



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

Grau en Enginyeria de Vehicles Aeroespacials

Treball de fi de grau

Study of differential GPS system for UAVs

Document content : ANNEX

Delivery date: 22/06/2016

Author: Oriol Trujillo Martí

Director: Antoni Barlabé Dalmau

Codirector: Manuel Soria Guerrero

Contents

List of Figures	iii
List of Tables	iv
A. GPS and DGPS concept	1
A.1 GPS overview	1
A.1.1 Position determination	1
A.1.2 Dilution Of Precision (DOP)	3
A.1.3 Reference coordinate systems	3
A.1.4 Segments	5
A.1.5 GPS satellite constellation	6
A.1.6 Ephemerides	8
A.1.7 Ranging Codes	8
A.2 Errors in Satellite Navigation	8
A.2.1 Satellite Clock Errors	9
A.2.2 Ephemeris Errors	9
A.2.3 Relativistic Effects	10
A.2.4 Atmospheric Effects	10
A.2.5 Receiver Noise and Resolution	11
A.2.6 Multipath Effects	11
A.2.7 Hardware Bias Errors	12
A.2.8 Error Comparison/Balance/Budget/.....	12
A.3 Augmentation Systems: DGPS	13
B. Market research	16
C. Navigation Message tables	17
C.1 Subframe 1	17
C.2 Subframe 2 and 3	20
D. Protocol messages' structures	21
D.1 NMEA Standard Messages	21
D.1.1 GGA: Global Positioning System fix data	21
D.1.2 GRS: GNSS Range Residuals	23
D.1.3 GSA: GNSS DOP and Active Satellites	23
D.1.4 TXT: Text Transmission	24

D.2	UBX	25
D.2.1	AID (0x0B)	26
D.2.2	NAV (0x01)	29
E.	Functions description	33
E.1	Main	33
E.2	Get Functions	35
E.3	Timing	46
E.4	Correction	49
E.5	Positioning	51
E.6	Export	53
E.7	Utils	54
F.	Main Structures	59
F.1	GNSSmessages	59
F.2	GNSSparameters	60
F.3	BaseStation	60
F.4	rover	60
F.5	SV	60
G.	Configuration files	61
G.1	Base Station	61
G.2	Rover	62
H.	Implemented code	64
H.1	DGPS.m	68
H.2	Include	73
H.2.1	messageDecoder.m	73
H.2.2	trackSV	78
H.2.3	correction	80
H.2.4	Export	91
H.2.5	getFunctions	95
H.2.7	timing	130
H.2.8	utils	137
I.	Bibliography	144

List of Figures

1. basic principle of GNSS, image from [2] 2
2. a) Good satellites' geometry b) Bad satellites geometry. Image from [2] 3
3. GPS segments, image from [3] 5
4. GPS constellation, image from [4] 6
5. Relative postions between user and satellite, image from [3] 7

List of Tables

1.	Ellipsoidal Earth model WGS 84	4
2.	Satellite and Physical constants	7
3.	GPS Standard & PPST contributions to UERE	13
4.	Only GPS vs. LADGPS typical contributions to UERE	15
5.	Raw capable GPS receivers	16
6.	Satellite Clock & Health Data, according to [9]	17
7.	DHI Navigation Message, according to [9]	18
8.	URA, according to [9]	19
9.	Satellite Ephemeris Data: Parameters, according to [9]	20
10.	NMEA Talker ID, according to [11]	21
11.	GGA: GPS fix data message structure, according to [11]	22
12.	GNSS Range Residuals message structure, according to [11]	23
13.	GNSS DOP & Active Satellites message structure, according to [11]	24
14.	Text Transmission message structure, according to [11]	25
15.	UBX variable types message structure, according to [11]	26
16.	AID-EPH message structure, according to [11]	27
17.	GPS Health, UTC & ionosphere parameters message structure, according to [11]	28
18.	Dilution Of Precision message structure, according to [11]	29
19.	Position Solution in ECEF message structure, according to [11]	30
20.	Navigation Solution Information message structure, according to [11]	32
21.	init.m description	33
22.	DGPS.m description	34
23.	messageDecoder.m description	34
24.	trackSV.m description	35
25.	generateSV.m description	36
26.	getActiveSV.m description	36
27.	getAID_EPH.m description	36
28.	getColor.m description	37
29.	getDistance.m description	37

30.	getDOP.m description	38
31.	getEphemeris.m description	38
32.	getfTOW.m description	39
33.	GNSSparameters.m description	39
34.	getGPGGA.m description	40
35.	getGPGRS.m description	40
36.	getGPGSA.m description	40
37.	getNAV_POSECEF.m description	41
38.	getNAV_POSLLH.m description	41
39.	getNAV_SOL.m description	42
40.	getNMEAfixData.m description	42
41.	getPOSECEF.m description	42
42.	getRange.m description	43
43.	getRangeResiduals.m description	43
44.	getSOL.m description	44
45.	getTimeTable.m description	44
46.	getTrueECEF.m description	45
47.	getTrueGeo.m description	45
48.	getUBFixData.m description	45
49.	addTime.m description	46
50.	assignTime.m description	46
51.	fixTime.m description	47
52.	unifyActiveSV.m description	47
53.	unifyTimings.m description	48
54.	computeCorrectionOFFSET.m description	49
55.	computeCorrectionReal.m description	50
56.	computeCorrectionVirtual.m description	50
57.	computePosition.m description	51
58.	getSVposition.m	51
59.	leastSquarePos.m description	52
60.	leastSquarePosNOdt.m description	53
61.	getTemplate.m description	53
62.	exportKML.m description	53
63.	check_t.m description	54
64.	checkOrder.m description	54
65.	clean Row.m description	55
66.	closestValue.m description	55

67.	ECEF2geodeticArray.m description	56
68.	geodetic2ECEFarray.m description	56
69.	hex2decimal.m description	56
70.	hex2str.m description	57
71.	twosComp2dec.m description	57
72.	utc2sec.m description	58
73.	Structure example: rover.posECEF	59

A. GPS and DGPS concept

It is not the goal of this annex to introduce GPS to the reader but to recall several aspects that are briefly described and are required for good comprehension of the report. A basic understanding of GNSS fundamentals is assumed.

A.1 GPS overview

A.1.1 Position determination

GNSS measures the time it takes for a signal transmitted by the emitter of the Space Vehicles at a known location to reach the user receiver, in what is known as Time Of Arrival (TOA) ranging.

By multiplying the travel time by the speed of light the distance between emitter and receiver is obtained, a spherical surface, centred at satellite location and with the measured range as radius, of possible receiver's location is determined.

The idea is to use multiple satellites in order to spot receiver's location at the sphere's intersection, in what is known as triangulation.

If were not any source of error, 3 spheres or equations would be necessary to determine user 3D position and they would intersect at that point. Such perturbations and sources of error exist, spheres do not all intersect at a single point but a region of possible locations is determined.

Those errors and their correction techniques are detailed in section A.2, but for computing user's position it is necessary to differentiate from the errors that can be corrected without using augmentations, those errors that will disturb the signal when travelling and the delay of the receiver clock with respect to the satellites, which are considered synchronized such are high accurate atomic clocks and emit their own clock biases as explained in section A.1.6.

Both types of error cause a perturbation on the pseudorange measurement, but the first ones are corrected by error predictions and models and the second one must be determined for each receiver by adding a fourth equation.

Therefore the minimum system of equations to determine user's position is shown in equations (6.1) to (6.4), where the example has been applied for the minimum case of 4 satellites (superscripts from 1 to 4) and the user is denoted by the subscript i and the pseudoranges by P , which have to be corrected.

$$P_i^1 = \sqrt{(X_i - X_1)^2 + (Y_i - Y_1)^2 + (Z_i - Z_1)^2} + cdt_i \quad (6.1)$$

$$P_i^2 = \sqrt{(X_i - X_2)^2 + (Y_i - Y_2)^2 + (Z_i - Z_2)^2} + cdt_i \quad (6.2)$$

$$P_i^3 = \sqrt{(X_i - X_3)^2 + (Y_i - Y_3)^2 + (Z_i - Z_3)^2} + cdt_i \quad (6.3)$$

$$P_i^4 = \sqrt{(X_i - X_4)^2 + (Y_i - Y_4)^2 + (Z_i - Z_4)^2} + cdt_i \quad (6.4)$$

This system of equations cannot be solved such the sphere's do not intersect into a single point even applying some corrections. The final solution must minimize the error between each sphere and itself. To do that there are several methods, the one performed in this project is the least square method explained in report, which is commonly linearized and solved iteratively.

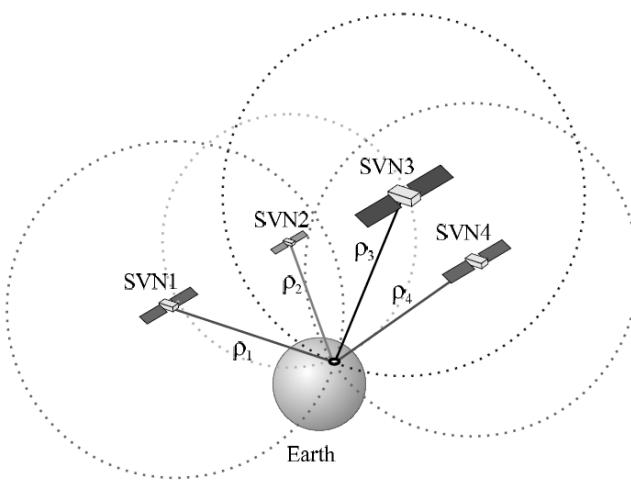


Figure 1: basic principle of GNSS, image from [2]

A.1.2 Dilution Of Precision (DOP)

In section A.1.1, it has been seen that user's position can be determined with 4 satellites reaching a final compromise solution. Such as the final solution is the best solution inside a region of possible solutions, a measure of this range of possibilities is required.

Dilution of precision is the magnitude that accounts for the effect of the geometry and the number of satellites used in the GPS solution, smaller values of DOP imply less error in the computed position or time. A large number of active Space Vehicles dispersed about the sky would give high precision results and low DOP.

In figure 2 it is shown the influence of the satellites position in the dilution of precision.

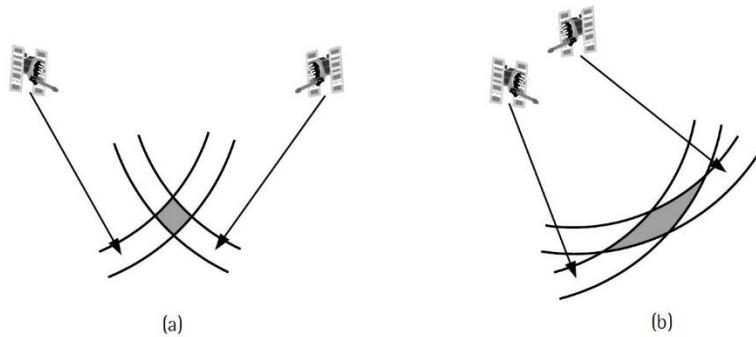


Figure 2: a) Good satellites' geometry b) Bad satellites geometry. Image from [2]

In general, vertical positioning is less accurate, and the VDOP higher, due to Earth surface blocks the signals from many satellites causing less dispersed SVs along this coordinate observed by the receiver.

Dilution of precision is not a single value but a set of statistical parameters, such as geometric DOP (GDOP), position DOP (PDOP), horizontal DOP (HDOP), vertical DOP (VDOP) and time DOP (TDOP). In the report it is shown how DOP is calculated.

As a global view, dilution of precision reflects the confidence of the method with its result. High values of dilution of precision means a higher region of possible solutions, and therefore, low confidence.

A.1.3 Reference coordinate systems

Position can be referenced to different coordinate systems, in satellite navigation two different systems are used, Earth-Centred Earth-Fixed coordinate system (ECEF) and the World Geodetic System 1984 (WGS 84). Each one is useful for different cases, and conversions from one representation to the other are continuously applied in this project.

Earth-Centred Earth-Fixed (ECEF)

This is a Cartesian coordinate system, so it is represented by 3 orthogonal axis (X, Y and Z) and, as the name indicates, it is centred at the Earth centre and it is fixed to the Earth, so the axis do not describe fixed directions in the inertial space but rotate with Earth.

This coordinate system is particularly useful to represent Space Vehicles' position, to compute ranges between satellites and receiver, to perform pseudoranges corrections as it is done in this project, etc.

The directions of the axis in the positive sense is defined as: the x-axis points the equator – prime meridian (5.31 arcseconds or 102.5m east from Greenwich meridian at Royal Observatory, Greenwich latitude) intersection and the z-axis the North Pole. Positive y-axis is chosen so as to form a right-handed coordinate system.

World Geodetic System 1984 (WGS 84)

This is a spherical coordinate system particularized for the Earth. Besides, it is also a physical model of the Earth established by the U.S. DoD in 1984, it accounts several phenomena such as gravitational irregularities and defines constants such as the speed of light.

The coordinates composing this system are latitude, longitude and height, and Earth is modelled as an ellipsoid where its minor semi-axis is perpendicular to the equatorial plane. Its parameters are represented at table 1.

<i>World geodetic System 1984 - Ellipsoidal Earth model parameters</i>	
Name	Definition
Major semi-axis a	$a = 6,378.137 \text{ km}$
Minor semi-axis b	$b = 6,356.7523142 \text{ km}$
Square Eccentricity	$e^2 = 1 - \frac{b^2}{a^2} = 0.00669437999014$
Flattening	$b = a \cdot (1 - f); \quad f = \frac{1}{298,257,223,563}$

Table 1: Ellipsoidal Earth model WGS 84

Historically heights had been referred to the geoid surface, which is geopotential surface or, in terms of least squares, the Mean Sea Level (MSL). In this project it has not been worked with MSL, but it is decoded and stored if the messages containing this parameters are activated.

A.1.4 Segments

In order to provide and control global positioning service, GPS is composed by three segments: space segment, ground-control segment and user segment.

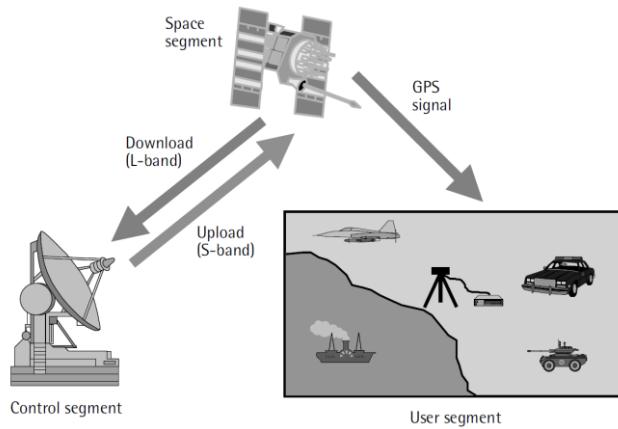


Figure 3: GPS segments, image from [3]

Space Segment

Space segment is formed by the satellite constellation, that it is described section A.1.2. GPS satellites or Space Vehicles emit PRN-coded signals, composed by two carrier frequencies (basically two sine waves), two digital codes and the navigation message that is described in Section 7.2.1. The carrier frequencies and the codes allow to make range measurements, and the navigation message provides information about satellite path and clock. This signal transmission is done in what is called the L-band as explained in section A.1.5. Space Vehicles can attend an unlimited number of receivers such they work as passive systems.

In order to make valid range measurements, because of the speed of light, Space Vehicles have highly accurate atomic clocks on board.

Satellites are equipped with several subsystems that allow to perform the following tasks:

- Grant GPS navigation.
- Control of the vehicle path and orientation, satellites emitter must point the Earth and the solar panels must point the sun.
- Detection and reporting of Earth-based radiation phenomena. This is a secondary task not related with GPS operability and it is performed by the nuclear detonation detection system (NUDET) according to [1].

Control Segment

Control segment maintains the proper operation of the Space Vehicles by controlling the orbits and monitoring their health and status. Also, has the responsibility of to keep the almanac, ephemeris and satellites' clock updated. Instructions are sent to the satellites in the S-band of frequencies.

User Segment

The user segment is integrated by receiving equipment, commonly called GPS receiver. They process the L-band signals and perform the calculations to determine user's position and velocity.

A.1.5 GPS satellite constellation

The satellite constellation consists of 24 satellites arranged in 6 orbital planes with 4 satellites per plane. Each orbital plane is inclined about 55° to the equator and equally spaced around the equator at 60° separation between adjacent orbits.

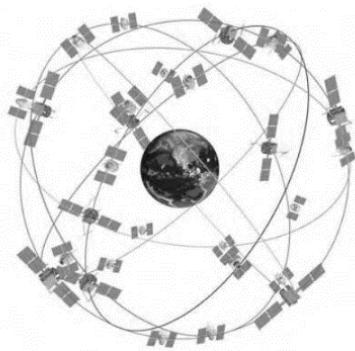


Figure 4: GPS constellation, image from [4]

The orbits are nearly circular and some of their main common parameters can be seen in table 2.

Satellite and Physical constants			
User to satellite data		GPS satellite data	
R_{MIN}	~20,000km	Radius of Earth	~6,398km
R_{MAX}	~25,593	Mean altitude above sea level	20,000km
Min Time Delay	~66ms	R_{SV}	~26,550km
Max Time Delay	~86ms	Orbital rate	2 orbits per day
-	-	Orbital speed	~3874m/s

Table 2: Satellite and Physical constants

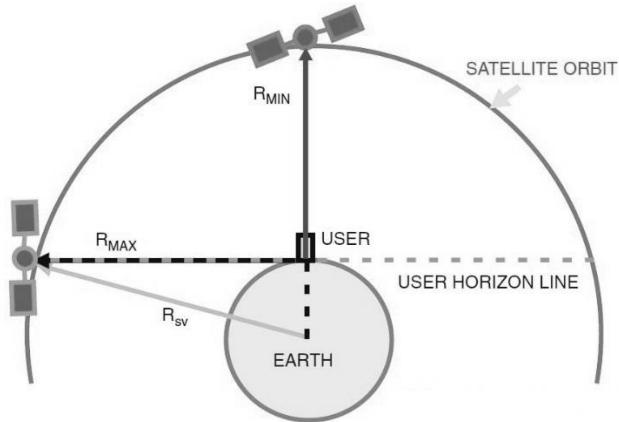


Figure 5: Relative positions between user and satellite, image from [3]

A.1.6 Ephemerides

Ephemeris, in singular, is the set of orbital parameters that allows to precisely determine the position and trajectory of a Space Vehicle. They are sent from the ground-control segment to each satellite and from each satellite to the receivers as a part of the navigation message. The whole packet of ephemeris is called ephemerides.

Ephemeris parameters are only valid for a time interval of 4 hours and are updated every 2 hours.

In the report ephemeris parameters and their collection are described more precisely.

A.1.7 Ranging Codes

As introduced previously in section A.1.1, satellites emit two different frequencies in the L-band, these are L1 (1,575.42 MHz) and L2 (1,227.6 MHz), and two different ranging codes, which are different for each Space Vehicle and has been selected because of their low cross-correlation properties, plus the navigation message. One of these codes is short and the other is long, these are referred as the coarse/acquisition or C/A and the precision or P(Y)-code, respectively.

However, for security reasons, the U.S. DoD does not allow to have access to P(Y)-code but for authorized personal, keeping the military advantage in what is referred as Selective Availability (SA) and anti-spoofing (AS), in what is known as Standard Positioning System (SPS) and Precision Positioning System (PPS).

Unauthorized users had an intentionally degraded satellite clock time until 1st May, 2000.

A.2 Errors in Satellite Navigation

Previously, it has been introduced Satellite Navigation in a very simplified way, taking the speed of light as its travelling through the vacuum, ignoring any disturbing phenomena and source of error, such as refraction, reflection, hardware effects, and so forth. In this section are presented the perturbing causes that distort the pseudorange measurement and consequently, the final positioning solution.

All phenomena explained next, contributes as delays or offsets in the pseudorange measurement. They have been classified as atmospheric, noise and interferences, multipath and hardware.

The total delay can be written as:

$$dt_D = dt_{atm} + dt_{noise\&int} + dt_{mp} + dt_{HW} \quad (6.5)$$

A.2.1 Satellite Clock Errors

Satellite atomic clocks are highly accurate, however, they are not perfect and accumulate error. In order to reduce that error as much as possible, correction second-order polynomial parameters are sent by each Space Vehicle in the navigation message. Furthermore, due to satellites' high velocities and the required level of precision, relativistic effects must be taken into account giving the following correcting equation:

$$dt_k = a_0 + a_1 \cdot (t - t_{oc}) + a_2 \cdot (t - t_{oc})^2 + \Delta t_r \quad (6.6)$$

Where t is the current time and t_{oc} is the reference time.

Applying the correction shown in equation 6.5, satellite clocks introduce an error to the pseudorange varying from 0.3m to 1m at zero age of data (ZAOD) and until 4m after 24 hours. Recall from section A.1.6 that ephemerides are updated every two hours, so the satellite clock error will be in the first rank.

A.2.2 Ephemeris Errors

As seen in section A.1.4, the control segment broadcast ephemeris data to the satellites which send them to the user. The error between the estimated satellite's position from ephemeris and the true position is a vector with typical magnitudes from 1 to 6m. The radial component of the residual error is smaller than the others, having a low effect in the pseudorange measurement (a typical value would be 0.8m).

A.2.3 Relativistic Effects

Relativistic effects are taken into account and corrected in the satellite clock correction and also is configured to compensate them by setting the clock frequency lower than the one that is wanted to be observed.

Besides, an uncorrected error is suffered caused by the space-time curvature due to the gravitational field of the Earth. The magnitude of this error can reach values of 18.7mm.

A.2.4 Atmospheric Effects

In the first approach to satellite navigation, speed of light has been considered constant taking its value of propagation in the vacuum, $c = 299,792,458$ m/s as defined within WGS 84. Which is not true, such the propagation velocity depends on the medium.

These effects can be observed in the ionosphere and troposphere layers of the atmosphere. Which the first is a dispersive medium and the latter not.

A.2.4.1 Ionospheric Effects

Ionosphere is a dispersive medium, this means that the propagation velocity is a function of the wave's frequency, lower frequencies higher delay and so on. This causes a delay on the signal's arrival and a bending on the GPS radio signal's path, which is greater for L2 frequency than L1. The latter effect can be neglected especially if the satellite elevation angle is greater than 5° , which is warranted in this project since satellites with such low elevations are rejected.

However, the change of propagation speed causes a significant range error, it speeds up carrier-phase and slows down PRN code by the same amount. The user would measure a shorter distance from the carrier phase and a longer one from the code. This way, military authorized users can correct easily ionosphere delay by comparing this measures. The common user, such in this project, must compensate this error by empirical models of the ionosphere (the most common is the Klobuchar model). As presented in the report, ionosphere correction parameters are sent as a part of the navigation message.

Generally, uncorrected ionospheric delay is of the order of 5 to 15m and after standard corrections up to 60% of the delay can be corrected.

A.2.4.2 Tropospheric Effects

Unlike the ionosphere, the troposphere is a non-dispersive medium so the propagation velocity does not depend on the frequency. So, the measure range will be longer than the real geometric range.

Tropospheric delay depends on the pressure, temperature and humidity along the signal pass through it. It can be split in two components, dry and wet. The first represents about the 90% of the delay and can be easily predicted by mathematical models, the latter is hard to predict.

Uncorrected tropospheric delay is decrease with the elevation angle being minimized at zenith (when the satellite is directly overhead). This error can be around 2.3 m at the zenith and 9.3m at 15° of elevation.

A.2.5 Receiver Noise and Resolution

GPS receivers must track the signal emitted by the Space Vehicles, by comparing the received C/A code with the pattern, which is displaced until it matches the received signal. The amount of displaced time is the measured travel time. However, both signals do not fit perfectly and a resolution error is introduced. A typical value of error is 1.2mm.

Receiver tracking loops are also affected by interferences and thermal noise. The resulting noise depends on the receiver's electronics and would vary depending on the quality of the GPS receiver. A typical average value for range error due to receiver's noise would be of the order of 0.6mm.

A.2.6 Multipath Effects

Multipath is the major source of error in the pseudorange measurement. As the own name says, it occurs when the signal reaches the receiver's antenna though different paths. These paths can be the direct line from the satellite to the receiver and reflected signals by the surrounding objects and environment. This reflected signals would interfere and distort the original signal causing range errors of the order of tens of meters, but they are highly dissipated by the electronics of the device.

A real problem with this source of error, occurs is the reflected paths arrives at the receiver's antenna before that the direct one. This may occur if the direct signal must cross an object that speeds it down as it passes through it, while the reflected signal find a faster way. For instance if the receiver is under a tree or inside a building with an open window, etc.

A.2.7 Hardware Bias Errors

A.2.7.1 Satellite Biases

Due to electronic limitations, there is a delay introduced by the satellite hardware in the signal. This vary for each carrier frequency that are imperfectly synchronized.

For the case that this projects is concerned, a single-frequency receiver (L1), besides the clock correction, another timing bias must be added, T_{GD} . This amount is contained in the word 7 of the subframe 1 of the navigation message and it is generally less than 8ns.

A.2.7.2 User's Equipment Biases

As Space Vehicles do, GPS receiver equipment also delays GPS signals as they travel through all the electronics, from antenna until the point where pseudorange and carrier-phase measurements are physically made. This delay produces biases of the order of a few millimetres, but can be estimated and reduced

A.2.8 Error Comparison/Balance/Budget/...

In order to have a basic comprehension on what the contribution of each source of error to the User Equivalent Range Error (UERE) is, in table 3 are shown some typical values that must be taken as guidelines. Final positioning error is hard to compute such is function of the UERE and the user-satellite geometry (DOP).

<i>GPS Standard and Precise Positioning Service Typical contributions to UERE</i>			
Segment Source	Error Source	Standard Error (m)	Precise Error (m)
Space/control	Broadcast clock	1.1	1.1
	L1 P(Y) – L1 C/A group delay	0.3	-
	Broadcast ephemeris	0.8	0.8
User	Ionospheric delay	7.0	0.1
	Tropospheric delay	0.2	0.2
	Receiver noise and resolution	0.1	0.1
	Multipath	0.2	0.2
Total (Root-Sum-Square)		7.1	1.4

Table 3: GPS Standard & PPST contributions to UERE

A.3 Augmentation Systems: DGPS

Augmentations are techniques to improve GPS performance. They can be satellite-based or ground-based.

In the first one, referred as Satellite-Based Augmentation System (SBAS), several ground-based stations analyse satellites' signals, supervise GNSS constellations and transmit corrections of the ephemerides or satellites' clocks to the user. The following SBAS systems are currently operative: EGNOS, WAAS, MSAS, QZSS and GAGAN; which are operated by Europe, U.S., Asia, Japan and India, respectively.

Upon ground-based systems, there are two general classes of augmentation: differential GPS (DGPS) and external sensors/systems, which are not of the interest of this project.

Differential GPS is a methodology that allows to improve GPS positioning or timing accuracy by using one or more ground-based reference stations at known locations (base station/s), that provide correcting data to the user GPS receiver (rover) via data link. It operates **cancelling most the natural and man-produced errors introduced in the normal GPS measurements.**

DGPS technique may be categorized in different ways: absolute or relative differential positioning; local, regional or wide area; and as code based or carrier based.

