LEOGPS

Release 1.3

Samuel Y. W. Low

Sep 22, 2021

GETTING STARTED

1 Installation 3

2 First Steps 5

3 First Scenario Run 9

4 Carrier Single Differencing 11

5 Carrier Double Differencing 15

6 Processing Flow 19

7 API Reference 21

Python Module Index 29

Index 31

Github https://github.com/sammmlow/LEOGPS

Documents https://leogps.readthedocs.io/en/latest/

Version 1.3 (Latest)

Author Samuel Y. W. Low

LEOGPS is an open-source Python software which performs relative satellite navigation between **two formation flying satellites**, with the objective of high accuracy relative positioning. Specifically, LEOGPS solves for the double-differenced baseline (using float ambiguity resolution) between satellites flying in formation in Low Earth Orbit (LEO). As such, the relative positioning accuracy diminishes with increasing formation baseline lengths.

Note: For formations with baselines kept below 200km, the relative positioning accuracy using the double-differenced kinematic technique can be kept under 1 meter for an automotive or space grade GPS receiver.

LEOGPS currently supports only observations from the GPS constellation (L1/L2 frequency), with observation files in RINEX v2.XX format. LEOGPS also uses the precise ephemeris (.EPH) and clock bias and drift files (.CLK) provided by the University of Bern, Center for Orbit Determination in Europe (CODE). As such, the coordinate frame used in the relative positioning is the International Terrestrial Reference Frame (ITRS) which is an Earth-Centered Earth-Fixed (ECEF) frame. Since the ephemeris files and RINEX v2 observations default to GPS Time, it is very important to also note that the time scale used in LEOGPS output files is GPS Time (as opposed to UTC).

This project also gives sincere appreciation and credit to the University of Bern, for their provision of the CODE FTP.

Documentation tree for LEOGPS is listed below.

CHAPTER

ONE

INSTALLATION

First, find the LEOGPS GitHub repository in this GitHub link, and download it. Alternatively, if you have Git installed, you can open Command Prompt or your Git Bash, enter the directory of your choice, and type:

```
git clone https://github.com/sammmlow/LEOGPS.git
```

Second, you should do a pip install in your terminal of Martin Valgur's Pythonic translation of Hatanaka (de)compression in Python:

```
pip install hatanaka
```

The Hatanaka library in Python was kindly contributed by Martin Valgur in v1.1, and replaces the older "RNX2CRX" and "GZIP" Windows-only executables, making the (de)compression possible across all operating systems.

That's it! No further setup is needed, unless you need any of the other package dependencies.

Note: Package dependencies include: copy, datetime, decimal, hatanaka, math, matplotlib, numpy, os, pathlib, PIL, tkinter, unlzw3, urllib, warnings

---

Next, we will run and explain the setup behind the default native scenario packaged in LEOGPS - a formation flying scenario at ~200km in-track baseline of the GRACE mission.

CHAPTER

TWO

FIRST STEPS

First, launch LEOGPS by running **'leogps.py'** in a Python IDE or your terminal, and you will see a user-interface (UI) with input parameters as shown below.

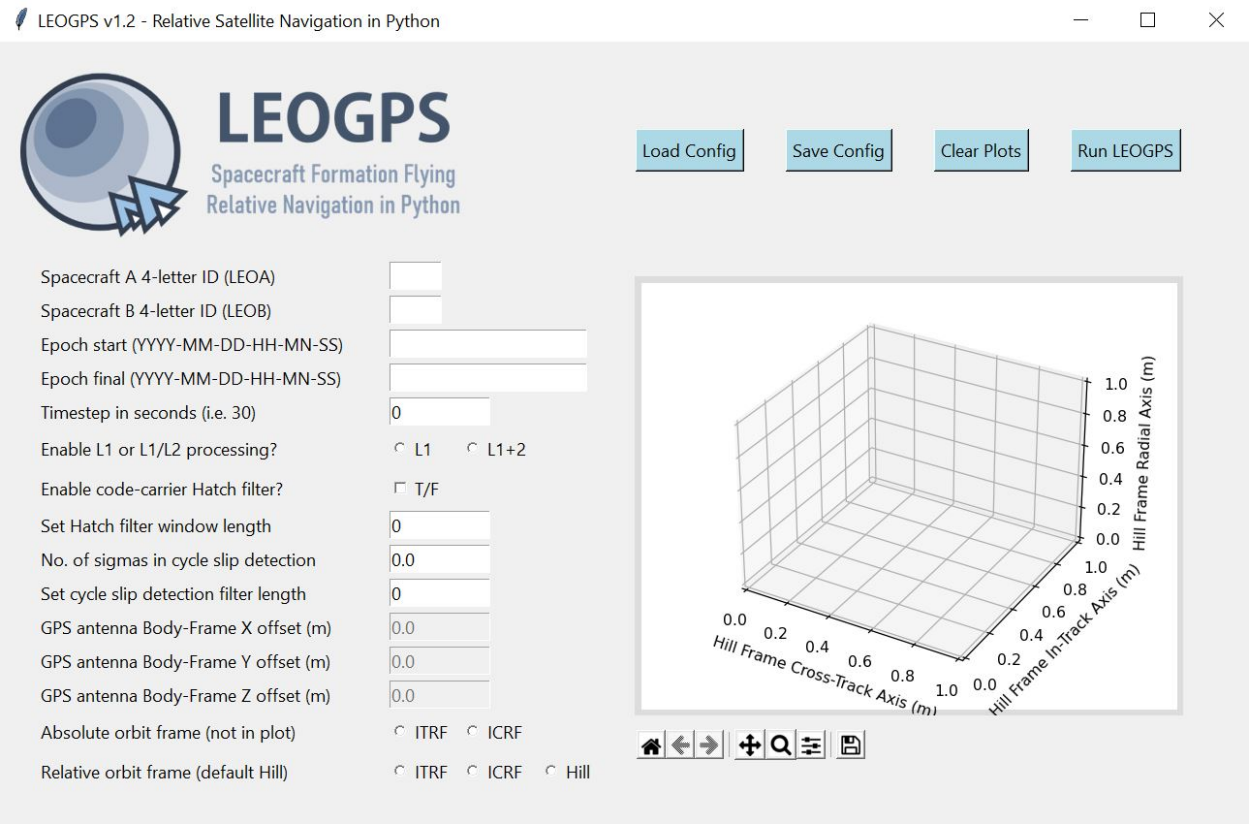


Fig. 1: Graphical User Interface (GUI) of LEOGPS

On the left face of the GUI, you would find text entries where you would input your scenario and spacecraft parameters. On the right face of the GUI, you would find an embedded plot object (matplotlib) which would display the relative orbit trajectory after resolving for the relative positions (you can select the frame, but the default should always be the local Hill Frame with satellite LEO-A as the chief or origin).

Let us walk through the individual input entries and options on the left side of the GUI. This will be where LEOGPS reads and loads the *config.txt* file into the user interface.

---

01. Spacecraft A 4-letter ID (LEOA)

LEOGPS searches for the Spacecraft A's RINEX file using the spacecraft 4-letter ID. You can create one arbitrarily (letters only), just make sure it matches the first 4-letters of the file. For example, the 4-letter ID of GRACE A was set to GRCA in the file name, followed by the 3-digit GPS day-of-the-year, a 1-digit zero, and a file extension of the 2-digit year plus an 'O' for an observation file, or 'D' for a Hatanaka-compressed observation file (GRCA2080.10D for example).

---

02. Spacecraft B 4-letter ID (LEOB)

LEOGPS searches for the Spacecraft B's RINEX file using the spacecraft 4-letter ID. You can create one arbitrarily (letters only), just make sure it matches the first 4-letters of the file. For example, the 4-letter ID of GRACE B was set to GRCB in the file name, followed by the 3-digit GPS day-of-the-year, a 1-digit zero, and a file extension of the 2-digit year plus an 'O' for an observation file, or 'D' for a Hatanaka-compressed observation file (GRCB2080.10D for example).

