Qgps

Tom Laclavère

December 2024

1 Introduction

This document aims to explain the Qgps code, using GPS data to compute the position and orientation of the QUBIC's calibration source. It will be divided into two main sections: one to explain how the code is written, how the computations are done, and the code's capability, and the second will explain how to use it in a practical case. The code of this document is to improve the clarity of the code and then to allow quick modification if needed and better accessibility for everybody.
In all this code, we assume that the two GPS antennas and the calsource are a rigid ensemble. Also, we had that because of the system, we were blind to rotations around the axis formed by the vector between the two antennas. Because of that, we recommend aligning this vector with the line of sight of the calibration source (which has a symmetric beam) to not be penalized by that. Also, to be able to compute the position of the calsource, it is better to place one of the two antennas. It is due to the same problem: because we are blind to rotation around this axis, we cannot know if the calsource rotated around it. So, having the calsource very close to one antenna will limit this issue.

2 Code

2.1 Organisation

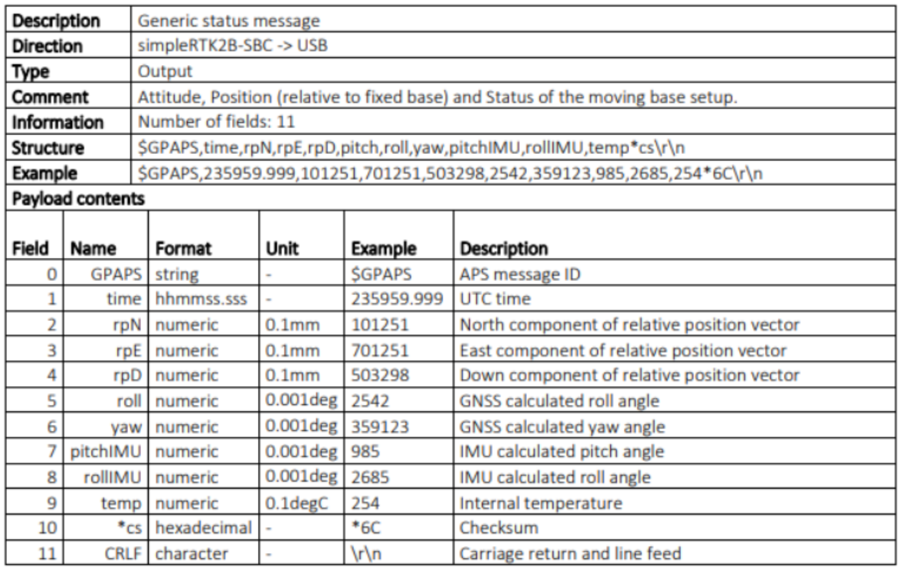
This code is divided into three classes, namely: GPSTools, GPSAntenna, and GPSCalsource.
As a brief resume, GPSTools contains all the base tools to be able to use the GPS data. GPSAntenna is used to compute the position of the two GPS antennas from GPS data and the known distance between the two. It inherits from GPSTools. GPSCalsource will inherit from GPSAntenna. It will use the initial known positions of the antennas and the calibration source (calsource in the following) to compute the position and orientation of it at any time from the position of the two antennas at this time.

2.2 GPSTools

As explained above, this code contains the main functions to convert the raw GPS data into usable variables.

2.2.1 Initialization

This function takes one single argument `gps_data_path`, which can be either the path of the GPS binary file or the dictionary containing the GPS data. When initializing the class, if this variable is a dictionary, it will store it as an internal variable name `gps_data`. If `gps_data_path` is a string, it as to be a path to a binary file, as given by the GPS. The program will import this file, convert the binary information into readable ones, and store it in a dictionary. In both cases, the dictionary will contain the following information :



Description	Generic status message
Direction	simpleRTK2B-SBC -> USB
Type	Output
Comment	Attitude, Position (relative to fixed base) and Status of the moving base setup.
Information	Number of fields: 11
Structure	$GPAPS,time,rpN,rpE,rpD,pitch,roll,yaw,pitchIMU,rollIMU,temp*cs\r\n
Example	$GPAPS,235959.999,101251,701251,503298,2542,359123,985,2685,254*6C\r\n