Absolute – Relative differential positioning

Absolute positioning references the user's final solution to the Earth using ECEF coordinate system. To achieve that, base station/s reference location/s must be also referred to these coordinates.

Otherwise, relative positioning, does not provide a solution attached to the Earth but referred to the base station, which ECEF position might be not perfectly determined. This technique is applied, for instance, if DGPS is implemented to land aircraft on an aircraft carrier.

Local, Regional and Wide Area

DGPS may be categorized by the geographic area that wants to be covered. So that, in first place are those DGPS systems designed to cover areas smaller than 100km with a single reference station, which is the case of this project, called Local Area.

To cover larger geographic regions, multiple reference stations and special algorithms are required. From 100km until 1000km more or less, the term Regional Area is employed and Wide Area if larger.

Code-based – Carrier-based

DGPS can be also classified between code-based and carrier-based techniques. As the own name indicates, code-based systems rely on code (pseudorange) measurements and carrier-based systems on carrier phase measurements, which are much more precise than pseudorange measurement but the whole number of cycles between satellite and receiver is not measurable so an ambiguity problem has to be solved.

<i>Only GPS vs. Local Area DGPS (LADGPS) typical contributions to UERE</i>			
Segment Source	Error Source	GPS (m)	Code-based LADGPS (m)
Space/control	Broadcast clock	1.1	0.0
	L1 P(Y) – L1 C/A group delay	0.3	0.0
	Broadcast ephemeris	0.8	0.1-0.6 mm/km
User	Ionospheric delay	7.0	0.2-4 cm/km
	Tropospheric delay	0.2	1-4 cm/km
	Receiver noise and resolution	0.1	0.1
	Multipath	0.2	0.3
Total (Root-Sum-Square)		7.1	0.3 + 1-6 cm/km

Table 4: Only GPS vs. LADGPS typical contributions to UERE

B. Market research

Even if it has not been the final choice, meanwhile it was not clear if the other options would work, the possibility of buying a pair of raw capable receivers was considered.

After analysing the market, some of the most feasible options attending the requirements and the limited sources have been listed in the following table:

<i>Raw capable GPS receivers</i>					
Model	Data Logging	Accuracy¹	Sensitivity	Preu	Conditions
NEO-7P	No	<1m	-161 dBm	89.00€	-
NEO-6P	No	<1m	-146 dBm	134.00€	-
LEA-6T	No	2.5m	-162 dBm	29.98\$	Free Shipping 14-30 business days
M8N	Yes	2.0m	-167dBm	44.00€	-
M8P	Yes	-	-	189.00€	-

Table 5: Raw capable GPS receivers

¹ The available GPS modules, u-Blox NEO 7-N, have 2.5m of accuracy.

C. Navigation Message tables

Navigation Message's structure has been already presented in the report, however some key information to decode it is missing and presented here.

C.1 Subframe 1

Subframe 1 – Satellite Clock and Health Data				
Parameter	No. of Bits	Scale Factor (LSB)	Units	
Code on L2	2	1	none	
Week Number	10	1	weeks	
L2 P data flag	1	1	none	
SV accuracy	4	See table 7		
SV Health	6	See table 8		none
Timing Group Delay: T_{GD}	8^*	2^{-31}	seconds	
Issue Of Data: Clock - IODC	10	-	-	
Clock parameters	Time Of Clock: t_{oc}	16	2^4	seconds
	a_{f2}	8^*	2^{-55}	sec/sec^2
	a_{f1}	16^*	2^{-43}	sec/sec
	a_{f0}	22^*	2^{-31}	seconds

Table 6: Satellite Clock & Health Data, according to [9]

Navigation Message – Data Health Indications		
The 6-bit word SV Health provides a one-bit summary of NAV's health status. (1 means healthy). The rest 5 LSBs specifies the status more accurately.		
MSB	LSB	Definition
1 0 0 0 0 0		All Signals OK
0 0 0 0 0 1		All Signals Weak*
0 0 0 0 1 0		All Signals Dead
0 0 0 0 1 1		All Signals Have No Data Modulation
0 0 0 1 0 0		L1 P Signal Weak
0 0 0 1 0 1		L1 P Signal Dead
0 0 0 1 1 0		L1 P Signal has No Data Modulation
0 0 0 1 1 1		L2 P Signal Weak
0 0 1 0 0 0		L2 P Signal Dead
0 0 1 0 0 1		L2 P Signal has No Data Modulation
0 0 1 0 1 0		L1 C Signal Weak
0 0 1 0 1 1		L1 C Signal Dead
0 0 1 1 0 0		L1 C Signal has No Data Modulation
0 0 1 1 0 1		L2 C Signal Weak
0 0 1 1 1 0		L2 C Signal Dead
0 0 1 1 1 1		L2 C Signal has No Data Modulation
0 1 0 0 0 0		L1 & L2 P Signal Weak
0 1 0 0 0 1		L1 & L2 P Signal Dead
0 1 0 0 1 0		L1 & L2 P Signal has No Data Modulation
0 1 0 0 1 1		L1 & L2 C Signal Weak
0 1 0 1 0 0		L1 & L2 C Signal Dead
0 1 0 1 0 1		L1 & L2 C Signal has No Data Modulation
0 1 0 1 1 0		L1 Signal Weak
0 1 0 1 1 1		L1 Signal Dead
0 1 1 0 0 0		L1 Signal has No Data Modulation
0 1 1 0 0 1		L2 Signal Weak
0 1 1 0 1 0		L2 Signal Dead
0 1 1 0 1 1		L2 Signal has No Data Modulation
0 1 1 1 0 0		SV Is Temporarily Out
0 1 1 1 0 1		SV Will Be Temporarily Out
0 1 1 1 1 0		One Or More Signals Are Deformed, However The Relevant URA Parameters Are Valid**
0 1 1 1 1 1		More Than One Combination Would Be Required To Describe Anomalies

* 3 to 6 dB below specified power level due to reduced power output, excess phase noise, SV altitude, etc.

*** Note: Deformed means one or more signals do not meet the requirements of Section 3. [9]

Table 7: DHI Navigation Message, according to [9]

User Range Accuracy (URA)			
URA Index	URA (meters)		
0	0.00	$< URA \leq$	2.40
1	2.40	$< URA \leq$	3.40
2	3.40	$< URA \leq$	4.85
3	4.85	$< URA \leq$	6.85
4	6.85	$< URA \leq$	9.65
5	9.65	$< URA \leq$	13.65
6	13.65	$< URA \leq$	24.00
7	24.00	$< URA \leq$	48.00
8	48.00	$< URA \leq$	96.00
9	96.00	$< URA \leq$	192.00
10	192.00	$< URA \leq$	384.00
11	384.00	$< URA \leq$	768.00
12	768.00	$< URA \leq$	1356.00
13	1356.00	$< URA \leq$	3072.00
14	3072.00	$< URA \leq$	6144.00
15	$6144.00 < URA \leq$ (Extremely inaccurate or not Available)		

Table 8: URA, according to [9]

C.2 Subframe 2 and 3

Subframe 2 and 3 - Satellite Ephemeris Data: Parameters			
Parameter	No. of Bits	Scale Factor (LSB)	Units ²
IODE	8	-	-
C_{rs}	16*	2^{-5}	meters
Δn	16*	2^{-43}	semicircles/s
M_0	32*	2^{-31}	semicircles
C_{uc}	16*	2^{-29}	radians
e	32	2^{-33}	dimensionless
C_{us}	16*	2^{-29}	radians
\sqrt{A}	32	2^{-19}	$\sqrt{\text{meters}}$
t_{oe}	16	2^4	seconds
C_{ic}	16*	2^{-29}	radians
Ω_0	32*	2^{-31}	semicircles
C_{is}	16*	2^{-29}	radians
i_0	32*	2^{-31}	semicircles
C_{rc}	16*	2^{-5}	meters
ω	32*	2^{-31}	semicircles
$\dot{\Omega}$	24*	2^{-43}	semicircles/sec
IDOT	14*	2^{-43}	semicircles/sec

Table 9: Satellite Ephemeris Data: Parameters, according to [9]

² The unit semicircle is radians divided by π .

D. Protocol messages' structures

In this chapter it is detailed the structure of those messages that have been used or had any utility at some point.

D.1 NMEA Standard Messages

As mentioned in the previous section, each message identifies its 'Talker' by two letters after '\$'. These identifiers are:

NMEA Talker ID						
Identifier	GA	GB	GL	GN	GP	P
GNSS type	SBAS	BeiDou	GALILEO	GLONASS	GPS	Proprietary

Table 10: NMEA Talker ID, according to [11]

Next will be introduced those sentences that have been useful in some way or another for this project.

D.1.1 GGA: Global Positioning System fix data

xxGGA messages provide information of UTC time, global positioning in geodetic coordinates, number of satellites used, horizontal dilution of precision, geoid separation and DGPS parameters if used (otherwise they are blank).

Message	GGA: Global Positioning System fix data				
Message Info	<i>ID for CFG-MSG</i>	<i>Number of fields</i>			
	0xF0 0x00	17			
Message Structure	\$xxGGA,time,lat,NS,long,EW,quality,numSV,HDOP,alt,M,sep,M,diffAge, diffStation*cs<CR><LF>				
Example	\$GPGGA,160006.00,4133.82991,N,00201.28764,E,1,05,2.49,286.5,M,49.3,M,,*5B				

Field No.	Name	Unit	Format	Example	Description
0	xxGGA	-	string	\$GPGGA	xx: Talker ID GGA: Message ID
1	time	-	hhmmss.ss	160006.00	UTC time
2	lat	-	ddmm.mmmmmm	4133.82991	Latitude
3	NS	-	character	N	North/South indicator
4	long	-	dddmm.mmmmmm	00201.28764	Longitude
5	EW	-	character	E	East/West indicator
6	quality	-	digit	1	Quality indicator (see table11)
7	numSV	-	numeric	05	Number of satellites used (0-12)
8	HDOP	-	numeric	2.49	Horizontal DOP
9	alt	m	numeric	286.5	Altitude above MSL
10	uAlt	-	character	M	Altitude units
11	sep	m	numeric	49.3	Geoid Separation
12	uSep	-	character	M	Separation units
13	diffAge	s	numeric	-	Age of differential corrections
14	diffStation	-	numeric	-	ID of differential station
15	cs	-	hexadecimal	*5B	Checksum
16	<CR> <LF>	-	character	-	Carriage Return and Line Feed

Quality Indicator	Description
0	No Fix / Invalid
1	Standard GPS (2D/3D)
2	Differential GPS
6	Estimated (DR) Fix

Table 11: GGA: GPS fix data message structure, according to [11]

D.1.2 GRS: GNSS Range Residuals

xxGRS messages provide Range Residuals information, which allows to estimate pseudoranges as detailed in the report.

Message	GNSS Range Residuals		
Message Info	<i>ID for CFG-MSG</i>	<i>Number of fields</i>	
	0xF0 0x06	17	
Message Structure	\$xxGRS,time, mode {,residual}*cs<CR><LF>		
Example	\$GPGRS,193353.60,1,0.2,-0.2,-0.9,0.6,0.6,-3.8,5.6,-1.8,-20.0,-2.9,3.4,*5E		

Field No.	Name	Unit	Format	Example	Description
0	xxGRS	-	string	\$GPGRS	xx: Talker ID GRS: Message ID
1	time	-	hhmmss.ss	193353.60	UTC time
2	mode	-	digit	1	Mode (always 1 in <i>u-Blox</i> receivers)
3 – 14	residual	m	numeric	0.2 – 3.4	Range residuals for SVs used in navigation. Until a maximum of 12 SVs matching xxGSA order.
15	cs	-	hexadecimal	*5E	Checksum
16	<CR><LF>	-	character	-	Carriage Return and Line Feed

Mode	Description
0	Residuals were used to calculate the position given in the matching GGA sentence.
1	Residuals were recomputed after the GGA position was computed.

Table 12: GNSS Range Residuals message structure, according to [11]

D.1.3 GSA: GNSS DOP and Active Satellites

xxGSA messages give information of Dilution of Precision, active satellites (understood as used satellites) and operational and navigation modes. Even though active satellites can be deduced from ephemerides, this message tell us from which Space Vehicle each range residual comes from.

Message	GNSS DOP and Active Satellites		
Message Info	<i>ID for CFG-MSG</i>	<i>Number of fields</i>	
	0xF0 0x02	17	
Message Structure	\$xxGSA,opMode,navMode{,sv},PDOP,HDOP,VDOP*cs<CR><LF>		
Example	\$GPGSA,A,3,27,08,16,10,18,11,26,01,14,22,32,,1.65,0.90,1.38*00		

Field No.	Name	Unit	Format	Example	Description
0	xxGSA	-	string	\$GPGSA	xx: Talker ID GSA: Message ID
1	opMode	-	character	A	Operation Mode
2	navMode	-	digit	3	Navigation Mode
3 – 14	SV	-	numeric	27– 32	Satellite number
15	PDOP	-	numeric	1.65	Position Dilution of Precision
16	HDOP	-	numeric	0.90	Horizontal Dilution of Precision
17	VDOP	-	numeric	1.38	Vertical Dilution of Precision
18	cs	-	hexadecimal	*00	Checksum
19	<CR><LF>	-	character	-	Carriage Return and Line Feed

Operation Mode	Description
M	Manually set to operate in 2D or 3D mode
A	Automatically switching between 2D or 3D mode

Navigation Mode	Description
1	Fix not available
2	2D Fix
3	3D Fix

Table 13: GNSS DOP & Active Satellites message structure, according to [11]

D.1.4 TXT: Text Transmission

xxTXT outputs some specific information about the receiver such as software version or errors. Particularly it has been useful in those cases when the transmission buffer was saturated and baud rate had to be increased.

Message	Text Transmission					
Message Info	<i>ID for CFG-MSG</i>	<i>Number of fields</i>				
	0xF0 0x41	7				
Message Structure	\$xxTXT,numMsg,msgNum,msgType,text*cs<CR><LF>					
Example	\$GPTXT,01,01,00,txbuf alloc*7F					

Field No.	Name	Unit	Format	Example	Description
0	xxTXT	-	string	\$GPTXT	xx: Talker ID TXT: Message ID
1	numMsg	-	numeric	01	Total number of messages in this transmission, 01..99
2	msgNum	-	numeric	01	Message number in this transmission, range 01..xx
3	msgType	-	numeric	00	Text identifier, u-blox GPS receivers specify the type of the message with this number. 00: Error 01: Warning 02: Notice 07: User
4	text	-	string	txbuf alloc	Position Dilution of Precision
5	cs	-	hexadecimal	*00	Checksum
6	<CR><LF>	-	character	-	Carriage Return and Line Feed

Table 14: Text Transmission message structure, according to [11]

D.2 UBX

UBX has defined some variable types, which are ordered in the **LittleEndian** format unless otherwise indicated. UBX variable types that has been used are shown in table 14.

Name	Type	Bytes	Comment	Min/Max
U1	Unsigned Char	1	-	0 ~ 255
X1	Bitfield	1	-	n/a
U2	Unsigned Short	2	-	0 ~ 65535
I2	Signed Short	2	2's complement	-32768 ~ 32767

U4	Unsigned Long	4	-	$0 \sim 4,294,967,295$
I4	Signed Long	4	2's complement	$-2,147,483,648 \sim 2,147,483,647$
X4	Bitfield	4	-	n/a
R8	IEEE 754 Double Precision	8	-	$-2^{1023} \sim 2^{1023}$

Table 15: UBX variable types message structure, according to [11]

D.2.1 AID (0x0B)

AID class contains aiding messages contained in the navigation message, as explained in the report, such as Ephemeris or Almanac, GPS health, ionospheric parameters and clock correction parameters. Due to all these messages come from the navigation message, they are linked in *U-Center* and cannot only enable specific information of the navigation message. Irrelevant messages shall be ignored.

D.2.1.1 AID-EPH (0x0B 0x31)

As explained in the report in order to compute satellites position, clock parameters and Ephemeris data are necessary. This information is transmitted in subframes 1 to 3 of the navigation message. UBX AID-EPH messages contain Space Vehicle ID, Hand-Over Word (HOW) of the first subframe and 24 words following the Hand-Over Word of mentioned subframes, SF1D0 to SF3D7. Note that this 24 words are from 3 to 10, included, of the 3 subframes, where the Telemetry Word (TLM) and the Hand-Over Word have been removed with the parity bits. Also the week number of subframe 1 has been modified to match Time Of Ephemeris (TOE).

Ephemerides are received in packets of 32 messages each one referring to a Space Vehicle of GPS constellation. For those Space Vehicles whose ephemeris are not available the payload is reduced to 8 bytes all of them set to zero, otherwise it is 104 bytes long.

Thus, UBX AID-EPH messages allow to read and store navigation message words, and once decoded, according to [9], can give us orbital and clock parameters required to compute satellite position.

Message	AID-EPH: Aiding Ephemeris Data for a SV				
Message Structure	<i>Header</i>	<i>ID</i>	<i>Length (Bytes)</i>	<i>Payload</i>	<i>Checksum</i>
	0xB5 0x62	0x0B 0x31	8 or 104	See below	CK_A CK_B

Payload Contents					
First Byte position³	Format⁴	Scaling	Name	Unit	Description
0	U4	-	SVID	-	SV ID for which this ephemeris data is (range: 1 to 32)
4	U4	-	HOW	-	Hand-Over Word of SF1
Start of optional block: sent if ephemeris are available for this SV					
8	U4	-	SF1D	-	Subframe 1 Words 3 to 10
40	U4	-	SF2D	-	Subframe 2 Words 3 to 10
72	U4	-	SF3D	-	Subframe 3 Words 3 to 10
End of optional block					

Table 16: AID-EPH message structure, according to [11]

D.2.1.2 AID-HUI (0x0B 0x02)

Message	GPS Health, UTC and ionosphere parameters				
Message Structure	<i>Header</i>	<i>ID</i>	<i>Length (Bytes)</i>	<i>Payload</i>	<i>Checksum</i>
	0xB5 0x62	0x0B 0x02	72	See below	CK_A CK_B

Payload Contents					
First Byte position⁵	Format	Scaling	Name	Unit	Description
0	X4	-	health	-	Bitmask, every bit represent a GPS SV

³ Location of the word first byte inside the payload.

⁴ The first byte of each AID-EPH payload field begin with 00 and shall be ignored.

⁵ Location of the word first byte inside the payload.

					(1-32). If the bit is set the SV is healthy.
4	R8	-	utcA0	-	UTC – parameter A0
12	R8	-	utcA1	-	UTC – parameter A1
20	I4	-	utcTOW	-	UTC – reference TOW
24	I2	-	utcWNT	-	UTC – reference Week Number
26	I2	-	utcLS	-	UTC – time difference due to leap seconds before event
28	I2	-	utcWNF	-	UTC – Week Number when next leap second event occurs
30	I2	-	utcDN	-	UTC – day of week when next leap second event occurs
32	I2	-	utcLSF	-	UTC – time difference due to leap seconds after event
34	I2	-	utcSpare	-	UTC – Spare to ensure structure is a multiple of 4 bytes
36	R4	-	klobA0	s	Klobuchar – alpha 0
40	R4	-	klobA1	s/semicircle	Klobuchar – alpha 1
44	R4	-	klobA2	s/semicircle ²	Klobuchar – alpha 2
48	R4	-	klobA3	s/semicircle ³	Klobuchar – alpha 3
52	R4	-	klobB0	s	Klobuchar – beta 0
56	R4	-	klobB1	s/semicircle	Klobuchar – beta 1
60	R4	-	klobB2	s/semicircle ²	Klobuchar – beta 2
64	R4	-	klobB3	s/semicircle ³	Klobuchar – beta 3
68	X4	-	flags	-	Flags, see table 17

Flags		
Name	Byte No.	Description
healthValid	0	Healthmask field in this message is valid
utcValid	1	UTC parameter fields in this message are valid
klobValid	2	Klobuchar parameter fields in this message are valid

Table 17: GPS Health, UTC & ionosphere parameters message structure, according to [11]

D.2.2 NAV (0x01)

NAV class contains those messages that provide information about the navigation solution, such as position in geodetic and ECEF coordinates, velocity, altitude, DOP, clock solution, etc.

D.2.2.1 NAV-DOP

Message	Dilution Of Precision				
Message Structure	<i>Header</i>	<i>ID</i>	<i>Length (Bytes)</i>	<i>Payload</i>	<i>Checksum</i>
	0xB5 0x62	0x01 0x04	18	See below	CK_A CK_B

Payload Contents					
First Byte position ⁶	Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS TOW of the navigation epoch.
4	U2	0.01	gDOP	-	Geometric DOP
6	U2	0.01	pDOP	-	Position DOP
8	U2	0.01	tDOP	-	Time DOP
10	U2	0.01	vDOP	-	Vertical DOP
12	U2	0.01	hDOP	-	Horizontal DOP
14	U2	0.01	nDOP	-	Northing DOP
16	U2	0.01	eDOP	-	Easting DOP

Table 18: Dilution Of Precision message structure, according to [11]

D.2.2.2 NAV-POSECEF

Message	Position Solution in ECEF				
Message Structure	<i>Header</i>	<i>ID</i>	<i>Length (Bytes)</i>	<i>Payload</i>	<i>Checksum</i>
	0xB5 0x62	0x01 0x01	20	See below	CK_A CK_B

⁶ Location of the word first byte inside the payload.

Payload Contents					
First Byte position ⁷	Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS TOW of the navigation epoch.
4	I4	-	ecefX	cm	ECEF X coordinate
8	I4	-	ecefY	cm	ECEF Y coordinate
12	I4	-	ecefZ	cm	ECEF Z coordinate
16	U4	-	pAcc	cm	Position Accuracy Estimate

Table 19: Position Solution in ECEF message structure, according to [11]

D.2.2.3 NAV-POSLH

Message	Geodetic Position Solution				
Message Structure	Header	ID	Length (Bytes)	Payload	Checksum
	0xB5 0x62	0x01 0x02	28	See below	CK_A CK_B

Payload Contents					
First Byte position ⁸	Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS TOW of the navigation epoch.
4	I4	1e-7	lon	deg	Longitude
8	I4	1e-7	lat	deg	Latitude
12	I4	-	height	mm	Height above ellipsoid
16	I4	-	hMSL	mm	Height above MSL
20	U4	-	hAcc	mm	Horizontal accuracy estimate
24	U4	-	vAcc	mm	Vertical accuracy estimate

⁷ Location of the word first byte inside the payload.

⁸ Location of the word first byte inside the payload.

D.2.2.4 NAV-SOL

Message	Navigation Solution Information				
Message Structure	Header	ID	Length (Bytes)	Payload	Checksum
	0xB5 0x62	0x01 0x06	52	See below	CK_A CK_B

Payload Contents					
First Byte position ⁹	Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS TOW of the navigation epoch.
4	I4	-	fTOW	ns	Fractional part of iTOW ¹⁰ (range: +/-500000).
8	I2	-	week	weeks	GPS week of the navigation epoch
10	U1	-	gpsFix	-	GPSfix Type (see table 20)
11	X1	-	flags	-	Fix Status flags (see table 20)
12	I4	-	ecefX	cm	ECEF X coordinate
16	I4	-	ecefY	cm	ECEF Y coordinate
20	I4	-	ecefZ	cm	ECEF Z coordinate
24	U4	-	pAcc	cm	3D Position Accuracy Estimate
28	I4	-	ecefVX	cm/s	ECEF X velocity
32	I4	-	ecefVY	cm/s	ECEF Y velocity
36	I4	-	ecefVZ	cm/s	ECEF Z velocity
40	U4	-	sAcc	cm/s	Speed Accuracy Estimate
44	U2	0.01	pDOP	-	Position DOP
46	U1	-	reserved1	-	Reserved
47	U1	-	numSV	-	Number of SVs used in Nav solution
48	U4	-	reserved2	-	Reserved

⁹ Location of the word first byte inside the payload.

¹⁰ The precise GPS time of week in seconds is: $(iTOW \cdot 10^{-3}) + (fTOW \cdot 10^{-9})$

<i>GPSfix Type</i>	
Number	Description
0x00	No fix
0x01	Dead Reckoning only
0x02	2D-fix
0x03	3D-fix
0x04	GPS + dead reckoning combined
0x05	Time only fix
0x06	reserved

<i>Bitfield Flags</i>	
Name	Description
GPSfixOK	>1 = Fix within limits
DiffSoln	1 = DGPS used
WKNSET	1 = Valid GPS week number
TOWSET	1 = Valid GPS time of week (iTOW and fTOW)

Table 20: Navigation Solution Information message structure, according to [11]

E. Functions description

E.1 Main

Name	<i>init.m</i>
Description	This script initialize the whole process and it is where the user must define some parameters such true base station location, base station and rover binary data text file, correction mode, some settings and post-processing.

Table 21: init.m description

Name	<i>DGPS.m</i>	
Description	DGPS coordinates all the processes necessary to implement the whole DGPS process, since reading the files until recompute the final position.	
Declaration	<code>[rover, baseStation, SV, GNSSmessages, GNSSparameters] = DGPS(BSfileName, RfileName, trueECEF, Rclean, BSclean, correctionMode, radius, nSatellites)</code>	
Inputs	BSfileName	Name of the text file that contains Base Station binary data.
	RfileName	Name of the text file that contains Rover binary data.
	trueECEF	True Base Station receiver location in ECEF coordinates.
	Rclean	Cleaning flag of the Rover data file. 1: contains row number and text encoding, must be cleaned.
	BSclean	Cleaning flag of the Base Station data file. 1: contains row number and text encoding, must be cleaned.
	correctionMode	Specifies the correction mode: <ul style="list-style-type: none">• OFFSET: Offset correction• CLASSIC: Classical DGPS correction• NavSolR: Navigation Solution – Real SVs

		<ul style="list-style-type: none"> • NavSolV: Navigation Solution – Virtual SVs
	radius	Is the radius of the sphere distribution where the SVs are spreaded on, in case of NavSolV mode.
	nSatellites	Number of virtual SVs distributed along a sphere, in case of NavSolV mode. Must be the square of an integer.
Outputs	rover	Structure containing all the parameters related with the rover.
	baseStation	Structure containing all the parameters related with the base station.
	SV	Structure containing all the parameters related with the Space Vehicles.
	GNSSmessages	Structure containing all the decoded messages.
	GNSSparameters	Structure containing all the parameters extracted from GNSS messages.

Table 22: DGPS.m description

Name	<i>messageDecoder.m</i>	
Description	messageDecoder decodes all binary messages from U-Center stored in the file 'fileName'.	
Declaration	[GNSSmessages] = messageDecoder(fileName, clean)	
Inputs	fileName	Name of the file that contains the binary data.
	clean	Cleaning flag. 1: each row must be cleaned.
Output	GNSSmessages	Structure containing all the decoded messages stored in 'fileName'.

Table 23: messageDecoder.m description

Name	<i>trackSV.m</i>	
Description	trackSV coordinates the processes required to track a SV. Reads the input AID-EPH message contained in the navigation message subframes 2 & 3 plus clock correction in subframe 1, applies clock corrections and compute SV final position. Returns ephemerides and position of the current active SV.	
Declaration	<code>[SV] = trackSV(UBX_AID_EPH, SV, nSatellites)</code>	
Inputs	UBX_AID_EPH	UBX AID-EPH message.
	SV	SV structure.
	nSatellites	Number of total active Space Vehicles.
Output	SV	Updated SV structure.