---

03. Epoch start (YYYY-MM-DD-HH-MN-SS)

Specify the starting epoch for processing in LEOGPS (not necessarily the epoch where the first RINEX entry is found). Follow the date-time format in the heading above, starting with YYYY. Note that the two RINEX files must have an overlapping time frame in order to compute valid differential GPS results. Note also that the time scale is in GPST and not UTC!

---

04. Epoch final (YYYY-MM-DD-HH-MN-SS)

Specify the ending epoch for processing in LEOGPS (not necessarily the epoch where the last RINEX entry is found). Follow the date-time format in the heading above, starting with YYYY. Note that the two RINEX files must have an overlapping time frame in order to compute valid differential GPS results. Note also that the time scale is in GPST and not UTC!

---

05. Timestep in seconds (i.e. 30)

Time step used in processing of RINEX data (integer seconds only).

---

06. Enable L1 or L1/L2 processing?

LEOGPS can read L1 or L2 signals from GPS observations in the RINEX file. The L1 signal is the oldest legacy GPS signal, comprising two parts: the Coarse/Acquisition Code (C1) and the Precision Code (P1). The P-code is reserved for military use, while the C/A is open to the public. The L1 signal uses the frequency 1575.42 MHz. The L2 uses the frequency 1227.60 MHz, and was implemented after the L1 and must be used along with L1 frequencies. It also has a military code and a civilian use code. Enabling both L1/L2 allows LEOGPS to perform single point positioning computations using the ionosphere-free linear combination, where a differential signal between L1 and L2 is used to cancel out correlated ionospheric delay errors.

---

07. Enable code-carrier Hatch filter?

LEOGPS allows for noisy (but unambiguous) code measurements to be smoothed using the precise (but ambiguous) carrier phase measurements. Such an algorithm combines measurements of both from the RINEX file and the filter that implements the algorithm is known as a Hatch filter. Note that enabling Hatch filtering significantly increases the processing time and memory usage.

---

08. Set Hatch filter window length

LEOGPS implements the Hatch filter as a sliding window filter. Thus, you can set the length of the sliding window here as an integer number of samples. For example, if your time step selected was 10 seconds, and the window length input in this option was set to 30, then your effective sliding window duration is 300 seconds.

---

09. Number of sigmas in cycle slip detection

LEOGPS implements a very basic carrier phase cycle slip detection algorithm by performing an interpolation of combined carrier phase data, and observing if there are any single points of data that exceed "X" sigmas of the interpolated carrier phase. "X" is the input specified in this option here. For L1-only processing, LEOGPS uses the Melbourne-Wubbena linear combination as the carrier phase combination. For L1/L2 processing, LEOGPS uses the geometry-free linear combination as the carrier phase combination. For carrier phase observations that exceed the "X" number of sigmas, LEOGPS will mark that point with a cycle slip flag print a warning message to the user in the terminal. However, as of Version 1.3, LEOGPS does not attempt to repair or reject that particular observation.

---

10. Set cycle slip detection filter length

LEOGPS performs interpolation across a sliding window of samples in the cycle slip detection process. The user specifies how many samples N are used for carrier phase interpolation and screening. The larger the value N, the longer the processing but the lesser the likelihood of flagging a false cycle slip.

---

11. Set the GPS antenna X offset (DISABLED)

As the GPS antenna is unlikely to be positioned exactly in the center of mass of the spacecraft, the navigation solutions likely center about the GPS antenna phase center rather than the body center of mass. This option allows the user to specify the body-frame offset. **Note: this option is disabled for now as LEOGPS has not been configured to read spacecraft attitude files, needed for orbit-to-body-frame transformations.**

---

12. Set the GPS antenna Y offset (DISABLED)

As the GPS antenna is unlikely to be positioned exactly in the center of mass of the spacecraft, the navigation solutions likely center about the GPS antenna phase center rather than the body center of mass. This option allows the user to specify the body-frame offset. **Note: this option is disabled for now as LEOGPS has not been configured to read spacecraft attitude files, needed for orbit-to-body-frame transformations.**

---

13. Set the GPS antenna Z offset (DISABLED)

As the GPS antenna is unlikely to be positioned exactly in the center of mass of the spacecraft, the navigation solutions likely center about the GPS antenna phase center rather than the body center of mass. This option allows the user to specify the body-frame offset. **Note: this option is disabled for now as LEOGPS has not been configured to read spacecraft attitude files, needed for orbit-to-body-frame transformations.**

---

14. Orbit frame (not shown in the plot)

LEOGPS will output a text file reporting the single point positions and velocities of the individual spacecraft, as well as the relative baseline vectors. This option allows the user to toggle the output reference frame for the single point positions and velocities. The ITRF-ICRF (ECEF-ECI) conversion uses the IAU1976 Theory of Precession and IAU1980 Theory of Nutation in the celestial-to-terrestrial (and vice versa) conversion. Note that the plotter displays only the relative baselines and not the single point positions.

---

15. Relative orbit frame (default Hill)

LEOGPS will plot in the GUI (the big plot box on the right) as well as output in a text file the relative baseline vectors. In practice, the Euler-Hill frame is typically used (and is the default selection) but the user can also toggle the relative baseline data and plots to be displayed in ITRF or ICRF too.

We may now proceed to run the default LEOGPS scenario that comes with the build, for the GRACE formation flying satellite mission. On the UI, click **Load Config**. This loads the configuration inputs from **config.txt** into the blank entries explained ebove. If you did not make any changes to **config.txt**, the inputs should correspond to the GRACE formation flying scenario on 27-07-2010, as the default example.

Whenever saving (**Save Config**) or running LEOGPS (**Run LEOGPS**), all inputs are saved in the **../LEOGPS/config/config.txt/** file.

Now, we are ready to run LEOGPS. Once we do, several things will happen (next page).

CHAPTER

THREE

FIRST SCENARIO RUN

Now, your LEOGPS scenario should have begun running.

First, let us discuss about the RINEX observation files. Notice that the RINEX observation files have file names that match the 4-letter ID and date-times in your LEOGPS scenario. For example, the GRACE satellites' decompressed observation file names are:

```
GRCA2080.10O
GRCB2080.10O
```

Notice also that in the **../LEOGPS/input/** directory, RINEX files ending with **D** in the file extension will automatically be Hatanaka-decompressed (file extension changes from 'D' to 'O'). LEOGPS accepts Hatanaka compressed and uncompressed RINEX files. The native LEOGPS v1.3 comes with four Hatanaka-compressed RINEX observation files for the GRACE A and B formation (a joint formation flying project by NASA and DLR), and the Lumelite A and B formation (by NUS STAR).

To use your own custom RINEX observation files for your own spacecraft, you should provide the RINEX files following the same file naming convention in the **/../LEOGPS/input/** folder, with RINEX version 2.XX only, one for each LEO satellite. The file naming convention for RINEX observation files is an 8-character string comprising the 4-letter ID of the spacecraft (you can create one arbitrarily), the 3-digit GPS day-of-the-year, followed by a single zero digit. The file extension is the last two digits of the year, followed by the letter 'O'. If the RINEX observation is Hatanaka-compressed, then the last letter of the file extension is a 'D'.

Note: For help in understanding the contents of RINEX formatted files better, LEOGPS comes packaged with the RINEX v2.10 documentation in the **reference** directory.

Second, for supporting files, notice also that LEOGPS will automatically download daily precise (final) ephemeris and clock files from the University of Bern's CODE FTP, and unzip them automatically. Ephemeris files downloaded will span one day before and one day after the scenario date(s) so that the GPS satellite orbit interpolation can be done beyond the edges of the scenario time-line to prevent edge effects (i.e. ringing, poor polynomial fits etc).

