High accuracy positioning using carrier-phases with the open source GPSTk software.

Salazar, D., Hernandez-Pajares, M., Juan, J.M., Sanz, J.

Grupo de Astronomia y Geomatica (gAGE), Universitat Politecnica de Catalunya

C/Jordi Girona 31, C-3, 2nd Floor, 209-210. 08034. Spain.

Email: Dagoberto.Jose.Salazar@upc.edu

ABSTRACT

The objective of this work is to show how using a proper GNSS data management strategy, combined with the flexibility provided by the open source "GPS Toolkit" (GPSTk), it is possible to easily develop both simple code-based processing strategies as well as basic high accuracy carrier-phase positioning techniques like Precise Point Positioning (PPP).

INTRODUCTION

The 'GPS Toolkit' (GPSTk) project [1] is an actively maintained international open source project initiated by the Applied Research Laboratories (ARL:UT). It aims to free researchers from implementing common GNSS algorithms, providing proven GNSS data processing classes. 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. Besides, some parts of it are reported to run in such disparate platforms as the Nokia 770 Internet Tablet and the Gumstix line of full function miniature computers [2].

GNSS DATA MANAGEMENT IN THE GPSTk

After developing the first code-based GNSS data processing classes, and starting to add carrier 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, as was often the case.

Therefore, a new set of data structures were developed, called *GNSS Data Structures* (GDS). These structures hold several kinds of GNSS-related data, properly indexed by station, epoch, satellite and type. In this way, both the data and corresponding metadata is preserved, and data management issues are properly addressed.

GNSS Data Management example: Rinex Observation Data

In order to better understand how GDS work, let's review the typical structure of a RINEX observation file. The data structure for any given RINEX epoch record may be modelled as an “inverted tree”, as shown in Fig. 1. We can see that the data in a RINEX observation file is organized using a hierarchy of indexes providing access to any given value. Data values themselves are not shown: they are "attached" to indexes in the bottom level of the tree.

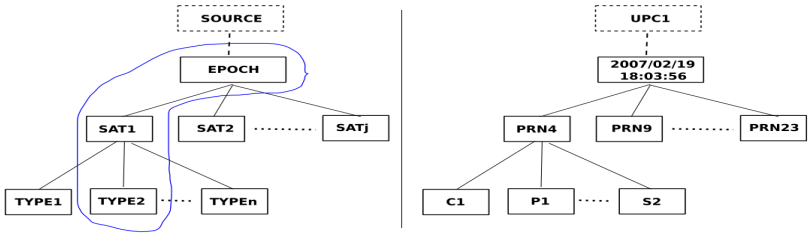

Fig. 1. Typical single-epoch RINEX data structure. General model at left, and indexes example at right.

Traversing the tree in a top-down direction, the first level is the **epoch** (time), the second level corresponds to **satellite PRN** (one satellite per row in RINEX observations files), and then comes the **type** the observation belongs to (related to each column in the data file). The right part of Fig. 1 shows an example of how it may look with some indexes set.

Please note that on top of the tree there is an *implicit* index: The **source**. Each RINEX observation file usually stores the data from one GNSS receiver only (note that from RINEX Version 2 onwards it is allowed to include observations from more than one site, but it is not recommended [3].). Also, please note the fact that for this data structure, the **source** and **epoch** indexes are common for all the values, whereas the **satellite PRN** and **data type** indexes are data-specific.

We want to emphasize that only four indexes were needed to fully identified each RINEX data value: *satID*, *typeID*, *sourceID* and *epochID*. Besides, with a careful implementation of these indexes (in particular *satID*), several types of GNSS (GPS, Galileo, Glonass, etc.) may be transparently handled with these data structures.

GNSS Data Management example: GNSS Signal Propagation Model

Typically, the signal propagation model for pseudorange processing is like the following:

$$P^j_i = \rho^j_i + c(dt_i - dt^j) + rel^j_i + T^j_i + \alpha_f I^j_i + K^j_{f,i} + M^j_{P,i} + \varepsilon^j_{P,i} \quad (1)$$

P^j_i: Pseudorange observation for Space Vehicle SV^j from receiver i (Rx_i).
ρ^j_i: Geometric distance between SV^j and Rx_i.
dt^j: Offset of SV^j clock with respect to GPS Time (GPST).
dt_i: Offset of Rx_i clock with respect to GPST.
rel^j_i: Bias due to relativistic effects.
T^j_i: Tropospheric delay.
$\alpha_f I^j_i$: Ionospheric delay. This effect is frequency-dependent ($\alpha_f = 40.3 / f^2$ [m^3/s^2]).
$K^j_{f,i}$: Frequency-dependent term due to the instrumental delays in SV^j and Rx_i electronics.
$M^j_{P,i}$: Multipath effect. It is environment-dependent, including frequency and code dependencies.
$\varepsilon^j_{P,i}$: Noise and unmodelled effects for code measurements. It is code-dependent.