Table 24: *trackSV.m* description

E.2 Get Functions

Name	<i>generateSV.m</i>	
Description	generateSV generates virtual SVs distributed along a sphere of given centre and radius. SVs position is generated for each rover position adapted to base station timing. Also returns the number of satellites used in the activeSV format.	
Declaration	<code>[SV, activeSV] = generateSV(centre, radius, nSatellites, roverPos)</code>	
Inputs	centre	Sphere's centre.
	radius	Sphere's radius.
	nSatellites	Number of satellites to be generated. It must be the square of an integer.

	roverPos	Rover's position adapted to base station timing.
Outputs	SV	SV structure.
	activeSV	Active SV in activeSV format. Includes SV ID and total number of active SVs.

Table 25: generateSV.m description

Name	<i>getActiveSV.m</i>	
Description	getActiveSV reads NMEA GPGSA sentence and returns array of SV ID used and its assigned message number.	
Declaration	[activeSV] = getActiveSV(GPGSA)	
Input	PGPSA	NMEA GPGSA sentence.
Output	activeSV	Active SVs ID, total number of SVs and message number.

Table 26: getActiveSV.m description

Name	<i>getAID_EPH.m</i>	
Description	AID_EPH reads UBX AID-EPH message payload (hexadecimal) and returns subframe words of the navigation message.	
Declaration	[AID_EPH] = getAID_EPH(payload)	
Input	payload	Payload of UBX AID-EPH messages.
Output	AID_EPH	Decoded navigation message subframes 1 to 3.

Table 27: getAID_EPH.m description

Name	<i>getColor.m</i>	
Description	getColor reads SV ID and assigns a color based on its ID and number of SVs.	
Declaration	<code>[color] = getColor(ID, SVs)</code>	
Inputs	ID	ID of the Space Vehicle.
	SVs	Number of active SVs.
Output	color	Assigned color

Table 28: *getColor.m* description

Name	<i>getDistance.m</i>	
Description	getDistance compute the distance between true location and all base station GPS receiver measures.	
Declaration	<code>[distance] = getDistance(posECEF, trueECEF)</code>	
Inputs	posECEF	Position array in ECEF coordinates.
	trueECEF	True location of the receiver in ECEF coordinates.
Output	distance	Distance array between each position and the true location.

Table 29: *getDistance.m* description

Name	<i>getDOP.m</i>	
Description	getDOP reads input GPGSA decode message and returns dilution of precision (PDOP, HDOP and VDOP) from NMEA GPGSA and message number previously assigned.	
Declaration	[DOP] = getDOP(GPGSA)	
Input	PGPSA	NMEA GPGSA decoded message.
Output	DOP	Dilution of Precision (PDOP, HDOP and VDOP).

Table 30: *getDOP.m* description

Name	<i>getEphemeris.m</i>	
Description	getEphemeris reads decoded navigation message words from subframe 1 to 3 and returns clock correction and ephemeris parameters.	
Declaration	[ephemeris] = getEphemeris(AID_EPH)	
Input	AID_EPH	UBX AID-EPH decoded message.
Output	eph	structure containing clock correction and ephemeris parameters.

Table 31: *getEphemeris.m* description

Name	<i>getfTOW.m</i>	
Description	getfTOW Assigns to geodetic position obtained by UBX NAV-POSLH messages a float precision TOW (fTOW). This value is the closest value of iTOW of UBX NAV-POSLH messages to the fTOW of UBX NAV-SOL. It is assumed that they come from the same navigation solution.	
Declaration	[geoPos] = getfTOW(geoPos, POSECEF)	

Inputs	geoPos	Position in WGS-84 coordinates.
	posECEF	Navigation solution final position in ECEF structure, also contains accurate TOW.
Output	geoPos	updated with high precision TOW geodetic position.

Table 32: getfTOW.m description

Name	<i>GNSSparameters.m</i>	
Description	getGNSSparameters coordinates all the GNSS parameters extraction process from decoded messages.	
Declaration	[GNSSparameters] = getGNSSparameters(GNSSmessages)	
Inputs	GNSSmessages	Structure containing all decoded messages.
	correctionMode	Correction mode, avoids unnecessary computations.
Output	GNSSparameters	structure containing all the extracted parameters.

Table 33: GNSSparameters.m description

Name	<i>getGPGGA.m</i>	
Description	getGPGGA reads NMEA GPGGA hexdecimal message and returns UTC time (hhmmss.sss), latitude (ddmm.mmmm), northing indicator, longitude (dddmm.mmmm), easstng indicator, Status, SVs used, HDOP, Alt(msl), geoid separation.	
Declaration	[GPGGA] = getGPGGA(payload)	
Input	payload	NMEA GPGGA payload (hexadecimal).

Output	GPGGA	structure containing UTC time(hhmmss.sss), latitude (ddmm.mmmm), northing indicator, longitude(ddmm.mmmm), easting indicator, Status, SVs used, HDOP, Alt(msl), geoid separation.
---------------	-------	---

Table 34: getGPGGA.m description

Name	<i>getGPGRS.m</i>	
Description	getGPGRS reads NMEA GPGRS hexdecimal message and returns UTC time and range residuals [m].	
Declaration	[GPGRS] = getGPGRS(payload)	
Input	payload	NMEA GPGRS payload (hexadecimal).
Output	GPGGA	structure containing UTC time and range residuals.

Table 35: getGPGRS.m description

Name	<i>getGPGSA.m</i>	
Description	getGPGSA Reads NMEA GPGSA hexdecimal message and returns operation mode, NAV mode, SVs used, SVID and dilution of precision (PDOP, HDOP, VDOP).	
Declaration	[GPGSA] = getGPGSA(payload)	
Input	payload	NMEA GPGSA payload (hexadecimal).
Output	GPGGA	structure containing operation mode, NAV mode, SVs used, SVID and dilution of precision (PDOP, HDOP, VDOP).

Table 36: getGPGSA.m description

Name	<i>getNAV_POSECEF.m</i>	
Description	getNAV_POSECEF reads UBX NAV-POSECEF message payload (hexadecimal) and returns TOW, ECEF-X, ECEF-Y, ECEF-Z and 3D position accuracy.	
Declaration	[NAV_POSECEF] = getNAV_POSECEF(payload)	
Input	payload	UBX NAV-POSECEF payload (hexadecimal).
Output	NAV_POSECEF	structure containing ms-precision TOW, ECEF-X, ECEF-Y, ECEF-Z and 3D position accuracy.

Table 37: *getNAV_POSECEF.m* description

Name	<i>getNAV_POSLLH.m</i>	
Description	getNAV_POSLLH reads UBX NAV-POSLLH message payload (hexadecimal) and returns TOW, lon [°], lat[°], height (above ellipsoid), hMSL, hAcc and vAcc.	
Declaration	[NAV_POSLLH] = getNAV_POSLLH(payload)	
Input	payload	UBX NAV-POSLLH message payload (hexadecimal).
Output	NAV_POSLLH	structure containing ms-precision TOW, lon [°], lat[°], height (above ellipsoid), hMSL, hAcc and vAcc.

Table 38: *getNAV_POSLLH.m* description

Name	<i>getNAV_SOL.m</i>	
Description	getNAV_SOL reads UBX NAV-SOL hexadecimal payload messages, decodes it and returns navigation solution parameters: Week Number, TOW, position fix type, fix flags, position ECEF (X,Y,Z), position accuracy estimation, velocity ECEF, velocity accurate estimate, PDOP and No. used SVs.	
Declaration	[NAV_SOL] = getNAV_SOL(payload)	

Input	payload	UBX NAV-SOL hexadecimal payload.
Output	NAV_SOL	structure containing Week Number, high precision TOW, position fix type, fix flags, position ECEF, position accuracy estimation, velocity ECEF, velocity accurate, PDOP and No. used SVs.

Table 39: getNAV_SOL.m description

Name	<i>getNMEAfixData.m</i>	
Description	getFixData reads NMEA GPGGA messages and returns lat, northing indicator, lon, easting indicator, altitude (MSL) and altitude (HAE).	
Declaration	[pos] = getNMEAfixData(GPGGA)	
Input	GPGGA	NMEA GPGGA message.
Output	geoPos	structure containing latitude, northing indicator, longitude, easting indicator, altitude above MSL and HAE.

Table 40: getNMEAfixData.m description

Name	<i>getPOSECEF.m</i>	
Description	getPOSECEF reads decode UBX NAV-POSECEF and assigns x, y and z in Earth-Centered Earth-Fixed coordinate system. Also assigns message number.	
Declaration	[posECEF] = getPOSECEF(NAV_POSECEF)	
Input	NAV_POSECEF	UBX NAV-POSECEF decoded parameters.
Output	posECEF	structure containing position in ECEF coordinates and message number.

Table 41: getPOSECEF.m description

Name	<i>getRange.m</i>	
Description	getRange Computes distance between 2 points (3D).	
Declaration	[GNSSparameters] = getGNSSparameters(GNSSmessages)	
Inputs	PointA	First point. Must be of the form: [point.x point.y point.z]
	PointB	Second point. Must be of the form: [point.x point.y point.z]
Output	range	range between point A and B.

Table 42: *getRange.m* description

Name	<i>getRangeResiduals.m</i>	
Description	getRangeResiduals reads NMEA GPGRS messages and returns range residuals associated to each SV and time-message number.	
Declaration	[SV] = getRangeResiduals(GPGRS, activeSV, nSatellites, SV)	
Inputs	GPGRS	NMEA GPGRS decoded message.
	activeSV	Active SVs.
	nSatellites	max number of satellites.
	SV	Space Vehicles structure.
Output	SV	Updated Space Vehicles structure.

Table 43: *getRangeResiduals.m* description

Name	<i>getSOL.m</i>	
Description	getSOL reads UBX NAV-SOL decoded messages and returns TOW, position ECEF (X,Y,Z) and SVs used.	
Declaration	[receiver, t_msg] = getSOL(NAV_SOL, receiver)	
Inputs	NAV_SOL	UBX NAV-SOL decoded message.
	receiver	Receiver structure.
Outputs	receiver	Updated receiver structure.
	t_msg	Table relating time and message number.

Table 44: *getSOL.m* description

Name	<i>getTimeTable.m</i>	
Description	getTimeTable Returns time - messages table from UTC time of NMEA GPGRS messages.	
Declaration	[t_msg] = getTimeTable(GPGRS)	
Input	GPGRS	NMEA GPGRS decoded message.
Output	t_msg	table that relates time and message number.

Table 45: *getTimeTable.m* description

Name	<i>getTrueECEF.m</i>
Description	getTrueECEF reads true location in initial format and geographic coordinate system and returns it in ECEF coordinate system.

Declaration	[trueECEF] = getTrueECEF(trueLocation)	
Input	trueLocation	True location in Google Earth format.
Output	trueECEF	true location in ECEF coordinates.

Table 46: getTrueECEF.m description

Name	<i>getTrueGeo.m</i>	
Description	getTrueGeo reads true location in initial format and geographic coordinate system and returns it in WGS-84 coordinates.	
Declaration	[trueGeo] = getTrueGeo(trueLocation)	
Input	trueLocation	True location in Google Earth format.
Output	trueGeo	true location in WGS-84 coordinates.

Table 47: getTrueGeo.m description

Name	<i>getUBXFixData.m</i>	
Description	getUBXFixData reads decoded UBX NAV-POSLH messages and returns fix geographic position.	
Declaration	[geoPos] = getUBXFixData(NAV_POSLLH)	
Input	NAV_POSLLH	UBX NAV-POSLH decoded message.
Output	geoPos	Fixed navigation solution in WGS-84 coordinates.

Table 48: getUBXFixData.m description

E.3 Timing

Name	<i>addTime.m</i>	
Description	addTime adds a given time increment to all the fields of the input receiver structure.	
Declaration	[receiver] = addTime(receiver, At)	
Inputs	receiver	Structure to modify.
	At	Time increment to add.
Output	receiver	Updated receiver structure.

Table 49: *addTime.m* description

Name	<i>assignTime.m</i>	
Description	assignTime reads data field and assigns receiving time from linear interpolation of time-message table and message number. Data field must contain subfield numMSG. After that, replaces numMSG by subfield time.	
Declaration	[dataField] = assignTime(dataField, t_msg)	
Inputs	dataField	Structure to assign time.
	t_msg	Time-message table.
Output	dataField	Updated structure.

Table 50: *assignTime.m* description

Name	<i>fixTime.m</i>	
Description	fixTime adapts unadapted input to fit reference timing.	
Declaration	[adapted] = fixTime(reference, unadapted, type)	
Inputs	reference	Taken as reference.
	unadapted	Unadapted structure..
Output	dataField	Adapted structure.

Table 51: *fixTime.m* description

Name	<i>unifyActiveSV.m</i>	
Description	unifyActiveSV unifies active SVs comparing active SVs information of rover and base station receivers.	
Declaration	[activeSV] = unifyActiveSV(time, activeSV_BSarray, activeSV_Rarray, activeSV_BS, activeSV_R)	
Inputs	time	Current time.
	activeSV_BSarray	Base station active SVs array.
	activeSV_Rarray	Rover active SVs array.
	activeSV_BS	Base station active SVs.
	activeSV_R	Rover active SVs.
Output	activeSV	Active SVs.

Table 52: *unifyActiveSV.m* description

Name	<i>unifyTimings.m</i>	
Description	unifyTimings coordinates time unification of different parameters taking base station receiver navigation solution as reference.	
Declaration	$[\text{rover}, \text{baseStation}, \text{SV}, \text{activeSV_BS}, \text{activeSV_R}] = \text{unifyTimings}(\text{rover}, \text{baseStation}, \text{SV}, \text{activeSV_BS}, \text{activeSV_R}, \text{correctionMode})$	
Inputs	rover	Rover structure.
	baseStation	Base station structure.
	SV	Space Vehicles structure.
	activeSV_BS	active SVs obtained from base station receiver.
	activeSV_R	active SVs obtained from rover receiver.
	correctionMode	Correction mode
Outputs	rover	Corrected rover structure.
	baseStation	Corrected base station structure.
	SV	Corrected Space Vehicles structure.
	activeSV_BS	Corrected active SVs obtained from base station receiver.
	activeSV_R	Corrected active SVs obtained from rover receiver.

Table 53: *unifyTimings.m* description

E.4 Correction

Name	<i>computeCorrectionOFFSET.m</i>	
Description	computeCorrectionOFFSET computes position domain corrections - OFFSET.	
Declaration	[rover] = computeCorrectionOFFSET(rover, baseStation, trueECEF)	
Inputs	rover	Rover structure.
	baseStation	Base station structure.
	trueECEF	True base station location in ECEF coordinates.
Output	rover	Corrected and updated rover position.

Table 54: *computeCorrectionOFFSET.m* description

Name	<i>computeCorrectionReal.m</i>	
Description	computeCorrectionReal computes DGPS correction applied to Navigation Solution using real active SVs.	
Declaration	[SV, rover, activeSV] = computeCorrectionReal(SV, rover, baseStation, trueECEF, activeSV_BS, activeSV_R)	
Inputs	SV	Space Vehicles structure.
	rover	Rover structure.
	baseStation	Base station structure.
	trueECEF	True base station location in ECEF coordinates.
	activeSV_BS	active SVs according to base station receiver.

	activeSV_R	active SVs according to rover receiver.
Outputs	SV	updated with pseudoranges SV structure.
	rover	time adapted and updated rover structure.
	activeSV	Active SVs.

Table 55: computeCorrectionReal.m description

Name	<i>computeCorrectionVirtual.m</i>	
Description	computeCorrectionVirtual computes DGPS correction applied to Navigation Solution using virtually generated SVs.	
Declaration	<code>[SV, rover] = computeCorrectionVirtual(SV, rover, baseStation, trueECEF)</code>	
Inputs	SV	Space Vehicles structure.
	rover	Rover structure.
	baseStation	Base station structure.
	trueECEF	True base station location in ECEF coordinates.
	activeSV_BS	active SVs according to base station receiver.
	activeSV_R	active SVs according to rover receiver.
Outputs	SV	updated with pseudoranges SV structure.
	rover	time adapted and updated rover structure.
	activeSV	Active SVs.

Table 56: computeCorrectionVirtual.m description

E.5 Positioning

Name	<i>computePosition.m</i>	
Description	correctPosition coordinates position computation once corrections have been applied.	
Declaration	[rover] = computePosition(SV, rover, activeSV)	
Inputs	SV	Space Vehicles structure
	rover	Rover structure.
	activeSV	Active Space Vehicles.
Output	rover	Updated rover position.

Table 57: *computePosition.m* description

Name	<i>getSVposition.m</i>	
Description	getSVposition returns Space Vehicle's position from ephemeris data and clock correction parameters.	
Declaration	[satPos] = getSVposition(eph, transmitTime)	
Inputs	eph	ephemeris and clock correction parameters.
	trasnmitTime	time at which the position is wanted to be estimated.
Output	satPos	Space Vehicle final position.

Table 58: *getSVposition.m*

Name	<i>leastSquarePos.m</i>	
Description	leastSquarePos computes final position by least square method. This method accounts for receiver clock delay.	
Declaration	$[pos, DOP] = \text{leastSquarePos}(SV, n, \text{activeSV}, \text{initPos})$	
Inputs	SV	Space Vehicles structure
	n	Current message number.
	activeSV	Active Space Vehicles.
	initPos	Initial position.
Outputs	pos	Position final solution.
	DOP	Dilution Of Precision (GDOP, PDOP, HDOP, VDOP)

Table 59: *leastSquarePos.m* description

Name	<i>leastSquarePosNOdt.m</i>	
Description	leastSquarePosNOdt computes final position by least square method. This method does not account for receiver clock delay.	
Declaration	$[pos, DOP] = \text{leastSquarePosNOdt}(SV, n, \text{activeSV}, \text{initPos})$	
Inputs	SV	Space Vehicles structure
	n	Current message number.
	activeSV	Active Space Vehicles.
	initPos	Initial position.

Outputs	pos	Position final solution.
	DOP	Dilution Of Precision (GDOP, PDOP, HDOP, VDOP)

Table 60: leastSquarePosNOdt.m description

E.6 Export

Name	<i>getTemplate.m</i>
Description	This script generates and stores KML file templates that can be used in exportKML.m file to export results in KML format that then can be visualized in Google Earth or Google Maps.

Table 61: getTemplate.m description

Name	<i>exportKML.m</i>	
Description	exportKML exports results generating a KML file. This function allows to export 3 routes. In this project it has been used to export uncorrected and corrected paths plus Base Station true location.	
Declaration	[] = exportKML(fileName, routeName1, data1, routeName2, data2, trueLocName, trueLoc)	
Inputs	filename	name of the generated file.
	routeName1	name of the first route.
	data1	route 1 in geodetic coordinates.
	routeName2	name of the second route.
	data2	route 3 in geodetic coordinates.
	trueLocName	name of true location.
	trueLoc	true location in geodetic coordinates.

Table 62: exportKML.m description

E.7 Utils

Name	<i>check_t.m</i>	
Description	check_t corrects time format accounting for beginning or end of week crossover.	
Declaration	[corrTime] = check_t(time)	
Input	time	Time in seconds.
Output	corrTime	Corrected time (seconds)

Table 63: *check_t.m* description

Name	<i>checkOrder.m</i>	
Description	checkOrder corrects byte order of input messages.	
Declaration	[corrected] = checkOrder(raw)	
Input	raw	uncorrected message.
Output	corrected	Correcrd byte order message.

Table 64: *checkOrder.m* description

Name	<i>cleanRow.m</i>	
Description	cleanRow cleans rows that contains line number, hexadecimal data and ASCII encoding. Line number and ASCII encoding shall be removed.	

Declaration	[row] = cleanRow(row)	
Input	row	Dirty row
Output	row	Clean row

Table 65: clean Row.m description

Name	<i>closestValue.m</i>	
Description	closestValue returns upper and lower closest values of an array to a given number. Returns their locations as well.	
Declaration	[lowClose, upClose, lowLoc, upLoc] = closestValue(value, array)	
Inputs	value	Value to be approximated.
	array	Input array.
Outputs	lowClose	lower value of the array closest to value.
	upClose	upper value of the array closest to value.
	lowLoc	position in the array of lower closest value.
	upLoc	position in the array of upper closest value.

Table 66: closestValue.m description

Name	<i>ECEF2geodeticArray.m</i>	
Description	geodetic2ECEFarray converts all cells of array from ECEF coordinate system based in WGS-84 coordinates to geodetic.	
Declaration	<code>[geodetic] = ECEF2geodetic(ECEF)</code>	
Input	ECEF	array of position in ECEF coordinates.
Output	geodetic	array of position in WGS-84 coordinates.

Table 67: ECEF2geodeticArray.m description

Name	<i>geodetic2ECEFarray.m</i>	
Description	geodetic2ECEFarray converts all cells of array from geodetic coordinates to ECEF coordinate system based in WGS-84.	
Declaration	<code>[ECEF] = geodetic2ECEFarray(geodetic)</code>	
Input	geodetic	array of position in WGS-84 coordinates.
Output	ECEF	array of position in ECEF coordinates.

Table 68: geodetic2ECEFarray.m description

Name	<i>hex2decimal.m</i>	
Description	hex2number reads hexadecimal message and translates it into decimal number byte by byte.	
Declaration	<code>[msg] = hex2decimal(hex)</code>	
Input	hex	Hexadecimal message.
Output	msg	Translated message.

Table 69: hex2decimal.m description

Name	<i>hex2str.m</i>	
Description	hex2str reads hexadecimal message and translates it into ASCII byte by byte.	
Declaration	[msg] = hex2str(hex)	
Input	hex	Hexadecimal message.
Output	msg	ASCII translated message.

Table 70: hex2str.m description

Name	<i>twosComp2dec.m</i>	
Description	twosComp2dec converts two's complement coded binary numbers to decimal integer.	
Declaration	[intNumber] = twosComp2dec(binaryNumber)	
Input	binaryNumber	two's complement binary number.
Output	intNumber	decimal integer.

Table 71: twosComp2dec.m description

Name	<i>utc2sec.m</i>	
Description	utc2seconds reads UTC time in format hhmmss.sss and converts it into seconds.	
Declaration	[seconds] = utc2seconds(UTC)	

Input	UTC	UTC time in format hhmmss.sss
Output	time	Time in seconds.

Table 72: utc2sec.m description

F. Main Structures

All the data that this code works with is stored in structures. The net of structures is too complex to be graphically represented, instead, a brief of explanation of the content of the main structures is given.

In general all parameters that depend on time, are structured arrays of the form, for instance in the case of the original rover's ECEF position:

rover.posECEF					
num Msg	TOW	time	x	y	z
8	2.4326619968 74740e+05	15.79953515 7013452	4.7898922000 00000e+06	1.7158279000 00000e+05	4.1941151000 00000e+06
:	:	:	:	:	:
2700	2.4334099973 39110e+05	90.59958159 4025950	4.7898984500 00000e+06	1.7158513000 00000e+05	4.1941176500 00000e+06

Table 73: Structure example: rover.posECEF

F.1 GNSSmessages

Contains all the decoded messages received from the GPS devices. Classifies messages by receiver, rover or base station, and protocol, UBX or NMEA. Inside the latter classification message classes store each payload and message number.

F.2 GNSSparameters

Contains all the extracted parameters from the decoded messages. Classifies parameters referring to base station or rover, and contain all the information extracted from the received messages such as receiver navigation solution, Space Vehicles, DOP, time-message table, etc.

F.3 BaseStation

Contains all the data provided by base station GPS receiver.

F.4 rover

Contains all the data provided by the rover GPS receiver plus the final post-correction solution such as position and DOP.

F.5 SV

It is an array of 32 parameters that contains all the information related with all the Space Vehicles, such as ephemeris , position, original range residuals of the base station and rover, time corrected position of the satellites, original ranges , pseudoranges and correction computed from base station and the original and corrected pseudoranges and range residuals of the rover.

Each cell of the array is a Space Vehicle and it is identified by the parameter ID. Those fields where data is missing will remain empty.

G. Configuration files

The configuration files that store the settings of the receivers' used in the tests are attached below. Exported into a text file, can be loaded to the receivers using *U-Center*.