As long as the user has an active internet connection, you do not need to provide your own ephemeris and clock files, as LEOGPS will automatically source it from the University of Bern, CODE. However, in the event where the FTP is down, but the user has their own precise ephemeris and clock files, you can transfer these files into the **/../input/** folder and rename these files to follow the CODE naming convention so that LEOGPS detects these files and can use them offline without access to the FTP.

The naming convention for precise clock and ephemeris files are also 8-character strings, starting with a 'COD', followed by the 4-digit GPS week number, followed by the 1-digit GPS day-of-week number. The file extension for precise ephemeris files is a '.EPH' string, and for clock files it is a '.CLK' string.

For example, an ephemeris (.EPH) and clock file (.CLK) from CODE, on the second day of the Week (Tuesday), and on GPS Week Number 1594, would be named:

```
COD15942.EPH
COD15942.CLK
```

Note: For a reference on geodetic calendar formats, refer to the this calendar.

Third, the interpolated data of GPS ephemeris and clocks are saved and plotted in the **../LEOGPS/output/** folders. There would be a text file report of interpolated GPS ephemeris, as well as graphical plots of GPS satellites of PRN IDs 1 to 32 (or fewer, due to spacecraft outages, manoeuvres, servicing etc).

Fourth and finally, the desired main output of LEOGPS - the single point positioning (SPP) and carrier phase differential GPS (CDGPS) solutions, are saved in **'LEOGPS_Results.txt'**. You can open this text file using Excel or any string formatting language you would like to use. Note that for the individual spacecraft ephemeris, the outputs are in the coordinate reference frame chosen in the GUI under the input label **Orbit frame**; for the relative baseline vector (relative position vector) the outputs are in the coordinate reference frame chosen in the GUI under the input label **Relative orbit frame**. It is recommended (and of more use) to display the relative orbit in the Euler-Hill reference frame, which takes Spacecraft A as the chief or reference spacecraft.

In the native LEOGPS build, the ground truths for the GRACE formation are also provided in the **reference** directory. These truths are the precise orbit determination solutions, with relative position magnitudes validated by the GRACE formation's actual K-Band ranging radar data.

This concludes the explanation of the basic setup of LEOGPS. For more information on relative baseline determination, refer to the tutorials on differential GPS in the next page(s).

CHAPTER

FOUR

CARRIER SINGLE DIFFERENCING

This segment of the LEOGPS documentation gets into the technicals of baseline estimation between formation flying satellites (or even between static stations in general), and it assumes that the user has sufficient knowledge on single-point positioning concepts. The algorithm described in this section is based on carrier phase differencing, and is largely implemented in the code file **ambest.py** (ambiguity estimation). For a tutorial on the more basic aspects of GNSS, the user is invited to peruse ESA's NaviPedia.

Carrier phase differential GPS, or CDGPS in summary, is the technique of estimating the relative position of a receiver with respect to another, over a short baseline. The carrier signal's phase is typically exploited for precise point positioning due to its ranging precision, rather than using the unambiguous code. In the case of GPS signals, the ranging accuracy of the C/A code signal is typically on the order of $\pm$3m for a typical GNSS receiver (on the modernized GPS block without selective availability); whereas the ranging accuracy of the carrier phase of the signal is on the order of millimeters due to the relatively short wavelength (19.05cm for L1 for example) and the accuracy achievable in most receiver phase-locked loops today.

The carrier phase range can be modelled as the observed phase, plus an integer number of wavelengths, and plus a series of systematic and random ranging errors. The goal of carrier phase differential GPS techniques is to remove these systematic ranging errors through differential measurements which cancel common and correlated sources of ranging errors observed by both receivers over a "short" baseline.

The integer number of wavelengths however, is unknown and must be estimated. This is known as the "integer ambiguity resolution" problem in GNSS literature, and it can be estimated as a float, with subsequent integer fixing techniques (i.e., wide-lane, narrow-lane, or if the error covariances are known, then integer fixing by Peter Teunissen's LAMBDA).

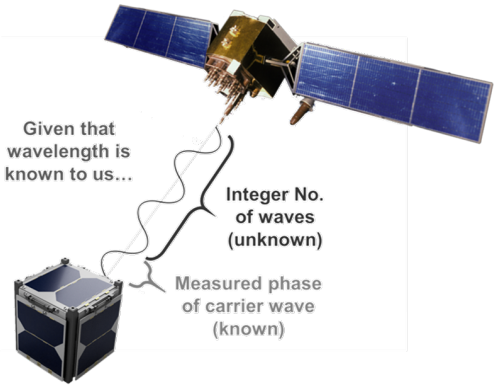


Fig. 1: Figure C1.1: Carrier phase ranging illustration highlighting the ambiguity N.

In any case, the receiver end only measures the instantaneous carrier phase modulo 360 degrees. It is unknown to the receiver how many integer cycles of the carrier signal has passed through the signal path taken from the emission from the GPS satellite to the receiver. Nevertheless, the RINEX file carrier phase observable would normally give an integer estimate nonetheless (usually using a pseudo-range model).

In order to very accurately characterize the range, the number of integer cycles, hereby known as N, must be made known. The integer ambiguity of the carrier phase is red-boxed below.

$$\theta = r + I + \delta T_{LEO} + \delta T^{(K)} + G + N + \{\varepsilon\}$$

Measured phase of carrier wave; True range; Ionosphere delay; LEO clock bias; GPS clock error; Relativistic effects (spacetime dilation and clock advance); Integer number of wave cycles (unknown)

Fig. 2: Equation C1.1: Carrier phase ranging model

On top of the integer ambiguity N, there are various other error sources that are modelled in the GPS signal range equation (Equation C1.1), of which are the following in descending order of importance and accuracy loss: the GPS satellite clock bias estimation errors, the LEO satellite clock bias, ionospheric path delays, and other relativistic effects such as clock advance effects and Shapiro path delays.

The key to mitigating errors is the realization that error components shown in Equation C1.1, are highly correlated among GPS receivers in proximity. Subtracting measurements (differencing) between receivers will therefore "cancel" out systematic errors between the LEO satellites.

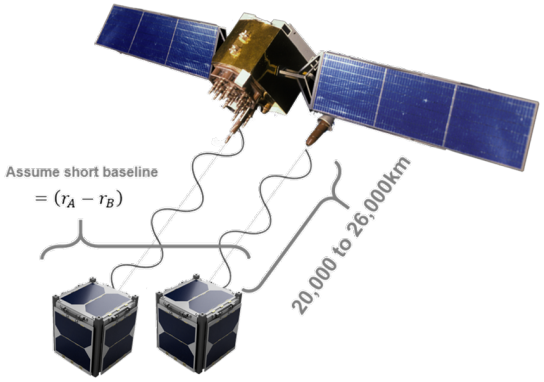


Fig. 3: Figure C1.2: Carrier phase single differencing between two LEOs

In the case of the single differencing scenarios, the absolute carrier phase measurements taken by two LEOs are subtracted from each other. Using the carrier phase ranging model from Equation C1.1, the error sources that are common to both are red-boxed below.

$$\theta_A = r_A + I + \delta T_A + \delta T^{(K)} + G + N_A + \{\varepsilon\}$$
$$\theta_B = r_B + I + \delta T_B + \delta T^{(K)} + G + N_B + \{\varepsilon\}$$

Fig. 4: Equation C1.2: Common errors for a single differencing scenario

Since the baseline of the LEO satellites are expected not to deviate beyond 200km, it is considered "short", and thus likely that the ionospheric errors are correlated. Also, this means that the total number of and pseudo-range IDs of GPS satellites in view will not be any different between the two LEOs. Thus, in the single differencing case, between two LEOs and some k-th GPS satellite, differencing the carrier phase measurements between LEO A and B will remove systematic biases in ionospheric path delays, the k-th GPS satellite clock offsets, and common relativistic effects.

