Faculty of Civil and Industrial Engineering
Department of Civil, Constructional and Environmental Engineering

# RTCM2PVT: an innovative real-time tool for GNSS precise PVT estimation

Candidate: Alessio Conte
Student ID: 1427719
Advisor: Augusto Mazzoni

Academic Year 2017-2018

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

The GPS positioning system is known for its capability to provide the receiver's coordinates in a given reference frame. Generally speaking, the receiver performs raw observations of the signals satellites signal's, that are also containing the navigational messages, necessary to know when and where is the satellite when it sends the signal. However, because of the structure of the problem, there are several types of errors that affect the final solution, like, for example, atmospheric noises or clocks offsets.

There are many established approaches for positioning, that, in general terms can be subdivided into two categories, that are absolute positioning and relative positioning. Absolute positioning, uses one single receiver, while relative positioning uses at least two receivers. The latter gives, as expected, more precise solutions, because, combining the observations of different receivers, reduces most of the uncertainties terms of the equations. However, absolute positioning, (with a stand-alone receiver) gives satisfactory solution for some application like navigation.

Another interesting approach with a stand-alone receiver is that one which uses the GPS as a velocimeter, hence, giving the 3D velocities of the receiver. This method was implemented by the Geodesy and Geomatics division of the Sapienza University of Rome [6]. The approach bases its's principle on the raw observations of the receiver, on two consecutive epochs. In this way is it possible to estimates the receiver displacements epoch by epoch, and, since the receiver works with 1 Hz frequency, those displacements are essentially velocities. This method is the so-called "variometric" approach.

Until now there wasn't a tool able to apply the variometric approach in real-time. It was only possible to take the raw observations and, in post-processing, apply the equations. *The objective of this thesis is to develop a tool able to apply the variometric approach in real-time.*

It was done basically into two different steps: decoding the signal coming from the receiver with the RTCM standard (Radio Technical Commission for Maritime Services); implement routines and algorithms capable to use the decoded quantities to implement the variometric approach in real-time. The receiver that has been used for testing is the permanent station M0SE (Rome, Italy).

For the decoding part I must give credits to JCMBSoft (`https://github.com/jcmb`) for sharing its version of the decoder on GitHub, which represented the starting point of my work. For the developing part, I give credits to the Geodesy and Geomatics division of Sapienza University of Rome, for sharing with me the Python library containing the set of class and functions necessary to the success of this work.

# Chapter 2

# The GPS System

The Global Positioning System bases its working principle on the observations of specific electromagnetic signals coming from a constellation of artificial satellites. Through the observations of those signals from a receiver station (standing or moving), is possible to get the position of the station in the cartesian geocentric reference system. The GPS System works appropriately with the contribution of three components: 1) a satellites constellations which transmit signals, 2) a control center on the earth able to manage the whole system, 3) devices able to interpret and elaborate signals and information.

## 2.1 GPS Signals

It is a complex signal made of the superimposition of different beacons: carrier, codes, and messages. On each satellite, four oscillators provide an electromagnetic signal with a frequency $f_0$= 10.23 MH. So from $f_0$ are retrieved the frequency of the two carriers waves: L1, frequency

$154 f_0$, wavelength $\approx$ 19 cm; L2, frequency $120 f_0$, wavelength $\approx$ 24 cm. The use of two different carrier waves is justified by the fact that the user wants to evaluate and manage the error due to the Ionospheric noise, which depends on the frequency. The two carriers L1 and L2 are modulated with three codes: C/A (Course acquisition), frequency $\frac{1}{10} f_0$; is repeated every millisecond, and it is different from one satellite to another because allows the satellite identification; P (precise), frequency $f_p = f_0$; D (data), frequency $f_d$= 50 Hz. C/A and P codes are known as Pseudo Random Noise since are sequences of pseudo-random +1 and -1. D is the navigational message, a well-structured signal, containing ephemerides, which are information required to solve the positioning problem.

## 2.2 GPS ephemerides

The ephemerides are the set of parameters and algorithms that allow calculating the satellite position at any time in a reference system assigned. There are two types of ephemerides:

- *broadcast ephemerides* (trough the navigational message D)
- *precise ephemerides.*

The *broadcast ephemerides* are calculated from the Control Center and are based on the previous orbits of that satellite. Data of the last week are processed, and a first trajectory is estimated with an error of about 100 m. Then, corrections are transmitted every 12/24 hours. In this way, the broadcast ephemerides have an error of about 1 m. The broadcast ephemerides are used both for real-time positioning and for post-processing.

*Precise ephemerides* are based on the tracking of the satellites from a network of 240 stations of the IGS (International GPS Service). They are called Precise ephemerides because are observed and not predicted, and so they have an accuracy of about 20 cm. Generally, they are available one week after the surveys; therefore, they are not usable for real-time positioning.

For the sake of completeness in Tab. 2.1 are described the set of parameters in the *broadcast ephemerides* used to calculate orbits. More details on every single parameter and precise algorithm of operations to compute orbits are in any reference book, like [2].

Transmitted Ephemeris

| Notation | Description |
|---|---|
| $M_0$ | average anomaly at the reference epoch; |
| $\Delta n$ | average difference of motion with respect to the reference motion |
| $e$ | eccentricity of the orbit |
| $\sqrt{A}$ | square root of the semi-major axis |
| $\Omega_0$ | Longitude of the ascending node at the beginning of the GPS week |
| $i_0$ | Inclinazione dell'orbita all'epoca di riferimento |
| $\omega$ | Argument of Perigee |
| $\dot{\Omega}$ | Time derivative of the right ascension |
| $IDOT$ | Time derivative of inclination |
| $C_{uc}$, $C_{us}$ | Cosine and sine of the latitude correction |
| $C_{rc}$, $C_{rs}$ | Cosine and sine of radius correction |
| $C_{ic}$, $C_{is}$ | Cosine and sine of inclination correction |
| $t_{oe}$ | Ephemeris reference epoch in sow |
| $IODE$ | Ephemeris updating epoch |

Tab. 2.1: Ephemeris parameters sent in the navigational message. Thy are used to calculate satellites orbits. Source [2]

## 2.3 GPS Observations

The signal sent by the satellites is picked up by the receiver, which reproduces the signal inside it; The two signals are identical but shifted in time.

Receivers can perform two different measures: pseudo-range observations (on the C/A and P) and phase-range observations on the carrier waves L1 and L2. Both measurements allow to determine the same quantity (receiver - satellite distance) but with different precisions.

### 2.3.1 Pseudo-range observation

The software inside the receiver measures the time delay between the signal reproduced and the signal received. The problem is that receiver and satellites have different clocks which are not synchronised. In fact, suppose that the two clocks are perfectly synchronised with the reference time $t$ (GPS time). In this case the time shift would be

$$\Delta t_R^S = t_R - t^S = \frac{\rho_R^S}{c}$$

being $\rho$ the geometric distance receiver-satellite and $c$ the signal speed in empty space. The problem is that the actual receiver and satellite clocks, respectively, $T_R$ and $T^S$, have a time offset wrt the GPS time $t$:

$$dt_R = T_R - t_S \qquad dt^S = T^S - t^S$$

The time offset of the satellite $dt^S$ is known because is estimated with a polynomial fitting, whose coefficients are inside the navigational message (*broadcast ephemerides*), while the receiver time offset $dt_R$ is unknown. Accordingly the time shift between the receiver clock and the satellite clock is

$$\Delta T_R^S = (t_R - t^S) + (dt_R - dt^S) = \tau + (dt_R - dt^S) \tag{2.1}$$

where $\tau$ indicates the time of flight of the signal. Consequently the Pseudo-Range distance $P_R^S$ (Pseudo because have the synchronisation error) is obtained multiplying (2.1) by the speed light $c$:

$$P_R^S = c\tau + c(dt_R - dt^S) \tag{2.2}$$

The term $c\tau$ must be corrected because, the signal propagates in the atmosphere and not in empty space, therefore, we must take into account the Ionospheric and Tropospheric corrections, respectively $I_R^S(t)$ and $T_R^S(t)$, considered known trough synthetic models. Therefore, the final pseudo-range equations is

$$P_R^S(t) = \rho_R^S(t) + c(dt_R(t) - dt^S(t)) + I_R^S(t) + T_R^S(t) \tag{2.3}$$

. being $\rho_R^S(t)$ the geometric distance receiver-satellite

$$\rho_R^S(t) = \sqrt{(X^S - X_R)^2 + (Y^S - Y_R)^2 + (Z^S - Z_R)^2}$$

In (2.3) the unknown are:

- $X_R$ , $Y_R$, $Z_R$ receiver position
- $dt_R(t)$, receiver clock offset.

In conclusion, each observed satellite provides one equation like (2.2), so, it is possible to get the receiver position through the resolution of a system with four unknown. The receiver's software solves the system in real-time giving its position in each epoch[1]. If the receiver observes more than four satellites is possible to estimate its position with greater precision through the least square estimation (see section 2.4.1).

### 2.3.2 Phase-range observation

The phase observation is made on the carrier waves L1 and L2 and what is measured is the phase difference between the signal internally reproduced by the receiver and the received signal. The observation equation at the time t is

$$\Delta\phi_R^S = \phi_R(t) - \phi^S(t-\tau)$$

where $\phi_R(t)$ is the phase of the signal reproduced inside the receiver at time t; $\phi^S(t-\tau)$ is the phase of the satellite signal at time $t-\tau$; $\tau$ is the time of flight of the signal. The measurement is made only in one cycle, ans consequently, $\Delta\phi_R^S$ is comprised between 0 and 1. Being the nominal frequency of the signal $f_0$, and considering the relation (2.4) between phase and frequency (assumed to be constant in time)

$$f_0 = \frac{d\phi}{dt} \tag{2.4}$$

[1]Time interval between two successive measures

with a Taylor expansion we have that

$$\begin{aligned} \phi^S(t-\tau) = \phi^S(t) - tf_0 - \tau f_0 \quad (2.5) \\ \phi^S(t) - f_0\tau - N_R^S(t) \quad (2.6) \end{aligned}$$

In the latter equation $N_R^S(t)$ is the integer number of full cycle the signal makes along the whole time of flight. Therefore [1]

$$\Delta\phi_R^S = \phi_R(t) - \phi^S(t) + f_0\tau + N_R^S(t)$$

As usual, the two clocks of the receiver and the satellite have a time offset, respectively $dt_R$ and $dt^S$, causing a phase offset with respect the phase of an ideal clock synchronized with the reference time (GPS time) $\phi(t)$, for which:

$$\phi^S(t) = \phi(t) + f_0 dt^S \qquad \phi_R(t) = \phi(t) + f_0 dt_R$$

Therefore the phase observation equation becomes:

$$\Delta\phi_R^S = f_0\tau(t) + f_0(dt_R(t) - dt^S(t)) + N_R^S(t) \quad (2.7)$$

The observation equation (2.7) is multiplied by the signal wavelength, obtaining [3]

$$L_R^S(t) = c\tau(t) + c(dt_R(t) - dt^S(t)) + \lambda N_R^S(t) \quad (2.8)$$

As we did with the pseudo-range, we must consider the Tropospheric and Ionospheric corrections, and, therefore, the finale phase-observation equation is:

$$L_R^S(t) = \rho_R^S(t) + c(dt_R(t) - dt^S(t)) + \lambda N_R^S(t) - I_R^S(t) + T_R^S(t) \quad (2.9)$$

The unknown in (2.9) are:

- $X_R$ , $Y_R$, $Z_R$ receiver position
- $dt_R(t)$, receiver clock offset
- $N_R^S(t)$, integer ambiguity

However, in this case, observing more satellite in one single epoch wouldn't help because, each observation provides the new unknown $N_R^S$. In conclusion, is not possible to estimate the receiver position in one single epoch with phase observations.

### 2.3.3 Summary of errors affecting observations

- measurement errors:$\approx [0.01 \div 0.001]\lambda$ (signal wavelength)
- broadcast ephemeris error: $\epsilon \mathbf{X}^S \approx 1m$
- satellite clock offset error: $\epsilon t^s \approx 1m$
- Ionospheric error (with models): $\epsilon I \approx 0 - 10m$
- Tropospheric error (with models): $\epsilon T \approx 0 - 1m$

I do not go deep in atmospheric models and other errors analysis because it goes out of what is concerning this work, but any reference book may help [4].

## 2.4 Real-Time Applications

### 2.4.1 Single Point Positioning

Single Point Positioning is a positioning method where the receiver coordinates are estimated in one single epoch, starting from the observations (of codes or phases) and the navigational message [4]. The latter provides us with the parameters to model (with errors), the satellite clock and the atmospheric noises terms. Let's consider the code observation equation (2.3). To apply the least square estimation, we need to linearise the equation around an approximate position of the receiver, say $\tilde{X_R}$, $\tilde{Y_R}$, $\tilde{X_R}$, supposed to be known a priori. Since the receiver coordinates are just in $\rho$ (time dependence is removed because we are working in one single epoch, and, for the sake of write cleaning, also S and R), and being

$$\tilde{\mathbf{x}} = \begin{bmatrix} \tilde{X_R} \\ \tilde{Y_R} \\ \tilde{Z_R} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} X_R \\ Y_R \\ Z_R \end{bmatrix} \qquad \delta\mathbf{x} = \mathbf{x} - \tilde{\mathbf{x}}$$

then, the linearisation of $\rho$ is

$$\rho \approx \tilde{\rho} + \nabla_\rho^T(\tilde{\mathbf{x}})\delta\mathbf{x} \tag{2.10}$$

where $\nabla_\rho(\tilde{\mathbf{x}})$ is the gradient of $\rho$ with respect to $\mathbf{x}$, evaluated in the approximate coordinates. Considering (2.10), the linearised observation equation becomes:

$$P_o = \nabla_\rho^T(\tilde{\mathbf{x}})\delta\mathbf{x} + cdt_R + \tilde{\rho} + -cdt^S + I + T \tag{2.11}$$

and letting

$$b_R^S = \tilde{\rho} + -cdt^S + I + T$$

becomes,

$$P_o = \nabla_\rho^T(\tilde{\mathbf{x}})\delta\mathbf{x} + cdt_R + b_R^S \tag{2.12}$$

Computing the gradient $\nabla_\rho(\tilde{\mathbf{x}})$, is possible to see that it is nothing but the unit vector from the approximate receiver and the satellite.

$$\nabla_\rho^T(\tilde{\mathbf{x}}) = \tilde{\mathbf{e}}_R^S$$

Therefore, for one satellite we have the (scalar) linearised observation equation:

$$P_o = b_R^S + \tilde{\mathbf{e}}_R^S\delta\mathbf{x} + cdt_R \tag{2.13}$$

For $m$ satellites it becomes the linear system (4.7) [4]

$$\mathbf{P_o} = \begin{bmatrix}\mathbf{E_R} & \mathbf{i}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x} \\ cdt_R\end{bmatrix} + \mathbf{b} \tag{2.14}$$

which can be solved with the least squares method, if $m > 4$. where:

$$\mathbf{P_0} = \begin{bmatrix} P_o^1 \\ P_o^2 \\ \vdots \\ P_o^m \end{bmatrix} \qquad \begin{bmatrix}\mathbf{E_R} & \mathbf{i}\end{bmatrix} = \begin{bmatrix} \dots\mathbf{e}_R^1\dots & 1 \\ \dots\mathbf{e}_R^2\dots & 1 \\ \vdots & \vdots \\ \dots\mathbf{e}_R^m\dots & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_R^1 \\ b_R^2 \\ \vdots \\ b_R^m \end{bmatrix}$$

.

### 2.4.2 Real Time Kinematic positioning

RTK (Real-Time Kinematic) is an established positioning technique that involves a *base* receiver, whose coordinates are known, and, a *rover* receiver whose coordinates are unknown. The base receiver communicates with the rover in real-time (for example with a network protocol), with an established data format such us RTCM. The data processing at the rover includes ambiguity resolution of a new kind of observation, which is the double difference observation, discussed below, and the estimation of the rover position. One significant drawback is that the maximum distance between base and rover must not exceed 10 -20 km, in order to be able to solve rapidly and reliably the carrier phase ambiguity. This limitation is caused by the distance-dependent biases such as orbit-error, ionospheric and tropospheric signal refraction. With RTK positioning can be reached centimeter-level accuracy [8].

Let's consider then two receivers (A and B) ant their phase-range observations (2.8) referred to the same satellite, S. We can build then a new observation called *single difference*[5]

$$\begin{aligned} L^S_{AB}(t) =& L^S_A(t) - L^S_B(t) && (2.15)\\ =& \rho^S_{AB}(t) + \lambda N^S_{AB}(t) - I^S_{AB} + T^S_{AB}(t) + c(dt_A(t) - dt_B(t)) && (2.16) \end{aligned}$$

where was used for the generic quantity $x$, $x_{AB} = x_A - x_B$. The single difference is a new observation where there aren't the satellite clock term, that would give its contribution to error analysis. Moreover, in this way, the error due to troposphere, ionosphere, and orbits are significantly reduced, because the signals travels trough mostly the same part of atmosphere.

