SATELLITE COMMUNICATION SYSTEMS

Simulation of a satellite receiver system for GNSS signals

Davide Tomasella (M.2037849) - davide.tomasella@studenti.unipd.it
Lorenzo Borsoi (M.2056099) - lorenzo.borsoi.1@studenti.unipd.it
Mattia Piana (M.2053041) - mattia.piana@studenti.unipd.it
Gabriele Andreetta (M.2053121) - gabriele.andreetta@studenti.unipd.it

University of Padova

Master's degree in ICT for Internet and Multimedia - Photonics
Master's degree in ICT for Internet and Multimedia - Telecommunications
Master's degree in Electronics Engineering

May 20, 2022

1 Project overview

Our project centers on developing an efficient MATLAB simulator for demodulating GNSS packets. It is part of the Satellite Communications Systems course and aims at on-the-field experience in the simulation of ICT systems and deepening our understanding of the related problems.
This project is part of a wider one that aims at emulating an entire communication system, able to simulate the exchange of packets between a Satellite and a User. The challenging context is a future GNSS constellation on the Moon with modulated signals similar to GPS and GALILEO's configuration.

We decide to develop our simulation distancing from the actual FPGA (Field Programmable Gate Array) implementation to exploit the MATLAB capability to parallelize array computations. In particular, we acquire the GNSS signal through a Parallel Code Phase Search (PCPS) and implement symbol- and segment-wise tracking and decoding. The tracking loop carries on a hybrid correlation, i.e., it performs an incoherent sum of small coherent intervals to evaluate the peak position at different delays (Code Phase tracking) and Doppler shifts (Doppler Frequency tracking). The demodulation compensates the residual Doppler envelope using a Feed-Forward (FF) Carrier Recovery with a Non-Decision-Aided (NDA) Phase Estimator before performing a coherent sum for each symbol of the coherent intervals (Doppler Residual Phase tracking).
Employing together all these techniques, we can evaluate their impact on the complete system and reach performances comparable to the ones of actual GNSS receivers without the complexity of performing real-time (sample-wise) processing.

2 Development strategy

We organize the development of our project into different phases to allow the development of the core functionalities autonomously. In the following, we will briefly describe our workflow.

1. **Initial considerations:** We discussed the aims of the project and our previous ideas about how a satellite receiver works and how we can realize its simulation. We define the working principle of our algorithm.
2. **Task division:** After a global understanding of the whole project, we split the algorithm into independent middle-level functionalities implemented with classes. We assigned each to a different person to start the development autonomously.
3. **High-level implementation:** We transcribe the simulator working principles into a high-level algorithm that calls the previously defined middle-level operations. We also decide the classes' input and output interfaces.
4. **Core functionalities' implementation:** Each class represents different middle-level functionalities and includes configuration, action, and low-level functions. We collaborated on the most challenging parts, developing and testing the code.
5. **Test of core functionalities:** We developed several test signals to cover different cases and tested the separated classes. We also evaluate how the configuration parameters impact the algorithms.
6. **Opinion sharing:** During the development, we conducted weekly meetings to present our advancement, discuss the implementation choices, and decide how to solve the problems and improve the simulator.
7. **Integration into high-level algorithm:** We then integrate the core functionalities into the high-level simulator to test the whole system and analyze how they interact.
8. **Performance evaluation:** Once the program behaves as expected, we tested a wide range of input signals to find the configuration parameters that provide the best results.
9. **Report writing:** We wrote the final report, collecting and summarizing the core functionalities and main results of the simulator.

A non-complete list of the script each group component worked on:

- **Davide Tomasella**
  `RECEIVER`, `RECEIVERTest`, `InOutInterface`, `CorrelationManager`, `TrackingManager`
- **Lorenzo Borsoi**
  `TrackingManager`, `MessageAnalyzer`
- **Mattia Piana**
  `RECEIVER`, `SignalManager`, `generateTestSignal`
- **Gabriele Andreetta**
  `CorrelationManager`, `DownconverterFilter`

3 Simulation algorithm

We implemented the entire simulation code with MATLAB, using file interfaces to acquire the input signal and parameters and save the simulation results on top of the performance plots.
The script `RECEIVER.m` contains the high-level algorithm, described in Fig. 8, and `consoleRUN.m` or `RUN.mlx` launches it after defining our configurable parameters.
Firstly, the read sampled signals are filtered to remove the high-frequency noise, and the acquisition procedure starts. Once the message synchronization pattern with the correct PRN is found, the simulation proceeds to the tracking phase, performed in parallel on multiple symbols. Here we compensate the Doppler envelope and filter the signal with a narrower bandwidth before the actual estimation of time and frequency shifts. The symbols are demodulated, and we repeat the tracking till the end of the message. Finally, we analyze the received message and save the simulation results.

The following sections describe the classes adopted to implement the core functionalities, focusing on their working principle and tunable parameters.

3.1 Requirements definition and Interfaces

We translate the requirements defined by the Scenario Manager into a list of parameters that can be provided to the simulation though a `JSON` file. In the following, we list them specifying the parameter name, type, and range, and its description:

- `fSampling`: `float`, 2-60 MHz, sampling frequency of the baseband signal (affected by Doppler shift)
- `quantizationBits`: `int`, 8/16/24/32, number of bit per I/Q sample in the binary file
- `scenarioDuration`: `float`, 1-60 s, duration of the simulated signal (determines the dimension of the `IQsamples` file)
- `SV_PRN_ID`: `int`, 1-50, PRN sequence of *Primary Codes for the E1-B Component* (see GALILEO OD SIS ICD [1])
- `CRCpolynomial`: `hex-string` (24 bit), generator polynomial G(x) for CRC value creation
- `SYNCpattern`: `bin-string` (10 bit), `0101100000`, fixed synchronization header
- `TAILpattern`: `bin-string` (6 bit) `000000`, fixed padding tail
- `SVIDlength`: `int`, 6, length of SV ID field
- `MIDlength`: `int`, 4, length of Message ID field
- `MBODYlength_ACK`: `int`, 30, length of Message Body field with ACK packets
- `CRClength`: `int`, 24, length of CRC sequence
- `nPRN_x_Symbol`: `int`, 1-10, number of PRN repetition for each symbol, determines the symbol period with the following parameters
- `nChip_x_PRN`: `int`, 4092, length of the PRN sequence of *Primary Codes for the E1-B Component* (see GALILEO OD SIS ICD [1])
- `chipRate`: `float`, 1.023 MHz, chip rate of the PRN sequence
- `maxDoppler`: `float`, 100 KHz, maximum Doppler frequency, useful to estimate the signal bandwidth and filter noise

In the same way, we also define the `JSON` file containing the simulation output, in particular, the decoded message and the log information of the algorithm:

- `version`: `string`, version of the algorithm
- `ACQUISITION_OK`: `bool`, debug log of acquisition algorithm
- `TRACKING_OK`: `bool`, debug log of the tracking algorithm
- `DEMODULATION_OK`: `bool`, debug log of the message analysis
- `SV_ID`: `string 6char`, binary ID of target satellite (demodulated)
- `message_ID`: `string 4char`, message ID (demodulated)
- `message_body`: `string 30char`, message body (demodulated)
- `CRC`: `string 24char`, CRC of the message (demodulated)
- `ACKed`: `bool`, true if the computed CRC is equal to the received one
- `isACKmessage`: `bool`, true if the `message_body` contains ACK flag (first bit is 1)
- `estimatedDopplerStart`: `float`, estimated doppler frequency after acquisition procedure
- `estimatedDopplerEnd`: `float`, estimated Doppler frequency after message demodulation (tracking procedure)
- `estimatedDelay`: `float`, estimated time delay (from the start of the `IQsamples` file) during the acquisition procedure
- `estimatedPhase`: `float`, 0 - $2\,\pi$, estimated envelope phase (at `estimatedDelay`) during the acquisition procedure

The class `InOutInterface` manages all the parameters thanks to two structures `settings` and `results` and performs the acquisition and saving of the `JSON` files. The additional file `PRNpattern.json` contains a list of all the possible PRN sequences in `hex` format.

- `createSettings(filename, prnfilename)`
  It creates the structure `InOutInterface.settings` by reading the `JSON`-formatted file `filename` using the method `readJsonFile()`. Before accepting the new configuration, the class performs a checked on the input parameters with `validateSettings()`. At the moment, the proposed requirements list has not been confirmed by the other groups, thus, we only verify that the file contains the necessary parameters. If not, we use a default version of them. Finally, the class loads `InOutInterface.PRNcode`, selected with `SV_PRN_ID`, from the file `prnfilename`.
- `saveResults(filename)`
  It saves the structure `InOutInterface.results` into a customizable `JSON`-formatted file. It automatically appends to the `filename` the creation date-time to present overriding.

3.2 Samples reading and management

The script in charge of reading and writing binary files is `SignalManager`, and it can handle 8, 16, 24, or 32 bits per sample, assuming to receive a signed data format. The first premise is that the functions inside this class treat every sample as a couple of in-phase and quadrature components. Indeed, a file with 100 samples according to `SignalManager` will contain only 50 `IQsamples`. It is due to the processing carried out in the simulator, which is done via an analytical signal, and I and Q components are combined into a "unique" sample.
The `SignalManager.IQsamples` matrix is created through `openReadCloseBinaryFile()` when a new portion of the received signal needs to be precessed. During the simulation, other classes (e.g., `DownconverterFilter`) can update the values instead of allocating new memory space. Finally, we implemented the saving method `saveToBynaryFile()` for the creation of the test signals.

