Frequency reconstruction of long transient gravitational wave signals from young neutron stars

Edoardo Giancarli edo.giancarli@gmail.com
Department of Physics
University of Rome "Tor Vergata"
Via della Ricerca Scientifica, 1, Rome, Italy

Tutors: Sabrina D'Antonio, Viviana Fafone

Contents

Abstract

In this work we present an updated and modified matched filtering method based on the detection of signals power excesses in the interferometer data. This pipeline can be implemented for reconstructing the frequency trend of (i.) a long transient gravitational wave signal which can be emitted from a newborn remnant compact object formed in a binary neutron star merger or (ii.) of a long-duration gravitational wave signal associated with magnetar giant flares or (iii.) gravitational

emission from young neutron stars. For the purpose of this we have run different simulations concerning the analysis of an injected long transient signal from a young neutron star into a sample of real data from LIGO Livingston[1]. For these simulations we have chosen two different models for the injected signal frequency: a power law frequency and an exponential one. The simulation has been implemented by using a new set of codes in Python[2] written specifically for this work and based on SNAG[3], a Matlab object oriented toolbox for data simulation and analysis of signal and noises for gravitational antennas. From the whole simulation we have obtained a frequency reconstruction in agreement with the chosen models for both the frequency trends. This method can be further generalized and tuned to cover different kind of gravitational signals with different time duration, being robust toward different signals waveform. This can be crucial in order to be able to detect signals whose shape cannot be fully predicted, such as gravitational waves emitted from core collapse supernova events.

Keywords: Gravitational Waves, Long Transient, Continuous Waves, Matched Filtering, Neutron Stars

1. Introduction

Today, the most detected sources of Gravitational Wave (GW) signals are binary systems. From the first detection (4) in September 2015 up to $O3$ observing run many binary black hole systems (BBH), binary neutron star systems (BNS) and binary systems formed by a neutron star and a stellar black hole (BHNS) have been detected (3), (5), (6), (7) by LIGO (1) and Virgo (8). These observed signals are called *transient signals* and they can last O(s) for BNSs down to $O(10^{-1}\text{s})$ for BBHs.
However, binary systems are not the only sources of GWs: other sources that have not been observed yet are core collapse supernovae, isolated neutron stars and magnetars, cosmic strings and the resulting stochastic background of GWs (10). The diversity of the GW signals expected from these sources require different detection algorithms. When the GW signal waveform is predicted analytically, matched filtering techniques are optimal data analysis methods for studying the signal (as explained later in Section 3.2) since their goal is to maximize the signal-to-noise ratio. In practice, this concerns mainly compact objects binary coalescence, cosmic strings signals (15) and GWs from pulsars (35), (16).
In these cases we can distinguish two types of GW signals: (i.) *continous waves* (CW) which are signals with slowly varying frequencies (21), (31) and (ii.) *long transient* (LT) signals, characterized by long lasting duration of O(hours − days) and frequency variations. We expect that recently born neutron stars (i.e. after a BNS merger or supernova event) will emit LT-GWs (11). In this work we have focused on GW signals emitted by isolated neutron stars. We expect that rapidly spinning neutron stars could emit GWs due to internal and structural induced deformations in which an oscillation mode is excited (12), (20), (28).

As the LIGO/Virgo detectors will become more sensitive in the next years, we should be able to observe more binary mergers and also CWs from neutron stars. In order to search and to analyze CWs and LT-GWs into the interferometer data we can take into account both modelled (24), (27) and unmodelled (36), (26) methods. However these all-sky and directed research are computationally

1. https://www.ligo.caltech.edu/LA
2. https://www.python.org/
3. http://grwavsf.roma1.infn.it/snag/

expensive, since one must search the entire data set in every position in the sky. This can lead to consume even weeks of computing power.
In the last years Machine Learning (ML) and Deep Learning (DL) based techniques have been explored to study GWs, from binary systems searches to noise reduction and also CWs and LT signals searches. It has been shown that ML and DL techniques are able to reduce the computation costs of the CWs and LTs research without diminishing the sensitivity (25), making ML and DL very useful tools. Future studies aimed to the selection of the data that represent the frequency reconstruction made in this work and to the estimation of the injected signals parameters can be done by using ML and DL techniques, as discussed in Section 4.2.