Consider now two receivers (A and B) and, two satellites (I and J). Then we can build the *double differences* observation which is the difference of two single differences [5]

$$L_{AB}^{IJ}(t) = L_{AB}^{I}(t) - L_{AB}^{J}(t) \tag{2.17}$$

$$= \rho_{AB}^{IJ}(t) + \lambda N_{AB}^{IJ}(t) - I_{AB}^{IJ}(t) + T_{AB}^{IJ}(t) \tag{2.18}$$

where, for the generic quantity $x$, $x_{AB}^{IJ} = x_{AB}^{I} - x_{AB}^{J}$. In this new observation, obtained with four phase observations, there are only the geometric unknowns and the double difference ambiguity. The use of double differences allows a *relative positioning*, meaning that we are able to estimate only the components of the three-dimensional vector between A and B [5].

# Chapter 3

# RTCM3 Decoder

*In this chapter, there is a description of the procedures to decode data stream in the RTCM format. Firstly, is described the data structure of the RTCM3 (version 3), then how the decoder works, and then how the tool developed provides useful outputs to implement the so-called variometric approach. For the decoding part of this work I give credits to JCMBSoft (`https://github.com/jcmb`), for sharing its version of the decoder on GitHub, which represented the starting point of my work.*

## 3.1 Data format

The tool developed is going to acquire, through a TCP socket, a byte-array of data sent by the receiver, which for the application is the M0SE permanent station (Rome, Italy). Each byte array is read in with a length of 1024 bytes.

After the socket configuration, the byte-array is stored into a variable with the command `Buffer = clientSocket.recv(1024)`. The variable `Buffer`, is subdivided in different slots called *Frames*, each containing several *Messages*, which are the crucial information for applications. Each Frame is subdivided itself in different slots, as shown in Fig. 3.2.

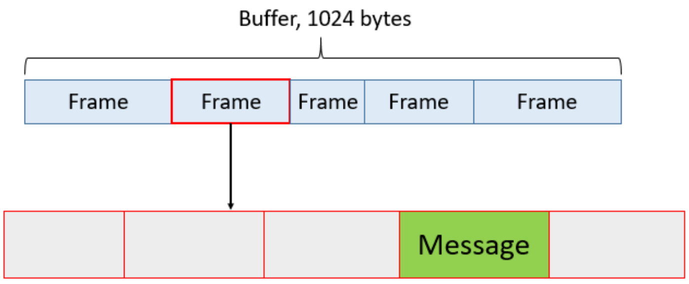
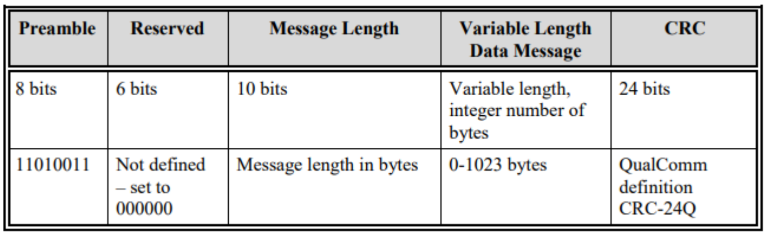


Fig. 3.1: Conceptual sketch representing Buffer and Frames.



| Preamble | Reserved | Message Length | Variable Length Data Message | CRC |
|---|---|---|---|---|
| 8 bits | 6 bits | 10 bits | Variable length, integer number of bytes | 24 bits |
| 11010011 | Not defined – set to 000000 | Message length in bytes | 0-1023 bytes | QualComm definition CRC-24Q |

Fig. 3.2: RTCM3's Frame structure. Picture from RTCM3 Standards [7].

The *Preamble* is a fixed 8-bit sequence. The next 6-bits are *Reserved* and should be fixed to 0 (in future versions these bits may contain the version number of the standard). The third slot expresses the *Message Length* in bytes. Then, the most important, the *Message* that contains the piece of information we are going to decode and to use. At the end of the Frame structure, there are 24 bits to ensure protection against random errors with the *CRC*-24Q algorithm by Qualcomm [9].

### 3.1.1 Message format

Messages are that part of the information that comes within the fourth slot of each Frame (green slot in Fig. 3.2) coming as a sequence of bits that we need to decode to extract usable information. There are many types of messages that the user may want to decode. Therefore they are numbered in *types*, from 1 to 1230.

The types used to implement the variometric approach in this work are

- type 1019 which is the Navigational message containing ephemerides for the orbits computation
- type 1006 containing information on the receiver and the approximate receiver position
- type 1004 which is the Observations message with pseudo-range and phase-range raw observations

However, the decoder works, eventually, for any message.

#### 3.1.1.1 Message's data fields

Each message contains specific set of data fields, sometimes repeated as information on several satellites is provided (like for message 1004).

| **DF#** | **DF Name** |
|---|---|
| DF002 | Message Number |
| DF004 | GPS Epoch Time (TOW) |
| DF009 | GPS Satellite ID |
| DF011 | GPS L1 Pseudorange |
| DF025 | Antenna Ref.Point ECEF-X |
| DF026 | Antenna Ref.Point ECEF-Y |
| DF027 | Antenna Ref.Point ECEF-Z |
| DF081 | GPS Time of emission |
| DF090 | GPS Eccentricity |

Tab. 3.1: A small subset of RTCM3 standard's data fields

The *data fields*, in each message, are broadcast in the order listed in the tables presented in 3.1.1.2, 3.1.1.3, 3.1.1.4. Data fields are essentially telling us how to subdivide the message bit-array. As a matter of example, in Table: 3.1 are shown some data field of the RTCM3 Standard. DF Numbers are just numerical identifiers. Each message has only specific data fields. For example, looking at the table above, DF004, DF009, and DF011 are in messages of type 1004; DF025, DF026, and DF027 are in messages of type 1006; DF081 and DF090 are in messages of type 1019. Data fields in RTCM3 standard range form DF001 to DF515. As a matter of example in the next table are shown just some of the data fields constituting message 1004:

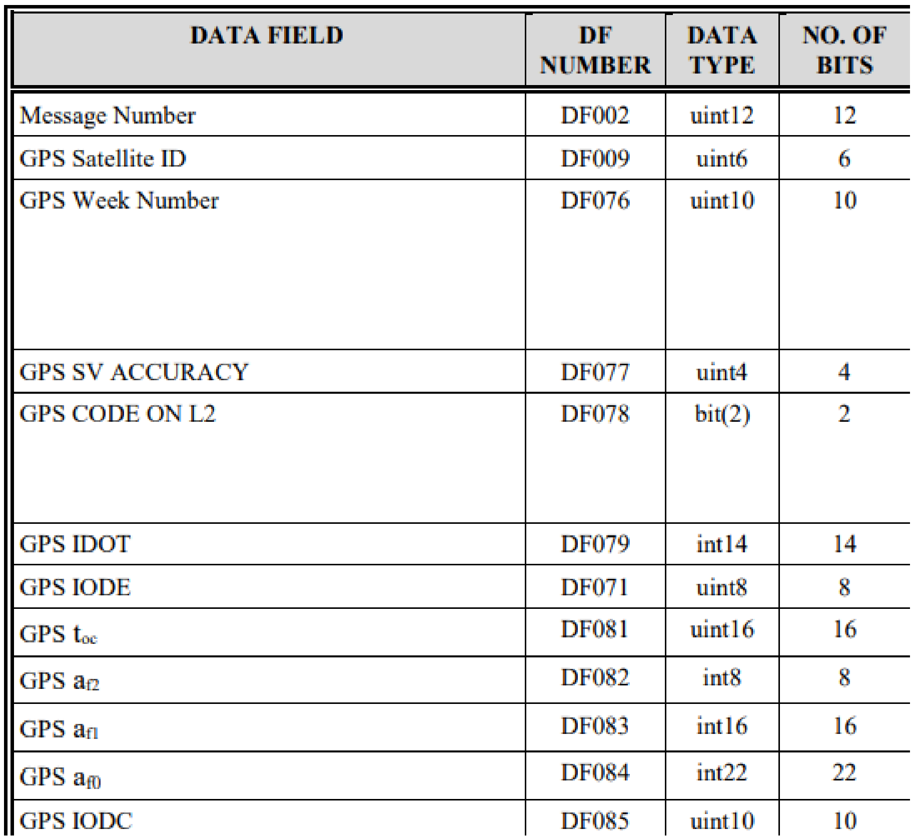
**Message 1004**

| **Data field** | **DF Number** | **Data Type** | **No. of Bits** |
|---|---|---|---|
| Message Number | DF002 | uint12 | 12 |
| GPS Epoch Time (TOW) | DF004 | uint30 | 30 |
| GPS Satellite ID | DF009 | uint6 | 6 |
| ⋮ | ⋮ | ⋮ | ⋮ |

The Variable data type indicates the numerical representation that the bit array of that data field is representing. The most used are: `bit(n)`, to represent true or false information; `intn`, n bit 2's complement to represent signed integers; `uintn`, n bit unsigned integer, to represent positive integers.

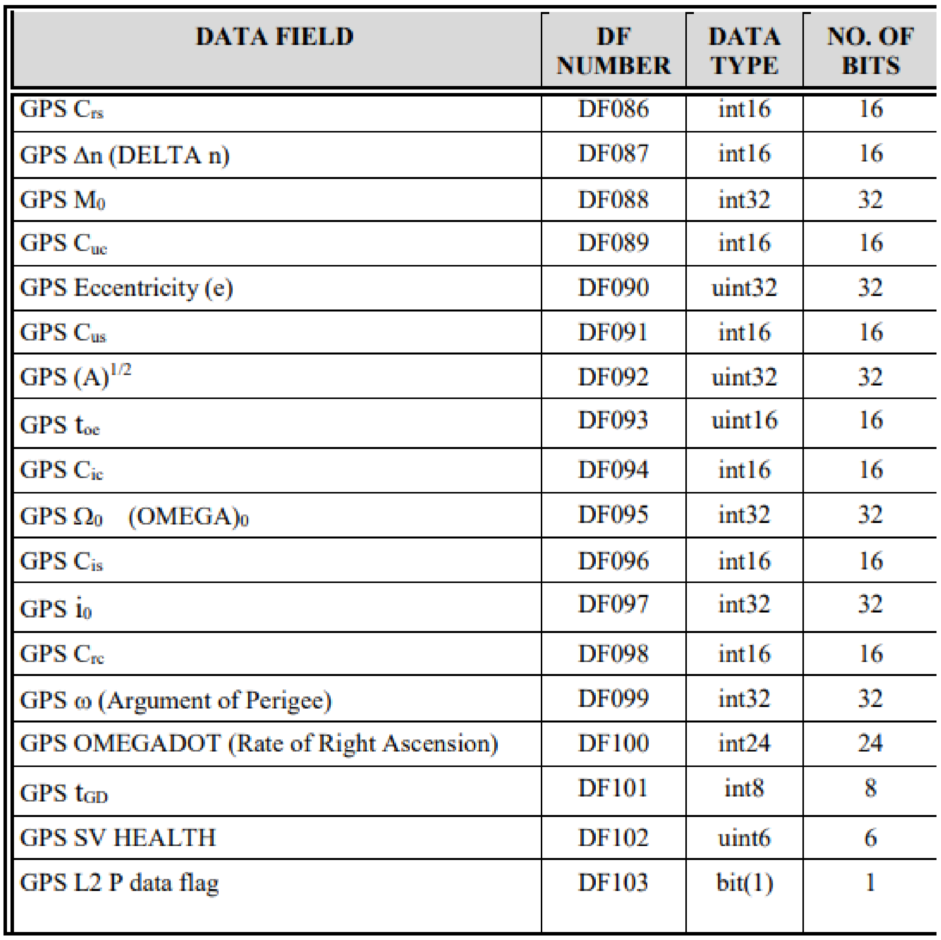
For example, the first 12 bits of each type of message are expressing the Message Number (DF002). If the first 12 bits of an arriving message are '001111101100', since DF002 is representing an unsigned integer (`uint12`), trough a normal binary to decimal conversion, is possible to conclude that the message is of type 1004 (001111101100 = 1004).

#### 3.1.1.2 Message 1019 data fields



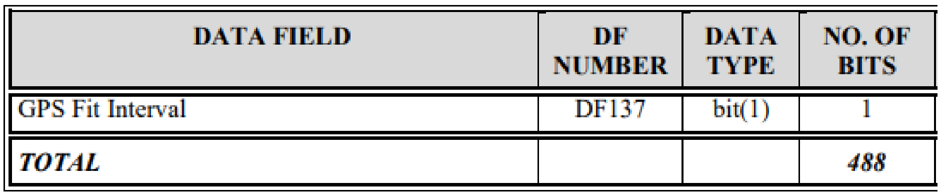
| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| Message Number | DF002 | uint12 | 12 |
| GPS Satellite ID | DF009 | uint6 | 6 |
| GPS Week Number | DF076 | uint10 | 10 |
| GPS SV ACCURACY | DF077 | uint4 | 4 |
| GPS CODE ON L2 | DF078 | bit(2) | 2 |
| GPS IDOT | DF079 | int14 | 14 |
| GPS IODE | DF071 | uint8 | 8 |
| GPS $t_{oc}$ | DF081 | uint16 | 16 |
| GPS $a_{f2}$ | DF082 | int8 | 8 |
| GPS $a_{f1}$ | DF083 | int16 | 16 |
| GPS $a_{f0}$ | DF084 | int22 | 22 |
| GPS IODC | DF085 | uint10 | 10 |

Fig. 3.3: Contents of the type 1019 Message, Part 1.



| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| GPS $C_{rs}$ | DF086 | int16 | 16 |
| GPS $\Delta n$ (DELTA n) | DF087 | int16 | 16 |
| GPS $M_0$ | DF088 | int32 | 32 |
| GPS $C_{uc}$ | DF089 | int16 | 16 |
| GPS Eccentricity (e) | DF090 | uint32 | 32 |
| GPS $C_{us}$ | DF091 | int16 | 16 |
| GPS $(A)^{1/2}$ | DF092 | uint32 | 32 |
| GPS $t_{oe}$ | DF093 | uint16 | 16 |
| GPS $C_{ic}$ | DF094 | int16 | 16 |
| GPS $\Omega_0$ $(OMEGA)_0$ | DF095 | int32 | 32 |
| GPS $C_{is}$ | DF096 | int16 | 16 |
| GPS $i_0$ | DF097 | int32 | 32 |
| GPS $C_{rc}$ | DF098 | int16 | 16 |
| GPS $\omega$ (Argument of Perigee) | DF099 | int32 | 32 |
| GPS OMEGADOT (Rate of Right Ascension) | DF100 | int24 | 24 |
| GPS $t_{GD}$ | DF101 | int8 | 8 |
| GPS SV HEALTH | DF102 | uint6 | 6 |
| GPS L2 P data flag | DF103 | bit(1) | 1 |

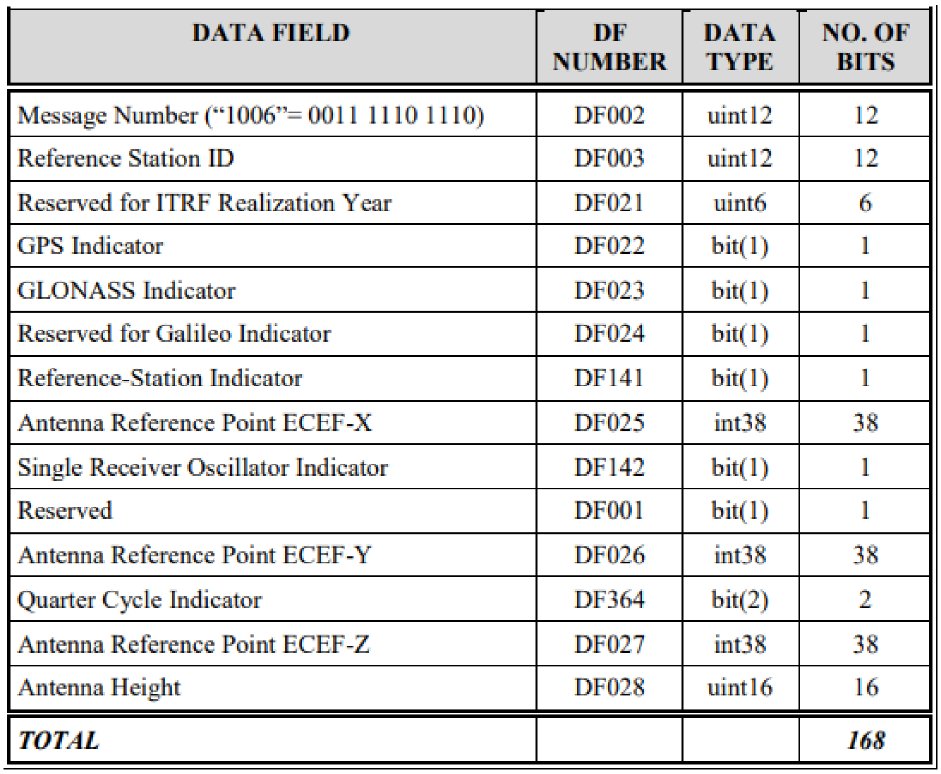
Fig. 3.4: Contents of the type 1019 Message, Part 2.



| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| GPS Fit Interval | DF137 | bit(1) | 1 |
| ***TOTAL*** | | | ***488*** |

Fig. 3.5: Contents of the type 1019 Message, Part 3.