Each term of (1) is identified (in general) by its type (P^j_i, ρ^j_i, rel^j_i, etc.), receiver it belongs to (i) and satellite (j). Please note that in this case **epoch** index (*epochID*) is implicit. Also, it is important to bear in mind that in this case **data types** *are beyond* the typical RINEX observables; 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. Please consult Fig. 2.

Comparing Fig. 2 with Fig. 1 we can confirm that, although the data structures are different, the same four basic indexes are used, and the major difference lies in the number of indexes that are common to the values.

GDS Implementation

Like in the former examples, several other types of GNSS-related data structures share these 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**.

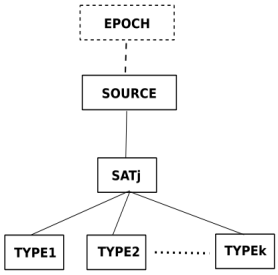

Fig. 2. Representation of a GNSS propagation model.

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 [4].

On the other hand, the GDS themselves are implemented taking advantage of the general "inverted tree" structure: The "trunk" (called *header*) comprises the information common to all the GNSS data values stored inside the structure, and the "branches" (*body*) include the GNSS data values plus the indexes needed to access them. Several types of GDS are provided, implementing different combinations of common and specific indexes, as well as their handling methods.

GDS PARADIGM

The GDS structures are complemented with several associated *processing* classes, completing what is called "the *GDS Paradigm*". 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.

With the *GDS paradigm* the GNSS data processing becomes like an *assembly line*, and the GDS are treated like *white boxes* that *'flow'* from one *'workstation'* (processing class) to the next in such assembly line. This approach allows for a clean, simple-to-read, simple-to-use software code that speeds up development and minimizes errors.

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, prefit residuals, and so on. It will also automatically remove those satellites missing critical data (as ephemeris, for example). Other example processing class is *CodeSmoother*, whose objects will take as parameters a given code observable type and a maximum window size, reading the corresponding code and phase (as well as the cycle slip flag) from the GDS. Then they will compute a new smoothed observable (self adjusting the window size along the way), and will replace the original observable with the new one.

The former ideas are coupled with a handy redefinition of C++ operator "**>>**", implemented in such a way that several operators may be concatenated, allowing a programming style that clearly shows how the data is *flowing* along the processing steps (Indeed, it resembles the typical *pipes* used in UNIX shell programming). Some examples follow.

PROCESSING CODE-BASED GNSS DATA WITH THE GPSTk AND GDS

First, let's proceed with a few lines of code implementing the core of a program making a standard, C1-based GPS data processing, solving with a plain Least-Mean-Squares (LMS) solver. Results are given in a XYZ reference frame:

```
1   while ( rinexFile >> gpsData ) {                          // Get data out of RINEX into GDS
2      gpsData >> limitsFilter >> model >> lmsSolver;         // GNSS data processing line
3      cout << lmsSolver.getSolution(TypeID::dx) << " ";      // Print results
4      cout << lmsSolver.getSolution(TypeID::dy) << " ";
5      cout << lmsSolver.getSolution(TypeID::dz) << endl;
6   }
```

Line #1 is a "*while loop*" that each time will take one full epoch of data out of the RINEX observations file (handled by *RinexObsStream* object "*rinexFile*") and will insert such data into the GDS called "*gpsData*" (provided by class *gnssRinex*). This will be done as long as there are epochs to process in the RINEX file, and line #6 closes the loop.

The data processing itself is done in line #2: *gpsData* is fed into *limitsFilter* (an object from *SimpleFilter* class) that checks if C1 is within reasonable limits. C1 values that don't meet the limits are taken out from *gpsData*, and the result is then fed into *model* (an object from aforementioned *ModeledPR* class). Finally, the expanded *gpsData* (including model-related data along the original observations) is taken by the processing line into *lmsSolver*, an object that belongs to *SolverLMS* class and solves the equation system. That finishes data processing.