In this work we present an updated and basic matched filtering technique based method for reconstructing the frequency trend of a LT signal. To analyze this LT we have simulated and injected a LT-like signal into a sample of real data from LIGO Livingston, which represents the interferometer's noise. In our analysis we have studied the frequency of the injected signals by considering two different frequency bands in which the interferometer output data was stored: $[107 - 108]$ Hz and $[295 - 300]$ Hz. The total output data contained in these slots represent a sub-sample of data collection during the $O3$ observing run.
In Section 2 we describe the models used for the characterization of the injected signals and we discuss the role of the breaking index, a parameter which distinguishes the signal frequency trends. In Section 3 we describe the pipeline of our simulation (Section 3.1), the matched filtering technique (Section 3.2) and then the injection/reconstruction and fitting methods which have been utilized (Section 3.3 and 3.4). We then show the obtained results in Section 4 and in the Appendix A and finally draw some conclusions in Section 5. The codes which have been used during the simulations are shown in both Appendix B, where we talk about how the signals where generated, and Appendix C, where we show other useful codes used in our analysis.

2. Injected Signal Models

In the simulations we have chosen two different models for the injected LT signals.
The first model is characterized by a power law dumped sinusoid amplitude $h(t)$ and a power law frequency $f(t)$, where (32), (17), (2)

$$h_0(t) = \frac{4\pi^2 G I_{zz}}{c^4}\frac{\epsilon}{d} f^2(t) \tag{1}$$

Here, I_{zz} is the neutron star principle moment of inertia, ϵ is its ellipticity, d is its distance (luminosity distance if cosmological correction has to be considered), G is the gravitational constant and c is the speed of light.
The total GW strain is the combination of the h_+ and $h_\times$ polarizations, such as $h(t) = F_+ h_+(t) + F_\times h_\times(t)$:

$$h(t) = h_0(t)\left[F_+ \frac{1+\cos^2(\iota)}{2}\cos\Phi(t) + F_\times \cos(\iota)\sin\Phi(t)\right] \tag{2}$$

$$\Phi(t) = \Phi_0 + 2\pi\int_0^t dt'\, f(t') \tag{3}$$

In this equation $F_+ = F_+(\theta,\phi,t)$ and $F_\times = F_\times(\theta,\phi,t)$ are the pattern functions, which characterize the angular sensitivity of the interferometer, ι is the inclination angle of the star's rotation axis with

respect to the line of sight and $\Phi(t)$ is the phase of the emitted gravitational radiation.
The signal frequency has the form

$$f(t) = f_0 \left(1 + \frac{t - t_{coes}}{\tau}\right)^{\frac{1}{1-n}} \tag{4}$$

where t_{coes} is the initial time of the signal, τ is the frequency characteristic time (which in this case represents the spindown time-scale) and n is the breaking index.
The breaking index is a measurable quantity that characterizes the neutron star rotational frequency (17) and is of the form

$$n = \frac{f|\ddot{f}|}{\dot{f}^2} \tag{5}$$

where f is the rotation frequency of the neutron star, $\dot{f}$ is the spindown, and $\ddot{f}$ is the rate of change of the spindown. In most pulsars the observed breaking indexes have values within $[1.4 - 3]$, indicative of an older age and of electromagnetic-dominated radiation (9), (14), (22). However, in our work we have chosen a breaking index $n = 5$, which characterizes a young neutron star for which the energy emission mechanism is dominated by GWs release due to a large quadrupolar deformation (34).

The second model is characterized by an exponential dumped sinusoid amplitude (with the same law of equation (2) and with the same discussion) and an exponential frequency:

$$f(t) = f_0 \exp\left(-\frac{t - t_{coes}}{\tau}\right) \tag{6}$$

In the following we will focus only on the frequency evolution with time, neglecting the amplitude evolution.