As a result, the single difference equation comprises only of the relative true ranges, the relative receiver clock bias estimation errors, and the relative carrier phase cycle integer ambiguities. For the epsilon error term, assuming both errors are Gaussian, this results also in a square-root-2 amplification in white Gaussian noise.

$$\theta_A - \theta_B = (r_A - r_B) + (\delta T_A - \delta T_B) + (N_A - N_B) + \{\varepsilon_{AB}\}$$
$$= \theta_{AB} \rightarrow \text{Single differencing}$$

Fig. 5: Equation C1.3: Carrier phase single differencing equation

If the receiver clock bias errors are small, then the baseline can actually already be derived from the single difference equation with little consequence to the accuracy of the baseline AB.

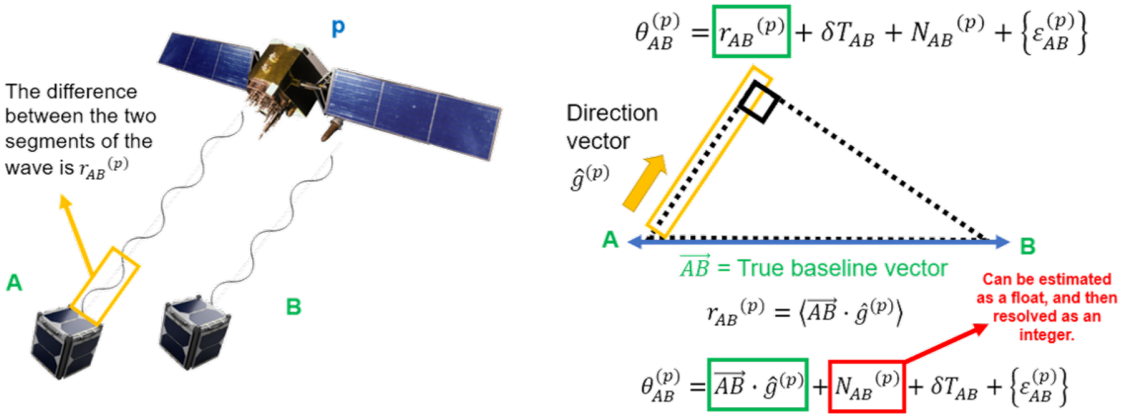


Fig. 6: Figure C1.3: Extraction of baseline vector AB from the single differencing equation.

In extracting the baseline vector AB in Figure C1.3, the experimenter makes a few assumptions. First, assume that the GPS satellite is far away enough such that the signal paths taken from the emitter to the receivers A and B are almost parallel. Second, assume also that the receiver will be able to estimate the direction vector from itself to the GPS satellite, which is given by g-hat in the Figure C1.3, without introducing new errors. This is possible only if coarse positioning estimates of itself is first achieved using the C/A or P code, and that the GPS navigation observables are correctly parsed. In most cases, since the distance of the GPS satellite to the LEO is on the order of about ~20,000km, any coarse positioning errors on the meter-level scale would not significantly affect the accuracy of the estimation of the g-hat vector. Next, a very rough estimate of the integer ambiguity can be estimated as a float using the pseudo-range values from code measurements, and the known carrier wavelength .

$$N_{float} = \left[\theta - \frac{\rho}{\lambda}\right]$$

One can now solve for the AB vector as seen in Figure C1.3, notwithstanding the fact that the receiver clock bias estimation errors were not cancelled out and will thus show up in some form in the accuracy of the positions. The single differenced baseline solution from LEOGPS for a 100km baseline separation is shown below:

Observably, the single differenced solution still faces an accuracy > 1m. Embedded in the error plots are likely the unaccounted relative receiver clock bias estimation errors, minor error sources such as antenna phase centre variations, et cetera, which were not differenced away, as well as other white noise error sources.

Note: Every step of differencing amplifies the random error sources by root-2, assuming the noises are well-modelled as a Gaussian.

We can go a step further to difference a pair of single-difference observations in order to eliminate the remnant relative

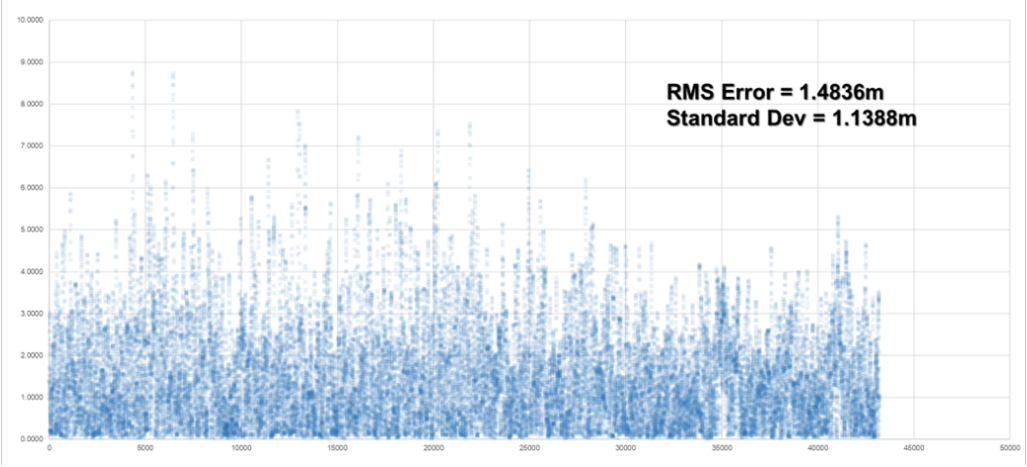


Fig. 7: Figure C1.4: Relative position error norm of a single differenced 100km LEO baseline.

receiver clock bias estimation errors. This technique is known as double-differencing, and it will be explained in the next section.

CHAPTER

FIVE

CARRIER DOUBLE DIFFERENCING

We can actually go one step further beyond single differencing, and that is to difference across two reference GPS satellites. This step is called double differencing, and it is the backbone algorithm used in LEOGPS. This removes the relative receiver clock bias estimation errors, and this section will detail the algorithm and the results below.

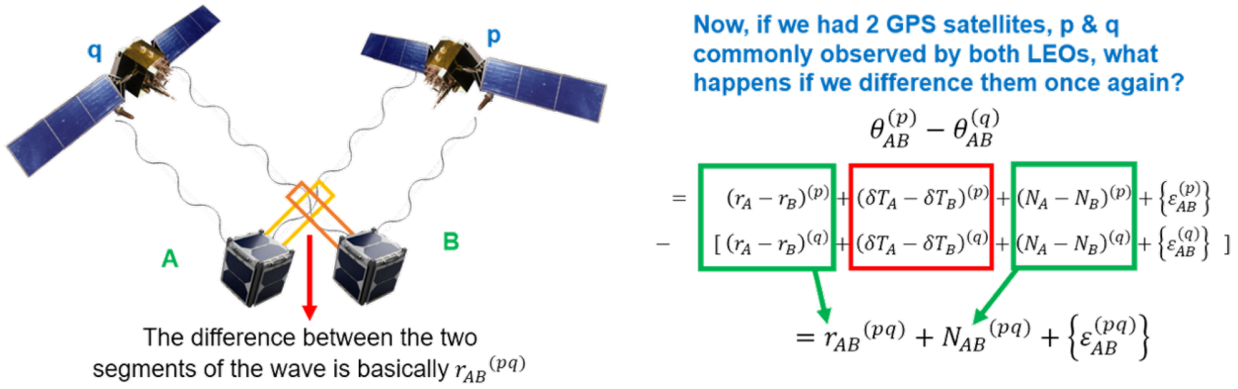


Fig. 1: Figure C2.1: Carrier phase double differencing equation setup.