- `openReadCloseBinaryFile(currentSample, nSamples)`
  It reads `nSamples` consecutive samples (both I and Q) starting from `currentSample`. If the reading exceeds the length of the binary file, a padding of zeros will be added to the reading: this is done because it facilitates the tracking script. The read data is stored inside the field `IQsamples` of the class.
- `saveToBynaryFile(samples, filename, append)`
  It saves the data `SignalManager.IQsamples` into a binary file named `filename`. In case the file is already present, if `append=true` the pre-existing file will be concatenated with the content of `IQsamples`, otherwise (`append=false`) the file will be overwritten.

3.3 Signal acquisition

The signal acquisition is performed by the MATLAB class `CorrelationManager`. The acquisition section aims to provide a first estimation of the signal parameters such as Doppler frequency and phase and time delay. These parameters will be used by the down converter that will compensate for the Doppler envelope and by the tracking algorithm as starting points.
This analysis is done considering only a part of the total signal (otherwise, the dimension of the resulting matrix would be huge). On the other hand, the number of samples must be enough to detect the sync pattern. For that reason, we chose to evaluate a signal portion of length equal to twice the sync pattern.
If the acquisition algorithm doesn't find the starting point of the information signal, the signal manager will provide the next portion of the signal.

Part 1: Calculate correlation matrix

The correlation matrix is calculated using the fast Fourier transform for each Doppler frequency inside the chosen interval

$$R(\tau) = \int r(t) \cdot \texttt{sync}(t+\tau) e^{-j2\pi f(t+\tau)} dt = \mathcal{F}^{-1}\left[\mathcal{F}[r(t)] \cdot \mathcal{F}\left[\texttt{sync}(t) e^{j2\pi f t}\right]^*\right] \tag{1}$$

with $\mathcal{F}$ Fourier transform operator. This method is known as Parallel Code Phase Search (PCPS) as described in [2]. This method parallelizes the delay search by computing one Fourier transform of the time domain signal for each frequency.
The space size is dominated by the time delay, and `MATLAB` has a built-in method to efficiently compute the FFT. They allow this method to reduce the algorithm complexity compared to the other two possible methods: the Serial Search Acquisition and the Parallel Frequency Space Search Acquisition. The first method does a serial scan of the time-frequency space, while the second scans the time delays and performs a parallel search of the frequency through FFT.

- `calcCorrelationMatrix(IQsamples, dimMatrix, maxDoppler, centralDoppler, fresolution, currentSample)`
  This method scans all the Doppler frequencies inside the interval $[\texttt{centralDoppler} - \texttt{maxDoppler}, \texttt{centralDoppler} + \texttt{maxDoppler}]$ with a frequency step given by the parameter `fresolution`. For each frequency it creates a signal that will be correlated with the input one, the signal is created starting from the PRN sequence, the `sync` code and the Doppler envelope.
  The output's correlation matrix does not have all the calculated correlations (it would be huge). On contrary, it contains the maximum correlation peak of small rectangular sub-matrices whose size depends on the parameter `dimMatrix`. There are other two outputs: the average correlation and the mean squared correlation, and these two values are relevant for the peak detection.

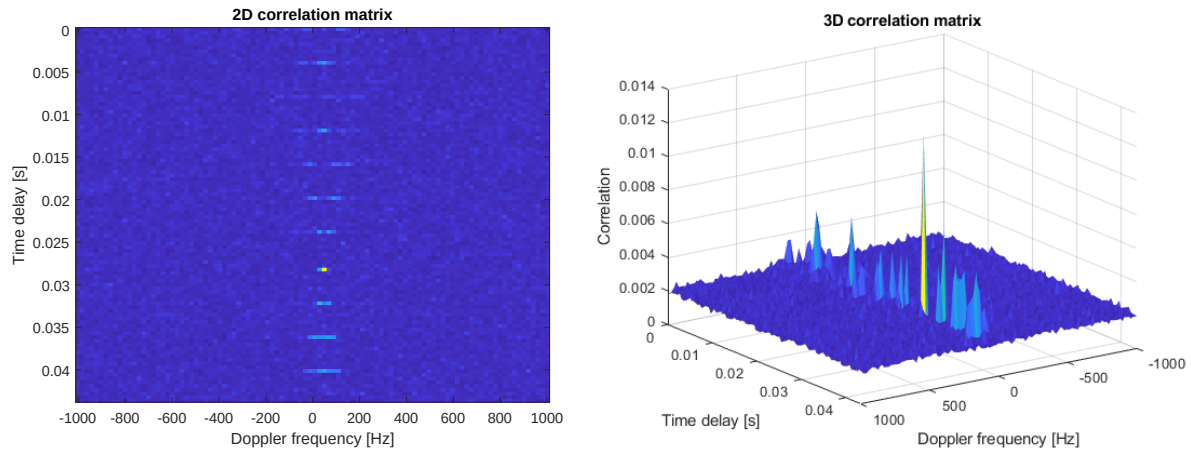


FIGURE 1: Correlation matrix between the input signal (IQ samples) and the internally generated `sync` sequence: successful acquisition.

Part 2: Search correlation peak

In this step, the algorithm finds the correlation peak, performing the equivalent `argmax` of the correlation. In Figure 1, we show the 2D and 3D correlation matrixes, with the correlation peak in yellow.
From the indices of the maximum correlation, the algorithm finds the corresponding values of the Doppler frequency and the delay. The phase of the Doppler is found exploiting the fact that the correlation between two complex exponential with same frequency and different phases is the following:

$$\int e^{j(2\pi ft+\varphi_d)} e^{-j(2\pi ft+\varphi_s)} dt = \int e^{j(\varphi_d-\varphi_s)} dt = \cos(\varphi) + j\sin(\varphi) \tag{2}$$

Therefore, the envelope phase corresponds to the angle of the correlation maximum complex number.
The method `calcCorrelationMatrix()` carry out the operations discussed in this part.

Part 3: Acquire signal

If the correlation peak overcomes a threshold, the acquisition is successful. If the signal at the input of the correlator is made by noise, there would still be a correlation maximum as output. It is due to the variance of the correlation and, thus, is meaningless because there's no signal to acquire.
If the threshold is overcomed the output parameters will be initialized: `estimatedDopplerStart`, `estimatedDelay` and `estimatedPhase`. At the end of the signal tracking, the latest estimate of the Doppler frequency is saved in the parameter `estimatedDopplerEnd`.

- `ifAcquiredFindCorrelationPeak(thresholdSTD, outInterface)`
  This method compares the maximum correlation given by the previous method with a threshold calculated in the following way:

$$th = \langle R(\tau, f) \rangle + \texttt{thresholdSTD} \cdot \sqrt{\langle R^2(\tau, f) \rangle - \langle R(\tau, f) \rangle^2} \tag{3}$$

  The peak must be above the mean value of the correlation, and higher than `thresholdSTD` times the standard deviation of the correlation. If the peak overcomes that value, the parameters of the signal acquired that are handled by the object `InOutInterface` will be updated. In Figure 2 it's shown an example of correlation matrix in the case of acquisition failed: there is no evident peak of correlation both on 2D and 3D plot.

The parameter `thresholdSTD` has been chosen after performing some tests of the correlator in the case of constant Doppler, in which we calculate the number of standard deviations between the correlation peak and the mean and take into account which tests gave a correct result. The best value that allows discriminating the signal from the noise is `thresholdSTD=4`.

Part 4: Monitor correlation peak position

The correlation peak updating is done during tracking, the tracking algorithm starts from the initial values of Doppler frequency, phase and delay, it searches for the new correlation peak, and it updates the new values through the method `updateCorrelationPeak()`.

- `updateCorrelationPeak(new_SamplesChipPeriod, advancement_startingSample)`
  When this method is called, the new values of Doppler frequency, phase, and delay are memorized.

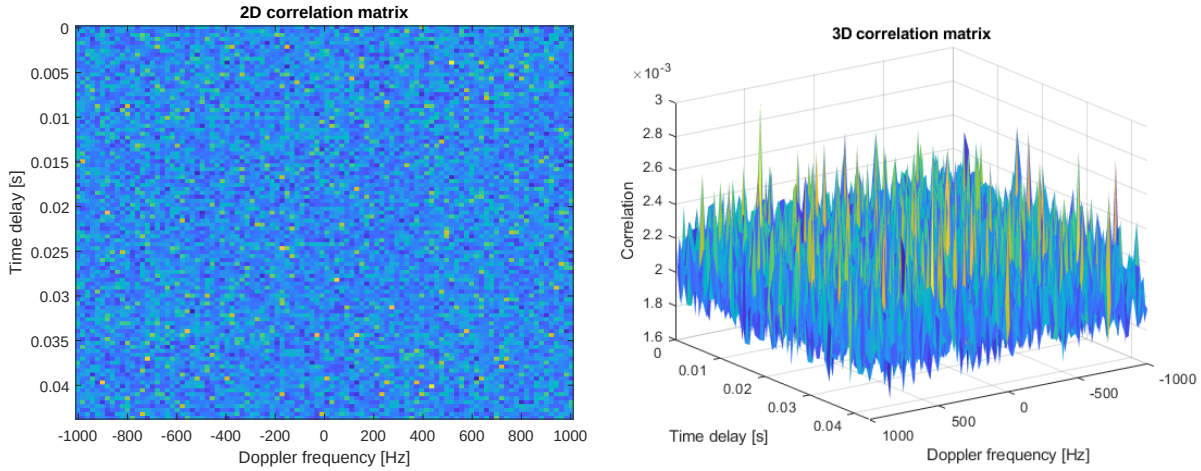


FIGURE 2: Correlation matrix between the input signal (IQ samples) and the internally generated `sync` sequence: acquisition failed.

