPCenter svPcenter( nominalPos );
svPcenter.setAntexReader( antexReader );

Antenna rXAntenna;
rXAntenna = antexread.getAntenna("AOAD/M_T        NONE");

CorrectObservables corr;
corr.setAntenna( rXAntenna );
```

3.1.3 Computing tidal values

An important part of PPP modelling is the estimation of tidal effects caused by solid tides, ocean loading tides and pole movement-induced tides.

The GPSTk supplies several classes to manage tidal effects, providing the respective correction vectors in an unified format (class `Triple`). These vectors must then be fed to a `CorrectObservables` object to be added to the other corrections (like the aforementioned antenna phase center variations. See Section 3.1.2).

In the following code snippet, line #1 declares a time-handling object called `epoch` (from `DayTime` class) initialized at 22:00:00 hours of August 12th,

2008. Then, the tides-handling objects are declared in lines #2 through #4. Note that `OceanLoading` objects need to load ocean loading parameters files (provided by [OSO, 2009]), and that `PoleTides` objects need x and y pole displacement parameters, in arcseconds, supplied by IGS' Earth Rotation Parameters (ERP) files.

Then, line #5 computes a `Triple` which is a combination of the computed tidal values. The source code ends declaring a `CorrectObservables` object and feeding it with the total tidal correction.

```
1  DayTime epoch(2008, 08, 12, 22, 00, 0.0);

2  SolidTides solid;

3  OceanLoading ocean("OCEAN-GOT00.dat");

4  PoleTides pole(0.02094, 0.42728);

5  Triple tides = solid.getSolidTide(epoch, nominalPos) +
                  ocean.getOceanLoading("ONSA", epoch)  +
                  pole.getPoleTide(epoch, nominalPos)    );

6  CorrectObservables corr;
7  corr.setExtraBiases(tides);
```

For more information about the GPSTk implementation of these models, please refer to Section A.8.

3.1.4 GPSTk exception handling mechanism and its uses

In software as complex as GNSS data processing software it is unavoidable to find many situations that impair proper operation. Issues as invalid values, time desynchronization, singular matrices and many others are common, and should be adequately handled in running time.

In order to manage these events, the GPSTk provides a powerful and complete set of exception handling classes, built upon the native C++ exception mechanism.

This approach is convenient and flexible, and may be extended to include other situations that, although not being run-time errors, may benefit from the same approach.

Decimation in PPP is one of those situations. Given that IGS precise satellite orbits and clocks are typically provided each 900 s, while observations are given at 30 s intervals, it turns out that for accurate cycle slip detection it is convenient to process data at the highest possible rate, but that data must not be feed to the solver except when accurate orbits and clocks are available.

In this regard, it is convenient to add that the main problem is related with the availability of precise clocks matching the corresponding observation epochs: The interpolation of precise orbits yields an accurate result, but clock interpolation is not accurate enough. Some GNSS data processing centers like Center for Orbit Determination in Europe (CODE) now provide precise clocks with rates down to 30 s, but only final IGS products are used in this chapter.

Source code below shows how decimation is approached in the GPSTk. Line #1 declares a `Decimate` object configured to decimate data each 900 s, with a tolerance of 5 s, and according to values stored in the SP3 ephemeris handling object called `SP3EphList`.

Then, data is extracted from RINEX observation files with the typical `while` loop (between lines #2 and #17), but now the processing line #4 is enclosed in a `try - catch` block.

In this way, if data epoch is not a multiple of 900 s then object `decimateData` in line # 4 will issue an "*exception*" (or more properly, a `DecimateEpoch` exception), effectively halting further processing of line #4.

Such `DecimateEpoch` exception is then "*caught*" by the `catch` block in lines #6 to #8, that just tells the program to continue processing the next epoch. Decimation is so achieved in an effective, efficient and compact way.

Besides, if processing line #4 encounters any other GPSTk-defined problem, the `catch` block between lines #9 and #12 takes over, printing an error message and continuing with next epoch processing.

Finally, any other unrecognized exception is handled by block #13 to #16, issuing a different message and continuing processing.

```
Decimate decimateData( 900.0,
                       5.0,
                       SP3EphList.getInitialTime() );

while( rinexFile >> gpsData ) {

   try {

      gpsData >> ... >> decimateData >> ...

   }
   catch(DecimateEpoch& d) {
      continue;
   }
   catch(Exception& e) {
      cout << "Exception at epoch: "
           << epoch << "; " << e << endl;
      continue;
   }
   catch(...) {
      cout << "Unknown exception at epoch: "
           << epoch << endl;
      continue;
   }
}
```

3.2 PPP data processing

After explaining the basic accessory classes, we are ready to present the core PPP processing code. Several of these objects need initialization, but that part is omitted here. Again, please consult GPSTk examples and API. Table 3.1 summarizes names, classes they belong to, and purpose of the objects in the following source code lines:

```
gpsData >> requireObs >> linear1 >> markCSLI
        >> markCSMW >> markArc >> decimateData
        >> basicModel >> eclipsedSV >> grDelay
        >> svPcenter >> corr >> windup
        >> computeTropo >> linear2 >> pcFilter
        >> phaseAlign >> linear3 >> baseChange
        >> cDOP >> pppSolver;
```

The GDS processing data chain is a single C++ line, although in this case (for clarity sake) it spans seven physical lines. This line must be enclosed within a `while` loop to process all available epochs, and also within a `try - catch`

OBJECT	CLASS NAME	PURPOSE
requireObs	RequireObservables	Checks if required `TypeID`'s are present
linear1	ComputeLinear	Computes linear combinations used to detect cycle slips
markCSLI markCSMW	LICSDetector2 MWCSDetector	Detect and mark cycle slips using ionospheric (LI) and Melbourne-Wubbena combinations
markArc	SatArcMarker	Keeps track of satellite arcs
decimateData	Decimate	If not a multiple of 900 s, then decimates data
basicModel	BasicModel	Computes the basic components of a GNSS model
eclipsedSV	EclipsedSatFilter	Removes from GDS satellites in eclipse
grDelay	GravitationalDelay	Computes gravitational delay effect due to changing gravity field along SV-RX ray.
svPcenter	ComputeSatPCenter	Computes the effect of satellite antenna phase center
corr	CorrectObservables	Corrects observables from tides, antenna phase center, eccentricity, etc.
windup	ComputeWindUp	Computes phase wind-up correction
computeTropo	ComputeTropModel	Models delays due to troposphere
linear2	ComputeLinear	Computes ionosphere-free combinations for code (PC) and phase (LC)
pcFilter	SimpleFilter	Filters out spurious data in PC combination
phaseAlign	PhaseCodeAlignment	Aligns phase with code values, preserving the the integer nature of phase ambiguities
linear3	ComputeLinear	Computes code and phase prefilter residuals
baseChange	XYZ2NEU	Prepares GDS to use a North-East-Up reference frame in pppSolver
cDOP	ComputeDOP	Computes DOP values
pppSolver	SolverPPP	Solves the equation system with an Extended Kalman Filter (EKF) configured in PPP mode

Table 3.1: *PPP processing objects and classes.*

block to manage exceptions.

Particular mention deserves object `pppSolver`, belonging to `SolverPPP` class. This object is an Extended Kalman Filter (EKF) preconfigured to solve the PPP equation system in a way consistent with [Kouba and Heroux, 2001]: Coordinates are treated as constants (static), receiver clock is considered white noise, the residual vertical wet tropospheric delay is processed as a random walk stochastic model (using the Niell mapping functions), and carrier phase ambiguities are treated as white noise when cycle slips happen and as constants thereafter. All of these parameters are configurable.

3.2.1 Static PPP results

Figure 3.1 plots the results from this PPP processing code applied to station MADR, May 27th., 2008, using the default configuration for `SolverPPP` objects (PPP with static coordinates) and a NEU coordinate frame.

The a priori position used was the one provided by the IGS in Solution Independent Exchange (SINEX) files for that epoch. These results are consistent with what it is expected from this processing strategy, showing a small residual bias in the "*Up*" coordinate of about 17 millimeters, reaching errors below 5 cm in less than 1.5 h of processing.

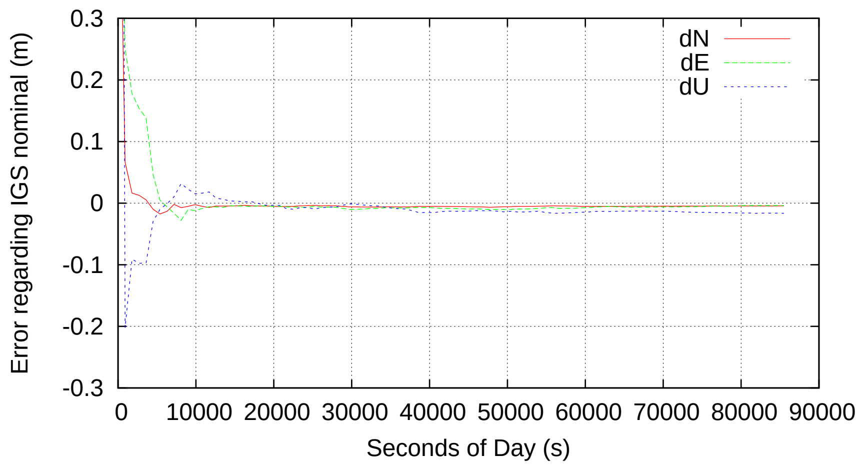


Figure 3.1: *Static PPP processing. MADR 2008/05/27.*

Figure 3.2 plots the 3D-positioning differences with respect to the IGS nominal position using several PPP processing tools provided by "*The Precise Point Positioning Software Centre*" (`http://gge.unb.ca/Resources/PPP`). This tool receives RINEX observation files and sends them to several on-line PPP processing facilities (using their corresponding standard configurations) such as:

- *CSRS-PPP (NRCAN)*:

  `http://www.geod.nrcan.gc.ca/online_data_e.php`

- *GPS Analysis and Positioning Software (GAPS)*:

  `http://gaps.gge.unb.ca/`

- *Automatic Precise Positioning Service (APPS)*, formerly *Auto-GIPSY*:

  `http://apps.gdgps.net/`

- *MagicGNSS (MAGIC)*:

  `http://magicgnss.gmv.com/ppp`

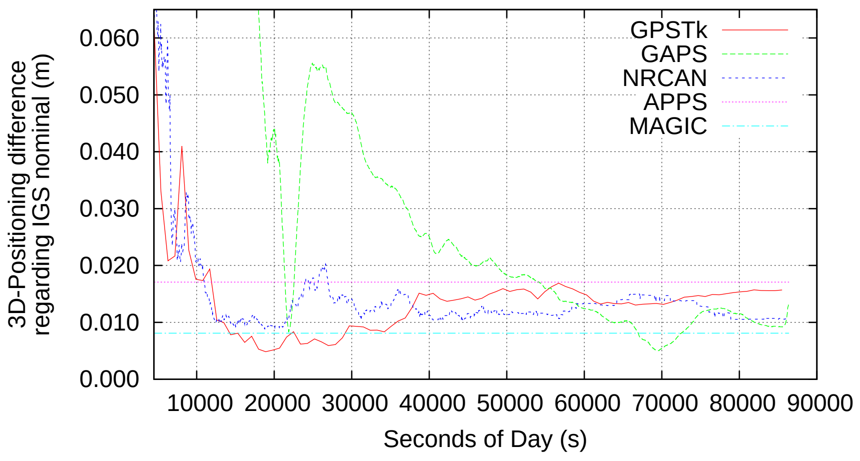


Figure 3.2: *Static PPP processing 3D errors. MADR 2008/05/27.*

This figure confirms that this relatively simple GPSTk-based PPP code compares both in precision and convergence time with other PPP processing tools (note that *APPS* and *MAGIC* work in static, forward-backward mode, providing only the last position solution).

3.2.2 Kinematic PPP results

The `pppSolver` has some preassigned stochastic models, but those models may be tuned and changed at will, given that they are objects inheriting from a general class called `StochasticModel`. This is a very important advantage of abstraction, and processing coordinates as white noise (kinematic mode) may be achieved in a very simple way:

```
1  WhiteNoiseModel newCoordinatesModel(100.0);

2  pppSolver.setCoordinatesModel(&newCoordinatesModel);
```

In line #1, a white noise stochastic model object (with a sigma of 100 meters) is declared, while in line #2 the `pppSolver` object is configured to use the new model for coordinate estimation. The vertical wet tropospheric effect is still treated as a random walk process, and the receiver clock continues as another white noise process (with a higher sigma).

Figure 3.3 presents the results, confirming the good quality of GPSTk model: the coordinates are consistently within 10 cm of the IGS values, with a 3D-RMS of 0.047 m for the convergence phase (from 1.5 h onwards).

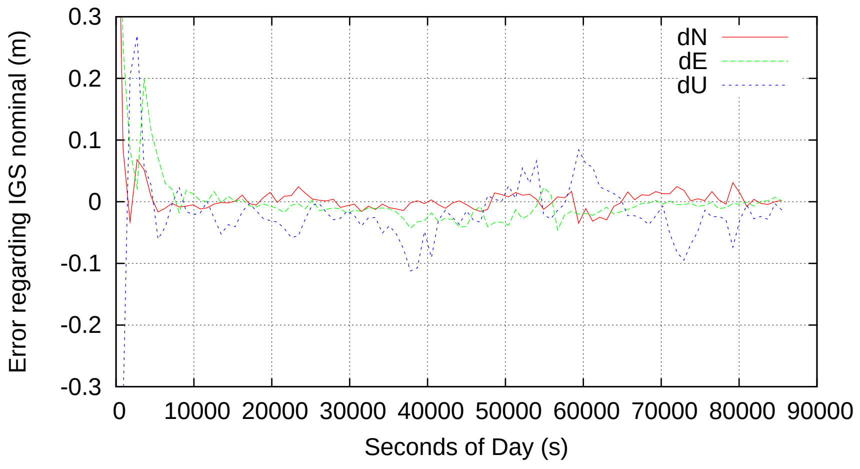


Figure 3.3: *Kinematic PPP processing. MADR 2008/05/27.*

The services of the "*The Precise Point Positioning Software Centre*" were used again, this time to compute the kinematic positioning[1]. Table 3.2 presents the 3D-RMS position difference with respect to the IGS SINEX position (for the convergence phase).

Another important characteristic of class `SolverPPP` is that the preassigned stochastic models for coordinates can be changed globally (like in the previous case), but they also may be adjusted *for each coordinate*. This can be done with the methods `setXCoordinatesModel()`, `setYCoordinatesModel()` and `setZCoordinatesModel()`.

Note that these methods work both in ECEF and NEU reference frames, depending how the `SolverPPP` was configured.

[1]*MagicGNSS* results are not shown because it provides only static solutions.

PPP positioning tool	3D-RMS (m)
APPS	0.034
GAPS	0.067
GPSTk	0.047
NRCAN	0.073

Table 3.2: *3D-RMS for Kinematic PPP position differences regarding IGS solution.*

The former capability may be very useful when dealing with kinematic positioning of surface vehicles: If some information about the vertical and horizontal velocities is known, then the corresponding stochastic models in the filter may be adjusted to match the expected vehicle behavior.

3.2.3 Forward-backward PPP results

The previous results were obtained with a Kalman filter that only runs forward. However, given that PPP is done in post-processing mode, an improved solution can be obtained running the filter in forward-backward mode, where ambiguity convergence achieved in a given forward run is used for the next backward run. It may be iterated at will.

An object of class `SolverPPPFB` is used for this. It encapsulates `SolverPPP` class functionality and adds a data management and storage layer to handle the whole process.

From the user's point of view, the main change is to replace the `SolverPPP` object (`pppSolver`) with a new `SolverPPPFB` object (`fbpppSolver`) inside the while loop that reads and processes the RINEX observation file.

After the first forward processing is done (and data is internally indexed and stored), it is simply a matter of telling the `fbpppSolver` object how many forward-backward cycles we want it to "re-process". For instance:

```
fbpppSolver.ReProcess(4);
```

After that, one last forwards processing is needed to get the time-indexed solutions out of `fbpppSolver`:

PPP positioning tool	3D-RMS (m)
APPS	0.0018
GPSTk	0.0048
MAGIC	0.0052
NRCAN	0.0069

Table 3.3: *RMS for zpd differences regarding IGS combined solution.*

```
while( fbpppSolver.LastProcess(gpsData) )
{
   cout << fbpppSolver.getSolution(TypeID::dLat) << ''  '';
   cout << fbpppSolver.getSolution(TypeID::dLon) << ''  '';
   cout << fbpppSolver.getSolution(TypeID::dH)   << endl;
}
```

In this case, the forward-backward processing (in static mode) is used to compute the zenith path delay estimation (zpd) for a full day. Figure 3.4 shows the results for APPS, MAGIC, GPSTk and NRCAN, as well as the official, combined IGS zpd. NRCAN only provides forward estimates, and GAPS does not provide zpd estimations. Table 3.3 presents the RMS of the zpd differences with respect to IGS values.

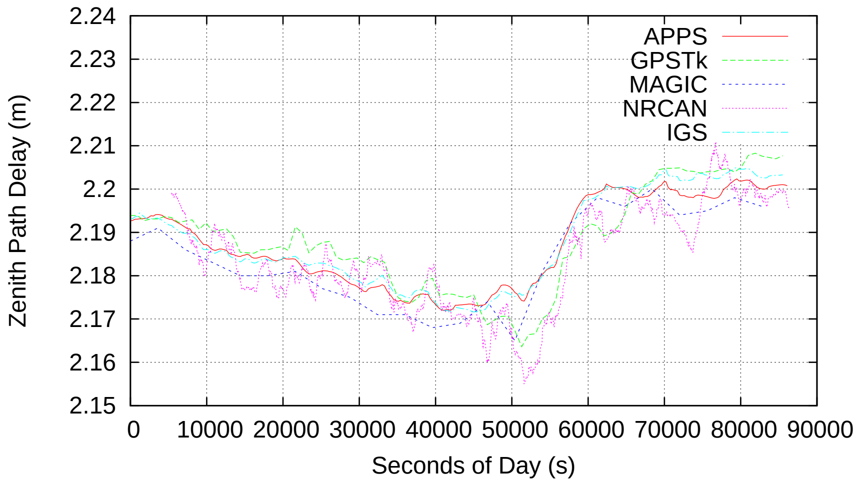


Figure 3.4: *Zenith path delay for several PPP processing tools. MADR 2008/05/27.*

3.3 Carrier phase-based DGPS

The techniques presented in Sections 2.5.5 and 2.5.6 for pseudorange-based DGPS may be combined with the ones for PPP, and with relatively minor modifications implement carrier phase-based DGPS processing with broadcast orbits and clocks:

```
1  Synchronize synchro( rinexRefFile, gpsData );

2  while( rinexFile >> gpsData )
   {

3     gpsDataRef >> synchro >> requireObs >> linear1
                 >> markCSLIRef >> markCSMWRef
                 >> markArcRef >> refBasicModel
                 >> eclipsedSV >> refGravDelay
                 >> refSVPcenter >> refCorr >> refWindup
                 >> refComputeTropo >> linear2
                 >> pcFilter >> linear3;

4     delta.setRefData(gpsDataRef.body);

5     gpsData >> requireObs >> linear1 >> markCSLI
              >> markCSMW >> markArc >> decimateData
              >> basicModel >> eclipsedSV >> grDelay
              >> svPcenter >> corr >> windup
              >> computeTropo >> linear2 >> pcFilter
              >> phaseAlign >> linear3 >> delta
              >> baseChange >> cDOP >> pppSolver;

6     cout << pppSolver.getSolution(TypeID::dLat) << '' '';
7     cout << pppSolver.getSolution(TypeID::dLon) << '' '';
8     cout << pppSolver.getSolution(TypeID::dH) << endl;
   }
```

As can be seen, three lines represent the core of code to achieve this type of GNSS data processing. Line #3 is in charge of processing reference station data, and line #4 sets the `delta` object. Please remember from Section 2.5.5 that `delta` objects belongs to class `DeltaOp`, and it is in charge of subtracting `gpsDataRef` prefilter residuals from the corresponding `gpsData` residuals, deleting (by default) satellites not common to both receivers. Finally, line #5 processes rover data.

Remember that object `synchro` synchronizes data between receivers (as in the pseudorange-based DGPS case). Also, take notice of the fact that some objects are shared between processing chains while others must be used for a given receiver only. This is partly because of initialization issues (for instance, some objects need the nominal position of a specific receiver), and partly because objects like cycle slip detectors are state-aware, and thence must not be shared between different data processing streams.

Line #5 is very similar to the main PPP example (Section 3.2), with the important addition of object `delta` just after computing prefilter residuals (`linear3` object), taking care of computing single differences.

Please note that the former `decimateData` object is no longer needed: Broadcast orbits and clocks are now used, and therefore the processing works at arbitrary sampling rates.

Figure 3.5 presents the errors regarding IGS nominal position for this carrier phase-based differential processing with static coordinates and floated phase ambiguities for EBRE station as rover and BELL station as reference, 2002/01/30 (baseline 115 km).

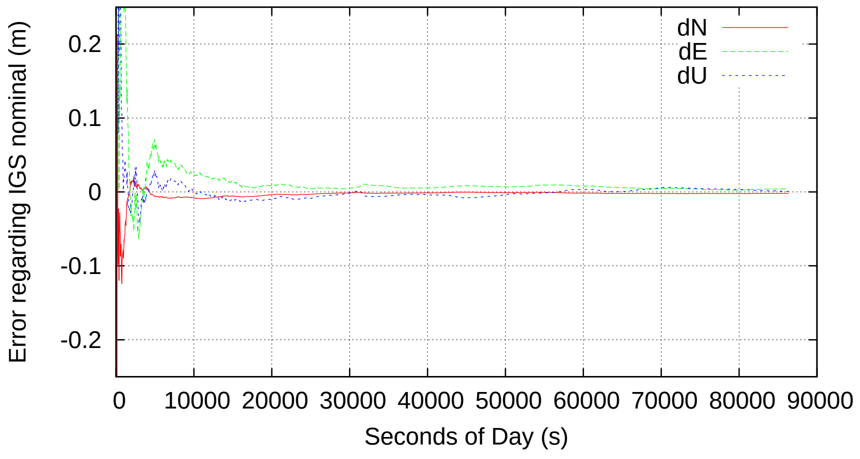


Figure 3.5: *Static phase-based DGPS errors regarding IGS nominal position. EBRE 2002/01/30. Phase ambiguities are floated.*

This code is fully implemented in `example10.cpp` of the GPSTk. Be aware that better results could be obtained using RTK techniques (not covered here), but usually those techniques are limited to baselines shorter than 20 km.

3.4 Abstraction and flexibility

Data abstraction is a very important part of what the GDS processing paradigm uses to get code that is both powerful and easy to read. Also, a good deal of its flexibility lies upon the abstraction concept. Some examples follow.

3.4.1 `ProcessingVector` and `ProcessingList`

As was mentioned before, all processing classes inherit from a common, *pure virtual* class called `ProcessingClass`.

Due to this design feature it is possible to write other classes that work with processing classes in an unified way. For instance, the standard STL template class named `std::vector` was modified to create a "*vector*" of processing classes[2].

This feature, although it may seem strange at first, allows writing very flexible and compact code. For example, the `ProcessingVector` class and its "`push_back()`" method may be used to store the PPP processing line presented in Section 3.2:

```
ProcessingVector pVector;

pVector.push_back(requireObs);
pVector.push_back(linear1);

   // . . . store the other PPP processing classes here . . .

pVector.push_back(baseChange);
pVector.push_back(cDop);
pVector.push_back(pppSolver);
```

After that, all the PPP data processing can be expressed in a very compact way:

```
while(  rinexFile >> gpsData )
{
   gpsData >> pVector;
}
```

This way of code management is not only compact, but also very flexible because it may change dynamically during run time. Moreover, given that reporting the name of their class is a requirement that all processing classes must fulfill, then things like the following are easy to implement:

[2]A `std::list`-based version, `ProcessingList`, is also available.

```
ProcessingVector pVector;

   // ... fill pVector with processing classes ...

ProcessingVector pVectorNew;

for( int i = 0; i < pVector.size(); ++i )
{
   if( filtersDisabled &&
        pVector[i].getClassName() == "SimpleFilter")
   {
      continue;
   }
   else
   {
      pVectorNew.push_back( pVector[i] );
   }
}

pVector =  pVectorNew;
```

The code snippet above is a simple yet effective way to dynamically modify the (previously) defined processing data chain according to some condition that should be met (`filtersDisabled` flag is set to `TRUE`), removing all processing objects belonging to class "`SimpleFilter`". Of course, the former can be achieved in other ways, but using the GDS-provided classes is usually very efficient (they are STL-based) and less error-prone, potentially saving considerable time.

Indeed, the `ProcessingVector` class is in turn a `ProcessingClass`, so it is possible to build "*vectors*" of "*processing vectors*", if such a construct were needed.

3.4.2 Abstraction of equation solvers

The data abstraction approach can be extended to several areas of GNSS data processing, and a class called `SolverGeneral` takes this concept far into the "*solver*" realm.

The `SolverGeneral` class is an Extended Kalman Filter (EKF) implementation that relies on another class named `EquationSystem`. As mentioned in Section 2.2.1.3, the equation systems may be modeled just as any other data structure.

Therefore, `EquationSystems` objects are composed of `Equations`, and the former are a set of rules in charge of generating all the vectors and matrices that `SolverGeneral` will internally need to compute the GNSS solution.

In turn, `Equations` are composed of a set of `Variable` objects. The later encapsulate information such as their `TypeID`, the `SourceID`'s and `SatID`'s they are applicable to, and the stochastic models associated with them.

With this design, the GNSS researcher will be able of establishing a set of rules to "*tune*" the solver to solve a given problem: He just needs to redefine the variables and equations. Complex multi-station and/or hybrid GNSS-INS problems can be tackled in this way with relatively few code lines, encouraging code reusability.

This subject will be further developed in Chapter 4.

3.5 Summary

In this chapter some applications of the GDS to carrier phase-based GNSS data processing strategies were presented, as well as important accessory classes that ease tackling these complex tasks. Reference implementations of these strategies are provided for the GNSS community in the GPSTk `examples` directory, as files `example8.cpp`, `example9.cpp` and `example10.cpp`.

When comparing the performance of these relatively simple GDS-based source code examples with other state-of-the art PPP suites, it was demonstrated that their results are among the best, confirming the validity of using the GPSTk combined with the GDS to get easy to write and maintain, yet powerful, GNSS data processing software.

Furthermore, given that the GDS design is based on data abstraction, it allows a very flexible handling of concepts beyond mere data encapsulation, including programmable general solvers, among others.

The application of GNSS Data Structures (GDS) to carrier phase-based data processing led to three publications in congress proceedings: [Salazar et al., 2008b], [Salazar et al., 2008a], and [Salazar et al., 2009b].

It also represents an important part of the paper published at the GPS Solutions journal ([Salazar et al., 2009a]).

Chapter 4

Precise Orbits Positioning

Former Section 3.4.2 briefly presented the advantages of GDS data abstraction regarding solvers, in particular, the possibility of establishing a set of rules to "*tune*" a solver to solve a given problem.

In this chapter, those advantages will be used to implement a kinematic PPP-like processing based on a network of stations, where satellite clock offsets will be estimated on-the-fly. This procedure is independent of precise clock information and only needs precise orbits to work; therefore it will be called *Precise Orbits Positioning (POP)*.

4.1 Background

Kinematic positioning using GPS is an important research line, and in particular airborne kinematic GPS positioning is a tough problem with an extense literature ([Castleden et al., 2004], [Mostafa, 2005], and [Zhang and Forsberg, 2007], to cite only a few). Different techniques have been applied to this problem, ranging from pseudorange-based DGPS to carrier phase-based techniques such as RTK, network-based RTK, and PPP, among others.

Among the differential techniques, RTK typically yields the best accuracy (at the centimeter level, when ambiguities are fixed), but it needs reference stations near the operation area (closer than 20 km for adequate performance), while pseudorange-based DGPS operates well with reference stations more than 100 km away, at the expense of decreased accuracy (at the meter level). Network-based RTK techniques like Virtual Reference Station (VRS) fill an intermediate niche with ambiguity fixing-level accuracy at about 50 km range from nearest reference station.

On the other hand, PPP is a standalone strategy, avoiding the expense and logistics of ad hoc reference stations[1]. However, it has the limiting factor that solution rate is set by the availability of precise satellite clock offsets, given that precise orbits can be interpolated without losing accuracy, whereas satellite clock offsets can not.

The former has been a recurrent problem when applying PPP techniques to kinematic positioning. Nevertheless, relatively recent developments have allowed data processing centers such as CODE to generate precise satellite clock corrections with higher data rates (typically 30 s, and more recently down to 5 s), using techniques consisting on phase-consistent interpolation of precise 5-minute clock results ([Hugentobler et al., 2006]).

However, in this chapter a completely different approach will be carried out: satellite clock offsets will be estimated *on-the-fly*. This procedure is independent of precise clock corrections and, therefore, it can achieve an arbitrary positioning rate (only limited by observation data rate), opening a window of opportunity to very interesting precise positioning techniques.

In order to achieve this in an efficient and reusable way, this chapter relies on the facilities provided by the GPSTk, and in particular on the *GNSS Data Structures* data processing paradigm (see Chapter 2).

The author of this dissertation considers that the use of the open source GPSTk-provided tools represents an important advantage for researchers, because in this way they will have a reference implementation available to test and experiment with. Therefore, he developed a reference implementation that is located at the `examples` directory of the GPSTk development repository, named "`example14.cpp`".

4.2 POP description

The POP procedure starts with selecting a set of reference stations and setting one of them as the *MASTER* station. *Master*'s clock will be set as the reference for the network, so all the other clocks will be computed with respect to it. The other unknowns for the *master* station will be the zenith tropospheric delay and the ambiguities.

Therefore, the corresponding equations for pseudorange and phase are:

$$PrefitPC_0^j = tmap_0^j.ztd_0 - c.dt^j \tag{4.1}$$

[1]Of course, PPP needs precise ephemeris products generated by an extense network of IGS reference stations, but they are already in place.

$$PrefitLC_0^j = tmap_0^j.ztd_0 + Bc_0^j - c.dt^j \tag{4.2}$$

Where:

- $PrefitPC_0^j$ and $PrefitLC_0^j$: These values are, respectively, the prefilter residual (observation minus modeled effects) of ionosphere-free pseudorange and phase combinations for satellite SV^j and master station 0.
- $tmap_0^j$: Tropospheric mapping function (Niell).
- ztd_0 : Zenith tropospheric path delay.
- $c.dt^j$: Relative clock delay between satellite SV^j and master station 0, in meters.
- Bc_0^j : Ionosphere-free carrier phase ambiguity.

The other "*reference*" stations will have similar equations, but adding their clock offsets (with respect to *master* clock) as an additional unknown. Hence,

$$PrefitPC_k^j = tmap_k^j.ztd_k + c.dt_k - c.dt^j \tag{4.3}$$

$$PrefitLC_k^j = tmap_k^j.ztd_k + Bc_k^j + c.dt_k - c.dt^j \tag{4.4}$$

where $c.dt_k$ is the relative clock delay between reference station k and master (in meters).

Finally, the "*rover*" receiver will have an equation similar to the standard PPP process, but adding the estimation of satellite clock offsets:

$$\begin{aligned} PrefitPC_r^j &= \Big(\frac{x_{r0} - x^j}{\rho_{r0}^j}\Big)dx + \Big(\frac{y_{r0} - y^j}{\rho_{r0}^j}\Big)dy + \Big(\frac{z_{r0} - z^j}{\rho_{r0}^j}\Big)dz \\ &\quad +tmap_r^j.ztd_r + c.dt_r - c.dt^j \end{aligned} \tag{4.5}$$

$$\begin{aligned} PrefitLC_r^j &= \Big(\frac{x_{r0} - x^j}{\rho_{r0}^j}\Big)dx + \Big(\frac{y_{r0} - y^j}{\rho_{r0}^j}\Big)dy + \Big(\frac{z_{r0} - z^j}{\rho_{r0}^j}\Big)dz \\ &\quad +tmap_r^j.ztd_r + Bc_r^j + c.dt_r - c.dt^j \end{aligned} \tag{4.6}$$

where (x_0, y_0, z_0) is the a priori *rover* receiver position, (x^j, y^j, z^j) is the position of satellite SV^j, and parameters (dx, dy, dz) are the corrections to (x_0, y_0, z_0).

It can be seen that the connection between receivers is achieved by the simultaneous estimation of satellite clock offsets. As said, this procedure allows rover precise positioning without precise satellite clock products.

Although the observations are not explicitly differentiated, the systems of Equations 4.1 to 4.6 is equivalent to a carrier phase-based differential DGPS system using the ionosphere-free combination of observations. The simultaneous estimation of the satellite clock offsets allow them to become the ligatures between the equations.

4.3 POP implementation

Please note that implementation of an equation system for Equations 4.1 to 4.6 is a complex task. This system involves multiple stations separated hundreds of kilometers and there are a great number of unknowns of several kinds: Some unknowns are *receiver-indexed* (or *receiver-specific*, like. ztd_i, dx, dy, etc.), some are *satellite-indexed* (dt^j), and others are both *receiver-* and *satellite-indexed*, like Bc_i^j. Therefore, the number of unknowns at a given epoch has a wide variation depending on the available station data and the number of visible satellites.

The GPSTk provides a class, `SolverGeneral`, to help implementing this kind of systems. The idea behind `SolverGeneral` is that equations and variables are *described* (as opposed to being hard coded in the software), indicating their stochastic models and relationships.

Then, at each epoch the `SolverGeneral` object will match the incoming data (observations and ephemeris) with the equations and variables descriptions, building the appropriate equation system for that epoch.

Implementation starts with declaration and initialization of the `Variable` objects to be used, as well as their associated stochastic models:

```
 1  WhiteNoiseModel coordinatesModel( 100.0 );
 2  TropoRandomWalkModel tropoModel;
 3  PhaseAmbiguityModel ambiModel;

 4  Variable dLat( TypeID::dLat, &coordinatesModel,
                   true, false, 100.0 );
 5  Variable dLon( TypeID::dLon, &coordinatesModel,
                   true, false, 100.0 );
 6  Variable dH( TypeID::dH,&coordinatesModel,true,false,100.0 );

 7  Variable cdt( TypeID::cdt );
       cdt.setDefaultForced(true);   // Force coefficient (1.0)

 8  Variable tropo( TypeID::wetMap,&tropModel,true,false,10.0 );

 9  Variable ambi( TypeID::BLC, &ambiModel, true, true );
       ambi.setDefaultForced(true);   // Force coefficient

10  Variable satClock( TypeID::dtSat, false, true );
       satClock.setDefaultCoefficient(-1.0); // Set coefficient
       satClock.setDefaultForced(true);      // Force coefficient

11  Variable prefitPC( TypeID::prefitC );
12  Variable prefitLC( TypeID::prefitL );
```

In the former code, lines #1 to #3 set the stochastic models to be used. Line #4 declares a `Variable` called `dLat`, of `TypeID` "`dLat`", with a white noise stochastic model (kinematic positioning). The first "`true`" parameter indicates that this Variable is "*source-indexed*" (i.e., it is a distinct variable for each `SourceID`, i.e., receiver), and the following "`false`" parameter tells that it is *not* "*satellite-indexed*", meaning that the same variable will be used for all visible satellites. The final numeric value (100.0) sets the initial sigma. Variables `dLon` and `dH` (lines #5 and #6) follow the same pattern.

Line #7 declares `cdt`, the `Variable` representing receiver clock offsets. The defaults are used (white noise model, source-indexed, not satellite indexed, big preset sigma), and it is forced to always use the value "1.0" as coefficient (by default, coefficients are looked for inside the GDS).

Declaration of variables `tropo`, `ambi` (ambiguities), and `satClock` (SV clock offsets) are similar, with the exception that ambiguities are *source-* and *satellite-indexed*, whereas satellite clocks are only *satellite-indexed*. The last couple of lines (#11, #12) declare default, dummy "variables" representing the independent terms of equations, `prefitPC` and `prefitLC`.

Again, it is important to emphasize that in the former procedure the variables *characteristics* were *described*, instead of *declaring* a variable for each possible receiver-satellite combination.

Once the `Variable`s are properly declared and initialized, it is the turn of describing the `Equation` objects. First, let's declare the equations for *master* station:

```
 1  Equation equPCMaster( prefitPC );

 2  equPCMaster.addVariable( tropo );
 3  equPCMaster.addVariable( satClock );
 4  equPCMaster.header.equationSource = master;

 5  Equation equLCMaster( prefitLC );

 6  equLCMaster.addVariable( tropo );
 7  equLCMaster.addVariable( satClock );
 8  equLCMaster.addVariable( ambi );
 9  equLCMaster.header.equationSource = master;

10  equLCMaster.setWeight( 10000.0 );
```

Line #1 declares the `Equation` object for pseudorange, setting the independent term type. Then, lines #2 and #3 add the variables to the equation and finally line #4 sets what receiver (data source) this equation applies to: `master` is an object of class `SourceID` holding the information corresponding to the *master* station.

Declaration of the equation for carrier phase is very similar, except for line #8, that adds an additional variable (`ambi`), and line #10 that sets the *relative weight* of this equation: the carrier phase sigma is 100 times smaller, so the associated weight is 100*100 times larger.

Equations for reference stations and rover receiver are declared in the same way. However, it must be noted that reference stations form a `SourceID` set, instead of a single station, so they need an additional treatment. Thus `equPCRef` and `equLCRef` are the equations for the reference stations' pseudorange and carrier phase, respectively:

```
1   equPCRef.header.equationSource = Variable::someSources;

2   equLCRef.header.equationSource = Variable::someSources;

3   for( std::set<SourceID>::const_iterator
         itSet = refStationSet.begin();
         itSet != refStationSet.end();
         ++itSet )
4   {
5      equPCRef.addSource2Set( (*itSet) );
6      equLCRef.addSource2Set( (*itSet) );
7   }
```

The special SourceID called "Variable::someSources" indicates that equations equPCRef and equLCRef will apply to more than one data source. Thus, it is necessary to add those data sources to each equation's internal set. The "for" loop spanning from line #3 to line #7 achieves this in a general, reusable way.

Finally, once all the Equation objects, and their corresponding Variables, have been described, they are added to an EquationSystem, which in turn feeds a SolverGeneral object:

```
EquationSystem equSystem;

equSystem.addEquation( equPCRover );
equSystem.addEquation( equLCRover );
equSystem.addEquation( equPCRef );
equSystem.addEquation( equLCRef );
equSystem.addEquation( equPCMaster );
equSystem.addEquation( equLCMaster );

