PHASE-BASED GNSS DATA PROCESSING (PPP) WITH THE GPSTk

1 author:

Dagoberto Salazar
Trimble Navigation
17 PUBLICATIONS **203** CITATIONS

PHASE-BASED GNSS DATA PROCESSING (PPP) WITH THE GPSTk.

Salazar, D., Sanz-Subirana, J., and Hernandez-Pajares, M.
Grupo de Astronomia y Geomatica (gAGE)
Universitat Politecnica de Catalunya
08034 Barcelona
Spain
Dagoberto.Jose.Salazar@upc.edu

Keywords: PPP GNSS data processing software GPSTk.

Abstract

The objective of this work is to show some of the advances made by the 'GPS Toolkit' (GPSTk) project since the past "VII Semana Geomatica", focusing in the data management strategy called 'GNSS Data Structures' (GDS) and its associated data processing paradigm.

The GDS allow for a clean, simple-to-read, simple-to-use software code that speeds up development and minimizes errors. Therefore, the geomatic researchers have a set of powerful but flexible tools at their disposal with which they can try novel approaches without devoting much time to data-management issues.

The methodology to be used is to briefly describe the GDS paradigm, and to show how the different GNSS data 'processing objects' may be combined. The flexibility of this approach is then used to easily develop a fully working "Precise Point Positioning" (PPP) data processing chain, whose results favourably compare with other PPP-enabled software suites.

1 Introduction

The "GPS Toolkit" (GPSTk) project is an advanced GNSS Open Source Software (GOSS) suite initiated 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 its main goals is "to free researchers to focus on research, not lower level coding" [1].

Currently it is actively maintained by more than a dozen developers around the world using the Internet as communication medium. Development facilities are provided by the popular SourceForge open source application repository.

The initial code of the GPSTk was released in summer 2004 and presented at the ION-GNSS-2004 [2], 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, antenna data in Antex format, mathematical, statistical, Matrix and Vector algorithms, time handling and conversions, ionospheric and tropospheric models, cycle slip detection and correction, solid tides, ocean loading and pole tides models, etc.

As a typical Open Source project, it is open to any researcher or institution because it is released under the GNU LGPL license, which allows freedom to develop both commercial and non-commercial software.

GPSTk is highly platform-independent because it uses ANSI C++. It is reported to run on operative systems like Linux, Solaris, AIX, MS Windows and Mac OS X. It may be compiled using several free and commercial compilers, both in 32 bits and 64 bits platforms. Also, some parts of it are reported to run in platforms such as the Nokia 770 Internet Tablet and the Gumstix line of full function minicomputers [3].

The GPSTk software suite consists of a core library and extra applications. The library provides several functions that solve processing problems associated with GNSS (for instance, using RINEX files), and it is the basis for more advanced applications distributed as part of the GPSTk suite. This work focus in library provisions to write clean, simple-to-read, simple-to-use software code that speeds up development

and minimizes errors, enabling at the same time to implement "Precise Point Positioning" processing [4].

2 GNSS Data Structures approach and processing paradigm

After developing the basic code-based GNSS data processing, and starting to add phase-based capabilities to the GPSTk, several project developers came to the conclusion that some kind of hierarchy of data structures should be added in order to easily cope with frequent data management situations that were very difficult to deal with when using just vectors and matrices.

For instance, let's consider that the typical GNSS data post-processing starts reading data from Rinex files. In case this file is from observations, apart from C1, L1, etc, the programmer must take care of metadata such as epoch and cycle slip receiver flags. If the file contains ephemeris, these must be properly stored and organized.

In the following processing steps, a lot of extra information must be computed from the original data: Observable combinations, satellite's arc numbers, elevation, azimuth, relativistic, tropospheric and ionospheric corrections, wind-up effects and antenna phase center variations are just a few.

Besides, all this extra information must be properly catalogued and cross-referenced according to, for instance satellite, receiver, data type and epoch each data piece is related to.