3.4 Down-conversion and filtering

The down-conversion and filtering class `DownconverterFilter` is needed to reduce the noise level added to the input signal and compensate for the Doppler envelope.

Part 1: Signal filtering

The filter is a low-pass Butterworth filter whose attenuation and bandwidth are decided by the user. The bandwidth depends on the chip frequency and the Doppler frequency, while the attenuation depends on the number of lobes of the spectrum we want to maintain.
In Figure 3-4, we show the effects of the two types of filters: the first filter cuts all the frequencies outside the first lobe, while the second one does the same with three lobes instead of one. In the first case, there is a high noise reduction, but the signal rise time of the is reduced, too. In the second one, the signal rise time is conserved, but the noise is more powerfull.

- `configFilter(ripple_dB, attenuation_dB_dec, fSampling)`
  With this method, the user decides the passband ripple and the attenuation.

- `downFilter(reader, passBand, chipFrequency)`
  This method filters the input signals and compensate for the delay caused by the filter. It modifies the samples stored in `reader.IQsamples` allocating only temporary memory.

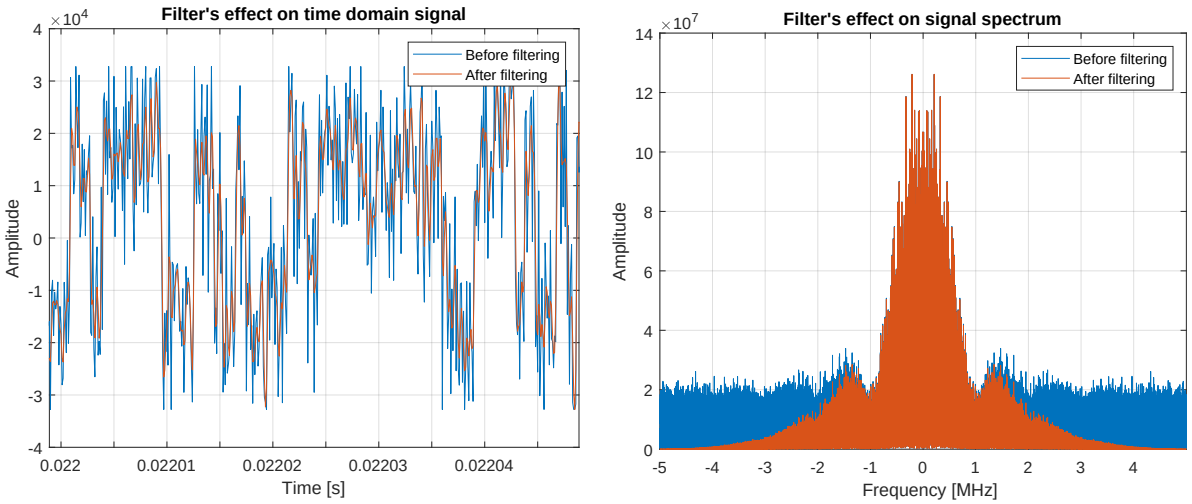


FIGURE 3: One-lobe filter effects on the time domain signal and the spectrum

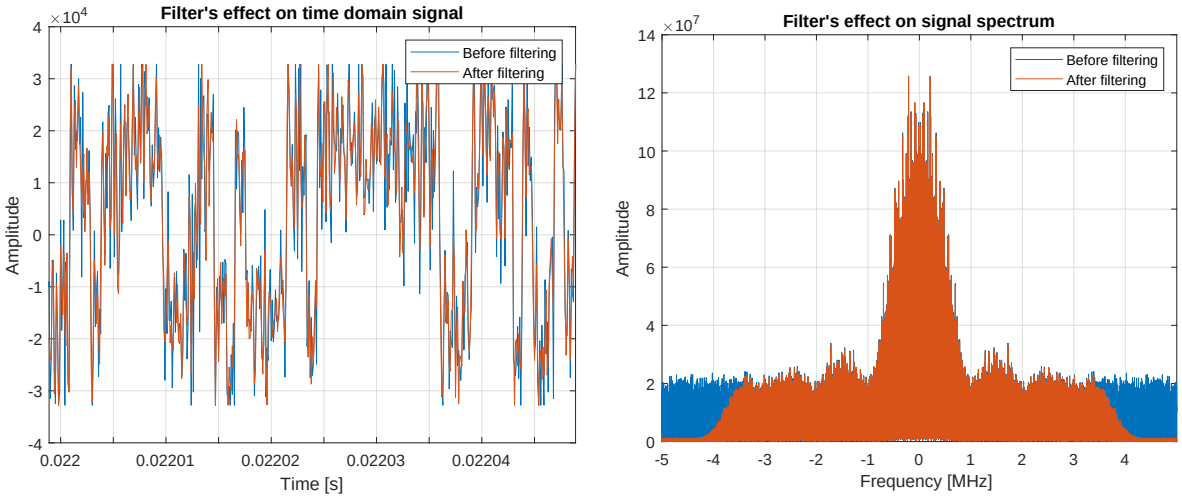


FIGURE 4: Three-lobes filter effects on the time domain signal and the spectrum

Part 2: Doppler envelope compensation

We compensate the Doppler by multiplying the input complex signal with a complex exponential with opposite frequency and phase.

$$s(t) = r(t) \cdot \exp\left[-j(2\pi f_d t - \varphi)\right] \tag{4}$$

Since the transmitted message contains only the in-phase component (BPSK modulation), the quadrature one should be zero because if the Doppler envelope is perfectly compensated, the signal return real, i.e., the spectrum becomes symmetrical.
In Figure 5, the effect of the resolution of the tracking algorithm is visible. In fact, the signal after the envelope compensation has a residual Doppler.

- `downConverter(reader, fdoppler, delay, phase)`
  This method performs the envelope compensation by multiplying the input In-phase and Quadrature signals (`reader.IQsamples`) with sine and cosine signals. The output is divided into In-phase and Quadrature components, too. The quadrature should be zero.

$$\begin{cases} Is(t) = Ir(t) \cdot \cos(2\pi f_d t + \varphi) - Qr(t) \cdot \sin(2\pi f_d t + \varphi) \\ Qs(t) = Ir(t) \cdot \sin(2\pi f_d t + \varphi) + Qr(t) \cdot \cos(2\pi f_d t + \varphi) \end{cases} \tag{5}$$

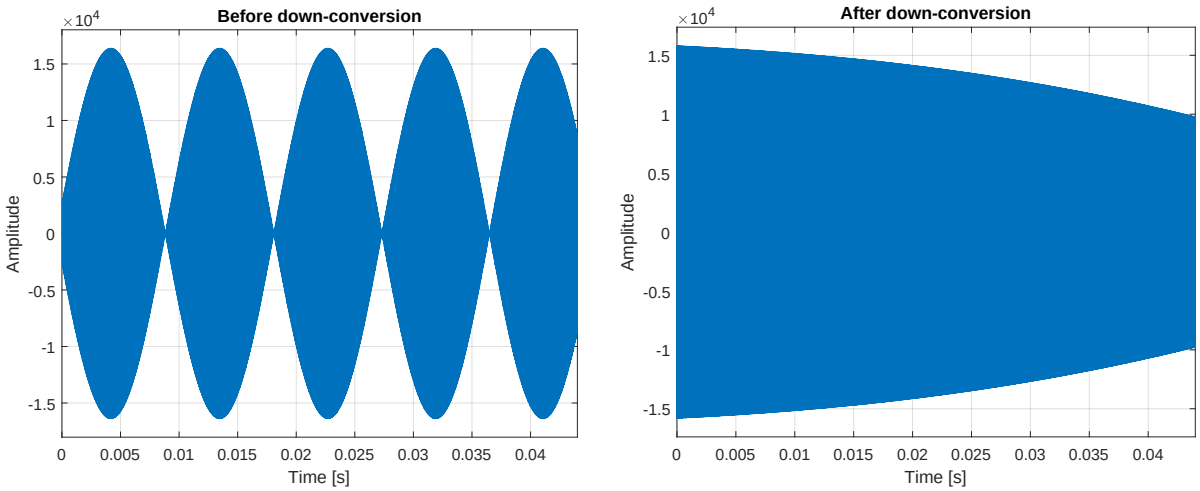


FIGURE 5: Doppler envelope compensation of the input signal, there is a residual envelope due to the resolution of the frequency estimation during correlation and tracking

3.5 Signal tracking

The class `TrackingManager` is in charge of refining the estimation of signal parameters such as Doppler phase and frequency and performs Code Delay evaluation. These parameters will update the old ones saved in the class `CorrelationManager`. Finally, it compensates for the residual envelope phase introduced by the down converter correctly demodulating it.
In Figure 6, we show an example of a successfully tracked and demodulated bit stream. We observe a steep peak in the 3D correlation matrix and the two characteristic clouds of demodulated symbols.