Lines #3, #4 and #5 are just used to print the XYZ solution to screen. Please note that each solution is referred to using its corresponding *TypeID* indexing object. This is not strictly necessary (there are several ways to refer to the solution, according to user preferences), but this way is preferred because of ease of use and consistence with GDS paradigm. In the former piece of code the initialization phase (and the user interface part) were not presented for brevity sake. Please consult GPSTk API and provided examples for details.

If the previous results were needed in a North-East-Up (NEU) reference frame, the system may be solved using *dLat*, *dLon* and *dH* data types, instead of *dx*, *dy*, *dz*. Modifying the core GNSS data processing line, we will have:

```
gpsData >> limitsFilter >> model >> baseChange >> neuSolver;
```

In this case, object *baseChange* (from *XYZ2NEU* class) computes the corresponding *dLat*, *dLon* and *dH* coefficients and adds then to *gpsData*. Also, *neuSolver* is a *SolverLMS* object but configured to solve a NEU equation system.

Now a more complex example: We want to process a smoothed ionosphere-free code observable (PC) and solve the equation system using a Weighted-LMS solver. First, get PC, LC, LI (geometry-free) and MW (Melbourne-Wübena) combinations, and with that information we mark cycle slips (using two different and complementary methods). Then smooth PC combination and compute the position using such combination and a Weighted-LMS method based in MOPS weights and MOPS tropospheric model. Everything is in a single C++ processing line:

```
gpsData >> getPC >> getLC >> getLI >> getMW >> markCSLI >> markCSMW >> smoothPC
        >> limitsFilter >> modelMOPS >> weightsMOPS >> baseChange >> wmsSolver;
```

Objects *getPC*, *getLC*, *getLI* and *getMW* belong to special classes to get those combinations (also, there is a general *ComputeLinear* class that may be used for this purpose). *markCSLI* and *markCSMW* belong to cycle slip detection classes, *smoothPC* belongs to aforementioned *CodeSmoother* class, *weightsMOPS* comes from *computeMOPSWeight* class, *wmsSolver* belongs to *SolverWMS* class,and the rest is as explained in the previous example.

The last code-based example applies differential GPS techniques. We start partially processing data from the reference station (*gpsDataRef*), and assigning the resulting GNSS data structure as the reference data of a *DeltaOp* object (*delta*):

```
gpsDataRef >> synchro >> limitsFilter >> modelReference;    // Partial reference data processing
delta.setRefData(gpsDataRef.body);                          // Single-differences object
```

Please note the new *synchro* object. This object belongs to *Synchronize* class, and it is preconfigured to take care of synchronization between *gpsDataRef* and *gpsData*. Afterwards, the rover receiver data (*gpsData*) is fully processed:

```
gpsData >> limitsFilter >> model >> weightsMOPS >> delta >> baseChange >> wmsSolver;
```

Object *delta* will substract, from *gpsData* prefit residuals, the corresponding *gpsDataRef* prefit residuals, deleting (by default) satellites not common to both receivers. The rest of the data processing is as shown before.

Fig. 3 shows the results of the last three processing strategies, plotting horizontal error regarding nominal position for EBRE station (2002/01/30). In the DGPS case we use BELL as reference station (a 115 km long baseline).

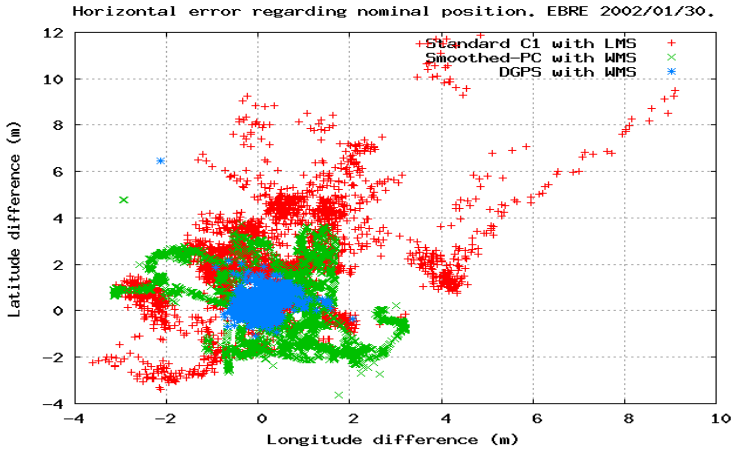

Fig. 3. Horizontal error regarding nominal position for three code-based data processing strategies

PROCESSING CARRIER PHASE-BASED GNSS DATA WITH THE GPSTk AND GDS

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 orbits and clocks are used in PPP, but these products are usually provided each 900 s, while observations are usually provided each 30 s. Therefore, some time management issues arise.