Payload contents

Field	**Name**	**Format**	**Unit**	**Example**	**Description**
0	GPAPS	string	-	$GPAPS	APS message ID
1	time	hhmmss.sss	-	235959.999	UTC time
2	rpN	numeric	0.1mm	101251	North component of relative position vector
3	rpE	numeric	0.1mm	701251	East component of relative position vector
4	rpD	numeric	0.1mm	503298	Down component of relative position vector
5	roll	numeric	0.001deg	2542	GNSS calculated roll angle
6	yaw	numeric	0.001deg	359123	GNSS calculated yaw angle
7	pitchIMU	numeric	0.001deg	985	IMU calculated pitch angle
8	rollIMU	numeric	0.001deg	2685	IMU calculated roll angle
9	temp	numeric	0.1degC	254	Internal temperature
10	*cs	hexadecimal	-	*6C	Checksum
11	CRLF	character	-	\r\n	Carriage return and line feed

Figure 1: Caption

The code will then extract all the GPS information from this dictionary into arrays. These data will be converted into International Units (meter and radian). For example, the data **rpN** will be extracted, converted into a ndarray, and then divided by 10000.
Important remark: the down data is inverse, to correspond to the more usual z-axis in Cartesian coordinates. I will still use the word Down but it has to be understood as a vertical axis pointing to the sky.

2.2.2 Plots

This class contains functions to plot the data with time. You can either plot only the position vector components, the angle, or both, using respectively: `plot_gps_position_vector`, `plot_gps_angles`, and `plot_gps_data`. For example, the last function gives this plot :

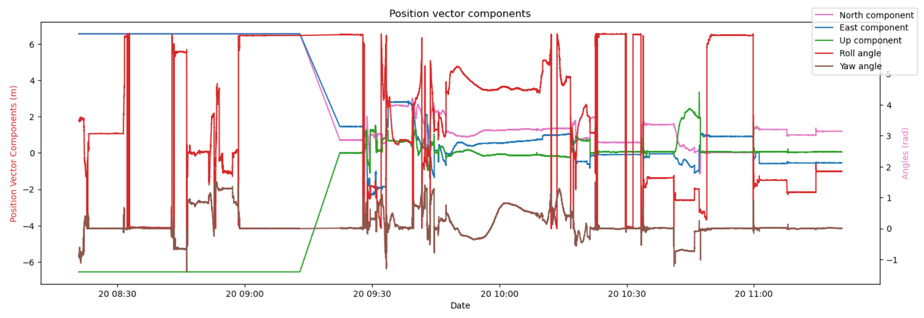


Figure 2: Caption

Each of these functions takes two optional arguments: the starting and the ending indices.

2.2.3 Other tools

GPSTools provides other tools, mainly used in the two other classes.

- `create_datetime_array` will convert the timestamp array in a datetime array (Datetime guide). This function takes a timestamp array and the option argument `utc_offset`, set to zero by default.
- `datetime_to_index` is a very useful function that takes two arguments: a datetime array and an observation date. This one can be in two different formats: string ('year-month-dayThour:minute: second') or datetime format (dt.datetime(year=2024, month=11, day=20, hour=10, minute=24, second=0)). It returns the index associated with the asked date.
- `get_observation_indices` is an extension of the previous function and takes to argument: a datetime array and an array of observation time. This array can have a shape of one (for one single observation time) or a shape of two (the first value is the beginning of observation and the second is the end of observation). It returns then an array of all the observation indices.

2.3 GPSAntenna

This class inherits from GPSTools. Then, it contains all the array of the GPS data. Its goal is to use them to compute the position of the two antennas.

2.3.1 Initialization

This class takes two arguments: `gps_data_path`, which is the argument needed to initialize GPSTools, and `distance_antennas`, which is the distance between the two antennas in meters. This distance is mandatory to compute the position of one of the two antennas. When initialized, the class computes the position of the two antennas.
Remark: the number of each antenna has been written before the understanding of the system. By so, antenna 2 position is trivially computed and we need it to compute antenna 1 position.

2.3.2 Antenna 2

It's very easy to compute the position of the antenna 2, using the **rpN**, **rpE**, and **rpD** variables. These three give directly the position of the antenna 2 in the (North, East, Down) frame, centered on the base antenna position. This computation is done by the function `get_position_antenna_2`, which takes the optional argument `base_antenna_position`. By default, this position is fixed at the origin of the frame in the code (but can be modified to QUBIC position for example).