G.1 Base Station

```
MON-VER - 0A 04 BE 00 31 2E 30 30 20 28 35 39 38 34 33 29 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
30 30 30 37 30 30 30 00 00 31 2E 30 30 20 28 35 39 38 34 32 29 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 50 52 4F 54 56 45 52 20 31 34 2E 30 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 46 49 53 20 30 78  
45 46 34 30 31 35 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04D 4F 44 20 4E 45 4F 2D 37 4E 2D 30 00  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 47 50 53 3B 53 42 41 53 3B 47 4C 4F 3B 51 5A 53 53 00 00 00  
00 00 00 00 00 00 00 00 00  
CFG-ANT - 06 13 04 00 1B 00 F0 B9  
CFG-DAT - 06 06 02 00 00 00  
CFG-GNSS - 06 3E 24 00 00 16 16 04 00 04 FF 00 01 00 00 00 01 01 03 00 01 00 00 00 05 00 03 00 01 00 00 00 06  
08 FF 00 00 00 00 00  
CFG-INF - 06 02 0A 00 00 00 00 00 00 00 00 00 00 00  
CFG-INF - 06 02 0A 00 01 00 00 00 87 87 87 87 87 87  
CFG-INF - 06 02 0A 00 03 00 00 00 00 00 00 00 00 00  
CFG-ITFM - 06 39 08 00 F3 AC 62 2D 1E 03 00 00  
CFG-LOGFILTER - 06 47 0C 00 01 00 00 00 00 00 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 30 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 32 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 33 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 31 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 01 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 0B 00 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 21 08 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 0B 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 09 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 02 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 06 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 07 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 21 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0A 08 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 60 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 22 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 31 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 04 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 01 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 02 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 01 07 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 32 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 06 00 01 00 00 00 00  
CFG-MSG - 06 01 08 00 01 03 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 30 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 20 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 21 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 11 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 01 12 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 02 20 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0D 03 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0D 01 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 0D 06 00 00 00 00 00 00 00  
CFG-MSG - 06 01 08 00 F0 00 01 00 01 01 01 01
```

CFG-MSG - 06 01 08 00 F0 01 01 00 01 01 01 01 01
 CFG-MSG - 06 01 08 00 F0 02 01 01 01 01 01 01 01
 CFG-MSG - 06 01 08 00 F0 03 01 00 01 01 01 01 01
 CFG-MSG - 06 01 08 00 F0 04 01 00 01 01 01 01 01
 CFG-MSG - 06 01 08 00 F0 05 01 00 01 01 01 01 01
 CFG-MSG - 06 01 08 00 F0 06 00 01 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F0 07 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F0 08 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F0 09 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F0 0A 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F0 0D 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F1 00 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F1 03 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 F1 04 00 00 00 00 00 00 00
 CFG-NAV5 - 06 24 24 00 FF FF 00 03 00 00 00 00 10 27 00 00 05 00 FA 00 FA 00 64 00 2C 01 00 3C 00 00 00 00 00
 00 00 00 00 00 00
 CFG-NAVX5 - 06 23 28 00 00 00 FF FF 0F 00 00 00 03 02 03 16 07 00 00 01 00 00 9B 06 00 00 00 00 01 01 00 00 00
 64 64 00 00 01 10 00 00 00 00 00
 CFG-NMEA - 06 17 0C 00 00 23 00 02 00 00 00 00 00 00 00 00 00
 CFG-PM2 - 06 3B 2C 00 01 06 00 00 00 90 02 00 E8 03 00 00 10 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 03 00 86 02 00 00 FE 00 00 00 64 40 01 00
 CFG-PRT - 06 00 14 00 00 00 00 00 84 00 00 00 00 00 00 07 00 03 00 00 00 00 00 00
 CFG-PRT - 06 00 14 00 01 00 00 00 C0 08 00 00 00 84 03 00 07 00 03 00 00 00 00 00
 CFG-PRT - 06 00 14 00 02 00 00 00 C0 38 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 CFG-PRT - 06 00 14 00 03 00 00 00 00 00 00 00 00 00 00 07 00 03 00 00 00 00 00 00
 CFG-PRT - 06 00 14 00 04 00 00 00 00 32 00 00 00 00 00 00 00 07 00 03 00 00 00 00 00 00
 CFG-RATE - 06 08 06 00 C8 00 01 00 01 00
 CFG-RINV - 06 34 18 00 00 4E 6F 74 69 63 65 3A 20 6E 6F 20 64 61 74 61 20 73 61 76 65 64 21 00
 CFG-RXM - 06 11 02 00 08 00
 CFG-SBAS - 06 16 08 00 01 03 03 00 D1 A2 06 00
 CFG-TP5 - 06 31 20 00 00 00 00 00 32 00 00 00 04 00 00 00 01 00 00 00 48 E8 01 00 A0 86 01 00 00 00 00 00 00 FF 00
 00 00
 CFG-TP5 - 06 31 20 00 01 00 00 00 32 00 00 00 04 00 00 00 01 00 00 00 48 E8 01 00 A0 86 01 00 00 00 00 00 00 FE 00
 00 00
 CFG-USB - 06 1B 6C 00 46 15 A7 01 00 00 00 64 00 02 01 75 2D 62 6C 6F 78 20 41 47 20 2D 20 77 77 77 2E 75
 2D 62 6C 6F 78 2E 63 6F 6D 00 00 00 00 00 75 2D 62 6C 6F 78 20 37 20 2D 20 47 50 53 2F 47 4E 53 53 20 52 65
 63 65 69 76 65 72 00
 00 00 00 00

G.2 Rover

MON-VER - 0A 04 BE 00 31 2E 30 30 20 28 35 39 38 34 33 29 00
 30 30 30 37 30 30 30 30 00 00 31 2E 30 30 20 28 35 39 38 34 32 29 00
 00 00 50 52 4F 54 56 45 52 20 31 34 2E 30 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 46 49 53 20 30 78
 45 46 34 30 31 35 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4D 4F 44 20 4E 45 4F 2D 37 4E 2D 30 00
 00
 00 00 00 00 00 00 00 00 00
 CFG-ANT - 06 13 04 00 1B 00 F0 B9
 CFG-DAT - 06 06 02 00 00 00
 CFG-GNSS - 06 3E 24 00 00 16 16 04 00 04 FF 00 01 00 00 00 01 01 03 00 01 00 00 00 05 00 03 00 01 00 00 00 06
 08 FF 00 00 00 00 00
 CFG-INF - 06 02 0A 00 00 00 00 00 00 00 00 00 00
 CFG-INF - 06 02 0A 00 01 00 00 00 87 87 87 87 87 87
 CFG-INF - 06 02 0A 00 03 00 00 00 00 00 00 00 00 00
 CFG-ITFM - 06 39 08 00 F3 AC 62 2D 1E 03 00 00
 CFG-LOGFILTER - 06 47 0C 00 01 00
 CFG-MSG - 06 01 08 00 0B 30 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0B 32 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0B 33 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0B 31 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0B 01 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0B 00 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 21 08 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 0B 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 09 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 02 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 06 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 07 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 21 00 00 00 00 00 00 00
 CFG-MSG - 06 01 08 00 0A 08 00 00 00 00 00 00 00 00

CFG-MSG - 06 01 08 00 01 60 00 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 22 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 31 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 04 00 01 00 00 00 00
CFG-MSG - 06 01 08 00 01 01 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 02 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 07 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 32 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 06 00 01 00 00 00 00
CFG-MSG - 06 01 08 00 01 03 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 30 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 20 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 21 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 11 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 01 12 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 02 20 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 0D 03 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 0D 01 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 0D 06 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F0 00 01 00 01 01 01 01
CFG-MSG - 06 01 08 00 F0 01 01 00 01 01 01 01
CFG-MSG - 06 01 08 00 F0 02 01 01 01 01 01 01
CFG-MSG - 06 01 08 00 F0 03 01 00 01 01 01 01
CFG-MSG - 06 01 08 00 F0 04 01 00 01 01 01 01
CFG-MSG - 06 01 08 00 F0 05 01 00 01 01 01 01
CFG-MSG - 06 01 08 00 F0 06 00 01 00 00 00 00
CFG-MSG - 06 01 08 00 F0 07 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F0 08 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F0 09 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F0 0A 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F0 0D 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F1 00 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F1 03 00 00 00 00 00 00
CFG-MSG - 06 01 08 00 F1 04 00 00 00 00 00 00
CFG-NAV5 - 06 24 24 00 FF FF 00 03 00 00 00 00 10 27 00 00 05 00 FA 00 FA 00 64 00 2C 01 00 3C 00 00 00 00 00
00 00 00 00 00 00
CFG-NAVX5 - 06 23 28 00 00 00 FF FF 0F 00 00 00 03 02 03 16 07 00 00 01 00 00 9B 06 00 00 00 00 01 01 00 00 00
64 64 00 00 01 10 00 00 00 00 00
CFG-NMEA - 06 17 0C 00 00 23 00 02 00 00 00 00 00 00 00 00
CFG-PM2 - 06 3B 2C 00 01 06 00 00 00 90 02 00 E8 03 00 00 10 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03 00 86 02 00 00 FE 00 00 00 64 40 01 00
CFG-PRT - 06 00 14 00 00 00 00 00 84 00 00 00 00 00 00 00 07 00 03 00 00 00 00 00
CFG-PRT - 06 00 14 00 01 00 00 00 C0 08 00 00 00 96 00 00 07 00 03 00 00 00 00 00
CFG-PRT - 06 00 14 00 02 00 00 00 C0 38 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
CFG-PRT - 06 00 14 00 03 00 00 00 00 00 00 00 00 00 00 00 07 00 03 00 00 00 00 00
CFG-PRT - 06 00 14 00 04 00 00 00 00 32 00 00 00 00 00 00 00 07 00 03 00 00 00 00 00
CFG-RATE - 06 08 06 00 C8 00 01 00 01 00
CFG-RINV - 06 34 18 00 00 4E 6F 74 69 63 65 3A 20 6E 6F 20 64 61 74 61 20 73 61 76 65 64 21 00
CFG-RXM - 06 11 02 00 08 00
CFG-SBAS - 06 16 08 00 01 03 03 00 D1 A2 06 00
CFG-TP5 - 06 31 20 00 00 00 00 32 00 00 00 04 00 00 00 01 00 00 00 48 E8 01 00 A0 86 01 00 00 00 00 00 FF 00
00 00
CFG-TP5 - 06 31 20 00 01 00 00 00 32 00 00 00 04 00 00 00 01 00 00 00 48 E8 01 00 A0 86 01 00 00 00 00 00 FE 00
00 00
CFG-USB - 06 1B 6C 00 46 15 A7 01 00 00 00 64 00 02 01 75 2D 62 6C 6F 78 20 41 47 20 2D 20 77 77 77 2E 75
2D 62 6C 6F 78 2E 63 6F 6D 00 00 00 00 00 75 2D 62 6C 6F 78 20 37 20 2D 20 47 50 53 2F 47 4E 53 53 20 52 65
63 65 69 76 65 72 00
00 00 00 00

H. Implemented code

```
%% TREBALL DE FI DE GRAU - ESTUDI D'UN GPS DIFERENCIAL PER A UAVs

%-----
%-----  

%This software has been developed for academic purposes and its  

%distribution and modification is totally free.  

%The goal of this code is to allow a posteriori DGPS corrections  

%interpreting binary messages from U-Center (a free software developed  

%by  

%U-Blox) provided by two U-Blox GPS receivers.  

%-----  

%-----  

%  

%This script initialize the whole process and it is where the user  

%must  

%define some parameters such true base station location, base station  

%and  

%rover binary data text file, correction mode, some settings and  

%post-processing.

%% ***** INITIALIZE DGPS PROCESS
*****  

%% Clean up the environment
=====  

clear; close all; clc;  

disp('***** DEBUG DGPS FILE *****')
disp(' ')
disp(' ')  

format ('compact');
format ('long', 'g');  

%% Add files to path
=====  

addpath include
addpath include/correction
addpath include/export
addpath include/getFunctions
addpath include/graphs
addpath include/positioning
addpath include/post_processing
addpath include/timing
addpath include/utils
```

```
%% Differential GPS correction
=====
% Set parameters -----
-----
% TRUE POSITION:
geoidHeight = 49.505;

truePos.lat.deg = 41;
truePos.lat.min = 22;
truePos.lat.sec = 42.84;
truePos.lat.N = 'N';

truePos.lon.deg = 2;
truePos.lon.min = 3;
truePos.lon.sec = 5.62;
truePos.lon.E = 'E';

truePos.h = 35;

truePos.h = truePos.h + geoidHeight;

BSfileName = 'solBaseStation2.txt';

% ROVER data file (.txt)
RfileName = 'solRover2.txt';

% Get true position in geodetic and ECEF coordinate system -----
-----
trueECEF = getTrueECEF(truePos);
trueGEO = getTrueGeo(truePos);

% Get messages, data and compute DGPS correction -----
-----
[rover, baseStation, SV, GNSSmessages, GNSSparameters] =
DGPS(BSfileName, RfileName, trueECEF, 0, 0, 'NavSolR', 1e3, 36);

%% POST-PROCESSING
=====
% statistics = postProcessing(SV, baseStation, rover, trueGEO);
% roverTruePos = truePos;
% statistics.roverDeviation = deviationMap(rover.final_pos,
roverTruePos);
% compareRangeResiduals(SV)
% comparePDOP(rover.DOP, rover.final_DOP);
% checkRoverInterpolation(baseStation, rover);
% checkSVInterpolation(SV, 'rRR');
% plotPath(rover)

%% PLOTS AND VISUALIZATION
=====
disp('===== PLOTS AND VISUALIZATION =====')
```

```
% showSV(SV, rover, baseStation, trueECEF)
% view(90, trueGEO.lat)

% figure, grid, hold on
% for i = 1:length(baseStation.posECEF)
% % plot(baseStation.posECEF(i).time, baseStation.posECEF(i).x,
% '.r')
% % plot(baseStation.posECEF(i).time, baseStation.posECEF(i).y,
% '.b')
% plot(baseStation.posECEF(i).time, baseStation.posECEF(i).z,
% '.g')
% end
% hold off

% figure, grid, hold on
% for i = 1:length(baseStation.geoPos)
% % plot(baseStation.geoPos(i).time, baseStation.geoPos(i).lat,
% '.r')
% plot(baseStation.geoPos(i).time, baseStation.geoPos(i).lon,
% '.b')
% % plot(baseStation.geoPos(i).time, baseStation.geoPos(i).height,
% '.g')
% end
% hold off

% plotPosComp(SV, rover, trueECEF, trueGEO)

% figure, hold on
% title('BASE STATION - DGPS Correction vs. time'), xlabel('Time [s]'), ylabel('Correction [m]'), grid
% for id = 1:length(SV)
% if ~isempty(SV(id).BaseStation)
% color = getColor(id, length(SV));
% for i = 1:length(SV(id).BaseStation)
% plot(SV(id).BaseStation(i).time,
SV(id).BaseStation(i).correction, '.', 'Color', color)
% end
% end
% hold off
%
%
% figure, hold on
% title('ROVER - DGPS Corrected Ranges vs. time'), xlabel('Time [s]'),
ylabel('Correction [m]'), grid
% for id = 1:length(SV)
% if ~isempty(SV(id).Rover)
% color = getColor(id, length(SV));
% for i = 1:length(SV(id).Rover)
% plot(SV(id).Rover(i).time,
SV(id).Rover(i).corrected_pseudorange, '.', 'Color', color)
% end
% end
% hold off
%
% n = 1;
% drawSQ(SV, activeSV(n).SVID, rover.final_pos(n), trueECEF, n)
% SV = computeRangeResiduals(SV, activeSV, rover.final_pos);
```

```
baseStation.geoPos = ECEF2geodeticArray(baseStation.posECEF);  
rover.geoPos = ECEF2geodeticArray(rover.posECEF);  
  
%% EXPORT KML FILE  
=====  
exportKML( 'TestSolitari BaseStation2', 'Uncorrected Path',  
baseStation.geoPos, 'Corrected Path', rover.final_pos, 'True  
Location', trueGEO, geoidHeight )  
exportKML( 'NavSolR - testSolitari', 'Uncorrected Path', rover.geoPos,  
'Corrected Path', rover.final_pos, 'True Location', trueGEO,  
geoidHeight )  
  
disp('----- THE END -----')
```

H.1 DGPS.m

Next, it is attached the implemented code base on *MATLAB*. Some of functions used to plot have been omitted since it has been considered that they are not of interest.

```

function [ rover, baseStation, SV, GNSSmessages, GNSSparameters ] =
DGPS(BSfileName, RfileName, trueECEF, Rclean, BSclean, correctionMode,
radius, nSatellites )
%DGPS coordinates all the processes necessary to implement the whole
DGPS
%process, since reading the files until recompute the final position.

[%[ rover, baseStation, SV, GNSSmessages, GNSSparameters ] =
DGPS(BSfileName, RfileName, trueECEF, Rclean, BSclean, correctionMode,
radius, nSatellites )
%
%    INPUTS:
%
%        BSfileName      - Name of the text file that contains Base
Station
%
%        RfileName       - Name of the text file that contains Rover
binary
%
%        trueECEF        - True Base Station receiver location in ECEF
coordinantes.
%
%        Rclean          - Cleaning flag of the Rover data file.
%        1: contains row number and text enconding,
must be
%
%        BSclean         - Cleaning flag of the Base Station data file.
%        1: contains row number and text enconding,
must be
%
%        correctionMode - Specifies the correction mode.
%
%        o OFFSET: Offset correction
%        o COMMON: Classical DGPS correction
%        o NavSolR: Navigation Solution - Real SVs
%        o NavSolV: Navigation Solution - Virtual
SVs
%
%        radius          - Is the radius of the sphere distribution
where
%
mode.
%
%        nSatellites     - Number of virtual SVs distributed along a
sphere,
%
an
%
%
%    OUTPUTS:
%
%        rover          - structure containing all the parameters
related
%
%        baseStation     - structure containing all the parameters
related
%
with the base station.

```

```
%      SV          - structure containing all the parameters
related
%
%      GNSSmessages - structure containing all the decoded
messages
%
%      GNSSparameters - structure containing all the parameters
extracted
%
%                  from GNSS messages.

%% Check all inputs are correct
=====

BS_ID = fopen(BSfileName);
if BS_ID ~= -1
    fclose(BS_ID);
    clear BS_ID
else
    error('Cannot find Base Station data file.')
end

R_ID = fopen(RfileName);
if R_ID ~= -1
    fclose(R_ID);
    clear R_ID
else
    error('Cannot find Rover data file.')
end

if rem(sqrt(nSatellites)-1, 1) ~= 0
    error('Number of SVs must be the square of an integer')
end

if ~strcmp(correctionMode, 'OFFSET') || strcmp(correctionMode,
'COMMON') || strcmp(correctionMode, 'NavSolR') ||
strcmp(correctionMode, 'NavSolV'))
    error('Invalid Correction Mode. (OFFSET, COMMON, NavSolR,
NavSolV)')
end

%% Base Station
=====

disp('===== BASE STATION =====')
disp(' ')

% Decode binary messages from data file -----
-----
GNSSmessages.BaseStation = messageDecoder(BSfileName, BSclean);

% Get GNSS parameters from decoded messages -----
-----
GNSSparameters.BaseStation =
getGNSSparameters(GNSSmessages.BaseStation, correctionMode);

for f = 1:length(GNSSparameters.BaseStation.fields)
    field = GNSSparameters.BaseStation.fields{f};

    switch field
        case 'receiver'
```

```

        baseStation = GNSSparameters.BaseStation.receiver;
    case 'DOP'
        baseStation.DOP = GNSSparameters.BaseStation.DOP;
        baseStation.fields{length(baseStation.fields)+1} = 'DOP';
    case 'activeSV'
        baseStation.activeSV =
GNSSparameters.BaseStation.activeSV;
        baseStation.fields{length(baseStation.fields)+1} =
'activeSV';
    case 'SV'
        SV = GNSSparameters.BaseStation.SV;
    otherwise
        warning('Ignored %s field from BaseStation.', field);
    end
end
clear field

% Get position in both used coordinate systems if necessary -----
-----
if isempty(baseStation.geoPos) && isempty(baseStation.posECEF)
    error('Navigation solution is missing in base station.')
elseif ~isempty(baseStation.geoPos) && isempty(baseStation.posECEF)
    baseStation.posECEF = geodetic2ECEFarray(baseStation.geoPos);
    disp('Base station ECEF position calculated.')
elseif isempty(baseStation.geoPos) && ~isempty(baseStation.posECEF)
    baseStation.geoPos = ECEF2geodeticArray(baseStation.posECEF);
else
    disp('Base station contains position in geodetic and ECEF
coordinate system.')
end
if isfield(GNSSparameters.BaseStation, 'SV')
    SV = GNSSparameters.BaseStation.SV;
else
    disp('Satellites has not been tracked.')
end
baseStation.distance = getDistance(baseStation.posECEF, trueECEF);
baseStation.fields{length(baseStation.fields)+1} = 'distance';

%% Rover
=====
disp('===== ROVER =====')
disp(' ')

% Decode binary messages from data file -----
-----
GNSSmessages.Rover = messageDecoder(RfileName, Rclean);

% Get GNSS parameters from decoded messages -----
-----
GNSSparameters.Rover = getGNSSparameters(GNSSmessages.Rover,
correctionMode);

for f = 1:length(GNSSparameters.Rover.fields)
    field = GNSSparameters.Rover.fields{f};

    switch field
        case 'receiver'
            rover = GNSSparameters.Rover.receiver;
        case 'DOP'

```

```

        rover.DOP = GNSSparameters.Rover.DOP;
        rover.fields{length(rover.fields)+1} = 'DOP';
    case 'activeSV'
        rover.activeSV = GNSSparameters.Rover.activeSV;
        rover.fields{length(rover.fields)+1} = 'activeSV';
    case 'SV'
        for s =
1:length(GNSSparameters.Rover.SV(rover.activeSV(1).SVID(1)).fields)
            SVfield =
GNSSparameters.Rover.SV(rover.activeSV(1).SVID(1)).fields{s};

            switch SVfield
                case 'range_residuals'
                    for id = 1:length(SV)
                        SV(id).rover_residuals =
GNSSparameters.Rover.SV(id).range_residuals;
                        SV(id).fields{length(SV(id).fields)+1} =
'rover_residuals';
                    end
                case 'pos'
                    warning('Ignored SV position from Rover
messages.')
                end
            end
        otherwise
            warning('Ignored %s field from Rover.', field);
        end
    end
clear field SVfield

% Get position in both used coordinate systems if necessary -----
-----
if isempty(rover.posECEF) && ~isempty(rover.geoPos)
    rover.posECEF = geodetic2ECEFarray(rover.geoPos);
    warning('Coordinates ECEF obtained from converting geodetic
position.')
end
if isempty(rover.geoPos) && isempty(rover.posECEF)
    error('Navigation solution is missing in base station.')
elseif ~isempty(rover.geoPos) && isempty(rover.posECEF)
    rover.posECEF = geodetic2ECEFarray(rover.geoPos);
    disp('Base station ECEF position calculated.')
elseif isempty(rover.geoPos) && ~isempty(rover.posECEF)
    rover.geoPos = ECEF2geodeticArray(rover.posECEF);
else
    disp('Base station contains position in geodetic and ECEF
coordinate system.')
    disp(' ')
end

%% Unify Active SV
=====
if isfield(baseStation, 'activeSV')
    activeSV_BS = baseStation.activeSV;
    activeSV_R = rover.activeSV;
    warning('Active SV taken from Base Station messages ignoring
Rover.')
else
    activeSV = [];
end

```

```

%% Unify Rover and Base Station timings
=====
if ~exist('SV', 'var')
    SV = [];
end

[rover, baseStation, SV, activeSV_BS, activeSV_R] =
unifyTimings(rover, baseStation, SV, activeSV_BS, activeSV_R,
correctionMode);

%% Compute DGPS correction
=====
disp('===== DGPS CORRECTION =====')
disp(' ')

% Compute Correction -----
-----
switch correctionMode
    case 'OFFSET' % ----- OFFSET -----
        disp('Correction Mode: OFFSET')
        rover = computeCorrectionOFFSET(rover, baseStation, trueECEF);
        if ~exist('SV', 'var')
            SV = [];
        end
    case 'COMMON' % ----- COMMON -----
        disp('Correction Mode: COMMON')
        [SV, rover, activeSV] = computeCorrectionCOMMON(SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R);

        % Recompute position after correction -----
        rover = computePosition(SV, rover, activeSV, correctionMode);

    case 'NavSolR' % ----- NavSol: REAL SVs -----
        disp('Correction Mode: RANGES - REAL SVs')
        [SV, rover, activeSV] = computeCorrectionReal(SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R);

        % Recompute position after correction -----
        rover = computePosition(SV, rover, activeSV, correctionMode);

    case 'NavSolV' % ----- NavSol: GENERATED SVs -----
        disp('Correction Mode: RANGES - VIRTUAL SVs')
        if strcmp(correctionMode, 'NavSolV') && (isempty(radius) ||
isempty(nSatellites))
            radius = 1e3;
            nSatellites = 36;
            disp('Information about generated SVs is missing. The
following parameters has been taken:')
            fprintf('Radius: %.0f\nnSatellites: %.0f\n', radius,
nSatellites)
        end
end

```

```

[SV, activeSV] = generateSV(trueECEF, radius, nSatellites,
rover.posECEF);
[SV, rover] = computeCorrectionVirtual(SV, rover, baseStation,
trueECEF);

% Recompute position after correction -----
-----
rover = computePosition(SV, rover, activeSV, correctionMode);

otherwise
    error('Undefined correction mode.')
end

baseStation.distance = getDistance(baseStation.posECEF, trueECEF);

if ~strcmp(correctionMode, 'OFFSET')
    SV = computeRangeResiduals(SV, activeSV, rover.final_pos);
end

end

```

H.2 Include

H.2.1 messageDecoder.m

```

function [ GNSSmessages ] = messageDecoder( fileName, clean )
%messageDecoder decodes all binary messages from U-Center stored in
the
%file 'fileName'.

 %[ GNSSmessages ] = messageDecoder( fileName, clean )
%
% INPUTS:
%     fileName           - Name of the file that contains the binary
data.
%     clean              - Cleaning flag. 1: each row must be cleaned.
%
% OUTPUT:
%     GNSSmessages      - structure containing all the decoded
messages
%                           stored in 'fileName'.
%-----
-----


%% Open .txt or .ubx data file
=====
fileID = fopen(fileName);

%% Get Payload from GPS messages
=====

% Initialize variables -----
-----
UBX.AID_EPH = [];
UBX.AID_HUI = [];

```

```

UBX.NAV_POSECEF = [];
UBX.NAV_POSLLH = [];
UBX.NAV_SOL = [];
UBX.NAV_DOP = [];

NMEA.GPGGA = [];
NMEA.GPGRS = [];
NMEA.GPGSA = [];

% Initialize auxiliar variables -----
-----
e = 0;
p = 0;
l = 0;
s = 0;
r = 0;
g = 0;
a = 0;
n = 1;
u = 1;
m = 0;
v = 0;
lineNumber = 0;
UBXcounter = 0;
NMEAcounter = 0;
binaryData = [];
GNSSmessages = [];
NMEAnum = [];
UBXnum = [];

%% Collect data from file line-by-line
=====

disp('----- MESSAGE DECODER -----')
fprintf('Decoding: %s\n', fileName);
disp('Reading...')

while ~feof(fileID)

    % Get row -----
-----
    row = fgetl(fileID);

    if clean
        row = cleanRow(row);
    else
        row(strfind(row, ' ')) = [];
    end
    lineNumber = lineNumber+1;

    % Make sure all letters are uppercase -----
-----
    row = upper(row);
    binaryData = [binaryData row];
end
fclose(fileID);

%% Find message headers location -----
-----
NMEAloc = strfind(binaryData, '244750');

```

```

UBXloc = strfind(binaryData, 'B562');

%% Numerate messages -----
-----
if isempty(UBXloc) && isempty(NMEAloc) % ----- NO MESSAGES -----
-----
    error('There are no UBX nor NMEA sentences.\n')
elseif ~isempty(UBXloc) && isempty(NMEAloc) % ----- ONLY UBX -----
-----
    fprintf('There are only UBX sentences.\n')
    UBXnum = 1:length(UBXloc);
elseif isempty(UBXloc) && ~isempty(NMEAloc) % ----- ONLY NMEA -----
-----
    fprintf('There are only NMEA sentences.\n')
    NMEAnum = 1:length(NMEAloc);
else % ----- NMEA AND UBX MESSAGES -----
-----
    fprintf('Data encoding uses NMEA and UBX protocols.\n')

    for m = 1:(length(NMEAloc)+length(UBXloc))
        if n <= length(NMEAloc) && u <= length(UBXloc)
            if NMEAloc(n) < UBXloc(u)
                NMEAnum(n) = m;
                n = n+1;
                continue
            end
            if UBXloc(u) < NMEAloc(n)
                UBXnum(u) = m;
                u = u+1;
                continue
            end
        else
            if n > length(NMEAloc)
                UBXnum = [UBXnum m:m+length(UBXloc)-u];
                break
            end
            if u > length(UBXloc)
                NMEAnum = [NMEAnum m:m+length(NMEAloc)-n];
                break
            end
        end
    end
end

disp('All sentences read.')
disp(' ')

%% Read extracted data
=====
%% UBX Protocol -----
-----
disp('Decoding UBX Protocol...')
while UBXcounter < length(UBXloc)

    %% Get next message
=====
    UBXcounter = UBXcounter+1;
    headLoc = UBXloc(UBXcounter);

