Python GNSS Receiver

An Object-Oriented Software Platform Suitable for Multiple Receivers

Eliot Wycoff, Yuting Ng, and Grace Xingxin Gao

INNOVATION INSIGHTS
with Richard Langley

Pull

TEXT. Text.
Text

“Innovation” is a regular feature that discusses advances in GPS technology and its applications as well as the fundamentals of GPS positioning. The column is coordinated by Richard Langley of the Department of Geodesy and Geomatics Engineering, University of New Brunswick. He welcomes comments and topic ideas. To contact him, see the “Contributing Editors” section on page 4.

With billions of GNSS-enabled devices in use today, the potential gains from harnessing data collected over a network of GNSS receivers has never been greater, yet the necessary architectures to handle and extract useful data collected over such networks are not well explored. Traditional uses of GNSS in cooperative positioning treat individual GNSS receivers as “black boxes” that merely output navigation solutions. As such, the wealth of information contained in each receiver’s raw signals is largely discarded.

Of particular interest are ideas such as inter-receiver aiding, in which networked receivers might share acquisition, tracking, and navigation information (possibly in real time) to improve receiver performance. In addition, a network of receivers might also be used as a sensing tool: it is expected that atmospheric parameters, for instance, could be recovered by analyzing the raw signal data arriving at an appropriately sized network.

In light of these interesting research areas, it would be expedient to develop a set of tools that can process and handle the raw data being produced at every receiver in a GNSS receiver network. Existing software-defined receivers (SDRs) have gone a long way towards making the fast prototyping of new receiver architectures possible. An SDR attempts to shift as many receiver functions, such as mixing and tracking, from being implemented in hardware to being implemented in software. This allows for fast prototyping as receiver components can be more quickly modified in software than in hardware. The hardware components that a GNSS SDR still requires are an antenna and a front end including an analog-to-digital converter (ADC). An analog GNSS signal is received at the antenna. It is then mixed to an intermediate frequency and digitized by the ADC. The digital stream is then processed by the SDR's software component.

But with regard to processing data from a receiver network, existing SDRs have a number of notable flaws. In brief, existing software receivers are designed to process the data arriving at one real-world receiver. Thus a procedural coding design is typically used. While procedural code is a good solution for the linear processes that occur in a single receiver (acquisition, tracking, demodulation of the navigation data, position calculations, and so on), this software design style does not adapt well to the task of performing all of these actions on multiple receivers with the additional goal that each receiver shares tracking data with every other one. In such scenarios, not only is there data being produced for every receiver in the network, but there is also data being produced about the relationships between the receivers in the network. Thus, an SDR that was originally designed to process data from only one receiver will prove difficult to adapt to the task of processing many.

Luckily, object-oriented programming, a well-known and widely used software design philosophy, is well suited to the receiver network problem. Therefore, for this work, we designed and implemented an object-oriented software platform for many receivers. Python was chosen as the programming language because of its support for object-oriented programming, its portability, its free cost, its numerical abilities (using open-source libraries such as NumPy and SciPy), and its ease of use. And as a reference, an existing Matlab software receiver was used as a basis for developing many of the core algorithms in this work. We call our development simply the Python Receiver.

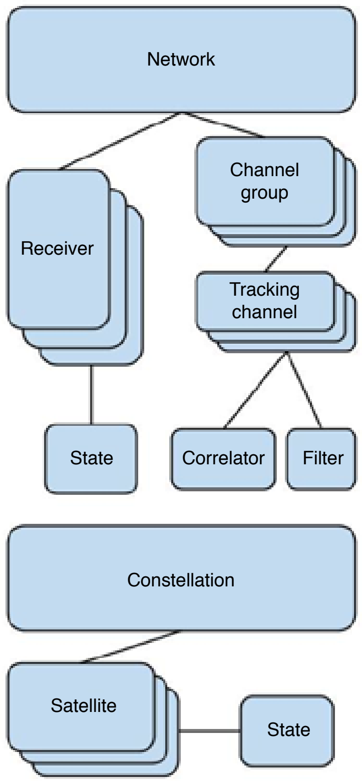
Design

Many of the core functions in the Python Receiver are modeled after those found in the Matlab development. Thus, this particular implementation is suited for the raw GPS L1 signal data mixed to an intermediate frequency by the SDR front end. In addition, the basic algorithms for acquisition, scalar tracking, and navigation are similar to the Matlab ones, with the exception that acquisition is made more robust by using multiple noncoherent integrations. The primary innovation of this software, however, is in the way in which the code is organized. For tracking multiple receivers, the Python Receiver was designed under an object-oriented approach.