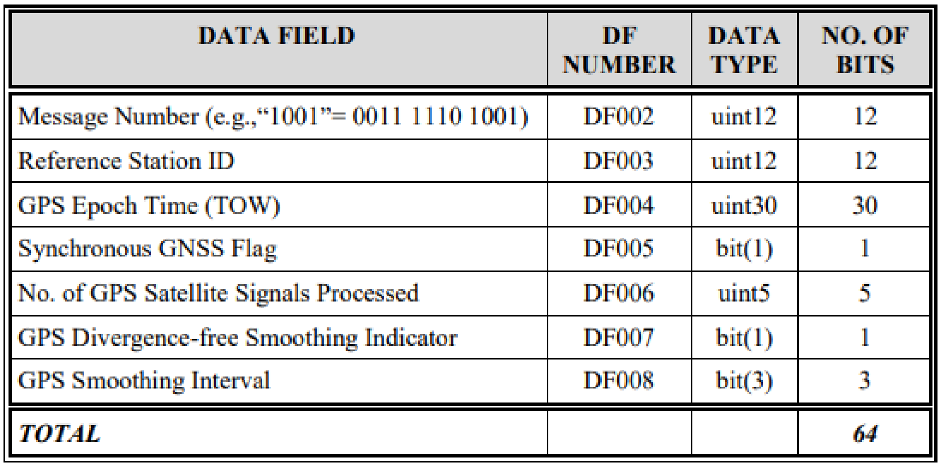
#### 3.1.1.3 Message 1006 data fields



| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| Message Number (“1006”= 0011 1110 1110) | DF002 | uint12 | 12 |
| Reference Station ID | DF003 | uint12 | 12 |
| Reserved for ITRF Realization Year | DF021 | uint6 | 6 |
| GPS Indicator | DF022 | bit(1) | 1 |
| GLONASS Indicator | DF023 | bit(1) | 1 |
| Reserved for Galileo Indicator | DF024 | bit(1) | 1 |
| Reference-Station Indicator | DF141 | bit(1) | 1 |
| Antenna Reference Point ECEF-X | DF025 | int38 | 38 |
| Single Receiver Oscillator Indicator | DF142 | bit(1) | 1 |
| Reserved | DF001 | bit(1) | 1 |
| Antenna Reference Point ECEF-Y | DF026 | int38 | 38 |
| Quarter Cycle Indicator | DF364 | bit(2) | 2 |
| Antenna Reference Point ECEF-Z | DF027 | int38 | 38 |
| Antenna Height | DF028 | uint16 | 16 |
| ***TOTAL*** | | | ***168*** |

Fig. 3.6: Contents of the type 1006 Message

#### 3.1.1.4 Message 1004's data fields



| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| Message Number (e.g.,"1001"= 0011 1110 1001) | DF002 | uint12 | 12 |
| Reference Station ID | DF003 | uint12 | 12 |
| GPS Epoch Time (TOW) | DF004 | uint30 | 30 |
| Synchronous GNSS Flag | DF005 | bit(1) | 1 |
| No. of GPS Satellite Signals Processed | DF006 | uint5 | 5 |
| GPS Divergence-free Smoothing Indicator | DF007 | bit(1) | 1 |
| GPS Smoothing Interval | DF008 | bit(3) | 3 |
| ***TOTAL*** | | | ***64*** |

Fig. 3.7: Contents of the Message Header, Type 1004

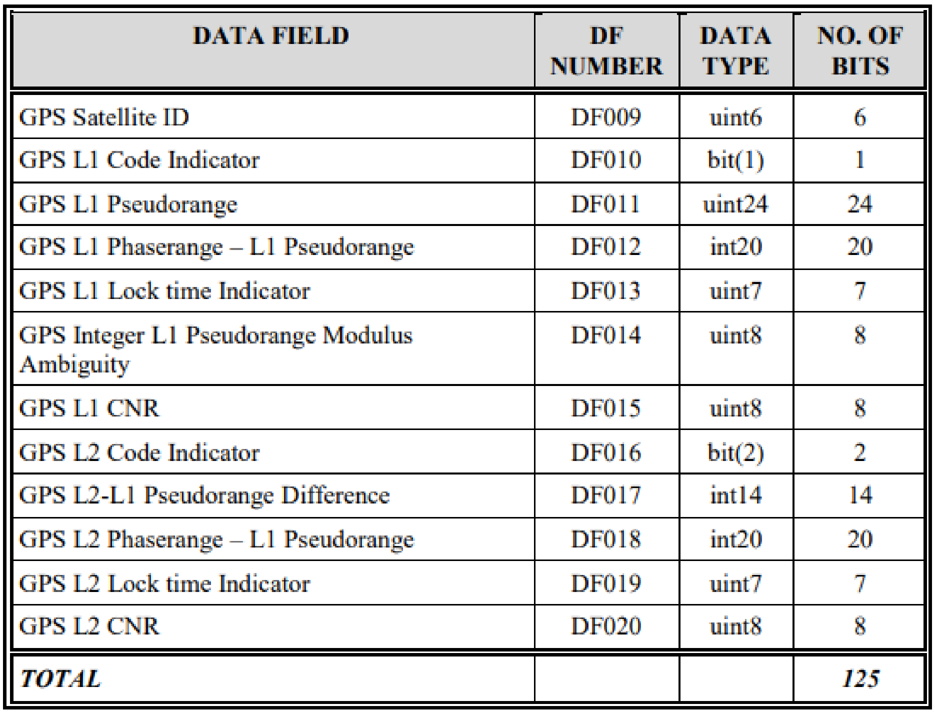
## 3.2 Preliminary procedural approach for decoding

Is it possible to summarize the decoding procedure in three steps:

- Representing data fields
- Processing the Buffer to extract Messages
- Decoding Messages

### 3.2.1 Representing data fields

Imagine the message as sequence of '1001100101...' that we need to decode. Before decoding, we must say how this bit-array is subdivided.



| DATA FIELD | DF NUMBER | DATA TYPE | NO. OF BITS |
|---|---|---|---|
| GPS Satellite ID | DF009 | uint6 | 6 |
| GPS L1 Code Indicator | DF010 | bit(1) | 1 |
| GPS L1 Pseudorange | DF011 | uint24 | 24 |
| GPS L1 Phaserange – L1 Pseudorange | DF012 | int20 | 20 |
| GPS L1 Lock time Indicator | DF013 | uint7 | 7 |
| GPS Integer L1 Pseudorange Modulus Ambiguity | DF014 | uint8 | 8 |
| GPS L1 CNR | DF015 | uint8 | 8 |
| GPS L2 Code Indicator | DF016 | bit(2) | 2 |
| GPS L2-L1 Pseudorange Difference | DF017 | int14 | 14 |
| GPS L2 Phaserange – L1 Pseudorange | DF018 | int20 | 20 |
| GPS L2 Lock time Indicator | DF019 | uint7 | 7 |
| GPS L2 CNR | DF020 | uint8 | 8 |
| ***TOTAL*** | | | ***125*** |

Fig. 3.8: Contents of the Satellite-Specific Portion of a Type 1004 Message, Each Satellite

For example, in a message of type 1004 the first 12 bits are referring to the Message Number data field (DF002), and the next 12 bits to the Reference Station ID (DF003), and so on. In this way, decoding can be made easily making an index that moves along the byte array, where the length of each move is the number of bits of that specific data field.

To import that information the tool must read some text files containing all the data fields specifications. There must be one text file for each message we wish to decode. For example, if we wish to decode messages of type 1004, 1006 and 1019, the tool must read in three files. Those files, from now on are called *Definitions Files*. The func-

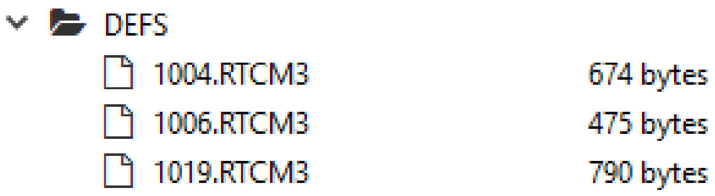

```
DEFS
  1004.RTCM3        674 bytes
  1006.RTCM3        475 bytes
  1019.RTCM3        790 bytes
```

Fig. 3.9: Example of Definition Files.

tion which processes those files is the `read_from_file` function, which take as input one file and gives all the data fields informations inside a Python list named `fields`. Therefore, for each file we will have a variable `fields`. For example, if `read_from_file` takes in the Definition file for message 1006, it gives as output the list represented in the following table:

fields1006

| Index | type | name | df_num | bitlength | value |
|---|---|---|---|---|---|
| 0 | UINT | Message Number | 2 | 12 | None |
| 1 | UINT | Reference Station ID | 3 | 12 | None |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 13 | UINT | Antenna Height | 28 | 16 | None |

Each row represent one element of the list which is a Python dictionary having as keys the variable in the first row: *type*, *name*, *df_number*, *bitlength*, *value*.

Example of first and second element of fields, for message of type 1006

```
>>fields1006[0]

{'type': 'UINT',
'name': 'Message Number',
'df_number': 2,
'bitlength': 12,
'value': None}

>> fields1006[1]

{'type': 'UINT',
'name': 'Reference Station ID',
'df_number': 3,
'bitlength': 12,
'value': None}
```

The last column is the variable for the decoded quantity, and, for each data field, is set to *None*. Indeed, *None* will be replaced with the actual value after decoding.

The Definition files to read in, must have a precise syntax, that depends on how the function process the file. Here is shown the Definition file for the message of type 1006, but is easy to extend the concept to the other types.

Definition File for message of type 1006

```
NAME: Stationary Antenna Reference Point, With Height Information
ID: 1006
UINT:12:2:Message Number
UINT:12:3:Reference Station ID
UINT:6:21:ITRF Year (Reserved)
UINT:1:22:GPS Indicator
UINT:1:23:GLONASS Indicator
UINT:1:24:Galileo Indicator (Reserved)
UINT:1:141:Reference Station Indicator
INT:38:25:ECEF-X
UINT:1:142:Single Receiver Oscillator Indicator
UINT:1:1:Reserved
INT:38:26:ECEF-Y
UINT:2:364:Quarter Cycle Indicator
INT:38:27:ECEF-Z
UINT:16:28:Antenna Height
END:
```

### 3.2.2 Processing the Buffer to extract Messages

Once the `Buffer` is received from the TCP socket, before to start decoding, the tool must extract messages by every single Frame of the Buffer (Fig. 3.2). Moreover, only if CRC test passes, messages can be decoded. Let's say for now that this procedure is performed by the function `process_data`, which takes in the Buffer and return a variable named `result` that represent how the process evolved. Therefore, the algorithm describing what `process_data` does is: As shown by the algorithm, at any step, a variable `result` is returned, and `Buffer` is updated. In this way the algorithm is inserted inside a while loop that process data until `result` becomes 0 (which means that more data are needed), that happens in two cases: a) `Buffer` length isn't enough,

```
Take the whole Buffer as input;
if Buffer length $\geq$ 48 bit[1] then
    go to the next block
else
    return result = 0
end
if the first 8 bit are the Preamble's bit then
    go to next block
else
    go on looking for the first Preamble;
    and update the Buffer;
    if Preamble is found then
        go to next block
    end
    if No Preamble is found then
        return result = 3
    end
end
Get the message length;
if message length $\geq$ Buffer length then
    return result = 0
else
    Go to next block
end
Make the CRC test;
if the CRC test passes then
    get the message bits and decode the message;
    return result = 4
else
    update the buffer;
    return result = 3
end
```

**Algorithm 1:** Procedure implemented by function process_data. At each step the buffer is updated and a variable result is returned.

| Value of result | Meaning |
|---|---|
| 0 | Buffer to process isn't enough |
| 3 | Message Un-decoded |
| 4 | Message Decoded |

and b) all the `Buffer` has been processed, and is finished. Once that the whole Buffer has been decoded, another Buffer comes in and the procedure starts again.

Example of how process_data is called in a while loop

```python
# Buffer is acquired
result = process_data(Buffer) # first processing
while result != 0:
    if result == 3:
        # message not decoded
    elif result == 4:
        # message decoded
    else:
        # this result doesn't exists

    # Process data again in the loop
    result=process_data(Buffer)
```

### 3.2.3 Decoding Messages

Now we have all the ingredients to decode messages. In fact the function `process_data` provides *Message*'s byte-arrays, and `read_from_file` provides informations about data fields. Therefore, the `decode` function takes the byte-array of the message and replace all the `None` in `fields` with the actual decoded value.

The conversion method must consider the *type* of the data field. So for example, if the data field type is `uint`, than the conversion is standard binary to decimal, otherwise, if the type is `int`, that a 2's complement conversion is needed. Here is shown a rough sketch of the *decode* function.

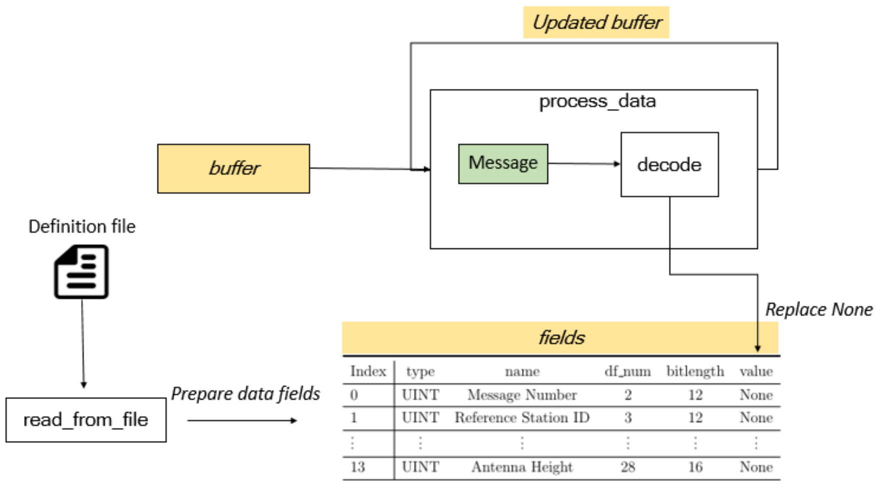
Rough sketch of the decoding function

```
def decode(message_data):
for field in fields: #each field is a dictionsry
    if field['type']=='UINT':
        field['value']= # make correct conversion
        current_bit += field["bitlength"]
    elif field['type']=='INT':
        field['value']= # make correct conversion
        current_bit += field["bitlength"]
```

Next table shows an example of how the variable **`fields`** looks like, after decoding.

fields1006

| Index | type | name | df_num | bitlength | value |
|---|---|---|---|---|---|
| 0 | UINT | Message Number | 2 | 12 | 1006 |
| 1 | UINT | Reference Station ID | 21 | 6 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 12 | UINT | ECEF-Z | 27 | 38 | 42368540159 |
| 13 | UINT | Antenna Height | 28 | 16 | 0 |



| Index | type | name | df_num | bitlength | value |
|---|---|---|---|---|---|
| 0 | UINT | Message Number | 2 | 12 | None |
| 1 | UINT | Reference Station ID | 3 | 12 | None |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 13 | UINT | Antenna Height | 28 | 16 | None |

Fig. 3.10: Conceptual work-flow for decoding a single Message. The fields variable is prepared on the base of the Definition file. On the other hand, the function process_data extract messages from each frame of the buffer and decodes them. The decoding function replaces None with the actual decoded quantity, in fields. Updated buffer enters again in process_data until there are no more enough bytes to process. Note that, in this way, at each cycle, decode overwrites the last value.

## 3.3 Real time decoding

Until now, the decoding work-flow was explained with a procedural approach, thus describing what each function does. This approach is the better way if to understand a quite sophisticated tool. The initial Decoder from `https://github.com/jcmb` was composed of three files, one main file, and two class files, and was quite challenging to understand all the dependencies, from one file to another. Therefore, the best way was to unpack methods and attributes from their classes and try to understand, what was going on, function by function and step by step.

### 3.3.1 *Decode* main file step by step

The main file of the initial Decoder (the starting point of this thesis work), is the *Decode.py*. The other two are *RTCM3.py*, containing a class representing each incoming `Buffer`, and *RTCM3_Definitions.py*, to manage the data fields representation. This thesis don't focus on

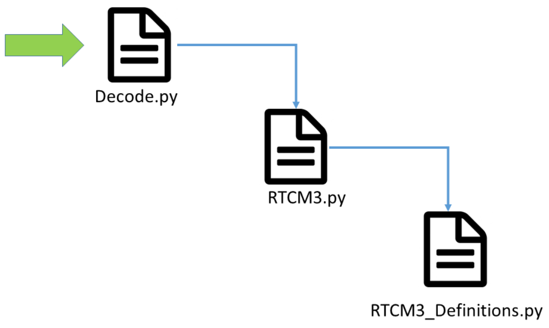


Fig. 3.11: How the three files are called by each other.

what each file does, line by line, but, to understand how the fundamental structure of the decoder works, here is analysed step by step the main file execution.