The key idea behind double differencing, is to create a differential between two single difference equations. For example, between two LEO units A and B, if some GPS satellite P was taken as the reference emitter satellite in the single difference equation, then a second single difference equation would be taken with respect to a different GPS satellite Q as the reference emitter. In both single difference equations, it makes no difference to the relative receiver clock bias estimation which reference GPS satellite is used, as these are the errors specific only to the receiver-side in the relative sense.

Thus, taking two single difference measurements, one can observe that the relative receiver clock bias estimation error has now become a common error source in the double differenced equation, which can be cancelled out (both relative receiver clock bias terms in the red box in Figure C2.1 are actually the same and thus they cancel).

In a similar fashion to the single differencing algorithm, the double differencing algorithm requires the estimation of the unit direction vector from the LEO A to reference GPS satellite P, and from the LEO B to the reference GPS satellite Q. Once again, with the double-differenced relative range, and on the assumption that the signal path pairs from P to A and P to B are parallel, and the path pairs from Q to A and Q to B are also parallel, one can de-construct the baseline vector AB since each single difference observation forms the base of a right-angled triangle.

Since two reference satellites are now used in the creation of a double differenced equation, for N number of common GPS satellites in view, one can expect N-1 number of double differenced equations. Solving the entire system of equations leads to the baseline vector solution, with the error norm plotted below (error taken against a ground truth for two LEO units at a 100km baseline separation). It is observed that the final elimination of receiver clock biases creates an almost ten fold increase in the navigation accuracy.

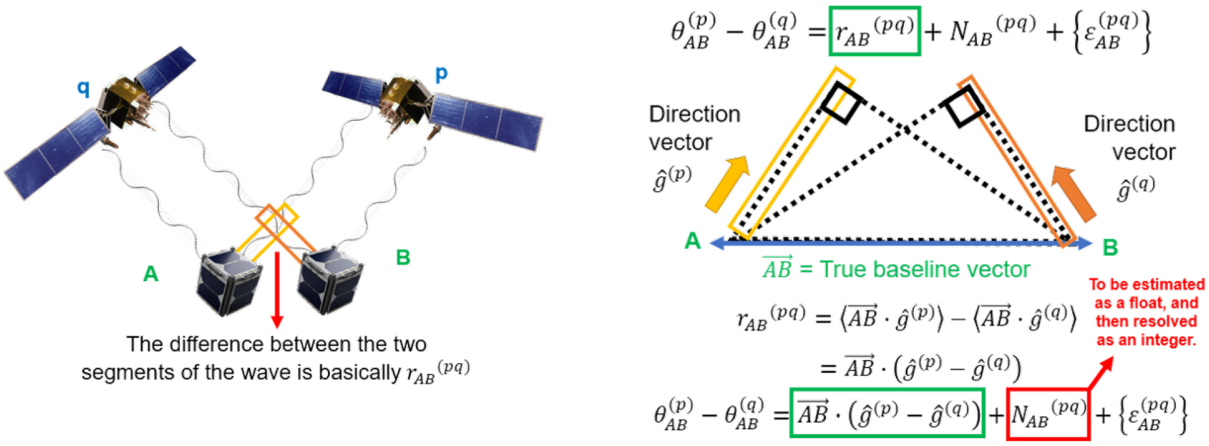


Fig. 2: Figure C2.2: Extraction of the true baseline vector via double differencing.

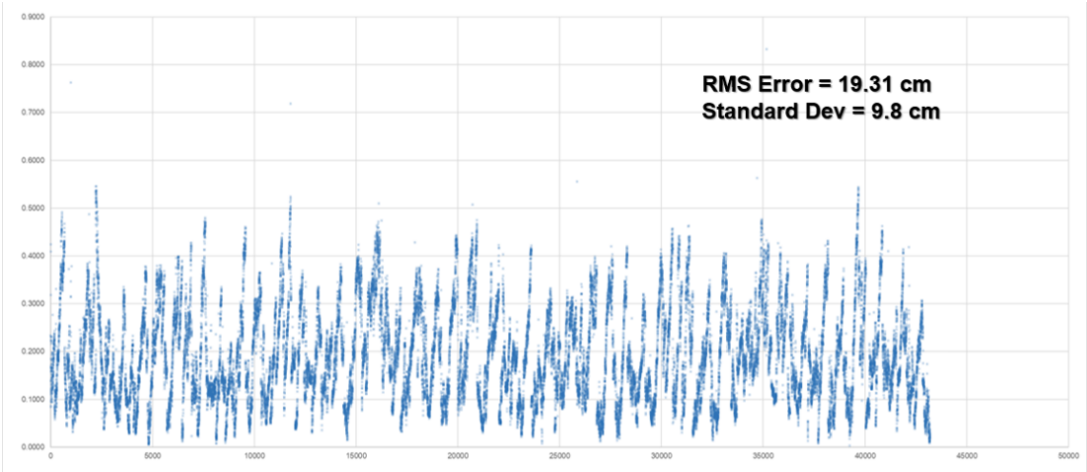


Fig. 3: Figure C2.3: Relative position error norm of a double differenced 100km LEO baseline.

In its essence, the single differencing paradigm changed the structure of the navigation problem, eliminating most nuisance errors, while the double differencing paradigm went further and essentially eliminated the receiver clock bias at the expense of one equation in the entire system of equations, while also redefining the ambiguity term into a double-differential ambiguity.

We can actually go one step further and perform triple-differencing, that is, performing a differencing measurement between two double-differences. This eliminates the ambiguity entirely, but it amplifies the random errors by another factor of root-2, and it further reduces the degrees of freedom by one, from N-1 to N-2, N being the number of GPS satellites.

At this point though, it becomes arguable that estimating the ambiguity with the C/A range (float ambiguity resolution) on a double-differencing measurement would give a better accuracy than performing triple differencing. This is the reason why LEOGPS does not use triple-differencing.