Writing software that appropriately keeps track of all these relationships is no easy task, and the ensuing complexity makes it very difficult to debug, maintain, and enhance.

As an answer to these data management issues, a new and flexible set of data structures, called the 'GNSS Data Structures' (GDS) were added to the GPSTk.

These structures hold several kinds of GNSS-related data, properly indexed by station, epoch, satellite and type, and store them efficiently. In this way, both the data and corresponding metadata is preserved, and data management issues are properly addressed. Details about the GDS design and implementation may be found on [5], as well as in the GPSTk project website [1].

In summary, GDS take advantage of the observation that several types of GNSS-related data structures share some common characteristics, and thence they can be modelled in an unified way, creating structures that index each data value with four different indexes. Those four indexes are implemented in the GPSTk as C++ classes *SourceID*, *SatID*, *TypeID*, and *DayTime*.

Objects associated with these classes provide the GNSS programmer with a large set of methods to work with them in an easy way. Please refer to *GPSTk Application Programming Interface* (API) document for further details [6].

Apart from the GDS themselves, a "GDS processing paradigm" was also developed, where GNSS data processing becomes like an 'assembly line', and the GDS are treated like 'white boxes' that 'flow' from one 'workstation' to the next in this 'assembly line'.

It is important to note here that most of the GDS-related classes presented in this work currently are only available in the development branch of the GPSTk. Please follow the instructions in the GPSTk website to download this branch.

3 Some simple GDS examples

In order to quickly harness the power of this approach, some simple examples are presented. In the first one, we'll just extract a single epoch worth of data out of a Rinex observation file and get that data into a GDS:

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 will take care of '`ebre0300.02o`' Rinex file.

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

3  rinexFile >> gpsData;
```

On the other hand, line #2 declares an object of class `gnssRinex`, which is a very handy GNSS Data Structure. Our data will be stored in this object, which will be called `gpsData`.

Finally, line # 3 does the real work: It will take *one epoch of data* out of `rinexFile` (file '`ebre0300.02o`', indeed), and will pour 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 it around.

The statement `rinexFile >> gpsData;` has the additional property that it is evaluated as `TRUE` if operation is completed without problems, and as `FALSE` otherwise (for instance, when end of Rinex file is reached).

The aforementioned property allows us to slightly modify the former example to get a much more useful behavior:

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

3  while( rinexFile >> gpsData ) {
       ... your GNSS data processing code here ...
4  }
```

In this case the `while` loop will repeat itself until end of Rinex file is reached, and in each repetition a single epoch data set of Rinex observations is encapsulated into `gpsData`, and is fully available to the researcher to process it.

3.1 Code-processing example

Now a code-processing example will be presented. For space reasons most of the initialization phase is skipped, so you must refer to GPSTk API to get the fine details. Also, the GPSTk provides carefully explained examples: Look at "example6.cpp" and "example7.cpp" in "examples" directory for code-based processing with GDS and the GPSTk.

We will start the example with the lines handling 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 to #2 declare objects to take care of ephemeris Rinex files and associated header. Line #3 then reads the header and store it. Please remember that Klobuchar ionospheric coefficients are stores in ephemeris Rinex header, so this step is important for lines #4 to #7.

The first two of them then declare an "ionospheric model" object (`IonoModel`) and fill it with Klobuchar coefficients, and after that declare a ionospheric model store (`ionoStore`) and push in the previously defined model. Please note that this approach allows to store models for multiple days.

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

```
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, line #14 setup a tropospheric model (there are several available), and line #15 then takes care of a very interesting object: A "modeler".

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 will carry up the tasks related with GPS data modelling, computing all the delays defined by the GPS standards and using them to compute the prefit residuals and geometric coefficients that will later be used by the solver. All this wealth of information will be inserted in the GDS at due time.

Then, line #16 declares the "solver", the object in charge of building and solving the equation system. In this case it uses a simple Least Mean Squares solving algorithm (there are others available).

The function of lines #17 and #18 was already described.

Finally, we present 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;

24  }
```

The real GNSS processing is done in line #20 (the "processing line"). In that line the epoch-worth of data that was just taken out from `rinexFile` and poured into `gpsData`, is then pushed through `gpsModel` (to generate the values associated with GPS signal modelling) and `solver` (to build and solve the navigation equation system).