FIGURE 1 illustrates the main objects that a user would be expected to use in the Python Receiver. Each object is defined as a class, and as such each object is capable of storing object-specific data as well as performing certain object-specific functions. The hierarchy of Figure 1 roughly illustrates which objects are defined as members of other classes for typical usage. Thus, inside any instance of the network class may exist any number of receiver objects. Likewise, an instance of the constellation class may be home to any number of satellite objects.

For data coming from a single real-world receiver, use of the Python Receiver would typically be as follows. First, a user would initialize an instance of the receiver class using a dictionary of predefined settings, such as the file location of the data source. Second, the user would initialize a constellation object of satellites by passing the pseudorandom noise (PRN) code values of each satellite to be included in the constellation. At this point, the user could then use built-in functionality in the receiver object to perform acquisition of all of the satellites in the constellation. Results of this acquisition attempt would be stored in the receiver object, where they could then be used to run the receiver's built-in scalar tracking functionality. Likewise, scalar tracking data would be stored in the receiver object, and again the user could use the receiver's built-in navigation functionality to decode the navigation bits produced during scalar tracking and perform navigation computations. Satellite-specific ephemerides would be stored in the relevant satellite objects.

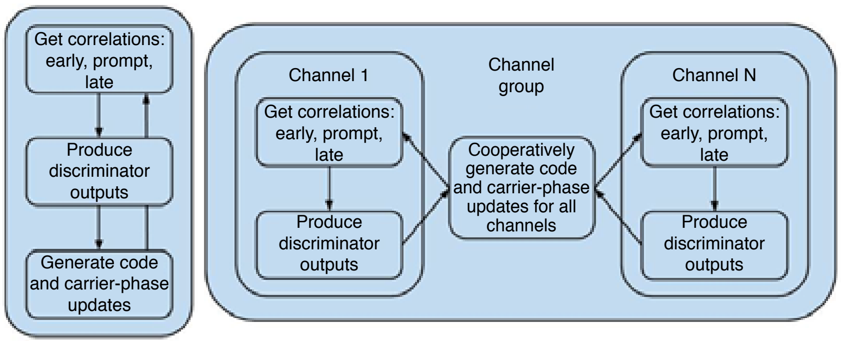
Navigation solutions are stored as a part of the receiver's state object. The state object, which is also used in the satellite class, is a container for holding



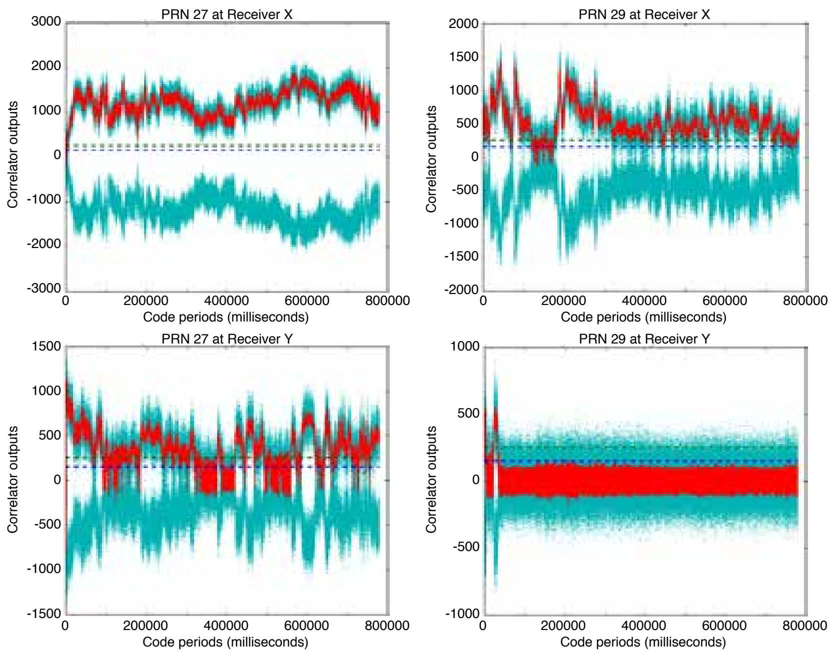
▲ **FIGURE 1** Typical object (class) hierarchy.