- `decodeOptimumShift(inSamples, segmentSize, shifts_delayPRN, shifts_nSamples_x_symbolPeriod, shifts_nSamples_x_chipPeriod, nCoherentFractions, outInterface)`
  It analyzes the down converted In-Phase and Quadrature signals (`IQsamples`), corresponding to `segmentSize` symbols. It performs carrier and code parameters evaluation, according to preselected Code Delay shifts (`shifts_delayPRN`) in the order of some chip fractions and Doppler Frequency shifts (`shifts_nSamples_x_chipPeriod`) which depend on both `chipRate` and `segmentSize`.
  To realize the signal tracker, we exploit In-phase & Quadrature multicorrelation, non-coherent and coherent integrations. Coherent integration is executed on a symbol period fraction chosen by `nCoherentFractions`. Eventually, `TRACKING_OK` flag saves the tracking outcome in `InOutInterface.results`.

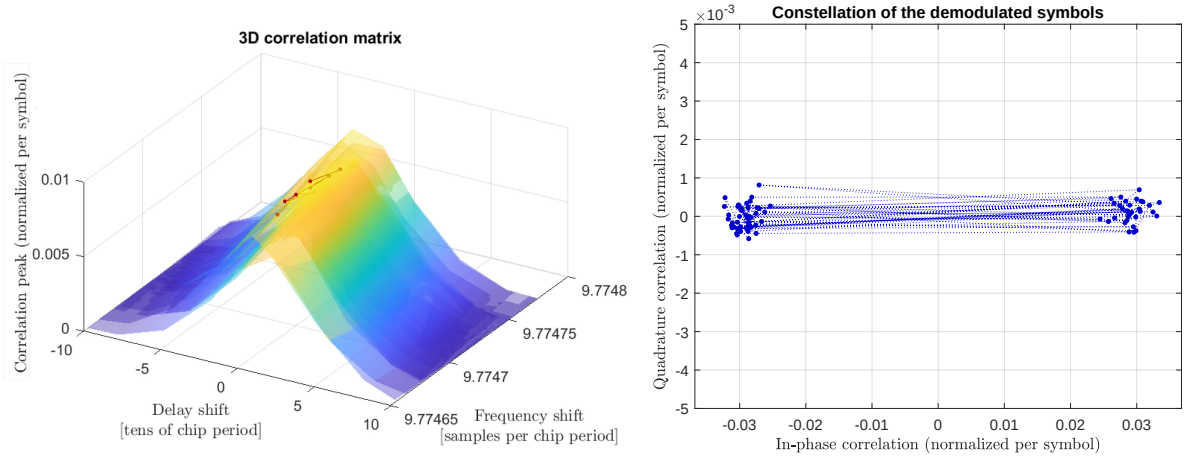


FIGURE 6: On the left, the correlation matrix between the input signal (`IQsamples`) and the generated PRN sequences. On the right, the constellation of the demodulated symbols. Successful tracking and demodulation.

Step 1: Generate sampled PRN sequence

The aim of this section is to create PRN patterns for each Code Delay and Doppler Frequency shift combination. Consequently, the `IQsamples` length is adapted to the originated PRN matrix.

- `getPRNFromChipPeriods(shifts_nSamples_x_chipPeriod, windowSize)`
  This method creates a matrix that contains the PRN sequences, one for each `shifts_nSamples_x_chipPeriod`, then, we repeat them according to the number of PRN in one symbol (`nPRN_x_Symbol`) and the number of symbols we are considering (`segmentSize`). The PRN code is 4092 chips long, and each chip has several samples decided by sampling frequency `fSampling` and the Doppler Frequency shifts. The resampling is performed by constant interpolation with the `MATLAB` function *interp1()*.
- `adaptSamplesToPRNlength(IQsamples, shifts_delayPRN, PRNlength)`
  When this method is called, the `IQsamples` length is adapted to the dimension of the PRN sequences matrix. It exploits zero padding.
- `createShiftedPRN(PRNsampled, shifts_delayPRN)`
  This method implements as many circular shifts as the `shifts_delayPRN` vector requires for each frequency-shifted PRN sequence. This operation creates the sampled PRN sequence that accounts for the Doppler frequency and Code Delay shifts.

Step 2: Calculate best time and frequency shift

This section is in charge of In-phase & Quadrature multi-correlation via non-coherent and coherent integrations. Finally, it evaluates the best-shifted PRN sequence in terms of maximum correlation with the received signal.
In Figure 7, we show an example of an unsuccessfully tracked and demodulated bit stream due to a low C/N_0. In the 3D correlation matrix plot, we observe a flattening of the correlation peak that corresponds to a loss of tracking. As a consequence, the constellation of the demodulated symbols is disrupted.

- `normMultiply(shiftedPRNsampled, mySamples, segmentSize)`
  This method generates two correlation matrices, for I and Q components, from the shifted PRN sequences and `IQsamples`, by sample-wise multiplication. Then, it normalizes each outcome by the specific number of In-phase or Quadrature samples.

- `sumOverCoherentFraction(corr, shifts_delayPRN, shifts_nSamples_x_symbolPeriod, nCoherentFractions, windowSize)`
  It performs the coherent integration for In-phase and Quadrature correlation matrices, respectively. In particular, all the correlation values within the same coherent fraction are summed up. This operation is repeated for each In-phase and Quadrature correlation vector.

As the last step, we do non-coherent integration of the coherent portions. In other words, it performs the squared sum between In-phase and Quadrature correlation vectors that originate from the same shifted PRN sequence. As a result, we obtain a single value for each Code Delay and Doppler Frequency shift combination, which conveys all the information on the correlation. The higher this correlation, the better the alignment.
Eventually, Code Delay and Doppler Frequency shifts of the maximum value overwrite the old ones in `CorrelationManager` with `updateCorrelationPeak()`.

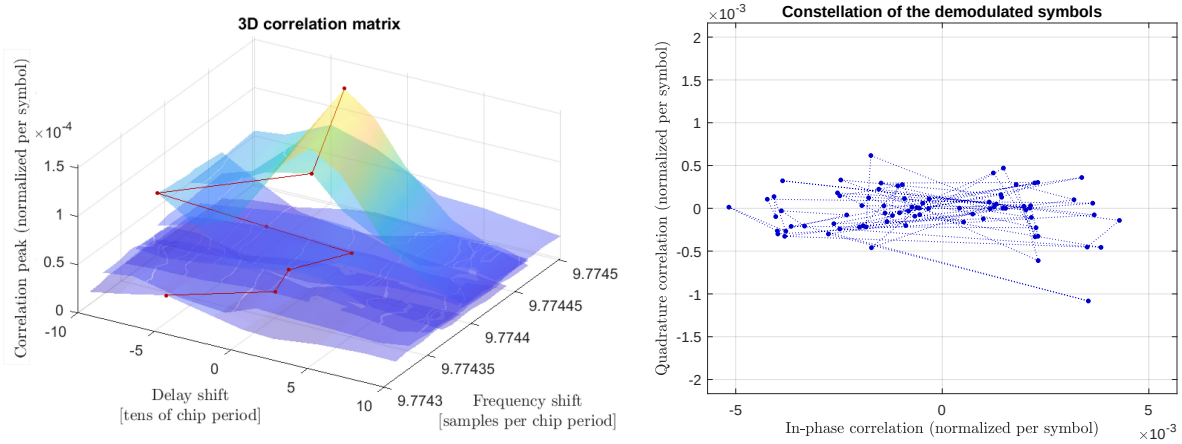


FIGURE 7: On the left, the correlation matrix between the input signal (`IQsamples`) and the generated PRN sequences. On the right, the constellation of the demodulated symbols. Unsuccessful tracking and demodulation.

Step 3: Evaluate residual phase

In addition to tracking the signal, the `TrackingManager` implements a compensation of the residual Doppler envelope originated from a non-perfect estimation of the Doppler frequency (or a variation) in the previous iteration. It is another tracking algorithm (performed internally to the class) of the residual envelope phase that behaves as an FF scheme.

- `compensateResidualEnvelope(bestCoherentCorrI, bestCoherentCorrQ)`
  We evaluate the residual envelope phase by the `atan` of the complex coherent correlation components, and we invert it, minimizing the correlation in the Quadrature component. The algorithm cannot discriminate a pi-shift, making it insensible to the modulated symbol, i.e., Non-Decision Aided (NDA) Phase Estimation. On the other hand, we can associate with confidence only the phases within a $\pm\pi/2$ shift from the previous one.
  We track the phase variations with a filter and impose a maximum shift of $\pi/2$ between subsequent samples to follow the phase drift beyond the confidence interval. Despite that, the phase noise at low received power ($\mathrm{SNR} < -30\,\mathrm{dB}$) leads to excursions larger than the $\pi/2$ and demodulation errors.

Step 4: Demodulate symbols

To perform demodulation, we take the In-phase & Quadrature coherent correlation vector, with compensated residual envelope and best correlation outcome. After the coherent sum, we evaluate the correlation sign related to each symbol.

- `sumFractionsOverSymbols(rotatedCoherentCorr, nCoherentFractions)`
  This method sums the `nCoherentFractions` coherent fractions inside each symbol period, for all symbols considered by `segmentSize`. The result is thus the coherent correlation over the symbols.

The flag `TRACKING_OK` checks if the obtained constellation of symbols is within a $\pi/4$ discrepancy from zero and π. As the final step, we map the sign of the correlation In-phase component ($-$ or $+$) into the demodulated symbols -1 and $+1$.

3.6 Message decoding

The class `MessageAnalyzer` is in charge of splitting and reading each section of the received message. In particular, the message structure includes Sync pattern, Space Vehicle ID (SV ID), Message ID, Message Body, Cyclic redundancy check (CRC) sequence, and Tail pattern. Afterward, we verify its correctness; if some error occurs, a specific warning message appears.