```

```

%% Assign message number
=====
numMSG = UBXnum(UBXcounter);

%% Read message
=====
% Set cursor to the header location-----
c = headLoc;

% Read message -----
header = binaryData(c:c+3); c = c+4;
class = binaryData(c:c+1); c = c+2;
id = binaryData(c:c+1); c = c+2;
Length = hex2dec(binaryData(c:c+1)); c = c+2;

c = c+2;

% Check if the last message is partially empty -----
if UBXcounter == length(UBXloc) && (c+Length*2-1) >
length(binaryData)
    continue
else
    payload = binaryData(c:c+Length*2-1);
end

%% Classify message
=====
switch class
    case '0B' %AID -----
        switch id
            case '31' %EPH
                if Length == 104
                    e = e+1;
                    UBX.AID_EPH(e).payload = getAID_EPH(payload);
                    UBX.AID_EPH(e).numMSG = numMSG;
                elseif Length ~= 8
                    error('AID-EPH invalid length. Must be 104 or
8 bytes.')
                end
            end
        case '01' %NAV -----
            switch id
                case '01' %POSECEF
                    if Length == 20
                        p = p+1;
                        UBX.NAV_POSECEF(p).payload =
getNAV_POSECEF(payload);
                        UBX.NAV_POSECEF(p).numMSG = numMSG;
                    elseif Length ~= 0
                        error('NAV-POSECEF invalid length. Must be 20
or 0 bytes.')
                    end
                case '02' %POSLLH
                    if Length == 28
                        l = l+1;
                end
            end
        end
    end
end

```

```

        UBX.NAV_POSLLH(l).payload =
getNAV_POSLLH(payload);
        UBX.NAV_POSLLH(l).numMSG = numMSG;
    end
case '06' %SOL
    if Length == 52
        s = s+1;
        UBX.NAV_SOL(s).payload = getNAV_SOL(payload);
        UBX.NAV_SOL(s).numMSG = numMSG;
    end
end
end

end
disp('UBX messages successfully decoded.')
disp(' ')

%% NMEA Protocol -----
-----
disp('Decoding NMEA Protocol...')

while NMEAcounter < length(NMEAloc)
    %% Get next message
=====
NMEAcounter = NMEAcounter+1;
headLoc = NMEAloc(NMEAcounter);

    %% Assign message number
=====
numMSG = NMEAnum(NMEAcounter);

    %% Read message
=====
    % Set cursor to the header location-----
    c = headLoc;

    % Read message -----
header = binaryData(c:c+11); c = c+14;

%     nextMsgLoc = min(NMEAloc(NMEAcounter+1), UBXloc(UBXcounter+1));
nextMsgLoc = [NMEAloc(NMEAnum == numMSG+1) UBXloc(UBXnum ==
numMSG+1)];
if isempty(nextMsgLoc)
    payload = binaryData(c:end);
else
    payload = binaryData(c:nextMsgLoc-1);
end

    %% Classify message
=====
switch header
    case '244750474741'    % $GPGGA -----
        g = g+1;
        NMEA.GPGGA(g).payload = getGPGGA(payload);
        NMEA.GPGGA(g).numMSG = numMSG;
-----
```

```

case '244750475253'    % $GPGRS -----
-----
r = r+1;
NMEA.GPGRS(r).payload = getGPGRS(payload);
NMEA.GPGRS(r).numMSG = numMSG;

case '244750475341'    % $GPGSA -----
-----
a = a+1;
NMEA.GPGSA(a).payload = getGPGSA(payload);
NMEA.GPGSA(a).numMSG = numMSG;

case '244750475356'    % $GPGSV -----
-----
v = v+1;
NMEA.GPGSV(v).payload = getGPGSV(payload);
NMEA.GPGSV(v).numMSG = numMSG;
end
end
disp('NMEA messages successfully decoded.')

fprintf('%s successfully decoded.\n', fileName)
disp(' ')
%% Export decoded messages
=====
GNSSmessages = [];
if ~isempty(UBX)
    GNSSmessages.UBX = UBX;
end
if ~isempty(NMEA)
    GNSSmessages.NMEA = NMEA;
end
end

```

H.2.2 trackSV

```

function [ SV ] = trackSV( UBX_AID_EPH, SV, nSatellites )
%trackSV coordinates the processes required to track a SV. Reads the
input
%AID-EPH message contained in the navigation message subframes 2 & 3
plus
%clock correction in subframe 1, applies clock corrections and compute
SV
%final position. Returns ephemerides and position of the current
active SV.

[% SV ] = trackSV( UBX_AID_EPH, SV, nSatellites )

% INPUTS:
%     UBX_AID_EPH      - UBX AID-EPH message.
%     SV                - SV structure.
%     nSatellites       - Number of total active satellites.
%
% OUTPUT:
%     SV                - Updated SV structure.
%-----
-----
```

```

disp('----- SATELLITES TRACKER -----')
disp('Tracking satellites...')

%% Check messages existance -----
if isempty(UBX_AID_EPH)
    error('No UBX AID-EPH messages found.')
end

%% Initiliaze variable
=====
% Initialize SV variable -----
for id = 1:nSatellites
    SV(id).counter = 0;
    SV(id).ephemeris = [];
    SV(id).pos = [];
end

% Initialize auxiliar variables -----
messages = length(UBX_AID_EPH);

%% Track Space Vehicles one by one
=====
for n = 1:messages

    % Get payload, time and message number from messages -----
    AID_EPH = UBX_AID_EPH(n).payload;
    transmitTime = UBX_AID_EPH(n).TOW;

    % Identify SV -----
    id = AID_EPH.ID;
    SV(id).counter = SV(id).counter+1;

    % Get SV's ephemeris and position -----
    if SV(id).counter == 1
        SV(id).ephemeris = getEphemeris(AID_EPH);
        SV(id).pos = getSVposition(SV(id).ephemeris(SV(id).counter),
transmitTime);
        SV(id).pos.numMSG = UBX_AID_EPH(n).numMSG;
    else
        SV(id).ephemeris(SV(id).counter) = getEphemeris(AID_EPH);
        SV(id).pos(SV(id).counter) = getSVposition(
SV(id).ephemeris(SV(id).counter), transmitTime );
        SV(id).pos(SV(id).counter).numMSG = UBX_AID_EPH(n).numMSG;
    end
end

disp('All satellites succesfully tracked')
disp(' ')
end

```

H.2.3 correction

H.2.3.1 computeCorrectionOFFSET.m

```

function [ rover ] = computeCorrectionOFFSET( rover, baseStation,
trueECEF )
%computeCorrectionOFFSET computes position domain corrections -
OFFSET.

[%[ rover ] = computeCorrectionOFFSET( rover, baseStation, trueECEF )
%
% INPUTS:
%     rover           - rover structure.
%     baseStation     - base station structure.
%     trueECEF        - true base station location in ECEF
coordinates.
%
% OUTPUT:
%     rover          - Corrected and updated rover position.
%-----
-----
```

```

disp('Computing correction and new position...')

%% Initialize variables
=====

for n = 1:length(baseStation.posECEF)
    time_arrayBS(n) = baseStation.posECEF(n).time;
end

for n = 1:length(rover.posECEF)
    time_arrayR(n) = rover.posECEF(n).time;
end

j = 0;

%% Fix Time
=====
for i = 1:length(baseStation.posECEF)
    if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time
        j = j+1;
        [ lowTime, upTime, lowLoc, upLoc ] =
closestValue(baseStation.posECEF(i).time, time_arrayR);
        if lowTime ~= upTime
            rover.DGPSpos(j).x = rover.posECEF(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).x - rover.posECEF(lowLoc).x);
            rover.DGPSpos(j).y = rover.posECEF(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).y - rover.posECEF(lowLoc).y);
            rover.DGPSpos(j).z = rover.posECEF(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).z - rover.posECEF(lowLoc).z);
        else
            rover.DGPSpos(j).x = rover.posECEF(lowLoc).x;
            rover.DGPSpos(j).y = rover.posECEF(lowLoc).y;
```

```

        rover.DGPSpos(j).z = rover.posECEF(lowLoc).z;
    end
    rover.DGPSpos(j).time = baseStation.posECEF(i).time;
end

%% Compute correction in ECEF coordinate system
=====
for i = 1:length(rover.DGPSpos)

    rover.final_pos(i).time = rover.DGPSpos(i).time;

    %% Correction
=====
    rover.correction(i).x = trueECEF.x - baseStation.posECEF(i).x;
    rover.correction(i).y = trueECEF.y - baseStation.posECEF(i).y;
    rover.correction(i).z = trueECEF.z - baseStation.posECEF(i).z;

    %% Apply correction
=====
    rover.final_pos(i).x = rover.DGPSpos(i).x + rover.correction(i).x;
    rover.final_pos(i).y = rover.DGPSpos(i).y + rover.correction(i).y;
    rover.final_pos(i).z = rover.DGPSpos(i).z + rover.correction(i).z;

end

%% Get geodetic coordinates from conversion
=====
[ geodetic ] = ECEF2geodeticArray( rover.final_pos );
for i = 1:length(rover.final_pos)
    rover.final_pos(i).lat = geodetic(i).lat;
    rover.final_pos(i).lon = geodetic(i).lon;
    rover.final_pos(i).h = geodetic(i).h;
end

disp('DGPS correction successfully completed.')
disp(' ')
end

```

H.2.3.2 computeCorrectionCOMMON.m

```

function [ SV, rover, activeSV ] = computeCorrectionCOMMON( SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R )
%computeCorrectionCLASSIC computes the COMMON DGPS correction.

%[ SV, rover, activeSV ] = computeCorrectionReal( SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R )
%
% INPUTS:
%   SV           - Space Vehicles structure.
%   rover        - rover structure.
%   baseStation  - base station structure.
%   trueECEF     - true base station receiver location in ECEF
%                   coordinates.
%   activeSV_BS  -
%   activeSV_R   - active SVs according to rover receiver.
%
% OUTPUTS:
%   SV           - updated with pseudoranges SV structure.
%   rover        - time adapted and updated rover structure.
%   activeSV     - active SVs.
%-----
-----

disp('Computing ranges and DGPS correction...')

%% Initialize variables
=====

c = 299792458; %m/s

for n = 1:length(baseStation.posECEF)
    time_arrayBS(n) = baseStation.posECEF(n).time;
end

for n = 1:length(rover.posECEF)
    time_arrayR(n) = rover.posECEF(n).time;
end

for n = 1:length(activeSV_BS)
    activeSV_BSarray(n) = activeSV_BS(n).time;
end

for n = 1:length(activeSV_R)
    activeSV_Rarray(n) = activeSV_R(n).time;
end

for id = 1:length(SV)
    for n = 1:length(SV(id).pos)
        SV_array(id).time(n) = SV(id).pos(n).time;
    end
end

for id = 1:length(SV)
    for n = 1:length(SV(id).range_residuals)

```

```

        SV_BSRR(id).time(n) = SV(id).range_residuals(n).time;
    end
end

for id = 1:length(SV)
    for n = 1:length(SV(id).rover_residuals)
        SV_RRR(id).time(n) = SV(id).rover_residuals(n).time;
    end
end

j = 0;
k = 0;
%% Fix Time
=====
for i = 1:length(baseStation.posECEF)
    if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time

        % Determining active SV -----
        -----
        k = k+1;
        activeSV(k).SVID = unifyActiveSV(baseStation.posECEF(i).time,
activeSV_BSarray, activeSV_Rarray, activeSV_BS, activeSV_R);
        activeSV(k).time = baseStation.posECEF(i).time;

        if ~isempty(activeSV)
            j = j+1;

            %% Coordinate rover position timing -----
            -----
            [lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, time_arrayR);
            if lowTime ~= upTime
                rover.DGPSpos(j).x = rover.posECEF(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).x - rover.posECEF(lowLoc).x);
                rover.DGPSpos(j).y = rover.posECEF(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).y - rover.posECEF(lowLoc).y);
                rover.DGPSpos(j).z = rover.posECEF(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).z - rover.posECEF(lowLoc).z);
            else
                rover.DGPSpos(j).x = rover.posECEF(lowLoc).x;
                rover.DGPSpos(j).y = rover.posECEF(lowLoc).y;
                rover.DGPSpos(j).z = rover.posECEF(lowLoc).z;
            end
            rover.DGPSpos(j).time = baseStation.posECEF(i).time;

            for m = 1:length(activeSV(k).SVID)
                id = activeSV(k).SVID(m);

                %% Coordinate SVs position timing -----
                -----
                [lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, SV_array(id).time);

                if lowTime ~= upTime

```

```

        SV(id).DGPSpos(j).x = SV(id).pos(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).x - SV(id).pos(lowLoc).x);
        SV(id).DGPSpos(j).y = SV(id).pos(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).y - SV(id).pos(lowLoc).y);
        SV(id).DGPSpos(j).z = SV(id).pos(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).z - SV(id).pos(lowLoc).z);
    else
        SV(id).DGPSpos(j).x = SV(id).pos(lowLoc).x;
        SV(id).DGPSpos(j).y = SV(id).pos(lowLoc).y;
        SV(id).DGPSpos(j).z = SV(id).pos(lowLoc).z;
    end
    SV(id).DGPSpos(j).time = baseStation.posECEF(i).time;

    %% Coordinate SVs BS range residuals timing -----
-----
[lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, SV_BSRR(id).time);

    if lowTime ~= upTime
        SV(id).DGPSbsRR(j).meters =
SV(id).range_residuals(lowLoc).meters + ((baseStation.posECEF(i).time -
lowTime)/(upTime - lowTime))*(SV(id).range_residuals(upLoc).meters -
SV(id).range_residuals(lowLoc).meters);
    else
        SV(id).DGPSbsRR(j).meters =
SV(id).range_residuals(lowLoc).meters;
    end
    SV(id).DGPSbsRR(j).time = baseStation.posECEF(i).time;

    %% Coordinate SVs R range residuals timing -----
-----
[lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, SV_RRR(id).time);

    if lowTime ~= upTime
        SV(id).DGPSrRR(j).meters =
SV(id).rover_residuals(lowLoc).meters + ((baseStation.posECEF(i).time -
lowTime)/(upTime - lowTime))*(SV(id).rover_residuals(upLoc).meters -
SV(id).rover_residuals(lowLoc).meters);
    else
        SV(id).DGPSrRR(j).meters =
SV(id).rover_residuals(lowLoc).meters;
    end
    SV(id).DGPSrRR(j).time = baseStation.posECEF(i).time;

end

    end
end

%% Estimate ranges, pseudoranges and compute correction
=====
for id = 1:length(SV)
    j = 0;
        for i = 1:length(baseStation.posECEF)

```

```

        if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time &&
j<length(SV(id).DGPSpos)
    j = j+1;

        %% Base Station
=====
        %% Correction of SV pos due to Earth Rotation -----
-----

        range = getRange(SV(id).DGPSpos(j),
baseStation.posECEF(i));
        traveltimes = range/c ;
        SVpos = correctEarthRot(traveltimes,
SV(id).DGPSpos(j));

        % Time -----
-----

        SV(id).BaseStation(j).time =
baseStation.posECEF(i).time;

        % Ranges -----
-----

        SV(id).BaseStation(j).range = getRange(SVpos,
trueECEF);

        % Pseudoranges -----
-----

        SV(id).BaseStation(j).pseudorange = getRange(SVpos,
baseStation.posECEF(i)) + SV(id).DGPSbsRR(j).meters;

        % Correction -----
-----

        SV(id).BaseStation(j).correction =
SV(id).BaseStation(j).range - SV(id).BaseStation(j).pseudorange;

        %% Rover
=====
        %% Correction of SV pos due to Earth Rotation -----
-----

        range = getRange(SV(id).DGPSpos(j), rover.DGPSpos(j));
        traveltimes = range/c ;
        SVpos = correctEarthRot(traveltimes,
SV(id).DGPSpos(j));

        % Time -----
-----

        SV(id).Rover(j).time = baseStation.posECEF(i).time;

        % Pseudoranges -----
-----

        SV(id).Rover(j).pseudorange = getRange(SVpos,
rover.DGPSpos(j)) - SV(id).DGPSrRR(j).meters;

        % Correction -----
-----

        SV(id).Rover(j).corrected_pseudorange =
SV(id).Rover(j).pseudorange + SV(id).BaseStation(j).correction;
    end
end
end

```

```

disp('DGPS correction successfully completed.')
disp(' ')
end

```

H.2.3.3 computeCorrectionReal.m

```

function [ SV, rover, activeSV ] = computeCorrectionReal( SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R )
%computeCorrectionReal computes DGPS correction applied to Navigation
Solution
%using real active SVs.

%[ SV, rover, activeSV ] = computeCorrectionReal( SV, rover,
baseStation, trueECEF, activeSV_BS, activeSV_R )
%
% INPUTS:
%   SV           - Space Vehicles structure.
%   rover        - rover structure.
%   baseStation  - base station structure.
%   trueECEF     - true base station receiver location in ECEF
%                   coordinates.
%   activeSV_BS - -
%   activeSV_R  - active SVs according to rover receiver.
%
% OUTPUTS:
%   SV           - updated with pseudoranges SV structure.
%   rover        - time adapted and updated rover structure.
%   activeSV     - active SVs.
%-----
-----
```

```

disp('Computing ranges and DGPS correction...')

%% Initialize variables
=====

for n = 1:length(baseStation.posECEF)
    time_arrayBS(n) = baseStation.posECEF(n).time;
end

for n = 1:length(rover.posECEF)
    time_arrayR(n) = rover.posECEF(n).time;
end

for n = 1:length(activeSV_BS)
    activeSV_BSarray(n) = activeSV_BS(n).time;
end

for n = 1:length(activeSV_R)
    activeSV_Rarray(n) = activeSV_R(n).time;
end

for id = 1:length(SV)
```

```

    for n = 1:length(SV(id).pos)
        SV_array(id).time(n) = SV(id).pos(n).time;
    end
end

j = 0;
k = 0;
% Fix Time
=====
for i = 1:length(baseStation.posECEF)
    if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time

        % Determining active SV -----
-----
        k = k+1;
        activeSV(k).SVID = unifyActiveSV(baseStation.posECEF(i).time,
activeSV_BSarray, activeSV_Rarray, activeSV_BS, activeSV_R);
        activeSV(k).time = baseStation.posECEF(i).time;

        if ~isempty(activeSV)
            j = j+1;
            [lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, time_arrayR);
            if lowTime ~= upTime
                rover.DGPSpos(j).x = rover.posECEF(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).x - rover.posECEF(lowLoc).x);
                rover.DGPSpos(j).y = rover.posECEF(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).y - rover.posECEF(lowLoc).y);
                rover.DGPSpos(j).z = rover.posECEF(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).z - rover.posECEF(lowLoc).z);
            else
                rover.DGPSpos(j).x = rover.posECEF(lowLoc).x;
                rover.DGPSpos(j).y = rover.posECEF(lowLoc).y;
                rover.DGPSpos(j).z = rover.posECEF(lowLoc).z;
            end
            rover.DGPSpos(j).time = baseStation.posECEF(i).time;

            for m = 1:length(activeSV(k).SVID)
                id = activeSV(k).SVID(m);
                [lowTime, upTime, lowLoc, upLoc] =
closestValue(baseStation.posECEF(i).time, SV_array(id).time);

                if lowTime ~= upTime
                    SV(id).DGPSpos(j).x = SV(id).pos(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).x - SV(id).pos(lowLoc).x);
                    SV(id).DGPSpos(j).y = SV(id).pos(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).y - SV(id).pos(lowLoc).y);
                    SV(id).DGPSpos(j).z = SV(id).pos(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(SV(id).pos(upLoc).z - SV(id).pos(lowLoc).z);
                else
                    SV(id).DGPSpos(j).x = SV(id).pos(lowLoc).x;
                    SV(id).DGPSpos(j).y = SV(id).pos(lowLoc).y;
                    SV(id).DGPSpos(j).z = SV(id).pos(lowLoc).z;
                end
            end
        end
    end
end

```

```

        end
    SV(id).DGPSpos(j).time = baseStation.posECEF(i).time;
end

end
end
end

%% Estimate ranges, pseudoranges and compute correction
=====
for id = 1:length(SV)
    j = 0;
        for i = 1:length(baseStation.posECEF)
            if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time &&
j<length(SV(id).DGPSpos)
                j = j+1;
                %% Base Station
=====
                % Time -----
-----
                SV(id).BaseStation(j).time =
baseStation.posECEF(i).time;

                % Ranges -----
-----
                SV(id).BaseStation(j).range =
getRange(SV(id).DGPSpos(j), trueECEF);

                % Pseudoranges -----
-----
                SV(id).BaseStation(j).pseudorange =
getRange(SV(id).DGPSpos(j), baseStation.posECEF(i));

                % Correction -----
-----
                SV(id).BaseStation(j).correction =
SV(id).BaseStation(j).range - SV(id).BaseStation(j).pseudorange;

                %% Rover
=====
                % Time -----
-----
                SV(id).Rover(j).time = baseStation.posECEF(i).time;

                % Pseudoranges -----
-----
                SV(id).Rover(j).pseudorange =
getRange(SV(id).DGPSpos(j), rover.DGPSpos(j));

                % Correction -----
-----
                SV(id).Rover(j).corrected_pseudorange =
SV(id).Rover(j).pseudorange + SV(id).BaseStation(j).correction;
            end
        end
    end
end

```

```

    disp('DGPS correction successfully completed.')
    disp(' ')
end

```

H.2.3.4 computeCorrectionVirtual.m

```

function [ SV, rover ] = computeCorrectionVirtual( SV, rover,
baseStation, trueECEF )
%computeCorrectionVirtual computes DGPS correction applied to
Navigation
%Solution using virtually generated SVs.

[%[ SV, rover ] = computeCorrectionVirtual( SV, rover, baseStation,
trueECEF )

%
% INPUTS:
%     SV           - Space Vehicles structure.
%     rover        - rover structure.
%     baseStation  - base station structure.
%     trueECEF     - true base station receiver location in ECEF
%                   coordinates.
%     activeSV_BS - active SVs according to base station
% receiver.
%     activeSV_R  - active SVs according to rover receiver.
%
% OUTPUTS:
%     SV           - updated with pseudoranges SV structure.
%     rover        - time adapted and updated rover structure.
%-----
-----

disp('Computing ranges and DGPS correction...')

%% Initialize variables
=====

for n = 1:length(baseStation.posECEF)
    time_arrayBS(n) = baseStation.posECEF(n).time;
end

for n = 1:length(rover.posECEF)
    time_arrayR(n) = rover.posECEF(n).time;
end

j = 0;

%% Fix Time
=====
for i = 1:length(baseStation.posECEF)
    if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time
        j = j+1;
        [ lowTime, upTime, lowLoc, upLoc ] =
closestValue(baseStation.posECEF(i).time, time_arrayR);
        if lowTime ~= upTime

```

```

        rover.DGPSpos(j).x = rover.posECEF(lowLoc).x +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).x - rover.posECEF(lowLoc).x);
        rover.DGPSpos(j).y = rover.posECEF(lowLoc).y +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).y - rover.posECEF(lowLoc).y);
        rover.DGPSpos(j).z = rover.posECEF(lowLoc).z +
((baseStation.posECEF(i).time - lowTime)/(upTime -
lowTime))*(rover.posECEF(upLoc).z - rover.posECEF(lowLoc).z);
    else
        rover.DGPSpos(j).x = rover.posECEF(lowLoc).x;
        rover.DGPSpos(j).y = rover.posECEF(lowLoc).y;
        rover.DGPSpos(j).z = rover.posECEF(lowLoc).z;
    end
    rover.DGPSpos(j).time = baseStation.posECEF(i).time;
end
end

%% Estimate ranges, pseudoranges and compute correction
=====
for id = 1:length(SV)
    j = 0;
        for i = 1:length(baseStation.posECEF)
            if baseStation.posECEF(i).time >= rover.posECEF(1).time &&
baseStation.posECEF(i).time <= rover.posECEF(end).time
                j = j+1;
                %% Base Station
=====
                % Time -----
-----
                SV(id).BaseStation(j).time =
baseStation.posECEF(i).time;

                % Ranges -----
-----
                SV(id).BaseStation(j).range =
getRange(SV(id).DGPSpos(1), trueECEF);

                % Pseudoranges -----
-----
                SV(id).BaseStation(j).pseudorange =
getRange(SV(id).DGPSpos(1), baseStation.posECEF(i));

                % Correction -----
-----
                SV(id).BaseStation(j).correction =
SV(id).BaseStation(j).range - SV(id).BaseStation(j).pseudorange;

                %% Rover
=====
                % Time -----
-----
                SV(id).Rover(j).time = baseStation.posECEF(i).time;

                % Pseudoranges -----
-----
                SV(id).Rover(j).pseudorange =
getRange(SV(id).DGPSpos(1), rover.DGPSpos(j));

```

```
% Correction -----
-----
SV(id).Rover(j).corrected_pseudorange =
SV(id).Rover(j).pseudorange + SV(id).BaseStation(j).correction;
    end
end
end

disp('DGPS correction successfully completed.')
disp(' ')

end
```

H.2.4 Export

H.2.4.1 exportKML.m

```
function [ ] = exportKML( fileName, routeName1, data1, routeName2,
data2, trueLocName, trueLoc, geoidHeight )
%exportKML exports results generating a KML file. This function allows
to
%export 3 routes. In this project it has been used to export
uncorrected
%and corrected paths plus Base Station true location.

%[ ] = exportKML( fileName, routeName1, data1, routeName2, data2,
trueLocName, trueLoc )
%
% INPUTS:
%   fileName      - name of the generated file.
%   routeName1    - name of the first route.
%   data1         - route 1 in geodetic coordinates.
%   routeName2    - name of the second route.
%   data2         - route 3 in geodetic coordinates.
%   trueLocName   - name of true location.
%   trueLoc       - true location in geodetic coordinates.
%   geoidHeight   - height of the geoid at the reference
location.
%-----
-----

disp('Exporting results...')

format ('compact');
format ('long', 'g');
load('KMLtemplate')

precision = 30;

fileName(end+1:end+4) = '.kml';
fileID = fopen(fileName, 'w');

for i = 1:N;
    row = plantilla(i).row;

    % Write file name -----
-----
FNbeginning = strfind(row, 'FILENAME');
```

```

if ~isempty(FNbeg)
    FNending = FNbeg+length(fileName);
    row(FNbeg:end) = [];
    row(FNbeg:FNending-1) = fileName;
    row(FNending:FNending+8) = '</name>\n';
    fileName(end-3:end) = [];
end

% Write routel name -----
RNbeginning = strfind(row, 'ROUTENAME1');
if ~isempty(RNbeginning)
    RNending = RNbeginning+length(routeName1);
    row(RNbeginning:end) = [];
    row(RNbeginning:RNending-1) = routeName1;
    row(RNending:RNending+8) = '</name>\n';
end

% Write coordinates1 -----
C1beginning = strfind(row, 'COORDINATES1');
if ~isempty(C1beginning)
    row(C1beginning:end) = [];
    for m = 1:length(data1)
        % Longitude
        add = num2str(data1(m).lon, precision);
        add = [add ','];
        row = [row add];

        % Latitude
        add = num2str(data1(m).lat, precision);
        add = [add ','];
        row = [row add];

        % Height
        % Check if other nomenclature has been used -----
        if isfield(data1, 'height')
            h = data1(m).height;
        elseif isfield(data1, 'altHAE')
            h = data1(m).altHAE;
        elseif isfield(data1, 'h')
            h = data1(m).h;
        else
            error('Height field not found.')
        end

        add = num2str(h - geoidHeight, precision);
        add = [add ' '];
        row = [row add];
    end
end

% Write route2 name -----
RNbeginning = strfind(row, 'ROUTENAME2');
if ~isempty(RNbeginning)
    RNending = RNbeginning+length(routeName2);
    row(RNbeginning:end) = [];
    row(RNbeginning:RNending-1) = routeName2;