The complete work-flow can be summarized with the following steps:

- Read in `Buffer` data
- Create a new object named `rtcm3` that contains the three main attributes:

  - `buffer`, which is the byte-array to be processed at each cycle
  - `commands`, which is a Python dictionary with keys 1006,1004 and 1019, and values object of class *RTCM3_Definition*. Those object have the variable `field` as attribute. So, for example, if we are interested in the data fields of messages of type 1006

    ```
    >>rtcm3.commands[1006].fields
    ```

    Furthermore, if we want the value of the third data field

    ```
    >>rtcm3.commands[1006].fields[2]['value']
    ```

  - `packet_ID`, integer number representing the message being decoded (e.g.: 1019)

- Process `Buffer` in the loop, until there are no more bytes to process
- Get another `Buffer`

In following pages the main file (*Decode.py*) is subdivided in different pieces (listing 3.1, 3.2, 3.3), to explain, step by step, how the process evolves. The main file is represented with a coloured background, while the console with the symbol `>>`

1. **Read in data**

Listing 3.1: Read in data.

```python
import socket # python module for socket managment
import RTCM3 # class RTCM file imported

clientSocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
ip=socket.gethostbyname("151.100.8.117")
port=2130
address=(ip,port)
clientSocket.connect(address)  #handshake

while i==1:
    Buffer=clientSocket.recv(1024)
```

```
>> len(Buffer)
1024
```

2. **Creating the rtcm3 object**

Listing 3.2: Creating the rtcm3 object

```python
    rtcm3=RTCM3.RTCM3()
    new_data = bytearray(Buffer)
    rtcm3.add_data(data=new_data)
```

```
>> rtcm3
<RTCM3.RTCM3 object at 0x00000282386DB5F8>
>> len(rtcm3.buffer)
1024
>>rtcm3.packet_ID
None
>>rtcm3.commands[1019][0]['value']
```

```
None
>>rtcm3.commands[1019][1]['value']
None
>>rtcm3.commands[1019][2]['value']
None
.
.
etc
```

Let's go on with the main file and see how the attributes of `rtcm3` change after the buffer processing.

3. **Buffer processing** (which include decoding as mentioned in 3.2.2)

Listing 3.3: Process Buffer in loop

```
15      result = rtcm3.process_data()
16      while result != 0:
17          if result == 3:
18              # don't do nothing
19          elif result== 4:
20              # print decoded values
21              # record values for further processing
22          else:
23              raise NameError('result',result, 'not identified')
24
25         # Process data again in the loop
26          result=process_data(Buffer)
```

After line 15 the rtcm3 attributes are:

```
>> len(rtcm3.buffer)
957
```

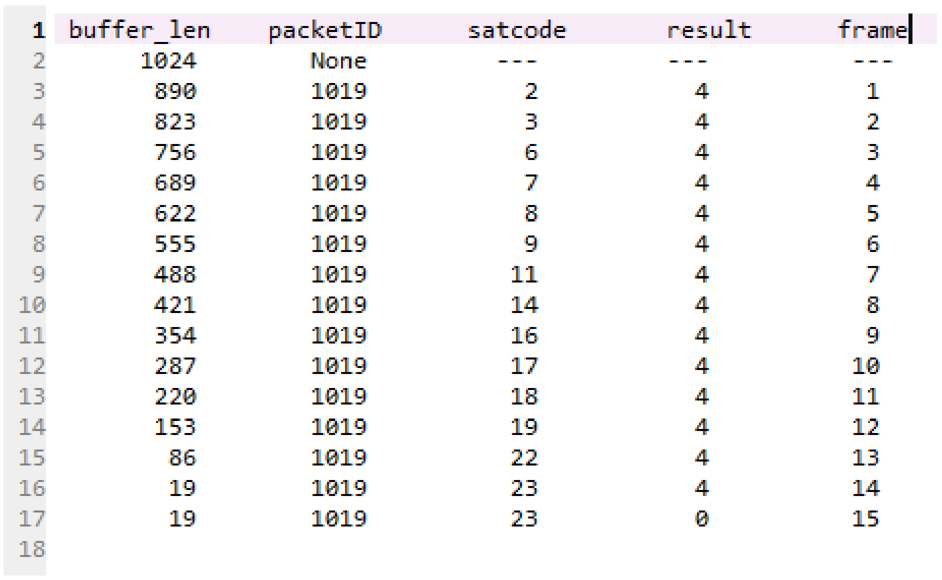
```
>> rtcm3.packet_ID
1019
>>rtcm3.commands[1019].fields[0]['value']
1019 #(Message Number)
>> rtcm3.commands[1019].fields[1]['value']
2 # (Satellite ID)
>> rtcm3.commands[1019].fields[2]['value']
1011 # (GPS week)
.
.
etc
```

Moreover, since `result = 4`, the code execution enters in the while loop (line 16 of code 3.3), where at each cycle the updated buffer is processed again and again, until `result = 0`.

The procedure above goes on until `result` is 0, which means that all `rtcm3.buffer` is processed. Then, a new Buffer comes in. For more clearness, in Fig. 3.12 is shown an example of how the procedure evolves, at each cycle.

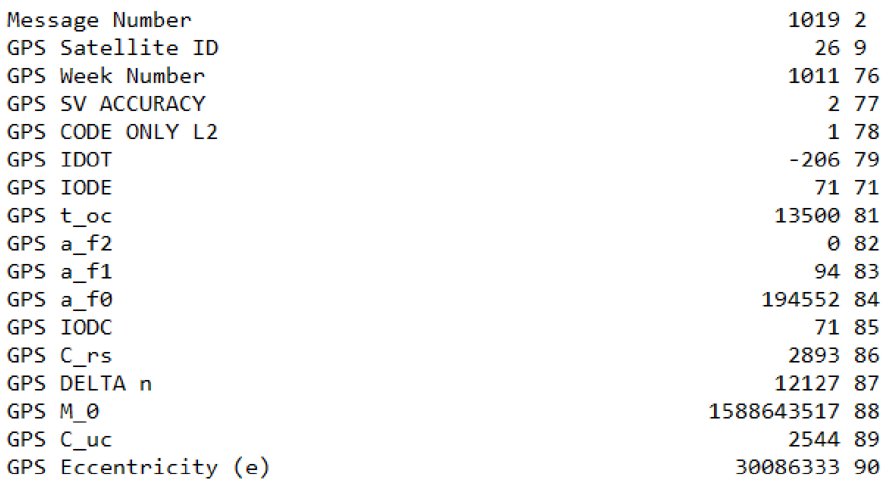
#### 3.3.1.1 Decoded values

At code 3.3 on the preceding page, in line 20, we can eventually print the decoded data. Following figures Fig. 3.13, Fig. 3.14, Fig. 3.15, show some examples of decoded values of messages 1019,1004 and 1006 respecitvely. For obvious reasons of space, only some of them are shown.



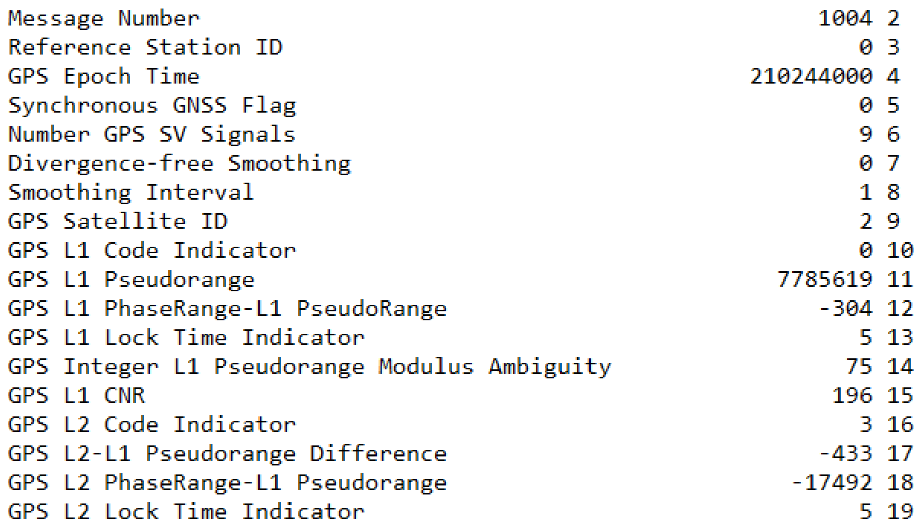
```
buffer_len    packetID     satcode      result       frame
      1024        None         ---         ---         ---
       890        1019           2           4           1
       823        1019           3           4           2
       756        1019           6           4           3
       689        1019           7           4           4
       622        1019           8           4           5
       555        1019           9           4           6
       488        1019          11           4           7
       421        1019          14           4           8
       354        1019          16           4           9
       287        1019          17           4          10
       220        1019          18           4          11
       153        1019          19           4          12
        86        1019          22           4          13
        19        1019          23           4          14
        19        1019          23           0          15
```

Fig. 3.12: Example of Buffer processing: it shows how rtcm3 attributes change at each cycle. You may see that in this buffer there were only navigational messages (type 1019).



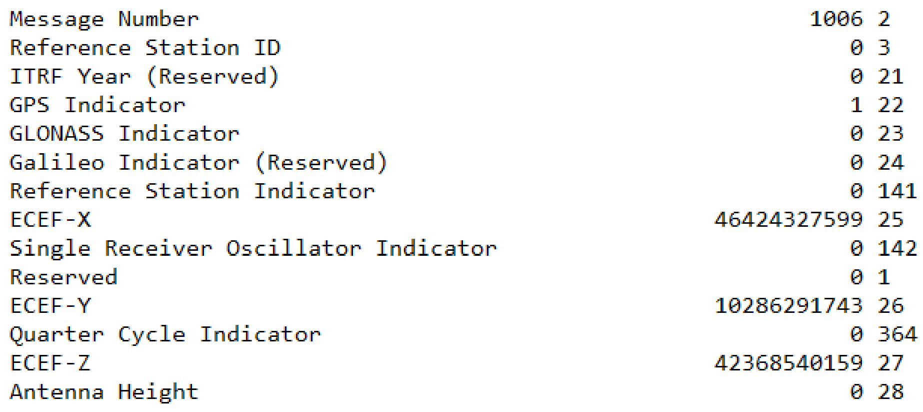
```
Message Number                                              1019 2
GPS Satellite ID                                              26 9
GPS Week Number                                             1011 76
GPS SV ACCURACY                                                2 77
GPS CODE ONLY L2                                               1 78
GPS IDOT                                                    -206 79
GPS IODE                                                      71 71
GPS t_oc                                                   13500 81
GPS a_f2                                                       0 82
GPS a_f1                                                      94 83
GPS a_f0                                                  194552 84
GPS IODC                                                      71 85
GPS C_rs                                                    2893 86
GPS DELTA n                                                12127 87
GPS M_0                                               1588643517 88
GPS C_uc                                                    2544 89
GPS Eccentricity (e)                                    30086333 90
```

Fig. 3.13: Message name, decoded value and data field number of message 1019.



```
Message Number                                                1004 2
Reference Station ID                                             0 3
GPS Epoch Time                                           210244000 4
Synchronous GNSS Flag                                            0 5
Number GPS SV Signals                                            9 6
Divergence-free Smoothing                                        0 7
Smoothing Interval                                               1 8
GPS Satellite ID                                                 2 9
GPS L1 Code Indicator                                            0 10
GPS L1 Pseudorange                                         7785619 11
GPS L1 PhaseRange-L1 PseudoRange                              -304 12
GPS L1 Lock Time Indicator                                       5 13
GPS Integer L1 Pseudorange Modulus Ambiguity                    75 14
GPS L1 CNR                                                     196 15
GPS L2 Code Indicator                                            3 16
GPS L2-L1 Pseudorange Difference                              -433 17
GPS L2 PhaseRange-L1 Pseudorange                            -17492 18
GPS L2 Lock Time Indicator                                       5 19
```

Fig. 3.14: Message name, decoded value and data field number of message 1004.



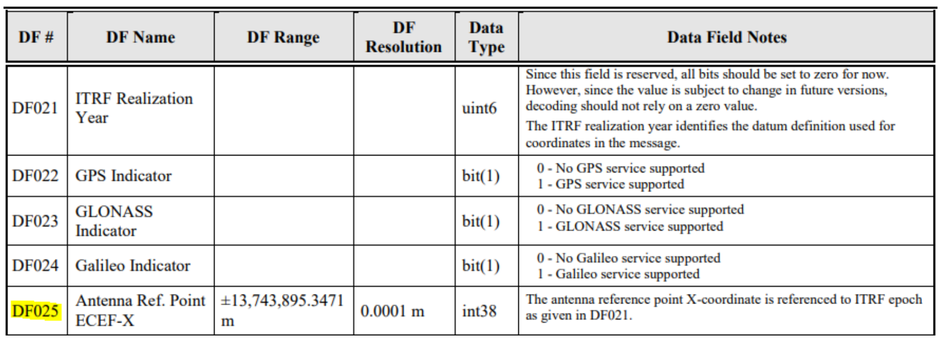
```
Message Number                                               1006 2
Reference Station ID                                            0 3
ITRF Year (Reserved)                                            0 21
GPS Indicator                                                   1 22
GLONASS Indicator                                               0 23
Galileo Indicator (Reserved)                                    0 24
Reference Station Indicator                                     0 141
ECEF-X                                                46424327599 25
Single Receiver Oscillator Indicator                            0 142
Reserved                                                        0 1
ECEF-Y                                                10286291743 26
Quarter Cycle Indicator                                         0 364
ECEF-Z                                                42368540159 27
Antenna Height                                                  0 28
```

Fig. 3.15: Message name, decoded value and data field number of message 1006.

## 3.4 The DECODER as an I/O function

### 3.4.1 From decoded values to real messages

As shown from Fig. 3.13, Fig. 3.14 and Fig. 3.15, decoded values are not really representing the corresponding field. For example, decode value of ECEF-X data filed (message 1006) is 46424327599, should represent the X approximate coordinates of the receiver in WGS84, but it clearly does not. So what are the real values of each data field? The answer is the *data field resolution*, which is specified inside the Standards manual [7], in the chapter *Data Fields*. Therefore, for each data



| DF # | DF Name | DF Range | DF Resolution | Data Type | Data Field Notes |
|---|---|---|---|---|---|
| DF021 | ITRF Realization Year | | | uint6 | Since this field is reserved, all bits should be set to zero for now. However, since the value is subject to change in future versions, decoding should not rely on a zero value. The ITRF realization year identifies the datum definition used for coordinates in the message. |
| DF022 | GPS Indicator | | | bit(1) | 0 - No GPS service supported 1 - GPS service supported |
| DF023 | GLONASS Indicator | | | bit(1) | 0 - No GLONASS service supported 1 - GLONASS service supported |
| DF024 | Galileo Indicator | | | bit(1) | 0 - No Galileo service supported 1 - Galileo service supported |
| DF025 | Antenna Ref. Point ECEF-X | ±13,743,895.3471 m | 0.0001 m | int38 | The antenna reference point X-coordinate is referenced to ITRF epoch as given in DF021. |

Fig. 3.16: Example of data fields list in the rtcm3 standard [7]. DF resolution of DF025 is 0.0001 meters.

field, the decoded quantity must be multiplied by its corresponding data field resolution. For example, the actual value of DF025 is

$$46424327599 * 0.0001m = 4642432.7599m$$

The operation described above is made into new class files, representing navigational messages, observation messages and, reference messages.

The class files mentioned have all the same scheme:

- The constructor method takes a list of all the data fields decoded inside the `Buffer`
- Get the index of the first data field (`index_start`)
- Get the index of the last data field, (`index_end`)
- Get a variable `Message` with data fields from `index_ start` to `index_end`
- Assign new attributes:

```
for field in Message:
    if field['df_number'] == X: #(df number, e.g.:2)
    self.name_dfX = field['value'] * DFresolution
```

In 3.4.1.1 , 3.4.1.2 and 3.4.1.3 are shown the three classes definitions. Then, in section 3.5, is illustrated how those classes are utilized. For now, it is enough to know objects of those classes, are the Decoder outputs.