state information in the Earth-centered Earth-fixed (ECEF) coordinate system (such as position and velocity) and clock terms, and it also provides the ability to return position coordinates in other systems, such as the GPS geodetic system (frame) of WGS 84. While it is not a key feature of the Python Receiver, the state object is designed as an object so that it can be readily used elsewhere should an algorithm need to store state information and have coordinate transformations readily available.

Tracking channels need not be restricted to the hierarchy shown in Figure 1. During operation for just one data source, the scalar tracking function defined at the receiver level will initialize a sufficient number of tracking channels to track all of its observed satellites. However, when operating on multiple sources of data and with the intent to share tracking outputs between channels, it is helpful to place tracking channels into groups, as shown in **FIGURE 2**. In the example that will be discussed in following sections, two real-world receivers observed a


Get correlations: early, prompt, late
Produce discriminator outputs
Generate code and carrier-phase updates
Channel 1
Channel group
Channel N
Cooperatively generate code and carrier-phase updates for all channels

▲ **FIGURE 2** Left: an independent tracking channel (corresponding to one tracking channel object). Right: a channel group. Note that in the channel group, updates to the code and carrier phase of each channel may be performed cooperatively.



▲ **FIGURE 3** The in-phase prompt correlator outputs for both receivers and satellites PRN 27 and 29. The cyan dots are correlator outputs, the red line is the locking metric, and the dashed green and blue lines are the thresholds set for determining good and poor lock, respectively. Locking metric values above the dashed green line represent a good lock, and values below the dashed blue line represent loss-of-lock. Note that y-axis values differ from graph to graph.

similar set of satellites. It was therefore helpful to define channel groups for each commonly observed satellite, with one channel in the group corresponding to the satellite as tracked by the first receiver, and the other channel corresponding to the satellite as tracked by the second. Tracking groups as a class, however, may be easily modified for other experimental purposes.

Independent tracking channels have an update function that processes the next segment of raw data in three main steps: computing correlations (early, late, and prompt), producing discriminator outputs, and generating code and carrier-frequency updates. For a group of channels, this sequence of steps is interrupted after discriminator outputs have been computed. At this point, the channel group may instruct the tracking channels to update their code and carrier frequencies independently or through some other cooperative means that considers data across all of the channels.

As for the last few classes: correlators and filters are defined as objects so that they can be easily changed depending on the experimental circumstances. And satellites, in addition to holding satellite-specific ephemerides, have built-in functionality to return their locations given a particular epoch of GPS Time.

Naturally, core functions such as these would be found in traditional software receivers, but by repackaging them into the object-oriented framework, both code reusability and modifiability increase. And in addition, by defining classes for networks of receivers and groups of tracking channels, simulations and experiments involving cooperative positioning of receivers become easier to conduct.

Experiment

To help illustrate how the Python Receiver lends itself to the task of cooperatively tracking multiple receivers, concurrent data from two SDR front ends was collected on a boat in Lake Titicaca just offshore from Puno, Peru. The boat was a small motorized ferry capable of transporting approximately twenty passengers. One antenna and front end, hereafter referred to as "Receiver X" was placed on the port side of the boat, while the other, "Receiver Y" was placed on the starboard side. Maintaining a fixed baseline, both receivers captured raw GPS L1 signals from separate portions of the sky and mixed them to an intermediate frequency of 5.456 MHz. Raw data collection was performed concurrently at both receivers for 15 minutes as the boat returned from the floating islands of the Uros people to the dock at Puno. Finally, while Lake Titicaca is at a high elevation in the Altiplano (the Andean Plateau), the surrounding mountains do not rise far above the horizon, and thus visibility was quite good in most directions.