SolverGeneral solver( equSystem );
```

From now on, object solver is an Extended Kalman Filter configured to solve the defined equation system (equSystem), building its internal matrices and vectors automatically according to the incoming data. It just needs to be fed with the appropriate GDS.

As previously said, program "example14.cpp" is a reference implementation of the POP algorithm, and it is freely available as open source software in the development version of the GPSTk at the examples directory. Please refer to the GPSTk website (http://www.gpstk.org) for details about downloading and installing the development version.

4.4 POP data processing

The approach to this multi-station problem is to pre-process all the stations, one by one, in a way similar to the one explained in Section 3.2 (PPP), but without applying the solver object.

The results from this preprocessing are stored in an appropriate multi-epoch, multi-station GNSS data structure that automatically takes care of all indexing (structure `gnssDataMap` is used for this). Then, an epoch-worth of data is extracted each time from the `gnssDataMap` GDS and fed to solver, and the results are printed.

For this experiment, 5 IGS stations were used: ACOR, MADR, SCOA, SFER and TLSE, forming a network across Iberian Peninsula spanning 1023 km (SFER-TLSE). Station ACOR was set as the "*master*", while MADR was the "*rover*", 392 km away from nearest reference station (SCOA). This network comprises more than 580,000 km^2 and can be seen in Figure 4.1.

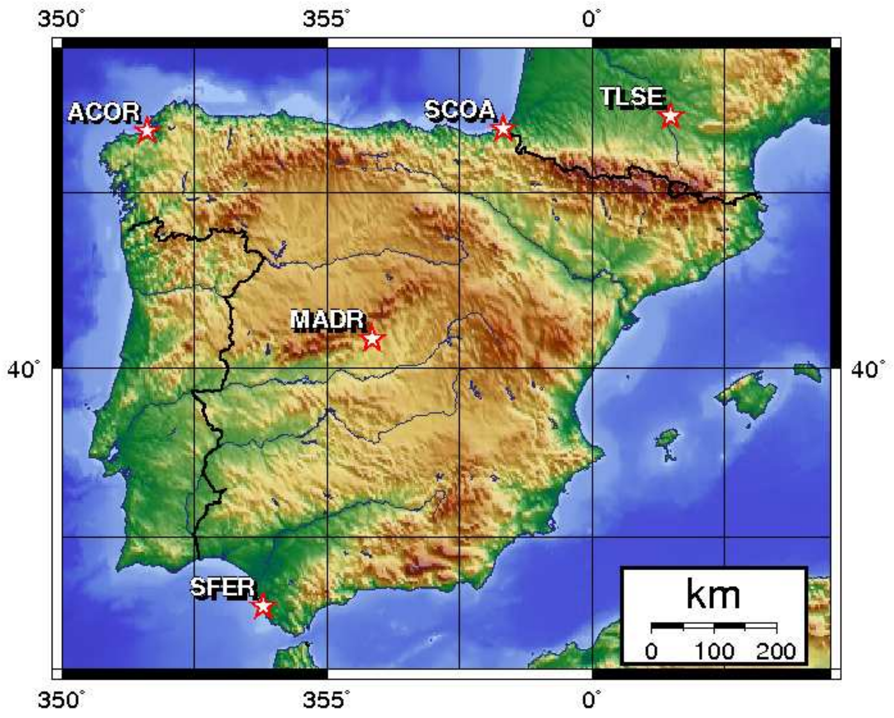


Figure 4.1: *POP network. MADR 2008/05/27.*

Standard IGS products (precise orbits and satellite clocks) with a 900 s data rate were used, but the data was processed at 30 s, the rate given by the RINEX observation files. Note again that in this case the IGS satellite clocks were not interpolated, but ignored: The SV clocks used for this POP positioning were estimated on-the-fly.

Figure 4.2 shows the good results from this approach, presenting both the 3D-error in position (with respect to the known IGS position) of POP, and the 3D-error for the standard kinematic PPP processing (see Section 3.2.2).

The results are very similar, as was expected: a 3D-RMS of 0.046 m for the kinematic PPP case versus a 3D-RMS of 0.049 m for the POP case (from 2 h onwards), but POP yields a higher positioning rate.

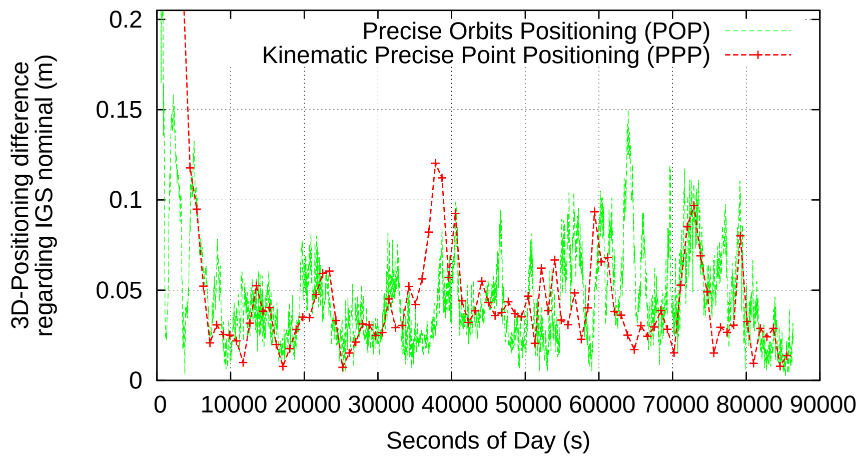


Figure 4.2: *POP versus kinematic PPP processing. MADR 2008/05/27.*

As previously said, although the observations are not explicitly differentiated, the POP procedure is equivalent to a carrier phase-based differential system using the ionosphere-free combination of observations. However, it is a network-base processing and this provides additional robustness to the results, even when using long baselines and for receivers outside the network area.

Take, for instance, the network shown in Figure 4.1 but with TLSE station as "*rover*" and station MADR as just another reference station. In this case, TLSE will be outside the network area and 257 km away from nearest reference station (SCOA).

The 3D-position error from this new processing is shown in Figure 4.3, and it can be seen that in this case the POP solution behaves better between epochs 35000 s and 50000 s, when some problem is affecting the kinematic PPP solution[2]. 3D-RMS values (from 2 h onwards) are 0.069 m for PPP and 0.044 m for POP.

[2]At this epoch, TLSE receiver suffered from the sudden lost and posterior gain of 2 satellites.

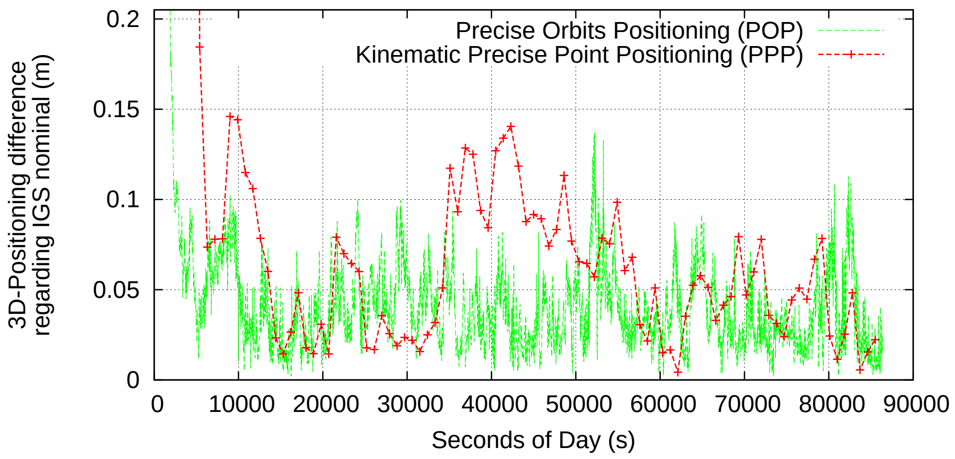


Figure 4.3: *POP versus kinematic PPP processing. TLSE 2008/05/27.*

Also, distance from "*rover*" to nearest reference station does not seem to be a critical factor. If station SCOA is taken out from Figure 4.1 leaving a 4 station network (including "*rover*") with TLSE still as "*rover*" and station MADR as nearest reference station (588 km away), the results are not significantly degraded as Figure 4.4 shows: In this case, the POP 3D-RMS values (from epoch 7200 s on) barely increases from 0.044 m to 0.049 m.

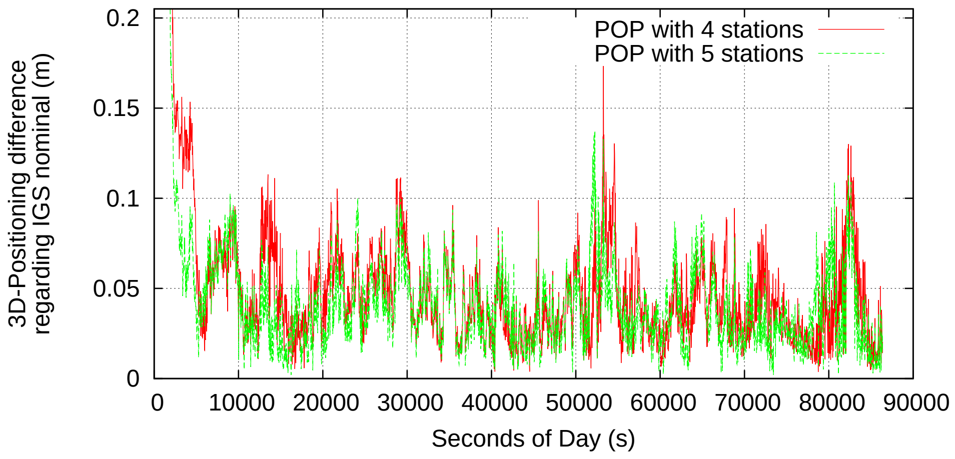


Figure 4.4: *POP results for 4 and 5-stations networks. TLSE 2008/05/27.*

The important aspect here is that the satellites being used should be in view from as many network stations as possible, because that will provide better on-the-fly estimations of the satellite clock offsets. When using the POP strategy with only two stations (MADR as "*master*" and TLSE station as "*rover*"), the data processing effectively becomes the aforementioned carrier phase-based DGPS with a 588 km-long baseline. With such a long baseline the results will degrade, given that the estimations of satellite clocks will not be as accurate, and there will be satellites that are not common for both stations.

Figure 4.5 illustrates this case. The POP 3D-RMS values (from 2 h on) for the 2 station processing raises to 0.061 m (compared with 0.049 m of the 4 station case).

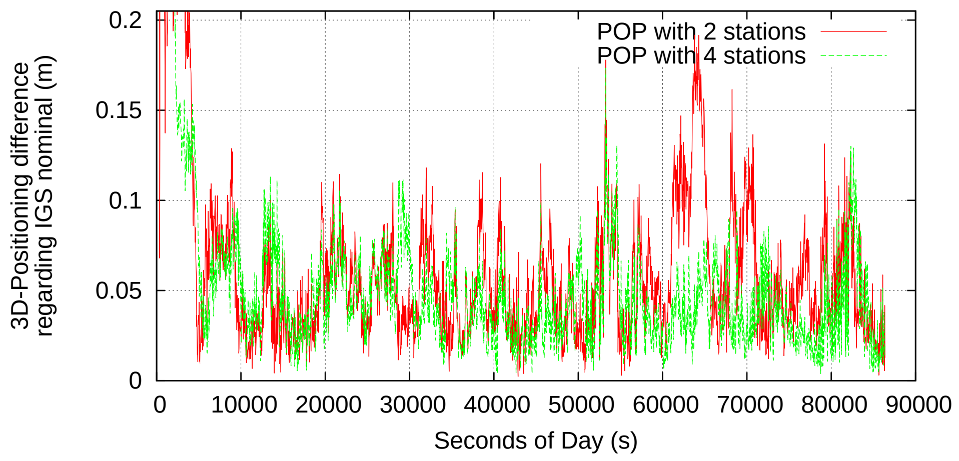


Figure 4.5: *POP results for 2 and 4-stations networks. TLSE 2008/05/27.*

4.5 POP convergence time

Regarding convergence time, Figure 4.6 plots the resulting 3D-RMS of error as function of the epoch since it is computed and the data processing strategy used. The values shown correspond to starting computing the 3D-RMS of error from 1800 s, 3600 s, 5400 s and 7200 s (i.e., 30 min, 1 h, 1 h:30 min and 2 h).

Convergence accelerates as the number of station increases, but up to some point, and the same can be said about the improvements in the 3D-RMS error figure, suggesting that the improvements achieved by having more observations available reach a limit shortly after 5 or 6 stations for a network of this size (this aspect should be further researched in the future). The results for a standard PPP processing with IGS products are shown for reference purposes.

The convergence time poses a problem when applying the POP to moving vehicles, specially if data arcs are short. Strategies to reduce convergence time should be a topic of future research in order to extend the usefullness of this data processing strategy.

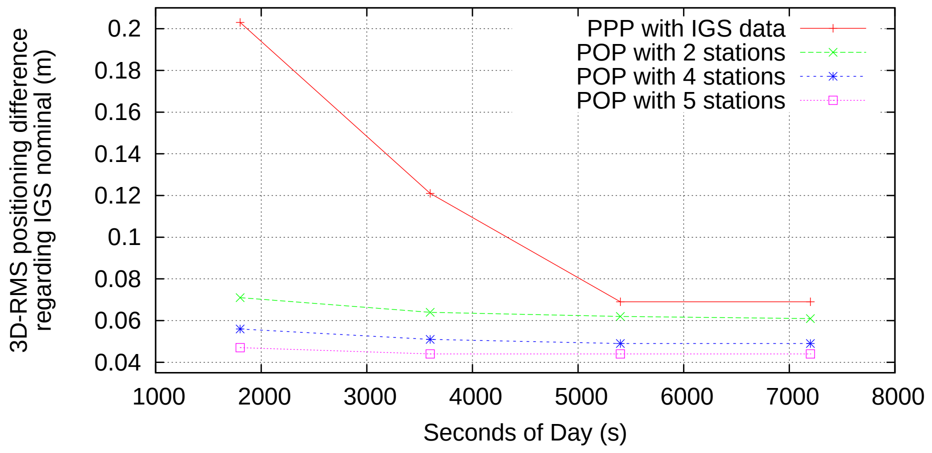


Figure 4.6: *Convergence time. TLSE 2008/05/27.*

4.6 Summary

In this chapter, the advantages of GDS data abstraction regarding solvers, and in particular the possibility to set up a "general solver" object, has been used to implement a kinematic PPP-like processing based on a network of stations. This procedure was named *Precise Orbits Positioning (POP)* because it is independent of precise clock information and it only needs precise orbits to work.

This procedure involved multiple stations separated hundreds of kilometers and there are a great number of unknowns of several kinds: Some unknowns are *receiver-indexed* (or *receiver-specific*, like ztd_i, dx, dy, etc.), some are *satellite-indexed* (dt^j), and others are both *receiver-* and *satellite-indexed*, like Bc_i^j. Therefore, the number of unknowns at a given epoch has a wide variation depending on the available station data and the number of visible satellites. The GPSTk-provided class `SolverGeneral` helps implement this kind of systems, *describing* (rather than hard coding the procedure in software), the equations, variables, and their associated stochastic models and relationships. Besides, the program `example14.cpp` is provided as reference implementation in the `examples` directory of the GPSTk.

The results from this approach were very similar (as expected) to the standard kinematic PPP processing (see Section 3.2.2) strategy, but yielding a higher positioning rate. Also, the network-based processing of POP seems to provide additional robustness to the results, even for receivers outside the network area. The distance from "*rover*" to nearest reference station does not seem to be a very critical factor, because in our test cases the results are not significantly degraded when this distance nearly doubled.

The convergence time improves in POP as the number of station in the network increases, but up to a limit. This issue still poses a problem when applying the POP method to moving vehicles, specially if data arcs are short.

These results have shown how the GPSTk-provided GDS, with their associated paradigm, allows one to develop code that is simple to read and maintain, but able to carry out complex GNSS data processing in an effective way. This work represented the main part of a paper published at the GPS Solutions journal ([Salazar et al., 2009a]).

Chapter 5

Velocity and acceleration determination

The former chapter presented the POP method to obtain, in post-process, the precise position of a vehicle using a wide network of reference receivers and precise orbits information.

This chapter extends the previous work to the precise estimation of velocity and acceleration. Taking as starting point a known carrier phase-based acceleration estimation method, several improvements are suggested and implemented, and the range of the previous method is greatly extended.

5.1 Background

GNSS-based velocity and acceleration determination can be obtained with several methods. A common method is by time-differentiating successive position solutions of the moving vehicle. However, this approach has several disadvantages, like: The velocity and acceleration precision are strongly dependent on position accuracy and the gain or loss of a satellite can introduce discontinuities ([Bruton, 2000]).

Another common approach is to use the Doppler observable, when available (see, for instance, [Parkinson and Spilker Jr., 1996]). The problem with this method is that the raw Doppler observable can be much noisier than the Doppler value obtained by deriving the carrier phase observable (see [Cannon et al., 1997], [Szarmes et al., 1997], [Hofmann-Wellenhof et al., 2008]).

A different method was proposed by [van Graas and Soloviev, 2004], where single differences between consecutive *epochs* of carrier phase observables are used. That paper reported, in static mode, standard deviations of velocity noise of

7.9 mm/s for the Up component, and 2.2 mm/s and 3.1 mm/s for the East and North components. Also, for a DC-3 aircraft test flight with low dynamics the standard deviations with respect to a position-based, DGPS-computed reference solution were 9.7 mm/s, 2.6 mm/s and 3.7 mm/s in the Up, East and North components, respectively.

A fourth approach, related to the former and the one to be followed in this work, is to use the carrier phase as observable and to numerically derivate it to get both range rate and range acceleration. This method, originally focused on acceleration estimation for airborne gravimetry purposes, was presented in [Jekeli, 1994] and [Jekeli and Garcia, 1997], and later expanded by [Kennedy, S., 2002b].

5.2 Carrier phase method fundamentals

The paper by [Jekeli and Garcia, 1997] implemented the carrier phase method using the measurements from only four satellites. This method was later expanded by [Kennedy, S., 2002a] and [Kennedy, S., 2002b] to incorporate all available measurements, adding a covariance model to weight them.

5.2.1 Velocity determination

The explanation of [Kennedy, S., 2002b] method will start with the geometry set up in Figure 5.1. From that figure it can be seen that:

$$\mathbf{x}_m^p = \rho_m^p \mathbf{e}_m^p \tag{5.1}$$

Where ρ_m^p is the geometric distance between SV^p and RX_m antenna phase centers, and $\mathbf{e}_m^p$ is the unit vector in the RX_m-SV^p direction. Satellite SV^p will be our *reference satellite*.

Another equation closely related to Equation 5.1 is:

$$\rho_m^p = \mathbf{e}_m^p \cdot \mathbf{x}_m^p \tag{5.2}$$

Differentiating Equation 5.2 yields:

$$\dot{\rho}_m^p = \dot{\mathbf{e}}_m^p \cdot \mathbf{x}_m^p + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m^p \tag{5.3}$$

If we substitute Equation 5.1 into Equation 5.3, the later becomes:

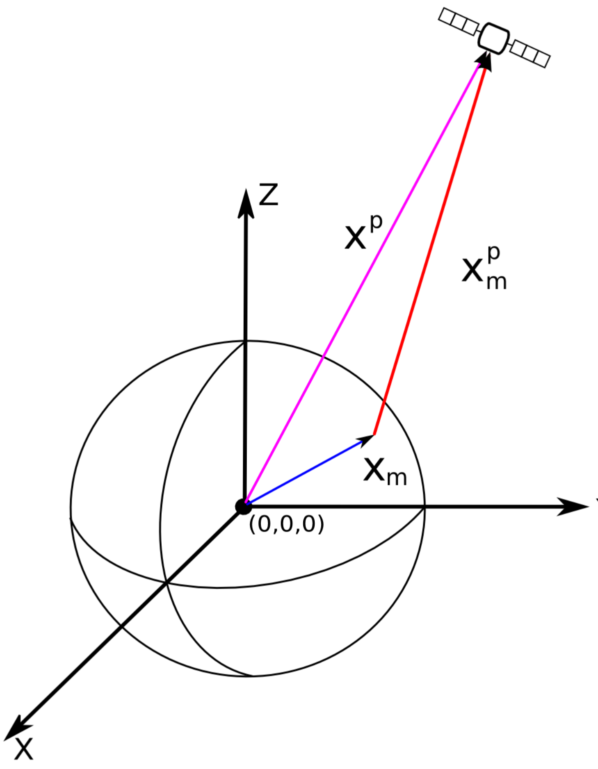


Figure 5.1: *RX-SV geometry for carrier phase method.*

$$\dot{\rho}_m^p = \rho_m^p(\dot{\mathbf{e}}_m^p \cdot \mathbf{e}_m^p) + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m^p \tag{5.4}$$

However, $\dot{\mathbf{e}}_m^p$ and $\mathbf{e}_m^p$ are orthogonal, so Equation 5.4 becomes:

$$\dot{\rho}_m^p = \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m^p \tag{5.5}$$

Now, let's introduce an equation for an additional satellite SV^q:

$$\dot{\rho}_m^q = \mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m^q \tag{5.6}$$

If we carry out single differences between satellites SV^q and SV^p:

$$\begin{array}{lcll} \nabla\dot{\rho}_m^{q,p} & = & \mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m^q - \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m^p & \Longrightarrow \\ \nabla\dot{\rho}_m^{q,p} & = & \mathbf{e}_m^q \cdot \dot{\mathbf{x}}^q - \mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m - (\mathbf{e}_m^p \cdot \dot{\mathbf{x}}^p - \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m) & \Longrightarrow \end{array}$$

$$\nabla\dot{\rho}_m^{q,p} + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}^p = \mathbf{e}_m^q \cdot \dot{\mathbf{x}}^q - \mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m \tag{5.7}$$

If we subtract $\mathbf{e}_m^p \cdot \dot{\mathbf{x}}^q$ from both sides of Equation 5.7, it yields:

$$\begin{aligned} \nabla \dot{\rho}_m^{q,p} + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}^p - \mathbf{e}_m^p \cdot \dot{\mathbf{x}}^q &= \mathbf{e}_m^q \cdot \dot{\mathbf{x}}^q - \mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m + \mathbf{e}_m^p \cdot \dot{\mathbf{x}}_m - \mathbf{e}_m^p \cdot \dot{\mathbf{x}}^q \quad \Longrightarrow \\ \nabla \dot{\rho}_m^{q,p} + \mathbf{e}_m^p \cdot (\dot{\mathbf{x}}^p - \dot{\mathbf{x}}^q) &= \mathbf{e}_m^q \cdot (\dot{\mathbf{x}}^q - \dot{\mathbf{x}}_m) - \mathbf{e}_m^p \cdot (\dot{\mathbf{x}}^q - \dot{\mathbf{x}}_m) \quad \Longrightarrow \end{aligned}$$

$$\nabla \dot{\rho}_m^{q,p} + \mathbf{e}_m^p \cdot (\dot{\mathbf{x}}^p - \dot{\mathbf{x}}^q) = (\mathbf{e}_m^q - \mathbf{e}_m^p) \cdot \dot{\mathbf{x}}_m^q \tag{5.8}$$

In Equation 5.8, if the position of the receiver is known with an accuracy better than a few meters, the direction vectors $\mathbf{e}_m^q$ and $\mathbf{e}_m^p$ could be computed without affecting the results ([Jekeli, 1994]). The satellite velocities $\dot{\mathbf{x}}^p$ and $\dot{\mathbf{x}}^q$ may be computed by different methods (more on this issue later), and the unknown is $\dot{\mathbf{x}}_m^q$. It is missing, then, the $\nabla \dot{\rho}_m^{q,p}$ term.

In order to get that term, let's present the expression for carrier phase measurements ϕ_m^p between satellite p and receiver m (more details in Section A.2):

$$\phi_m^p = \rho_m^p + c(dt_m - dt^p) + rel_m^p + T_m^p - \alpha_f I_m^p + B_m^p + \omega_{\phi,m}^p + m_{\phi,m}^p + \varepsilon_{\phi,m}^p \tag{5.9}$$

Where:

- ρ_m^p: Geometric distance between SV^p and RX_m antenna phase centers.
- dt^p: Offset of Space Vehicle (SV) clock with respect to GPS Time (GPST).
- dt_m: Offset of Receiver (RX) clock with respect to GPST.
- rel_m^p: Bias due to relativistic effects (linked to SV^p orbit eccentricity).
- T_m^p: Effect of the tropospheric delay.
- $\alpha_f I_m^p$: Effect due to the ionospheric delay. This effect is frequency-dependent ($\alpha_f = 40.3 \cdot 10^{16}/f^2$ when I is expressed in TECU and f is in Hz).
- B_m^p: The phase ambiguity term, including the carrier phase instrumental delays.
- $\omega_{\phi,m}^p$: This is the *wind-up* effect that appears in GNSS systems as GPS.
- $m_{\phi,m}^p$: Multipath effect. This effect is much smaller than in the code-based measurements.

- $\varepsilon^p_{\phi,m}$: Unmodeled noise for the phase measurement (it is in the millimeter range).

The method by [Kennedy, S., 2002b] uses the carrier phase measurements in L1 frequency due to wider availability and lower noise figure than L2 measurements and LC (ionosphere-free) combination.

Assuming that *no* cycle slip happens, differentiating Equation 5.9 regarding time remove the phase ambiguity term and most part of systematic errors and slow-varying terms, resulting in:

$$\dot{\phi}^p_m = \dot{\rho}^p_m + c(\dot{dt}_m - \dot{dt}^p) + \dot{\varepsilon}^p_{\phi,m} \tag{5.10}$$

Where the $\dot{\varepsilon}^p_{\phi,m}$ noise term absorbs the higher order terms.

In order to eliminate the clock drift a reference station k may be included, resorting to double differencing of Equation 5.10 between *nearby* receivers. This yields the approximation stated in Equation 5.11.

$$\Delta\nabla\dot{\phi}^{q,p}_{m,k} \simeq \Delta\nabla\dot{\rho}^{q,p}_{m,k} \tag{5.11}$$

Working with Equation 5.11, an approximation for term $\nabla\dot{\rho}^{q,p}_m$ is obtained:

$$\begin{array}{rcll} \Delta\nabla\dot{\phi}^{q,p}_{m,k} & \simeq & \Delta\nabla\dot{\rho}^{q,p}_{m,k} & \Longrightarrow \\ \Delta\nabla\dot{\phi}^{q,p}_{m,k} & \simeq & \nabla\dot{\rho}^{q,p}_m - \nabla\dot{\rho}^{q,p}_k & \Longrightarrow \end{array}$$

$$\nabla\dot{\rho}^{q,p}_m \simeq \Delta\nabla\dot{\phi}^{q,p}_{m,k} + \nabla\dot{\rho}^{q,p}_k \tag{5.12}$$

Where the term $\nabla\dot{\rho}^{q,p}_k$ is accurately known because it belongs to reference station k.

Substituting Equation 5.12 into Equation 5.8 and rearranging:

$$\begin{array}{rcll} \Delta\nabla\dot{\phi}^{q,p}_{m,k} + \nabla\dot{\rho}^{q,p}_k + \mathbf{e}^p_m \cdot (\dot{\mathbf{x}}^p - \dot{\mathbf{x}}^q) & = & (\mathbf{e}^q_m - \mathbf{e}^p_m) \cdot \dot{\mathbf{x}}^q_m & \Longrightarrow \\ \Delta\nabla\dot{\phi}^{q,p}_{m,k} + \nabla\dot{\rho}^{q,p}_k + \mathbf{e}^p_m\dot{\mathbf{x}}^p - \mathbf{e}^p_m\dot{\mathbf{x}}^q & = & (\mathbf{e}^q_m - \mathbf{e}^p_m) \cdot (\dot{\mathbf{x}}^q - \dot{\mathbf{x}}_m) & \Longrightarrow \end{array}$$

$$\Delta\nabla\dot{\phi}^{q,p}_{m,k} + \nabla\dot{\rho}^{q,p}_k + \mathbf{e}^p_m\dot{\mathbf{x}}^p - \mathbf{e}^q_m\dot{\mathbf{x}}^q = (\mathbf{e}^p_m - \mathbf{e}^q_m) \cdot \dot{\mathbf{x}}_m \tag{5.13}$$

Equation 5.13 is the expression used to compute the rover receiver velocity.

5.2.2 Acceleration determination

In order to obtain an expression for carrier phase-based acceleration, Equation 5.5 is differentiated a second time:

$$\ddot{\rho}_m^p = \mathbf{e}_m^p \cdot \ddot{\mathbf{x}}_m^p + \dot{\mathbf{e}}_m^p \cdot \dot{\mathbf{x}}_m^p \tag{5.14}$$

On the other hand, differentiation of Equation 5.1 yields:

$$\dot{\mathbf{x}}_m^p = \dot{\rho}_m^p \mathbf{e}_m^p + \rho_m^p \dot{\mathbf{e}}_m^p \tag{5.15}$$

From there, term $\dot{\mathbf{e}}_m^p$ can be found:

$$\dot{\mathbf{e}}_m^p = \frac{1}{\rho_m^p}\left[\dot{\mathbf{x}}_m^p - \dot{\rho}_m^p \mathbf{e}_m^p\right] \tag{5.16}$$

Substituting Equation 5.16 into Equation 5.14:

$$\begin{array}{lcll}
\ddot{\rho}_m^p & = & \mathbf{e}_m^p \cdot \ddot{\mathbf{x}}_m^p + \frac{1}{\rho_m^p}\left[\dot{\mathbf{x}}_m^p - \dot{\rho}_m^p \mathbf{e}_m^p\right] \cdot \dot{\mathbf{x}}_m^p & \Longrightarrow \\
\ddot{\rho}_m^p & = & \mathbf{e}_m^p \cdot \ddot{\mathbf{x}}_m^p + \frac{1}{\rho_m^p}\left[|\dot{\mathbf{x}}_m^p|^2 - \dot{\rho}_m^p \mathbf{e}_m^p \dot{\mathbf{x}}_m^p\right] & \Longrightarrow \\
\ddot{\rho}_m^p & = & \mathbf{e}_m^p \cdot (\ddot{\mathbf{x}}^p - \ddot{\mathbf{x}}_m) + \frac{1}{\rho_m^p}\left[|\dot{\mathbf{x}}_m^p|^2 - (\dot{\rho}_m^p)^2\right] & \Longrightarrow
\end{array}$$

$$\ddot{\rho}_m^p - \mathbf{e}_m^p \cdot \ddot{\mathbf{x}}^p - \frac{1}{\rho_m^p}\left[|\dot{\mathbf{x}}_m^p|^2 - (\dot{\rho}_m^p)^2\right] = -\mathbf{e}_m^p \cdot \ddot{\mathbf{x}}_m \tag{5.17}$$

Introducing an equation for an additional satellite SV^q, and carrying out single differences between satellites SV^q and SV^p:

$$\begin{aligned}
\nabla\ddot{\rho}_m^{q,p} - \mathbf{e}_m^q \cdot \ddot{\mathbf{x}}^q + \mathbf{e}_m^p \cdot \ddot{\mathbf{x}}^p - \frac{1}{\rho_m^q}\left[|\dot{\mathbf{x}}_m^q|^2 - (\dot{\rho}_m^q)^2\right] \\
+\frac{1}{\rho_m^p}\left[|\dot{\mathbf{x}}_m^p|^2 - (\dot{\rho}_m^p)^2\right] = -(\mathbf{e}_m^q - \mathbf{e}_m^p) \cdot \ddot{\mathbf{x}}_m
\end{aligned} \tag{5.18}$$

Now, Equation 5.12 is differentiated again:

$$\nabla\ddot{\rho}_m^{q,p} \simeq \Delta\nabla\ddot{\phi}_{m,k}^{q,p} + \nabla\ddot{\rho}_k^{q,p} \tag{5.19}$$

And substituting Equation 5.19 into Equation 5.18 and rearranging yields the expression for the carrier phase-based acceleration:

$$\Delta\nabla\ddot{\phi}^{q,p}_{m,k} + \nabla\ddot{\rho}^{q,p}_{k} + \mathbf{e}^{p}_{m}\cdot\ddot{\mathbf{x}}^{p} - \mathbf{e}^{q}_{m}\cdot\ddot{\mathbf{x}}^{q} + \frac{1}{\rho^{p}_{m}}\left[|\dot{\mathbf{x}}^{p}_{m}|^{2} - (\dot{\rho}^{p}_{m})^{2}\right]$$
$$-\frac{1}{\rho^{q}_{m}}\left[|\dot{\mathbf{x}}^{q}_{m}|^{2} - (\dot{\rho}^{q}_{m})^{2}\right] = (\mathbf{e}^{p}_{m} - \mathbf{e}^{q}_{m})\cdot\ddot{\mathbf{x}}_{m} \qquad (5.20)$$

Note that the rover velocity is a prerequisite to compute the acceleration. This is evident in terms like $\dot{\mathbf{x}}^{q}_{m}$, but also to compute terms like $\dot{\rho}^{q}_{m} = \mathbf{e}^{q}_{m}\cdot\dot{\mathbf{x}}^{q}_{m}$. Also, approximations of ρ^{q}_{m} and ρ^{p}_{m} are used, but the error will be small if rover position is known with an accuracy better than a few meters.

5.2.3 Numerical differentiation

Numerical differentiation of GNSS observables to find velocity and acceleration is an issue thoroughly studied in [Bruton, 2000] and [Bruton et al., 1999]. Several types of differentiation filters are studied there, including Taylor series approximations, Fourier series-based filters, Remez Algorithm Exchange-based filters, etc., comparing them with the ideal differentiator and weighting in their practical advantages in a GNSS data processing setting.

Relying on [Bruton, 2000], the filters that are used in [Kennedy, S., 2002a] and [Kennedy, S., 2002b] are of the Finite Impulse Response (FIR) kind because they have *linear phase*, meaning that they introduce a constant time delay that facilitates a correct time-tagging of the data. Also, only odd-length filters were used to maintain integer time delay and avoid interpolation.

Specifically, the work at [Kennedy, S., 2002b] proposes the use of a 5th order Taylor series approximation[1] FIR filter. When using a 1 Hz sampling rate, the bandwidth of that filter appropriately covers the typical dynamics found in airborne gravimetry applications, finding a compromise between bandwidth, simplicity and noise suppression ([Bruton et al., 1999] and [Kennedy, S., 2002b]).

The impulse response of the 5th order Taylor series approximation FIR filter is shown in Equation 5.21, where T is the sampling period in seconds.

$$\mathbf{h}_5[n] = \frac{1}{T}\begin{bmatrix}\frac{1}{1260} & \frac{-5}{504} & \frac{5}{84} & \frac{-5}{21} & \frac{5}{6} & 0 & \frac{-5}{6} & \frac{5}{21} & \frac{-5}{84} & \frac{5}{504} & \frac{-1}{1260}\end{bmatrix} \qquad (5.21)$$

The filter at Equation 5.21 is the one to be used to find terms like $\Delta\nabla\dot{\phi}^{q,p}_{m,k}$ and $\Delta\nabla\ddot{\phi}^{q,p}_{m,k}$, as well as $\nabla\dot{\rho}^{q,p}_{k}$, etc.

[1]The filter order represents the number of samples used on either side of the central differentiator.

The convolution summation is used to apply the former differentiating filter to a discrete data set $\mathbf{x}[n]$, obtaining a differenced signal $\mathbf{x}'[n]$:

$$\mathbf{x}'[i] = \sum_{j=-M}^{M} \mathbf{h}_M[j]\mathbf{x}[i-j] \tag{5.22}$$

Where M is the order of the differentiating filter $\mathbf{h}_M[n]$. Equation 5.22 already compensates for the constant time delay introduced by a $2M+1$ kernel length filter.

5.2.4 Covariance model

Equation systems build from Equation 5.13 or Equation 5.20 can be solved using either Least Mean Squares (LMS) or Weighted-Least Mean Squares (WMS) solvers. If the later is used, a covariance model is needed.

It can be shown ([Kennedy, S., 2002a]) that a covariance model developed for carrier phase observations may be adapted to be used for carrier phase observations derivatives. Given that the numerical differentiation is a linear combination of carrier phases, then the variances can be propagated into the derivatives, as done in single and double differencing.

There is, however, one condition for this: That the variances are constant over the time period where the differencing filter works. When using a 1 Hz sampling rate and a 5th order FIR filter this interval is 10 seconds, and this assumption reasonably holds.

Under the former conditions, if the covariance matrix of the carrier phase observables is C_ϕ and the filter *kernel* is $\mathbf{h}[n]$ (see Equation 5.21 for an example kernel), the resulting covariance matrix of the carrier phase derivatives ($C_{\dot{\phi}}$) is:

$$C_{\dot{\phi}} = \sum_{0}^{n} \mathbf{h}[n]^2 C_\phi \tag{5.23}$$

And for the second carrier phase derivatives:

$$C_{\ddot{\phi}} = (\sum_{0}^{n} \mathbf{h}[n]^2)^2 C_\phi \tag{5.24}$$

Regarding the covariance model itself, [Kennedy, S., 2002b] takes a model by [Radovanovic et al., 2001] which modeled tropospheric variances and expanded it to include ionospheric variances ([Kennedy, S., 2002a]). The covariance model is elevation-based and it also models the physical correlations between measurements as function of the separation angle between satellites and the baseline length between receivers.

5.2.4.1 Variance of a single measurement

According to [Kennedy, S., 2002a], the model for the variance of a single measurement from satellite p to receiver k is shown in Equation 5.25.

$$\sigma_k^{p2} = m_T(\varepsilon_p)^2 {\sigma_T}^2 + m_I(\varepsilon_p)^2 {\sigma_I}^2 + {\sigma_{mp}}^2 \tag{5.25}$$

Where:

- σ_k^{p2}: Variance from satellite p to receiver k.
- ε_p: Elevation angle of satellite p.
- ${\sigma_T}^2$: Tropospheric variance. In this model, this parameter is fixed at $0.02^2\, m^2$.
- ${\sigma_I}^2$: Ionospheric variance.
- ${\sigma_{mp}}^2$: Multipath variance. It is considered constant, with a value of $0.005^2\, m^2$.
- $m_T()$: Mapping function for troposphere.
- $m_I()$: Mapping function for ionosphere.

Regarding the mapping functions, $m_T()$ corresponds to Niell's '*dry*' mapping function ([Niell, 1996]), used in the UNB3 tropospheric model ([Collins, 1999]), while $m_I()$ is the ionospheric mapping function in [Misra and Enge, 2006] and presented at Equation 5.26.

$$m_I(\varepsilon) = \sqrt{1 - \left[\frac{\cos(\varepsilon)}{1 + h/R_e}\right]^2} \tag{5.26}$$

Where h is the height of the ionospheric shell (350 km), and R_e is Earth's radius.

The last term to be defined is the ionospheric variance ${\sigma_I}^2$. Its computation is done with the following procedure:

- For a given flight leg, find the satellite with the highest elevation, SV^p.
- For SV^p, compute the LC (ionosphere-free carrier phase) combination for each epoch.
- Compute (L1 - LC) to get the first order ionospheric error.
- Subtract from the former a 'line of best fit' to eliminate ionosphere first order trend. This will leave ionospheric second and third order effects, and carrier-phase noise (increased by LC computation).
- The ionospheric variation (according to previous steps) is mapped to zenith using the aforementioned mapping function.
- The *variance* of former ionospheric variation results will be taken as the ionospheric variance for the whole flight leg.

5.2.4.2 Covariance between two satellites

In order to compute the covariance between measurements from a receiver p to two satellites p and q (denoted as $c(\phi_k^p, \phi_k^q)$), Equation 5.27 is used.

$$c(\phi_k^p, \phi_k^q) = m_T(\varepsilon_p) m_T(\varepsilon_q) e^{-\theta/\Omega} {\sigma_T}^2 + m_I(\varepsilon_p) m_I(\varepsilon_q) e^{-\theta/\Omega} {\sigma_I}^2 \tag{5.27}$$

The separation angle between satellites p and q is θ, and it is computed with the expression:

$$\cos(\theta) = \sin(\varepsilon_p)\sin(\varepsilon_q) + \cos(\varepsilon_p)\cos(\varepsilon_q)\cos(A_p - A_q) \tag{5.28}$$

Where A_p and A_q are the azimuth angles for satellites p and q, respectively, and Ω is the *correlation angle*. The value of this correlation angle is set empirically to 40 degrees ([Kennedy, S., 2002a]).

5.2.4.3 Covariance between two receivers

Equation 5.29 presents the covariance between measurements from two receivers m and k to a common satellite p.

$$c(\phi_m^p, \phi_k^p) = m_T(\varepsilon_p)^2 e^{-d/D} {\sigma_T}^2 + m_I(\varepsilon_p)^2 e^{-d/D} {\sigma_I}^2 \tag{5.29}$$

Being d the baseline length between receivers m and k, and D the *correlation distance*. According to [Radovanovic et al., 2001], the value of D is set to 350 km.

5.2.4.4 Covariance between different receivers and satellites

Finally, the covariance between measurements made to different satellites from different receivers is shown in Equation 5.30.

$$c(\phi_m^p, \phi_k^q) = m_T(\varepsilon_p) m_T(\varepsilon_q) e^{-\theta/\Omega} e^{-d/D} {\sigma_T}^2 + m_I(\varepsilon_p) m_I(\varepsilon_q) e^{-\theta/\Omega} e^{-d/D} {\sigma_I}^2 \tag{5.30}$$

Where all terms have been already defined in the previous sections.

5.2.4.5 Comments on covariance model

The implementation and testing of the covariance model explained in the former sections raised some issues that it is worth commenting:

- The expressions 5.29 and 5.30 concerning two receivers propose using the same correlation distance D for both troposphere and ionosphere. These are very different physical phenomena with different correlation distances, and therefore those equations may not accurately portray the covariances they are intended to compute.
- Under some input conditions, the implementation of the full covariance model yielded covariance matrices that were ill-conditioned and very difficult to invert.
- The implementation of scaled-down versions of the proposed covariance model, using only Equation 5.25 and Equation 5.27, or just Equation 5.25 (simple diagonal covariance matrix) generated software that ran several times faster and produced results with negligible differences regarding the full covariance model implementation.

It is out of the scope of this work to evaluate the best covariance model to use with the carrier phase-based method of velocity and acceleration determination. Nevertheless, assessing simpler alternatives for these covariance models is suggested as a future research line.

5.3 Improving velocity results

The first proposed change to [Kennedy, S., 2002b] method is straightforward and consists in modifying the way satellite velocities and accelerations are computed.

At page #966, [Kennedy, S., 2002b] writes:

> *"Satellite velocities and accelerations are also required in the carrier phase method of receiver acceleration determination. These quantities can be derived from the Lagrange polynomials as well. Numerical differentiation would require satellite positions at several epochs to calculate velocity and acceleration. This is inconvenient and unnecessary. The Lagrange polynomial functions can be analytically differentiated and evaluated at the desired time."*