#### 3.4.1.1 Navigational message class definition

Listing 3.4: class *Navigational*

```
class Navigational:
    def __init__(self,fields):
        index_start=fields.index({'type':'UINT',
                                  'name':'Message Number',
                                  'df_number':2,
                                  'bitlength':12,
                                  'value':1019})
        self.name=fields[index_start]['value']
        self.index_end= index_start + 31
        self.MESSAGE=fields[index_start:self.index_end]
        for field in self.MESSAGE:
            if field['df_number']==9:   #satellite ID
                self.Id = field['value']
            if self.Id<=9:
                self.satcode=("G"+"0"+str(self.Id))
            else:
                self.satcode=("G"+str(self.Id))
            elif field['df_number']==76:   #satellite week
                self.GPSweek = field['value'] # * 1 week
            elif field['df_number']==77:   #GPS SV Accuracy [meters]
                self.SVa = field['value']  # N/A
            elif field['df_number']==78:   #GPS CODE ON L2
                self.codes= field['value'] * 1
            elif field['df_number']==79:   #GPS IDOT
                self.idot = field['value'] * (2**(-43))* np.pi
            elif field['df_number']==71:   #GPS IODE
                self.IODE = field['value'] *1
            elif field['df_number']==81:   #GPS t_oc
                self.Toc = field['value'] *(2**4)
            elif field['df_number']==82:   # GPS a_f2
                self.a2= field['value'] * (2 **(-55))
            elif field['df_number']==83:   # GPS a_f1
                self.a1= field['value'] * (2 **(-43))
            elif field['df_number']==84:   # GPS a_f0
                self.a0= field['value'] * (2**(-31))
            elif field['df_number']==85:   # GPS IODC
                self.IODC= field['value'] * 1
            elif field['df_number']==86:   # GPS Crs
```

```
                self.Crs= field['value'] * (2**(-5))
            elif field['df_number']==87:   # GPS DELTA n
                self.Deltan= field['value'] * (2**(-43)) * np.pi
            elif field['df_number']==88:   # GPS M_0
                self.M0= field['value'] * (2 ** (-31))*np.pi
            elif field['df_number']==89:   # GPS C_uc
                self.Cuc= field['value'] * (2**(-29))
            elif field['df_number']==90:   # GPS Eccentricity (e)
                self.e= field['value'] * (2**(-33))
            elif field['df_number']==91:   # GPS C_us'
                self.Cus= field['value'] * (2**(-29))
            elif field['df_number']==92:   # GPS A^1/2'
                self.sqrtA= field['value']  * (2**(-19))
            elif field['df_number']==93:   # GPS toe
                self.TOE= field['value']  * (2**(4))
            elif field['df_number']==94:   # GPS C_ic
                self.Cic= field['value'] * (2 **(-29))
            elif field['df_number']==95:   # GPS OMEGA_0
                self.OMEGA = field['value'] *(2 ** (-31))*np.pi
            elif field['df_number']==96:   # GPS C_is
                self.Cis= field['value'] * (2**(-29))
            elif field['df_number']==97:   # GPS i_0
                self.i0= field['value'] * (2 ** (-31)) * np.pi
            elif field['df_number']==98:   # GPS Crc
                self.Crc= field['value']* (2 **(-5))
            elif field['df_number']==99:# GPS w (Argument of Perigee)
                self.omega0= field['value'] * (2 ** (-31))   * np.pi
            elif field['df_number']==100:# GPS OMEGADOT
                self.OMEGA_DOT= field['value']* (2 ** (-43)) * np.pi
            elif field['df_number']==101:   # GPS t_GD
                self.TGD= field['value'] * (2**(-31))
            elif field['df_number']==102:   # GPS SV HEALT
                self.SVh= field['value'] *1
            elif field['df_number']==103:   # GPS L2 P data flag
                self.L2Pdata= field['value']* 1
            elif field['df_number']==137:   # GPS
                self.HFI = field['value'] * 1
```

#### 3.4.1.2 Observations message class definition

Listing 3.5: class *Observations*

```
class Observations:
    def __init__(self,fields):
        self.header=7 # n data fields in message header
        self.each_satellite=12 # n data fields for eac satellite
        self.Ambiguity=[]
        self.pseudorangeRaw=[]
        self.L1phaserangeL1pseudorangeRaw = []
        self.satID= []
        self.satcode=[]

        index_start=fields.index({'type': 'UINT',
                                  'name': 'Message Number',
                                  'df_number': 2,
                                  'bitlength': 12,
                                  'value': 1004})
        index_nsat = index_start + 4
        index_sow=index_start + 2
        self.name=fields[index_start]['value']
        self.nsat=fields[index_nsat]['value']
        self.sow=fields[index_sow]['value'] / 1000
        length=self.nsat * self.each_satellite + self.header
        self.index_end=index_start + length
        self.MESSAGE= fields[index_start : self.index_end]
        for field in self.MESSAGE:
            if field['df_number']==14:
                self.Ambiguity.append(field['value'])
            elif field['df_number']==11:
                self.pseudorangeRaw.append(field['value'])
            elif field['df_number']==9:
                self.satID.append(field['value'])
            elif field['df_number'] == 12:
                self.L1phaserangeL1pseudorangeRaw.append(
                                                            field['value'])
            for satID in self.satID:
                if satID<=9:
                    self.satcode.append("G"+"0"+str(satID))
                else:
                    self.satcode.append("G"+str(satID))

    def GL1Pseudorange(self):
        pseudorangeRaw = np.array(self.pseudorangeRaw)
        Ambiguity = np.array(self.Ambiguity)
        GPSL1Pseudorange = ((pseudorangeRaw * 0.02)
                            % 299792.458) + (Ambiguity *299792.458)
        return(GPSL1Pseudorange)

    def GL1Phaserange(self):
        L1phaserangeL1pseudorangeRaw = (
                        np.array(self.L1phaserangeL1pseudorangeRaw))
        GL1Phaserange = (self.GL1Pseudorange()
                        + L1phaserangeL1pseudorangeRaw * 0.0005)
        return(GL1Phaserange)
```

The class *Observations*, has something more than what is in class *Navigational* and class *ReferencePoint*: it has the methods *GL1Pseudorange* and *GL1Phaserange* that calculate respectively the pseudo-range observation on the L1 carrier wave and, the phase-range observation on the L1 carrier wave, both introduced in section 2.3). The operations performed inside the two methods are well specified in the RTCM3 Standard [7].

### 3.4.1.3 Reference point message class definition

Listing 3.6: class *ReferencePoint*

```
class ReferencePoint:
def __init__(self, fields):
    index_start=fields.index({'type': 'UINT',
                              'name': 'Message Number',
                              'df_number': 2,
                              'bitlength': 12,
                              'value': 1006})
    self.name=fields[index_start]['value']
    self.index_end= index_start + 14
    self.MESSAGE=fields[index_start:self.index_end]
    for field in self.MESSAGE:
        if field['df_number']== 2:
            self.MessageNumber = field['value']
        elif field['df_number']== 3:
            self.ReferenceStationID = field['value']
        elif field['df_number']== 22:
            self.GPS_Indicator = field['value']
        elif field['df_number']== 141:
            self.ReferenceStationIndicator = field['value']
        elif field['df_number']== 25:
            self.ECEF_X = field['value'] * 0.0001
        elif field['df_number']== 26:
            self.ECEF_Y = field['value'] * 0.0001
        elif field['df_number']== 27:
            self.ECEF_Z = field['value'] * 0.0001
        elif field['df_number']== 28:
            self.AntennaHeight = field['value'] * 0.0001
```

## 3.5 Decoder Outputs

For each Buffer acquired, the Decoder should give as output:

- A list of navigational messages, if any
- A list of observation messages, if any
- A list of reference point messages, if any

The function which is in charge of this is the *CalculateBufferOutput*, shown in code 3.7

```
Output = CalculateBufferOutput(Input)
```

It takes as input a Python list containing all the data fields decoded in the `Buffer`. This information is inside the attribute `All_Data_Fields` of the `rtcm3` object [1]. Each element of the list is, as usual, a Python dictionary with keys: 'type', 'name', 'df_number', 'bitlength' and 'value'. It gives as output a Python dictionary, with keys: 1019, 1004 and 1006, and, as values, list of objects of class, respectively *Navigational*, *Observations*, *ReferencePoint*, as shown in Fig. 3.17. The entire decoding procedure can be wrapped in a single function taking as input, the entire byte-array coming through the TCP socket, and giving, as output, the result of *CalculateBufferOutput*.

[1]This attribute wasn't in the decoder when I took it from GitHub.

Listing 3.7: The *CalculateBufferOutput* function

```python
def CalculateBufferOutput(All_data_fields):
    """Gives final output from an entire buffer"""
    startnav={'type': 'UINT', 'name': 'Message Number',
        'df_number': 2, 'bitlength': 12, 'value': 1019}
    startobs={'type': 'UINT', 'name': 'Message Number',
          'df_number': 2, 'bitlength': 12, 'value': 1004}
    startref={'type': 'UINT', 'name': 'Message Number',
          'df_number': 2, 'bitlength': 12, 'value': 1006}
    n1019 = All_data_fields.count(startnav)
    n1004 = All_data_fields.count(startobs)
    n1006 = All_data_fields.count(startref)
    navigational_list=[0]*n1019
    observations_list=[0]*n1004
    reference_list=[0]*n1006
    copy=All_data_fields.copy()
    for i in range(0, n1019):
        navigational_list[i]=messages.Navigational(copy)
        copy=copy[navigational_list[i].index_end :]

    copy=All_data_fields.copy()
    for i in range (0,n1004):
        observations_list[i]=messages.Observations(copy)
        copy=copy[observations_list[i].index_end :]

    copy=All_data_fields.copy()
    for i in range (0,n1006):
        reference_list[i]=messages.ReferencePoint(copy)
        copy=copy[reference_list[i].index_end :]
    result={}
    if n1019 != 0:
        result[1019] = navigational_list
    if n1004 != 0:
        result[1004] = observations_list
    if n1006 != 0:
        result[1006] = reference_list
    return (result)
```

### 3.5.1 Final DECODER function

Listing 3.8: The *DECODER* function

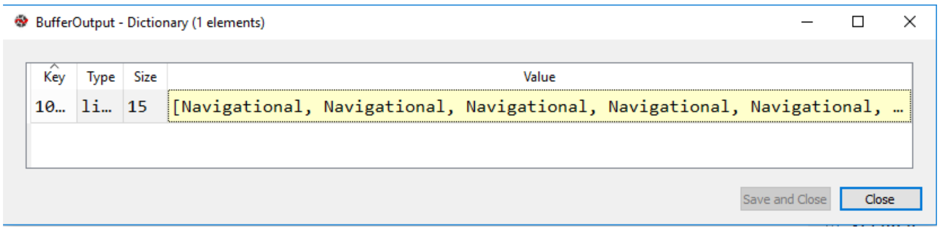
```
def DECODER(Buffer):
    rtcm3=RTCM3.RTCM3()
    new_data = bytearray(Buffer)
    rtcm3.add_data(data=new_data)
    result = rtcm3.process_data()
    while result != 0:  # RESULT = 0 means Need More
        if result == 3:
            print('Undecoded Data')
        elif result== 4:
            rtcm3.dump2()
        else:
            raise NameError('result',result , 'not identified')
        result=rtcm3.process_data()
    BufferOutput = CalculateBufferOutput(rtcm3.Data_Fields_Record)
    return BufferOutput
```

It's worth noting that:

- At line 10, the attribute `Data_Fields_Record` must be updated. That is done by the method *dump2* in *RTCM3.py* class file, A.1.2
- The function *CalculateBufferOutput* is called out of the cycle, when all the Buffer has been processed.
- *CalculateBufferOutput* takes as input the `rtcm3` attribute containing all the decoded data fields (`rtcm3.Data_Fields_Record`)

### 3.5.2 Output examples

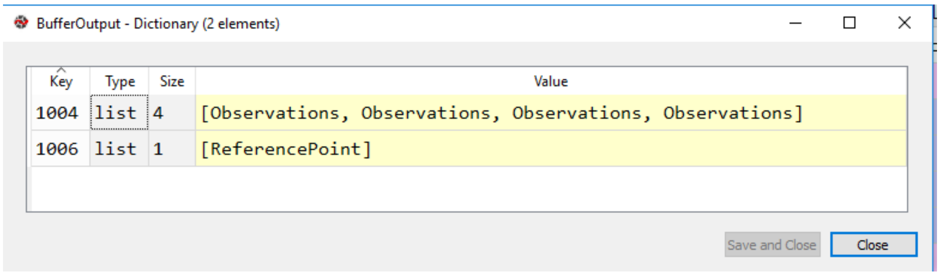
The following figures are representing the contents of the *BufferOutput* variable. The images are screen-shots of the Spyder Variable Explorer.



BufferOutput - Dictionary (1 elements)

| Key | Type | Size | Value |
|---|---|---|---|
| 10… | li… | 15 | [Navigational, Navigational, Navigational, Navigational, Navigational, … |

(a) *BufferOutput* after first Buffer Processing.

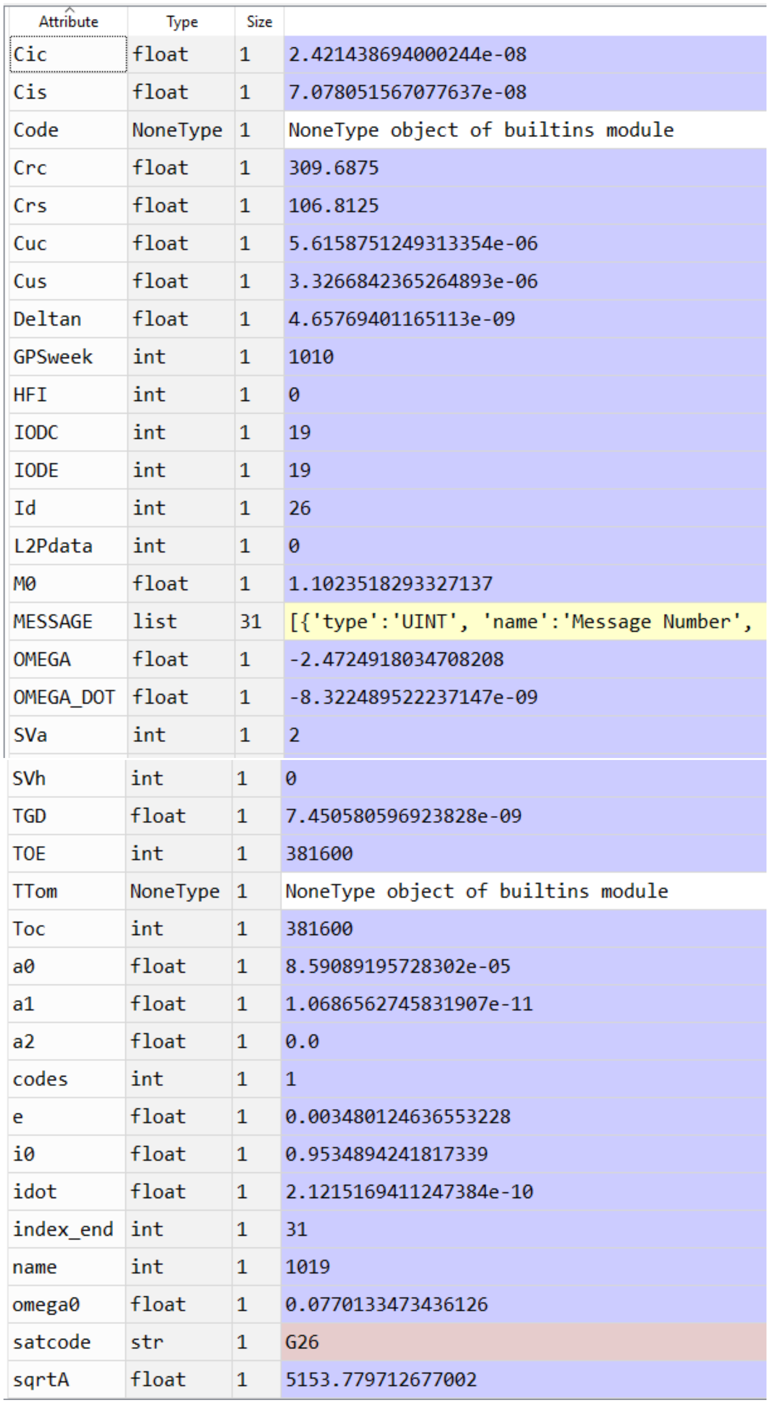


BufferOutput - Dictionary (2 elements)

| Key | Type | Size | Value |
|---|---|---|---|
| 1004 | list | 4 | [Observations, Observations, Observations, Observations] |
| 1006 | list | 1 | [ReferencePoint] |

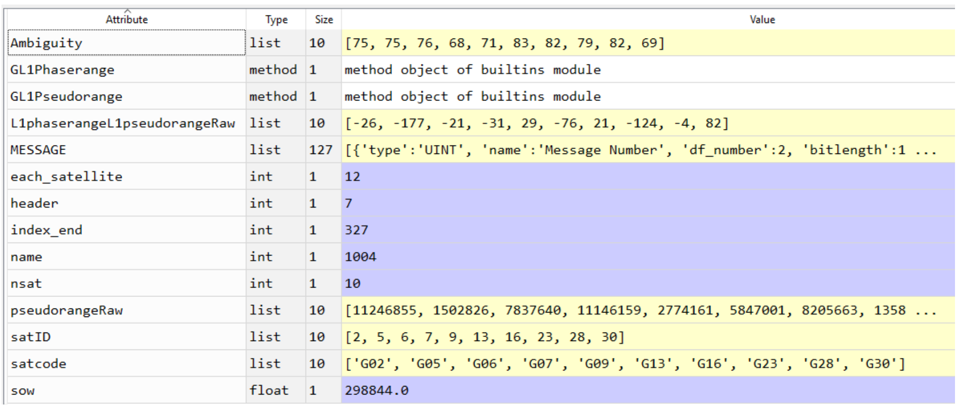
(b) *BufferOutput* after second Buffer Processing.