- `analyzeMessage(decodedSymbols, outInterface)`
  It reads the symbols previously decoded by `decodeOptimumShift.m` of the entire message, then, we call `validateAndSplitMessage()` and `calcChecksum()` methods; some verifications follow:

  - `isACKmessage`: `bool`, checks if the first bit of `Message Body` equals one;
  - `ACKed`: `bool`, checks if the CRC remainder is zero, if so, the `Message Body` is correctly received;
  - `DEMODULATION_OK`: `bool`, checks if `ACKed` is `true` and the message is valid, otherwise, a warning message with the received `CRC message` and CRC remainder appears.

  Finally, it saves the split message, the `isACKmessage` and `ACKed` flags in `InOutInterface.results`.

Step 1: Validate message

Here, we split the received bit-string according to the GNSS message standards[1], also, it performs some verifications.

- `validateAndSplitMessage(decodedSymbols)`
  In this method, we verify if the received message length is the expected one (80 bits), then, we validate and extract the message sections in the following order:

  - *Sync pattern (10 bits)*: since it is known, we can verify its correctness. A warning message appears if incorrect;
  - *SV ID (6 bits)*: each satellite has its own PRN sequence, hence, since we evaluate it through the `CorrelationManager.m`, we can verify if it's the expected one. If not, a warning message appears;
  - *Message ID (4 bits) and Body (30 bits)*: The actual message we want to decode;
  - *CRC message (24 bits)*: computed only for `Message ID` and `Message Body`, it is used to detect the reception of corrupted data;
  - *Tail pattern (6 bits)*: a bit-string of six zeros; if incorrectly received, a warning message appears.

Step 2: Calculate and check CRC

Twenty-four bits of CRC parity provides protection against burst as well as random errors with a probability of undetected error $\mathcal{P}_{UE} \leq 2^{-24}$ for all channels with BER ≤ 0.5 [3]. The CRC message is a sequence of 24 parity bits $(p_1, p_2, ..., p_{24})$ calculated in the forward direction on a given message, which is a sequence of information bits $(m_1, m_2, ..., m_{34})$, using a seed of 24 zeros. It is done through a CRC key of 25 bits, generated by the polynomial (using binary polynomial algebra)

$$\begin{cases} G(X) = (1+X)P(X) \\ P(X) = X^{23} + X^{17} + X^{13} + X^{12} + X^{11} + X^9 + X^8 + X^7 + X^5 + X^3 + 1 \end{cases} \tag{6}$$

- `calcChecksum(messageToCheck)`
  The CRC algorithm performs the `bit-xor` calculation between the information message (*Message ID*, *Message body* and *CRC message*) and the CRC key (which is the above $G(X)$). The binary polynomial division is calculated, in the forward direction, by aligning the first non-zero bits at each step. The algorithm ends when the CRC remainder is a bit-string of zeros (`ACKed` condition). Otherwise, it runs until the CRC key reaches the last information bit.

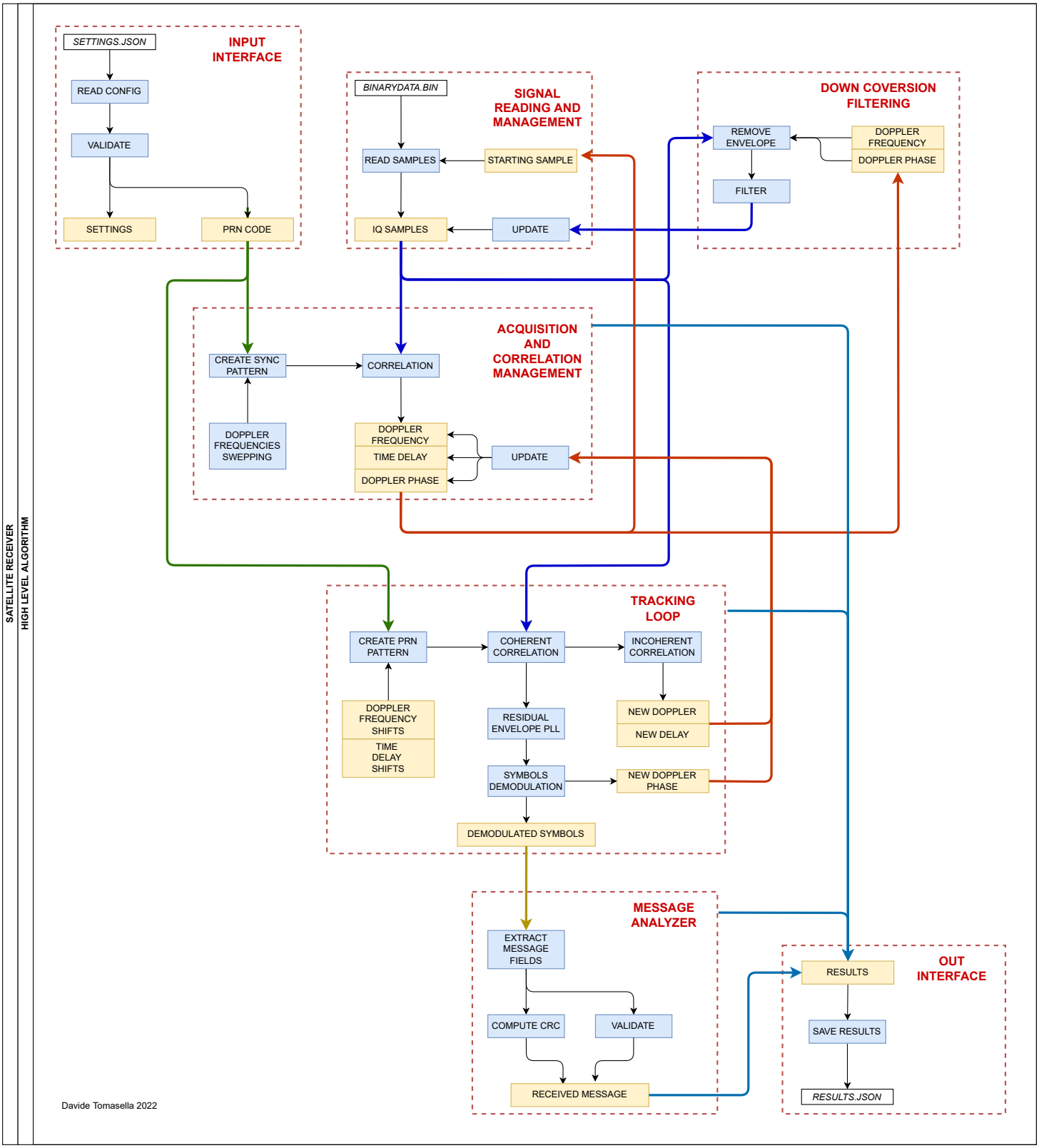


FIGURE 8: Scheme of the high-level steps of the simulation algorithm.

4 Testing and results

In the development of the different classes, we identify which parameters have an impact on the behavior of the whole simulator, and we decide to characterize the performances with different combinations. The considered parameters are the followings.

- `filterBandMultiplier` (f_B): $[0, 1, 3]$, it defines the bandwidth of the filter in `DownconverterFilter`, as a multiple of the modulation bandwidth. When it is 0, no filter is used. The filter band also defines the slope `filAttenuation_dB_dec`, needed to select 1 or 3 lobes of the `sync`-spectrum, equals to $390\,\mathrm{dB/dec}$ and $2990\,\mathrm{dB/dec}$.
- `reducedMaxDoppler` (r_D): $[100, 500, 1000]$, it defines the maximum Doppler frequency to be searched around the central one `centralDoppler`. Since the tests perform only 100 frequency evaluations in `CorrelationManager`, it defines the resolution and thus error in the Doppler estimation after the acquisition procedure. In particular, we measure an error of about $0.5\,\mathrm{Hz}$, $2.5\,\mathrm{Hz}$, and $7.5\,\mathrm{Hz}$ after before the tracking loop.
- `ppSegmentSize` (ss): $[1, 2, 5, 10]$, it is the number of symbols handled together in one iteration of the tracking system, and the size of the segment analyzed by `TrackingManager`. Using longer time intervals improves the correlation and the tracking by reducing the noise, but it also increases the local residual-envelope phase error in the case of variable Doppler.
- `nCoherentFractions` (n_C): $[1, 3, 5]$, it is the number of fractions in which we split each symbol for the coherent correlation phase in `TrackingManager`. Larger values improve the tracking of the residual envelope but make the system less robust to noise, i.e., phase noise in symbol demodulation.

The performance evaluation takes into account different levels of noise power and Doppler variation Δf during the message marking which received signal is correctly acquired, tracked and demodulated. In particular, we consider a linear range for the SNR from $-40\,\mathrm{dB}$ to $-14\,\mathrm{dB}$ that is equivalent to C/N_0 from $30\,\mathrm{dB\,Hz}$ to $56\,\mathrm{dB\,Hz}$, and a quadratic range for Δf from 0 to $1.3\,\mathrm{Hz/symbol}$ ($1.3\,\mathrm{kHz/s}$) with an initial Doppler frequency equal to $45.52\,\mathrm{Hz}$.
The class `RECEIVERTest` implements the Unit tests directly inside `MATLAB` by running all the possible configurations of the `TestParameter`. Moreover, we run all the tests without adding noise to the signal, verifying the integrity of our code, for a total of over $20\,\mathrm{k}$ tests.
Before presenting the results, we briefly discuss how we generate the test signals though a signal-oriented model of a GNSS transmitter emulator.