We present the core processing part, with explanations at right side:

```
1   corr.setExtraBiases(tides);      // Set correcting object with tides information, RX antenna
2                                    //     phase center, and eccentricity
3   gpsData >> basicModel            // Compute the basic components of model
4           >> eclipsedSV            // Remove satellites in eclipse
5           >> gravDelay             // Compute gravitational delay
6           >> svPcenter             // Compute the effect of satellite phase center
7           >> corr                  // Correct observables from tides, RX phase centers, etc.
8           >> windup                // Compute wind-up effect
9           >> computeTropo          // Compute tropospheric effect (Niell model)
10          >> linearCombination1    // Compute common linear combinations (PC, LC, LI, MW...)
11          >> limitsFilter          // Filter out spurious data
12          >> markCSLI              // Mark cycle slips: LI algorithm
13          >> markCSMW              // Mark cycle slips: Melbourne-Wübbena algorithm
14          >> markArc               // Keep track of satellite arcs
15          >> phaseAlign            // Align phases with codes
16          >> linearCombination2    // Compute prefit residuals
17          >> decimateData          // If not a multiple of 900 s, decimate data
18          >> baseChange            // Prepare to use North-East-UP reference frame
19          >> computeDOP            // Compute DOP figures
20          >> pppSolver;            // Solve equations with a Kalman filter
```

The GDS processing data chain is indeed a single C++ line, although in this case (for clarity sake) spans from line #3 to line #20. Several of these objects need initialization, but that part is omitted here (consult GPSTk examples and API).

Particular mention deserves object *pppSolver*, belonging to *SolverPPP* class. This object is preconfigured to solve the PPP equation system in a way consistent with [5]: Coordinates are treated as constants (static), receiver clock is considered white noise, and vertical tropospheric effect is processed as a random walk stochastic model. Fig 4. plots the results of PPP processing for ONSA station (2005/08/12), compared to IGS nominal position. Biases up to of a few centimeters are normal when comparing with IGS SINEX products.

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:

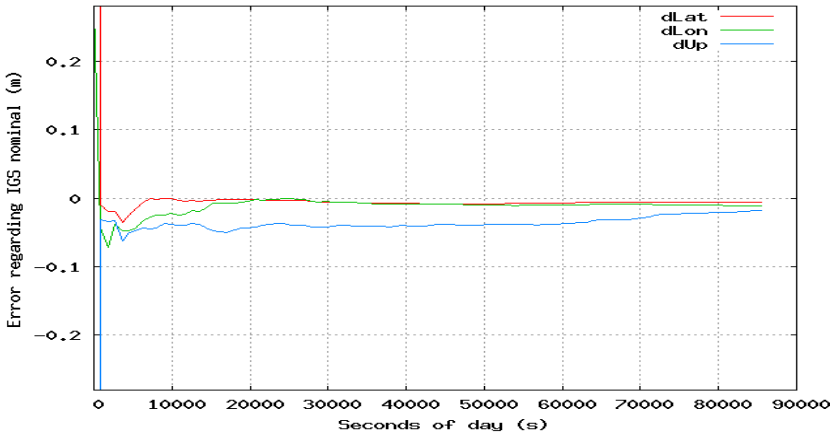

Fig. 4. PPP processing for ONSA station (2005/08/12). Forward filter with static coordinates

```
WhiteNoiseModel newCoordinatesModel(100.0);          // 'newCoordinatesModel' has a sigma of 100 m
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. Fig. 5. presents the results for this type of processing, confirming the good quality of GPSTk model (results are within +/- 10 cm from nominal value).

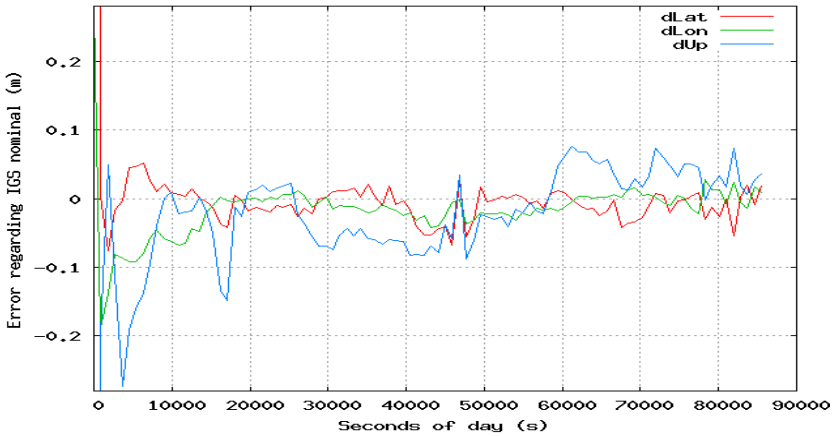

Fig. 5. PPP processing for ONSA station (2005/08/12). Forward filter with kinematic coordinates

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". 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 for 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 *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* how many forward-backward cycles we want it to "re-process". For instance:

```
fbpppSolver.ReProcess(4);                    // Carry out 4 forward-backward cycles
```

After that, one last forwards processing is needed to get the time-indexed solutions out of *fbpppSolver* (see Fig. 6):

```
while( fbpppSolver.LastProcess(gpsData) ) {                    // One last forwards processing
   cout << fbpppSolver.getSolution(TypeID::dLat) << "   ";     // Print dLat, dLon and dUp
   cout << fbpppSolver.getSolution(TypeID::dLon) << "   ";
   cout << fbpppSolver.getSolution(TypeID::dH) << endl;
}
```

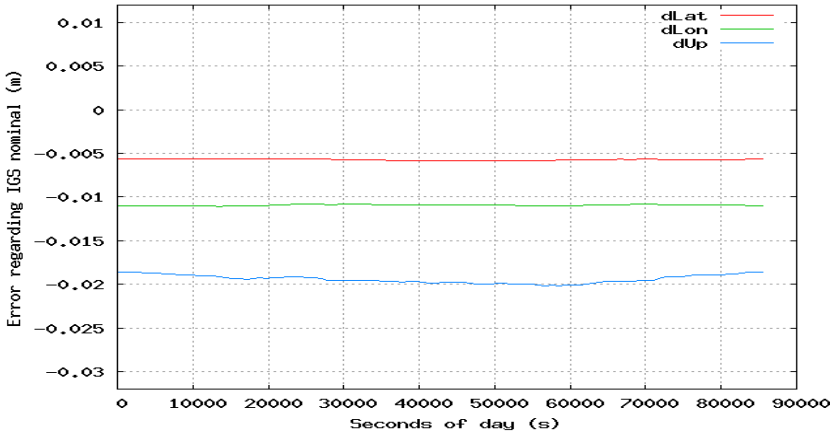

Fig. 6. PPP processing for ONSA station (2005/08/12). Forward-backward filter with static coordinates

Carrier phase-based single-differences processing with broadcast orbits and clocks

Finally, we may combine the techniques presented for code-based DGPS with the ones for PPP, and with relatively minor modifications implement carrier phase-based DGPS processing with broadcast orbits and clocks:

```
1   corr.setExtraBiases(tides);            // Set correcting object with tides information,
2                                          //    RECEIVER antenna phase center, and eccentricity
3
4   refCorr.setExtraBiases(reftides);      // Set correcting object with tides information,
5                                          //    REFERENCE antenna phase center, and eccentricity
6
7   // The following lines process reference station data
8   gpsRefData >> synchro >> refBasicModel >> eclipsedSV >> refGravDelay >> refSvPcenter
9              >> refCorr >> refWindup >> refComputeTropo >> linearCombination1
10             >> refMarkCSLI >> refMarkCSMW >> markArc >> linearCombination2;
11
12  delta.setRefData(gpsRefData.body);     // Object in charge of computing single differences
13
14  // The following lines process receiver data
15  gpsData >> basicModel             // Compute the basic components of model
16          >> eclipsedSV             // Remove satellites in eclipse
17          >> gravDelay              // Compute gravitational delay
18          >> svPcenter              // Compute the effect of satellite phase center
19          >> corr                   // Correct observables from tides, etc.
20          >> windup                 // Compute wind-up effect
21          >> computeTropo           // Compute tropospheric effect (Niell model)
22          >> linearCombination1     // Compute common linear combinations (PC, LC, LI, MW,...)
23          >> limitsFilter           // Filter out spurious data
24          >> markCSLI               // Mark cycle slips: LI algorithm
25          >> markCSMW               // Mark cycle slips: Melbourne-Wubbena algorithm
26          >> markArc                // Keep track of satellite arcs
27          >> phaseAlign             // Align phases with codes
28          >> linearCombination2     // Compute prefit residuals
29          >> delta                  // Compute simple differences between code and phase prefits
30          >> baseChange             // Prepare to use North-East-UP reference frame
31          >> computeDOP             // Compute DOP figures
32          >> pppSolver;             // Solve equations with a Kalman filter
```

Lines #8 through #10 is a single C++ code line, in charge of processing reference station data. Special mention deserves object *synchro* (synchronizing data between receivers), as well as 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 (some objects need the nominal position of a specific receiver, for instance), and partly because objects like cycle slip detectors are state-aware and must not be shared between different processing streams.

On the other hand, lines #15 through #32 (a single C++ code line, indeed) are very similar to the PPP example, with the important addition of object *delta* in line #29, taking care of computing single differences. Please note that the former *decimateData* object is no longer needed: we are now using broadcast orbits and clocks, and therefore we can work at arbitrary sampling rates. Fig. 7. presents the results for this carrier phase-based differential processing with static coordinates and floated ambiguities, comparing it with the results of previous code-based DGPS. Be mindful that better results could be obtained using RTK techniques (not covered here), but usually those techniques are limited to baselines shorter than 20 km. Please note the change of vertical scale at right.

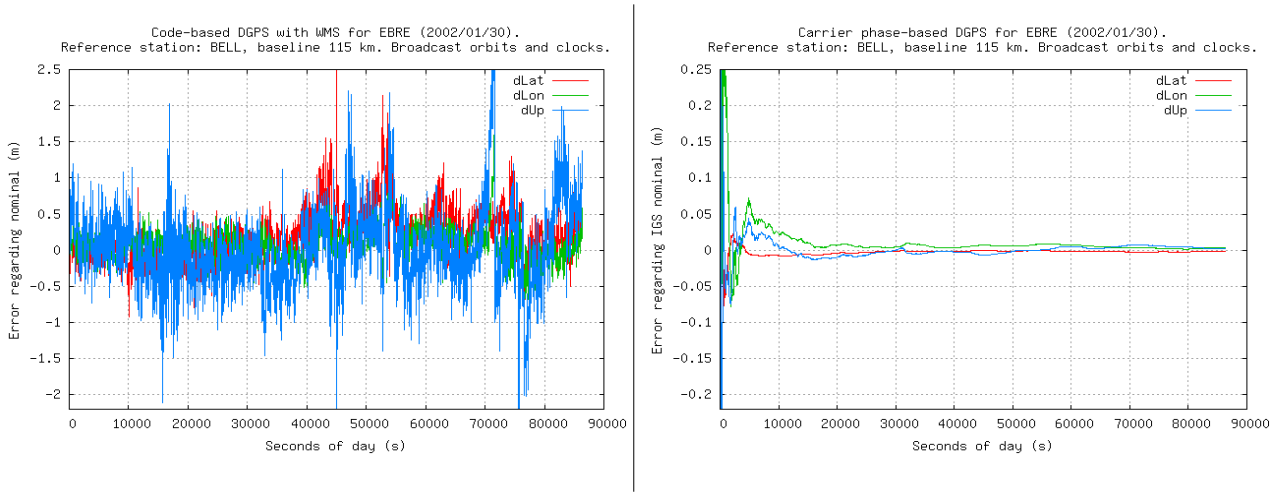

Fig. 7. Comparison of code-based and carrier phase-based DGPS processing for EBRE. Broadcast orbits and clocks

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, from simple standard C1-based strategies, to high accuracy carrier-phase positioning techniques such as Precise Point Positioning (PPP) and differential carrier phase-based positioning.

The GPSTk is actively maintained, and there are several lines of work being currently pursued, among 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 and take advantage of our code base.

REFERENCES

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

[2] D. Salazar, M. Hernandez-Pajares, J.M. Juan, and J. Sanz. "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. December 2006.

[3] Gurtner, W. (2001). “RINEX: The Receiver Independent Exchange Format Version 2.10”. Astronomical Institute, University of Berne.

[4] “GPS Toolkit Software Library Documentation”. http://www.gpstk.org/doxygen/.

[5] J. Kouba, and P. Héroux. “Precise Point Positioning Using IGS Orbit and Clock Products”. *GPS Solutions*. vol.5, pp. 2-28. October, 2001.