Fig. 3.17: Example of two outputs of the function *BufferOutput*. It is possible to see that they are dictionaries. Keys are on the left and, values, are lists (of objects), on the right with a yellow background. In this specific example, there were: 15 navigational messages in the first buffer (a); 4 observations messages and one reference point message in the second buffer (b).



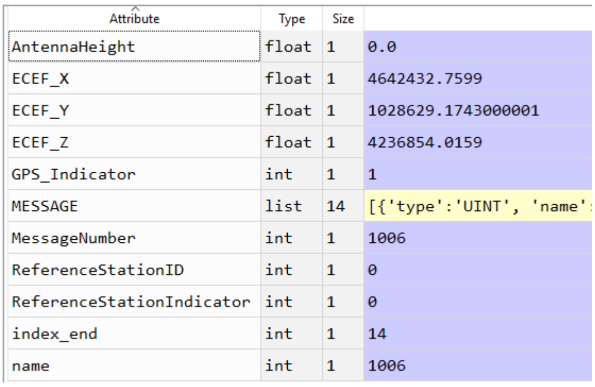
| Attribute | Type | Size | |
|---|---|---|---|
| Cic | float | 1 | 2.421438694000244e-08 |
| Cis | float | 1 | 7.078051567077637e-08 |
| Code | NoneType | 1 | NoneType object of builtins module |
| Crc | float | 1 | 309.6875 |
| Crs | float | 1 | 106.8125 |
| Cuc | float | 1 | 5.6158751249313354e-06 |
| Cus | float | 1 | 3.3266842365264893e-06 |
| Deltan | float | 1 | 4.65769401165113e-09 |
| GPSweek | int | 1 | 1010 |
| HFI | int | 1 | 0 |
| IODC | int | 1 | 19 |
| IODE | int | 1 | 19 |
| Id | int | 1 | 26 |
| L2Pdata | int | 1 | 0 |
| M0 | float | 1 | 1.1023518293327137 |
| MESSAGE | list | 31 | [{'type':'UINT', 'name':'Message Number', |
| OMEGA | float | 1 | -2.4724918034708208 |
| OMEGA_DOT | float | 1 | -8.322489522237147e-09 |
| SVa | int | 1 | 2 |
| SVh | int | 1 | 0 |
| TGD | float | 1 | 7.450580596923828e-09 |
| TOE | int | 1 | 381600 |
| TTom | NoneType | 1 | NoneType object of builtins module |
| Toc | int | 1 | 381600 |
| a0 | float | 1 | 8.59089195728302e-05 |
| a1 | float | 1 | 1.0686562745831907e-11 |
| a2 | float | 1 | 0.0 |
| codes | int | 1 | 1 |
| e | float | 1 | 0.003480124636553228 |
| i0 | float | 1 | 0.9534894241817339 |
| idot | float | 1 | 2.1215169411247384e-10 |
| index_end | int | 1 | 31 |
| name | int | 1 | 1019 |
| omega0 | float | 1 | 0.0770133473436126 |
| satcode | str | 1 | G26 |
| sqrtA | float | 1 | 5153.779712677002 |

Fig. 3.18: *Navigational* message content. Note that contains the ephemerides for orbits calculation. The satellite from which this particular message come is the G26, as you may see from the *satcode* attribute.



| Attribute | Type | Size | Value |
|---|---|---|---|
| Ambiguity | list | 10 | [75, 75, 76, 68, 71, 83, 82, 79, 82, 69] |
| GL1Phaserange | method | 1 | method object of builtins module |
| GL1Pseudorange | method | 1 | method object of builtins module |
| L1phaserangeL1pseudorangeRaw | list | 10 | [-26, -177, -21, -31, 29, -76, 21, -124, -4, 82] |
| MESSAGE | list | 127 | [{'type':'UINT', 'name':'Message Number', 'df_number':2, 'bitlength':1 ... |
| each_satellite | int | 1 | 12 |
| header | int | 1 | 7 |
| index_end | int | 1 | 327 |
| name | int | 1 | 1004 |
| nsat | int | 1 | 10 |
| pseudorangeRaw | list | 10 | [11246855, 1502826, 7837640, 11146159, 2774161, 5847001, 8205663, 1358 ... |
| satID | list | 10 | [2, 5, 6, 7, 9, 13, 16, 23, 28, 30] |
| satcode | list | 10 | ['G02', 'G05', 'G06', 'G07', 'G09', 'G13', 'G16', 'G23', 'G28', 'G30'] |
| sow | float | 1 | 298844.0 |

Fig. 3.19: *Observations* message content. Note that some information, like *satcode* or *Ambiguity* for example, are represented by lists because they are one for each satellite observed. The satellite observed are in the *satcode* attribute.



| Attribute | Type | Size | |
|---|---|---|---|
| AntennaHeight | float | 1 | 0.0 |
| ECEF_X | float | 1 | 4642432.7599 |
| ECEF_Y | float | 1 | 1028629.1743000001 |
| ECEF_Z | float | 1 | 4236854.0159 |
| GPS_Indicator | int | 1 | 1 |
| MESSAGE | list | 14 | [{'type':'UINT', 'name': |
| MessageNumber | int | 1 | 1006 |
| ReferenceStationID | int | 1 | 0 |
| ReferenceStationIndicator | int | 1 | 0 |
| index_end | int | 1 | 14 |
| name | int | 1 | 1006 |

Fig. 3.20: *ReferencePoint* message content. In this application we will only use the three attributes ECEFX, ECEFY, ECEFZ to get the approximate receiver position

# Chapter 4

# Real-time applications

The Decoder now gives in real-time, all the information we need to:

1. Estimate receiver position with single point positioning
2. Apply the so-called ”variometric” approach to estimate 3D displacements between two consecutive epochs

The next chapters illustrates the entire work flow starting from observations, passing trough single point positioning and arriving finally to the variometric approach.

It’s important to highlight that the RTCM standard does not transmit the coefficients for applying the Klobuchar ionospheric model in real-time. Therefore, in this tool implementation, the ionospheric correction terms are considered null. However, while that affects the coordinates estimation single-point positioning (4.1), it is not affecting much solutions of the variometric approach, because of the time difference observation equations, that reduces the ionospheric term, as is described in 4.2.

## 4.1 Single-point positioning

The receiver position is estimated with the single point positioning approach, introduced in 2.4.1 on page 12. Remember that, to apply the least square methods, which, for our purpose, can be seen as a function (yet it is a Python function in the tool of this thesis work) we need first, for each satellite, the following information:

- Satellite coordinates $X_S$, $Y_S$, $Z_S$
- The term $b_R^S = \tilde{\rho} + -cdt^S + I + T$
- The unit vector $\tilde{\mathbf{e}}_{\mathbf{R}}^{\mathbf{S}}$, from the approximate receiver to the satellite.

Receiver position estimation is managed by the tool with the interaction of two new data types. They are the *navpack* class and the *recsat* class.

The class *navpack* is essentially the same as the *Navigational* class. That means that an object of class *navpack*, contains all the attributes contained in the navigational message, hence all the ephemerides, summarized in Tab. 2.1 on page 6, with which is possible to get the satellite coordinates, by means of a well-known algorithm, which can be found in any GPS reference book [2].

### 4.1.1 The Receiver-Satellite (recsat) class

The class *recsat* represents all the information we need to apply the least square method. In fact it contains essentially: the satellite position $X_S$, $Y_S$, $Z_S$ (in WGS84); the approximate receiver position $\tilde{X_R}$,

$\tilde{Y}_R$, $\tilde{Z}_R$ (in WGS84); the geometric distance $\tilde{\rho}$; the unit vector $\tilde{\mathbf{e}}_{\mathbf{R}}^{\mathbf{S}}$; the satellite clock term correction $dt^s$; the atmosphere correction terms $I_R^S$ and $T_R^S$; the pseudo-range and phase-range observations. In other words we can say that ***recsat* contains all the known terms of the observation equation, referred to one single epoch**.

Listing 4.1: recsat class definition

```
class RECSAT:
    def __init__(self):
    self.sat=POINT() # satllite position
    self.rec=POINT() # receiver position
    self.satcode="X00" # satellite code
    self.sow=0.0       # second of week
    self.tof=0.075     # time of flight
    self.range=0.0     # geometric distance
    self.clock=0.0     # satellite clock correction
    self.CosX=0.0      # e_x versor
    self.CosY=0.0      # e_y
    self.CosZ=0.0      # e_z
    self.CosE=0.0      #     versor in nord est up
    self.CosN=0.0      #
    self.CosUp=0.0     #
    self.Elevation=0.0 # satellite elevation
    self.Azimuth=0.0   # satellite azimuth
    self.Trp=0.0       # troposphere correction
    self.Flag=0        # (check flag)
    self.Iono=0.0      # ionosphere correction
    self.obs=0.0       # pseudo-range L1
    self.obsf1=0.0     # phase-range L1
    self.obsf2=0.0
    self.obsf3=0.0
    self.var1=0.0
```

You may observe that the receiver and satellite position are represented by object of class *Point*, which, for the sake of completeness, is shown here below.

Listing 4.2: The *Point* class

```
class POINT:
    def __init__(self):
        self.X=0.0
        self.Y=0.0
        self.Z=0.0
        self.Fideg=0.0
        self.Lamdeg=0.0
        self.h=0.0
        self.Firad=0.0
        self.Lamrad=0.0
```

As seen from the *recsat* class definition (code 4.1) a new *recsat* object is initialized with all attributes set to zero (except for the time of flight *tof*). Therefore it's worth to say that *recsat* attributes are "filled" in two different step:

1. From the *Observations* message, after decoding. In this phase, the attributes that are filled in are: pseudo-range, phase-range, sow, and satellite code which are identified by `obs, obsf1, sow` and `satcode`, respectively. This operation is performed by the function *makeRECSAT*, shown in code 4.3. Note that, since in one incoming *Buffer*, there might be more *Observations* messages (type 1004), the function takes as input a list of messages. Therefore, the notation used in the code and that is used from now on here, is:

    - `recsat`, indicates one single receiver satellite object
    - `RECSAT`, indicates a list of `recsat`, in which each `recsat` refer to a different satellite; all the `recsat` in the list refers to the same epoch (`sow`)

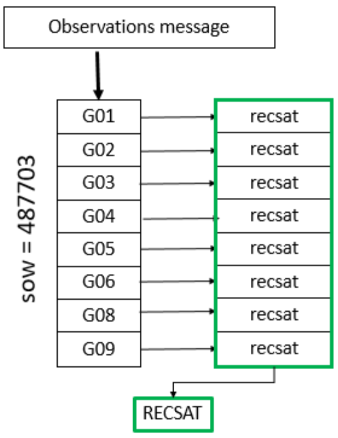


Fig. 4.1: How a list of *recsat* is created from an *Observations* message. If an observation message contains observations to 8 satellites, then the function *makeRECSAT*, produces a list of 8 *recsat*, (indicated in the code with *RECSAT*, with capital letters). Each observation in one message is referred to the same second of week sow.

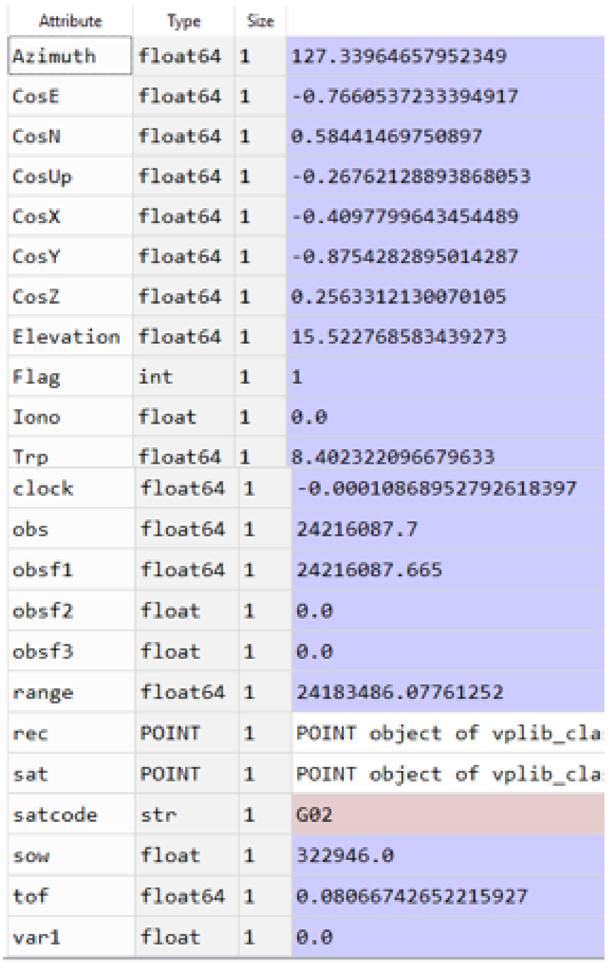
- `RRECSAT`, indicates a list of `RECSAT`; it come in handy when there is more than one observation message in the buffer (`len(BufferOutput[1004])>1`)

2. After the orbits computation, all the other attributes are assigned

In Fig. 4.1 is shown a simple sketch representing how a list of recsat is created from one observations message. In Fig. 4.2 is shown an example of "full" *recsat*, with the description of the main attributes.

Listing 4.3: The *makeRECSAT* function

```
def makeRECSAT(Observations_list):
"""
This function takes a  Observations objects and returns a list of
recsat objects (RECSAT) for each observation messsage in input"""
    RRECSAT = [0] * len(Observations_list)
    for message in Observations_list:
        recsat =[0]*message.nsat
        for i in range(0, message.nsat):
    # create new recsat
            recsat[i]=RECSAT()
    # assign satcode
            recsat[i].satcode=message.satcode[i]
    # assing sow
            recsat[i].sow=message.sow
    # assign psudorange
            recsat[i].obs = message.GL1Pseudorange()[i]
    # assign phaserange
            recsat[i].obsf1= message.GL1Phaserange()[i]
        idx = Observations_list.index(messagetype1004)
        RRECSAT[idx]=recsat
    return(RRECSAT)
```



| Attribute | Type | Size | |
|---|---|---|---|
| Azimuth | float64 | 1 | 127.33964657952349 |
| CosE | float64 | 1 | -0.7660537233394917 |
| CosN | float64 | 1 | 0.58441469750897 |
| CosUp | float64 | 1 | -0.26762128893868053 |
| CosX | float64 | 1 | -0.4097799643454489 |
| CosY | float64 | 1 | -0.8754282895014287 |
| CosZ | float64 | 1 | 0.2563312130070105 |
| Elevation | float64 | 1 | 15.522768583439273 |
| Flag | int | 1 | 1 |
| Iono | float | 1 | 0.0 |
| Trp | float64 | 1 | 8.402322096679633 |
| clock | float64 | 1 | -0.00010868952792618397 |
| obs | float64 | 1 | 24216087.7 |
| obsf1 | float64 | 1 | 24216087.665 |
| obsf2 | float | 1 | 0.0 |
| obsf3 | float | 1 | 0.0 |
| range | float64 | 1 | 24183486.07761252 |
| rec | POINT | 1 | POINT object of vplib_cla |
| sat | POINT | 1 | POINT object of vplib_cla |
| satcode | str | 1 | G02 |
| sow | float | 1 | 322946.0 |
| tof | float64 | 1 | 0.08066742652215927 |
| var1 | float | 1 | 0.0 |

Fig. 4.2: Example of a "full" *recsat* (after orbits computation). The unit vector **e** is in CosX, CosY CosZ; The atmosphere terms $I_R^S$ and $T_R^S$ are in Iono and Trp, respectively; the satellite clock term $dt_S$ is in clock; the pseudo-range and phase-range are in obs and obsf1, respectively; the distance term $\tilde{\rho}$ is in range. Furthermore you may notice that, this particular recsat is referring to satellite G02 at the epoch 322946.0 sow (satcode and sow attributes, respectively).

### 4.1.2 Satellite matching for orbits computation

It may happen (actually is very frequent), that there is more satellite sending the Navigational message, than observed satellites. Suppose, for example that, at the decoder output, there are 16 Navigational messages (15 *navpack*), and an *Observations* message with 10 satellites. Then, the function *makeRECSAT*, will produce a list of 10 *recsat*. It may seem trivial but, it is worth to point out that all the *navpack* that does not have any sat code in any *recsat*, should be excluded. In other terms, we need to compute orbits, just for the observed satellites.