Some important remarks are in order: The first part of line #20 is `gpsData >> gpsModel`, which generates the model. During that phase, all new data generated by the model is input to `gpsData`, appropriately indexed.

For instance, the relativistic delay between receiver and satellite, let's say, PRN17, is computed and stored

with all metadata needed to take it apart from relativistic delays from other satellites. This is done in an automatic way and no user intervention nor coding is needed.

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 **original** `gpsData` structure **plus** 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 in `gpsData` all the information it needs to build and solve the navigation equation system.

This approach to GNSS data processing is called the "*GDS processing paradigm*", where the GDS processing becomes akin to how a car is built in a car factory. The GDS (like the car) "flows from one "workstation" to the next, where in this particular case the "workstations" are represented by objects from "processing classes" `ModelObs` and `SolverLMS`.

From the developer (and researcher) point of view, the GDS is just like a "box" (a "white box", indeed) that holds and automatically organizes all the information needed, in his only concern is to push this "box" through all the "processing classes" he needs to.

This process, which may seem computationally complex and convoluted at first sight, is however efficiently implemented using the C++ Standard Template Library (STL). A Rinex observation file (at 30 s data rate) of a full day processed in this way takes less than 0.2 seconds in an average laptop PC with Linux [1].

Coming back to our code-based processing example, the final lines #21 to #23 just take care of printing the solution to the screen, using the standard C++ `cout` printing object. Please take note of the handy 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.

4 PPP data processing accessory classes

PPP implementation is a complex task, and issues like wind-up effect, solid, oceanic and polar tides, antenna phase centers, etc. must be taken into account. Also, precise satellite orbits and clocks are used in PPP, but these products are usually provided each 900 s, while observations are usually provided each 30 s. Thence, some time management issues also arise.

We will start presenting some accessory classes that ease these complex issues. However, take into account that, again, most initialization details are skipped. For complete 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.

4.1 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:

[1] The former figure disregards output time. Redirecting output to a file in the laptop hard disk raises the overall processing time to about 0.7 seconds.

```
# Default section

   tolerance, allowed difference between time stamps = 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").

4.2 Handling 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 [7].

These values are now stored in "Antex" format files, and 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.

It follows a typical usage:

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

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

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

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

4.3 Computing tidal values

An important part of PPP modelling are 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):

In this code snippet, line #1 declares a time-handling object called `epoch` (from `DayTime` class) initialized

```
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);
```

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 [8]), and that `PoleTides` objects need x and y pole displacement parameters, in arcseconds (supplied by IGS' ERP files).

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

4.4 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 these situations. Remember 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.

Code bellow shows how this situation 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.

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

2  while( rinexFile >> gpsData ) {

3     try {

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

5     }
6     catch(DecimateEpoch& d) {
7        continue;
8     }
9     catch(Exception& e) {
10       cout <<"Exception at epoch: "<< epoch <<"; "<< e << endl;
11       continue;
12    }
13    catch(...) {
14       cout << "Unknown exception at epoch: " << epoch << endl;
15       continue;
16    }
17 }
```

5 PPP data processing code

After explaining the basic accessory classes, we are ready to present the core PPP processing code:

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

The GDS processing data chain is indeed a single C++ line, although in this case (for clarity sake) spans from line #1 to line #5. Also, all this line must be enclosed within a `while` loop to process all available epochs, and within a `try - catch` block to manage exceptions.

Several of these objects need initialization, but that part is omitted here. Again, please consult GPSTk examples and API. Table 1 summarizes object names, classes they belong to, and purpose.