Some challenges, however, present themselves in this data set. While Receiver X was able to acquire eight satellites, and Receiver Y was able to acquire 10, the signal quality at Receiver Y was generally poor. In **FIGURE 3**, in-phase prompt correlator outputs from traditional scalar tracking are shown for both Receivers X and Y and satellites with PRN codes 27 and 29. For satellite 27, Receiver Y loses lock of the signal

Satellite	Independent scalar tracking lock frequency		Cooperative scalar tracking lock frequency	
	X	Y	X	Y
PRN 3	99.98%	88.76%	99.98%	88.60%
PRN 6	99.83%	98.71%	99.82%	98.72%
PRN 18	99.89%	91.31%	99.89%	91.61%
PRN 22	99.99%	99.96%	99.99%	99.96%
PRN 27	99.73%	84.40%	99.73%	84.66%
PRN 29	98.71%	02.53%	98.72%	71.62%

▲ **TABLE 1** Percent of time each tracking channel spent locked. Lock was designated if the locking metric was above 150. The best values for Receiver Y are highlighted in green, with the most notable improvement occurring for satellite 29.

between code periods 100,000 and 200,000, and for satellite 29, it completely loses track of the signal after only a few thousand code periods. (Recall that the C/A-code period is one millisecond.)

To better characterize the tracking performance of each receiver-satellite pair, a locking metric was designed and implemented, the values of which are shown as the red lines in the graphs of Figure 3. Inspired by the earlier use of the square-law detector, we have expressed the metric as:

$$\sqrt{\frac{\sum_{i=1}^{N}\left(I_i^2 - Q_i^2\right)}{N}} \qquad (1)$$

where N is the number of most recent correlator samples, I_i and Q_i are the ith in-phase and quadrature-phase prompt correlator outputs, and the square-root operator returns the negative square root of the absolute value of the expression under the radical if that expression is negative.

After visually examining the relationship of this locking metric with the quality of the in-phase prompt correlator outputs, two thresholds were determined in order to better characterize the quality of the tracking loop lock. The first threshold, represented as the dashed green lines in the graphs of Figure 3, is the threshold above which the tracking loops were considered locked well. Its value was set to 250. The second threshold, whose value was set to 150 and is represented by the dashed blue lines, is the threshold below which the tracking loops were considered to be in a complete loss-of-lock situation. Locking metric values between 150 and 250 were considered as representing a situation in which the tracking loops were weakly locked to the incoming signals.

Despite the poor performance of Receiver Y in tracking many of its signals, navigation functionality in the Python Receiver was still able to recover sufficient ephemerides from the tracking data to perform position calculations. **FIGURE 4** shows the navigation solutions for Receiver Y over a 13-minute interval, roughly capturing the route that the ferry took westward back to Puno. Note that the moustache-shaped region in the right-hand side of the map is the collection of floating islands of the Uros. Just as the ferry left these islands, the navigation solutions for Receiver Y become much nosier. Possible reasons for this are the slight change in heading that the ferry made, or the thicket of reeds that surrounded the boat during this portion of the journey. Navigation results for Receiver X were much less noisy.

Cooperative Scalar Tracking

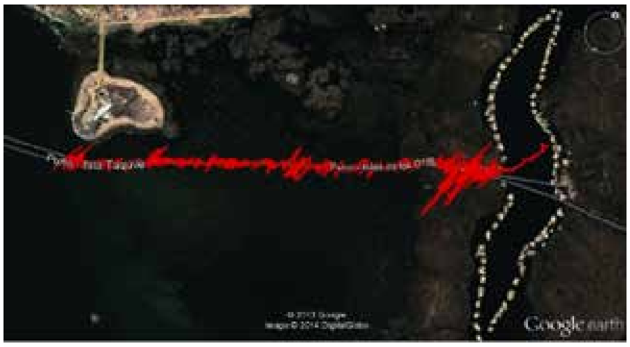
While all of these traditional results were obtained using the Python Receiver, they could have just as easily been obtained using procedurally coded receivers. Assuming, however, that one is interested in performing experiments that involve data sharing between multiple receivers, the Python Receiver lends itself handily to the task.