Situation with more navigational messages than observed satellites

```
>> for recsat in RECSAT: print(recsat.satcode, end = '\t')
G02     G06     G10     G12     G13     G15     G17     G19     G24
G25     G32

>> for nav in NAVPACK: print(nav.satcode, end='\t')
G05     G10     G06     G25     G19     G02     G15     G24     G08
G07     G32     G28     G12     G17     G20     G13
```

### 4.1.3 Navigational message updating

In GPS System, the navigational messages are updated every two hours; therefore the tool, which is thought to work continuously, should deal with that. The attribute *TOE* (Time Of Emission), of each *navpack*, tells us when the navigational message is sent. The function that manages *navpack* updating is the *update_NAVPACK* function. It essentially, when a new *Buffer* arrives, checks the sat codes of the new navigational message and:

- if there are new satellites adds them to the navigational message list

- if there are the same satellites, substitute them in the navigational message list

For the sake of completeness, the function is showed in A.1.5. Is interesting to see how this was managed using Python sets instead of lists. In fact sets, compared to lists, are much more efficient for iterating operation such as for cycle.

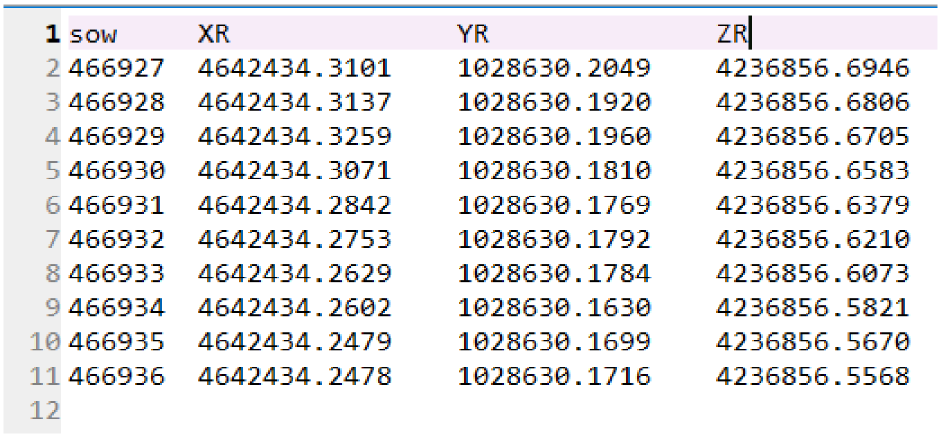
### 4.1.4 Final solutions

The operations consisting in computing orbits and checking the satellite matching, is done inside the function *SinglePositioning*, which takes a list of *recsat*, `RECSAT` (one recsat for each satellite observed, and all referred to the same epoch), a list of *navpack*, `NAVPACK` (already updated), and the approximate receiver position which are inside the ReferencePoint message at the attributes ECEF-X, ECEF-Y and ECEF-Z.

Example of *SinglePositioning* calling

```
sol ,fullRECSAT , conf=SinglePositioning (RECSAT,NAVPACK,X,Y,Z)
```

As you may notice, the function gives as output the solution (*sol*), a list with "full" *recsat* objects, a set of configuration parameters that are essentially telling us what kind of corrections models (troposphere and/or ionosphere) have been used. Here, in Fig. 4.3, an example of the solution file that the tool can provide (in real-time).



```
sow     XR              YR              ZR
466927  4642434.3101    1028630.2049    4236856.6946
466928  4642434.3137    1028630.1920    4236856.6806
466929  4642434.3259    1028630.1960    4236856.6705
466930  4642434.3071    1028630.1810    4236856.6583
466931  4642434.2842    1028630.1769    4236856.6379
466932  4642434.2753    1028630.1792    4236856.6210
466933  4642434.2629    1028630.1784    4236856.6073
466934  4642434.2602    1028630.1630    4236856.5821
466935  4642434.2479    1028630.1699    4236856.5670
466936  4642434.2478    1028630.1716    4236856.5568
```

Fig. 4.3: Example of single positioning solution file. It was made by the acquisition of 10 Buffer. X, Y and Z are expressed in meters. sow is the second of week of the observation.

## 4.2 Real-time variometric approach

The so-called "variometric" approach is an established technique for the estimation of co-seismic displacements with a stand-alone GPS receiver[6]. The approach is based on the *time single difference* of carrier phase observations collected at a high rate ($\geq$ 1 Hz) using a stand-alone receiver, and on standard GPS broadcast ephemerides. Since, in our application, the permanent station M0SE works at 1 Hz frequency, the estimated displacements are essentially velocities [6].

### 4.2.1 The (simplified) variometric model

As I mentioned in 2.3.2, the phase observation of a receiver $R$ to one satellite $S$ at epoch $t$ is:

$$L_R^S(t) = \rho_R^S(t) + c(\delta t_R(t) - \delta t^S(t)) + \lambda N_R^S(t) - I_R^S(t) + T_R^S(t) \quad (4.1)$$

where, as usual, $\rho$ is the geometric range receiver-satellite; $\lambda$ is the carrier phase wavelength; $c$ is the speed of light; $\delta t_R$ and $\delta t^S$ are the receiver and the satellite clock errors, respectively; $T_R^S$ and $I_R^S$ are the tropospheric and ionospheric delays along the path; $N_R^S$ is the initial phase ambiguity. If we differentiate (4.1) in the time between two consecutive epochs (t, t+1), and supposing that no cycle slips occur, we get the *time single difference observation*

$$\begin{aligned}\Delta L_R^S(t,t+1) =& \Delta\rho_R^S(t,t+1) + c(\Delta\delta t_R(t,t+1) - \Delta\delta t^S(t,t+1)) \\ &+\Delta T_R^S(t,t+1) + \Delta I_R^S(t+1,t)\end{aligned} \tag{4.2}$$

It's worth to see now that, since the path of the signal at time $t$ and the path of the signal at time $t+1$, are very similar, then the terms $\Delta T_R^S(t,t+1)$ and $\Delta I_R^S(t+1,t)$, are very little. That's way the fact that the RTCM does not transmit Klobuchar coefficient is not affecting the solutions of this "simplified" variometric model.

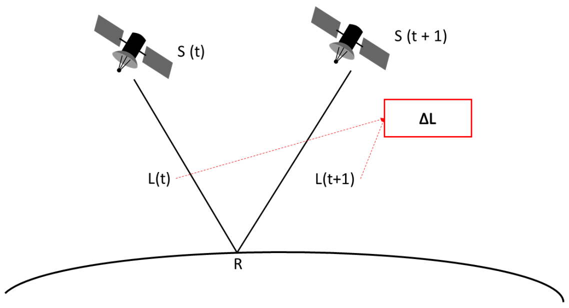


Fig. 4.4

If we hypothesize that the receiver is fixed in an Earth Centred Earth Fixed (ECEF) reference frame, the term $\Delta\rho_R^S(t,t+1)$, depends upon the change of the geometric range due to the satellite orbital motion and the Earth's rotation $[\Delta\rho_R^S(t, t+1)]_{OR}$, so that, [6]

$$\Delta\rho_R^S(t, t+1) = [\Delta\rho_R^S(t, t+1)]_{OR} \tag{4.3}$$

On the other hand, if we hypothesize that the receiver underwent a 3d displacement $\boldsymbol{\Delta}\xi_R(t, t+1)$, during the interval $t, t+1$, the term $\Delta\rho_R^S(t, t+1)$ also includes the effect of $\boldsymbol{\Delta}\xi_R$ projected along the line-of-sight, which is approximately the same for the two consecutive epochs if the observation rate is $\geq 1$ Hz. Therefore, if we indicate with $\mathbf{e}$ the unit vector from the receiver to the satellite at epoch $t$, we can write [6]:

$$\begin{aligned}\Delta\rho_R^S(t, t+1) =& [\Delta\rho_R^S(t, t+1)]_{OR} + [\Delta\rho_R^S(t, t+1)]_D \\ =& [\Delta\rho_R^S(t, t+1)]_{OR} + \mathbf{e}^T\Delta\xi_R(t, t+1)\end{aligned} \tag{4.4}$$

Since the tool works with a 1 Hz frequency, the displacement $\boldsymbol{\Delta}\xi_R(t, t+1)$, is essentially a velocity vector.

The term $\Delta I_R^S(t, t+1)$ represents the variation of the tropospheric delay, during the interval $(t, t+1)$, and is known by modelling it with the *Klobuchar* model.

Therefore, substituting equation (4.4) in (4.2), and omitting the time dependencies, we obtain:

$$\Delta L_R^S = \left(\mathbf{e}^T\boldsymbol{\Delta}\xi_R + c\Delta\delta t_R\right) + \left([\Delta\rho_R^S]_{OR} - c\Delta\delta t^S + \Delta T_R^S + \Delta I_R^S\right) \tag{4.5}$$

where $\left(\mathbf{e}^T\boldsymbol{\Delta}\xi_R + c\Delta\delta t_R\right)$ contains the four unknown parameters (the three components of displacement and the receiver clock error), and $\left([\Delta\rho_R^S]_{OR} - c\Delta\delta t^S + \Delta T_R^S + \Delta I_R^S\right)$ is the known term than can be computed on the basis of the correction models and the navigational message. Writing the equation (4.5) for each satellite in sight (at least 4), we obtain a system that can be solved with the least square estimation. Thus, we obtain, for two consecutive epochs an estimation of the 3D velocities.

## 4.2.2 Tool implementation

### 4.2.2.1 Preliminary considerations

To exploit the variometric approach we need a routine able to keep, for each satellite S, two consecutive "full" receiver-satellite objects (recsat). Consecutive means with the consecutive second of week (`sow`) attribute. As we saw in 4.1.4. Remember that full *recsat* is an output of the *SinglePositioning* function.

Example of two consecutive recsat, referring to the same satellite

```
>> recsat1.sow
558405.0
>> recsat2.sow
558406.0
>> recsat1.satcode
"G02"
>> recsat2.satcode
"G02"
```

Once we have two consecutive *recsat*, is possible to get all the known terms of the equation 4.5. For example, the observation $\Delta L_R^S(t+1,t)$ is `recsat2.obsf1 - recsat1.obsf1`; The satellite clock term $\Delta\delta t^S(t+1,t)$ is `recsat2.clock - recsat1.clock`. It's worth to spend few words on the range term $\Delta\rho_R^S(t+1,t)$: in fact, while the range at time $t$, (first epoch), is just `recsat1.range`, at time $t+1$, is the distance between the receiver position at time t and the satellite position at time t+1. So, for example, considering the function *GEORANGE* (A.1.3), which calculates the distance between two points, than, $\Delta\rho_R^S(t+1,t)$ would be `GEORANGE(recsat1.rec, recsat2.sat) - recsat1.range`.

#### 4.2.2.2 The ***StartVariometric*** function

The example above is made with one single *recsat*, hence, for one single satellite. Obviously, to implement the least square method we need more than 4 observations, hence the tool manage list of *recsat*, named `RECSAT` (with capital letters), where each element refers to one specific satellite. The function that provides two consecutive `RECSAT` and starts the variometric approach is the *StartVariometric* function.

The *StartVariometric* function takes as input: `RRECSAT` which contain the `RECSAT` of the present buffer; `RECSAT_4_VARIO`, which contains the `RECSAT` from previous buffer; and `conf` which, as usual, identifies if are utilized correction models for the atmosphere errors. It gives as output an (updated) `RECSAT_4_VARIO`, which contains the last `RECSAT` of the present buffer, and which will be an input of the function in the next cycle.

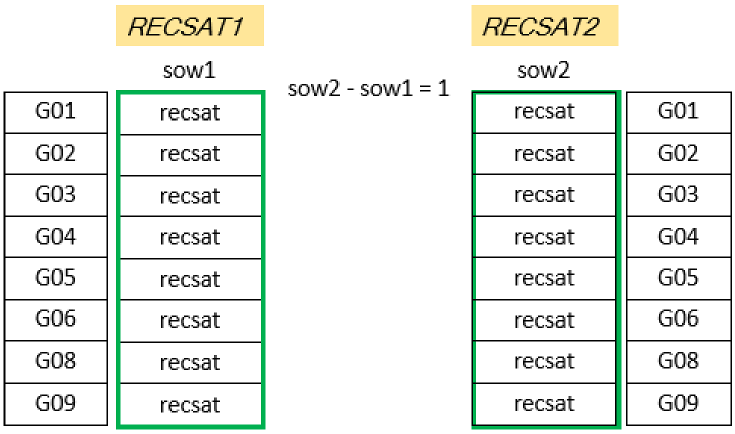


Fig. 4.5: Simple sketch of two RECSAT with two consecutive epochs. Obviously, each recsat in RECSAT1 must have the corresponding recsat with the same satellite in RECSAT2.

Listing 4.4: The *StartVariometric* function

```
def StartVariometric(RRECSAT,RECSAT_4_VARIO, conf):
    """ StartVariometric prepares two consecutive recsat and
        launch the variometric approach (VARIOMETRIC function)"""

    n_of_1004 = len(RRECSAT)
    if n_of_1004==1:
        RRECSAT_4_VARIO.append(RRECSAT[0])
        print("Only one recsat in this frame")
    elif n_of_1004 > 1:
        for k in range(0,n_of_1004):
            RRECSAT_4_VARIO.append(RRECSAT[k])
    for RECSAT in RRECSAT_4_VARIO:
        print('sow:', RECSAT[0].sow)

    if len(RRECSAT_4_VARIO)>1:
        RECSAT12 = []
        for k in range(0,len(RRECSAT_4_VARIO)):
            if k>1:
                RECSAT12.pop(0)
            RECSAT12.append(RRECSAT_4_VARIO[k])
            if len(RECSAT12)==2:
                tests.test_sow_in_RECSAT_bis(RECSAT12[0])
                tests.test_sow_in_RECSAT_bis(RECSAT12[1])
                sow1 = RECSAT12[0][0].sow
                sow2 = RECSAT12[1][0].sow
                if sow2 - sow1 == 1.0:
                    RECSAT1 = RECSAT12[0]
                    RECSAT2 = RECSAT12[1]
                    varsol = VARIOMETRIC(RECSAT1,RECSAT2,conf)
            else:
                print 'No consecutive epochs' )
        for k in range(0, len(RRECSAT_4_VARIO)-1):
            RRECSAT_4_VARIO.pop(0)
    return RECSAT_4_VARIO
```

#### 4.2.2.3 Variometric equations

Line 29 of the *StartVariometric* function (code 4.4) calls `VARIOMETRIC`. This function takes the two consecutive `RECSAT`, and gives the solution of the variometric approach. As shown in 4.5 variometric equation for one single satellite is

$$\Delta L_R^S = \left(\mathbf{e}^T \boldsymbol{\Delta}\xi_R + c\Delta\delta t_R\right) + \left([\Delta\rho_R^S]_{OR} - c\Delta\delta t^S + \Delta T_R^S + \Delta I_R^S\right) \quad (4.6)$$

Therefore, for m satellites, the system

$$\boldsymbol{\Delta}\mathbf{L} = \begin{bmatrix}\mathbf{E_R} & \mathbf{i}\end{bmatrix}\begin{bmatrix}\boldsymbol{\Delta}\xi_R \\ c\Delta\delta t_R\end{bmatrix} + \mathbf{b} \quad (4.7)$$

where:

$$\boldsymbol{\Delta}\mathbf{L} = \begin{bmatrix}\Delta L^1 \\ \Delta L^2 \\ \vdots \\ \Delta L^m\end{bmatrix} \qquad \begin{bmatrix}\mathbf{E_R} & \mathbf{i}\end{bmatrix} = \begin{bmatrix}\ldots\mathbf{e}_R^1\ldots & 1 \\ \ldots\mathbf{e}_R^2\ldots & 1 \\ \vdots & \vdots \\ \ldots\mathbf{e}_R^m\ldots & 1\end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix}b_R^1 \\ b_R^2 \\ \vdots \\ b_R^m\end{bmatrix}$$

To represent the system in Python the `RECSAT1` and `RECSAT2` are gathered together into a new data structure called `varin`. So for example, if the two `RECSAT` observe 6 satellites, than

```
varin[0] = RECSAT1[0]
varin[1] = RECSAT1[1]
varin[2] = RECSAT1[2]
.
.
```

```
.
varin[6] = RECSAT2[0]
varin[7] = RECSAT2[1]
varin[8] = RECSAT2[2]
.
.
.
varin[11]=RECSAT2[5]
```

`VARIOMETRIC` function builds the arrays of the system as following:

- Design matrix A

  ```
  A=np.zeros((len(varin),4),dtype=float)
  for i in range(len(varin)):
      A[i,0]=varin[i].CosX1
      A[i,1]=varin[i].CosY1
      A[i,2]=varin[i].CosZ1
      A[i,3]=1.0
  ```

- Observation vector Y

  ```
  Y=np.zeros((len(varin),1),dtype=float)
  for i in range(len(varin)):
      Y[i]=varin[i].obs2-varin[i].obs1
  ```

- Known terms vector B

  ```
  B=np.zeros((len(varin),1),dtype=float)
  ```

```
for i in range(len(varin)):
    B[i]=((varin[i].range2-varin[i].clock2*vpc.CLIGHT)-
  (varin[i].range1-varin[i].clock1*vpc.CLIGHT))
    if conf.T >0:
    B[i]=B[i]+varin[i].Trp2-varin[i].Trp1
```

The three arrays above represent the functional model of the Least Square. The stochastic model is represented by the diagonal matrix Q, with the square of the Up component of the unit vector **e** in each term of the diagonal.