```

```

    row(RNending:RNending+8) = '</name>\n';
end

% Write coordinates2 -----
-----
C2beginning = strfind(row, 'COORDINATES2');
if ~isempty(C2beginning)
    row(C2beginning:end) = [];
    for m = 1:length(data2)
        % Longitude
        add = num2str(data2(m).lon, precision);
        add = [add ','];
        row = [row add];

        % Latitude
        add = num2str(data2(m).lat, precision);
        add = [add ','];
        row = [row add];

        % Height
        add = num2str(data2(m).h - geoidHeight, precision);
        add = [add ' '];
        row = [row add];
    end
end

% Write true location name -----
-----
RNbeginning = strfind(row, 'TRUELOCNAME');
if ~isempty(RNbeginning)
    RNending = RNbeginning+length(trueLocName);
    row(RNbeginning:end) = [];
    row(RNbeginning:RNending-1) = trueLocName;
    row(RNending:RNending+8) = '</name>\n';
end

% Write true location coordinates -----
-----
C3beginning = strfind(row, 'TRUELOC');
if ~isempty(C3beginning)
    row(C3beginning:end) = [];

    % Longitude
    add = num2str(trueLoc.lon, precision);
    add = [add ','];
    row = [row add];

    % Latitude
    add = num2str(trueLoc.lat, precision);
    add = [add ','];
    row = [row add];

    % Height
    add = num2str(trueLoc.h - geoidHeight, precision);
    add = [add ' '];
    row = [row add];
end

fprintf(fileID, row);
end

```

```
fclose(fileID);  
  
fprintf('Results saved in %s.kml\n', fileName);  
disp(' ')  
  
end
```

H.2.4.2 getTemplate.m

```
%% ===== GENERATE GOOGLE EARTH FILE KML TEMPLATE
=====

%This script generates and stores KML file templates that can be used
in
%exportKML.m file to export results in KML format that then can be
%visualized in Google Earth or Google Maps.

clear; close all; clc

%% Load KML file used as a model
=====
fileID = fopen('KMLtemplate.kml');

%% GET TEMPLATE
=====
n = 0;
while ~feof(fileID)

    % Get row -----
    n = n+1;
    row = fgets(fileID);
    if row == -1
        fseek(fileID, 0, 1);
    end

    plantilla(n).row = row;

end
fclose(fileID);

%% SAVE TEMPLATE
=====
N = n;
clear row fileID n
save('KMLtemplate')

disp('KML Template created.')
```

H.2.5 getFunctions

H.2.5.1 generateSV.m

```
function [ SV, activeSV ] = generateSV( centre, radius, nSatellites,
roverPos )
%generateSV generates virtual SVs distributed along a sphere of given
%centre and radius. SVs position is generated for each rover position
%adapted to base station timing. Also returns the number of satellites
used
%in the activeSV format.

%[ SV, activeSV ] = generateSV( centre, radius, nSatellites, roverPos
)
```

```
%    INPUTS:
%      centre           - sphere's centre.
%      radius            - sphere's radius.
%      nSatellites       - Number of satellites to be generated. It
must be
%                           the square of an integer.
%      roverPos          - rover's position adapted to base station
timing.

%
%    OUTPUTS:
%      SV                - SV structure.
%      activeSV          - Active SV in activeSV format. Includes SV ID
and
%                           total number of active SVs.

disp('Creating imaginary Space Vehicles...')

N = sqrt(nSatellites)-1;
%% Check if number of SVs is correct
=====
if rem(N, 1) ~= 0
    error('Number of SVs must be the square of an integer')
end

%% Initialize variables
=====
id = 0;
SVarray = [];

%% Generate points
=====
[X,Y,Z] = ellipsoid(centre.x, centre.y, centre.z, radius, radius,
radius, round(N));

for i = 1:length(X)
    for j = 1:length(X)
        id = id+1;
        SV(id).ID = id;
        for n = 1:length(roverPos)
            SV(id).DGPSpos(n).x = X(i,j);
            SV(id).DGPSpos(n).y = Y(i,j);
            SV(id).DGPSpos(n).z = Z(i,j);
        end
        SVarray = [SVarray id];
    end
end

%% Create activeSV variable with message format
=====
for n = 1:length(roverPos)
    activeSV(n).SVID = SVarray;
end

disp('Spaces Vehicles successfully generated.')
end
```

H.2.5.2 getActiveSV.m

```
function [ activeSV ] = getActiveSV( GPGSA )
%getActiveSV reads NMEA GPGSA sentence and returns array of SV ID used
and
%its assigned message number.

% [ activeSV ] = getActiveSV( GPGSA )
%
% INPUT:
%     GPGSA      - NMEA GPGSA sentence.
%
% OUTPUT:
%     activeSV   - Active SVs ID, total number of SVs and message
%                   number.

%% Read and assign values
=====
for i = 1:length(GPGSA)
    activeSV(i).SVID = GPGSA(i).payload.SVID;
    activeSV(i).SVs = GPGSA(i).payload.SVs;
    activeSV(i).numMSG = GPGSA(i).numMSG;
end

end
```

H.2.5.3 getAID_EPH.m

```
function [ AID_EPH ] = getAID_EPH( payload )
%AID_EPH reads UBX AID-EPH message payload (hexadecimal) and returns
%subframe words of the navigation message.

% [ AID_EPH ] = getAID_EPH( payload )
%
% INPUT:
%     payload      - payload of UBX AID-EPH messages.
%
% OUTPUT:
%     AID_EPH     - decoded navigation message subframes 1 to 3.

%% Clean payload
=====
payload(7:8:end) = ' ';
payload(8:8:end) = ' ';
payload(strfind(payload, ' ')) = [];

%% Get data from navigation message
=====
firstByteLoc = 1:6:length(payload)-1;
i = 1;

% SV ID
```

```

AID_EPH.ID =
hex2dec(checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1)-1))); i =
i+2;

% Navigation Message Subframe 1
AID_EPH.SF1D00 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D01 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D02 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D03 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D04 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D05 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D06 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF1D07 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;

% Navigation Message Subframe 2
AID_EPH.SF2D00 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D01 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D02 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D03 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D04 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D05 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D06 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF2D07 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;

% Navigation Message Subframe 3
AID_EPH.SF3D00 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D01 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D02 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D03 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D04 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D05 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D06 =
checkOrder(payload(firstByteLoc(i):firstByteLoc(i+1))); i = i+1;
AID_EPH.SF3D07 = checkOrder(payload(firstByteLoc(i):end));
end

```

H.2.5.4 getColor.m

```

function [ color ] = getColor( ID, SVs )
%getColor reads SV ID and assigns a color based on its ID and number
of SVs.

%[ color ] = getColor( ID, SVs )
%
%   INPUTS:
%       ID      - ID of the Space Vehicle.
%       SVs     - number of active SVs.
%
%   OUTPUT:
%       color   - assigned color.
%-----
```

```

x = ID/SVs;

% Red -----
-----
if x < 0.18 || x > 0.85
    red = 1;
elseif x <= 0.35 && x >= 0.18
    red = 1 - ((x - 0.18)/(0.35 - 0.18));
elseif x <= 0.85 && x >= 0.67
    red = (x - 0.67)/(0.85 - 0.67);
else
    red = 0;
end

% Green -----
-----
if x < 0.51 && x > 0.17
    green = 1;
elseif x <= 0.17
    green = x/0.17;
elseif x <= 0.68 && x >= 0.51
    green = 1 - ((x - 0.51)/(0.68 - 0.51));
else
    green = 0;
end

% Blue -----
-----
if x < 0.84 && x > 0.51
    blue = 1;
elseif x <= 0.51 && x >= 0.35
    blue = (x - 0.35)/(0.51 - 0.35);
elseif x >= 0.84
    blue = 1 - ((x - 0.84)/(1 - 0.84));
else
    blue = 0;
end

color = [blue green red];

```

end

H.2.5.5 getDistance.m

```

function [ distance ] = getDistance( posECEF, trueECEF )
%getDistance compute the distance between true location and all base
%station GPS receiver measures.

%[ distance ] = getDistance( posECEF, trueECEF )

%    INPUTS:
%        posECEF      - position array in ECEF coordinates.
%        trueECEF     - true location of the receiver in ECEF
coordinates.
%
%    OUTPUT:
%        distance     - distance array between each position and the
true
%                        location.
%-----
-----
```

%% Compute distance

```

=====
```

```

for i = 1:length(posECEF)
    distance(i).time = posECEF(i).time;
    distance(i).module = getRange(posECEF(i), trueECEF);
    distance(i).x = posECEF(i).x - trueECEF.x;
    distance(i).y = posECEF(i).y - trueECEF.y;
    distance(i).z = posECEF(i).z - trueECEF.z;
end
```

end

H.2.5.6 getDOP.m

```

function [ DOP ] = getDOP( GPGSA )
%getDOP reads input GPGSA decode message and returns dilution of
precision
%(PDOP, HDOP and VDOP) from NMEA GPGSA and message number previously
%assigned.

%[ DOP ] = getDOP( GPGSA )

%    INPUT:
%        GPGSA      - NMEA GPGSA decoded message.
%
%    OUTPUT:
%        DOP       - Dilution of Precision (PDOP, HDOP and VDOP).
%-----
-----
```

```
%% Read and assign values
=====
for i = 1:length(GPGSA)
    DOP(i).PDOP = GPGSA(i).payload.PDOP;
    DOP(i).HDOP = GPGSA(i).payload.HDOP;
    DOP(i).VDOP = GPGSA(i).payload.VDOP;
    DOP(i).numMSG = GPGSA(i).numMSG;
end
end
```

H.2.5.7 getEphemeris.m

```
function [ eph ] = getEphemeris( AID_EPH )
%getEphemeris reads decoded navigation message words from subframe 1
to 3
%and returns clock correction and ephemeris parameters.

[% ephemeris ] = getEphemeris( AID_EPH )
%
% INPUT:
%     AID_EPH      - UBX AID-EPH decoded message.
%
% OUTPUT:
%     eph          - structure containing clock correction and
ephemeris
%                         parameters.
%-----
-----
```

```
%% Initialize constants
=====
% GPS constants -----
-----
PI           = 3.1415926535898; % Pi used in the GPS coordinate system

% Check if message is empty -----
if isempty(AID_EPH)
    error('AID-EPH Message is empty')
end

% Get Ephemeris from UBX AID-EPH messages
=====
% Subframe 1 -----
-----

% Split SF1D00 in packets of bits -----
SF1D00bits = dec2bin(hex2dec(AID_EPH.SF1D00));
while length(SF1D00bits) < 24
    SF1D00bits = [0 SF1D00bits];
end

eph.weekNumber = bin2dec(SF1D00bits(1:10)) + 1024;
eph.accuracy = bin2dec(SF1D00bits(13:16));
```

```

eph.health = bin2dec(SF1D00bits(17:22));
eph.T_GD = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF1D04(5:6)), 8)) *
2^(-31);
eph.IODC = bin2dec([SF1D00bits(23:24)
dec2bin(hex2dec(AID_EPH.SF1D05(1:2)))]);
eph.toc = hex2dec(AID_EPH.SF1D05(3:6)) * 2^4;
eph.af2 = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF1D06(1:2)), 8)) *
2^(-55);
eph.af1 = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF1D06(3:6)), 16)) *
2^(-43);

% Split SF1D07 in packets of bits -----
-----
SF1D07bits = dec2bin(hex2dec(AID_EPH.SF1D07));
while length(SF1D07bits) < 24
    SF1D07bits = [0 SF1D00bits];
end
eph.af0 = twosComp2dec(SF1D07bits(1:22)) * 2^(-31);

%% Subframe 2 -----
-----
eph.IODE = hex2dec(AID_EPH.SF2D00(1:2));
eph.Crs = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF2D00(3:6)), 16)) *
2^(-5);
eph.An = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF2D01(1:4)), 16)) * 2^(-
43) * PI;
eph.M0 = twosComp2dec(dec2bin(hex2dec([AID_EPH.SF2D01(5:6)
AID_EPH.SF2D02(1:6)]), 32)) * 2^(-31) * PI;
eph.Cuc = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF2D03(1:4)), 16)) * 2^(-
29);
eph.e = (hex2dec([AID_EPH.SF2D03(5:6) AID_EPH.SF2D04(1:6)])) * 2^(-
33);
eph.Cus = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF2D05(1:4)), 16)) * 2^(-
29);
eph.sqrtA = (hex2dec([AID_EPH.SF2D05(5:6) AID_EPH.SF2D06(1:6)])) * 2^(-
19);
eph.toe = hex2dec(AID_EPH.SF2D07(1:4)) * 2^4;

%% Subframe 3 -----
-----
eph.Cic = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF3D00(1:4)), 16)) * 2^(-
29);
eph.omega0 = twosComp2dec(dec2bin(hex2dec([AID_EPH.SF3D00(5:6)
AID_EPH.SF3D01(1:6)]), 32)) * 2^(-31) * PI;
eph.Cis = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF3D02(1:4)), 16)) * 2^(-
29);
eph.i0 = twosComp2dec(dec2bin(hex2dec([AID_EPH.SF3D02(5:6)
AID_EPH.SF3D03(1:6)]), 32)) * 2^(-31) * PI;
eph.Crc = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF3D04(1:4)), 16)) * 2^(-
5);
eph.omega = twosComp2dec(dec2bin(hex2dec([AID_EPH.SF3D04(5:6)
AID_EPH.SF3D05(1:6)]), 32)) * 2^(-31) * PI;
eph.omegaDot = twosComp2dec(dec2bin(hex2dec(AID_EPH.SF3D06(1:6)), 24)) *
2^(-43) * PI;
eph.IODE = hex2dec(AID_EPH.SF3D07(1:2));

```

```
% Split SF3D07 in packets of bits -----
-----
SF3D07bits = dec2bin(hex2dec(AID_EPH.SF3D07));
while length(SF3D07bits)<24
    SF3D07bits = [0 SF3D07bits];
end
eph.iDot = twosComp2dec(SF3D07bits(1:14)) * 2^(-43) * PI;
end
```

H.2.5.8 getfTOW.m

```
function [ geoPos ] = getfTOW( geoPos, posECEF )
%getfTOW Assigns to geodetic position obtained by UBX NAV-POSLLH
messages
%a float precision TOW (fTOW). This value is the closest value of iTOW
%of
%UBX NAV-POSLLH messages to the fTOW of UBX NAV-SOL. It is assumed
%that
%they come from the same navigation solution.

[% geoPos ] = getfTOW( geoPos, POSECEF )
%
%   INPUTS:
%       geoPos      - position in WGS-84 coordinates.
%       posECEF     - navigation solution final position in ECEF
%structure,
%                           also contains accurate TOW.
%
%   OUTPUTS:
%       geoPos      - updated with high precision TOW geodetic
%position.
%-----
-----

%% Create fTOW array
=====
for i = 1:length(posECEF)
    fTOWarray(i) = posECEF(i).TOW;
end

%% Assign closest value
=====
for i = 1:length(geoPos)
    [ lowTime, upTime, ~, ~ ] = closestValue( geoPos(i).iTOW,
fTOWarray );
    geoPos(i).TOW = min(abs(geoPos(i).iTOW - lowTime),
abs(geoPos(i).iTOW - upTime));
end
```

H.2.5.9 gteGNSSparameters.m

```
function [ GNSSparameters ] = getGNSSparameters( GNSSmessages,
correctionMode )
%getGNSSparameters coordinates all the GNSS parameters extraction
process
%from decoded messages.
```

```
%[ GNSSparameters ] = getGNSSparameters( GNSSmessages )
%
%   INPUTS:
%       GNSSmessages           - structure containing all decoded
% messages.
%       correctioMode          - correction mode, avoids unnecessary
% computations.
%
%   OUTPUT:
%       GNSSparameters         - structure containing all the extracted
% parameters.
%-----
-----

% Check messages existance -----
if isempty(GNSSmessages)
    error('No GNSS messages found.')
end

disp('Extracting parameters...')

%% Initialize variables
=====
GNSSparameters = [];
activeSV = [];
receiver.geoPos = [];
receiver.posECEF = [];
receiver.vel = [];
t_msg = [];
DOP = [];
nSatellites = 32;
messages = [];
m = 0;
f = 0;
r = 0;
s = 0;

for id = 1:nSatellites
    SV(id).ID = id;
    SV(id).pos = [];
    SV(id).range_residuals = [];
end

%% Get GNSS Messages
=====

%% NMEA Protocol -----
if ~isempty(GNSSmessages.NMEA)
    % GPGGA -----
    if ~isempty(GNSSmessages.NMEA.GPGGA) &&
    isempty(GNSSmessages.UBX.NAV_POSLLH)
        m = m+1;
        messages{m} = 'NMEA GPGGA';
        receiver.geoPos = getNMEAFixData(GNSSmessages.NMEA.GPGGA);
        geoPos = 'GPGGA';
        if isempty(GNSSmessages.UBX.NAV_SOL)
```

```

        t_msg = getTimeTable(GNSSmessages.NMEA.GPGGA);
    end
end

% GPGSA -----
if ~isempty(GNSSmessages.NMEA.GPGSA)
    m = m+1;
    messages{m} = 'NMEA GPGSA';
    DOP = getDOP(GNSSmessages.NMEA.GPGSA);
    activeSV = getActiveSV(GNSSmessages.NMEA.GPGSA);
end

% GPGRS -----
if ~isempty(GNSSmessages.NMEA.GPGRS)
    % UTC time is only used if there's no UBX NAV-SOL messages
    m = m+1;
    messages{m} = 'NMEA GPGRS';
    if isempty(GNSSmessages.UBX.NAV_SOL) && isempty(t_msg)
        t_msg = getTimeTable(GNSSmessages.NMEA.GPGRS);
    end
    SV = getRangeResiduals(GNSSmessages.NMEA.GPGRS, activeSV,
nSatellites, SV);
    s = s+1;
    for id = 1:length(SV)
        SV(id).fields{s} = 'range_residuals';
    end
end
end

%% UBX Protocol -----
if ~isempty(GNSSmessages.UBX)

    % NAV-SOL -----
if ~isempty(GNSSmessages.UBX.NAV_SOL)
    m = m+1;
    messages{m} = 'UBX NAV-SOL';
    [ receiver, t_msg ] = getSOL(GNSSmessages.UBX.NAV_SOL,
receiver);
end

    % NAV-POSECEF -----
% Ignore NAV_POSECEF if NAV-SOL messages are received.
if ~isempty(GNSSmessages.UBX.NAV_POSECEF) &&
isempty(GNSSmessages.UBX.NAV_SOL)
    m = m+1;
    messages{m} = 'UBX NAV-POSECEF';
    receiver.poseCEF = getPOSECEF(GNSSmessages.UBX.NAV_POSECEF);
end

    % NAV-POSLLH -----
if ~isempty(GNSSmessages.UBX.NAV_POSLLH)
    m = m+1;
    messages{m} = 'UBX NAV-POSLLH';
    receiver.geoPos = getUBXFixData(GNSSmessages.UBX.NAV_POSLLH);
end

```

```

    geoPos = 'POSLH';
end

% AID-EPH -----
-----
if ~isempty(GNSSmessages.UBX.AID_EPH) && ~isempty(t_msg)
    m = m+1;
    messages{m} = 'UBX AID-EPH';
    GNSSmessages.UBX.AID_EPH = assignTime(GNSSmessages.UBX.AID_EPH,
t_msg);
end
if ~isempty(GNSSmessages.UBX.AID_EPH) && (strcmp(correctionMode,
'NavSolR') || strcmp(correctionMode, 'COMMON'))
    SV = trackSV(GNSSmessages.UBX.AID_EPH, SV, nSatellites);
    s = s+1;
    for id = 1:length(SV)
        SV(id).fields{s} = 'pos';
    end
end
end

% Display used messages -----
-----
disp('Data successfully readed from decoded messages:')
for m = 1:length(messages)
    fprintf('%s\n', messages{m});
end
disp(' ')

%% Assign and correct time
=====
disp('----- TIME CORRECTION -----')
disp('Correcting and assigning time ...')

% Space Vehicle -----
-----
if exist('activeSV', 'var') && isfield(activeSV, 'SVID')
    if ~isempty(activeSV(1).SVID(1))
        if isfield(SV(activeSV(1).SVID(1)), 'pos')
            if ~isempty(SV(activeSV(1).SVID(1)).pos)
                GNSSparameters.SV = SV;
            end
        end
    end
end
for id = 1:nSatellites
    if exist('activeSV', 'var') && isfield(activeSV, 'SVID')
        if ~isempty(activeSV(1).SVID(1))
            if ~isempty(SV(activeSV(1).SVID(1)).pos)
                if ~isempty(SV(id).pos) && (strcmp(correctionMode,
'NavSolR') || strcmp(correctionMode, 'COMMON'))
                    SV(id).pos = assignTime(SV(id).pos, t_msg);
                end
            end
            if ~isempty(SV(activeSV(1).SVID(1)).range_residuals)
                if ~isempty(SV(id).range_residuals) &&
(strcmp(correctionMode, 'NavSolR') || strcmp(correctionMode,
'COMMON'))

```

```

SV(id).range_residuals =
assignTime(SV(id).range_residuals, t_msg);
    end
end
    end
end
end

% Active Satellites -----
-----
if exist('activeSV', 'var') && isfield(activeSV, 'SVID')
    if ~isempty(activeSV) && ~isempty(t_msg)
        activeSV = assignTime(activeSV, t_msg);
    end
end

% Dilution Of precision -----
-----
if ~isempty(DOP) && ~isempty(t_msg)
    DOP = assignTime(DOP, t_msg);
end

% Receiver -----
-----
if ~isempty(receiver.posECEF) && ~isempty(t_msg)
    r = r+1;
    receiver.fields{r} = 'posECEF';
    receiver.posECEF = assignTime(receiver.posECEF, t_msg);
end
if ~isempty(receiver.geoPos) && ~isempty(t_msg)
    switch geoPos
        case 'GPGGA'
            r = r+1;
            receiver.fields{r} = 'geoPos';
            receiver.geoPos = assignTime(receiver.geoPos, t_msg);
        case 'POSLLH'
            if ~isempty(receiver.posECEF)
                r = r+1;
                receiver.fields{r} = 'geoPos';
                receiver.geoPos = getfTOW(receiver.geoPos,
receiver.posECEF);
                receiver.geoPos = assignTime(receiver.geoPos, t_msg);
            end
        otherwise
            error('Invalid message for obtaining Geodetic Position.')
        end
    end
if ~isempty(receiver.vel) && ~isempty(t_msg)
    r = r+1;
    receiver.fields{r} = 'vel';
    receiver.vel = assignTime(receiver.vel, t_msg);
end
disp('Time successfully corrected and assigned.')
disp(' ')
=====

if ~isempty(receiver)
    f = f+1;

```

```

GNSSparameters.fields{f} = 'receiver';
GNSSparameters.receiver = receiver;
end
if exist('activeSV', 'var')
    if ~isempty(activeSV)
        f = f+1;
        GNSSparameters.fields{f} = 'activeSV';
        GNSSparameters.activeSV = activeSV;
    end
end
if exist('activeSV', 'var') && isfield(activeSV, 'SVID')
    if ~isempty(activeSV(1).SVID(1))
        if ~isempty(SV(activeSV(1).SVID(1)).pos) ||
~isempty(SV(activeSV(1).SVID(1)).range_residuals)
            f = f+1;
            GNSSparameters.fields{f} = 'SV';
            GNSSparameters.SV = SV;
        end
    end
end
if ~isempty(t_msg)
    f = f+1;
    GNSSparameters.fields{f} = 't_msg';
    GNSSparameters.t_msg = t_msg;
end
if ~isempty(DOP)
    f = f+1;
    GNSSparameters.fields{f} = 'DOP';
    GNSSparameters.DOP = DOP;
end

disp('Data sent.')
disp(' ')
end

```

H.2.5.10 getGPGGA.m

```

function [ GPGGA ] = getGPGGA( payload )
%getGPGGA reads NMEA GPGGA hexdecimal message and returns UTC time
%(hhmmss.sss), latitude (ddmm.mmmm), northing indicator, longitude
%(dddmm.mmmm), eassting indicator, Status, SVs used, HDOP, Alt(msl),
geoid
%separation.

%[ GPGGA ] = getGPGGA( payload )
%
% INPUT:
%     payload      - NMEA GPGGA payload (hexadecimal).
%
% OUTPUT:
%     GPGGA       - structure containing UTC time(hhmmss.sss),
latitude
%                   (ddmm.mmmm), northing indicator,
longitude (dddmm.mmmm),
%                   eassting indicator, Status, SVs used, HDOP,
Alt (MSL),
%                   geoid separation.
%-----
-----
```

```

%% Find commas location
=====
commasLoc = strfind(payload, '2C');
c = 0;

%% Get data from message
=====
GPGGA.UTC = hex2str(payload(1:commasLoc(c+1)-1));           c =
c+1;
GPGGA.lat = hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1));   c =
c+1;
GPGGA.N = hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1));   c =
c+1;
GPGGA.lon = hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1));   c =
c+1;
GPGGA.E = hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1));   c =
c+1;
GPGGA.status =
str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1)));   c =
c+1;
GPGGA.SVs = str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-
1)));           c =
c+1;
GPGGA.HDOP = str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-
1)));           c =
c+1;
GPGGA.alt = str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-
1)));           c =
c+2;
GPGGA.geoidSep =
str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1)));
end

```

H.2.5.11 getGPGRS.m

```

function [ GPGRS ] = getGPGRS( payload )
%getGPGRS reads NMEA GPGRS hexdecimal message and returns UTC time and
%range residuals [m].
%
% [ GPGRS ] = getGPGRS( payload )
%
% INPUT:
%     payload      - NMEA GPGRS payload (hexadecimal).
%
% OUTPUT:
%     GPGRS       - structure containing UTC time and range
% residuals.
%-----
-----
```

```

%% Find commas location
=====
commasLoc = strfind(payload, '2C');

if length(commasLoc) <= 6
    GPGRS.UTC = NaN;
    GPGRS.mode = NaN;
    for i = 1:12
        GPGRS.range_residuals(i) = NaN;
    end
else

    %% Get data from messages
=====
    GPGRS.UTC = hex2str(payload(1:commasLoc(1)-1));
    GPGRS.mode =
str2double(char(hex2dec(payload(commasLoc(1)+2:commasLoc(2)-1)))); %c = 2;

    for i = 1:12
        if c+1 <= length(commasLoc)
            pl = payload(commasLoc(c)+2:commasLoc(c+1)-1);
        else
            if c <= length(commasLoc)
                pl = payload(commasLoc(c)+2:end);
            else
                pl = NaN;
            end
        end

        if isempty(pl)
            break
        elseif isnan(pl)
            break
        else
            GPGRS.range_residuals(i) = str2double(hex2str(pl));
        end
        c = c+1;
    end
end

end

```

H.2.5.12 getGPGSA.m

```

function [ GPGSA ] = getGPGSA( payload )
%getGPGSA Reads NMEA GPGSA hexdecimal message and returns operation
mode,
% NAV mode, SVs used, SVID and dilution of precision (PDOP, HDOP,
VDOP).

%[ GPGSA ] = getGPGSA( payload )
%
% INPUT:
%     payload      - NMEA GPGSA payload (hexadecimal).
%
% OUTPUT:

```

```
%      GPGSA      - structure containing operation mode, NAV mode,
SVs
%
%           used, SVID and dilution of precision (PDOP, HDOP,
VDOP) .
%-----
```

```
%% Find commas and asterisk location
=====
commasLoc = strfind(payload, '2C');
asterisk = strfind(payload, '2A');

%% Get data from message
=====
GPGSA.OP = char(hex2dec(payload(1:commasLoc(1)-1)));
GPGSA.NAV =
str2double(char(hex2dec(payload(commasLoc(1)+2:commasLoc(2)-1))));
GPGSA.SVID = [];
c = 2;

for i = 1:12
    if length(commasLoc) >= c+1
        pl = payload(commasLoc(c)+2:commasLoc(c+1)-1);
    else
        break
    end

    if ~isempty(pl)
        GPGSA.SVID(i) = str2double(hex2str(pl));
    end
    c = c+1;
end

GPGSA.SVs = length(GPGSA.SVID);
if c+1 <= length(commasLoc) &&
mod(length(payload(commasLoc(c)+2:commasLoc(c+1)-1)),2) == 0
    GPGSA.PDOP =
str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1))); c =
c+1;
else
    GPGSA.PDOP = NaN;
    c = c+1;
end
if c+1 <= length(commasLoc) &&
mod(length(payload(commasLoc(c)+2:commasLoc(c+1)-1)),2) == 0
    GPGSA.HDOP =
str2double(hex2str(payload(commasLoc(c)+2:commasLoc(c+1)-1))); c =
c+1;
else
    GPGSA.HDOP = NaN;
    c = c+1;
end

if length(asterisk) == 1 && c <= length(commasLoc)
    if mod(length(payload(commasLoc(c)+2:asterisk-1)),2) == 0
        GPGSA.VDOP =
str2double(hex2str(payload(commasLoc(c)+2:asterisk-1)));
    else
        GPGSA.VDOP = NaN;
```

```

    end
else
    GPGSA.VDOP = NaN;
end

end

```

H.2.5.13 getNAV_POSECEF.m

```

function [ NAV_POSECEF ] = getNAV_POSECEF( payload )
%getNAV_POSECEF reads UBX NAV-POSECEF message payload (hexadecimal)
and
%returns TOW, ECEF-X, ECEF-Y, ECEF-Z and 3D position accuracy.

%[ NAV_POSECEF ] = getNAV_POSECEF( payload )
%
% INPUT:
%     payload           - UBX NAV-POSECEF payload (hexadecimal).
%
% OUTPUT:
%     NAV_POSECEF      - structure containing ms-precision TOW, ECEF-
X,
%                         ECEF-Y, ECEF-Z and 3D position accuracy.
%-----
-----
```

```

% Time of Week (Low accuracy [s]) -----
-----
NAV_POSECEF.TOW = hex2dec(checkOrder(payload(1:8))) * 1e-3;

% ECEF Coordinates -----
-----
NAV_POSECEF.X =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(9:16))), 32)) * 1e-2;
NAV_POSECEF.Y =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(17:24))), 32)) * 1e-2;
NAV_POSECEF.Z =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(25:32))), 32)) * 1e-2;

% 3D Estimated Accuracy -----
-----
NAV_POSECEF.pAcc = hex2dec(checkOrder(payload(33:40))) * 1e-2;
end

```

H.2.5.14 getNAV_POSLLH.m

```

function [ NAV_POSLLH ] = getNAV_POSLLH( payload )
%getNAV_POSLLH reads UBX NAV-POSLLH message payload (hexadecimal) and
%returns TOW, lon [°], lat[°], height (above ellipsoid), hMSL, hAcc
and
%vAcc.

%[ NAV_POSLLH ] = getNAV_POSLLH( payload )
%
% INPUT:

```

```
% payload - UBX NAV-POSLH message payload
(hexadecimal).
%
% OUTPUT:
% NAV_POSLLH - structure containing ms-precision TOW, lon
[°],
% lat[°], height (above ellipsoid), hMSL, hAcc
and
% vAcc.
%-----
```

```
% Time of Week (Low accuracy [s]) -----
-----
NAV_POSLLH.TOW = hex2dec(checkOrder(payload(1:8))) * 1e-3;

% Geographic Fix Solution -----
-----
NAV_POSLLH.lon =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(9:16)))), 32)) * 1e-7;
NAV_POSLLH.lat =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(17:24)))), 32)) * 1e-7;
NAV_POSLLH.height =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(25:32)))), 32)) * 1e-3;
NAV_POSLLH.hMSL =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(33:40)))), 32)) * 1e-3;

% 3D Estimated Accuracy -----
-----
NAV_POSLLH.hAcc = hex2dec(checkOrder(payload(41:48))) * 1e-3;
NAV_POSLLH.vAcc = hex2dec(checkOrder(payload(49:56))) * 1e-3;
end
```

H.2.5.15 getNAV_SOL.m

```
function [ NAV_SOL ] = getNAV_SOL( payload )
%getNAV_SOL reads UBX NAV-SOL hexadecimal payload messages, decodes it
%and returns navigation solution parameters: Week Number, TOW,
position fix
%type, fix flags, position ECEF (X,Y,Z), position accuracy estimation,
%velocity ECEF, velocity accurate estimate, PDOP and No. used SVs.

%[ NAV_SOL ] = getNAV_SOL( payload )
%
% INPUT:
% payload - UBX NAV-SOL hexadecimal payload.
%
% OUTPUT:
% NAV_SOL - structure containing Week Number, high precision
TOW,
% position fix type, fix flags, position ECEF,
% accuracy estimation, velocity ECEF, velocity
accurate,
% PDOP and No. used SVs.
%-----
```

```
% Precise Time of Week -----
-----
NAV_SOL.iTOW = hex2dec(checkOrder(payload(1:8)));
NAV_SOL.fTOW =
twosComp2dec(dec2bin(hex2dec(checkOrder(payload(9:16))), 32));
NAV_SOL.weekNumber = hex2dec(checkOrder(payload(17:20)));

% GPS Fix and Flags Status -----
-----
NAV_SOL.gpsFix = hex2dec(checkOrder(payload(21:22)));
% flags_ = hex2dec(checkOrder(payload(23:24)));
% NAV_SOL.flags.TOWSET = flags

% ECEF Coordinates -----
-----
NAV_SOL.X = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(25:32))), 32)) * 1e-2;
NAV_SOL.Y = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(33:40))), 32)) * 1e-2;
NAV_SOL.Z = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(41:48))), 32)) * 1e-2;

% 3D Position Estimate Accuracy -----
-----
NAV_SOL.pAcc = hex2dec(checkOrder(payload(49:56))) * 1e-2;

% ECEF Coordinates -----
-----
NAV_SOL.VX = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(57:64))), 32)) * 1e-2;
NAV_SOL.VY = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(65:72))), 32)) * 1e-2;
NAV_SOL.VZ = twosComp2dec(dec2bin(hex2dec(checkOrder(payload(73:80))), 32)) * 1e-2;

% 3D Position Estimate Accuracy -----
-----
NAV_SOL.sAcc = hex2dec(checkOrder(payload(81:88))) * 1e-2;

% Position Dilution of Precision (pDOP) -----
-----
NAV_SOL.pDOP = hex2dec(checkOrder(payload(89:92))) * 0.01;

% Number of used SVs -----
-----
NAV_SOL.SVs = hex2dec(checkOrder(payload(95:96)));
end
```

H.2.5.16 getNMEAfixData.m

```
function [ geoPos ] = getNMEAfixData( GPGGA )
%getFixData reads NMEA GPGGA messages and returns lat, northing
indicator,
%lon, easting indicator, altitude (MSL) and altitude (HAE).

%[ pos ] = getNMEAfixData( GPGGA )
%
% INPUT:
```

```
%      GPGGA      - NMEA GPGGA message.
%
%  OUTPUT:
%      geoPos      - structure containing latitude, northing
indicator,
%                  longitude, easting indicatior, altitude above MSL
and
%                  HAE.
%-----
-----
```

%% Read message line-by-line

```
=====
for i = 1:length(GPGGA)
    if ~isempty(GPGGA(i).payload.lat)
        geoPos(i).lat = str2double(GPGGA(i).payload.lat(1:2)) +
str2double(GPGGA(i).payload.lat(3:end))/60;
        geoPos(i).N = GPGGA(i).payload.N;
        geoPos(i).lon = str2double(GPGGA(i).payload.lon(1:3)) +
str2double(GPGGA(i).payload.lon(4:end))/60;
        geoPos(i).E = GPGGA(i).payload.E;
        geoPos(i).altMSL = GPGGA(i).payload.alt;
        geoPos(i).althAE = GPGGA(i).payload.alt +
GPGGA(i).payload.geoidSep;
        geoPos(i).numMSG = GPGGA(i).numMSG;

        if geoPos(i).N == 'S' && geoPos(i).lat > 0
            geoPos(i).lat = -geoPos(i).lat;
        end
        if geoPos(i).E == 'W' && geoPos(i).lon > 0
            geoPos(i).lon = -geoPos(i).lon;
        end
    end
end
end
```

H.2.5.17 getPOSECEF.m

```
function [ posECEF ] = getPOSECEF( NAV_POSECEF )
%getPOSECEF reads decode UBX NAV-POSECEF and assigns x, y and z in
%Earth-Centered Earth-Fixed coordinate system. Also assigns message
number.
```

```
%[ posECEF ] = getPOSECEF( NAV_POSECEF )
%
%  INPUT:
%      NAV_POSECEF      - UBX NAV-POSECEF decoded parameters.
%
%  OUTPUT:
%      posECEF         - structure containing position in ECEF
coordinates
%                  and message number.
%-----
-----
```

%% Read messages line-by-line

```
=====
```

```

for i = 1:length(NAV_POSECEF)
    poseCEF(i).x = NAV_POSECEF(i).payload.X;
    poseCEF(i).y = NAV_POSECEF(i).payload.Y;
    poseCEF(i).z = NAV_POSECEF(i).payload.Z;
    poseCEF(i).numMSG = NAV_POSECEF(i).numMSG;
end
end

```

H.2.5.18 getRange.m

```

function [ range ] = getRange( PointA, PointB )
%getRange Computes distance between 2 points (3D).

% [ range ] = getRange( PointA, PointB )
%
% INPUTS:
%     PointA      - First point. Must be of the form:
%                   [point.x point.y point.z]
%     PointB      - Second point. Must be of the form:
%                   [point.x point.y point.z]
%
% OUTPUT:
%     range       - range between point A and B.
%-----
-----
```

```

range = sqrt((PointA.x-PointB.x)^2 + (PointA.y-PointB.y)^2 +
(PointA.z-PointB.z)^2);
end

```

H.2.5.19 getRangeResiduals.m

```

function [ SV ] = getRangeResiduals( GPGRS, activeSV, nSatellites, SV )
%
%getRangeResiduals reads NMEA GPGRS messages and returns range
residuals
%associated to each SV and time-message number.
```

```

% [ SV ] = getRangeResiduals( GPGRS, activeSV, nSatellites, SV )
%
% INPUTS:
%     GPGRS          - NMEA GPGRS decoded message.
%     activeSV       - active SVs.
%     nSatellites   - max number of satellites.
%     SV             - Space Vehicles structure.
%
% OUTPUT:
%     SV             - updated Space Vehicles structure.
%-----
-----
```

```

%% Initialize counter
=====
for n = 1:nSatellites
    SV(n).range_residuals.c = 0;
end

if length(GPGRS) == length(activeSV)
```

```

        sameLength = true;
else
    sameLength = false;
end

%% Fix NMEA GPGRS - active SV different length problem
=====
while ~sameLength
    distInit = abs(GPGRS(1).numMSG - activeSV(1).numMSG);
    distEnd = abs(GPGRS(end).numMSG - activeSV(end).numMSG);
    if length(GPGRS) > length(activeSV)
        if distInit < distEnd
            GPGRS(end) = [];
        else
            GPGRS(1) = [];
        end
    else
        if distInit < distEnd
            activeSV(end) = [];
        else
            activeSV(1) = [];
        end
    end
end

if length(GPGRS) == length(activeSV)
    sameLength = true;
    disp('Adjusted Range Residuals with active Space Vehicles.')
else
    sameLength = false;
end
end

%% Read all NMEA GPGRS messages
=====
for i = 1:length(GPGRS)
    % Assign range residual to SV -----
    j = 0;
    for n = 1:activeSV(i).SVs
        j = j+1;
        id = activeSV(i).SVID(n);
        SV(id).range_residuals(1).c = SV(id).range_residuals(1).c+1;
        if j <= length(GPGRS(i).payload.range_residuals)
            SV(id).range_residuals(SV(id).range_residuals(1).c).meters
            = GPGRS(i).payload.range_residuals(j);
            SV(id).range_residuals(SV(id).range_residuals(1).c).numMSG
            = GPGRS(i).numMSG;
        end
    end
end

%% Remove auxiliar c field
for id = 1:nSatellites
    if SV(id).range_residuals(1).c == 0
        SV(id).range_residuals = [];
    else
        SV(id).range_residuals = rmfield(SV(id).range_residuals, 'c');
    end
end

```

end

H.2.5.20 getSOL.m

```

function [ receiver, t_msg ] = getSOL( NAV_SOL, receiver )
%getSOL reads UBX NAV-SOL decoded messages and returns TOW, position
ECEF
%(X,Y,Z) and SVs used.

%[ receiver, t_msg ] = getSOL( NAV_SOL, receiver )
%
%   INPUTS:
%       NAV_SOL           - UBX NAV-SOL decoded message.
%       receiver          - receiver structure.
%
%   OUTPUTS:
%       receiver          - updated receiver structure.
%       t_msg              - table relating time and message number.
%-----
-----

%% Get initial TOW
=====
TOW0 = (NAV_SOL(1).payload.itOW)*1e-3 + (NAV_SOL(1).payload.ftOW)*1e-
9;

%% Read messages line-by-line
=====

for i = 1:length(NAV_SOL)

    TOW = (NAV_SOL(i).payload.itOW)*1e-3 +
(NAV_SOL(i).payload.ftOW)*1e-9;
    numMSG = NAV_SOL(i).numMSG;

    t_msg(i).numMSG = numMSG;
    t_msg(i).t = TOW - TOW0;
    t_msg(i).TOW = TOW;

    receiver.posECEF(i).TOW = TOW;
    receiver.posECEF(i).numMSG = numMSG;
    receiver.posECEF(i).x = NAV_SOL(i).payload.X;
    receiver.posECEF(i).y = NAV_SOL(i).payload.Y;
    receiver.posECEF(i).z = NAV_SOL(i).payload.Z;
    receiver.posECEF(i).acc = NAV_SOL(i).payload.pAcc;

    receiver.vel(i).TOW = TOW;
    receiver.vel(i).numMSG = numMSG;
    receiver.vel(i).x = NAV_SOL(i).payload.VX;
    receiver.vel(i).y = NAV_SOL(i).payload.VY;
    receiver.vel(i).z = NAV_SOL(i).payload.VZ;
    receiver.vel(i).acc = NAV_SOL(i).payload.sAcc;

```

```
end
end
```

H.2.5.21 getTimeTable.m

```
function [ t_msg ] = getTimeTable( GPGRS )
%getTimeTable Returns time - messages table from UTC time of NMEA
GPGRS
%messages.

%[ t_msg ] = getTimeTable( GPGRS )
%
% INPUT:
%     GPGRS      - NMEA GPGRS decoded message.
%
% OUTPUT:
%     t_msg       - table that relates time and message number.
%-----
-----

%% Get initial UTC time
=====
UTC_0 = GPGRS(1).payload.UTC;
t0 = utc2sec(UTC_0);

%% Read all NMEA GPGRS messages
=====
for i = 1:length(GPGRS)
    % Get time-message table -----
    %

    t_msg(i).numMSG = GPGRS(i).numMSG;
    t_msg(i).t = utc2sec(GPGRS(i).payload.UTC) - t0;
    t_msg(i).UTC = GPGRS(i).payload.UTC;
    %

end
end
```

H.2.5.22 getTrueECEF.m

```
function [ trueECEF ] = getTrueECEF( trueLocation )
%getTrueECEF reads true location in initial format and geographic
%coordinate system and returns it in ECEF coordinate system.

[%[ trueECEF ] = getTrueECEF( trueLocation )
%
% INPUT:
%     trueLocation      - true location in Google Earth format.
%
% OUTPUT:
%     trueECEF          - true location in ECEF coordinates.
%-----
-----
```

trueGeo.lat = trueLocation.lat.deg + (trueLocation.lat.min/60) +
(trueLocation.lat.sec/3600);
trueGeo.lon = trueLocation.lon.deg + (trueLocation.lon.min/60) +
(trueLocation.lon.sec/3600);
trueGeo.h = trueLocation.h;

if trueGeo.lat > 0 && trueLocation.lat.N == 'S'
 trueGeo = -trueGeo;
end
if trueGeo.lon > 0 && trueLocation.lon.E == 'W'
 trueGeo = -trueGeo;
end

wgs84 = wgs84Ellipsoid('meters');
[trueECEF.x, trueECEF.y, trueECEF.z] =
geodetic2ecef(trueGeo.lat*(pi/180), trueGeo.lon*(pi/180), trueGeo.h,
wgs84);
end

H.2.5.23 getTrueGeo.m

```
function [ trueGeo ] = getTrueGeo( trueLocation )
%getTrueGeo reads true location in initial format and geographic
%coordinate system and returns it in WGS-84 coordinates.
```

```
%[ trueGeo ] = getTrueGeo( trueLocation )
%
% INPUT:
%     trueLocation      - true location in format Google Earth
format.
%
% OUTPUT:
%     trueGeo          - true location in WGS-84 coordinates.
%-----
-----
```

trueGeo.lat = trueLocation.lat.deg + (trueLocation.lat.min/60) +
(trueLocation.lat.sec/3600);
trueGeo.lon = trueLocation.lon.deg + (trueLocation.lon.min/60) +
(trueLocation.lon.sec/3600);
trueGeo.h = trueLocation.h;

```

if trueGeo.lat > 0 && trueLocation.lat.N == 'S'
    trueGeo = -trueGeo;
end
if trueGeo.lon > 0 && trueLocation.lon.E == 'W'
    trueGeo = -trueGeo;
end
end

```

H.2.5.24 getUBXFixData.m

```

function [ geoPos ] = getUBXFixData( NAV_POSLLH )
%getUBXFixData reads decoded UBX NAV-POSLLH messages and returns fix
%geographic position.

%[ geoPos ] = getUBXFixData( NAV_POSLLH )
%
% INPUT:
%     NAV_POSLLH      - UBX NAV-POSLLH decoded message.
%
% OUTPUT:
%     geoPos          - Fixed navigation solution in WGS-84
coordinates.
%-----
-----

%% Read all messages line-by-line
=====

for i = 1:length(NAV_POSLLH)
    geoPos(i).numMSG = NAV_POSLLH(i).numMSG;
    geoPos(i).iTOW = NAV_POSLLH(i).payload.TOW;
    geoPos(i).lat = NAV_POSLLH(i).payload.lat;
    geoPos(i).lon = NAV_POSLLH(i).payload.lon;
    geoPos(i).height = NAV_POSLLH(i).payload.height;
    geoPos(i).hMSL = NAV_POSLLH(i).payload.hMSL;
    geoPos(i).hAcc = NAV_POSLLH(i).payload.hAcc;
    geoPos(i).vAcc = NAV_POSLLH(i).payload.vAcc;
end
end

```

H.2.6 positioning

H.2.6.1 computePosition.m

```

function [ rover ] = computePosition( SV, rover, activeSV,
correctionMode )
%correctPosition coordinates position computation once corrections
have
%been applied.

[%[ rover ] = computePosition( SV, rover, activeSV )
%
%    INPUTS:
%        SV          - Space Vehicle structure.
%        rover       - rover structure.
%        activeSV    - active Space Vehicles.
%
%    OUTPUT:
%        rover      - rover updated structure.
%-----
```

```

-----
```

```

disp('Recomputing position after correction...')

% Computes number of rows -----
nRows = 0;
for id = 1:length(SV)
    if length(SV(id).Rover) > nRows
        nRows = length(SV(id).Rover);
    end
end

%% Calculate rover position by least square method
=====
switch correctionMode
    case 'COMMON'
        for n = 1:nRows
            [rover.final_pos(n), rover.final_DOP(n) ] =
leastSquarePos(SV, n, activeSV(n).SVID, rover.DGPSpos(n), 1);
        end
    case 'NavSolR'
        for n = 1:nRows
            [rover.final_pos(n), rover.final_DOP(n) ] =
leastSquarePosNOdt(SV, n, activeSV(n).SVID, rover.DGPSpos(n));
        end
    case 'NavSolV'
        for n = 1:nRows
            [rover.final_pos(n), rover.final_DOP(n) ] =
leastSquarePosNOdt(SV, n, activeSV(n).SVID, rover.DGPSpos(n));
        end
    otherwise
        error('Internal error.')
end

%% Assign Time & Delete NaN
=====
```

```

n = 1; k = 0;
while n < nRows-k
    n = n+1;
    rover.final_pos(n).time = rover.DGPSpos(n).time;
    if isnan(rover.final_pos(n).x)
        rover.final_pos(n) = [];
    k = k+1;
    n = n-k;
end
end

disp('New position successfully computed.')
disp(' ')
end

```

H.2.6.2 correctEarthRot.m

```

function [ SVpos ] = correctEarthRot( traveltimes, SVpos )
%correctEarthRot returns rotated satellite ECEF coordinates due to
Earth
%rotation during signal travel time.

%X_sat_rot = e_r_corr(traveltimes, X_sat);
%
% INPUTS:
%     traveltimes - signal travel time
%     SVpos       - satellite's ECEF coordinates
%
% OUTPUTS:
%     SVpos       - rotated satellite's coordinates (ECEF)
%
%-----
-----

earthRotRate = 7.292115147e-5; % rad/sec

%% Find rotation angle
=====
earthRotAng = earthRotRate * traveltimes;

if ~isempty(earthRotAng)
    %% Make a rotation matrix
=====
R3 = [ cos(earthRotAng) sin(earthRotAng) 0;
       -sin(earthRotAng) cos(earthRotAng) 0;
       0 0 1];

    %% Do the rotation
=====
X(1) = SVpos.x;
X(2) = SVpos.y;
X(3) = SVpos.z;
X = X';
X_sat_rot = R3 * X;

SVpos.x = X_sat_rot(1);
SVpos.y = X_sat_rot(2);

```

```

SVpos.z = X_sat_rot(3);

else
    SVpos.x = [];
    SVpos.y = [];
    SVpos.z = [];
end

end

```

H.2.6.3 getSVposition.m

```

function [ satPos ] = getSVposition( eph, transmitTime )
%getSVposition returns Space Vehicle's position from ephemeris data
and
%clock correction parameters.

%[ satPos ] = getSVposition( eph, transmitTime )
%
% INPUTS:
%     eph:           - ephemeris and clock correction parameters.
%     transmitTime   - time at which the position is wanted to be
%                      estimated.
%
% OUTPUT:
%     satPos         - Space Vehicle final position.
%-----
-----
```

```

%% Initialize variables
=====
satPos = [];

%% Initialize constants
=====
% GPS constants -----
-----
PI = 3.1415926535898;           % Pi used in the GPS coordinate
                                  system

% Earth constants -----
-----
omegaEdot = 7.2921151467e-5;    % Earth rotation rate, [rad/s]
GM = 3.986005e14;               % Universal gravitational constant
times          % the mass of the Earth, [m^3/s^2]

% Relativistic constant -----
-----
F = -4.442807633e-10;          % Constant, [sec/(meter)^(1/2)]
```

```

%% Process satellite
=====
```

```

%% Find initial satellite clock correction -----
-----
% Find time difference -----
dt = check_t(transmitTime - eph.toc);

% Calculate clock correction -----
satClkCorr = (eph.af2 * dt + eph.af1) * dt + ...
              eph.af0 - ...
              eph.T_GD;

time = transmitTime - satClkCorr;

%% Find satellite's position -----
-----
% Restore semi-major axis
a = eph.sqrtA * eph.sqrtA;

% Time correction
tk = check_t(time - eph.toe);

% Initial mean motion
n0 = sqrt(GM / a^3);
% Mean motion
n = n0 + eph.An;

% Mean anomaly
M = eph.M0 + n * tk;
% Reduce mean anomaly to between 0 and 360 deg
M = rem(M + 2*PI, 2*PI);

% Initial guess of eccentric anomaly
E = M;

% Iteratively compute eccentric anomaly -----
-----
for j = 1:10
    E_old = E;
    E = M + eph.e * sin(E);
    dE = rem(E - E_old, 2*PI);

    if abs(dE) < 1.e-12
        % Necessary precision is reached, exit from the loop
        break;
    end
end

% Reduce eccentric anomaly to between 0 and 360 deg
E = rem(E + 2*PI, 2*PI);

% Compute relativistic correction term
dtr = F * eph.e * eph.sqrtA * sin(E);

% Calculate the true anomaly
nu = atan2(sqrt(1 - eph.e^2) * sin(E), cos(E)-eph.e);

```

```
% Compute angle phi
phi = nu + eph.omega;
%Reduce phi to between 0 and 360 deg
phi = rem(phi, 2*PI);

% Correct argument of latitude
u = phi + ...
    eph.Cuc * cos(2*phi) + ...
    eph.Cus * sin(2*phi);
% Correct radius
r = a * (1 - eph.e*cos(E)) + ...
    eph.Crc * cos(2*phi) + ...
    eph.Crs * sin(2*phi);
% Correct inclination
i = eph.i0 + eph.iDot * tk + ...
    eph.Cic * cos(2*phi) + ...
    eph.Cis * sin(2*phi);

% Compute the angle between the ascending node and the Greenwich
meridian
Omega = eph.omega0 + (eph.omegaDot - omegaEdot)*tk - ...
    omegaEdot * eph.toe;
% Reduce to between 0 and 360 deg
Omega = rem(Omega + 2*PI, 2*PI);

% Compute satellite coordinates -----
-----
satPos.x = cos(u)*r * cos(Omega) - sin(u)*r * cos(i)*sin(Omega);
satPos.y = cos(u)*r * sin(Omega) + sin(u)*r * cos(i)*cos(Omega);
satPos.z = sin(u)*r * sin(i);
satPos.numMSG = [];

% %% Include relativistic correction in clock correction -----
%
% satClkCorr(satNr) = (eph.af2 * dt + eph.af1) * dt + ...
%                         eph.af0 - ...
%                         eph.T_GD + dtr;

end
```

H.2.6.4 leastSquarePos.m

```
function [pos, DOP ] = leastSquarePos(SV, n, activeSV, initPos,
earthCorr)
%leastSquarePos computes final position by least square method. This
method
%accounts for receiver clock delay.

%[pos, DOP ] = leastSquarePos(SV, n, activeSV, initPos)
%
% INPUTS:
%   SV           - Space Vehicle structure.
%   n            - current message number.
%   activeSV     - active Space Vehicles at the current time.
%   initPos      - initial position.
%   earthCorr    - flag that enables Earth's rotation correction
```

```
%    OUTPUTS:
%      pos          - position final solution.
%      DOP          - Dilution Of Precision (GDOP, PDOP, HDOP, VDOP)
%-----
%-----

% Initialization
=====
c = 299792458; %m/s
delta1 = 1e-3; %m
delta2 = 1e-10; %s
nSatellites = length(activeSV);

pos = [];
DOP = [];

it = 0;
x = zeros(3);

pos.x = initPos.x;
pos.y = initPos.y;
pos.z = initPos.z;
pos.dt = 0;
Adt = 10;

if nSatellites < 4
    error('Not enough SVs. Minimum 4')
end

while abs(x(1)) >= delta1 || abs(x(2)) >= delta1 || abs(x(3)) >=
delta1 || abs(Adt) >= delta2 || it == 0
    it = it+1;
    A = [];
    B = [];
    for i = 1:nSatellites
        id = activeSV(i);
        %% Corrects SV position due to Earth's rotation
=====
        if earthCorr
            range = getRange(SV(id).DGPSpos(n), pos);
            travelttime = range/c ;
            SVpos = correctEarthRot(travelttime, SV(id).DGPSpos(n));
        else
            SVpos = SV(id).DGPSpos(n);
        end

        geomR = getRange(pos, SVpos);
        B(i) = SV(id).Rover(n).corrected_pseudorange - geomR;
        A(i, :) = [-(SVpos.x - pos.x)/geomR ...
                    -(SVpos.y - pos.y)/geomR ...
                    -(SVpos.z - pos.z)/geomR ...
                    1];
    end
    B = B';

% Ax = B;
x = A\B;
```

```

pos.x = pos.x + x(1);
pos.y = pos.y + x(2);
pos.z = pos.z + x(3);

Adt = pos.dt;
pos.dt = x(4)/c;
Adt = pos.dt-Adt;
end
fprintf('#MSG: %.0f      #IT: %.0f\n', n, it)

% Convert ECEF coordinates into geodetic coordinate system
=====
wgs84 = wgs84Ellipsoid('meters');
[pos.lat, pos.lon, pos.h] = ecef2geodetic(pos.x, pos.y, pos.z, wgs84);
pos.lat = (180/pi)*pos.lat;
pos.lon = (180/pi)*pos.lon;
pos.h = pos.h;

% Calculate Dilution Of Precision
=====
% Initialize output -----
-
dop      = zeros(1, 5);

% Calculate DOP -----
-
Q        = inv(A'*A);

DOP.GDOP = sqrt(trace(Q));
DOP.PDOP = sqrt(Q(1,1) + Q(2,2) + Q(3,3));
DOP.HDOP = sqrt(Q(1,1) + Q(2,2));
DOP.VDOP = sqrt(Q(3,3));
DOP.TDOP = sqrt(Q(4,4));
DOP.time = SV(activeSV(1)).Rover(n).time;
end

```

H.2.6.5 leastSquareNOdt.m

```

function [pos, DOP] = leastSquarePosNOdt(SV, n, activeSV, initPos)
%leastSquarePosNOdt computes final position by least square method.
%This method does not account for receiver clock delay.