Regarding the former paragraph, it must be stated that the analytically differentiated Lagrange polynomial functions[2] *also require* satellite positions at several epochs to work. Therefore, they don't represent an advantage in this respect.

More importantly, the analytical differentiation of Lagrange polynomial interpolation does not necessarily reflect the physical nature of satellite orbits. The Lagrange polynomial fit of a given set of points may yield oscillations, an effect called "*Runge phenomenon*" (see for instance [Dahlquist and Bjork, 1974]). Those oscillations may not pose a problem when computing satellite positions, but they may (and indeed do) introduce unwanted biases in the satellite velocity determination.

Therefore, the modification consists in also using differentiator FIR filters to compute satellite velocity and acceleration. In order to test the effects of this change, 5 hours of data from two static GPS stations called `UPC1` and `UPC2` were processed using the full Kennedy method. Those stations have a very short baseline (37.86 m), so most observation errors are cancelled during double differences. Data rate was 1 Hz and data collection corresponds to August 8th, 2009.

The results in Figure 5.2 show 5 minutes averages of 3D velocity errors for both approaches: Lagrange differentiator vs. FIR differentiator.

It can be seen that the difference between using one method or the other is remarkable: The RMS of the 5-min average velocity 3D error using the Lagrange

[2]Lagrange polynomial functions are used to interpolate satellite positions from SP3 precise ephemeris files ([Hofmann-Wellenhof et al., 2008]).

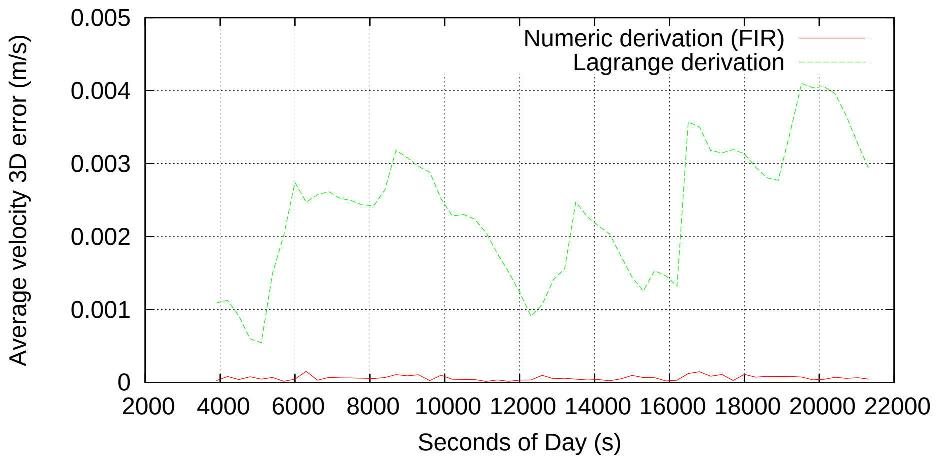


Figure 5.2: *Average velocity 3D error (5 minutes interval) for UPC2.*

	FIR average (mm/s)	FIR σ (mm/s)	Lagrange average (mm/s)	Lagrange σ (mm/s)
Up	0.02	3.51	-2.15	3.63
East	0.01	1.65	-0.58	1.69
North	0.00	1.68	0.51	1.76

Table 5.1: *Velocity averages and sigmas for static results (UPC2-UPC1).*

analytical differentiation is 2.52 mm/s, while the corresponding RMS value for the FIR differentiator is under 0.07 mm/s.

Most of this difference comes from biases, specially in the Up velocity component. Table 5.1 presents the averages and standard deviations for Up, East and North components for both approaches.

The results for acceleration yield negligible differences when using one method or the other, as Table 5.2 shows. This may be explained by looking back at Equation 5.20, where relative velocity errors are scaled down by a factor of ρ_m^{SV}, which is a very large value. This is further explained in [Kennedy, S., 2002a].

Please note that the static error sigmas for velocities in the FIR case (Table 5.1) are below what is reported by [van Graas and Soloviev, 2004] (please refer to

	FIR average $(\mathbf{mm/s^2})$	**FIR** σ $(\mathbf{mm/s^2})$	**Lagrange average** $(\mathbf{mm/s^2})$	**Lagrange** σ $(\mathbf{mm/s^2})$
Up	$-4.0\,10^{-3}$	5.45	$-4.2\,10^{-3}$	5.45
East	$3.2\,10^{-3}$	2.57	$3.2\,10^{-3}$	2.57
North	$0.7\,10^{-3}$	2.61	$0.3\,10^{-3}$	2.61

Table 5.2: *Acceleration averages and sigmas for static results (UPC2-UPC1).*

	Velocity average $(\mathbf{mm/s})$	**Velocity** σ $(\mathbf{mm/s})$	**Acceleration average** $(\mathbf{mm/s^2})$	**Acceleration** σ $(\mathbf{mm/s^2})$
Up	-0.13	4.47	$3.3\,10^{-3}$	6.92
East	0.06	2.22	$2.4\,10^{-3}$	3.43
North	0.03	2.42	$-4.2\,10^{-3}$	3.76

Table 5.3: *Velocity and accelerations results for PLAN-EBRE.*

Section 5.1). Thence, the question that arises is if the results will be still good if the receivers were separated a longer distance.

In order to test the former hypothesis, 3 hours of data at 1 Hz data rate were processed for receivers PLAN (*rover*) and EBRE (*reference*) for January 15th, 2010. The baseline between those receivers is 142 km.

Table 5.3 presents the results for this longer baseline. It can be seen that although the results are not as good as with the very short baseline (as expected), they still show consistently better sigmas than [van Graas and Soloviev, 2004] static velocity results, specially in the Up direction (σ_{Up} = 7.9 mm/s, σ_{East} = 2.2 mm/s, and σ_{North} = 3.1 mm/s).

5.4 EVA: Extended velocity and acceleration determination

Kennedy's method uses L1 carrier phase observable because of its lower noise figure, and relies on double differencing to eliminate or minimize error sources.

This approach limits the baseline length between rover receiver and reference station because, depending on the ionospheric conditions, a point will be reached where the errors not cancelled out by the double differencing will offset the advantages of using the L1 carrier phase.

The following sections will present modifications to this method where undifferenced LC (ionosphere-free) carrier phase combinations are used to overcome the baseline limitation. The new method borrows ideas from the POP method presented in Chapter 4 to estimate clock drift and clock drift rates, and will henceforth be called "Extended Velocity and Acceleration determination (EVA)".

5.4.1 Computing the velocity

Let's start considering Equation 5.6. It may be rewritten as:

$$\dot{\rho}_m^q - \mathbf{e}_m^q \cdot \dot{\mathbf{x}}^q = -\mathbf{e}_m^q \cdot \dot{\mathbf{x}}_m \tag{5.31}$$

The idea is to find a way to estimate the term $\dot{\rho}_m^q$. In order to achieve that, Equation 5.9 is rewritten to consider the ionosphere-free carrier phase combination $\phi_{LC,m}^q$.

$$\phi_{LC,m}^q = \rho_m^q + c(dt_m - dt^q) + rel_m^q + T_m^q + \delta T_m^q + BC_m^q + \omega_{LC,m}^q + m_{LC,m}^q + \varepsilon_{LC,m}^q \tag{5.32}$$

Where, in addition to the terms previously explained, we have:

- δT_m^q: *Unmodeled* tropospheric delay.
- BC_m^q: Ionosphere-free phase ambiguity .

Now, let's introduce a *reference clock* dt_0 to which all the other clocks will refer to. Hence:

$$\begin{aligned} d\tau_m &= dt_m - dt_0 \\ d\tau^q &= dt^q - dt_0 \end{aligned}$$

And Equation 5.32 becomes:

$$\phi^q_{LC,m} - rel^q_m - T^q_m - \omega^q_{LC,m} = \rho^q_m + c(d\tau_m - d\tau^q) + \delta T^q_m + BC^q_m + m^q_{LC,m} + \varepsilon^q_{LC,m} \tag{5.33}$$

Where terms to the left are the observation minus modelable effects, and they will be denoted as $\phi^q_{0,m}$. Arranging terms yields:

$$\rho^q_m = \phi^q_{0,m} - c(d\tau_m - d\tau^q) - \delta T^q_m - BC^q_m - \varepsilon^q_{LC,m} \tag{5.34}$$

Where term $m^q_{LC,m}$ is considered as absorbed into $\varepsilon^q_{LC,m}$. Derivating Equation 5.34 regarding time, and assuming that no cycle slip happens, results in Equation 5.35.

$$\dot{\rho}^q_m = \dot{\phi}^q_{0,m} - cd\dot{\tau}_m + cd\dot{\tau}^q - \delta\dot{T}^q_m - \dot{\varepsilon}^q_{LC,m} \tag{5.35}$$

The former Equation 5.35 could be considered as equivalent to Equation 6.33 at page 218 of [Misra and Enge, 2006]. If we further consider that troposphere variation rate $\delta\dot{T}^q_m$ is negligible, as well as first order errors, then:

$$\dot{\rho}^q_m \simeq \dot{\phi}^q_{0,m} - cd\dot{\tau}_m + cd\dot{\tau}^q \tag{5.36}$$

For Equation 5.36 to properly work, receiver clock millisecond adjustments must be taken care of, either preprocessing RINEX observation files or using clock steering-style receivers. Then, combining Equation 5.36 with Equation 5.31 results in the new equation for rover velocity estimation:

$$\dot{\phi}^q_{0,m} - \mathbf{e}^q_m \cdot \dot{\mathbf{x}}^q = -\mathbf{e}^q_m \cdot \dot{\mathbf{x}}_m + cd\dot{\tau}_m - cd\dot{\tau}^q \tag{5.37}$$

The unknowns in the former equation are the rover velocity $\dot{\mathbf{x}}_m$, the rover clock drift $cd\dot{\tau}_m$, and the satellite clock drift $cd\dot{\tau}^q$.

Following the ideas set forth for POP method in Chapter 4, let's introduce a fixed *master* station "0". The receiver clock of this station is the reference clock dt_0 introduced earlier. Then, Equation 5.37 for *master* station becomes:

$$\dot{\phi}^q_{0,0} - \mathbf{e}^q_0 \cdot \dot{\mathbf{x}}^q = -cd\dot{\tau}^q \tag{5.38}$$

As it can be seen, the equations corresponding to *master* station allow to estimate satellite clock drifts[3]. Following the same procedure, let's introduce some *reference* stations denoted with "k":

$$\dot{\phi}^q_{k,m} - \mathbf{e}^q_k \cdot \dot{\mathbf{x}}^q = cd\dot{\tau}_k - cd\dot{\tau}^q \tag{5.39}$$

The equations provided by the reference stations reinforce the estimation of satellite clock drifts. As in Section 4.3, Equations 5.37 to 5.39 allow to build an equation system suitable to be solved using the GPSTk `SolverGeneral` class.

5.4.2 Computing the acceleration

From Equation 5.17, we have for a satellite q that:

$$\ddot{\rho}^q_m - \mathbf{e}^q_m \cdot \ddot{\mathbf{x}}^q - \frac{1}{\rho^q_m}\left[|\dot{\mathbf{x}}^q_m|^2 - (\dot{\rho}^q_m)^2\right] = -\mathbf{e}^q_m \cdot \ddot{\mathbf{x}}_m \tag{5.40}$$

On the other hand, further derivating Equation 5.36 regarding time yields:

$$\ddot{\rho}^q_m \simeq \ddot{\phi}^q_{0,m} - cd\ddot{\tau}_m + cd\ddot{\tau}^q \tag{5.41}$$

Combining Equation 5.40 and Equation 5.41 we obtain the new equation to estimate rover acceleration:

$$\ddot{\phi}^q_{0,m} - \mathbf{e}^q_m \cdot \ddot{\mathbf{x}}^q - \frac{1}{\rho^q_m}\left[|\dot{\mathbf{x}}^q_m|^2 - (\dot{\rho}^q_m)^2\right] = -\mathbf{e}^q_m \cdot \ddot{\mathbf{x}}_m + cd\ddot{\tau}_m - cd\ddot{\tau}^q \tag{5.42}$$

Introducing again a *master* station "0" and several *reference* stations "k":

$$\ddot{\phi}^q_{0,0} - \mathbf{e}^q_0 \cdot \ddot{\mathbf{x}}^q - \frac{1}{\rho^q_0}\left[|\dot{\mathbf{x}}^q|^2 - (\dot{\rho}^q_0)^2\right] = -cd\ddot{\tau}^q \tag{5.43}$$

$$\ddot{\phi}^q_{0,k} - \mathbf{e}^q_k \cdot \ddot{\mathbf{x}}^q - \frac{1}{\rho^q_k}\left[|\dot{\mathbf{x}}^q|^2 - (\dot{\rho}^q_k)^2\right] = cd\ddot{\tau}_k - cd\ddot{\tau}^q \tag{5.44}$$

[3]In fact, satellite clock drifts with respect to *master* station clock drift.

The former equation system can also be solved using the GPSTk `SolverGeneral` class.

5.5 Applying EVA method to aircraft data

In this section, an application of the EVA method to aircraft velocity and acceleration determination was tested. The data comes from the flight of a light aircraft from Perpignan airport, south of France, to La Cerdenya aerodrome, in Spain's Pyrenees, with a previous pass close to Llivia aerodrome. The distance from Perpignan to Llivia are about 79 km, and the distance between Llivia and La Cerdenya are 13 km.

This experiment will use data from five Institut Cartografic de Catalunya (ICC) reference stations: AVEL, CREU, EBRE, LLIV and MATA (Please see Figure 5.3). The LLIV (Llivia) station will *not* be used for EVA method data processing, being saved for reference determination purposes with Kennedy's method, while the remaining four-station network has MATA as the closest station to LLIV, 111 km away.

It is important to note that the Kennedy method used for comparisons is the modified version according to Section 5.3.

5.5.1 Aircraft description

The aircraft used was a *Robin DR 400-140B*, shown in Figure 5.4. This is a light airplane with some properties that make it very appropriate for this experiment:

- It is a low-wing aircraft, with an excellent sky view from the cockpit.
- Most of the cockpit roof is made of Plexiglas, transparent to radiowaves.
- Most of the aircraft structure is made of wood and fabric instead of metal, minimizing multipath.

5.5.2 Data collection description

The GPS receiver used was a two-frequency geodetic-grade *Septentrio PolaRx2*. The antenna was a two-frequency AERAT2775_43 model, and it was attached on the upper right side of the instrument panel, inside the cockpit.

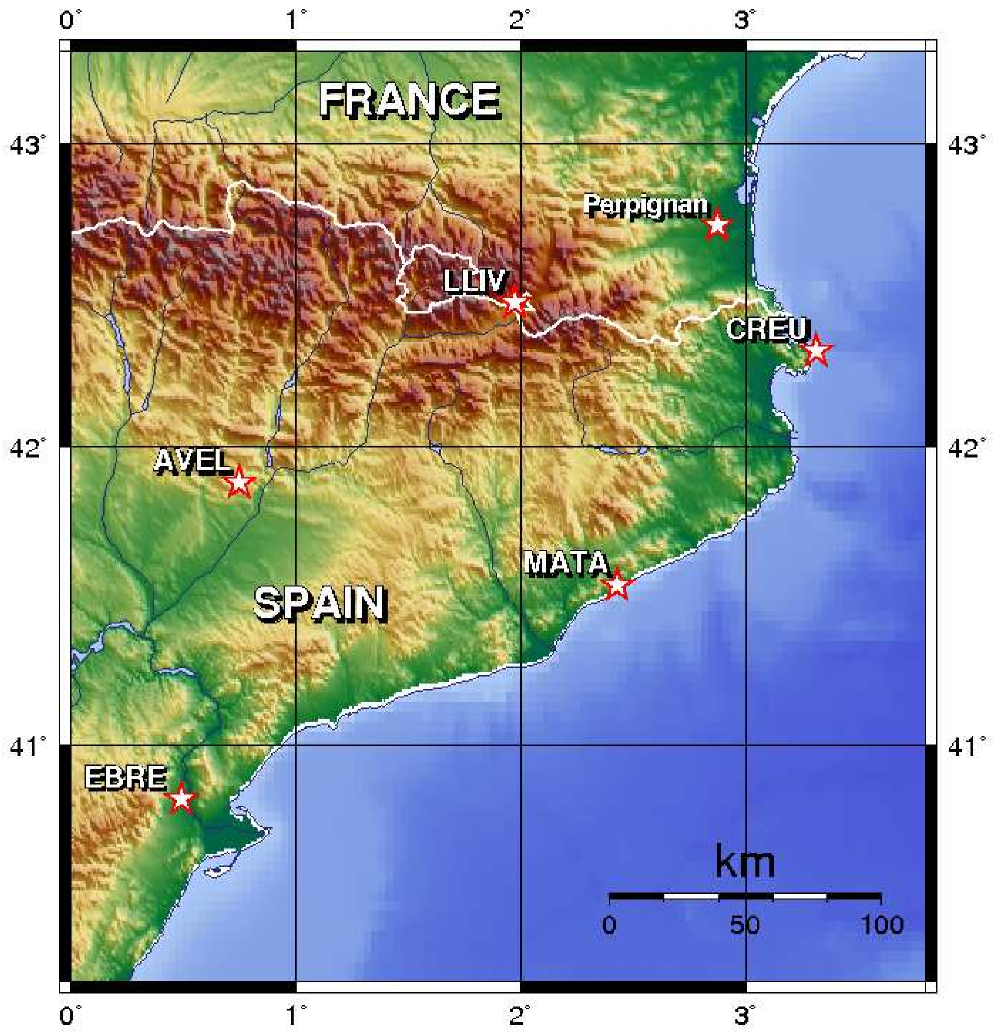


Figure 5.3: *Station network for Pyrenees flight.*

Figure 5.4: *Aircraft Robin DR400-140B.*

The test flight was carried out on March 21st, 2009, under Visual Flight Rules (VFR) with clear skies along the flight path. It lasted a little over 1 hour (from 11:36 to 12:40, local time) and was composed of the following phases:

1. Preflight check (with the GPS receiver on) at Perpignan airport parking area.
2. Takeoff from Perpignan airport.
3. Cruise flight to the Pyrenees, passing over the "Coll de la Percha" pass.
4. Approach maneuver to Llivia aerodrome, followed by a missed approach (previously planed).
5. Approach maneuver, and not-previously-planned missed approach, to La Cerdenya aerodrome.
6. Final approach maneuver and landing at La Cerdenya aerodrome.
7. Postflight check (with the GPS receiver on) at La Cerdenya aerodrome parking area.

5.5.3 Aircraft data processing

In order to process data with the EVA method the aircraft *a priori* position has to be updated each epoch. For this, the ionosphere-free pseudorange observable smoothed with the ionosphere-free carrier phase, was used at each epoch to generate an approximate position (see Section 2.5.3 for a similar procedure), which would then be used as *a priori* for the modeling phase. The rest of the procedure is as explained in Section 5.4.

On the other hand, the Kennedy method was also implemented but including the improvements proposed in Section 5.3 (called hereafter modified-Kennedy)[4], and the fixed LLIV station was used as reference. It is important to remark here that the data was processed when the aircraft was relatively close to LLIV (less than 37 km).

Figure 5.5 shows the results for aircraft horizontal velocity using both approaches. The match is remarkable, and the apparent differences in the plot are due to areas where one method is providing solutions while the other is not. The flat area at lower right is because the aircraft has already landed and it is parked.

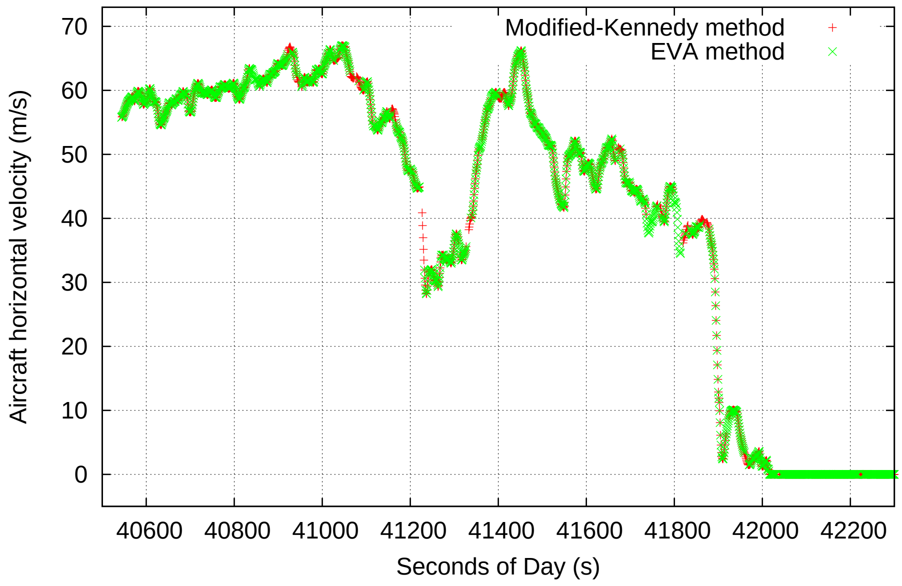


Figure 5.5: *Aircraft horizontal velocity.*

Figure 5.6 plot the differences in velocities of EVA with respect to modified-Kennedy for the three coordinates. The standard deviation of velocity differences is 6.68 mm/s for the "Up" component and 2.75 mm/s for both the "East" and "North" velocity components. Regarding acceleration, the corresponding standard deviation values are 8.39 mm/s^2 (Up), 3.35 mm/s^2 (East) and 3.41 mm/s^2 (North). This results can be considered as very good, specially

[4]The modified-Kennedy method is used instead of the original one in order to achieve a thirty-fold improvement in the velocity biases estimation.

taking into account that in this scenario the aircraft is always close to modified-Kennedy's reference station (LLIV), and the EVA method closest reference station is over one hundred kilometers away.

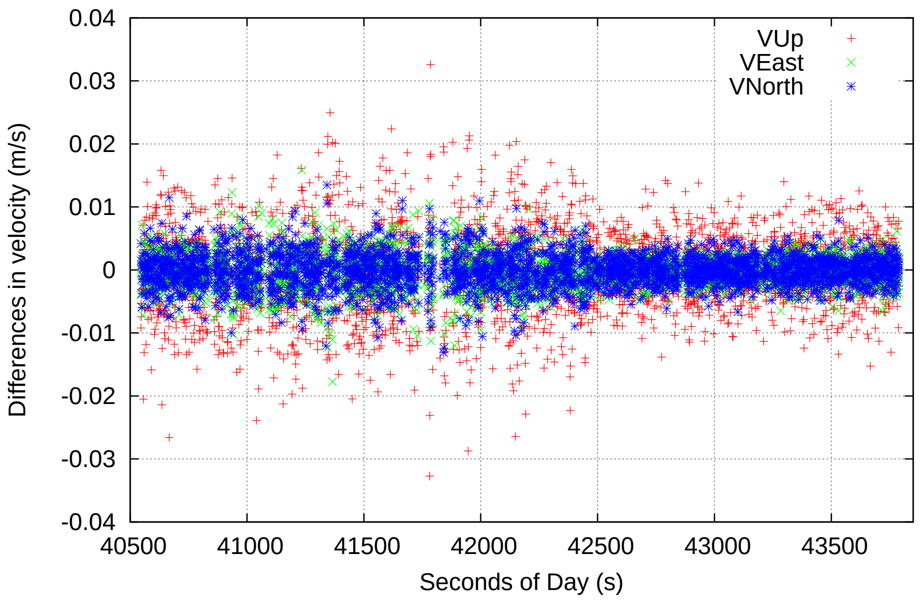


Figure 5.6: *Aircraft velocity differences of EVA with respect to modified-Kennedy.*

A better way to assess the results of each method is to study the period of time when the aircraft is parked at La Cerdenya airport, at 13 km from LLIV reference station. This period spans from second of day 42130 s up to 43790 s. For comparison purposes, a RTK solution generated with the RTKLIB tool ([Takasu and Yasuda, 2009]) was also included. The RTK solution was twice numerically differenced using the FIR filter described in Section 5.2.3 in order to obtain both velocity and acceleration reference values. The LLIV station was also used as fixed station for the RTK procedure.

As Figure 5.7 shows, the results of the three methods are not very different when compared to a known reference velocity, being the modified-Kennedy method the one producing slightly better results.

The acceleration estimation yields similar results, although modified-Kennedy and EVA methods show an advantage with respect to RTK. Table 5.4 presents a summary of average and standard deviations for the "Up" components during the static period.

Overall, the results of these three methods are similar and very good, although modified-Kennedy method shows some advantages. On the other hand, EVA is a little behind RTK regarding velocity estimation, but both modified-Kennedy

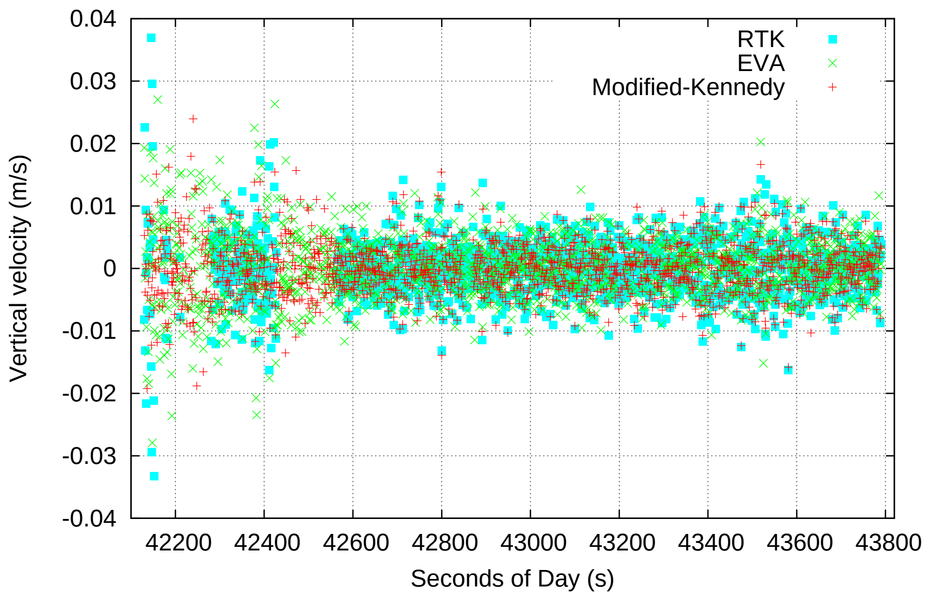


Figure 5.7: *Aircraft vertical velocity during static section.*

	RTK	Modified-Kennedy	EVA
VUp AVG $(\mathrm{mm/s})$	0.004	-0.015	0.158
VUp σ $(\mathrm{mm/s})$	4.890	4.447	5.299
AUp AVG $(\mathrm{mm/s^2})$	0.025	-0.017	-0.017
AUp σ $(\mathrm{mm/s^2})$	7.942	6.750	6.870

Table 5.4: *Aircraft velocity and acceleration averages and sigmas for static period.*

Station	Latitude	Longitude	Distance to BOGT
AREQ	$-16.47°$	$-71.49°$	$2339\ km$
BOGT	$4.64°$	$-74.08°$	-
CRO1	$17.76°$	$-64.58°$	$1777\ km$
GLPS	$-0.74°$	$-90.3°$	$1893\ km$
KOUR	$5.25°$	$-52.81°$	$2347\ km$

Table 5.5: *South America network data.*

and EVA outperform RTK in acceleration estimation. All studied methods have better performance than [van Graas and Soloviev, 2004].

5.6 Applying EVA method to very long ranges

The previous section compared EVA method performance with differential methods when there was a reference station near the working area. However, the purpose of EVA method is to provide velocity and acceleration estimations when there is no nearby reference station, and it is in this scenario where EVA method excels.

In order to test the EVA method in a demanding setting, a very wide area in equatorial South America was processed, using 1 Hz data taken from January 15th., 2010, from 19:00 to 20:00 UT (about local noon). The processed network had 5 stations, using BOGT as "*rover*" and station CRO1 as "*master*". Table 5.5 shows station data, while Figure 5.8 presents a map of this network.

The modified-Kennedy method was used for comparison purposes taking station CRO1 as reference. Table 5.6 shows the results for velocity and acceleration when using modified-Kennedy method, while Table 5.7 presents EVA method results.

It can be seen that given the distances, the latitude, and the epoch of day involved, modified-Kennedy method results could be considered as good, but the results obtained with the EVA method are considerable better.

The former statement is reinforced when the vertical velocity component is plot for both methods, as Figure 5.9 shows. Not only the EVA method has consid-

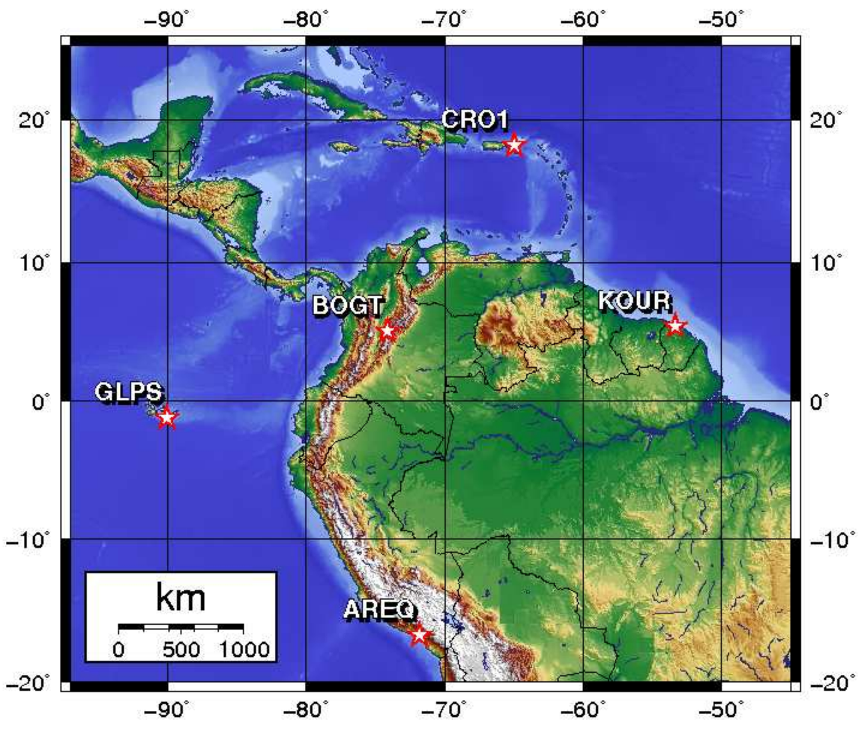


Figure 5.8: *South America network.*

	Velocity average $(\mathbf{mm/s})$	**Velocity** σ $(\mathbf{mm/s})$	**Acceleration average** $(\mathbf{mm/s^2})$	**Acceleration** σ $(\mathbf{mm/s^2})$
Up	0.54	6.90	$0.6\,10^{-3}$	10.27
East	0.67	2.31	$1.8\,10^{-3}$	3.34
North	0.48	4.74	$6.8\,10^{-3}$	7.08

Table 5.6: *Velocity and accelerations results for BOGT-CRO1 (Modified-Kennedy method).*

	Velocity average (mm/s)	**Velocity** σ (mm/s)	**Acceleration average** (mm/s^2)	**Acceleration** σ (mm/s^2)
Up	$-9.5\,10^{-3}$	1.67	$-6.3\,10^{-3}$	2.37
East	0.02	1.29	$5.6\,10^{-3}$	1.84
North	-0.02	0.95	$-2.7\,10^{-3}$	1.35

Table 5.7: *Velocity and accelerations results for BOGT (EVA method).*

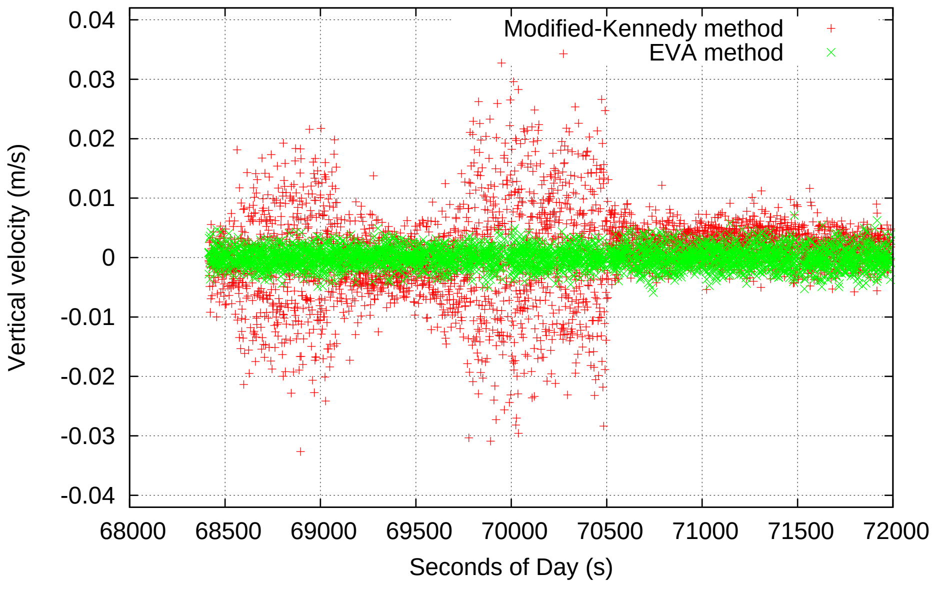


Figure 5.9: *Vertical velocity for BOGT station.*

erable less dispersion, but also shows less biases. The same behavior can be observed for the rest of velocity and acceleration components. This improvement was expected, and it is ascribable both to the improved geometry provided by the network and to the use of ionosphere-free carrier phase observations.

5.7 Summary

In this chapter several methods to compute velocity and acceleration were reviewed, emphasizing on the carrier phase-based Kennedy method because of its good performance. This method was explained in detail.

A reference implementation of Kennedy method was developed, and several experiments were carried out. Experiments done with very short baselines showed a flaw in the way satellite velocities were computed, introducing biases in the velocity solution.

A relatively simple modification was proposed to fix the former issue. The modified version reduced the RMS of 5-min average velocity 3D errors by a factor of over 35. Also, preliminary results suggest that simpler and faster covariance models could yield equivalent results, and this is proposed as a future research line.

Then, borrowing ideas from the modified-Kennedy method and the POP method presented in Chapter 4, a new velocity and acceleration determination procedure was developed and implemented that greatly extends the effective range. This method was named "Extended Velocity and Acceleration determination (EVA)".

An experiment was setup using a light aircraft flying over the Pyrenees. This experiment showed that both the modified-Kennedy and EVA methods were able to cope with the dynamics of this type of flight. An additional RTK-derived solution was also generated, and when comparing the three methods to a known zero-velocity reference, the results were very similar, although the modified-Kennedy method showed some advantages. EVA was a little behind RTK regarding velocity estimation, but both modified-Kennedy and EVA outperformed RTK in acceleration estimation.

Afterwards, modified-Kennedy and EVA method were applied to a very wide network on equatorial South America, near local noon, with baselines over 1770 km. Under this scenario, the EVA method showed a clear advantage in both averages and standard deviations for all components of velocity and acceleration, confirming its effectiveness with long baselines.

Finally, a research article summarizing these results is being written, and reference implementations of both modified-Kennedy and EVA methods (as well as its associated classes) are currently under development and will be added to the GPSTk project in the near future.

Conclusions

The work carried out during the development of this Ph.D. thesis achieved both the general and specific research objectives set forth in the Introduction of this document, including the study, development and implementation of algorithms for GNSS navigation, focusing on precise position, velocity and acceleration determination very far from reference stations in post-process mode.

Given that one of the goals of this thesis was to develop a set of state-of-the-art GNSS data processing tools, and make them available for the research community, the software development effort was done within the frame of the preexistent open source GPSTk project. In particular, Chapter 1 presented the general characteristics of that project such as structure, basic facilities and development philosophy, showing the high level of portability of the GPSTk.

Validation of the GPSTk pseudorange-based processing capabilities was carried out during this phase of the work. The results of comparisons with a trusted GPS data processing tool (BRUS) showed an excellent agreement both in the positioning and the modeling domains, confirming the viability of the GPSTk as a source code base for developing reliable GNSS data processing software.

GNSS data management proved to be an important issue when trying to extend GPSTk capabilities to carrier phase-based data processing algorithms. This task was tackled in Chapter 2, where the GNSS Data Structures (GDS) were presented, including the motivation for their development, the implementation overview, and their associated *processing paradigm*. Several types of pseudorange-based data processing strategies were included in Chapter 2 in order to better show how the GNSS Data Structures (GDS) can be used.

The main contribution of the GDS relies in the fact that they preserve both the data and corresponding "*metadata*" (data relationships), internally indexing all the GNSS-related information. With the associated GDS paradigm, the GNSS data processing then becomes like an "*assembly line*", where all the processing steps are performed sequentially. This approach provides an easy and straightforward way to encapsulate and process data, allowing writing clean, simple to read and use software that speeds up development and reduces errors.

The extension of GPSTk capabilities to carrier phase-based data processing algorithms was the focus of Chapter 3. In that chapter some applications of the GDS to carrier phase-based GNSS data processing strategies were presented, as well as important accessory classes that ease tackling these complex tasks. Reference implementations of those strategies were provided for the GNSS community in the GPSTk `examples` directory, as files `example8.cpp`, `example9.cpp` and `example10.cpp`.

The performance comparison of these relatively simple GDS-based source code examples with other state-of-the art Precise Point Positioning (PPP) suites, demonstrated that their results are among the best, confirming the validity of using the GPSTk combined with the GDS to get easy to write and maintain, yet powerful, GNSS data processing software. Furthermore, given that the GDS design is based on data abstraction, it allows a very flexible handling of concepts beyond mere data encapsulation, including programmable general solvers, among others.

Chapter 4 dealt with the problem of post-process precise positioning of GPS receivers hundreds of kilometers away from nearest reference station at arbitrary data rates, overcoming an important limitation of classical post-processing strategies like PPP. The advantages of GDS data abstraction regarding solvers, and in particular the possibility to set up a "general solver" object, were used to implement a kinematic PPP-like processing based on a network of stations. This procedure was named *Precise Orbits Positioning (POP)* because it is independent of precise clock information and it only needs precise orbits to work.

The POP procedure involved multiple stations separated hundreds of kilometers and there are a great number of unknowns of several kinds: Some unknowns are *receiver-indexed* (or *receiver-specific*, like ztd_i, dx, dy, etc.), some are *satellite-indexed* (dt^j), and others are both *receiver-* and *satellite-indexed*, like Bc_i^j. Therefore, the number of unknowns at a given epoch has a wide variation depending on the available station data and the number of visible satellites. The GPSTk-provided class `SolverGeneral`, developed during this thesis, helps implement this kind of systems, *describing* (rather than hard coding the procedure in software), the equations, variables, and their associated stochastic models and relationships. The program `example14.cpp` was provided as a reference implementation of this data processing method.

The results from this approach were very similar (as expected) to the standard kinematic PPP processing strategy, but yielding a higher positioning rate. Also, the network-based processing of POP seems to provide additional robustness to the results, even for receivers outside the network area. The distance from "*rover*" to nearest reference station does not seem to be a critical factor, because in the tests carried out the results were not significantly degraded when this distance nearly doubled.

On the other hand, the convergence time improves in POP as the number of station in the network increases, but up to a limit. This issue poses a problem if the POP method is going to be applied to vehicles, specially if data arcs are short.

The last part of this thesis focused on implementing, improving and testing algorithms for the precise determination of velocity and acceleration hundreds of kilometers away from nearest reference station. Chapter 5 reviewed several methods to compute velocity and acceleration, emphasizing on the carrier phase-based Kennedy method because of its good performance, explaining it in detail.

A reference implementation of Kennedy method was developed, and several experiments were carried out. Experiments done with very short baselines showed a flaw in the way satellite velocities were computed, introducing biases in the velocity solution. A relatively simple modification was proposed, and it reduced the RMS of 5-min average velocity 3D errors by a factor of over 35, leading to a modified version of the Kennedy method. In addition, preliminary results experimenting with the covariance models suggested that simpler and faster covariance models could yield equivalent results to the full model originally proposed by Kennedy.

Then, borrowing ideas from Kennedy method and the POP method presented in Chapter 4, a new velocity and acceleration determination procedure was developed and implemented that greatly extends the effective range. This method was named "Extended Velocity and Acceleration determination (EVA)".

An experiment using a light aircraft flying over the Pyrenees showed that both the modified-Kennedy and EVA methods were able to cope with the dynamics of this type of flight. When comparing these methods to a known zero-velocity reference the results were very similar, although modified-Kennedy method showed some advantages. EVA performance was a little behind RTK-derived velocity estimations, but modified-Kennedy and EVA outperformed RTK in acceleration estimations.

Finally, both modified-Kennedy and EVA method were applied to a very wide network on equatorial South America, near local noon, with baselines over 1770 km. Under this scenario, the EVA method showed a clear advantage in both averages and standard deviations for all components of velocity and acceleration. This confirms that EVA is an effective method to precisely compute velocities and accelerations when the distance to the nearest reference station is over one thousand kilometers.

Contributions

The development of the GNSS Data Structures (GDS) and its processing paradigm is one of the contributions from this thesis work. The GDS solve some GNSS data management issues preserving both data and metadata, providing a way to write software that speeds up development and reduces errors.

The POP procedure is considered another contribution. Although not an original strategy (similar methods have been previously used in the literature), its implementation is a novel approach to solve this kind of problems, using a run-time programmable solver (`SolverGeneral`) where the equations, variables, and their associated stochastic models and relationships are *described* rather than *hard coded* in software. Also, this approach is flexible enough to be used in other types of complex problems, as it was demonstrated in Chapter 5.

The study of carrier phase-based velocity and acceleration methods provided other contribution: The modification of way the Kennedy method computes the satellite velocities yielded improvements of over one order of magnitude in the biases of velocity estimations.