Particular mention deserves object `pppSolver`, belonging to `SolverPPP` class. This object is preconfigured to solve the PPP equation system in a way consistent with [4]: Coordinates are treated as constants (static), receiver clock is considered white noise, and vertical tropospheric effect is processed as a random walk stochastic model.

Figure 1 plots the results for former PPP processing code applied to station MADR, May 27th., 2008. As mentioned, the positioning strategy is the default one for `SolverPPP` objects: PPP with static coordinates.

These results are consistent with what is expected from these processing strategies, showing a small residual bias in the "Up" coordinate of about 17 millimeters.

Those pre-assigned stochastic models may be tuned and even changed at will, given that they are objects inheriting from general class `StochasticModel`. In this regard, a way to check how good GPSTk PPP modelling is consists in treating coordinates as white noise (kinematic). The former is achieved during initialization phase in a very simple way:

```
1  WhiteNoiseModel newCoordinatesModel(100.0);
2  pppSolver.setCoordinatesModel(&newCoordinatesModel);
```

Now, `pppSolver` object will consider both coordinates and receiver clock as white noise stochastic variables, while vertical tropospheric effect is still treated as a random walk process. The coordinates' model has an

OBJECT	CLASS NAME	PURPOSE
requireObs	RequireObservables	Check if required observations are present
linear1	ComputeLinear	Compute linear combinations used to detect cycle slips
markCSLI	LICSDetector2	Detect and mark cycle slips using LI combination
markCSMW	MWCSDetector	Detect and mark cycle slips using Melbourne-Wubbena combination
markArc	SatArcMarker	Keep track of satellite arcs
decimateData	Decimate	If not a multiple of 900 s, decimate
basicModel	BasicModel	Compute the basic components of a GNSS model
eclipsedSV	EclipsedSatFilter	Remove satellites in eclipse
grDelay	GravitationalDelay	Compute gravitational delay
svPcenter	ComputeSatPCenter	Compute the effect of satellite phase center
corr	CorrectObservables	Correct observables from tides, antenna phase center, etc.
windup	ComputeWindUp	Compute wind-up effect
computeTropo	ComputeTropModel	Compute tropospheric effects
linear2	ComputeLinear	Compute ionosphere-free combinations PC and LC
pcFilter	SimpleFilter	Filter out spurious data in PC combination
phaseAlign	PhaseCodeAlignment	Align phases with codes
linear3	ComputeLinear	Compute code and phase prefit residuals
baseChange	XYZ2NEU	Prepare data to use North-East-UP reference frame
cDOP	ComputeDOP	Compute DOP figures
pppSolver	SolverPPP	Solve equations with a Kalman filter configured in PPP mode

Table 1: PPP processing objects and classes.

assigned sigma of 100.0 meters (see last part of line # 1).

Figure 2 presents the results for this type of processing, confirming the good quality of GPSTk model (results are consistently within +/- 10 cm from nominal value).