- Stochastic model Q

```
Q=np.zeros((len(varin),len(varin)),dtype=float)
for i in range(len(varin)):
    Q[i][i]=1
    if conf.W==1:
        Q[i][i]=varin[i].CosUp2**2
```

The systems can now be solved with the least square method implemented as a Python function.

## 4.2.3 Results

The tool provides in real-time csv files (comma separated variable) with the velocities along the directions East, North and Up, and the corresponding plot which updates itself in real time. An example is shown in Fig. 4.6. A preliminary quality check of the solution has

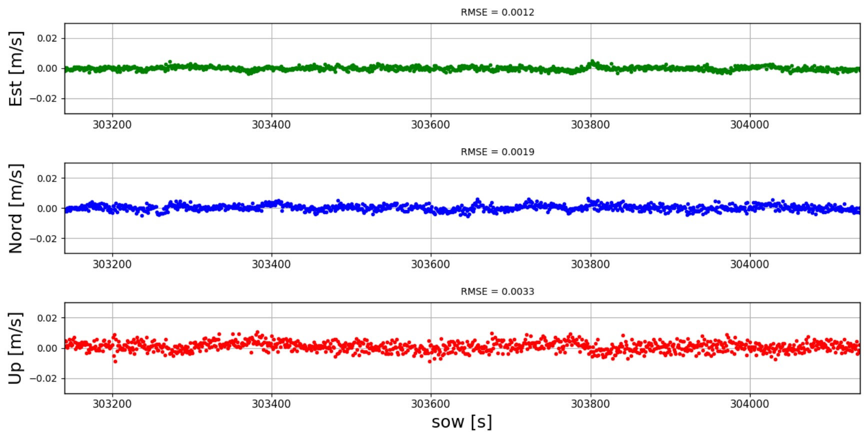


Fig. 4.6: Example of the plot which the tool provides in real-time. The plot shows the results of the variometric approach. Sow(GPS second of week) on the x-axis and the velocities along the directions East, North and Up on the y-axis.

been carried out with respect post processed solutions obtained by the `VADASE` software [6]. The quality check shows that **RTCM2PVT** gives fully compliant solutions for the variometric approach, with some small differences due to the simplified model without Klobuchar corrections used in **RTCM2PVT**. The assessment of the sensitivity of **RTCM2PVT**'s real time velocities estimation is at the level of 2 mm/s for the horizontal components, and 5 mm/s for the vertical component.

## 4.3 Final algorithm for real-time applications

For the sake of clarity, is shown here the complete procedure performed by the **RTCM2PVT** tool.

```
Result: Single positioning and variometric approach
initialization;
RECSAT_4_VARIO = [];
while receiving data do
    Decode data with the DECODER ;
    if there are navigational message then
        make NVPACK;
        update NAVPACK
    end
    if there are observation messages then
        make RECSAT
    end
    if There are reference point messages then
        get approximate X Y Z;
        make single positioning SinglePositioning();
        RECSAT ← orbits;
        start variometric approach with StartVariometric();
        RECSAT_4_VARIO ← last RECSAT for next cycle
    end
end
```

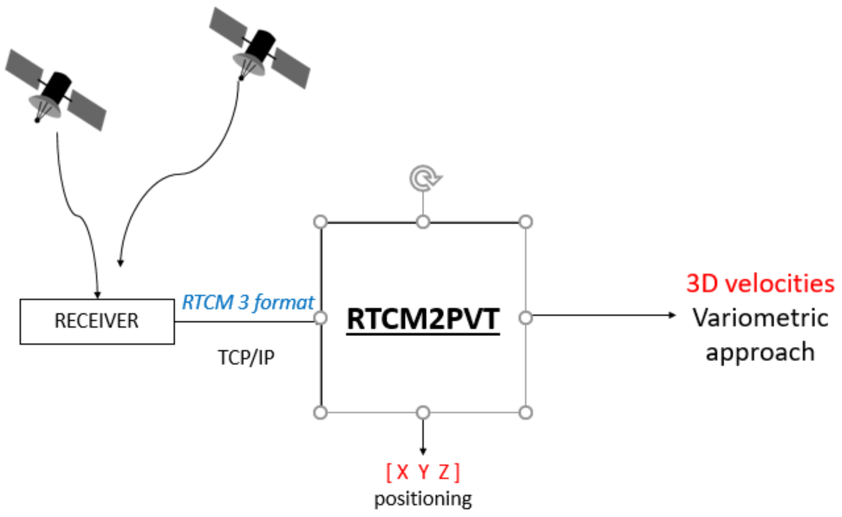


Fig. 4.7: The receiver sends navigational messages and observations with the RTCM3 format via a TCP/IP transport protocol. The tool gives as estimates the receiver position X, Y, Z in single positioning, fill the Receiver-Satellites (recsat), and estimates the receiver 3D velocities two consecutive epochs. Everything works in real-time.

# Chapter 5

# Conclusions

The objective of this thesis work was to implement a tool able to perform real-time operations with a stand-alone receiver and the broadcast GPS ephemerides available in real-time.

The main task was decoding data with the version 3 of the RTCM (Radio Technical Commission for Maritime Services). For testing I used data sent by the permanent station M0SE (Rome, Italy), but eventually it can be used with any receiver able to send data with this type of format.

Once data was decoded, was possible to conduct operations such as single point positioning and the so-called "variometric" approach, used to estimate 3D velocities. Both the methods were already implemented as Python routines, but, there was the primary need of decoded data, and secondly, to put those routines into a robust program. Therefore, this thesis work consisted mainly in, decoding RTCM data, and secondly, making everything working inside the tool **RTCM2PVT** in real-time. I must give credits to the Geodesy and Geomatic division of the Sapienza University of Rome for giving me the entire library with all the function used in this work. In particular, the functions I used are: `SinglePositioning`, `COMPUTE_ORBITS` to compute the satellite orbits; `LS` which apply the least square estimation method; the `VARIOMETRIC` function, which applies the veriometric approach; the `RECSAT` and `NAVPACK` classes.

The starting point, as usual in these kinds of operations, was looking online for something already done, and likely, I found an RTCM3 decoder on GitHub at `https://github.com/jcmb` (unlikely there isn't the name of the author), whom I express my best thanks. Unfortunately, as always, at the first trials, a bunch of errors came up! First of all, it was in version two of Python and so, was re-written in Python 3. Secondly, to fix the errors, there was the need to understand all the dependencies among all the files (one main file and two modules with classes and functions). Well, after that everything was translated in Python 3, and the flow execution was transparent, there was still something to fix. For example, it did not manage the messages with repeating data fields (satellite specific portion) and didn't do the correct conversion for data fields with the `int` type.

Once the decoder was correctly working, it was the time to make working the implemented routines developed by the Geodesy and Geomatics divison. At the end we achieved the target we intended to reach: make the so-called "variometric" approach in real-time. The final **RTCM2PVT** gives the 3D velocities Fig. 4.6, and, the receiver coordinates Fig. 4.3 in real-time. It gives them in external file, so that is possible to do futher analysis in post-processing. The assessment of the sensitivity of **RTCM2PVT**'s real time velocities estimation is at the level of 2 mm/s for the horizontal components, and 5 mm/s for the vertical component.

However, this is just the first version of the tool, many improvements could be made. First of all the complete variometric approach can be implemented in the tool, considering also dual frequency observations. Then it would be nice to build a GUI (Graphical User Interface) to interact easier with it. In order to widen the use of the tool it would be interesting to apply it on observations coming from Android based smartphones. This paves the way to innovative real-time big data transportation applications, such as vehicles precise telemetry, safety systems, optimization of fleet management, contriubuting to V2V and V2I applications.

# Appendix A

## A.1 Python routines

### A.1.1 appendToFields method

Method of the class RTCM3. It is necessary to decode messages 1004 which have repeating data fields, for each observed satellite.

```
def appendToFields(self,packet_data):
"""Append To Fields for Repetition in Messages """
    #55 :position of the first bit of df "number of satellite"
    # 7 : number of data fields in the message header
    # 4 : position of the data field "number of sattellite"
    # bitValue is a function that convers bits to decimal
    # nsatlen: lenght in bit of the df "number of satellite"
    nsatlen = self.commands[self.packet_ID].fields[4]['bitlength']
    NSat=bitValue(makeBitArray(packet_data), 55, nsatlen)
    toAppend=self.commands[self.packet_ID].fields[7:]
    for i in range(0, NSat-1):
        for k in toAppend:
            self.commands[self.packet_ID].fields.append(k.copy())
```

### A.1.2 dump2 method

Method of the RTCM3 class. Is used to record decoded data fields inside each Buffer.

```
def dump2(self):
""" record decoded data fields"""
    if self.packet_ID in self.commands:
        for field in self.commands[self.packet_ID].fields:
        self.Data_Fields_Record.append(field.copy())
        # eventually, if you wish, print decoded value
```

### A.1.3 GEORANGE function

It is an ancillary function used to calculate geometric distance between two objects of class *Point* (see code 4.2)

```
def GEORANGE(POINT1, POINT2):
""" Geometric distance between two points"""
georange=sqrt((POINT1.X-POINT2.X)**2
+(POINT1.Y-POINT2.Y)**2
+(POINT1.Z-POINT2.Z)**2)

return georange
```

### A.1.4 makeNAVPACK function

This function copies all the attributes in navigational message inside the new data structure navpack, which will be an input of the *SinglePositioning* function. *makeNAVPACK* is a function of the `ProcessData_fun` module.

```
def makeNAVPACK(navigational_list):
    """ makes NAVPACK from Navigational message """

    NAVPACK=[0]*len(navigational_list)
    for i in range(0, len(navigational_list)):
        NAVPACK[i]=vpc.NAVPACK()
        NAVPACK[i].Id = navigational_list[i].Id
        NAVPACK[i].Crs = navigational_list[i].Crs
        NAVPACK[i].Crc = navigational_list[i].Crc
        NAVPACK[i].Cuc = navigational_list[i].Cuc
        NAVPACK[i].Cus = navigational_list[i].Cus
        NAVPACK[i].Cic = navigational_list[i].Cic
        NAVPACK[i].Cis = navigational_list[i].Cis
        NAVPACK[i].Deltan = navigational_list[i].Deltan
        NAVPACK[i].M0 = navigational_list[i].M0
        NAVPACK[i].IODE = navigational_list[i].IODE
        NAVPACK[i].e = navigational_list[i].e
        NAVPACK[i].sqrtA = navigational_list[i].sqrtA
        NAVPACK[i].TOE = navigational_list[i].TOE
        NAVPACK[i].OMEGA = navigational_list[i].OMEGA
        NAVPACK[i].SVa = navigational_list[i].SVa
        NAVPACK[i].SVh = navigational_list[i].SVh
        NAVPACK[i].TGD = navigational_list[i].TGD
        NAVPACK[i].IODC = navigational_list[i].IODC
        NAVPACK[i].TTom = navigational_list[i].TTom
        NAVPACK[i].HFI = navigational_list[i].HFI
        NAVPACK[i].i0 = navigational_list[i].i0
        NAVPACK[i].omega0 = navigational_list[i].omega0
        NAVPACK[i].OMEGA_DOT = navigational_list[i].OMEGA_DOT
        NAVPACK[i].idot = navigational_list[i].idot
        NAVPACK[i].codes = navigational_list[i].codes
        NAVPACK[i].GPSweek = navigational_list[i].GPSweek
        NAVPACK[i].L2Pdata = navigational_list[i].L2Pdata
        NAVPACK[i].Toc = navigational_list[i].Toc
        NAVPACK[i].a0 = navigational_list[i].a0
        NAVPACK[i].a1 = navigational_list[i].a1
        NAVPACK[i].a2 = navigational_list[i].a2
        NAVPACK[i].satcode = navigational_list[i].satcode
    return NAVPACK
```

### A.1.5 update_NAVPACK function

This function is used to update the navigational messages. It is in the `ProcessData_fun` module.

Listing A.1: The *update_NAVPACK* function

```
def update_NAVPACK(OLD_NAVPACK, NEW_NAVPACK):
"""  Updates Navigational messages
    OLD_NAVPACK is the list of navpack out of the DECODER at time k
    NEW_NAVPACK is the list of navpack out of the DECODER at time k+1"""
# from list to set
    NEW_NAVPACK_set = set(NEW_NAVPACK)
    OLD_NAVPACK_set = set(OLD_NAVPACK)
# prepare sets for satcodes
    satCodesNEW_NAVPACK = set()
    satCodesOLD_NAVPACK = set()
# satcodes arrived in new NAVPACK
    for navpack in NEW_NAVPACK_set:
        satCodesNEW_NAVPACK.add(navpack.satcode)
# satcodes arrived in old NAVPACK
    for navpack in OLD_NAVPACK_set:
        satCodesOLD_NAVPACK.add(navpack.satcode)
# find new satellites with sets difference
    new_arrived = satCodesNEW_NAVPACK - satCodesOLD_NAVPACK
# find same satellites with sets intersecion
    same_arrived = satCodesNEW_NAVPACK & satCodesOLD_NAVPACK
# if there are new satellites add them to OLD_NAVPACK set.
    if new_arrived:
        for satcode in new_arrived:
            for navpack in NEW_NAVPACK_set:
                if satcode == navpack.satcode:
                    OLD_NAVPACK_set.add(navpack)
# if there are the same satellites substitute them:
```

```python
# make a set of navpack to remove
    if same_arrived:
        navpackToRemove = set()
        navpackToAdd = set()
        for satcode in same_arrived:
            for navpack in OLD_NAVPACK_set:
                if satcode == navpack.satcode:
                    navpackToRemove.add(navpack)
# make a set of navpack to add
            for navpack in NEW_NAVPACK_set:
                if satcode == navpack.satcode:
                    navpackToAdd.add(navpack)
# substitute navpack with same satcodes
# only if the TOE is different
            for new_navpack in navpackToAdd:
                for old_navpack in navpackToRemove:
                    if new_navpack.satcode == old_navpack.satcode:
                        if new_navpack.TOE != old_navpack.TOE:
                            print("NEW TOE")
# update navpack with sets operations:
# - difference ;  | union
        OLD_NAVPACK_set = (OLD_NAVPACK_set
                        - navpackToRemove) | navpackToAdd
# make a copy of updated navpacks
    UPDATED_NAVPACK_set = OLD_NAVPACK_set.copy()
# satcodes in updated navpacks
    SatCodes_UPDATED = set()
    for navpack in UPDATED_NAVPACK_set:
        SatCodes_UPDATED.add(navpack.satcode)
# go back to list
    UPDATED_NAVPACK = list(UPDATED_NAVPACK_set)
    return(UPDATED_NAVPACK, same_arrived, new_arrived)
```

# Bibliography

[1] Equazioni di osservazione. Slides of the University course Geomatics and Geographic Information Systems held by Professor Mattia Giovanni Crespi, University of Rome La Sapienza. 10

[2] Ludovico Biagi. *I fondamentali del GPS*, chapter 3, Il sistema GPS. Geomatic Workbooks, 2009. ix, 5, 6, 56

[3] Ludovico Biagi. *I fondamentali del GPS*, chapter 4, Le osservazioni, la propagazione e i disturbi atmosferici. Geomatic Workbooks, 2009. 10

[4] Ludovico Biagi. *I fondamentali del GPS*, chapter 5, Posizionamento Assoluto. Geomatic Workbooks, 2009. 11, 12, 13

[5] Ludovico Biagi. *I fondamentali del GPS*, chapter 6, Posizionamento Relativo. Geomatic Workbooks, 2009. 14, 15

[6] A. Mazzoni G.Colosimo, M. Crespi. Real-time gps seismology with a stand -alone receiver: A preliminary feasibility demonstration. *Journal of Geophysical Research Atmospheres*, November 2011. 2, 64, 66, 74

[7] Radio Technical Commission for Maritime Services, 1611 N. Kent St., Suite 605 Arlington, Virginia 22209-2143, U.S.A. *RTCM STANDARD 10403.3 - DIFFERENTIAL GNSS SERVICES*, October 2016. vii, viii, 18, 41, 46

[8] Lambert Wanninger. Introduction to network rtk. International Association of Geodesy, June 2008. 14

[9] Ross N. Williams. A painless guide to crc error detection algorithms. 19