2.3.3 Antenna 1

The antenna 1 position is a bit more tricky to compute, as the GPS does not provide directly its coordinates. It provides instead two angles between the vector formed by the two antennas and the North and Down axis of the frame, namely **roll** and **yaw**. These two give then the azimuth and elevation angles of antenna 1 from the antenna 2 point of view. It is why we need the distance between the two, to be able to apply the coordinates change formula:

$$\begin{aligned} rpN &= r\cos(\theta)\sin(\pi/2-\phi), \\ rpE &= r\sin(\theta)\sin(\pi/2-\phi), \\ rpD &= r\cos(\pi/2-\phi), \end{aligned} \tag{1}$$

where $\theta = roll$ and $\phi = yaw$.
After that, we need to add the coordinates of antenna 2 in order to have the coordinates of antenna 1 in the same frame.

2.4 GPSCalsource

This class inherits from GPSAntenna. It then also inherits from GPSTools and contains all the GPS data in addition to the antennas' position. It aims to compute the orientation and the position of the calibration source using the movement of the two antennas. As said before, it is mandatory to consider the system formed by the two antennas and the calsource as rigid, otherwise we can't compute the calsource's position and orientation.

2.4.1 Initialization

This class has more arguments than the previous ones. We have `gps_path_data` used to call the GPSAntennas class. We have to give the initial position of the two antennas and of the calsource in the base (North, East, Down) frame. We will consider that at this time the calsource is pointing directly at the QUBIC instrument, and then use these initial positions as a reference to compute movements and changes in orientation. Finally, we have to give the `observation_data` array, which contains the observation time, or the range of the observation period, as described in the GPSTools section.
GPSCalsource takes also three optional arguments (we can think about adding others to make the code more general, in particular regarding the plots). The first is `distance_antennas`. You can define by hand the distance between the two GPS antennas, by default is False. In this specific case, we use the initial position of the two antennas to compute the distance between them. Also, we have `qubic_position` in the (North, East, Down) frame centered on the base antenna, by default is (0, 0, 0). We then have the boolean `observation_only`. If True, it will keep only the GPS data within the observation range, by default is False.

When initialized, the function will compute the distance between the two antennas (or use the given one if it is the case) to initialize the GPSAntenna class build datetime and observations indices array, using the tools in GPSTools. If `observation_only` is True, we will keep only the GPS data from GPSTools within the observation time range.
After that, we define the initial vectors between antenna 1 and antenna 2, and between the calibration source and QUBIC, and the vector between the two antennas at each time. We will use them to compute the orientation and the position of the calsource.

2.4.2 Calsource orientation

To compute the calsource orientation at each time, the idea is to compute the rotation between the initial vector and the current vectors between antennas, and then apply this rotation to the initial vector between the calsource and QUBIC to know where the source is pointing at another time.
To compute this rotation, we used the Rotation module from `scipy.spatial.transform.Rotation` (Scipy Rotation). It is done in the `compute_rotation` function. This function computes in a very general manner the rotation between two vectors. The idea is to compute the dot and cross products between these vectors, and then use the following formula:

$$\begin{aligned} \vec{v_1} \cdot \vec{v_2} &= |\vec{v_1}| \cdot |\vec{v_2}| \cdot \cos(\theta) \\ \vec{v_1} \times \vec{v_2} &= |\vec{v_1}| \cdot |\vec{v_2}| \cdot \sin(\theta) \end{aligned} \tag{2}$$

Here, θ defines the angles between the two vectors. We prefer to use the `np.arctan2` method rather than `np.arccos`, as the first is define within $[-\pi, \pi]$ and the first one within $[0, \pi]$, meaning that `np.arccos` does not provide the full angle range. These angles are computed in the function `compute_angle`, and stored in `deviation_angle`.
In `compute_rotation`, we compute the rotation between two vectors. In order to do that, we compute the angles between them using the previous function and we define the unitary rotation vector as $\vec{n} = \frac{\vec{v_1} \times \vec{v_2}}{||\vec{v_1} \times \vec{v_2}||}$. We initialize the SciPy Rotation instance using the product between the two. We finally define the function `apply_rotation`, which takes a vector and a Rotation instance and returns the rotated vector. It is used to compute the deviated line of sight of the calsource, the vector director of this axis is stored in `vector_calsource_orientation`.