4.1 Test signals generation

The script `generateTestSignal`, as the name suggests, deals with the generation of the signals used to test both tracking and acquisition scripts and to estimate the performance of the simulator. The signals it allows to generate are heterogeneous, and each has its parameters handled by a struct whose fields will be clarified along with the code design explanation.

Step 1: Message definition and encoding

The first step we need to take is defining the information bits to be sent, following the message structure from ground to satellite. Then we proceed in adding random bits in front and tail that will be useful to check whether the acquisition with the sync pattern works. Finally, we perform a BPSK modulation on the bits by simply converting the bits into $+1$ and -1 symbols (power is set later). Useful parameters:

- `outBits_Length`: number of random bits to be added at the beginning and at the end of the packet.

Step 2: PRN implementation

What we do here is multiply (one of) the GAL E1B PRN [1], converted into symbols, by the sequence of bits of step 1. Moreover, we add some extra PRN chips (4092 at maximum) to see the effects these bits may have on the correlations performed later. We can see in Figure 9 a portion of the signal modulated by the PRN. Useful parameters:

- `outPRN_Length`: number of PRN symbols to be added in front and after the multiplication by the PRN.

Step 3: Doppler creation

Since the satellite is not "moon-synchronous" the speed seen by the receiver fixed on the moon's surface is changing. This change will cause a varying Doppler since it depends on the relative speed between the satellite and receiver. With this in mind, we have decided to generate signals with different Doppler behaviors resumed in Figure 9.
Useful parameters:

- `dstart`: starting Doppler frequency.
- `dend`: last Doppler frequency.
- `dmode`: available Doppler behavior are 1=linear, 2=triangular, 3=cosine, 4=quadratic.
- `deveryChip`: boolean variable, if set to true the Doppler increment (or decrement) will be added every chip period, otherwise it will be added every symbol period.

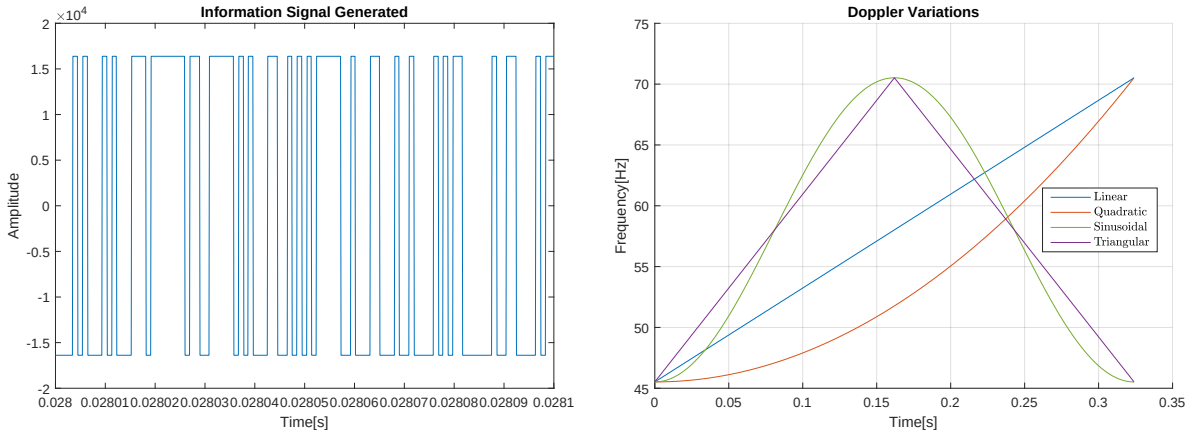


FIGURE 9: Information signal modulated by the PRN and different Doppler behaviours

Step 4: Noise creation

To build a realistic signal, Gaussian Noise needs to be added. We adjoin noise inside the generated signal and noise samples in front/tails of it, emulating a further delay at the reception. Useful parameters:

- `inNoise`: it is the standard deviation $\sigma = \sqrt{N_0}$ of the Gaussian Noise normalized to the maximum amplitude of the information signal, i.e., $2\hat{}(\texttt{quantizationBits} - 2)$. We remark that the noise power is equally divided into the In-phase and Quadrature components.
- `outNoise_Length`: number of pure noise samples added before and after the useful signal.
- `outNoise`: it is equivalent to `inNoise`, but it refers to the noise before and after the transmitted message.

Step 5: Signal and Binary file generation

The Doppler causes not only the presence of a sinusoidal envelope that the down-conversion at the nominal frequency cannot resolve (since the Doppler is varying), but also the shrinking/stretching of the symbol's period. We took this effect into account by generating symbols whose duration is variable according to the Doppler they experience. Since the sampling frequency is fixed, a symbol time reduction (caused by a positive Doppler) will result in a lower number of samples available of that symbol (for a negative Doppler, we have the opposite effect). Here we also add the attenuation of the signal, and we exploit the class `SignalManager` to produce a binary file containing the I/Q samples, emulating the output of a sampling operation of the analytic signal. On the left of Figure 10 we can see how the Doppler causes a Power Exchange among the I/Q components of the signal, while on the right, we can notice that a time-varying Doppler causes a change in the frequency of the envelope that modulates the In-Phase component. The useful parameters are reported below.

- `envelope`: set to 1 to add the sinusoidal envelope caused by the Doppler on top of the signal.
- `powerReduce`: exponent of the signal's amplitude attenuation with respect to the maximum amplitude, that is $2\hat{}$(`quantizationBits` $-\ 2$). E.g., if we set 3, the signal is reduced by $2^3 = 8$ times.
- `name`: name of the binary file where the generated signal will be saved.

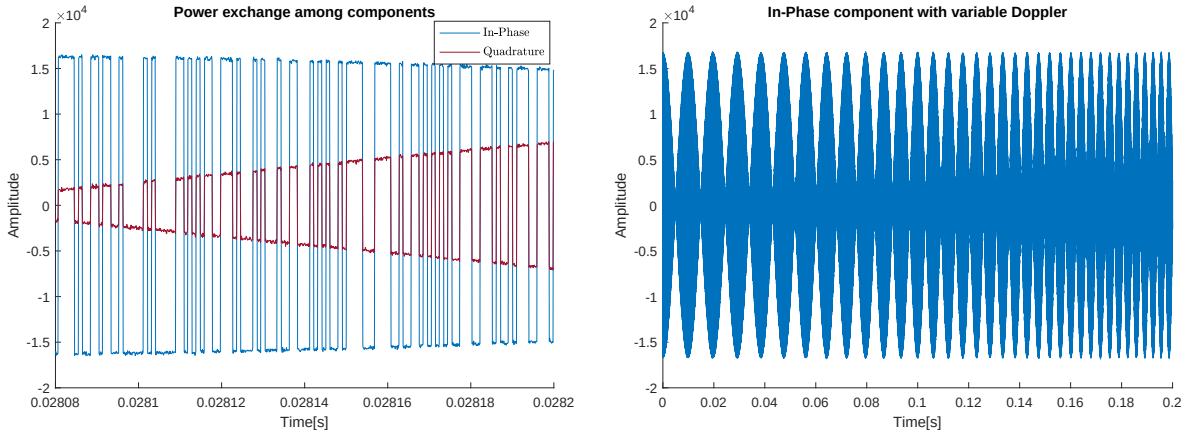


FIGURE 10: I/Q power exchange and varying Doppler effect

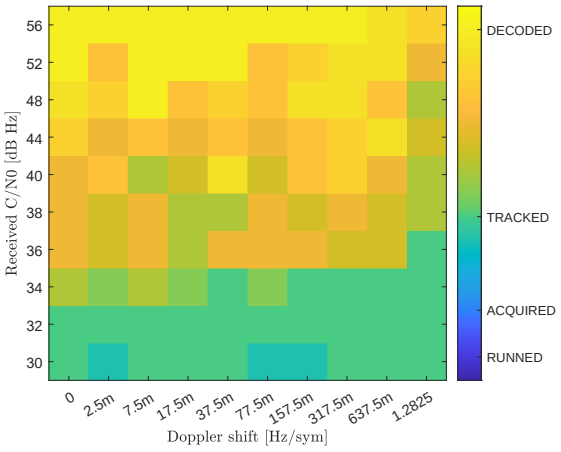
4.2 Result discussion

We run the test over multiple hours, generating a temporary signal with random noise (and always the same random message) and performing the simulation without graphical outputs. We estimate the performance of our algorithm on an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz with an integrated Graphic Card. The Signal Generation lasts about $0.9\,\mathrm{s}$, the more demanding Signal Acquisition takes about $8.5\,\mathrm{s}$, the Signal Tracking completes in about 4.0-$5.6\,\mathrm{s}$ (increasing with `ppSegmentSize`), the Signal Filtering and Down Conversion involves about $0.3\,\mathrm{s}$ each.
Since the test results are numerous, we develop a strategy to summarize them. In particular, in the following Figures, we present some matrices reporting the result status, as defined in the output file, as a function of the received SNR and Doppler Variations. Each matrix cell contains the average results over all the simulations sharing the same chosen configuration. It is important to remark that we only ran one simulation per configuration, leading to noisy results that need some interpretation.