Additionally, the development of the new "Extended Velocity and Acceleration determination (EVA)" method solves the problem of post-process precise velocity and acceleration determination thousands of kilometers from nearest reference station. This is considered an important and original contribution that could have an impact in fields such as aerogravimetry, where the original Kennedy method was applied.

Other relatively minor contributions were the validation of the initial GPSTk code base, the demonstration of its porting process to an embedded platform, and the extension of GPSTk capabilities to process carrier phase-based data, in particular PPP processing. The reference implementations of several data processing strategies should also be very useful for GNSS researchers and students.

In summary, this work has provided both *scientific* and *logistic* contributions for the GNSS research community, striving to provide tools to increase the productivity of GNSS researchers.

Publications

This thesis work resulted in a publication in a peer-reviewed journal:

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "GNSS data management and processing with the GPSTk". GPS Solutions, DOI: 10.1007/s10291-009-0149-9, 2009.

Also, several publications in congress proceedings were related to this thesis:

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "Rapid Open Source GPS software development for modern embedded systems: Using the GPSTk with the Gumstix". Proceedings of the 3rd ESA Workshop on Satellite Navigation User Equipment Technologies NAVITEC '2006. Noordwijk. The Netherlands. December 2006.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "The GPS Toolkit: World class open source software tools for the GNSS research community". Proceedings of the 7th Geomatic Week. Barcelona. Spain. February 2007.

Harris, R.B., Conn, T., Gaussiran, T.L., Kieschnick, C., Little, J.C., Mach, R.G., Munton, D.C., Renfro, B.A., Nelsen, S.L., Tolman, B.W., Vorce, J. and D. Salazar. "The GPSTk: New Features, Applications and Changes". Proceedings of the 20th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2007). Fort Worth, Texas. September 2007.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "Open source Precise Point Positioning with GNSS Data Structures and the GPSTk". Geophysical Research Abstracts, Vol 10, EGU2008-A-03925, 2008.

Salazar, D., Hernandez-Pajares, M., Juan, J.M. and J. Sanz. "High accuracy positioning using carrier-phases with the open source GPSTk software". Proceedings of the 4th ESA Workshop on Satellite Navigation User Equipment Technologies NAVITEC 2008. Noordwijk. The Netherlands. December 2008.

Salazar, D., Sanz-Subirana, J. and M. Hernandez-Pajares. "Phase-based GNSS data processing (PPP) with the GPSTk". Proceedings of the 8th Geomatic Week. Barcelona. Spain. February 2009.

Gaussiran, T.L., Hagen, E., Harris, R.B., Kieschnick, C., Little, J.C., Mach, R.G., Munton, D.C., Nelsen, S.L., Petersen, C.P., Rainwater, D.L., Renfro, B.A., Tolman, B.W., and D. Salazar. "The GPSTk: GLONASS, RINEX Version 3.00 and More". Proceedings of the 22nd International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2009). Savannah, Georgia. September 2009.

Finally, a research article about the EVA method is currently being prepared to be submitted to a peer-reviewed journal.

Further research

There are several research lines suggested by the development of this thesis. A list follows with the lines considered as most promising.

- The convergence time of the POP method accelerates as the number of station increases, but up to some point, as well as the improvements in the 3D-RMS error figure. Optimum or near-optimum network topology should be researched in order to guarantee a given degree of performance with the minimum use of computational resources.
- The convergence time is a problem when applying the POP to moving vehicles, specially if data arcs are short. Strategies to reduce convergence time should be a topic of future research in order to extend the usefulness of this data processing strategy.
- Accuracy and convergence time of the POP method could be greatly improved if ambiguity solving strategies could be applied to it. Recent work on PPP ambiguity fixing by [Wang and Gao, 2006], [Ge et al., 2008] and [Laurichesse et al., 2009], among others, provide a foundation that could be also applied to POP.
- Preliminary results when doing tests with the Kennedy method suggested that simpler and faster covariance models could yield equivalent results for velocity and acceleration determination. This finding should be further explored to provide better covariance models for both short and long baselines.
- Previous works such as [Serrano et al., 2004] have tried to extend the Kennedy method of velocity and acceleration determination to the real time realm, using an stand-alone receiver, broadcast ephemerides and a simple first order differentiator. The author of this thesis believes that better results could be obtained by using more advanced Infinite Impulse Response (IIR) differentiating filters and including SBAS-provided corrections into the computation.
- Work by [Kubo, 2009] showed how velocity information could be used to improve the performance of RTK integer ambiguity resolution. The author of this thesis work thinks that it would be interesting to try to fuse the POP and EVA methods with the aforementioned PPP ambiguity solving strategies to create a robust and precise post-process positioning system, able to operate thousands of kilometers from nearest reference station.

Appendix A

GNSS fundamentals

This appendix will describe the characteristics of GNSS. Given that GPS is currently the only full-operating GNSS, most of the references are related to that system. However, a great deal of the material presented here is also applicable to other GNSS systems such as GLONASS and Galileo.

A.1 Parts of a GNSS

In general, GNSS are composed of three main parts or *segments*:

- *Space Segment* : This segment includes all satellites (called SV) in charge of transmitting the ranging signals.
- *Control Segment* : It is the responsible for the proper operation of all the system, and it is in charge of several critical tasks such as:
  - To control and monitor the health and configuration of all SV's.
  - To predict SV's ephemeris and on board clocks.
  - To keep the time scale used by the system (GPST in the case of GPS).
  - To update SV's navigation messages.
- *User Segment* : This comprises the GNSS signal receivers. Its main function is to compute an estimation of Position, Velocity, Time (PVT) for the current location of the receiver.

All the former parts work together in order to provide the users the information they need, which usually is the PVT estimation. However, more sophisticated users may require additional sources of information in order to fulfill their goals.

One of this extra information sources is the International GNSS Service (IGS) , which acts as a supplemental part of the GNSS systems providing precise orbits and clocks data, Earth rotation parameters, ionospheric delay corrections, etc. [Beutler et al., 1999].

General details describing GPS, GLONASS and Galileo systems may be found in [Hernandez-Pajares et al., 2001], [Enge and Misra, 1999], [Daly, 1993] and [Hein et al., 2001].

A.2 GNSS measurements

There are several GNSS measurements, or *observables* to work with. Each one of them have several distinctive characteristics, being the noise level one of the most important ones.

In the following sections the most important observables are discussed.

A.2.1 Code measurements

In general terms, the basic observable in the current GNSS systems is the traveling time of the signal from the antenna phase center of the transmitter (satellite) to the antenna phase center of the receiver. This traveling time, Δt, is found correlating the satellite code received with a copy of it stored in the receiver. See for instance [Leick, 1995] or [Parkinson and Spilker Jr., 1996].

Once this Δt is escalated by the speed of light, the pseudorange[1] or *code-based* measurement is found:

$$P_i^j = c\Delta t = c[t_i(T_2) - t^j(T_1)] \tag{A.1}$$

In the former expression it is important to emphasize that:

- $t_i(T_2)$ is the reception time, measured according to receiver "i" clock, in the receiver's time scale (T_2).
- $t^j(T_1)$ is the transmission time, measured according to satellite "j" clock, in the transmitter's time scale (T_1).

[1]This is not the real SV-RX distance because it is altered by other effects

Both clocks have their respective *clock bias* with respect to the official time scale of the given GNSS (for instance, GPST in the case of GPS).

The term P_i^j defined in this way is taking into account the traveling time, as said before. This implies that the geometric SV-RX distance is included here, but there are also other elements that have exerted influence on traveling time as well. When taking into account those elements, the corresponding expression becomes:

$$P_i^j = \rho_i^j + c(dt_i - dt^j) + rel_i^j + T_i^j + \alpha_f I_i^j + K_{f,i}^j + M_{P,i}^j + \varepsilon_{P,i}^j \qquad \text{(A.2)}$$

Where:

- ρ_i^j: Geometric distance between SV^j and RX_i antenna phase centers.
- dt^j: Offset of SV clock with respect to GPST.
- dt_i: Offset of RX clock with respect to GPST.
- rel_i^j: Bias due to relativistic effects (linked to SV^j orbit eccentricity).
- T_i^j: Effect of the tropospheric delay.
- $\alpha_f I_i^j$: Effect due to the ionospheric delay. This effect is frequency-dependent ($\alpha_f = 40.3 \cdot 10^{16}/f^2$ when I is expressed in TECU and f is in Hz).
- $K_{f,i}^j$: Frequency- and code-dependent term due to the instrumental delays in SV and RX electronics.
- $M_{P,i}^j$: Multipath effect. It is frequency-dependent and code-dependent (depends on the the chip rate of the code used).
- $\varepsilon_{P,i}^j$: Unmodeled noise for the code measurement. It is also code-dependent and ranges in the meters.

For code-based positioning, further details regarding how to model these terms may be found in [ARINC Research Corp., 2000].

A.2.2 Carrier phase measurements

The alignment between the carrier of the received signal and the copy generated inside the RX yields another method to measure the apparent distance between SV and RX: The phase measurement [Leick, 1995].

In order to align the received signal and its copy, it is necessary to take into account the Doppler shift in the frequency. This can be achieved using Phase Lock Loop (PLL) circuits and their associated carrier loop discriminators and filters ([Hofmann-Wellenhof et al., 2008]), which are able to get phase measurements with precisions in the order of 1% of carrier cycle. Given that typical carrier frequencies for GNSS are in L-band, this implies centimeter-level (or better) precision per observation.

When the PLL locks the SV signal, it is then able to accurately measure the change in the SV-RX distance. However, it is *unable* to know how many signal cycles have elapsed *before* signal locking.

This unknown integer number of cycles is what is called the *phase ambiguity*, and it is necessary to solve it in order to be able to use the more precise phase-based measurement methods. The phase ambiguity term, denoted from now on as B_i^j, is particularly important in the context of this work.

Then, the expression for phase measurements may be written:

$$L_i^j = \rho_i^j + c(dt_i - dt^j) + rel_i^j + T_i^j - \alpha_f I_i^j + B_i^j + \omega_{L,i}^j + m_{L,i}^j + \varepsilon_{L,i}^j \quad \text{(A.3)}$$

Where the new terms with respect to Equation A.2 mean:

- $\omega_{L,i}^j$: This is the *wind-up* effect that appears in GNSS systems as GPS. In this case, the signals are circular-polarized and therefore, a relative rotation between the antennas is (erroneously) interpreted as a change in distance.
- B_i^j: The phase ambiguity term, including the carrier phase instrumental delays.
- $\alpha_f I_i^j$: Effect due to the ionospheric delay. Note that the sign is opposite than in the code-based measurements case.
- $m_{L,i}^j$: Multipath effect. This effect is much smaller than in the code-based measurements.
- $\varepsilon_{L,i}^j$: Unmodelled noise for the phase measurement. As said before, it is about 1000 times smaller than the pseudorange-based measurements noise (in the millimeter range).

As long as the receiver keeps the lock with SV signal, the phase ambiguity remains constant and it is said that observations are within a given *arc*. However, several events may produce a momentary loss of lock, and when such an event occurs the former phase ambiguity integer is no longer valid and it is said that a *cycle slip* has occurred.

A.3 Observable combinations

Given the measurements or observables presented in the former sections, it is possible to define combinations of those measurements that have different characteristics.

A.3.1 Ionospheric-free combinations

The ionospheric effect is an important one and it is difficult to model. For instance, the Klobuchar model ([ARINC Research Corp., 2000]) is able to remove, in average, just about 50% of this effect.

Hence, the GNSS's are designed to work with at least two frequencies[2]. As stated in A.2.1, the ionospheric effect depends on the frequency squared ($\alpha_f = 40.3/f_f^2$). Taking this into account, it is possible to cancel out this effect both in code (P) and phase (L) measurements using the following combinations of measurements for two different frequencies:

$$PC = \frac{f_1^2 P_1 - f_2^2 P_2}{f_1^2 - f_2^2} \tag{A.4}$$

$$LC = \frac{f_1^2 L_1 - f_2^2 L_2}{f_1^2 - f_2^2} \tag{A.5}$$

These combinations have an equivalent wavelength different than the original signals. For instance, in the case of GPS $\lambda_c =$ 10.7 cm, instead of the original 19.03 cm (λ_1) and 24.42 cm (λ_2).

Also, it is important to emphasize that the ambiguity of this combination for the phase case is no longer an integer number of λ_c wavelengths.

[2]In the case of GPS, however, the second frequency is reserved for military uses.

A.3.2 Ionospheric combinations

These combinations cancel the non-dispersive effects and leave just the ionospheric effect and the instrumental delays:

$$PI = P_2 - P_1 \tag{A.6}$$

$$LI = L_1 - L_2 \tag{A.7}$$

A.3.3 Narrow-lane and wide-lane combinations

The expressions for these combinations are:

$$P\delta = \frac{f_1 P_1 + f_2 P_2}{f_1 + f_2} \tag{A.8}$$

$$LW = \frac{f_1 L_1 - f_2 L_2}{f_1 - f_2} \tag{A.9}$$

The *wide-lane* (LW) combination has the advantage of yielding an observable with a relatively long wavelength ($\lambda_w = 86.2cm$ in the GPS case). This is very useful for cycle slip detections.

On the other hand, more modern GNSS (Galileo, modernized GPS) will provide Extra Wide Lane (EWL) combinations, with an equivalent wavelength of about 5.9 meters [Hernandez-Pajares et al., 2003a].

A.3.4 Melbourne-Wübbena combination

This combination is used along the wide-lane in order to detect cycle slips and to estimate the wide-lane ambiguity, as well. Its equation follows:

$$W = LW - P\delta \tag{A.10}$$

A.4 Solving the navigation equations

Each satellite in view from a given receiver yields several equations as A.2 and A.3, depending on the number of frequencies used an the capability of the receiver to supply phase measurements.

Taking as example code measurements, equation A.2 may be rewritten as:

$$P_i^j + cdt^j - rel_i^j - T_i^j - \alpha_f I_i^j - Kf_i^j = \rho_i^j + cdt_i + M_{P,i}^j + \varepsilon_{P,i}^j \qquad \text{(A.11)}$$

Given that:

$$\rho_i^j = \sqrt{(x - x^j)^2 + (y - y^j)^2 + (z - z^j)^2} \qquad \text{(A.12)}$$

Therefore equation A.11 is not linear. If ρ is linearized around a reference point (x_0, y_0, z_0), that represents an approximate position for receiver[3], then equation A.11 becomes:

$$prefit_i^j = \frac{x_0 - x^j}{\rho_0^j} dx + \frac{y_0 - y^j}{\rho_0^j} dy + \frac{z_0 - z^j}{\rho_0^j} dz + cdt_i + M_{P,i}^j + \varepsilon_{P,i}^j \quad \text{(A.13)}$$

Where the *prefilter residual* $(prefit_i^j)$ represents the difference between the measurement and the modeled terms, the terms $\frac{x_0 - x^j}{\rho_0^j}$, $\frac{y_0 - y^j}{\rho_0^j}$, $\frac{z_0 - z^j}{\rho_0^j}$ are the unity vectors pointing from SV^j to receiver's *a priori* position (x_0, y_0, z_0), and dx, dy, dz are the differences between the real RX coordinates and the a priori position.

If the terms making up the prefit residual where properly measured/modeled, in general it can be supposed that biases are small respect to noise levels and therefore noise terms $(M_{P,i}^j, \varepsilon_{P,i}^j)$ have zero mean. If such conditions apply, the set of n available equations may be written in matrix form:

[3]This point may be found in several ways, being the Bancroft method a typical one. See [Bancroft, 1985] and [Yang and Chen, 2001] for references.

$$\begin{bmatrix} prefit^1 \\ \vdots \\ prefit^n \end{bmatrix} = \begin{pmatrix} \frac{x_0-x^1}{\rho_0^j} & \frac{y_0-y^1}{\rho_0^j} & \frac{z_0-z^1}{\rho_0^j} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ \frac{x_0-x^n}{\rho_0^j} & \frac{y_0-y^n}{\rho_0^j} & \frac{z_0-z^n}{\rho_0^j} & 1 \end{pmatrix} \begin{bmatrix} dx \\ dy \\ dz \\ cdt_i \end{bmatrix} \tag{A.14}$$

Please note that SV's clock offsets are taken as terms that can be modeled, but RX clock offset is treated as an unknown. Also, it is taken for granted that SV's positions are somehow known.

The system presented in A.14 is, in general, overdimensioned[4] and incompatible. It may be solved using several methods such as LMS, WMS or the Kalman filter (please see [Bierman, 1977] or [Hernandez-Pajares et al., 2001] for further details).

A.5 Differential positioning

If instead of one receiver there are two receivers available, and the second RX has fixed, known coordinates and is located not too far away from the first one, then a significant improvement in positioning may be achieved.

This is due to the fact that several of the errors are common to both receivers (SV ephemeris and clocks, ionosphere, troposphere to some extent, etc.). Then, the receiver with known coordinates (named from now on *reference receiver*) is able to compute range corrections for each satellite in view and most of the common errors are canceled out.

Therefore, equation system A.14 may be expanded using the additional information provided by the reference receiver. This is called *differential positioning*.

For a reference RX with known coordinates the equivalent equation to A.13 may be written:

$$prefit_i^j = cdt_i + M_{P,i}^j + \varepsilon_{P,i}^j \tag{A.15}$$

Hence, subtracting A.15 from A.13 yields:

[4]Usually, there are more than 4 SV's in view

$$\Delta prefit_i^j = \frac{x_0 - x^j}{\rho_0^j}dx + \frac{y_0 - y^j}{\rho_0^j}dy + \frac{z_0 - z^j}{\rho_0^j}dz + \Delta(cdt) + \Delta M_{P,i}^j + \Delta\varepsilon_{P,i}^j \tag{A.16}$$

As said before, in the former equation the most common errors of term $\Delta\varepsilon_{P,i}^j$ should have canceled out. On the other hand, take into account that multipath errors are strictly local, and no reduction may be achieved using this method.

Like in the case of A.14, if the error is supposed to have zero mean, the following equation system can be built and solved with the same methods mentioned before:

$$\begin{bmatrix} \Delta prefit^1 \\ \vdots \\ \Delta prefit^n \end{bmatrix} = \begin{pmatrix} \frac{x_0-x^1}{\rho_0^j} & \frac{y_0-y^1}{\rho_0^j} & \frac{z_0-z^1}{\rho_0^j} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ \frac{x_0-x^n}{\rho_0^j} & \frac{y_0-y^n}{\rho_0^j} & \frac{z_0-z^n}{\rho_0^j} & 1 \end{pmatrix} \begin{bmatrix} dx \\ dy \\ dz \\ \Delta(cdt) \end{bmatrix} \tag{A.17}$$

Please take into account that in this case just the *relative difference* between receivers' clocks biases ($\Delta(cdt)$) is estimated, and not the bias relative to the GNSS system time.

A.6 Double differences positioning

It is possible to take method explained in A.5 one step further and take not only a reference station, but also a *reference satellite* (usually the SV with the highest elevation at the given epoch).

In order to present this method, the following notation is introduced:

$$\begin{aligned} &\Delta\diamond^\bullet \equiv \diamond_{RX}^\bullet - \diamond_{RefRX}^\bullet \\ &\nabla\diamond_\bullet \equiv \diamond_\bullet^j - \diamond_\bullet^k \\ &\Delta\nabla\diamond \equiv \nabla\diamond^j - \nabla\diamond^R = \Delta\diamond_{RX} - \Delta\diamond_{RefRX} \end{aligned}$$

Whereas the single differences between receivers (Δ) tend to cancel out common terms associated with the satellite (SV clock bias, ephemeris errors, atmospheric propagation...), and the single differences between satellites tend to cancel out common errors associated with receivers (implying $\Delta(cdt) = 0$), therefore the corresponding equation system may be written as:

$$\begin{bmatrix} \Delta\nabla prefit^1 \\ \vdots \\ \Delta\nabla prefit^n \end{bmatrix} = \begin{pmatrix} \nabla\left[\frac{x_{RX,0}-x^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^1}{\rho^1_{RX,0}}\right] \\ \vdots & \vdots & \vdots \\ \nabla\left[\frac{x_{RX,0}-x^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^n}{\rho^n_{RX,0}}\right] \end{pmatrix} \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \tag{A.18}$$

Where terms like $\nabla\left[\frac{z_{RX,0}-z^j}{\rho^j_{RX,0}}\right]$ mean the single difference between SV j-th and the one took as reference SV.

The same techniques mentioned before are applicable in this case to solve the equation system.

A.7 Ambiguity resolution

A relatively simple way of dealing with ambiguity resolution is to extend the method presented in Section A.6 to also include phase measurements. In this case, the (double-differenced) phase ambiguities will be added to the vector holding the unknowns, resulting in the equation system at A.19.

Such equation system may be solved using a Kalman filter. Each double-differenced ambiguity $\Delta\nabla B^i$ may be considered as constant as long as no cycle slip happens, and as white noise at the epoch when a cycle slip appears. This way of ambiguity resolution is often called "*floated ambiguities*".

The former strategy not only works for double-differenced ambiguities, but also for single-differenced and undifferenced ambiguities. The later case is what is used in Precise Point Positioning (PPP) data processing.

A.8 Tides modeling

In precise GNSS data processing using undifferenced equations, such as PPP, a good modeling of phenomena such as tidal effects caused by solid tides, ocean loading tides and pole movement-induced tides is a must.

The GPSTk supplies several classes to manage tidal effects, providing the respective correction vectors in an unified format (class `Triple`). More information

$$\begin{bmatrix} \Delta\nabla prefit(P)^1 \\ \Delta\nabla prefit(L)^1 \\ \vdots \\ \Delta\nabla prefit(P)^n \\ \Delta\nabla prefit(L)^n \end{bmatrix} = \begin{pmatrix} \nabla\left[\frac{x_{RX,0}-x^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^1}{\rho^1_{RX,0}}\right] & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \nabla\left[\frac{x_{RX,0}-x^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^1}{\rho^1_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^1}{\rho^1_{RX,0}}\right] & 0 & \cdots & \underbrace{1}_{k} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & & & & & & \\ \nabla\left[\frac{x_{RX,0}-x^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^n}{\rho^n_{RX,0}}\right] & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \nabla\left[\frac{x_{RX,0}-x^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{y_{RX,0}-y^n}{\rho^n_{RX,0}}\right] & \nabla\left[\frac{z_{RX,0}-z^n}{\rho^n_{RX,0}}\right] & 0 & \cdots & 0 & \cdots & \underbrace{1}_{l} & \cdots & 0 \end{pmatrix} \begin{bmatrix} dx \\ dy \\ dz \\ \Delta\nabla B^1 \\ \vdots \\ \Delta\nabla B^k \\ \vdots \\ \Delta\nabla B^l \\ \vdots \\ \Delta\nabla B^s \end{bmatrix} \tag{A.19}$$

about the current GPSTk implementation follows.

A.8.1 Solid tides

For the solid tides the model used is the simple quadrupole response model described by [Williams, 1970] and implemented in the GIPSY/OASIS software ([Sovers and Border, 1990]).

In this model, and assuming a phase lag $\phi = 0$ degrees[5], the tidal displacement vector δ in a topocentric Up-East-North (UEN) reference frame may be found by the sum of the perturbations from several perturbing sources s:

$$\delta = \sum_s [hg_{1s}, lg_{2s}, lg_{3s}]^T \tag{A.20}$$

Being:

$$g_{1s} = \frac{3\mu_s r_p^2}{R_s^5}\left[\frac{(\mathbf{r_p}\cdot\mathbf{R_s})^2}{2} - \frac{r_p^2 R_s^2}{6}\right] \tag{A.21}$$

$$g_{2s} = \frac{3\mu_s r_p^2}{R_s^5}(\mathbf{r_p}\cdot\mathbf{R_s})(Y_s x_p - X_s y_p)\frac{|\mathbf{r_p}|}{\sqrt{x_p^2 + y_p^2}} \tag{A.22}$$

$$g_{3s} = \frac{3\mu_s r_p^2}{R_s^5}(\mathbf{r_p}\cdot\mathbf{R_s})\left[Z_s\sqrt{x_p^2 + y_p^2} - \frac{z_p}{\sqrt{x_p^2 + y_p^2}}(X_s x_p - Y_s y_p)\right] \tag{A.23}$$

Where:

- h and l are the Love numbers.
- μ_s is the ratio of the mass of the disturbing object s to the Earth mass.
- $\mathbf{R_s} = [X_s, Y_s, Z_s]^T$ is the vector to the disturbing object s, given in an Earth-Centered, Earth-Fixed (ECEF) reference frame.
- $\mathbf{r_p} = [x_p, y_p, z_p]^T$ is the vector to the location of interest (GPS receiver position), given in an ECEF reference frame.

[5] This is a common assumption for these models.

A.8.2 Ocean loading

The variation in ocean mass distribution due to tides, and the associated load on the Earth crust, produce time-varying deformations on the Earth surface that may reach up to 0.1 m ([IERS, 2009]) and is called "*ocean loading*".

According to [IERS, 2009], the ocean loading at a particular place due to a given tidal harmonic is computed integrating the tide height with a weighting function (Green's function), and the total loading may be obtaining by summing the effect of all harmonics, as shown by Equation A.24.

$$\delta = \sum_j A_{cj} \cos(\chi_j(t) - \phi_{cj}) \tag{A.24}$$

Where δ is the displacement in a given direction, A_{cj} and ϕ_{cj} are respectively the amplitudes and phases of the loading response for the position being studied, and $\chi_j(t)$ is an astronomical argument computed from the main 11 tides and that can be computed using the IERS-provided `ARG.f` subroutine.

The main 11 tides considered are the semidiurnal waves M_2, S_2, N_2 and K_2, the diurnal waves K_1, O_1, P_1 and Q_1, and the long-period waves M_f, M_m and S_{sa} ([IERS, 2009]). Those tides may be found, for several models, from the Scherneck's free ocean loading provider service ([Scherneck, 1991]) at:

`http://www.oso.chalmers.se/ loading/`

Amplitudes and phases for other tidal components may be obtained interpolating from the main eleven, and the IERS currently proposes using 342 constituent tides found by spline interpolation. If only the main components are used (the GPSTk current approach), errors may go up to 5 mm RMS at high latitudes.

A.8.3 Pole tides

The movement of the instantaneous Earth rotation pole generates Earth crust deformations, called "*pole tides*", that may produce GNSS receiver displacements up to 25 mm in the radial direction, and up to 7 mm in the horizontal plane.

Given a location with longitude λ and co-latitude θ, the IERS conventions ([IERS, 2009]) propose the expressions at Equations A.25 to A.27 in order to compute the displacements in the “Up” (S_r), “South” (S_θ) and “East” (S_λ) directions, in meters.

$$S_r = -0.033\sin(2\theta)(m_1\cos(\lambda) + m_2\sin(\lambda)) \tag{A.25}$$

$$S_\theta = -0.009\cos(2\theta)(m_1\cos(\lambda) + m_2\sin(\lambda)) \tag{A.26}$$

$$S_\lambda = 0.009\cos(\theta)(m_1\sin(\lambda) - m_2\cos(\lambda)) \tag{A.27}$$

Where terms m_1 and m_2 are the secular variations of Earth’s mean rotation pole computed from the IERS-provided polar motion variables (x_p, y_p) and the corresponding running averages $\overline{x}_p$ and $\overline{y}_p$ (all in arcseconds), as shown in Equation A.28.

$$m_1 = x_p - \overline{x}_p, \qquad m_2 = -(y_p - \overline{y}_p) \tag{A.28}$$

In order to estimate the averages it is used the IERS conventional mean pole linear model in Equations A.29 and A.30, where t is in calendar years, t_0 is year 2000.0, and the variations rates are in arcseconds per year.

$$\overline{x}_p(t) = \overline{x}_p(t_0) + (t - t_0)\dot{\overline{x}}_p(t_0) \tag{A.29}$$

$$\overline{y}_p(t) = \overline{y}_p(t_0) + (t - t_0)\dot{\overline{y}}_p(t_0) \tag{A.30}$$

$$\overline{x}_p(t_0) = 0.054, \qquad \dot{\overline{x}}_p(t_0) = 0.00083$$

$$\overline{y}_p(t_0) = 0.357, \qquad \dot{\overline{y}}_p(t_0) = 0.00395$$

Appendix B

C++ basics

In order to understand what is the GPSTk and its characteristics, it is necessary to explain some characteristics of the C++ programming language. Skip this appendix if you already have knowledge about C++.

B.1 C++ Basics

The C++ programming language was developed by Bjarne Stroustrup and its first official version was used in 1983 by American Telephone and Telegraph (AT&T) [Guerin, 2005]. Stroustrup's intention was to keep C language concepts but adding better type checking, data abstraction and Object-Oriented Programming (OOP) [Stroustrup, 2006a]. In this sense, C++ may be seen as a kind of *superset* of C.

One of the advantages of C++ is that it is a modern language, emphasizing in clean and structured code writing. It is also very portable given that since 1997 there is an ANSI/International Organization for Standardization (ISO) C++ standard in place: ISO/International Electrotechnical Commission (IEC) 14882 [Stroustrup, 2006b]. The last revision of the standard was issued in 2004.

In the following sections some of the most relevant characteristics of C++ are briefly explained.

B.1.1 Object-oriented programming

OOP is a style of programming that relays in the concepts of inheritance, encapsulation and polymorphism. In the context of C++ this means programming using *classes* [Stroustrup, 2006a].

A *class* may be defined as an abstraction that encapsulates certain types of data (*fields*) and the operations (*methods*) that apply on that data. The concept of method is akin to C functions, but focusing in acting on class data, and not on arbitrary data [Karniadakis and Kirby, 2005].

Classes represent ideas and concepts, thus helping to organize code. Given that classes *encapsulate* data and operations on that data, the user of a well defined class is presented with a clean and simple interface.

On the other hand, the developer's task is eased because the implementation of the class is hidden from users, implying that changes in implementation can be made without interfering with the rest of the code.

The following example, adapted from the GPSTk, shows how to declare a class named *Triple* to represent three-dimensional vectors. The implementation is not shown, just the declaration:

```
class Triple
{
public:

      // Default constructor. Initializes triple as (0,0,0).
   Triple();

      // Construct from three doubles.
   Triple(double a, double b, double c);

      /// Destructor
   virtual ~Triple();

      // Computes the Magnitude of this vector
   double mag();

      // Computes Dot Product of this Triple and "otherTriple"
   double dot(Triple otherTriple);

      // Computes Cross Product of this Triple and "otherTriple"
   Triple cross(Triple otherTriple);

}; // End of class Triple
```

If a class is an abstract representation of a concept, then the programmer needs an specific way to use or *instantiate* a real-life example of that class. Such an example is called an *object* (from here the term OOP) and the way to create an object is using a special method called *Constructor*.

In the former example, the *Triple* objects are constructed using three doubles, and therefore, a *Triple* object named *receiverLocation* may be declared this way:

```
    Triple receiverLocation( 23450.0, 32124.5, 98403.1 );
```

Please note that, for all practical purposes, declaring a new class is equivalent to *creating a new data type* (look at how we put "Triple" in front of "receiverLocation"). So, even though *Triple* encapsulates data *and* operations, it is at an equivalent level as classic plain data: doubles, integers, chars, etc.

Each time an object is instantiated, computational resources are allocated. When the object is no longer needed, those resources should be freed. That is the task of another special method called *Destructor*, and denoted with the same name of the constructor, but prefixed with $\sim$.

In the former example class, some useful methods for working with triples have been added: *mag()* computes the magnitude, *dot()* calculate the scalar product of two triples, and *cross()* computes their vectorial product.

A way to use these methods would be:

```
  double sizeA, dotAB;
  Triple rxA( 350.0, 214.2, 98.4 );
  Triple svB( 3340.0, 4323.5, 1803.1 );
  Triple result();

  sizeA = rxA.mag();       // Get rxA magnitude an put it in sizeA
  dotAB = rxA.dot(svB);    // Compute dot product
  result = rxA.cross(svB); // Get cross product between rxA and svB,
                           // store it in "result" object (a Triple)
```

Please take note of the *object.member* notation to gain access to class members.

B.1.2 Inheritance

In the former sections it was asserted that C++ emphasizes structured coding and at the same time eases developer's work. Inheritance is a way to achieve this.

Class *Triple* allows representation of tridimensional vectors. Now, let's suppose that some of those vectors may represent a *position* on a geodetic reference ellipsoid. This new abstraction is related with Triples, but adds new characteristics. Then, it is possible to create a new *Position* class that is a "child" of, or *inherits* from, class *Triple*.

```
// This class is adapted from Position class in GPSTk
#include "Triple.hpp"       // File with declaration of Triple
#include "GeoidModel.hpp"   // File with declaration of GeoidModel

  class Position : public Triple
  {
  public:

      // Constructor from doubles
    Position(double a, double b, double c, GeoidModel *geoid=NULL);

      // Constructor from Triples
    Position(Triple ABC, GeoidModel *geoid = NULL)

      // Destructor
    ~Position();


    ///////////////// Extra Methods  /////////////////////

      // Get geodetic latitude.
    double getGeodeticLatitude();

      // Get geodetic longitude.
    double getGeodeticLongitude();

   }; // End of class Position
```

As can be seen, the new class *Position* is a descendant from *Triple* but takes into account a new parameter: The model of the geoid the position is given on. Also, it adds two new methods that are only related to positions: *getGeodeticLatitude()* and *getGeodeticLongitude()*.

Take into account that that *Position* inherits from *Triple* all the *Triple* methods: *mag()*, *dot()*, etc.

In this way, inheritance has helped to achieve two key goals:

- The developer is forced to think in advance about the abstractions he or she is working on, and the different relations among them. This tends to produce more logic code.
- Given that inherited methods (as *mag()*, for example) are declared and encapsulated in the parent classes, if any of them needs to be modified, it will be changed only once in the original class implementing it, and the changes will be automatically transferred to the children classes.

B.1.3 Polymorphic methods

Another useful characteristic of C++ is that it accepts *polymorphism*. This means that a single method may have many forms, each one with a different set of input parameters. The compiler will automatically determine which instance of the method is called according to those parameters.

An example of polymorphism was presented in the former section in the *Position* constructor: It may be called using three doubles and a GeoidModel, or with a Triple and a GeoidModel:

```
  // Constructor from doubles
Position(double a, double b, double c, GeoidModel *geoid = NULL);

  // Constructor from Triples
Position(Triple ABC, GeoidModel *geoid = NULL)
```

B.1.4 Operator overloading

A powerful yet somewhat strange feature of C++ is operator overloading. This implies that a given class may *redefine* the way common operators (i.e.: +, -, *, =, etc.) behave when applied to it.

For instance, let's overload the + operator for *Triple* class:

```
Triple Triple::operator+(Triple otherTriple)
{
  Triple temp;
  temp.theArray = this->theArray + otherTriple.theArray;
  return temp;
}
```

Several explanations are in order regarding this example:

- The notation *Triple::operator+* means that we are implementing the operator "+" from "Triple" class.
- The field named *theArray* is an internal field from the Triple class holding the three doubles. It did not appear before because the implementation of Triple has not yet been shown.

- *this* is a special pointer pointing to the current object. When using objects pointers instead of plain objects, access to members is done with -> instead of dot (.).

From now on, the following code is perfectly correct:

```
Triple rxA( 350.0, 214.2, 98.4 );
Triple svB( 3340.0, 4323.5, 1803.1 );
Triple result();

result = rxA + svB;
```

Remember that thanks to inheritance, the overloaded "+" operator of class *Triple* is also enjoyed by class *Position*[1].

B.1.5 Templates

Often, a given class has members that need to be of different types in different situations. Sometimes using polymorphism may be a solution, but it is not always the case [Stephens et al., 2006].

What the developer needs is a class that allows to choose the types to work on when it is instantiated, instead of having them fixed. This is achieved with *Class Templates*. With Templates, the developer writes the class with placeholders instead of types. In that way, the user is allowed to choose which type to work with.

Let's suppose we have a class named *Vector*, and we want to be able to use Vector objects with doubles, integers, etc. Then, the Vector class may be defined as:

```
template <class T>
class Vector
{
public:

    // Default constructor
  Vector();
```

[1]This is a simple example and, as such, an overloaded "+" operator may not have sense for *Position* class. However, an overloaded equality operator (==), taking tolerance as an extra parameter, may be very useful for both classes.

```
    // Constructor giving an initial size and default value
    // for all elements.
  Vector(integer size, T defaultValue);

    ///// ... More fields and methods here ...  /////

}; // End of class Vector
```

The line *template* <*class T*> just at the start of class declaration indicates that this is not an ordinary class, but one that will work with the type *T*, supplied by the user.

Using this feature, it is allowed to write:

```
Vector <bool>     boolVect;  // Vector of booleans elements
Vector <integer>  intVect;   // Vector of integers
Vector <double>   doubVect;  // Vector of doubles
Vector <Position> posVect;   // Vector of Position objects
```

Templates offer much flexibility at the expense of code more difficult to read.

B.1.6 Exception handling

During the execution of a program it may appear exceptional circumstances, errors or anomalies (such as numerical overflow, division by zero, memory unavailable, etc.) that hamper the normal behavior of the program. In C++, exceptions provide a way to react to those circumstances [Soulie, 2006].

Exceptions transfer the control to *handlers*, special functions designed to deal with the problem. In order to be able to *catch* exceptions, the code under watch must be enclosed in a *try* block. On the other hand, handlers are enclosed in *catch* blocks.

Each time an exception occurs, it must be *thrown*. An example follows:

```
try {

    // Potential problem here, throwing an integer
  if (problemWithInteger()) throw 20;

    // Potential problem here, throwing a char
  if (problemWithChar()) throw 'z';
} // End of try block. Handlers follow
catch(int parameter) { printf("Integer exception."); }
catch(char parameter) { printf("Char exception."); }
catch(...) { printf("Other exception."); }
```

As can be seen, the *throw* keyword is in charge of throwing the exception. Then, processing flow will jump to the proper handler. The ellipsis (...) are used in the last *catch* block as a default exception, in case a problem arose but no other handler caught the exception.

Given that *thrown* is able to use several different data types, the C++ Standard Template Library (STL) [Silicon Graphics, Inc., 2006] supplies an specific base class called *exception* designed to declare objects to be thrown as exceptions.

This provides a lot of flexibility when handling exceptions in programs. For instance, the GPSTk takes advantage of this fact and declares specific exception classes to deal with different kinds of problems. In this way, we find classes such as *InvalidTropModel*, *OutOfMemory*, *NoEphemerisFound*, *InvalidDOP*, etc.

B.1.7 C++ summary

As a summary of the former sections the following may be said regarding C++:

- It is a portable, modern and standards-based programming language.
- Programming with C++ tends to produce an organized, easy to maintain code.
- It is powerful and very flexible language.
- Flexibility makes C++ a complex language.
- C++ syntaxis may look "esoteric" for the unaware programmer, particularly if templates are used.

Appendix C

GPSTk basics

This appendix presents a simple introduction to the GPSTk, including short programs and their results.

C.1 GPSTk overview

The GPSTk is both a set of *libraries* to write GNSS software, and a suite of example *applications*. It is "Open Software", released under the GNU Lesser General Public License (LGPL), allowing to develop both non-commercial *and* commercial applications.

In particular, the LGPL license means that:

- The original code belonged to the Advanced Research Laboratory (ARL) of the University of Texas (ARL:UT), but it was later released to the public.
- New features *added to the library* are property of their authors, but them must be also released as LGPL.
- New software developed *taking advantage of* the GPSTk library (**linking**) is property of their authors.

Initiated at ARL:UT, the GPSTk now also includes several official developers around the world. The main place where development efforts are coordinated is the project website at http://www.gpstk.org.

The GPSTk is written in ISO-standard C++, following object-oriented principles. This approach eases maintenance and extensibility, lowering overall programming costs.

Also, the ISO-standard C++ provides a great deal of portability to the project, both from the operative system point of view (it works on Microsoft Windows, as well as Linux, Solaris, Macintosh OS X, AIX, etc.) and from the hardware platform point of view (it works in big servers and small embedded systems, both 32 and 64 bits).

Last but not least, the GPSTk is very well documented thanks to the use of the `Doxygen` documentation system, providing a very complete API.

C.2 Some current GPSTk features

It follows a brief list of some current features provided by the GPSTk and its accessory libraries:

- Time handling and conversions.
- RINEX files reading/writing:
  - Observation.
  - Ephemeris.
  - Meteorological.
- Ephemeris computation both in broadcast and SP3 formats.
- Mathematical tools: Matrices, vectors, interpolation, numeric integration, Least Mean Squares (LMS) and Weighted-Least Mean Squares (WMS) solvers, extended Kalman filters, etc.
- Application development support:
  - Exceptions handling.
  - Command line framework.
  - Configuration files management.
- Cycle slip detection and correction.
- Code positioning, Receiver Autonomous Integrity Monitoring (RAIM), Differential GPS (DGPS) and Precise Point Positioning (PPP) support.
- Several tropospheric models like Saastamoinen, Goad-Goodman, New Brunswick, Niell, etc.
- Support for Klobuchar ionospheric model as well as Ionosphere Map Exchange (IONEX) files.

- Classes for precise modeling: Antenna phase centers (relative and absolute), Antenna Exchange Format (ANTEX) files, wind-up effects, gravitational delay, etc.
- Tidal models: Solid tides, ocean loading, pole tides.
- Probabilistic functions (normal, chi-square, etc.) and stochastic models.
- Run-time programmable solvers.
- Advanced GNSS Data Structures (GDS) for data processing and management.

C.3 GPSTk advantages

The use of the GPSTk in GNSS-related projects will provide numerous advantages that can be summarized as follows:

- Programmers don't have to "reinvent the wheel":
  - The programmer doesn't have to use his time to program, test and debug common, non-interesting routines (like RINEX parsing, for instance).
  - The programmer uses his time to learn and experiment.
  - The programmer uses his time to develop new techniques.
- The GPSTk can be trusted: Its performance has been validated with other state-of-the-art GNSS data processing software suites.
- It includes data management facilities that support clean and easy to maintain source code.
- The GPSTk is open-source, and therefore:
  - It is excellent for learning how complex algorithms work.
  - It eases reimplementation in other languages and/or hardware.

C.4 GPSTk disadvantages

As convenient as it may be, the GPSTk also has some disadvantages with respect to other ways to tackle GNSS data processing problems. Some of those disadvantages are:

- The programmer needs C++ knowledge to use it effectively.
- The C++ language is very flexible and powerfull, but it may become complex and “exotic”.
- The compilation process is slower and more complex than when using interpreted languages.
- Many engineers are more confortable with tools like MATLAB (although they are **very** slow compared with a compiled GPSTk-based program).

C.5 How to compile the GPSTk

In order to compile the GPSTk, it is needed:

- A C++ compiler like `g++`, Microsoft Visual Studio C++ .NET 2003 (Version 7), Microsoft Visual C++ Express 2005 (Version 8), Forte Developer, IBM VisualAge, etc.
- A compilation system such as `jam` or `make`[1].

The compilation process is relatively simple[2], and detailed instructions for different operative systems are provided in the project website (http://www.gpstk.org). In general, it is composed of four steps:

- Download the stable source code from GPSTk website, or the last development version from subversion repository.
- Decompress if needed.
- Compile. The `jam` process is currently the easiest way.
- After compiling, install the library and accessory files in the system.

In a Linux/UNIX platform with the `jam` tool installed, once the corresponding `tar.gz` ball is downloaded the specific compilation steps are:

```
$ tar xvzf gpstk.tar.gz
$ cd gpstk
```

[1]Note that some compiler suites provide their own compilation system. In such cases, the `jam` or `make` instructions must be adapted to the compiler suite used.

[2]Besides, there are pre-compiled binaries available for selected platforms.

```
$ jam
$ su
# jam install
```

C.6 GPSTk examples

In the following sections the source code of some simple GPSTk-based programs is presented, in order to provide an initial help to those new to GPSTk and C++.

C.6.1 Vectors

The following example explains how to use the GPSTk-provided `Vector` class.

```
#include <iostream>
#include <cmath>
#include "Vector.hpp"

using namespace std;
using namespace gpstk;

int main()
{

   cout << fixed << setprecision(1);   // Set output format

   Vector<double> vect1(4);
   Vector<double> vect2(4);

   for( int i = 0; i < 4; ++i )
   {
      vect1(i) = 3.0 + i;
      vect2(i) = 5.0 - 2.5*i;
   }

   Vector<double> vect3 = vect1 + vect2;

   cout << "My first GPSTk program using Vector class." << endl;
   cout << vect1 << endl;
   cout << vect2 << endl;
   cout << vect3 << endl;

   exit(0);

}
```

In this example, lines #1 and #2 include standard C++ libraries dealing with output (`iostream`) and mathematics (`cmath`), while line #3 includes the non-standard, GPSTk-provided `Vector` class.

After that, lines #4 and #5 sets the "namespaces" to be used. The practical consequence of these lines is that any class name not found by the compiler, will be looked for in the "standard" and "gpstk" namespaces.

The program starts in line #6 and ends in line #22. The initial action is to set a proper output format, instructing the "standard output" object `cout` to use fixed precision with one decimal place (line #8).

The first use of `Vector` appears in lines #9 and #10. This class takes advantage of the "template" characteristics provided by the Standard Template Library (STL), and therefore the `Vector` declarations must include the type of objects that the new `Vectors` will contain (in this case, they are made up of `double` precision numbers). Both objects, `vect1` and `vect2`, will be composed of 4 elements, and lines #11 to #15 just initialize them with some data.

Then, line #16 will declare a third `Vector` called `vect3`, that is composed of the sum of `vect1` and `vect2`. Note how the "sum" operator (+) is *overloaded*, i.e., it is redefined to properly take care of the sum between `Vector`s.

Next, lines #17 to #20 print out the results using the `iostream` object `cout`. First, line #17 prints a string followed by a new line (object `endl`), and the other lines print the vectors, one line each. Again, note how the operator (<<) is *overloaded* to allow simple printing of `Vector` objects. Finally, line #21 ends the program, returning "0" to the operative system.

The output of this program is as follows:

```
My first GPSTk program using Vector class.
 3.0 4.0 5.0 6.0
 5.0 2.5 0.0 -2.5
 8.0 6.5 5.0 3.5
```

There are several ways to compile the former program, depending on the compiler used. For instance, in a Linux environment this program should be compiled with a single command line[3] such as:

```
  $ g++ -Wall -ansi -pedantic test-vector.cpp -o test-vector
        -I/usr/local/include/gpstk/ -L/usr/local/lib/ -lgpstk -lm
```

[3]The compilation command line is a single line, although it may appear as two lines because of typesetting constraints.

Where:

- `g++` is the name of the GNU compiler,
- `-Wall`, `-ansi` and `-pedantic` are compiler flags to show all compilation warnings and enforce strict ANSI compliance,
- `test-vector.cpp` and `test-vector` are the names of the input source and output binary files, respectively,
- `-I/usr/local/include/gpstk/` is the default place where GPSTk *include* or *header* (`*.hpp`) files are installed,
- `-L/usr/local/lib/` is the default place where GPSTk library (and accessory libraries) are installed, and
- `-lgpstk` and `-lm` are the flags to call GPSTk and mathematical libraries, respectively.

C.6.2 Matrices

This example presents the GPSTk-provided `Matrix` class. Lines #1 to #8 are very similar to the `Vector` example, but in this case the GPSTk-provided `Matrix` class is included, and the output configured to use fixed precision with 3 decimal places.

Lines #9 to #12 define the matrices to be used: `Matrix A` has a size of 3x3 and all its elements have a value of 5.0, `Matrix I` is an identity matrix of the same size as `A`, while `Matrix B` is a linear combination of matrices `A` and `I` (note the *overloading* of operators "`-`" and "`*`").

`Matrix C` is declared with size 3x3 but its elements are left undefined. Afterwards, lines #13 to #19 will input specific values for the elements of this matrix.

Then, line #20 defines a new `Matrix`, `CT`, as the transpose of `Matrix C`, while line #21 defines `Matrix D` es the inverse of the product of `C` and its transpose `CT`.

Finally, lines #22 to #27 print the results (again, note the overloading of operator <<), and line #28 ends the program.

```
#include <iostream>
#include <cmath>
#include "Matrix.hpp"

using namespace std;
using namespace gpstk;

int main()
{

   cout << fixed << setprecision(3);   // Set output format

   Matrix<double> A(3, 3, 5.0);
   Matrix<double> I = ident<double>(3);
   Matrix<double> B = A - 2.0*I;
   Matrix<double> C(3, 3);

   for( int row = 0; row < 3; ++row  )
   {
      for( int col = 0; col < 3; ++col  )
      {
         C(row, col) = row+0.9*B(row,col);
      }
   }

   Matrix<double> CT = transpose(C);
   Matrix<double> D = inverseChol( CT*C );

   cout << A  << endl << endl;
   cout << I  << endl << endl;
   cout << B  << endl << endl;
   cout << C  << endl << endl;
   cout << CT << endl << endl;
   cout << D  << endl << endl;

   exit(0);

}
```

C.6.3 Solvers

The GPSTk-provided solver classes are very useful when combined with `Vector` and `Matrix` objects. The following example shows how to use a Least Mean Squares (LMS) solver.

Lines #1 to #8 are very similar to the former examples, but including the GPSTk-provided `SolverLMS` class. This class in turn needs `Vector` and `Matrix` classes to work so those classes are included from `SolverLMS`. That is the reason because those class are also available from within our example without explicit inclusion.

```
 1  #include <iostream>
 2  #include <cmath>
 3  #include "SolverLMS.hpp"

 4  using namespace std;
 5  using namespace gpstk;

 6  int main()
 7  {

 8     cout << fixed << setprecision(3);   // Set output format

 9     Vector<double> vect(3, 0.0);
10     vect(0) =  94.50;
11     vect(1) = 112.01;
12     vect(2) = 121.86;

13     Matrix<double> mat(3, 2, 0.0);

14     for( int row = 0; row < 3; ++row  )
15        for( int col = 0; col < 2; ++col  )
16           mat(row, col) = 1.5 + row + 0.9*row*col;

17     SolverLMS solver;

18     solver.Compute( vect, mat );

19     cout << solver.solution(0) << " : "
20          << solver.solution(1) << endl;

21     exit(0);

22  }
```

Next, lines #9 to #16 simply declare and initialize a `Vector` (for independent terms) and a `Matrix` (for equation coefficients).

Then, line #17 declares a `SolverLMS` object to solve the equation system defined by the former `Vector` and `Matrix` objects using a Least Mean Squares (LMS) method. The solution of the equation system is achieved in line #18, where the `Compute()` method is invoked.

In order to print the solution to the equation system the lines #19 and #20 query the `solution()` field of the `solver` object.

C.6.4 Time management

```
 1  #include <iostream>
 2  #include <cmath>
 3  #include "DayTime.hpp"

 4  using namespace std;
 5  using namespace gpstk;

 6  int main()
 7  {

 8     cout << fixed << setprecision(3);   // Set output format

 9     DayTime epoch1(2009,10,28,10,30,0.0);

10     DayTime epoch2 = epoch1 + 3600.0;

11     cout << epoch1 << endl
12          << epoch2 << endl;

13     cout << epoch1.GPSfullweek() << " "
14          << epoch1.DOY()         << " "
15          << epoch1.GPSsecond()   << endl;

16     exit(0);

17  }
```

Time management is a very important aspect of every GNSS data processing software. The GPSTk has several classes to manage time, and one of the most important one is the `DayTime` class.

In the source code above, at line #9 a `DayTime` object is declared (`epoch1`), initialized to October 28th. 2009 at 10:30:00. Afterwards, line #10 declares another `DayTime` object (`epoch2`), initializing it as `epoch1` plus 3600 seconds (one hour). Note again the overloading of "+".

Lines #11 to #15 print the results, starting with both epochs and then printing the GPS week number, day of year and GPS seconds of week for `epoch1`. The output of this program follows:

```
10/28/2009 10:30:00
10/28/2009 11:30:00
1555 301 297000.000
```

C.6.5 Position

Proper handling of the position of points on or near Earth surface is another important part of GNSS data processing software. The `Position` class is one of the facilities provided by the GPSTk to ease such handling.

```
 1  #include <iostream>
 2  #include "Position.hpp"

 3  using namespace std;
 4  using namespace gpstk;

 5  int main()
 6  {

 7     cout << fixed << setprecision(3);   // Set output format

 8     Position posBCN( 4789031.0, 176583.0, 4195015.0 );

 9     Position posHANOI;
10     posHANOI.setGeodetic( 21.033, 105.85, 308.0 );

11     Position posSAT( -22300542.564, 9466839.117, 10798193.375 );

12     cout << posBCN    << endl
13          << posHANOI  << endl
14          << posHANOI.elevation( posSAT ) << " : "
15          << posHANOI.azimuth( posSAT )   << endl;

16     exit(0);

17  }
```

In the former piece of code, line #8 defines a `Position` class object called `posBCN`, initializing it with a set of Earth-Centered, Earth-Fixed (ECEF) coordinates (in meters) corresponding to a place in Barcelona city, Spain. Line #11 does a similar job for a fictitious satellite.

On the other hand, lines #9 and #10 define another `Position` object, but it is initialized using a *geodetic* set of coordinates, i.e., latitude, longitude and ellipsoidal height (in this case, for Hanoi city, Vietnam).

The most interesting part happens in lines #12 to #15, where printing takes place. The output follows:

```
4789031.0000 m 176583.0000 m 4195015.0000 m
21.03300000 degN 105.85000000 degE 308.0000 m
30.993 : 76.113
```

In the first two lines the positions are printed, but note that the `Position` objects remember the reference frame used (ECEF for `posBCN` and *geodetic* for `posHanoi`).

On the other hand, internal conversions are carried out automatically when needed: Lines #14 and #15 compute and print the elevation and azimuth of `posSat` satellite with respect to `posHanoi`, although both objects were initialized using different reference frames.

C.6.6 RINEX observation files

Parsing of Receiver INdependent EXchange format (RINEX) files is a fundamental task, and the GPSTk provides several classes for this. In the following code a RINEX observations file is processed in a very simple way.