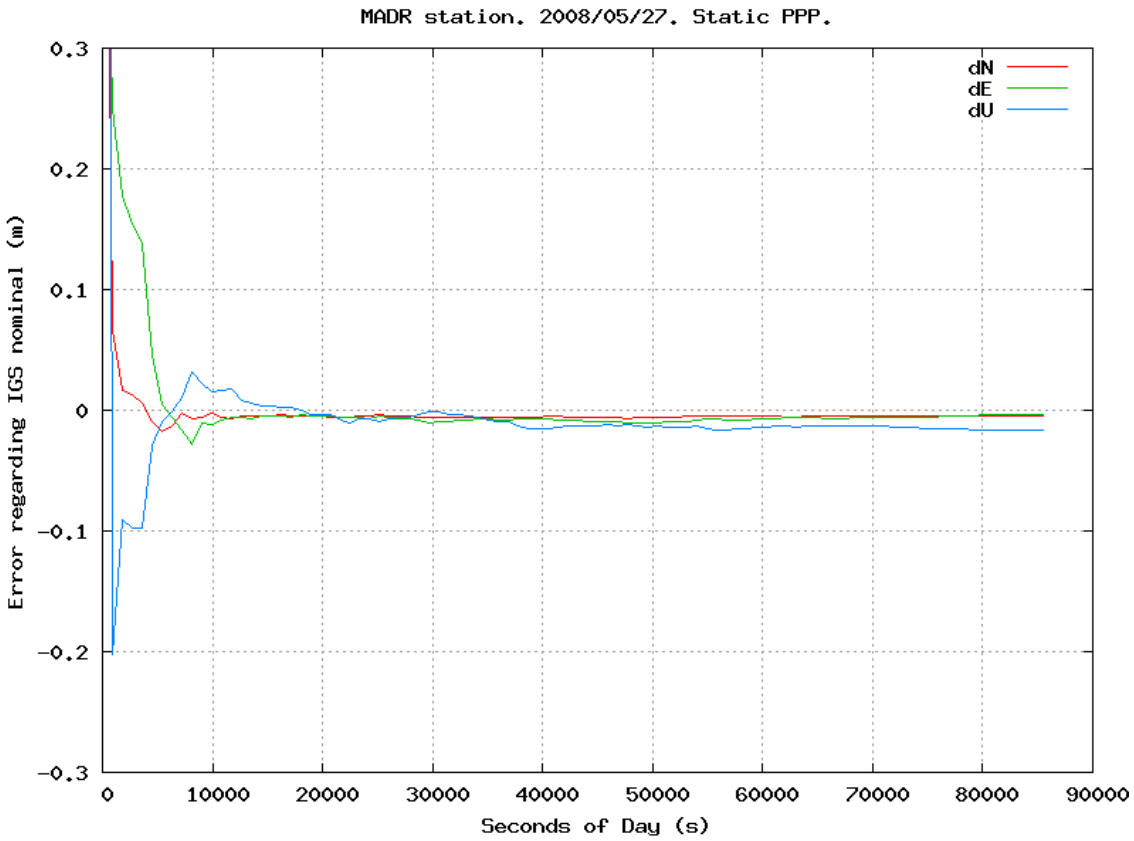

Figure 1: Static PPP positioning error regarding IGS nominal. MADR station, 2008/05/27.

5.1 Forward-Backward Kalman Filter

Previous results were obtained with a Kalman filter that only used information from the "past". However, PPP is done in post-process, and thence the filter could also use information from the "future" as well from the past. This is usually done running the filter in forward-backward mode, where the filter takes advantage of ambiguity convergence achieved in the previous forwards run, and uses that information for the next backwards run. This process may be iterated at will.

An object of class `SolverPPPFB` is used to achieve this. This class 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 `SolverPPP` object (`pppSolver`) with a new `SolverPPPFB` object (`fbpppSolver`, for instance) inside the `while` loop reading and processing RINEX data file.

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

For instance, to carry out 4 forward-backward additional cycles:

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

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

Please note that in this case the solution is given in a NEU reference frame instead of ECEF. To achieve that we must tell so to the solver object when declaring it, and insert a `XYZ2NEU` object in the processing