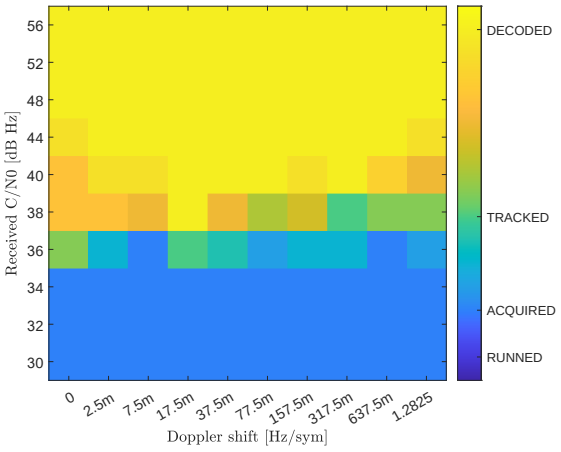
In Figure 11a, we show the reference results with only 1 `ppSegmentSize` and 1 `nCoherentFractions`, average with all the chosen filter configurations and initial Doppler estimation error. As we can see, the results are poor, i.e., most of the signal demodulated failed due to non-compensated phase errors.
From this base condition, we can move to Figure 11b where using 3 `nCoherentFractions` per symbol to determine the modulated bit improves the performance stability. It also reduces the coherence interval and thus the compensation of the white noise, preventing the tracking of the low-power signals.
On the other hand, if we double the `ppSegmentSize` (2 symbols) (Figure 11c), the noise is averaged out, expanding the lower bound. In Figure 11d, we report the results when `ppSegmentSize` $= 2$ `nCoherentFractions` $= 3$. Here we have a contraction of the power limit as in Figure 11b, and an increment in the result variance due to more noisy residual phase compensations.
Moreover, keeping 1 coherent fraction per symbol, i.e., summing incoherent only the contribution from different symbols and increasing the length of the parallelized segment to 5 and 10, as in Figure 11e-11f, limits the highest Doppler drift we can track. It originates from the maximum shift in the Doppler frequency we handle during the tracking procedure for each segment ($1\,\mathrm{Hz}$). The wrong demodulation indicates that the residual envelope, i.e., incorrect estimation of the Doppler frequency, varies too quickly.
From this analysis, we observe that the best performances, obtained with `ppSegmentSize` $= 2 - 5$ and `nCoherentFractions` $= 1$, handle signals with up to $C/N_0 = 36\,\mathrm{dB\,Hz}$ and up to $\Delta f \approx 0.6 - 0.1\,\mathrm{Hz/symbol}$.

If we now consider Figure 12, we study the effects of the filter and the acquisition's Doppler resolution on the previous best results.
In the left column, we show the performance with the different filter configurations. The no-filter results (Figure 12a) are improved in terms of lower power bound by introducing a one-lobe (12c) and three-lobe (12e) filter. We also notice that the latter can handle lower Doppler drift, probably due to the addition of distortion to the still noisy signals.
On the right column, we analyze the Doppler resolution passing from $2\,\mathrm{Hz}$ (Figure 12b), to $10\,\mathrm{Hz}$ (Figure 12d), to $20\,\mathrm{Hz}$ (Figure 12f). Here, we cannot extrapolate significant differences among the three indices because the first tracking iteration entirely compensates for the initial frequency error. Moreover, with high Doppler shifts, the additional error introduced by averaging the Doppler estimation over 10 symbols ($\sim 5\,\mathrm{Hz}$) is the dominant contribution.

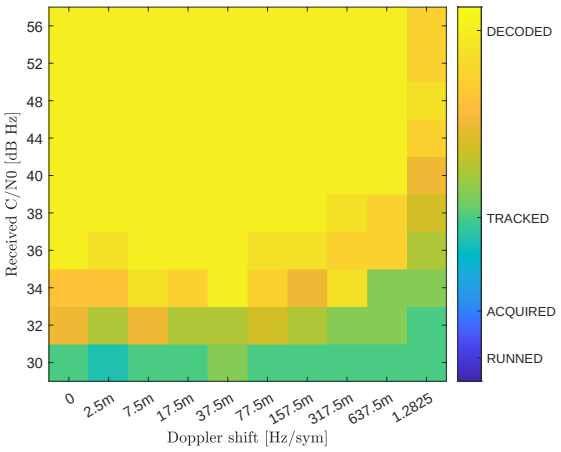
Finally, in Figure 13 we compare the performances with the best `filterBandMultiplier` and `reducedMaxDoppler` configurations between splitting the symbols into smaller coherent fractions and processing more at each tracking step. The latter solution is better and shows reliable demodulation with all the considered Doppler drifts and up to $C/N_0 \approx 34\,\mathrm{dB\,Hz}$, and with a certain degree of probability up to $30\,\mathrm{dB\,Hz}$.
The crucial limit is the tracking of the residual Doppler phase since we choose to compensate for very high-frequency drifts, compared to the typical GPS values ($2\,\mathrm{mHz/symbol}$ with stationary target) because of the more dynamical environment we expect on the Moon. The introduced limits on the duration of the phase estimation averaging interval (lower than one symbol) affect the minimum C/N_0 handled by the system more than expected.



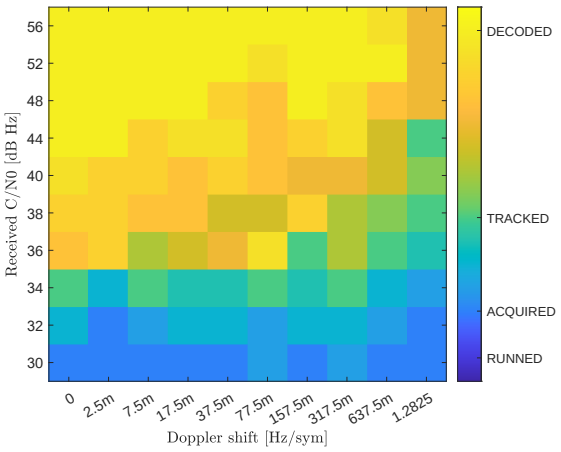
(A) $ss = 1$, $n_C = 1$



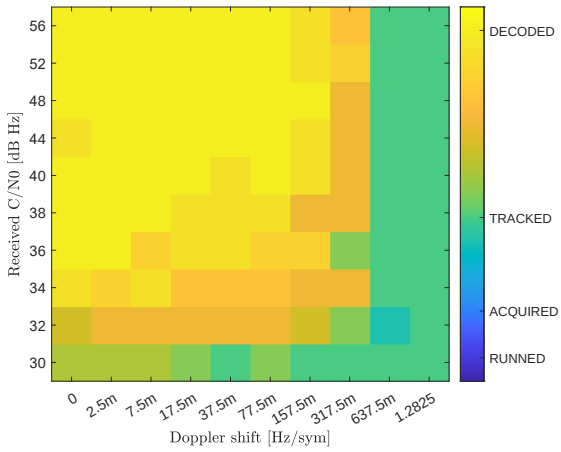
(B) $ss = 1$, $n_C = 3$



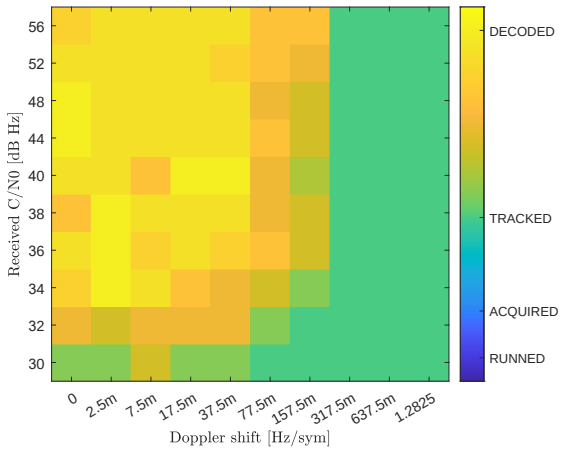
(C) $ss = 2$, $n_C = 1$



(D) $ss = 2$, $n_C = 3$

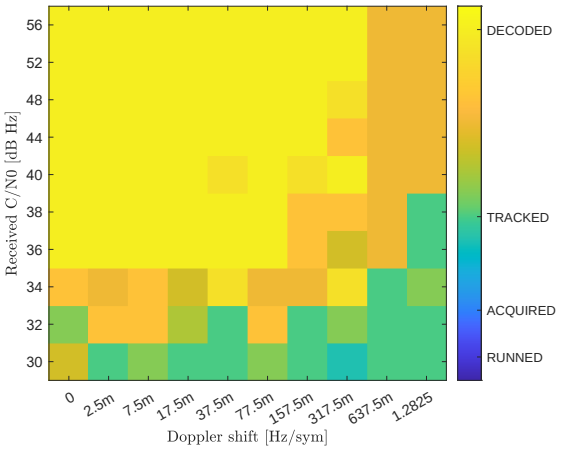


(E) $ss = 5$, $n_C = 1$

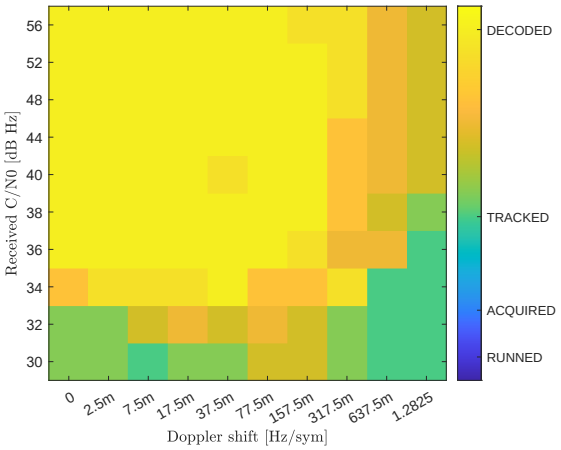


(F) $ss = 10$, $n_C = 1$

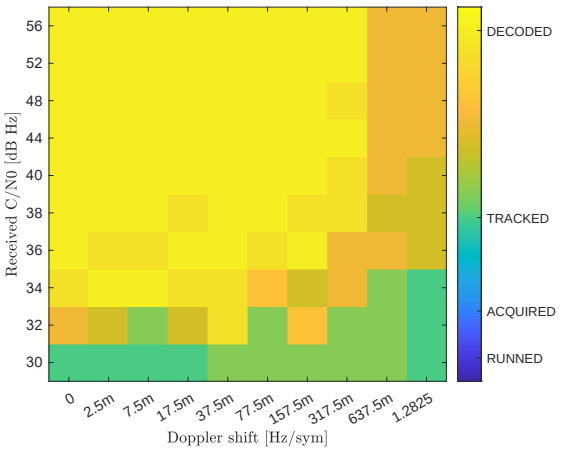
FIGURE 11: Analysis over all the filter bandwidth f_B and the searching Doppler range r_D.