```
#include <iostream>
#include <iomanip>

#include "RinexObsBase.hpp"
#include "RinexObsHeader.hpp"
#include "RinexObsData.hpp"
#include "RinexObsStream.hpp"

using namespace std;
using namespace gpstk;

int main()
{

   RinexObsStream rinexFile("bahr1620.04o");

   RinexObsHeader rinexHeader;
   rinexFile >> rinexHeader;

   cout << rinexHeader.markerName << " : "
        << rinexHeader.antType    << " : "
        << rinexHeader.firstObs   << endl;

   RinexObsData obsData;

   while( rinexFile >> obsData )
   {
      obsData.dump( cout );
   }

   exit(0);

}
```

It can be seen from the `include` lines that RINEX handling requires using several different classes. One of those classes is shown in line #11, where the "`bahr1620.04o`" observations file is "opened" by a `RinexObsStream` object.

Then, the RINEX "header" is handled in lines #12 and #13. Note how the overloading of >> operator allows to take RINEX header data out of `rinexFile` and into `rinexHeader`. After that, several fields from the header (marker name, antenna type and epoch of first observation) are printed out in lines #14 and #16.

The observations are dealed with using a `RinexObsData` object. That object (called `obsData` in this example) is able to encapsulate a single-epoch worth of data from the observations file, and therefore, a mechanism is needed to query the `rinexFile` object in an epoch-by-epoch basis.

Such mechanism is provided by the use of the overloaded >> operator inside a `while` loop, as lines #18 to #21 shown. The `rinexFile` object will "*pour*" one epoch of data into `obsData` object *while* there is data left in `bahr1620.04o` file. When the end of file is reached, a FALSE condition is returned, leaving the "`while`" loop and ending the program (line #22).

For each epoch the `dump()` method of `obsData` is called (line #20), printing the data to the screen. In a full-fledged GNSS application this line would be replaced by the data processing we are interested in.

C.6.7 Ephemeris files

Ephemeris files are other important part of GNSS data processing. This example will show a basic way to manage SP3 files for precise ephemeris.

The SP3 file is "loaded" into a `SP3EphemerisStore` object in line #10, and given that this example will query about the orbit of an specific GPS satellite (PRN 10), a `SatID` object is created to represent it (line #11). Afterwards, two `DayTime` objects are created to store the beginning and end epochs of SP3 file (lines #12 and #13).

Lines #14 to #30 comprise a "`while`" loop that will take care of querying for `sat` position every minute (line #16) *while* the end epoch of SP3 file data is not exceeded (line #14).

The position and velocity of the satellite will be stored in a `Xvt` class object called `svPos`. Line #20 uses `getXvt()` method to achieve this for a specific satellite-epoch combination, and lines #21 to #24 print the epoch (in seconds of day) and the three ECEF coordinates (in meters).

```
 1  #include <iostream>
 2  #include <iomanip>

 3  #include "SP3EphemerisStore.hpp"

 4  using namespace std;
 5  using namespace gpstk;

 6  int main()
 7  {

 8     cout << fixed << setprecision(3);   // Set output format

 9     SP3EphemerisStore SP3EphList;
10     SP3EphList.loadFile( "igs13354.sp3" );

11     SatID sat( 10, SatID::systemGPS );

12     DayTime epoch( SP3EphList.getInitialTime() );
13     DayTime epochEnd( SP3EphList.getFinalTime() );

14     while( epoch < epochEnd )
15     {

16        epoch += 60.0;

17        Xvt svPos;

18        try
19        {
20           svPos = SP3EphList.getXvt( sat, epoch );
21           cout << epoch.DOYsecond() << "  "
22                << svPos.x[0]        << "  "
23                << svPos.x[1]        << "  "
24                << svPos.x[2]        << endl;
25        }
26        catch (InvalidRequest& e)
27        {
28           continue;
29        }

30     }

31     exit(0);

32  }
```

Please note the use of a "`try-catch`" combination to protect against the posibility that no orbit information is available for the desired satellite at a given epoch. Such situation would make the program to abort, but this is avoided when line #26 "*catches*" the issued `InvalidRequest` exception, and properly handles the situation in line #28 telling the program to just continue with the next iteration.

C.6.8 Solid tides

Precise GNSS data processing requires modelling of several parameters. In this example, the GPSTk-provided tools to model solid tides are presented.

```
 1  #include <iostream>

 2  #include "SolidTides.hpp"

 3  using namespace std;
 4  using namespace gpstk;

 5  int main()
 6  {

 7     cout << fixed << setprecision(3);   // Set output format

 8     Position posHANOI;
 9     posHANOI.setGeodetic( 21.033, 105.85, 308.0 );

10     SolidTides solid;

11     DayTime epoch(2009,10,28,0,0,0.0);
12     DayTime epochEnd(2009,10,29,0,0,0.0);

13     while( epoch < epochEnd )
14     {

15        Triple hanoiTide = solid.getSolidTide( epoch, posHANOI );

16        cout << epoch.DOYsecond()  << " "
17             << hanoiTide[0]       << " "
18             << hanoiTide[1]       << " "
19             << hanoiTide[2]       << endl;

20        epoch += 300.0;

21     }

22     exit(0);

23  }
```

The solid tide effect will be computed for Hanoi city, Vietnam. Therefore, lines #8 and #9 declare and initialize a `Position` object (`posHANOI`) with the proper coordinates. After that, a `SolidTides` object is declared: it encapsulates all the modeling.

The computations will be done for October 28th, 2009, and lines #11 and #12 define `DayTime` objects encompassing such 24 h period.

The "`while`" loop in lines #13 to #21 takes care of computing and printing the tidal values. Line #15 is the main statement, calling the `solid` object's `getSolidTide()` method taking as parameters the epoch and position of interest. The result is stored in a `Triple` object: A handy encapsulation of a 3D vector.

The printing is simple: First the seconds of day for the current epoch and then the tidal effects (in meters) given in a topocentric Up-East-North (UEN) reference frame.

Finally, line #20 increments the epoch of interest in 5 minutes (300 seconds) intervals, and the loop will repeat itself until the first second of the next day is reached.

Appendix D

GPSTk documentation

The GPSTk has an excellent documentation thanks to its use of the `Doxygen` documentation system. This is a very important feature of the project because in this way it provides a very complete API to the programmer.

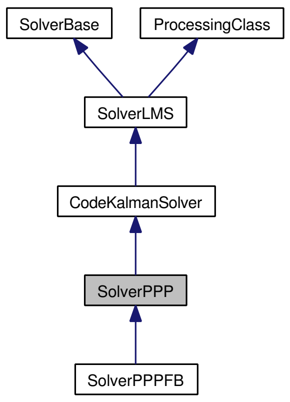
`Doxygen` allows to create a very complete set of documentation right from the GPSTk source code. Using special comment tags, the GPSTk developers write the documentation *at the same time* they write the code.

As an example, in the following section there is an abridged version of the documentation generated for the `SolverPPP` class.

D.1 SolverPPP Class Reference

`#include <SolverPPP.hpp>`

Inheritance diagram for SolverPPP:



Detailed Description

This class computes the Precise Point Positioning (PPP) solution using a Kalman solver that combines ionosphere-free code and phase measurements.

This class may be used either in a Vector- and Matrix-oriented way, or with GNSS data structure objects from "DataStructures" class (much more simple to use it this way).

A typical way to use this class with GNSS data structures follows:

```
      // INITIALIZATION PART

      // EBRE station nominal position
   Position nominalPos(4833520.192, 41537.1043, 4147461.560);
   RinexObsStream rin("ebre0300.02o");  // Data stream

      // Load all the SP3 ephemerides files
   SP3EphemerisStore SP3EphList;
   SP3EphList.loadFile("igs11512.sp3");
   SP3EphList.loadFile("igs11513.sp3");
   SP3EphList.loadFile("igs11514.sp3");

   NeillTropModel neillTM( nominalPos.getAltitude(),
                           nominalPos.getGeodeticLatitude(),
                           30 );

      // Objects to compute the tropospheric data
   BasicModel basicM(nominalPos, SP3EphList);
   ComputeTropModel computeTropo(neillTM);

      // More declarations here: ComputeMOPSWeights, SimpleFilter,
      // LICSDetector, MWCSDetector, SolidTides, OceanLoading,
      // PoleTides, CorrectObservables, ComputeWindUp, ComputeLinear,
      // LinearCombinations, etc.

      // Declare a SolverPPP object
   SolverPPP pppSolver;

     // PROCESSING PART

   gnssRinex gRin;

   while(rin >> gRin)
   {
      try
      {
         gRin  >> basicM
               >> correctObs
               >> compWindup
               >> computeTropo
               >> linear1      // Compute combinations
               >> pcFilter
               >> markCSLI2
               >> markCSMW
               >> markArc
               >> linear2      // Compute prefit residuals
               >> phaseAlign
               >> pppSolver;
      }
      catch(...)
      {
         cerr << "Unknown exception at epoch: " << time << endl;
         continue;
      }

         // Print results
      cout << time.DOYsecond()      << "  "; // Output field #1
      cout << pppSolver.solution[1] << "  "; // dx: Output field #2
      cout << pppSolver.solution[2] << "  "; // dy: Output field #3
      cout << pppSolver.solution[3] << "  "; // dz: Output field #4
      cout << pppSolver.solution[0] << "  "; // wetTropo: Out field #5
      cout << endl;
   }
```

The "SolverPPP" object will extract all the data it needs from the GNSS data structure that is "gRin" and will try to solve the PPP system of equations using a Kalman filter. It will also insert back postfit residual data (both code and phase) into "gRin" if it successfully solves the equation system.

By default, it will build the geometry matrix from the values of coefficients wetMap, dx, dy, dz and cdt, IN THAT ORDER. Please note that the first field of the solution will be the estimation of the zenital wet tropospheric component (or at least, the part that wasn't modeled by the tropospheric model used).

You may configure the solver to work with a NEU system in the class constructor or using the "setNEU()" method.

In any case, the "SolverPPP" object will also automatically add and estimate the ionosphere-free phase ambiguities. The independent vector is composed of the code and phase prefit residuals.

This class expects some weights assigned to each satellite. That can be achieved with objects from classes such as "ComputeIURAWeights", "ComputeMOP-SWeights", etc.

If these weights are not assigned, then the "SolverPPP" object will set a value of "1.0" to code measurements, and "weightFactor" to phase measurements. The default value of "weightFactor" is "10000.0". This implies that code sigma is 1 m, and phase sigma is 1 cm.

By default, the stochastic models used for each type of variable are:

- Coordinates are modeled as constants (**StochasticModel**).
- Zenital wet tropospheric component is modeled as a random walk (see class **RandomWalkModel**), with a qPrime value of 3e-8 m∗m/s.
- Receiver clock is modeled as white noise (**WhiteNoiseModel**).
- Phase biases are modeled as white noise when cycle slips happen, and as constants between cycle slips (**PhaseAmbiguityModel**).

You may change this assignment with methods "setCoordinatesModel()", "setX-CoordinatesModel()", "setYCoordinatesModel()", "setZCoordinatesModel()", "setTroposphereModel()" and "setReceiverClockModel()". However, you are not allowed to change the phase biases stochastic model.

For instance, in orden to use a 'full kinematic' mode we assign a white noise model to all the coordinates:

```
      // Define a white noise model with 100 m of sigma
   WhiteNoiseModel wnM(100.0);

      // Configure the solver to use this model for all coordinates
   pppSolver.setCoordinatesModel(&wnM);
```

Be aware, however, that you MUST NOT use this method to set a state-aware stochastic model (like **RandomWalkModel**, for instance) to ALL coordinates, because the results will certainly be erroneous. Use this method ONLY with non-state-aware stochastic models like 'StochasticModel' (constant coordinates) or 'WhiteNoiseModel'.

In order to overcome the former limitation, this class provides methods to set different, specific stochastic models for each coordinate, like:

```
      // Define a white noise model with 2 m of sigma for horizontal
      // coordinates (in this case, the solver is previously set to use
      // dLat, dLon and dH).
   WhiteNoiseModel wnHorizontalModel(2.0);

      // Define a random walk model with 0.04 m*m/s of process spectral
      // density for vertical coordinates.
   RandomWalkModel rwVerticalModel(0.04);

      // Configure the solver to use these models
   pppSolver.setXCoordinatesModel(&wnHorizontalModel);
   pppSolver.setYCoordinatesModel(&wnHorizontalModel);
   pppSolver.setZCoordinatesModel(&rwVerticalModel);
```

Warning:

"SolverPPP" is based on a Kalman filter, and Kalman filters are objets that store their internal state, so you MUST NOT use the SAME object to process DIFFERENT data streams.

See also:

SolverBase.hpp, **SolverLMS.hpp** and **CodeKalmanSolver.hpp** for base classes.

Examples:

example10.cpp, **example8.cpp**, and **example9.cpp**.

Definition at line 210 of file SolverPPP.hpp.

Public Member Functions

- **SolverPPP** (bool useNEU=false)

  Common constructor.

- virtual int **Compute** (const **Vector**< double > &prefitResiduals, const **Matrix**< double > &designMatrix, const **Matrix**< double > &weightMatrix) throw (InvalidSolver)

  Compute the PPP Solution of the given equations set.

- virtual int **Compute** (const **Vector**< double > &prefitResiduals, const **Matrix**< double > &designMatrix, const **Vector**< double > &weightVector) throw (InvalidSolver)

  Compute the PPP Solution of the given equations set.

- virtual **gnssSatTypeValue** & **Process** (**gnssSatTypeValue** &gData) throw (ProcessingException)

  Returns a reference to a gnnsSatTypeValue object after solving the previously defined equation system.

- virtual **gnssRinex** & **Process** (**gnssRinex** &gData) throw (ProcessingException)

  Returns a reference to a gnnsRinex object after solving the previously defined equation system.

- virtual **SolverPPP** & **Reset** (const **Vector**$<$ double $>$ &newState, const **Matrix**$<$ double $>$ &newErrorCov)

  Resets the PPP internal Kalman filter.

- virtual **SolverPPP** & **setNEU** (bool useNEU)

  Sets if a NEU system will be used.

- virtual double **getWeightFactor** (void) const

  Get the weight factor multiplying the phase measurements sigmas.

- virtual **SolverPPP** & **setWeightFactor** (double factor)

  Set the weight factor multiplying the phase measurement sigma.

- **StochasticModel** $*$ **getXCoordinatesModel** () const

  Get stochastic model pointer for dx (or dLat) coordinate.

- **SolverPPP** & **setXCoordinatesModel** (**StochasticModel** $*$pModel)

  Set coordinates stochastic model for dx (or dLat) coordinate.

- **StochasticModel** $*$ **getYCoordinatesModel** () const

  Get stochastic model pointer for dy (or dLon) coordinate.

- **SolverPPP** & **setYCoordinatesModel** (**StochasticModel** $*$pModel)

  Set coordinates stochastic model for dy (or dLon) coordinate.

- **StochasticModel** $*$ **getZCoordinatesModel** () const

  Get stochastic model pointer for dz (or dH) coordinate.

- **SolverPPP** & **setZCoordinatesModel** (**StochasticModel** $*$pModel)

  Set coordinates stochastic model for dz (or dH) coordinate.

- virtual **SolverPPP** & **setCoordinatesModel** (**StochasticModel** $*$pModel)

  Set a single coordinates stochastic model to ALL coordinates.

- virtual **StochasticModel** $*$ **getTroposphereModel** (void) const

  Get wet troposphere stochastic model pointer.

- virtual **SolverPPP** & **setTroposphereModel** (**StochasticModel** $*$pModel)

  Set zenital wet troposphere stochastic model.

- virtual **StochasticModel** ∗ **getReceiverClockModel** (void) const

  Get receiver clock stochastic model pointer.

- virtual **SolverPPP** & **setReceiverClockModel** (**StochasticModel** ∗pModel)

  Set receiver clock stochastic model.

- virtual **StochasticModel** ∗ **getPhaseBiasesModel** (void) const

  Get phase biases stochastic model pointer.

- virtual **SolverPPP** & **setPhaseBiasesModel** (**StochasticModel** ∗pModel)

  Set phase biases stochastic model.

- virtual **Matrix**< double > **getPhiMatrix** (void) const

  Get the State Transition **Matrix** *(phiMatrix).*

- virtual **SolverPPP** & **setPhiMatrix** (const **Matrix**< double > &pMatrix)

  Set the State Transition **Matrix** *(phiMatrix).*

- virtual **Matrix**< double > **getQMatrix** (void) const

  Get the Noise covariance matrix (QMatrix).

- virtual **SolverPPP** & **setQMatrix** (const **Matrix**< double > &pMatrix)

  Set the Noise Covariance **Matrix** *(QMatrix).*

- virtual int **getIndex** (void) const

  Returns an index identifying this object.

- virtual std::string **getClassName** (void) const

  Returns a string identifying this object.

- virtual ∼**SolverPPP** ()

  Destructor.

Classes

- struct **filterData**

  A structure used to store Kalman filter data.

Constructor & Destructor Documentation

SolverPPP (bool useNEU = `false`)

Common constructor.

Parameters:

useNEU If true, will compute dLat, dLon, dH coordinates; if false (the default), will compute dx, dy, dz.

Definition at line 57 of file SolverPPP.cpp.

References SolverPPP::setNEU().

virtual ∼SolverPPP () `[inline, virtual]`

Destructor.

Definition at line 484 of file SolverPPP.hpp.

Member Function Documentation

int Compute (const Vector< double > & prefitResiduals, const Matrix< double > & designMatrix, const Matrix< double > & weightMatrix) throw (InvalidSolver) `[virtual]`

Compute the PPP Solution of the given equations set.

Parameters:

prefitResiduals **Vector** of prefit residuals

designMatrix Design matrix for the equation system

weightMatrix **Matrix** of weights

Warning:

A typical Kalman filter works with the measurements noise covariance matrix, instead of the matrix of weights. Beware of this detail, because this method uses the later.

Returns:

0 if OK -1 if problems arose

Reimplemented from **CodeKalmanSolver**.

Definition at line 168 of file SolverPPP.cpp.

References SimpleKalmanFilter::Compute(), SolverBase::covMatrix, GPSTK_-RETHROW, GPSTK_THROW, gpstk::inverseChol(), ConstMatrixBase::isSquare(), SimpleKalmanFilter::P, SolverBase::postfitResiduals, Matrix::rows(), SolverBase::solution, SolverBase::valid, and SimpleKalmanFilter::xhat.

Referenced by SolverPPP::Compute(), and SolverPPP::Process().

int Compute (const Vector< double > & prefitResiduals, const Matrix< double > & designMatrix, const Vector< double > & weightVector) throw (InvalidSolver) `[virtual]`

Compute the PPP Solution of the given equations set.

Parameters:

prefitResiduals	**Vector** of prefit residuals
designMatrix	Design matrix for the equation system
weightVector	**Vector** of weights assigned to each satellite.

Warning:

A typical Kalman filter works with the measurements noise covariance matrix, instead of the vector of weights. Beware of this detail, because this method uses the later.

Returns:

0 if OK -1 if problems arose

Reimplemented from **CodeKalmanSolver**.

Definition at line 117 of file SolverPPP.cpp.

References SolverPPP::Compute(), GPSTK_THROW, and SolverBase::valid.

gnssSatTypeValue & Process (gnssSatTypeValue & gData) throw (ProcessingException) `[virtual]`

Returns a reference to a gnnsSatTypeValue object after solving the previously defined equation system.

Parameters:

gData Data object holding the data.

Reimplemented from **CodeKalmanSolver**.

Reimplemented in **SolverPPPFB**.

Definition at line 284 of file SolverPPP.cpp.

References gpstk::StringUtils::asString(), gnssData::body, SolverPPP::getClassName(), SolverPPP::getIndex(), GPSTK_THROW, gnssRinex::header, and Exception::what().

Referenced by SolverPPPFB::LastProcess(), SolverPPPFB::Process(), and method SolverPPPFB::ReProcess().

gnssRinex & Process (gnssRinex & gData) throw (ProcessingException) `[virtual]`

Returns a reference to a gnnsRinex object after solving the previously defined equation system.

Parameters:

gData Data object holding the data.

Reimplemented from **CodeKalmanSolver**.

Reimplemented in **SolverPPPFB**.

Definition at line 325 of file SolverPPP.cpp.

References gpstk::StringUtils::asString(), gnssData::body, SolverPPP::Compute(), SolverBase::covMatrix, TypeID::CSL1, field SolverLMS::defaultEqDef, method

SolverPPP::getClassName(), SolverPPP::getIndex(), StochasticModel::getPhi(), StochasticModel::getQ(), GPSTK_THROW, gnssData::header, TypeID::postfitC, TypeID::postfitL, SolverBase::postfitResiduals, TypeID::prefitL, method StochasticModel::Prepare(), SimpleKalmanFilter::Reset(), Vector::resize(), Matrix::resize(), SolverBase::solution, TypeID::weight, and Exception::what().

virtual SolverPPP& Reset (const Vector< double > & newState, const Matrix< double > & newErrorCov) `[inline, virtual]`

Resets the PPP internal Kalman filter.

Parameters:

newState System state vector
newErrorCov Error covariance matrix

Warning:

Take care of dimensions: In this case newState must be 6x1 and newErrorCov must be 6x6.

Definition at line 290 of file SolverPPP.hpp.

References SimpleKalmanFilter::Reset().

SolverPPP & setNEU (bool useNEU) `[virtual]`

Sets if a NEU system will be used.

Parameters:

useNEU Boolean value indicating if a NEU system will be used

Reimplemented in **SolverPPPFB**.

Definition at line 730 of file SolverPPP.cpp.

References gnssData::body, TypeID::cdt, SolverLMS::defaultEqDef, TypeID::dH, TypeID::dLat, TypeID::dLon, TypeID::dx, TypeID::dy, TypeID::dz, gnssData::header, TypeID::prefitC, and TypeID::wetMap.

Referenced by SolverPPPFB::setNEU(), SolverPPP::SolverPPP(), and method SolverPPPFB::SolverPPPFB().

virtual double getWeightFactor (void) const `[inline, virtual]`

Get the weight factor multiplying the phase measurements sigmas.

This factor is the code_sigma/phase_sigma ratio.

Definition at line 307 of file SolverPPP.hpp.

References gpstk::sqrt().

virtual SolverPPP& setWeightFactor (double factor) `[inline, virtual]`

Set the weight factor multiplying the phase measurement sigma.

Parameters:

factor Factor multiplying the phase measurement sigma

Warning:

This factor should be the code_sigma/phase_sigma ratio. For instance, if we assign a code sigma of 1 m and a phase sigma of 10 cm, the ratio is 100, and so should be "factor".

Definition at line 319 of file SolverPPP.hpp.

StochasticModel∗ getXCoordinatesModel () const `[inline]`

Get stochastic model pointer for dx (or dLat) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 324 of file SolverPPP.hpp.

SolverPPP& setXCoordinatesModel (StochasticModel ∗ pModel) `[inline]`

Set coordinates stochastic model for dx (or dLat) coordinate.

Parameters:

pModel Pointer to **StochasticModel** associated with dx (or dLat) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 333 of file SolverPPP.hpp.

StochasticModel∗ getYCoordinatesModel () const `[inline]`

Get stochastic model pointer for dy (or dLon) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 338 of file SolverPPP.hpp.

SolverPPP& setYCoordinatesModel (StochasticModel ∗ pModel) `[inline]`

Set coordinates stochastic model for dy (or dLon) coordinate.

Parameters:

pModel Pointer to **StochasticModel** associated with dy (or dLon) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 347 of file SolverPPP.hpp.

StochasticModel∗ getZCoordinatesModel () const `[inline]`

Get stochastic model pointer for dz (or dH) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 352 of file SolverPPP.hpp.

SolverPPP& setZCoordinatesModel (StochasticModel ∗ pModel) `[inline]`

Set coordinates stochastic model for dz (or dH) coordinate.

Parameters:

pModel Pointer to **StochasticModel** associated with dz (or dH) coordinate.

Reimplemented from **CodeKalmanSolver**.

Definition at line 361 of file SolverPPP.hpp.

SolverPPP & setCoordinatesModel (StochasticModel ∗ pModel) `[virtual]`

Set a single coordinates stochastic model to ALL coordinates.

Parameters:

pModel Pointer to **StochasticModel** associated with coordinates.

Warning:

Do NOT use this method to set the SAME state-aware stochastic model (like **RandomWalkModel**, for instance) to ALL coordinates, because the results will certainly be erroneous. Use this method only with non-state-aware stochastic models like 'StochasticModel' (constant coordinates) or 'WhiteNoiseModel'.

Reimplemented from **CodeKalmanSolver**.

Definition at line 774 of file SolverPPP.cpp.

virtual StochasticModel∗ getTroposphereModel (void) const `[inline, virtual]`

Get wet troposphere stochastic model pointer.

Definition at line 380 of file SolverPPP.hpp.

virtual SolverPPP& setTroposphereModel (StochasticModel ∗ pModel) `[inline, virtual]`

Set zenital wet troposphere stochastic model.

Parameters:

pModel Pointer to **StochasticModel** associated with zenital wet troposphere.

Warning:

> Be aware that some stochastic models store their internal state (for instance, 'RandomWalkModel' and 'PhaseAmbiguityModel'). If that is your case, you MUST NOT use the SAME model in DIFFERENT solver objects.

Definition at line 394 of file SolverPPP.hpp.

virtual StochasticModel∗ getReceiverClockModel (void) const `[inline, virtual]`

Get receiver clock stochastic model pointer.

Reimplemented from **CodeKalmanSolver**.

Definition at line 399 of file SolverPPP.hpp.

virtual SolverPPP& setReceiverClockModel (StochasticModel ∗ pModel) `[inline, virtual]`

Set receiver clock stochastic model.

Parameters:

> **pModel** Pointer to **StochasticModel** associated with receiver clock.

Warning:

> Be aware that some stochastic models store their internal state (for instance, 'RandomWalkModel' and 'PhaseAmbiguityModel'). If that is your case, you MUST NOT use the SAME model in DIFFERENT solver objects.

Reimplemented from **CodeKalmanSolver**.

Definition at line 413 of file SolverPPP.hpp.

virtual StochasticModel∗ getPhaseBiasesModel (void) const `[inline, virtual]`

Get phase biases stochastic model pointer.

Definition at line 418 of file SolverPPP.hpp.

virtual SolverPPP& setPhaseBiasesModel (StochasticModel ∗ pModel) `[inline, virtual]`

Set phase biases stochastic model.

Parameters:

pModel Pointer to **StochasticModel** associated with phase biases.

Warning:

Be aware that some stochastic models store their internal state (for instance, 'RandomWalkModel' and 'PhaseAmbiguityModel'). If that is your case, you MUST NOT use the SAME model in DIFFERENT solver objects. This method should be used with caution, because model must be of **PhaseAmbiguityModel** class in order to make sense.

Definition at line 435 of file SolverPPP.hpp.

virtual Matrix<double> getPhiMatrix (void) const `[inline, virtual]`

Get the State Transition **Matrix** (phiMatrix).

Reimplemented from **CodeKalmanSolver**.

Definition at line 440 of file SolverPPP.hpp.

virtual SolverPPP& setPhiMatrix (const Matrix< double > & pMatrix) `[inline, virtual]`

Set the State Transition **Matrix** (phiMatrix).

Parameters:

pMatrix State Transition matrix.

Warning:

Process() methods set phiMatrix and qMatrix according to the stochastic models already defined. Therefore, you must use the **Compute()** methods directly if you use this method.

Reimplemented from **CodeKalmanSolver**.

Definition at line 453 of file SolverPPP.hpp.

virtual Matrix<double> getQMatrix (void) const `[inline, virtual]`

Get the Noise covariance matrix (QMatrix).

Reimplemented from **CodeKalmanSolver**.

Definition at line 458 of file SolverPPP.hpp.

virtual SolverPPP& setQMatrix (const Matrix< double > & pMatrix) `[inline, virtual]`

Set the Noise Covariance **Matrix** (QMatrix).

Parameters:

pMatrix Noise Covariance matrix.

Warning:

Process() methods set phiMatrix and qMatrix according to the stochastic models already defined. Therefore, you must use the **Compute()** methods directly if you use this method.

Reimplemented from **CodeKalmanSolver**.

Definition at line 471 of file SolverPPP.hpp.

int getIndex (void) const `[virtual]`

Returns an index identifying this object.

Reimplemented from **CodeKalmanSolver**.

Reimplemented in **SolverPPPFB**.

Definition at line 43 of file SolverPPP.cpp.

Referenced by SolverPPP::Process().

std::string getClassName (void) const `[virtual]`

Returns a string identifying this object.

Reimplemented from **CodeKalmanSolver**.

Reimplemented in **SolverPPPFB**.

Definition at line 48 of file SolverPPP.cpp.

Referenced by SolverPPP::Process().

The documentation for this class was generated from the following files:

- **SolverPPP.hpp**
- **SolverPPP.cpp**

Appendix E

Porting the GPSTk to the Gumstix

The GPSTk is a highly platform-independent software code base thanks to its use of the ANSI C++ programming language.

In order to explore both the portability characteristics of the GPSTk and its capabilities to work in very small embedded system, the author carried out some research in this area during the development of this thesis, presenting the results in the proceedings of the 3rd. ESA Workshop on Satellite Navigation User Equipment Technologies (NAVITEC '2006) [Salazar et al., 2006].

The process to port part of the GPSTk to the Basix 200 Gumstix is now briefly described. For more details, please consult [Salazar et al., 2006].

E.1 Description of Gumstix boards

The Gumstix computer boards are tiny Full Function Miniature Computers (FFMC), measuring just 80 mm x 20 mm x 6.3 mm and weighting about 8 g, based on the Intel PXA-255 processor. These FFMC's are addressed for markets needing high function, low cost development and production platforms (for an example of their use in research, see [Holland et al., 2005]).

All Gumstix boards include a complete Linux kernel (version 2.6) and cross-compile tools that allow programmers to develop and test applications on a host Personal Computer (PC) before transferring them to the embedded board.

Similar hardware products have already been used in the context of European Geostationay Navigation Overlay System (EGNOS) applications, notably a SISNeT-enabled Personal Digital Assistant (PDA) running with a processor akin to the Intel PXA-255 ([Toran-Marti et al., 2002]).

In this sense, the author believe that, given the closed nature of a PDA, a more "Open Software/Open hardware" approach would benefit a wider range of potential applications, and therefore, the GPSTk could be very useful. The argument for this is that a faster, more reliable and flexible developing process may be achieved thanks to the advantage of having a widely portable, solid, tested and easily reusable open code base to build upon.

In this work the lowest-end board was used: The Basix 200, running at 200 MHz with 64MB SDRAM, 4MB Strataflash and a RS-MMC slot. This board has a power requirement of less than 250 mA at full load, and its price was about 80 Euros. Figure 1.1 shows a Basix 200 Gumstix board.

E.2 Installing the cross-compiling tools

In order to compile software for the Gumstix boards, a "buildroot" needs to be installed. A buildroot consists of several "Makefiles" and patches used to generate cross-compilation tools, as well as the "root" filesystem to be installed in the embedded system ([Petazzoni et al., 2006]). The cross-compilation tools are critical, because they allow compiling software for a different type of processor (called "target") from the processor acting as "host" system (usually, a common PC).

The buildroot used for the Gumstix is described in [Waysmall Computers, 2006], and it is designed for use in Linux. In order to download it, a "subversion" [CollabNet Inc., 2001] client is needed on the host. Please note that the Gumstix board used in [Salazar et al., 2006] was flashed with buildroot version 773.

The next step is to build the buildroot, but the C++ library and compiler must be enabled before (they are disabled by default). How to do this depends on the buildroot version. For version 773 the process is simple and implies to go into the downloaded "`gumstix-buildroot`" directory and modifying the "Makefile" file there. In that file, the variable "`INSTALL_LIBSTDCPP`" must be set to "`true`". After enabling C++, the buildroot is generated issuing the command "`make`". This is a lengthy process involving downloading several software packages from the Internet.

After this process is finished, the new C++ library is found in the directory "`build_arm_nofpu/root/lib`" and it is called "`libstdc++.so.6.0.2`". After installing the library on the Gumstix board, it is ready to run C++ software. No further repetitions of this process are needed, unless you want to change your Gumstix buildroot version.

E.3 Compiling the GPSTk for the Gumstix

In order to cross-compile the GPSTk, the first step is to download it using the same subversion client mentioned above. Once this is done, change to the "`dev`" subdirectory (where all the GPSTk software resides).

Before any further action, you must be sure that the cross-compile tools (specifically, the `arm-linux-g++` compiler) are in your "`PATH`". They may be found in the subdirectory:

```
  gumstix-buildroot/build\_arm\_nofpu/staging\_dir/bin
```

Please be sure to add the full path to this directory to your current "`PATH`" variable[1].

Please note that the GPSTk uses the "`GNU build`" system ([Taylor, I.L., 1998]) to ease the compilation process. This implies that the following tools must be installed in your host system: `aclocal`, `autoconf`, `automake` and `make`.

Then, the next step is to modify the file "`configure.ac`" in order to disable some characteristics that cause problems in the Gumstix PXA-255 processor. Please find the lines "`AC_FUNC_MALLOC`" and "`AC_FUNC_REALLOC`" and delete or disable them.

Once the former steps are complete, we may proceed to prepare the GPSTk for cross-compilation issuing the following commands in the `dev` directory:

```
   aclocal
   autoconf
   automake -a
   ./configure --host=arm-linux
```

The former process creates a complex `Makefile` with all the instructions needed to cross-compile the GPSTk library object files, applications and examples. In order to proceed with the real compilation, you must then use the command `make`.

The compilation is a long process. Please be aware that you are dealing with a development version and, from time to time, some errors may appear, in particular in the companion applications. If that happens, try to re-issue the `make`

[1]The process to achieve this varies according to the Linux shell used in your host system

command and, if it does not work, disable the problematic application in the corresponding `Makefile.am` file and rerun the process from the `automake -a` command.

Once the compilation is finished, all the library object files must be collected into a single dynamic library. Change to the `dev/src` subdirectory and issue the following command:

```
    arm-linux-ar -rs libgpstk.so *.o
```

The GPSTk library and applications are now ready to be transferred to the Gumstix board.

E.4 Compiling and running GPSTk-based applications

Once the cross-compilation tools are installed in the host system, and the C++ and GPSTk libraries have been transferred to the Gumstix board, the system is ready to start developing GPSTk-based applications.

One of the great advantages of the GPSTk is its flexibility and ease of use. In this section, such flexibility will be emphasized by taking an example application (`example5.cpp`) that comes included[2] and adapting it, with very little effort, to carry out three different kinds of GNSS pseudorange processing strategies. From there, three new examples will be developed:

- `example-a.cpp`: This program will process C1 pseudoranges applying an standard ionospheric model (Klobuchar), a simple tropospheric model (called "`GCAT`" and described in [Hernandez-Pajares et al., 2001]), and the solution will be found using a standard LMS algorithm.
- `example-b.cpp`: An improvement with respect to the former one, it will process C1 pseudoranges and apply Klobuchar ionospheric model, but the tropospheric model will be the one described in the RTCA/DO-229D document (called the Minimum Operational Performance Standards (MOPS) [RTCA/SC-159., 2006]). The solution will be found using a WMS algorithm, computing the weights according the aforementioned MOPS document. It is worth noting that the MOPS algorithms are used in Satellite-Based Augmentation System (SBAS) systems receivers such as EGNOS ones, although EGNOS-broadcasted information has not been used in this simple example.

[2]`example5.cpp` was added to the GPSTk by this author, and shows how to carry out a basic C1 pseudorange-based positioning with some GPSTk's high-level classes.

- `example-c.cpp`: Very similar to `example-b.cpp`, in this case the ionosphere-free pseudorange (PC) observables are used, and therefore the ionospheric model is dropped. Tropospheric and solver algorithms remain the same as before (MOPS).

All these programs will read the observables and ephemeris data from corresponding RINEX files. Broadcast ephemeris will be used (although simple changes allow for SP3 ephemeris), and the output will be epoch (in seconds of day) and deviations in latitude, longitude and height (in meters) from a nominal position. The full source code for these examples is not shown.

A typical use of the GPSTk follows: In the following lines you will find the code added to `example5.cpp` in order to allow `example-a.cpp`, `example-b.cpp`, and `example-c.cpp` output the data in the format explained above:

```
       // Object holding nominal position
 1  Position nominalPos(4833520.2269, 41537.00768, 4147461.489);

       // Object that will hold the difference in position
 2  Position diffPos;

       // Difference between current solution and nominal position
 3  diffPos = solPos - nominalPos;

       // Azimuth, elevation of solution with respect to nominal
 4  double azimuth = nominalPos.azimuthGeodetic(solPos);
 5  double elev = nominalPos.elevationGeodetic(solPos);

       // Magnitude of the difference between solution and nominal
 6  double magnitude = RSS( diffPos.X(), diffPos.Y(), diffPos.Z() );

       // Print results
 7  cout << rData.time.DOYsecond() << " ";            // Secs of day
 8  cout << magnitude*sin(azimuth*DEG_TO_RAD) << " "; // Lon change
 9  cout << magnitude*cos(azimuth*DEG_TO_RAD) << " "; // Lat change
10  cout << magnitude*sin(elev*DEG_TO_RAD) << " ";    // Alt change
```

As can be seen in the former lines, GPSTk objects encapsulate much of the functionality needed to do common tasks in GNSS data processing. Several details are worth noting regarding the previous sample code:

- At line #1, an object named `nominalPos` (from class `Position`) is declared in order to hold the nominal position, which in this case is also set simultaneously in ECEF coordinates.
- At line #2, another `Position` object named `diffPos` is declared to

hold the difference between the "solution position" (`solPos`) and the nominal one.

- Note that the substraction ("-") operator in line #3 is *overloaded* for `Position` objects in order to allow them to be easily subtracted.

- `Position` objects encapsulate several handy methods to operate on them. For instance, the relative azimuth and elevation between the nominal position and the solution position are found using the methods `azimuthGeodetic()` and `elevationGeodetic()` in lines #4 and #5, respectively.

- When printing the results in lines #7 to #10, please note the method used in the object `rData.time` (from class `DayTime`) in order to easily get the proper output format: `DOYsecond()`.

There are plenty of features like these in the GPSTk.

Other small modifications remain in order to allow the original `example-5.cpp` to fulfill the requisites stated above for `example-a.cpp`:

- Declaring an object `gcatTM` of class `GCATTropModel` (tropospheric modelling).

- Declare object `solver` belonging to class `SolverLMS` (standard LMS algorithm).

Please remember that the purpose of these additional examples is just to show how several different processing strategies may be carried out with minor changes. For specific details about what these classes implement please consult the GPSTk's API document.

The main modifications to convert `example-a.cpp` into `example-b.cpp` include:

- Tropospheric modelling will be carried out by object `mopsTM` of class `MOPSTropModel`.

- `solver` object now must belong to class `SolverWMS` (WMS algorithm).

- A new object, `mopsWeights`, belonging to `MOPSWeight` class, must be added to compute weights.

Some minor details remain: for instance, `solver` invocation must include a vector of weights supplied by the field `mopsWeights.weightsVector`, and `mopsTM` need to be initialized, but it is easy to change from a processing strategy to another thanks to GPSTk encapsulation of important algorithms.

Finally, the change from `example-b.cpp` to `example-c.cpp` involves mainly three simple modifications:

- Object `obsC1`, originally belonging to class `ExtractC1`, must be declared as belonging to class `ExtractPC`.
- Given that the Total Group Delay (TGD) is not applicable when using PC observables[3], the object in charge of modeling the pseudoranges (`modelPR`, of class `ModeledPR`) will be configured to ignore the TGD (`modelPR.useTGD = false`).
- All references to Klobuchar ionospheric modelling are now useless and must be deleted.

The source code files for these new programs were stored in a new subdirectory: `dev/tmp`. In order to cross-compile them for the Gumstix board, change to that directory and issue commands as the following (it is a single command line):

```
arm-linux-g++ -ansi -pedantic example-a.cpp -o example-a
-I../src/ -L../src/ -lgpstk
```

The resulting executables were transferred to the Gumstix board and used to process data for station EBRE, corresponding to January 30th, 2002. Figure E.1 plots the vertical error regarding the nominal position as a function of time for these three programs, while Figure E.2 shows the horizontal error, with the corresponding horizontal Root Mean Square (RMS) values for each strategy. The results are consistent with what it is expected from these processing strategies.

[3]When they are formed using P1 and P2 observables. However, this statement is not true when using the C1 observable to compute PC.

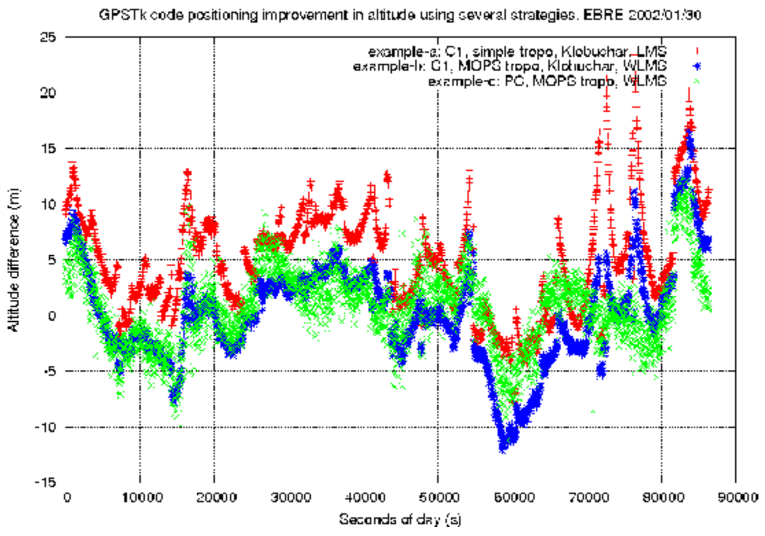


Figure E.1: *Vertical error for different processing strategies. EBRE 2002/01/30.*

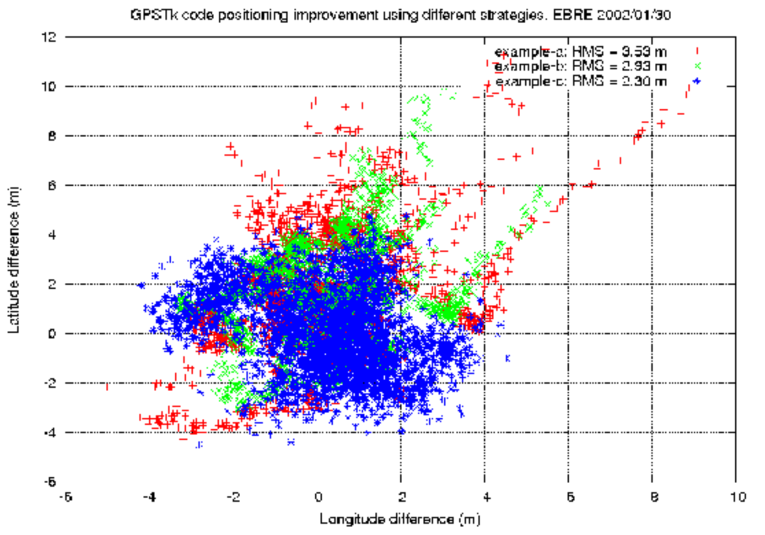


Figure E.2: *Horizontal error for different processing strategies. EBRE 2002/01/30.*

Bibliography

[ARINC Research Corp., 2000] ARINC Research Corp., 2000. Navstar GPS Space Segment / Navigation User Interfaces (ICD-GPS-200).

[Bancroft, 1985] Bancroft, S., 1985. An algebraic solution of the GPS equations. IEEE Transactions on Aerospace and Electronic Systems. 21(7), pp. 56–59.

[Beutler et al., 1999] Beutler, R., Rothacher, M., Schaer, S., Kouba, J. and Neilan, R., 1999. The International GPS Service (IGS): An interdisciplinary service in support of Earth sciences. Advances in Space Research 23(4), pp. 631–653.

[Bierman, 1977] Bierman, G. J., 1977. Factorization Methods for Discrete Sequential Estimation. Mathematics in Science and Engineering, Vol. 128, Academic Press.

[Bruton, 2000] Bruton, A. M., 2000. Improving the Accuracy and Resolution of SINS/DGPS Airborne Gravimetry. PhD thesis, Department of Geomatics Engineering, University of Clagary, Calgary, Alberta, Canada. Report No. 20145.

[Bruton et al., 1999] Bruton, A. M., Glennie, C. L. and Schwarz, K. P., 1999. Differentiation for High-Precision GPS Velocity and Acceleration Determination. GPS Solutions 2(4), pp. 7–21.

[Cannon et al., 1997] Cannon, M. E., Lachapelle, G., Szarmes, M. C., Hebert, J. M., Keith, J. and Jokerst, S., 1997. DGPS Kinematic Carrier Phase Signal Simulation Analysis for Precise Velocity and Position Determination. Journal of the Institute of Navigation 44(2), pp. 231–246.

[Castleden et al., 2004] Castleden, N., Hu, G. R., Abbey, D. A., Weihing, D., Ovstedal, O., Earls, C. J. and Featherstone, W. E., 2004. First results from Virtual Reference Station (VRS) and Precise Point Positioning (PPP) GPS research at the Western Australian Centre for Geodesy. Journal of Global Positioning Systems 3(1-2), pp. 79–84.

[CollabNet Inc., 2001] CollabNet Inc., 2001. Subversion. Website: http://subversion.tigris.org.

[Collins, 1999] Collins, J., 1999. Assessment and Development of a Tropospheric Delay Model for Aircraft users of the Global Positioning System. Technical Report 203, The University of New Brunswick, New Brunswick, Canada.

[Dahlquist and Bjork, 1974] Dahlquist, G. and Bjork, A., 1974. Numerical Methods. Prentice Hall. ISBN 0-13-627315-7.

[Daly, 1993] Daly, P., 1993. Navstar GPS and GLONASS: global satellite navigation systems. Electronics & Communication Engineering Journal 5(6), pp. 349–357.

[DoD, USA, 2008] DoD, USA, 2008. Global Positioning System Standard Positioning Service Performance Standard . http://pnt.gov/public/docs/2008/spsps2008.pdf.

[Enge and Misra, 1999] Enge, P. and Misra, P., 1999. Special Issue on Global Positioning System. In: Proceedings of the IEEE 87(1), pp. 3–15.

[Ge et al., 2008] Ge, M., Gendt, G., Rothacher, M., Shi, C. and Liu, J., 2008. Resolution of GPS carrier-phase ambiguities in Precise Point Positioning (PPP) with daily observations. Journal of Geodesy 82(7), pp. 389–399.

[Gendt, 2005] Gendt, G., 2005. Switch the absolute antanne model within the IGS. IGSMAIL-5272. Website: http://igscb.jpl.nasa.gov/mail/igsmail/2005/msg00193.html.

[Guerin, 2005] Guerin, B., 2005. Lenguaje C++. Ediciones Software SL, Po Ferrocarriles Catalanes 97-117, 2da planta, Ofic 18, Barcelona, Spain.

[Gurtner, 2001] Gurtner, W., 2001. RINEX: the Receiver Independent Exchange Format Version 2.10. Website: http://www.ngs.noaa.gov/CORS/Rinex2.html.

[Harris et al., 2007] Harris, R. B., Conn, T., Gaussiran, T., Kieschnick, C., Little, J., Mach, R., Munton, D., Renfro, B., Nelsen, S., Tolman, B., Vorce, J. and Salazar, D., 2007. The GPSTk: New Features, Applications and Changes. In: Proceedings of the 20th International Technical Meeting of the Satellite Division of the Institute of Navigation. ION GNSS 2007, Forth Worth, Texas, USA, pp. 1286–1296.

[Harris et al., 2006] Harris, R. B., Craddock, T., Conn, T., Gaussiran, T., Hagen, E., Hughes, A., Little, J., Mach, R., Nelsen, S., Renfro, B. and Tolman, B., 2006. Open Signals, Open Software: Two Years of Collaborative Analysis using the GPS Toolkit. In: Proceedings of the 19th International

Technical Meeting of the Satellite Division of the Institute of Navigation. ION GNSS 2006, Long Beach, California, USA, pp. 2865–2876.

[Hein et al., 2001] Hein, G., Godet, J., Issler, J., Martin, J., Lucas-Rodriguez, R. and Pratt, T., 2001. The GALILEO Frecuency Structure and Signal Design. In: Proceedings of ION GPS'01, Salt Lake City, Utah, USA, pp. 1273–1282.

[Hernandez-Pajares et al., 2001] Hernandez-Pajares, M., Juan-Zornoza, J. and Sanz-Subirana, J., 2001. GPS Data Processing: Code and Phase. Algorithms, Techniques and Recipes. CPET, UPC, Barcelona, Spain.

[Hernandez-Pajares et al., 2003a] Hernandez-Pajares, M., Juan-Zornoza, J., Sanz-Subirana, J. and Colombo, O., 2003a. Feasibility of Wide-Area Subdecimeter Navigation With GALILEO and Modernized GPS. IEEE Transactions on Geoscience and Remote Sensing. 41(9), pp. 2128–2131.

[Hernandez-Pajares et al., 2003b] Hernandez-Pajares, M., Juan-Zornoza, J., Sanz-Subirana, J., Prats, X. and Baeta, J., 2003b. Basic Research Utilities for SBAS (BRUS). In: 5th Geomatics Week, Barcelona, Spain.

[Hofmann-Wellenhof et al., 2008] Hofmann-Wellenhof, B., Lichtenegger, H., K. and Wasle, E., 2008. GNSS - Global Navigation Satellite Systems.. Springer-Verlag, Wien, Austria.

[Holland et al., 2005] Holland, O., Woods, J., De Nardi, R. and Clark, A., 2005. Beyond swarm intelligence: the ultraswarm. In: Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE, pp. 217–224.

[Hugentobler et al., 2006] Hugentobler, U., Meindl, M., Beutler, G., Bock, H., Dach, R., Jaggi, A., Urschl, C., Mervart, L., Rothacher, M., Schaer, S., Brockmann, E., Ineichen, D., Wiget, A., Wild, U., Weber, G., Habrich, H. and Boucher, C., 2006. CODE IGS Analysis Center. Technical Report 2003/2004, Jet Propulsion Laboratory (JPL), Pasadena, California, USA. Gowey K., Neilan R., Moore A. (eds). IGS 2004 technical reports. IGS Central Bureau.

[IERS, 2009] IERS, 2009. IERS Conventions update: Chapter 7. http://tai.bipm.org/iers/convupdt/convupdt_c7.html.

[Jekeli, 1994] Jekeli, C., 1994. On the Computation of Vehicle Accelerations Using GPS Phase Accelerations. In: Proceedings of the International Symposium on Kinematic Systems in Geodesy, Geomatics and Navigation (KIS94), Banff, Canada, pp. 473–481.

[Jekeli and Garcia, 1997] Jekeli, C. and Garcia, R., 1997. GPS Phase Accelerations for Moving-base Vector Gravimetry. Journal of Geodesy 71(10), pp. 630–639.

[Karniadakis and Kirby, 2005] Karniadakis, G. E. and Kirby, R. M., 2005. Parallel Scientific Computing in C++ and MPI. Cambridge University Press, New York, USA.

[Kennedy, S., 2002a] Kennedy, S., 2002a. Acceleration Estimation from GPS Carrier Phases for Airborne Gravimetry. PhD thesis, Department of Geomatics Engineering, University of Clagary, Calgary, Alberta, Canada. Report No. 20160.

[Kennedy, S., 2002b] Kennedy, S., 2002b. Precise Acceleration Determination from Carrier Phase Measurements. In: Proceedings of the 15th International Technical Meeting of the Satellite Division of the Institute of Navigation. ION GPS 2002, Portland, Oregon, USA, pp. 962–972.

[Kouba and Heroux, 2001] Kouba, J. and Heroux, P., 2001. Precise Point Positioning Using IGS Orbit and Clock Products. GPS Solutions 5(2), pp. 12–28.

[Kubo, 2009] Kubo, N., 2009. Advantage of velocity measurements on instantaneous RTK positioning . GPS Solutions 13(4), pp. 271–280.

[Laurichesse et al., 2009] Laurichesse, D., Mercier, F., Berthias, J., Broca, P. and Cerri, L., 2009. Integer Ambiguity Resolution on Undifferenced GPS Phase Measurements and Its Application to PPP and Satellite Precise Orbit Determination. NAVIGATION 56(2), pp. 135–149.

[Leick, 1995] Leick, A., 1995. GPS Satellite Surveying. John Wiley & Sons.

[Misra and Enge, 2006] Misra, P. and Enge, P., 2006. Global Positioning System. Signals, Measurements and Performance.. Ganga-Jamuna Press, Massachusetts, USA.

[Mostafa, 2005] Mostafa, M. M. R., 2005. Precise Airborne GPS Positioning Alternatives for the Aerial Mapping Practice. In: Proceedings of FIG Working Week 2005 and GSDI-8, Cairo, Egypt.

[Niell, 1996] Niell, A. E., 1996. Global mapping functions for the atmosphere delay at radio wavelengths. Journal of Geophysical Research (101), pp. 3227–3246.

[OSO, 2009] OSO, 2009. Ocean tide loading provider. Website: http://www.oso.chalmers.se/~loading/.

[Parkinson and Spilker Jr., 1996] Parkinson, B. W. and Spilker Jr., J. J., 1996. Global Positioning System: Theory and Applications. Volumes I & II. American Institute for Aeronautics and Astronautics, Inc.

[Petazzoni et al., 2006] Petazzoni, T., Kruse, K., Ludd, N. and Herren, M., 2006. Buildroot - Usage and documentation. Website: http://buildroot.uclibc.org/buildroot.html.

[Radovanovic et al., 2001] Radovanovic, R. S., El-Sheimy, N. and Teskey, W., 2001. Rigorous Combination of GPS Data From Multiple Base Stations for Mobile Platform Positioning. In: Proceedings of the 3rd. International Symposium on Mobile Mapping Technology, Cairo, Egypt.

[Rothacher and Schmid, 2006] Rothacher, M. and Schmid, R., 2006. ANTEX: the antenna exchange format version 1.3. Website: ftp://igscb.jpl.nasa.gov/pub/station/general/antex13.txt.

[RTCA/SC-159., 2006] RTCA/SC-159., 2006. Minimum Operational Performance Standards For Global Positioning System / Wide Area Augmentation System Airborne Equipment. Technical Report RTCA-DO229D, Radio Technical Commission for Aeronautics.

[Salazar et al., 2006] Salazar, D., Hernandez-Pajares, M., Juan, J. and Sanz, J., 2006. Rapid Open Source GPS software development for modern embedded systems: Using the GPSTk with the Gumstix. In: Proceedings of the 3rd. ESA Workshop on Satellite Navigation User Equipment Technologies. NAVITEC '2006, Noordwijk, The Netherlands.

[Salazar et al., 2007] Salazar, D., Hernandez-Pajares, M., Juan, J. and Sanz, J., 2007. The GPS Toolkit: World class open source software tools for the GNSS research community. In: Proceedings of the 7th. Geomatic Week, Barcelona. Spain.

[Salazar et al., 2008a] Salazar, D., Hernandez-Pajares, M., Juan, J. and Sanz, J., 2008a. High accuracy positioning using carrier-phases with the open source GPSTk software. In: Proceedings of the 4th. ESA Workshop on Satellite Navigation User Equipment Technologies. NAVITEC '2008, Noordwijk, The Netherlands.

[Salazar et al., 2008b] Salazar, D., Hernandez-Pajares, M., Juan, J. and Sanz, J., 2008b. Open source Precise Point Positioning with GNSS Data Structures and the GPSTk. In: Geophysical Research Abstracts. EGU2008-A-03925, Vol. 10, Vienna, Austria.

[Salazar et al., 2009a] Salazar, D., Hernandez-Pajares, M., Juan, J. and Sanz, J., 2009a. GNSS data management and processing with the GPSTk. GPS Solutions.

[Salazar et al., 2009b] Salazar, D., Sanz, J. and Hernandez-Pajares, M., 2009b. Phase-based GNSS data processing (PPP) with the GPSTk. In: Proceedings of the 8th. Geomatic Week, Barcelona. Spain.

[Scherneck, 1991] Scherneck, H., 1991. A parametrized solid Earth tide mode and ocean loading effects for global geodetic base-line measurements. Geophysical Journal International 106(3), pp. 677–694.

[Serrano et al., 2004] Serrano, L., Kim, D. and Langley, R., 2004. A Single GPS Receiver as a Real-Time, Accurate Velocity and Acceleration Sensor. In: Proceedings of the 17th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2004), Long Beach, CA, USA, pp. 2021–2034.

[Silicon Graphics, Inc., 2006] Silicon Graphics, Inc., 2006. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/index.html.

[Soulie, 2006] Soulie, J., 2006. C++ Language Tutorial. http://www.cplusplus.com/doc/tutorial/.

[Sovers and Border, 1990] Sovers, O. J. and Border, J. S., 1990. Observation Model and Parameter Partials for the JPL Geodetic GPS Modeling Software 'GPSOMC'. Technical Report 87-21, Rev. 2, Jet Propulsion Laboratory (JPL), Pasadena, California, USA.

[Stephens et al., 2006] Stephens, D. R., Diggins, C., Turkanis, J. and Cogswell, J., 2006. C++ Cookbook. O'Reilly Media Inc., Sebastopol, California, USA.

[Stroustrup, 2006a] Stroustrup, B., 2006a. Bjarne Stroustrup's FAQ. http://www.research.att.com/∼ bs/bs_faq.html.

[Stroustrup, 2006b] Stroustrup, B., 2006b. The C++ Programming Language. http://www.research.att.com/∼ bs/c++.html.

[Szarmes et al., 1997] Szarmes, M., Ryan, S., Lachapelle, G. and Fenton, P., 1997. DGPS High Accuracy Aircraft Velocity Determination Using Doppler Measurements. In: Proceedings of the International Symposium on Kinematic Systems (KIS97), Banff, Alberta, Canada.

[Takasu and Yasuda, 2009] Takasu, T. and Yasuda, A., 2009. Development of the low-cost rtk-gps receiver with an open source program package rtklib. In: Proceedings of International Symposium on GPS/GNSS, Jeju, Korea.

[Taylor, I.L., 1998] Taylor, I.L., 1998. The GNU configure and build system. Website: http://www.airs.com/ian/configure/.

[Tolman et al., 2004] Tolman, B. W., Harris, R. B., Gaussiran, T., Munton, D., Little, J., Mach, R., Nelsen, S., Renfro, B. and Schlossberg, D., 2004. The GPS Toolkit - Open Source GPS Software. In: Proceedings of ION GNSS 2004, Long Beach, California, USA.

[Toran-Marti et al., 2002] Toran-Marti, F., Ventura-Traveset, J. and Chen, R., 2002. The esa sisnet technology: Real-time access to the egnos services through wireless networks and the internet. In: Proceedings of the 15th International Technical Meeting of the Satellite Division of the Institute of Navigation ION GPS 2002.

[van Graas and Soloviev, 2004] van Graas, F. and Soloviev, A., 2004. Precise Velocity Estimation Using a Stand-Alone GPS Receiver. NAVIGATION: Journal of The Institute of Navigation 51(4), pp. 283–292.

[Wang and Gao, 2006] Wang, M. and Gao, Y., 2006. GPS Un-Differenced Ambiguity Resolution and Validation. In: Proceedings of the 19th International Technical Meeting of the Satellite Division of the Institute of Navigation (ION GNSS 2006), Fort Worth, Texas, USA., pp. 292–300.

[Waysmall Computers, 2006] Waysmall Computers, 2006. Buildroot. Website: http://docwiki.gumstix.org/Buildroot.

[Williams, 1970] Williams, J. G., 1970. Solid Earth Tides. Technical Report IOM 391-109, Jet Propulsion Laboratory (JPL), Pasadena, California, USA.

[Yang and Chen, 2001] Yang, M. and Chen, K., 2001. Performance Assessment of a Noniterative Algorithm for Global Positioning System (GPS) Absolute Positioning. In: Proceedings National Science Council ROC(A), Vol. 25number 2, pp. 102–106.

[Zhang and Forsberg, 2007] Zhang, X. and Forsberg, R., 2007. Assessment of long-range kinematic GPS positioning errors by comparison with airborne laser altimetry and satellite altimetry. Journal of Geodesy 81(3), pp. 210–211.

Index