2.4.3 Calsource Position

To compute the calsource position, we need to consider another hypothesis to simplify our problem. Indeed, as said in the introduction, we are blind to rotation around the axis formed by the two antennas. It means that we have a degeneracy on the position of the calsource: it can be anywhere on the circle formed by the intersection of the two spheres of radius calsource-antenna1 and calsource-antenna2, centered respectively on antenna 1 and antenna 2. To break it, we made the very simplification that the calsource position can't rotate around the axis formed by the antennas vector. This approximation seems a bit unpleasant as it does not feel well justified. To make this hypothesis reasonable, we will use another specificity in our physical system: we will align the line of sight of the calsource and the antennas vector. As the calsource's beam is symmetrical, we don't care about the rotation around this axis. Also, by placing one of the two antennas very close to the calsource, we limit the possibility that the calsource can rotate around this axis, and in any case, it makes it negligible from the instrument's point of view.
To come back to the code, and with these simplifications, it becomes very easy to compute the position of the calsource at any time. We just have to compute the translation to go from the antenna placed close to the calsource at the initial position to the same antenna at the current time. Then, we just have to apply this translation to the initial position of the calsource. This is done in the function `get_calsource_position`, where we consider that it's antenna 2 which is placed close to the calsource (become the GPS gives directly its position, which is not the case for antenna 1 position which depends on the distance between the antennas and then on another approximation). We store the position of the calsource in `position_calsource`.
To have more exploitable information, we write the `cartesian_to_azel` method to convert a given position in cartesian coordinates into (Azimuth, Elevation) coordinates. We apply this function to the position of the calsource to build `position_calsource_azel`.

2.4.4 Plots

To end this guide, we write a function to plot the state of the system at a given time and compare it to the system at the initial time. This is done in `plot_calsource_deviation`, which relies on plotly to have a clean and interactive plot.
Remark: The function `plot_calsource_deviation_alt` does the same thing but using matplotlib instead of plotly.
This function takes only one argument: the observation index. It will plot the initial and current positions of the two antennas and the calsource, the positions of the base antenna, and QUBIC. It will also plot the initial and current vectors between the two antennas and the orientation of the calsource. Here is an example of this function :

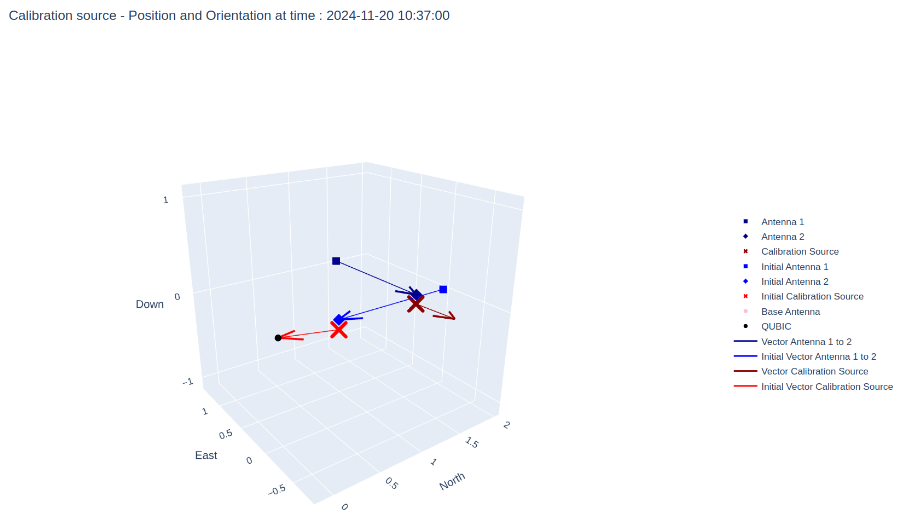


Figure 3: Caption

3 User guide

In this section, we provide an example of how to use the `Qgps` code. This example can be found on the qubicsoft GitHub repository, following: Qgps example.