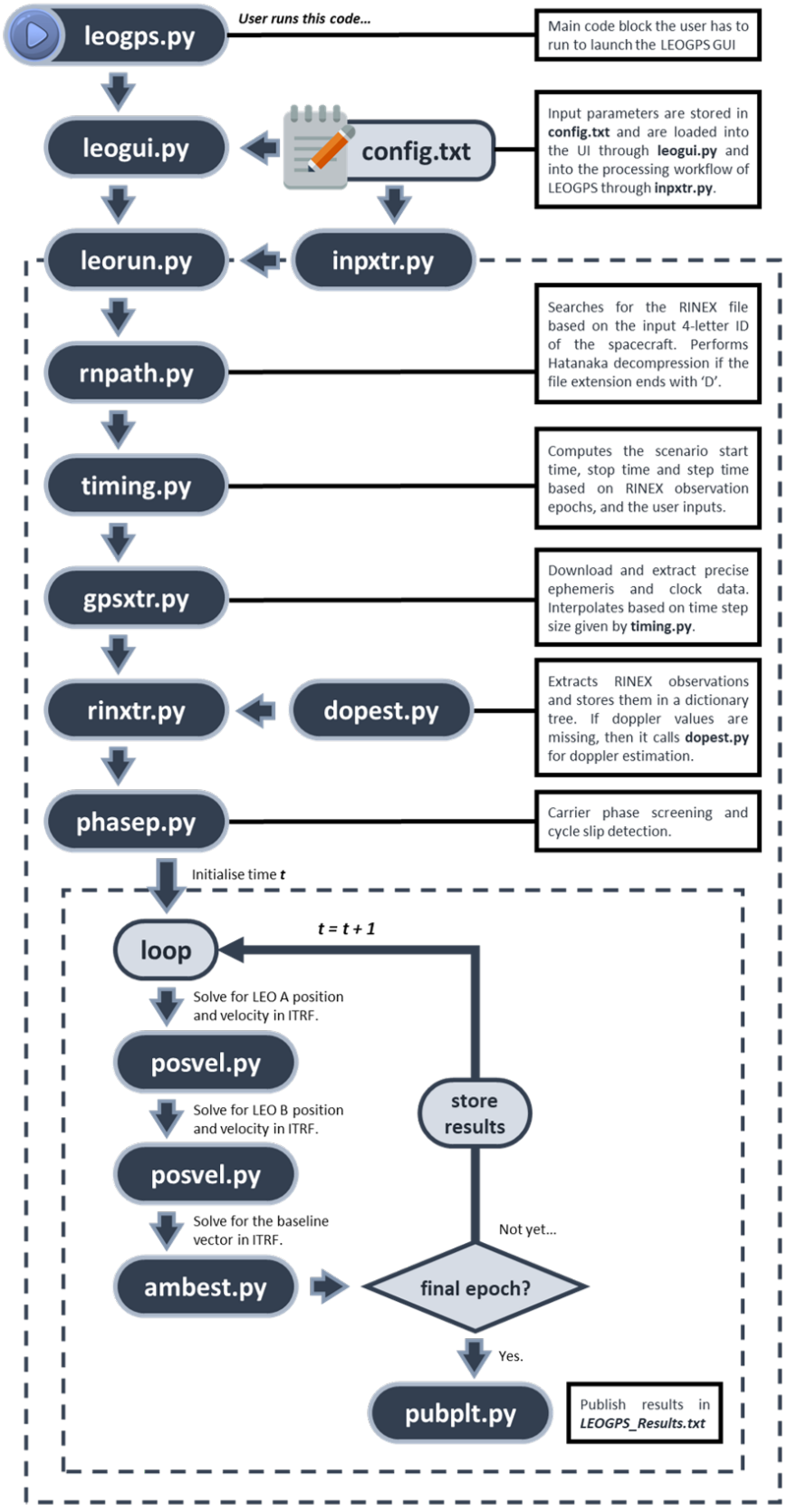
CHAPTER

SIX

PROCESSING FLOW

A summary of the processing flow in LEOGPS is given by the flow chart below. This is the main processing tree that runs internally behind the GUI, in the **run()** function of the module **leorun.py**.

If the user wishes to write their own processing workflow (by overwriting **leorun.py**), it is advisable to refer to the API reference on the next page.


leogps.py
User runs this code...
Main code block the user has to run to launch the LEOGPS GUI
leogui.py
config.txt
Input parameters are stored in config.txt and are loaded into the UI through leogui.py and into the processing workflow of LEOGPS through inpxtr.py.
leorun.py
inpxtr.py
rnpath.py
Searches for the RINEX file based on the input 4-letter ID of the spacecraft. Performs Hatanaka decompression if the file extension ends with 'D'.
timing.py
Computes the scenario start time, stop time and step time based on RINEX observation epochs, and the user inputs.
gpsxtr.py
Download and extract precise ephemeris and clock data. Interpolates based on time step size given by timing.py.
rinxtr.py
dopest.py
Extracts RINEX observations and stores them in a dictionary tree. If doppler values are missing, then it calls dopest.py for doppler estimation.
phasep.py
Carrier phase screening and cycle slip detection.
Initialise time t
loop
t = t + 1
Solve for LEO A position and velocity in ITRF.
posvel.py
Solve for LEO B position and velocity in ITRF.
posvel.py
Solve for the baseline vector in ITRF.
ambest.py
store results
Not yet...
final epoch?
Yes.
pubplt.py
Publish results in LEOGPS_Results.txt

CHAPTER

SEVEN

API REFERENCE

The order of functions in this API reference goes according to the chronological order of which they are called in the native LEOGPS processing work flow (see previous page). Functions that are currently not in use in the current native work flow are listed at the end of this page.

7.1 leogui.py

The first item to be called when running **leogps.py** is the root GUI object (tkinter.Tk() object). This object interfaces directly with **config.txt** to save or load your input parameters.

7.2 leorun.py

The **leorun.run()** function is called once the user clicks on **Run LEOGPS**.

7.3 inpxtr.py

The first thing that is run in the native work flow is to extract all the inputs from **config.txt**. This is done through the **inpxtr** module which comprises two functions. First, **inpxtr.inptim()**, extracts the dates and times from the config file and outputs them in GPS time format. Second, **inpxtr.inpxtr()** extracts all the other processing parameters.

`inpxtr.inptim(t)`

Extract time parameters from **config.txt**, and translates them into five timing-related parameters to be used for processing later on. Called internally only in *inpxtr.inpxtr()*.

Parameters `t` (*datetime.datetime*) – An epoch datetime object.

Returns

- **yyyy** (*int*) – 4-digit Gregorian year
- **yy** (*str*) – 2-digit Gregorian year (for RINEX file name)
- **doy** (*str*) – 3-digit day-of-year (for RINEX file name)
- **wkday** (*int*) – 1-digit day-of-week (Sunday = 0, Saturday = 6)

- **wwww** (*int*) – GPS week number (generally 4-digits)

inpxtr.**inpxtr**()

Extract all parameters from **config.txt**. No input arguments.

Returns **inputdict** – A dictionary of key-value pairs comprising of key-values found in the **config.txt** file. For example, the time step of the scenario would be the key-value pair *{ 'timestep' : 30 }*

Return type dict

7.4 rnpath.py

The second step is to search for the RINEX file paths based on the 4-letter IDs of both spacecraft, and to perform Hatanaka decompression if necessary.

7.5 timing.py

The third step involves deconflicting all the timing parameters. The routine **timing.py** will take in three sets of timings: the user-specified start-stop times in the GUI or in **config.txt**, and the start-stop times from both of the RINEX observation files. This module will then output the intersection of all three timelines. This module will also check if the time steps of the RINEX files and time step specified by the user are compatible.

timing.**get_startstop**(*rnxlines*)

Gets the first and last epoch after reading one RINEX file.

Parameters **rnxlines** (*list*) – List of strings as lines of the RINEX file

Returns

- **rnxstart** (*datetime.datetime*) – Observed initial time stamp in RINEX file
- **rnxstop** (*datetime.datetime*) – Observed final time stamp in RINEX file
- **rnxstep** (*datetime.timedelta*) – Observed time step in RINEX file

timing.**tcheck**(*rnx1file*, *rnx2file*, *inps*)

Reads both RINEX files and output the desired start and stop times. Sets the start, stop, and time steps for the entire LEOGPS scenario.

Parameters

- **rnx1file** (*pathlib.Path*) – Path of RINEX file of LEO-A
- **rnx2file** (*pathlib.Path*) – Path of RINEX file of LEO-B
- **inps** (*dict*) – A dictionary of inputs created by *inpxtr.inpxtr()*

Returns

- **tstart** (*datetime.datetime*) – Scenario start time for processing
- **tstop** (*datetime.datetime*) – Scenario stop time for processing
- **tstep** (*datetime.timedelta*) – Scenario time step used in processing
- **rnx1step** (*datetime.timedelta*) – Observed time step in the RINEX file

7.6 gpsxtr.py

The fourth step is to extract out the interpolated GPS satellite ephemeris and clock data.

The primary output of **gpsxtr.gpsxtr()** is the **gpsdata** dictionary, which is a three-layer nested dictionary comprising all of the Lagrange-interpolated ephemeris and clock data. The first layer keys are the epochs (*datetime.datetime*). The second layer keys are the PRN IDs of GPS satellites. The third layer keys are the XYZ coordinates of the position vectors, XYZ coordinates of the velocity vectors, the clock bias, and clock drift. Third layer values are all floats.