An experiment was devised in which scalar tracking performed at both Receivers X and Y would be done cooperatively. In particular, it was observed that often when one of the two receivers momentarily lost track of its signal for a particular satellite, the other receiver would be tracking well. In addition, it was noted that because the two receivers maintained a fixed baseline during tracking, their tracking channels should have maintained a steady difference in code phases that changed slowly provided that the receiver-satellite geometry did not change quickly. As shown in **FIGURE 5**, the only violation of this scenario would occur when one of the two receivers lost lock and thus allowed for drift in its code-tracking loop. It should be noted that unlike the situation in Figure 5, the reported code difference between the two receivers suffered from a bias that grew linearly in time. This bias, which was likely due to clock errors in one or both of the receiver front ends, was eliminated through a linear regression before the plotting of the figure.

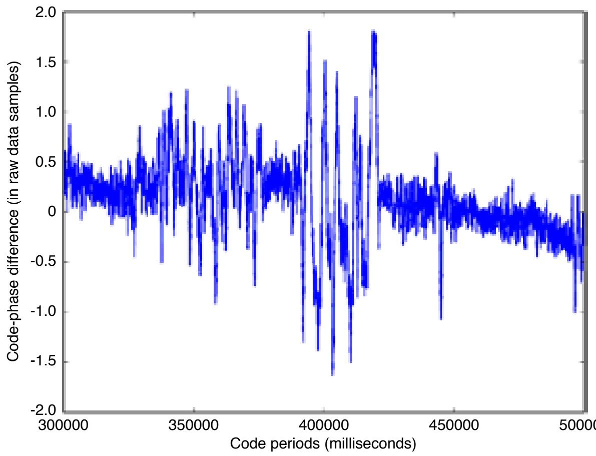
All of these observations motivated the following cooperative scalar tracking design. First, any satellite that was observed by only one receiver would be independently tracked by that receiver in the traditional manner. A single tracking loop object would be allocated in Python for this particular receiver-satellite pair. Second, any satellite that was observed by both receivers would have a channel group object allocated in Python. This channel group would contain two tracking channel objects, one for each receiver.

As shown in Figure 2, this channel group required specific code to be written to handle the cooperative updates of both receivers' code and carrier frequencies. The algorithm was designed as follows. For each update epoch (generated by a call of the channel group's update function), if both of the tracking channels were locked to their incoming signals, the channel group would save their code-phase difference for that code period. And since both channels were locked, both would update their code and carrier frequencies in the traditional manner, relying on discriminator outputs only.

If, on the other hand, one of the tracking channels was in a loss-of-lock situation, the channel group would search the previous 5,000 milliseconds of data for code periods during which, presumably, both tracking channels were mutually locked. This data would contain information about the expected code-phase difference between the two tracking channels at the



▲ **FIGURE 4** The trip back to Puno on the left (west) from the floating islands of the Uros on the right (east) as determined by traditional scalar tracking and navigation at Receiver Y. Image courtesy of Google Earth and the GPS Visualizer.

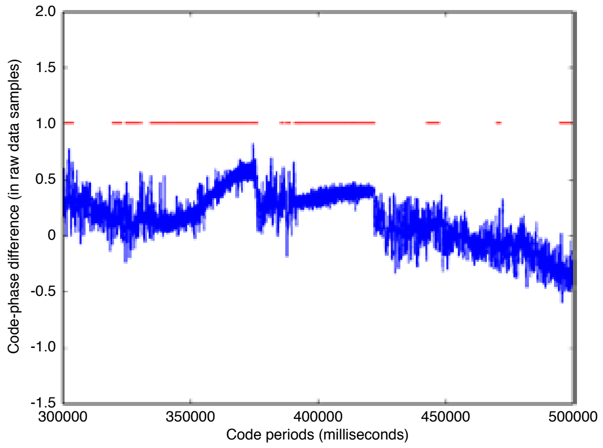


▲ **FIGURE 5** The code-phase difference between Receivers X and Y for PRN 27 from 300,000 to 500,000 milliseconds. Note the large variance around 400,000 milliseconds corresponding to a loss-of-lock for Receiver Y.

current code period. At this point, a linear regression on the data from the mutually locked code periods was used to determine this expected code-phase difference. Finally, we note again that this expected code-phase difference would only remain valid under the assumption that the receiver-satellite geometry was not changing rapidly, as was the case for this data. But acknowledging that some changes in the geometry might occur (such as a change in heading of the boat) is the reason why the search interval for mutually locked data was limited to five seconds.