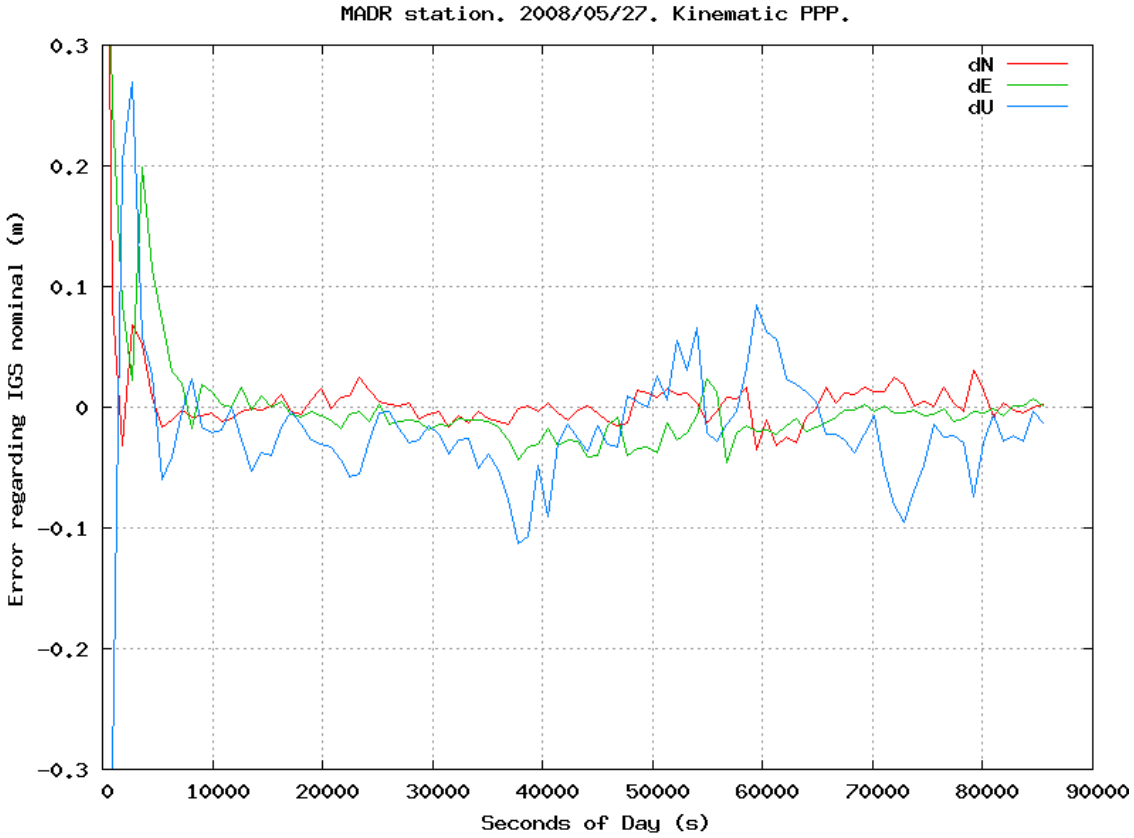

Figure 2: Kinematic PPP positioning error regarding IGS nominal. MADR station, 2008/05/27.

```
1  while( fbpppSolver.LastProcess(gpsData) ) {
2     cout << fbpppSolver.getSolution(TypeID::dLat) << "  ";
3     cout << fbpppSolver.getSolution(TypeID::dLon) << "  ";
4     cout << fbpppSolver.getSolution(TypeID::dH) << endl;
5  }
```

chain in order to compute the appropriate geometric matrix coefficients.

This forward-backward processing is particularly useful to get the zenith path delay estimation (zpd) for the whole processed day. Figure 3 shows the remarkable match achieved when compared with the official, combined IGS zpd.

6 Conclusions

This work has shown how the open source "GPS Toolkit" (GPSTk), coupled with an innovative and flexible GNSS data management strategy called "GDS paradigm", makes it possible to easily develop GNSS data processing techniques, including high accuracy carrier-phase positioning techniques such as Precise Point Positioning (PPP).

Besides, the GPSTk provides classes to implement non-trivial algorithms such as tides modelling (solid, ocean and pole-related), advanced exception handling, configuration files reading, and full Antex file support, among others.

The resulting code of using the GDS and its processing paradigm is remarkably clean, compact and easy to follow, yielding better code maintainability and supporting the overall GPSTk design goal of "freeing researchers to focus on research".