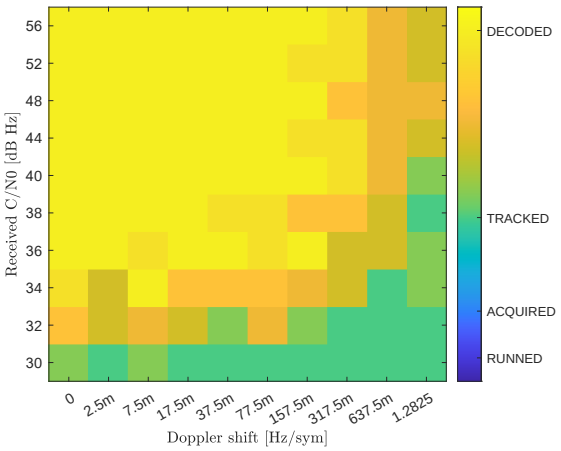
(A) $f_B = 0$, $ss = 2 - 5$, $n_C = 1$



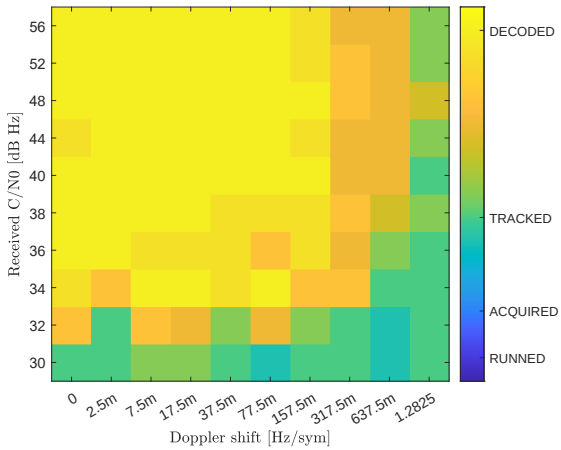
(B) $r_D = 100$, $ss = 2 - 5$, $n_C = 1$



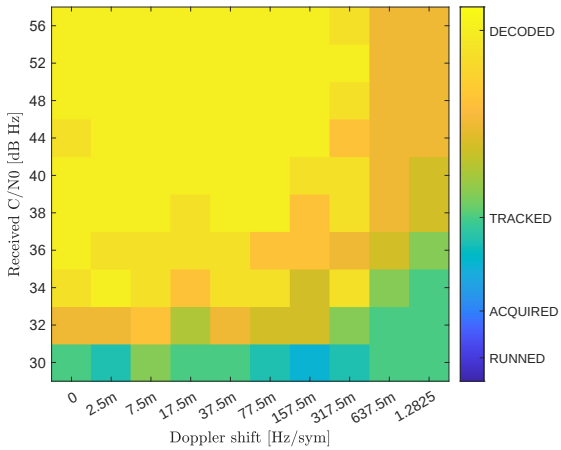
(C) $f_B = 1$, $ss = 2 - 5$, $n_C = 1$



(D) $r_D = 500$, $ss = 2 - 5$, $n_C = 1$

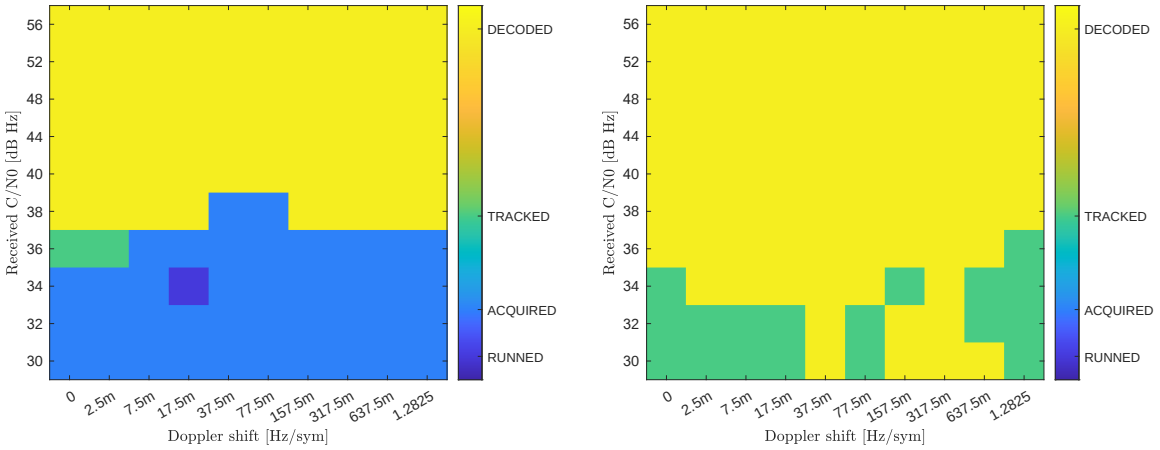


(E) $f_B = 3$, $ss = 2 - 5$, $n_C = 1$



(F) $r_D = 1000$, $ss = 2 - 5$, $n_C = 1$

FIGURE 12: Previous best results as function of the filter bandwidth f_B and the Doppler range r_D.



(A) $f_B = 1$, $r_D = 100$, $ss = 1$, $n_C = 3$

(B) $f_B = 1$, $r_D = 100$, $ss = 2$, $n_C = 1$

FIGURE 13: Comparison between best results without and with non-coherent correlation.

5 Conclusions

In this project, we were able to implement a working GNSS receiver and evaluate its performance over a wide range of modulated signals. In particular, we presented the workflow of the code development. We described our algorithm and summarized its behavior through an overview of the different core functionalities, their implementation, and tunable parameters. Finally, we reported how these parameters influence the performance of the simulator. Thus, we demonstrated the correct demodulation of signals with high probability when the received SNR $C/N_0 \geq 34\,\mathrm{dB\,Hz}$ and the Doppler drift $\Delta f \leq 1300\,\mathrm{Hz/s}$.

The main limits of our simulator lie in the tracking of the Residual Doppler Phase. Since we integrate over an insufficient time interval, the resulting values do not provide an accurate estimation, preventing the correct demodulation of the symbols. We could implement a different setup that averages the results over multiple segments by assuming a slower drift in the Doppler frequency. Another enhancement implies the use of the CRC for error correction in the payload (in addition to error detection) and all the other Channel Coding techniques.
Other improvements affect the acquisition and tracking algorithm. At first, we could reduce the computation time by performing the correlation search in two phases with different granularity. E.g., evaluating only one sample per chip period in the first phase will reduce the computational cost without missing the potential correlation peak. We could introduce non-coherent correlation in the acquisition phase, too.
Regarding the second, we could implement a better maximum discrimination algorithm instead of looking for the maximum value, improving the future prevision of the estimate. Moreover, the chosen shifts in code phase and Doppler frequency could be optimized through numerical simulation to maximize the noise resistance of the tracking algorithm.

Finally, we have shown that the most performance-demanding task of our simulator is signal acquisition. We could improve it by under-sampling the incoming signal to roughly estimate the Doppler frequency and Time Delay before searching for more refined values on a smaller correlation space. Moreover, we could improve the performances by using the Parallel Computing Toolbox to save the signal samples directly into `gpuArray` for GPU acceleration.

A Configuration of development environment

We developed our simulator in `MATLAB 2021a` and `MATLAB 2021b`, but later versions should be compatible. We employ `GitHub` as a distributed version control system to keep track of the software updates. Excluding `GIT`, we do not use external software or dependencies, but we cannot grant the correct execution of the scripts without the following `MATLAB` Add-Ons:

- Signal Processing Toolbox
- Data Acquisition Toolbox

The final code release is also available at
https://github.com/DavideTomasella/Satellite-lab/releases/download/v1.0/SAT_RECEIVER.rar

References

[1] European Union and European Space Agency, European GNSS (Galileo) OpenService: Signal In Space Interface Control Document (OS SIS ICD) v1.1 (2010).

[2] K. Borre, D. M. Akos, N. Bertelsen, P. Rinder, S. H. Jensen, A software-defined GPS and Galileo receiver: a single-frequency approach, Springer Science & Business Media, 2007.

[3] GPS Interface Control Working Group, Interface Control Document (SIS ICD): GPS Interface Specification IS-GPS-200, Revision M (2021).