Assuming that one of the receivers was in a loss-of-lock situation and that sufficient data from the past five seconds existed to generate an estimate of the current expected code-phase difference, the channel group could then make a cooperative update of the lockless tracking channel. For this channel, the channel group would replace the traditional code-tracking discriminator outputs with the offset of the expected code-phase difference d_{exp} from the currently observed code-phase difference d_{cur}. In the following equation, the new discriminator output is denoted as c:

$$c = d_{cur} - d_{\exp}. \tag{2}$$



▲ **FIGURE 6** The code-phase difference between Receivers X and Y for PRN 27 from 300,000 to 500,000 milliseconds, this time using cooperative scalar tracking. Presence of the red line indicates code periods during which cooperative code-phase updates were made for Receiver Y. Note that noisy drifting of the code-phase difference is suppressed.

Expressing $d_{cur}=y_{cur}-x_{cur}$ and $d_{exp}=y_{exp}-x_{exp}$, where $x_{cur/exp}$ and $y_{cur/exp}$ represent current and expected code phases at two receivers, we can rewrite Equation 2 as

$$c = \left(y_{cur} - x_{cur}\right) - \left(y_{exp} - x_{exp}\right) \tag{3}$$

or

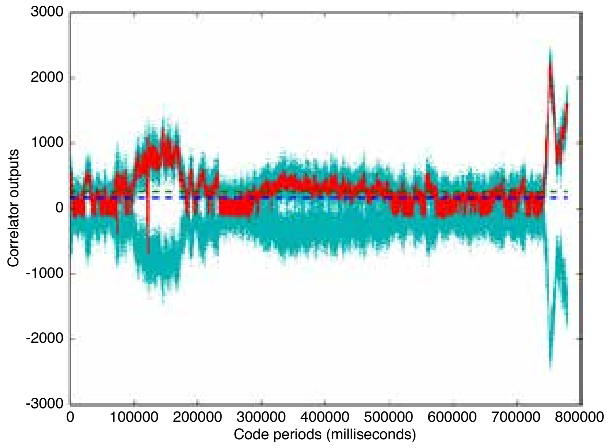
$$c \cong y_{cur} - y_{exp} \tag{4}$$

since we expect the x receiver to be locked, and therefore $x_{cur} \cong x_{exp}$.

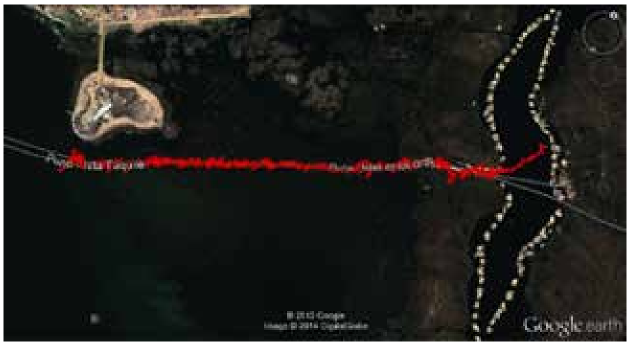
Some finer points to mention include that the “loss-of-lock” and “tracking well” designations were determined by way of the locking metric defined in the previous section. In addition, if a receiver was “tracking weakly,” it would update its code and carrier frequencies by relying solely on its own discriminator outputs. Also, because in traditional scalar tracking loss-of-lock might occur for an extended interval greater than five seconds at one receiver (such as Receiver Y’s tracking of satellite 27 seen in Figure 3 between 300,000 and 400,000 milliseconds), whenever the channel group was called to cooperatively update a lockless tracking channel’s code frequency, it would record the current code-phase difference between both receivers. Under all scenarios, the carrier-frequency update would be done independently at each channel using discriminator outputs alone. And finally, in order for both receivers to share relevant data with each other during tracking, clock bias terms found after traditional scalar tracking were used to align in time the raw data files for each receiver appropriately.