The GPSTk is actively maintained, and there are several lines of work being currently pursued, among

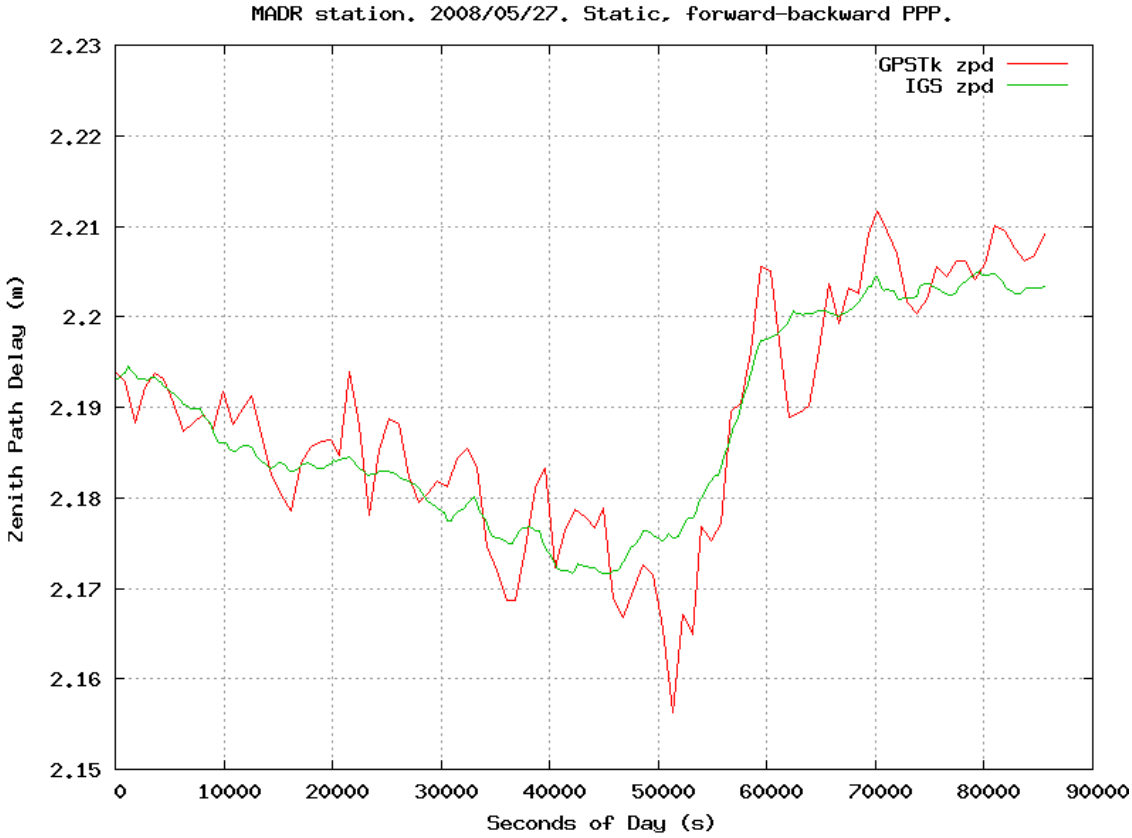

Figure 3: Zenith path delay. MADR station, 2008/05/27.

them: RINEX version 3 handling, more carrier phase-based processing classes (including RTK), IONEX files processing, robust outlier detection, etc. We warmly invite the GNSS community to join us as developers, testers or plain users, and to take advantage of our code base.

References

[1] GPSTk Project Website. http://www.gpstk.org.

[2] B. Tolman et al., 2004. "The GPS Toolkit – Open Source GPS Software". Proceedings 17th International Technical Meeting of the Satellite Division of the ION (ION GNSS 2004). Long Beach, California.

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

[4] J. Kouba, and P. Heroux., 2001. "Precise Point Positioning Using IGS Orbit and Clock Products". GPS Solutions. Vol. 5, pp. 2-28.

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

[6] GPS Toolkit Software Library Documentation. http://www.gpstk.org/doxygen/.

[7] IGSMAIL-5272. "Switch the absolute antanne model within the IGS." http://igscb.jpl.nasa.gov/mail/igsmail/2005/msg00193.html.

[8] Ocean tide loading provider. http://www.oso.chalmers.se/~loading/.