```

```

%[pos, DOP] = leastSquarePosNOdt(SV, n, activeSV, initPos)
%
% INPUTS:
%   SV           - Space Vehicle structure.
%   n            - current message number.
%   activeSV    - active Space Vehicles at the current time.
%   initPos     - Initial position.
%
% OUTPUTS:
%   pos          - position final solution.
%   DOP          - Dilution Of Precision (GDOP, PDOP, HDOP, VDOP)
%-----
%-----

% Initialization
=====
```

```

c = 299792458; %m/s
delta = 1e-3; %m
nSatellites = length(activeSV);

pos = [];
DOP = [];

it = 0;
x = zeros(3);

pos.x = initPos.x;
pos.y = initPos.y;
pos.z = initPos.z;

if nSatellites < 3
    pos.x = NaN;
    pos.y = NaN;
    pos.z = NaN;
    A = NaN(3);
    warning('#MSG: %.0f      Less than 3 SV are visible.', n)
else
    while abs(x(1)) >= delta || abs(x(2)) >= delta || abs(x(3)) >=
delta || it == 0
        it = it+1;
        A = [];
        B = [];
        for i = 1:nSatellites
            id = activeSV(i);
            range = getRange(SV(id).DGPSpos(n), pos);
            travelttime = range/c ;

            SVpos = SV(id).DGPSpos(n);
            geomR = getRange(pos, SVpos);
            B(i) = SV(id).Rover(n).corrected_pseudorange - geomR;
            A(i, :) = [-(SVpos.x - pos.x)/geomR ...
                        -(SVpos.y - pos.y)/geomR ...
                        -(SVpos.z - pos.z)/geomR ...
                    ];
        end
        B = B';

        % Ax = B;
        x = A\B;
        pos.x = pos.x + x(1);
        pos.y = pos.y + x(2);
        pos.z = pos.z + x(3);
    end
end
fprintf('#MSG: %.0f      #IT: %.0f\n', n, it)

% Convert ECEF coordinates into geodetic coordinate system
=====
wgs84 = wgs84Ellipsoid('meters');
[pos.lat, pos.lon, pos.h] = ecef2geodetic(pos.x, pos.y, pos.z, wgs84);
pos.lat = (180/pi)*pos.lat;
pos.lon = (180/pi)*pos.lon;
pos.h = pos.h;

```

```
% Calculate Dilution Of Precision
=====
% Initialize output -----
-
dop      = zeros(1, 5);

% Calculate DOP -----
-
Q       = inv(A'*A);

DOP.GDOP = sqrt(trace(Q));
DOP.PDOP = sqrt(Q(1,1) + Q(2,2) + Q(3,3));
DOP.HDOP = sqrt(Q(1,1) + Q(2,2));
DOP.VDOP = sqrt(Q(3,3));
DOP.time = SV(activeSV(1)).Rover(n).time;
end
```

H.2.7 timing

H.2.7.1 addTime.m

```
function [ receiver ] = addTime( receiver, At )
%addTime adds a given time increment to all the fields of the input
%receiver structure.
```

```
%[ receiver ] = addTime( receiver, At )
%
%   INPUTS:
%       receiver      - structure to modify.
%       At            - time increment to add.
%
%   OUTPUT:
%       receiver      - updated structure.
%-----
```

```
%% Input struct contain different fields
=====
if isfield(receiver, 'fields')
    fields = receiver.fields;

    for f = 1:length(fields)
        field = fields{f};

        switch field
            case 'geoPos'
                for i = 1:length(receiver.geoPos)
                    receiver.geoPos(i).time =
receiver.geoPos(i).time+At;
                end
            case 'poseCEF'
                for i = 1:length(receiver.poseCEF)
                    receiver.poseCEF(i).time =
receiver.poseCEF(i).time+At;
                end
            case 'vel'
                for i = 1:length(receiver.vel)
```

```

            receiver.vel(i).time = receiver.vel(i).time+At;
        end
    case 'activeSV'
        for i = 1:length(receiver.activeSV)
            receiver.activeSV(i).time =
receiver.activeSV(i).time+At;
        end
    case 'DOP'
        for i = 1:length(receiver.DOP)
            receiver.DOP(i).time = receiver.DOP(i).time+At;
        end
    case 'distance'
        for i = 1:length(receiver.distance)
            receiver.distance(i).time =
receiver.distance(i).time+At;
        end
    otherwise
        warning('Not added time difference to field %s.',
field)
    end
end

%% Input struct is the field to add time
=====
else
    for i = 1:length(receiver)
        receiver(i).time = receiver(i).time+At;
    end
end

end

```

H.2.7.2 assignTime.m

```

function [ dataField ] = assignTime( dataField, t_msg )
%assignTime reads data field and assigns receiving time from linear
%interpolation of time-message table and message number. Data field
must
%contain subfiel numMSG. After that, replaces numMSG by subfield time.

```

```

%[ dataField ] = assignTime( dataField, t_msg )
%
% INPUTS:
%     dataField      - structure to assign time.
%     t_msg          - time-message table.
%
% OUTPUT:
%     dataField      - updated structure.
%-----
-----
```

```

%% Create numMSG array from time-message table
=====
for n = 1:length(t_msg)
    numMSG_array(n) = t_msg(n).numMSG;
end

```

```

%% Initialize auxiliar variables
=====
j = 0;

%% Read data field line by line
=====
for i = 1:length(dataField)
    i = i-j;
    if ~isempty(dataField(i).numMSG)
        if (dataField(i).numMSG > numMSG_array(1)) &&
(dataField(i).numMSG < numMSG_array(end))
            % Find upper and lower closest numMSG in time-message
            table ---
                [ lowClose, upClose, lowLoc, upLoc ] = closestValue(
dataField(i).numMSG , numMSG_array );

                % Get equivalent time to message number -----
                ----
                    lowTime = t_msg(lowLoc).t;
                    upTime = t_msg(upLoc).t;
                    if isfield(t_msg, 'TOW')
                        lowTOW = t_msg(lowLoc).TOW;
                        upTOW = t_msg(upLoc).TOW;
                    else
                        lowTOW = [];
                        upTOW = [];
                        warning('TOW not found. TOW is required to get an
accurate measure of time.')
                    end

                    % Calculate time by linear interpolation -----
                    ----
                    if upTime == lowTime
                        time = lowTime;
                        TOW = lowTOW;
                    else
                        time = lowTime + ((dataField(i).numMSG -
lowClose) / (upClose - lowClose))*(upTime - lowTime);
                        TOW = lowTOW + ((dataField(i).numMSG -
lowClose) / (upClose - lowClose))*(upTOW - lowTOW);
                    end

                    %% Assign time adn TOW
                    =====
                    dataField(i).time = time;
                    dataField(i).TOW = TOW;
                else
                    %% Remove data out of time assignment
                    =====
                    dataField(i) = [];
                    j = j+1;
                end
            else
                %% Remove data out of time assignment
                =====
                dataField(i) = [];
                j = j+1;
            end
        end
    end

```

```
%% Remove numMSG subfield
=====
% dataField = rmfield(dataField, 'numMSG');
```

```
end
```

H.2.7.3 fixTime.m

```
function [ adapted ] = fixTime( reference, unadapted, type )
%fixTime adapts unadapted input to fit reference timing.
```

```
%[ adapted ] = fixTime( reference, unadapted, type )
%
%   INPUTS:
%       reference      - structure taken as reference.
%       unadapted      - unadapted structure.
%
%   OUTPUT:
%       adapted        - adapted structure.
%-----
```

```
k = 0;
```

```
%% Create reference time array
=====
for i = 1:length(unadapted)
    time_array(i) = unadapted(i).time;
end
```

```
%% Read and Fix Time line-by-line
=====
for i = 1:length(reference)

    if time_array(1) > reference(i).time || time_array(end) <
reference(i).time
        continue
    else
        % Calculate time by linear interpolation -----
    ----
        k = k+1;
        [ lowTime, upTime, lowLoc, upLoc ] = closestValue(
reference(i).time , time_array );

        % Get interpolated value -----
        lowValue = unadapted(lowLoc);
        upValue = unadapted(upLoc);

        switch type
            case 'position' %
        =====
            if (upValue.x == lowValue.x) && (upValue.y ==
lowValue.y) && (upValue.z == lowValue.z)
                value = lowValue;
            else
                X = lowValue.x + ((reference(i).time -
lowTime)/(upTime - lowTime))*(upValue.x - lowValue.x);
```

```

        Y = lowValue.y + ((reference(i).time -
lowTime) / (upTime - lowTime)) * (upValue.y - lowValue.y);
        Z = lowValue.z + ((reference(i).time -
lowTime) / (upTime - lowTime)) * (upValue.z - lowValue.z);
    end
    adapted(k).time = reference(i).time;
    adapted(k).x = X;
    adapted(k).y = Y;
    adapted(k).z = Z;

    case 'range residuals' %
=====
    if upValue.meters == lowValue.meters
        meters = lowValue.meters;
    else
        meters = lowValue.meters + ((reference(i).time -
lowTime) / (upTime - lowTime)) * (upValue.meters - lowValue.meters);
    end
    adapted(k).time = reference(i).time;
    adapted(k).meters = meters;
end
end
end

```

H.2.7.4 unifyActiveSV.m

```

function [ activeSV ] = unifyActiveSV( time, activeSV_BSarray,
activeSV_Rarray, activeSV_BS, activeSV_R )
%unifyActiveSV unifies active SVs comparing active SVs information of
rover
%and base station receivers.

[% activeSV ] = unifyActiveSV( time, activeSV_BSarray,
activeSV_Rarray, activeSV_BS, activeSV_R )
%
% INPUTS:
%     time                  - current time.
%     activeSV_BSarray      - base station active SVs array.
%     activeSV_Rarray        - rover active SVs array.
%     activeSV_BS            - base station active SVs.
%     activeSV_R              - rover active SVs.
%
% OUTPUT:
%     activeSV               - active SVs.
%-----
-----
```

```

[ lowTime, upTime, lowLoc, upLoc ] = closestValue(time,
activeSV_BSarray);
lowDist = abs(time - lowTime);
upDist = abs(time - upTime);

if upDist < lowDist
    SV_BS = activeSV_BS(upLoc).SVID;
else
    SV_BS = activeSV_BS(lowLoc).SVID;
end

```

```
[ lowTime, upTime, lowLoc, upLoc ] = closestValue(time,
activeSV_Rarray);
lowDist = abs(time - lowTime);
upDist = abs(time - upTime);

if upDist < lowDist
    SV_R = activeSV_R(upLoc).SVID;
else
    SV_R = activeSV_R(lowLoc).SVID;
end

activeSV = [];
for i = 1:length(SV_R)
    add = SV_BS(SV_R(i) == SV_BS);
    activeSV = [activeSV add];
end
end
```

H.2.7.5 unifyTimings.m

```
function [ rover, baseStation, SV, activeSV_BS, activeSV_R ] =
unifyTimings( rover, baseStation, SV, activeSV_BS, activeSV_R,
correctionMode )
%unifyTimings coordinates time unification of different parameters
taking
%base station receiver navigation solution as reference.

[%[ rover, baseStation, SV, activeSV_BS, activeSV_R ] = unifyTimings(
rover, baseStation, SV, activeSV_BS, activeSV_R, correctionMode )
%
% INPUTS:
%     rover           - rover structure.
%     baseStation     - base station structure.
%     SV              - Space Vehicles structure.
%     activeSV_BS    - active SVs obtained from base station
receiver.
%     activeSV_R     - active SVs obtained from rover receiver.
%     correctionMode - correction mode.
%
% OUTPUTS:
%     rover          - corrected rover structure.
%     baseStation    - corrected base station structure.
%     SV             - corrected SV structure.
%     activeSV_BS   - corrected active SVs obtained from base
station
%
%     activeSV_R     - corrected active SVs obtained from rover
receiver.
%-----
=====
```

```
disp('Unifying timings...')

%% Find existing fields
=====
% Space Vehicle -----
-----
if ~isempty(SV)
    SVfields = SV(activeSV_BS(1).SVID(1)).fields;
else
```

```

        SVfields = [];
end

%% Find earliest TOW
=====
% Base Station -----
-----
BSTOW0 = baseStation.posECEF(1).TOW;

% Rover -----
-----
RTOW0 = rover.posECEF(1).TOW;

%% Timing correction
=====
if BSTOW0 < RTOW0
    At = RTOW0-BSTOW0;
    rover = addTime(rover, At);
    if ~isempty(activeSV_R)
        activeSV_R = addTime(activeSV_R, At);
    end
    for id = 1:length(SV)
        if ~isempty(SV(id).rover_residuals) && (strcmp(correctionMode,
'NavSolR') || strcmp(correctionMode, 'CLASSIC'))
            SV(id).rover_residuals = addTime(SV(id).rover_residuals,
At);
        end
    end
elseif RTOW0 < BSTOW0
    At = BSTOW0-RTOW0;
    baseStation = addTime(baseStation, At);
    if ~isempty(activeSV_BS)
        activeSV_BS = addTime(activeSV_BS, At);
    end
    for id = 1:length(SV)
        if SV(id).counter > 0 && (strcmp(correctionMode, 'NavSolR') ||
strcmp(correctionMode, 'CLASSIC'))
            for f = 1:length(SVfields)
                SVfield = SVfields{f};
                switch SVfield
                    case 'range_residuals'
                        SV(id).range_residuals =
addTime(SV(id).range_residuals, At);
                    case 'pos'
                        SV(id).pos = addTime(SV(id).pos, At);
                    otherwise
                        warning('Field %s ignored.', SVfield)
                end
            end
        end
    end
end
end

```

H.2.8 utils

H.2.8.1 check_t.m

```
function [ corrTime ] = check_t( time )
%check_t corrects time format accounting for beginning or end of week
%crossover.

%[ corrTime ] = check_t( time )
%
% INPUT:
%     time          - time in seconds
%
% OUTPUT:
%     corrTime      - corrected time (seconds)
%-----
----
```

```
half_week = 302400;      % seconds

corrTime = time;

if time > half_week
    corrTime = time - 2*half_week;
elseif time < -half_week
    corrTime = time + 2*half_week;
end

end
```

H.2.8.2 checkOrder.m

```
function [ corrected ] = checkOrder( raw )
%checkOrder corrects byte order of input messages.

%[ corrected ] = checkOrder( raw )
%
% INPUT:
%     raw          - uncorrected message.
%
% OUTPUT:
%     corrected    - corrected byte order message.
%-----
----
```

```
% Initialize variable -----
-----
corrected = [];

%% Corrects order
=====
for i = 1:2:length(raw)-1;
    corrected = [raw(i:i+1) corrected];
end
end
```

H.2.8.3 cleanRow.m

```
function [ row ] = cleanRow( row )
%cleanRow cleans rows that contains line number, hexadecimal datand
ASCII
%encoding. Line number and ASCII encoding shall be removed.

%[ row ] = cleanRow( row )

% INPUT:
%     row          - dirty row.
%
% OUTPUT:
%     row          - clean row.
%-----
-----
```

```
row(1:11) = [];
row(48:end) = [];
row(strfind(row, ' ')) = [];

end
```

H.2.8.4 closestValue.m

```
function [ lowClose, upClose, lowLoc, upLoc ] = closestValue( value,
array )
%closestValue returns upper and lower closest values of an array to a
given
%number. Returns their locations as well.

%[ lowClose, upClose, lowLoc, upLoc ] = closestValue( value, array )
%
% INPUTS:
%     value        - value to be approximated.
%     array        - input array
%
% OUTPUTS:
%     lowClose     - lower value of the array closest to value.
%     upClose      - upper value of the array closest to value.
%     lowLoc       - position in the array of lower closest value.
%     upLoc        - position in the array of upper closest value.
%-----
-----
```

```
%% Initialize variables
lowDist = 1e20;
upDist = 1e20;
lowClose = [];
upClose = [];
lowLoc = [];
upLoc = [];
```

```
%% Read array
=====
for i = 1:length(array)
    if value >= array(i) && abs(value-array(i)) < lowDist
        lowDist = abs(value-array(i));
        lowClose = array(i);
        lowLoc = i;
    end
    if array(i) >= value && abs(value-array(i)) < upDist
        upDist = abs(value-array(i));
        upClose = array(i);
        upLoc = i;
    end
end

if isempty(upLoc) && ~isempty(lowLoc)
    upLoc = lowLoc;
    upClose = lowClose;
elseif ~isempty(upLoc) && isempty(lowLoc)
    lowLoc = upLoc;
    lowClose = upClose;
end
end
```

H.2.8.5 ECEF2geodeticArray.m

```
function [ geodetic ] = ECEF2geodeticArray( ECEF )
%geodetic2ECEFarrray converts all cells of array from ECEF coordinate
system
%based in WGS-84 coordinates to geodetic.
```

```
%[ geodetic ] = ECEF2geodetic( ECEF )
%
% INPUT:
%     ECEF          - array of position in ECEF coordinates.
%
% OUTPUT:
%     geodetic      - array of position in WGS-84 coordinates.
%-----
```

```
%% Define WGS-84 Ellipsoid
=====
```

```
wgs84 = wgs84Ellipsoid('meters');
```

```
%% Applies operation line-by-line
=====
```

```
for i = 1:length(ECEF)

    % Assigns time if required -----
    %
    if isfield(ECEF, 'time')
        geodetic(i).time = ECEF(i).time;
    elseif isfield(ECEF, 't')
        geodetic(i).time = ECEF(i).t;
    end

```

```
% Convert and assigns new coordinates -----
-----
[geodetic(i).lat, geodetic(i).lon, geodetic(i).h] =
ecef2geodetic(ECEF(i).x, ECEF(i).y, ECEF(i).z, wgs84);

geodetic(i).lat = geodetic(i).lat*(180/pi);
geodetic(i).lon = geodetic(i).lon*(180/pi);
geodetic(i).h = geodetic(i).h;

end
end
```

H.2.8.6 geodetic2ECEFarray.m

```
function [ ECEF ] = geodetic2ECEFarray( geodetic )
%geodetic2ECEFarray converts all cells of array from geodetic
coordinates to
%ECEF coordinate system based in WGS-84.

%[ ECEF ] = geodetic2ECEFarray( geodetic )

%
% INPUT:
%     geodetic      - array of position in WGS-84 coordinates.
%
% OUTPUT:
%     ECEF          - array of position in ECEF coordinates.
%-----
-----

%% Define WGS-84 Ellipsoid
=====
wgs84 = wgs84Ellipsoid('meters');

%% Check possible equivalent field names
=====
if ~isfield(geodetic, 'h')
    if isfield(geodetic, 'althAE')
        for i = 1:length(geodetic)
            geodetic(i).h = geodetic(i).althAE;
        end
    elseif isfield(geodetic, 'height')
        for i = 1:length(geodetic)
            geodetic(i).h = geodetic(i).height;
        end
    end
end

%% Applies operation line-by-line
=====
for i = 1:length(geodetic)

    % Assigns time if required -----
    if isfield(geodetic, 'time')
        ECEF(i).time = geodetic(i).time;
    elseif isfield(geodetic, 't')
```

```

        ECEF(i).time = geodetic(i).t;
    end

    % Convert and assigns new coordinates -----
-----
    [ECEF(i).x, ECEF(i).y, ECEF(i).z] =
geodetic2ecef(geodetic(i).lat*(pi/180), geodetic(i).lon*(pi/180),
geodetic(i).h, wgs84);
end
end

```

H.2.8.7 hex2decimal.m

```

function [ msg ] = hex2decimal( hex )
%hex2number reads hexadecimal message and translates it into decimal
number
%byte by byte.

```

```

%[ msg ] = hex2decimal( hex )
%
% INPUT:
%     hex          - hexadecimal message.
%
% OUTPUT:
%     msg          - translated message.
%-----
-----
```

```

% Initialize variable -----
-----
msg = [];

% Check message parity -----
-----
if rem(length(hex), 2)
    error('Number of characters must be multiple of two')
end

% Get and translate byte by byte -----
-----
for i = 1:2:length(hex)
    byte = hex2dec(hex(i:i+1));
    msg = [msg byte];
end
end

```

H.2.8.8 hex2str.m

```

function [ msg ] = hex2str( hex )
%hex2str reads hexadecimal message and translates it into ASCII byte
by
%byte.

```

```

%[ msg ] = hex2str( hex )
%
% INPUT:
%     hex          - hexadecimal message.

```

```
%  
%     OUTPUT:  
%         msg          - ASCII translated message.  
%-----  
-----  
  
% Initialize variable -----  
-----  
msg = [];  
  
% Check message parity -----  
-----  
if rem(length(hex), 2)  
    msg = NaN;  
else  
    % Get and translate byte by byte -----  
    -----  
    for i = 1:2:length(hex)  
        byte = char(hex2dec(hex(i:i+1)));  
        msg = [msg byte];  
    end  
end  
end
```

H.2.8.9 twosComp2dec.m

```
function [ intNumber ] = twosComp2dec( binaryNumber )  
%twosComp2dec converts two's complement coded binary numbers to  
decimal  
%integer.  
  
%[ intNumber ] = twosComp2dec( binaryNumber )  
%  
%     INPUT:  
%         binaryNumber      - two's complement binary number.  
%  
%     OUTPUT:  
%         intNumber        - decimal integer.  
%-----  
-----  
  
% Check if the input is string -----  
-----  
if ~ischar(binaryNumber)  
    error('Input must be a string.')  
end  
  
% Convert from binary form to a decimal number -----  
-----  
intNumber = bin2dec(binaryNumber);  
  
% If the number was negative, then correct the result -----  
-----  
if binaryNumber(1) == '1'  
    intNumber = intNumber - 2^size(binaryNumber, 2);  
end  
end
```

H.2.8.10 utc2sec.m

```
function [ time ] = utc2sec( UTC )
%utc2seconds reads UTC time in format hhmmss.sss and converts it into
%seconds.

[% seconds ] = utc2seconds( UTC )

%
% INPUT:
%     UTC          - UTC time in format hhmmss.sss
%
% OUTPUT:
%     time         - time in seconds.
%-----
```

```
%% Check that UTC is string
if ~ischar(UTC)
    error('UTC time must be a string in format hhmmss.sss')
end

h = str2num(UTC(1:2));
min = str2num(UTC(3:4));
sec = str2num(UTC(5:end));

time = h*3600+min*60+sec;

end
```

I. Bibliography

- [1] Trimble Navigation. *GPS diferencial explicado claramente.* 1993
- [2] Christopher, and Elliot Kaplan. *Understanding GPS. Principles and Applications.* Norwood: Artech House, 2006.
- [3] Doberstein, Dan. *Fundamentals of GPS Receivers. A hardware approach.* NewYork: Springer, 2012.
- [4] Dennis, Nicolaj, Peter, Søren, and Kai Borre. *A software-defined GPS and Galileo receiver. A single-frequency approach.* Boston: Birkhäuser, 2007.
- [5] 3D Robotics Store – 3DR uBlox GPS with Compass Kit. <https://store.3dr.com/products/3dr-gps-ublox-with-compass>, February 2016.
- [6] GPS Information – Almanac an Ephemeris Data as used by GPS receivers. <http://gpsinformation.net/main/almanac.txt>, March 2016.
- [7] The Radio Technical Comision for Maritime Services – RTCM Standards. <http://www.rtcm.org/overview.php>, March 2016.
- [8] GPS Information – NMEA Data. <http://www.gpsinformation.org/dale/nmea.htm>, March 2016.
- [9] Global Positioning System directorate – Systems Engineering & Integration – Interface Specification – IS-GPS-200H. <http://www.gps.gov/technical/icwg/IS-GPS-200H.pdf>, April 2016.
- [10] Global Positioning System directorate – Systems Engineering & Integration – Interface Specification – IS-GPS-200G. <http://www.gps.gov/technical/icwg/IS-GPS-200G.pdf>, April 2016.
- [11] u-blox 7 Receiver Description – Including Protocol Specification V14. [https://www.u-blox.com/sites/default/files/products/documents/u-blox7-V14_ReceiverDescrProtSpec_\(GPS.G7-SW-12001\)_Public.pdf](https://www.u-blox.com/sites/default/files/products/documents/u-blox7-V14_ReceiverDescrProtSpec_(GPS.G7-SW-12001)_Public.pdf), May 2016.