Results and Discussion

Using cooperative scalar tracking, drifting of the code-phase difference during code periods when one of the receivers is experiencing loss-of-lock is expected to be suppressed. And indeed, results such as those shown in **FIGURE 6** verify this expectation. Since cooperative scalar tracking does not attempt to modify the way either receiver tracks during periods of good



▲ **FIGURE 7** The in-phase prompt outputs for Receiver Y and PRN 29 using cooperative scalar tracking. Compare this to the bottom-right graph in Figure 3. Inter-receiver aiding allowed Receiver Y to track this signal for a majority of the code periods.



▲ **FIGURE 8** The trip back to Puno as determined by Receiver Y after cooperative scalar tracking and navigation computations. Compared to Figure 4, the navigation solutions are less noisy. Image courtesy of Google Earth and the GPS Visualizer.

lock, this type of modified scalar tracking is not expected to produce less noisy tracking results. It is expected, however, to help lockless tracking channels to regain track after short signal outages, similar to the benefits of vector tracking.

Strikingly, this form of cooperative tracking allowed for Receiver Y to continually track the signal from satellite 29 (albeit with occasional outages) for the full thirteen minutes of data shown in **FIGURE 7**. Whereas in Figure 3, Receiver Y very quickly loses track of satellite 29, Figure 7 shows that Receiver Y, under cooperative scalar tracking, can maintain a good enough lock on the signal that by roughly 750,000 code periods, it is able to pick up the signal again quite strongly. This change in signal strength may have been due to a slight change in heading that the ferry made near Isla Taquile towards the end of this data set (see Figure 4 and **FIGURE 8**).

Given the locking metric defined in the section "Experiment," quantitative measures of how often each channel spent locked or in loss-of-lock can be made. In total, both receivers tracked six common satellites (with each receiver also tracking other satellites independently). **TABLE 1** shows the locking frequencies for each commonly tracked satellite.

Granted that the drift in the code phase for lockless tracking channels is curtailed in cooperative scalar tracking, an improvement in navigation solutions is also expected. This expectation is verified by comparing the qualitative level of noise in the solutions of Figure 8 to the solutions in Figure 4. Notably, the noise in the reed thicket (the section of the route immediately after leaving the moustache-shaped floating islands region) is suppressed. Not shown are the navigation solutions for the port side receiver, Receiver X, which by comparison to Receiver Y were relatively good in both forms of scalar tracking.

Conclusion

The experiment we carried out highlighted the abilities of the Python Receiver. Data from two SDR front ends and associated antennas placed on either side of a small transport ferry was used to track both receivers by using groups of tracking channels that could cooperatively modify their individual channels' code and carrier frequencies. In this way, loss-of-lock in many of the tracking channels was avoided leading to improved navigation precision. More importantly, it is expected that future experiments like these can be easily implemented within the framework of the Python Receiver, and thus topics like cooperative vector tracking might be more easily investigated.

Acknowledgments

This article is based, in part, on the paper "A Python Software Platform for Cooperatively Tracking Multiple GPS Receivers" presented at ION GNSS+ 2014, the 27th International Technical Meeting of the Satellite Division of The Institute of Navigation, held in Tampa, Florida, September 8–12, 2014.

Manufacturers

The Python Receiver uses SiGe GN3S v3 Samplers, developed by the **University of Colorado** and **SiGe Semiconductor** (acquired by **Skyworks Solutions Inc.** (*www.skyworksinc.com*), Woburn, Massachusetts) and marketed by **SparkFun Electronics** (*www.sparkfun.com*), Niwot, Colorado.

ELIOT WYCOFF received his B.S. in applied mathematics from Columbia University, New York, in 2011. While working on the Python Receiver, he was a graduate student in the Department of Aerospace Engineering at the University of Illinois at Urbana-Champaign (UIUC).

YUTING NG obtained a B.S. in electrical and computer engineering from UIUC in 2014. She is currently a graduate student in the Department of Aerospace Engineering, UIUC.

GRACE XINGXIN GAO is an assistant professor in the Department of Aerospace Engineering, UIUC. She received her B.S. in mechanical engineering in 2001 and her M.S. in electrical engineering in 2003, both from Tsinghua University, China. She obtained her Ph.D. in electrical engineering at Stanford University in 2008. Before joining UIUC in 2012, Gao was a research associate at Stanford University.

MORE ONLINE

Further Reading

For references related to this article, go to *gpsworld.com* and click on *Innovation* in the navigation bar.