All GPS coordinates are in ITRF by default. No coordinate reference frame transformations were performed between the raw and the interpolated GPS ephemeris.

An example structure of this dictionary is given below:

```
gpsdata = {epoch1 : {1 : {px:0,   py:0,   pz:0,
                          vx:0,   vy:0,   vz:0,
                          clkb:0, clkd:0      },

                     2 : {px:0,   py:0,   pz:0,
                          vx:0,   vy:0,   vz:0,
                          clkb:0, clkd:0      }, ...

                         ... ... ... ... ... ... ...

                    32 : {px:0,   py:0,   pz:0,
                          vx:0,   vy:0,   vz:0,
                          clkb:0, clkd:0      }},

           epoch2 : {1 : {px:0,   py:0,   pz:0,
                          ... ... ... ... ... ...} ...} ...}
```

The secondary output of **gpsxtr.gpsxtr()** is the **goodsats** list, which is a sorted list of GPS satellite PRN IDs whose observables had no outages.

An auxiliary function **gpsxtr.gpsweekday()** exists but is not documented here. It converts a datetime object into two string variables: the GPS day-of-week and the GPS week number. This is a similar function to **inpxtr.inptim()**.

7.7 rinxtr.py

The fifth step involves the extraction of RINEX observations C1/P1, P2, L1, L2, D1, D2, with carrier phase marking, and cycle slip detection performed using the **phasep.py** module.

An additional 'flag' key will be added to the RINEX observables to mark them. These are the possible values belonging to the 'flag' keys.

- **"start" :** start of carrier phase observed, from an N-th GPS satellite, in a sequence.
- **"end" :** last carrier phase observed, from some N-th GPS satellite, in a sequence.
- **"solo" :** single carrier phase observation by some N-th GPS satellite, no sequence.
- **"none" :** carrier phase observation within a sequence, from some N-th GPS satellite.
- **"slip" :** cycle slip flag for carrier phase observed from some N-th GPS satellite.

An additional carrier phase term, L4, will also be added to the RINEX observables. This helps LEOGPS to perform cycle slip detection. If the user opts for single frequency processing, then L4 is the Melbourne-Wubbena linear combination. If the user opts for dual frequency processing, then L4 is the geometry-free linear combination.

If Doppler observables D1/D2 are not found in the RINEX observations, then Doppler values will be estimated through the **dopest.py** module, by estimating a first-order derivative of the L1/L2 phase values numerically using polynomial fitting.

The output RINEX observations are structured as a dictionary of epochs, each epoch with a sub-dictionary of GPS satellites based on IDs (1 to 32), and each ID with a sub dictionary of the various observations (C1/P1, P2, L1, L2, D1, D2). Example output structure:

```
rnxproc = {epoch1 : {1 : {'C1':123, 'L1':123, ...
                          'L4':321, 'flag':'none'} ...} ...
           epoch2 : {2 : {'C1':123, 'L1':123, ...
                          'L4':321, 'flag':'slip'} ...} ...

                   ...

           epochX : {1 : {'C1':123, 'L1':123, ...
                          'L4':321, 'flag':'none'} ...}}
```

If the user opts for code-carrier smoothing Hatch filter (either through the GUI or in the config file), then hatch filtering will be called in **rinxtr.rinxtr()** using the **phasep.py** module.

You may also change the length of the cycle slip filter, and the filter tolerance in terms of the number of standard deviations through the GUI or manually in the config file.

Do ensure that RINEX observation files follow 4-letter ID naming convention followed by the DOY + 0, with the file extension .YYO.

7.8 phasep.py

The **phasep.py** module augments the RINEX data dictionary parsed out by **rinxtr.py** module. It comprises the cycle slip detection and marking function **phsmrk()**, as well as the hatch filtering algorithm **ph1fil()** for L1 observables, and **ph2fil()** for L1 + L2 observables. Within the hatch filtering loop, each code-phase data point at each time step is computed by the **hatch1()** or **hatch2()** functions.

In this module, the carrier phase cycle slip detection algorithm is done by performing an interpolation of combined L4 (see **rinxtr.py** above) carrier phase data, and observing if there are any single points of data that exceed "X" sigmas of the interpolated carrier phase. "X" refers to the user-specified number of standard deviations as a cut-off point for declaring deviant observations as cycle slips. Such deviant observations will trigger the script to mark that epoch's observation and GPS PRN ID with a cycle slip string flag, and print a warning message to the user in the terminal. However, as of Version 1.3, LEOGPS does not attempt to repair or reject that particular observation.

Note: The RINEX data dictionary returned by **phasep.phsmrk(), phasep.ph1fil(), phasep.ph2fil()** all share the same nested dictionary key-value pairs as the output of the **rinxtr.rinxtr()** function.

7.9 dopest.py

If Doppler observables D1/D2 are not found in the RINEX observations, then Doppler values will be estimated through the **dopest.py** module, by estimating a first-order derivative of the L1/L2 phase values numerically using polynomial fitting.

7.10 posvel.py

The sixth step is to perform single point positioning (SPP), using the code pseudorange equations, solved via weighted least squares epoch-wise. Thus, the primary function **posvel.posvel()** is called once for each satellite and for each epoch.

In the code phase ranging equation, GPS satellite clock biases are offset from the output of the **gpsxtr.py** module. Ionospheric delays are handled too. In the L1 case, the GRAPHIC linear combination is used. In the L2 case, the ionosphere-free linear combination is used. Other effects such as the signal time-of-flight, the effects of Earth rotation, relativistic Shapiro effect, relativistic clock delays and clock advances are also offset. Functions to compute the relativistic effects are given in the next section, under **einstn.py**.

Doppler-based estimation of velocities will also be performed if Doppler data is available in the RINEX data dictionary parsed out by **rinxtr.py** module. By default, if Doppler data is missing, the **dopest.py** module would have worked its magic to estimate the Doppler values.

Note: Tropospheric effects are **not** handled in **posvel.py** as LEOGPS was built for spaceborne receivers and not for terrestrial receivers. However, if you wish to include terrestrial receivers, you can include the tropospheric modelled offsets (i.e. Saastamoinen, Hopfield, or Differential Refraction models etc) to the observed range variable *gpsrng_obsv* from lines 258 to 297.

In the main work flow **leorun.run()**, this function will be called once in each epoch for each of the two satellites. Both SPP results will be used in the carrier phase ambiguity estimation.

7.11 einstn.py

The 'Einstein' module, comprises two main functions: one to compute the clock advance and one to compute the Shapiro path delay. In both functions, the output is converted to the equivalent path-length in meters.

Note: The rate of advance of two identical clocks, one in the LEO satellite and the other on the GPS satellite, will differ due to differences in the gravitational potential and to the relative speed between them.

Note: **Shapiro Delay:** Due to the space time curvature produced by the gravitational field, the Euclidean range travelled by the signal, which is computed by **posvel.py** must be corrected by the extra distance travelled. Typically, Shapiro effects corrupt the range model with about ~2cm ranging error.

This module will be called in **posvel.py** during the setup of pseudorange model in the iterative least squares solution of single-point positioning.

7.12 azimel.py

While elevation-dependent or azimuth-dependent weighting is not built into the iterative least squares processing of LEOGPS for single point positioning, this module exists (but is not used) if the user wishes to compute azimuths and elevations from the LEO to GPS satellites anyway.

7.13 ambest.py

This is the seventh processing step, which is the carrier phase integer or float ambiguity resolution step. This module contains functions that support epoch-wise processing for integer ambiguity resolution step.

The chief function in the module is the **'ambest()'** function, which outputs the precise relative baseline vector between the two spacecraft. This processing is done snapshot-wise, and thus has to be called for each epoch of carrier phase observations. All other supporting functions are called within **'ambest()'**. At the user-level, it is advised to modify contents only within 'ambest' unless the user wishes to modify the core float ambiguity resolution algorithm.

An option exists to perform integer fixing (see the *fix* argument above) using Peter Teunissen's LAMBDA method. A Pythonic translation of his original LAMBDA Integer-Least-Squares (ILS) Search-and-Shrink algorithm has been provided in the **ambfix.py** module.

Note: If the user wishes to set their own custom zero difference covariance matrix, the user can input this in the optional *covZD* argument. If the user does not specify the *covZD* argument, then *covZD* by default will revert to an identity matrix scaled by the *sigma* argument above. Thus, running LAMBDA (by setting $fix = True$ when calling **ambest.py**), but not setting a custom *covZD* argument, only has the equivalent effect of integer rounding.

7.14 ambfix.py

This module holds the classical LAMBDA method that was originally authored by Teunissen, Jonge, and Tiberius (1993). The code was later written in MATLAB by Dr Sandra Verhagen and Dr Bofeng Li. It takes in a vector of float ambiguities to the integer least-squares (ILS) problem, and covariance of the float ambiguities. It then runs the LAMBDA's ILS search-&-shrink and spits out the ambiguity integers. The other 5 methods in original LAMBDA MATLAB code are not supported here (feel free to edit the code and implement it youself). The default $ncands = 2$, as per original code. All supporting functions from the original MATLAB code (*decorrel*, *ldldecom*, *ssearch*) have been nested within the main function as sub functions.

Note: LAMBDA always first applies a decorrelation before the integer estimation. For ILS this is required to guarantee an efficient search. For rounding and bootstrapping it is required in order to get higher success rates (although rounding and bootstrapping is not included in LEOGPS).

7.15 frames.py

This is the eighth step in the LEOGPS native processing work flow. This step performs the conversion of the coordinate reference frames between ITRF and ICRF, via the IAU1976 Theory of Precession and IAU1980 Theory of Nutation. For the visualisation of the formation geometry, it is recommended that the user select the Hill frame as the relative orbit coordinate frame. By default, the reference frame in the downloaded ephemeris files in AIUB CODE's FTP is the ITRF.

7.16 pubplt.py

In the final stage, after all processing is done, the **pubplt.py** module publishes the information into output files in the *outputs* folder, found in the LEOGPS root directory.

Specifically, there are three functions in this module: a function to save as a plot graph the interpolated GPS ephemeris and clock biases; a function to save as a text report the interpolated GPS ephemeris and clock biases; and a function to save the final ephemeris and precise baselines estimated of both LEO-A and LEO-B.

7.17 consts.py

The following is a list of common constants used throughout LEOGPS, extracted from the University of Bern, Center for Orbit Determination in Europe (CODE):

```
C      = 299792458.0      # VELOCITY OF LIGHT                M/SEC
FREQ1  = 1575420000.0     # L1-CARRIER FREQUENCY   GPS       1/SEC
FREQ2  = 1227600000.0     # L2-CARRIER FREQUENCY   GPS       1/SEC
FREQ5  = 1176450000.0     # L5-CARRIER FREQUENCY   GPS       1/SEC
FREQP  = 10230000.0       # P-CODE     FREQUENCY   GPS       1/SEC
FREQG1 = 1602000000.0     # L1-CARRIER FREQUENCY   GLONASS   1/SEC
FREQG2 = 1246000000.0     # L2-CARRIER FREQUENCY   GLONASS   1/SEC
DFRQG1 = 562500.0         # L1-CARRIER FREQ. DIFF. GLONASS   1/SEC
DFRQG2 = 437500.0         # L2-CARRIER FREQ. DIFF. GLONASS   1/SEC
FREQGP = 5110000.0        # P-CODE     FREQUENCY   GLONASS   1/SEC
FRQE1  = 1575420000.0     # L1-CARRIER FREQUENCY   GALILEO   1/SEC
FRQE5  = 1191795000.0     # L5-CARRIER FREQUENCY   GALILEO   1/SEC
FRQE5a = 1176450000.0     # L5a-CARRIER FREQUENCY  GALILEO   1/SEC
FRQE5b = 1207140000.0     # L5b-CARRIER FREQUENCY  GALILEO   1/SEC
FRQE6  = 1278750000.0     # L6-CARRIER FREQUENCY   GALILEO   1/SEC
FRQS1  = 1575420000.0     # L1-CARRIER FREQUENCY   SBAS      1/SEC
FRQS5  = 1176450000.0     # L5-CARRIER FREQUENCY   SBAS      1/SEC
FRQC1  = 1589740000.0     # L1-CARRIER FREQUENCY   COMPASS   1/SEC
FRQC2  = 1561098000.0     # L2-CARRIER FREQUENCY   COMPASS   1/SEC
FRQC5b = 1207140000.0     # L5b-CARRIER FREQUENCY  COMPASS   1/SEC
FRQC6  = 1268520000.0     # L6-CARRIER FREQUENCY   COMPASS   1/SEC
FRQJ1  = 1575420000.0     # L1-CARRIER FREQUENCY   QZSS      1/SEC
FRQJ2  = 1227600000.0     # L2-CARRIER FREQUENCY   QZSS      1/SEC
FRQJ5  = 1176450000.0     # L5-CARRIER FREQUENCY   QZSS      1/SEC
FRQJ6  = 1278750000.0     # L6-CARRIER FREQUENCY   QZSS      1/SEC
```

(continues on next page)

(continued from previous page)

```
GM      = 398.6004415e12   # GRAVITY CONSTANT*EARTH MASS      M**3/SEC**2
GMS     = 1.3271250e20     # GRAVITY CONSTANT*SOLAR MASS      M**3/SEC**2
GMM     = 4.9027890e12     # GRAVITY CONSTANT*LUNAR MASS      M**3/SEC**2
AU      = 149597870691     # ASTRONOMICAL UNIT                M
AE      = 6378137.0        # EQUATORIAL RADIUS OF EARTH       M
CONRE   = 6371000.0        # MEAN RADIUS OF THE EARTH         M
J2      = 1.0826359e-3     # DYNAMICAL FORM-FACTOR IERS(2003) 1
FACTEC  = 40.3e16          # IONOSPHERIC FACTOR               M/SEC**2/TECU
P0      = -0.94e-7         # NOMINAL RAD.PR. ACCELERAT.       M/SEC**2
OMEGA   = 7292115.1467e-11 # ANGULAR VELOCITY OF EARTH        RAD/USEC
EPHUTC  = 55.0             # EPH. TIME (ET) MINUS UTC         SEC
WGTPHA  = 1.0              # WEIGHT FOR PHASE OBSERVATIONS    1
WGTCOD  = 1.0e-4           # WEIGHT FOR CODE OBSERVATIONS     1
HREF    = 0.0              # REFERENCE HEIGHT FOR METEO MODEL M
PREF    = 1013.25          # PRESSURE AT HREF                 MBAR
TREF    = 18.0             # TEMPERATURE AT HREF              DEG. CELSIUS
HUMREF  = 50.0             # HUMIDITY AT HREF                 %
ERR     = 7.2921150e-5     # EARTH INERTIAL ROTATION RATE     RAD/SEC
```

This API reference was automatically generated using Sphinx' Autodoc feature, using the NumPy docstring format, and last updated on 11th September 2021.

For bugs, raise the issues in the GitHub repository. For collaborations, reach out to me (sammmlow@gmail.com). The project is licensed under the MIT license.

Written By: Samuel Y. W. Low

PYTHON MODULE INDEX

i

inpxtr, 21

t

timing, 22

INDEX

G

get_startstop() (*in module timing*), 22

I

inptim() (*in module inpxtr*), 21
inpxtr
    module, 21
inpxtr() (*in module inpxtr*), 22

M

module
    inpxtr, 21
    timing, 22

T

tcheck() (*in module timing*), 22
timing
    module, 22