3. Methods

3.1 Simulation and Analysis pipeline

In this work we studied the ability of a newly implemented method to reconstruct gravitational signals by improving their Signal-to-Noise Ratio (SNR) through the application of a modified matched filtering algorithm (discussed in Section 3.2) on a data sample which includes the signals data and the interferometer noise. During the study we limited ourselves to test the performance and the efficiency of this new method in recovering the signals waveform, while the characterization of the procedure in terms of detection sensitivity will be the object of future works.
To perform our simulation, we have injected software simulated signals (discussed in Section 3.3) in the LIGO Livingston $O3$ data (representing the interferometer noise), stored in structures called Band Sampled Data (BSD) (30).
The total data is then processed by applying a bank of N frequency filters having a Gaussian transfer function such that each filter is able to select the excess of signal power into a specific frequency region within the whole frequency bands ($[107 - 108]$ Hz or $[295 - 300]$ Hz). The most significant candidates among the N output filter templates are then selected by considering their Critical Ratio. The time evolution of the selected candidates in each small frequency region and their frequency values, represented by the mean values of the Gaussian templates, allows us to reconstruct the frequency trends of the injected signals.

To test the performance of our simulation, we also varied the injected signals amplitude values relative to the noise amplitudes. This allowed us to simulate different scenarios that are typically encountered when searching for hidden GW signals within the interferometer data.
Finally, we fitted the obtained output candidates to verify the accuracy of our signal reconstruction, as discussed in Section 3.4.
Overall, our simulation demonstrated the effectiveness of our pipeline in recovering signals frequency evolution and provided valuable insights into the performance of signal detection algorithms in the presence of noise.

3.2 Matched Filtering

Matched Filtering (37) is a data analysis technique which use a matched filter, represented by a signal template. This technique is used when searching for a known signal embedded in noise (usually modeled as white Gaussian noise). The filter gives the maximum SNR among all linear filters, so an optimal detection can be achieved.
Matched filters play a central role in GW detection (33), since they can be used to exalt and to parameterize GW signals. This technique is also commonly used in other contexts such as (a.) radar or sonar echoes in which a known signal is sent out, and the reflected signal is examined for common elements of the out-going signal; (b.) radio signals analysis; (c.) image data (e.g., to improve SNR for X-ray); (d.) seismic signals; ecc.
This filter is applied to the data by computing the *convolution* between the data and the function which gives the wave form of the considered signal.
So, if we measure an output $s(t)$ which is the sum of the noise $n(t)$ and of the searched signal $h(t)$, we can consider a matched filter equals to $h(t)$ to improve the SNR of the measurement, since typically $|h(t)| \lesssim |n(t)|$. The convolution between the output data and the filter will be:

$$(s * h)(\tau) = \int_{-\infty}^{\infty} dt\, s(t) h(t - \tau) \tag{7}$$

and the signal hidden in the detector data stream will appear at the time where the convolution between the detector output and the filter function is maximized.

Since we are dealing with the convolution between two signals, we can simplify the computation by using the Convolution Theorem and switch to the frequency domain (or Fourier domain/Fourier space):

$$\mathcal{F}(s * h) = \mathcal{F}(s)\mathcal{F}(h) \tag{8}$$

where $\mathcal{F}$ represents the Fourier transform operator. The transition to the frequency domain is common when dealing with the convolution operation since the computational cost highly decreases (13). If we take N samples of $s(t)$ and of the filter $h(t)$, the discrete convolution will be:

$$(s * h)_n = \sum_m s_m h^*_{m-n} \tag{9}$$

where h^* represents the data sample h with its elements in a reversed order. Each output value of the convolution requires N multiplications and $N-1$ additions, so the computational cost is of the order of $O(N^2)$ operations. In the Fourier space, because of the conversion to multiplication, we will have only $O(N \log N)$ required operations, which can be significant for large values of N.

In this work, however, we implemented and tested a modified matched filtering method since from a physical point of view we do not know yet the correct waveform of these LT signals.
In this pipeline, the generated bank of Gaussian filters is meant to overcome the lack of the filter $h(t)$ that represent the searched known signal. Each of the Gaussian templates is optimized to extract those excess in power of the LT signals that are injected in the interferometer data samples at a certain frequency in the whole considered frequency slots.
Through this implemented method we are able to extract the LT signals embedded in noise without knowing any details of their waveform. This method also offers a further implementation since the bank of Gaussian filters can be arranged in such a way as to best cover different time evolution of the searched signals.
The study of the optimal filter grid to apply on the signals will be object of future studies, where this modified matched filtering method might also be integrated with a machine learning based analysis of the output candidates to better select them for the signals reconstruction.

3.3 Injection and reconstruction of the signals frequency

The injected signals were generated using a set of Matlab codes. These codes had the purpose to create the signals and to adapt them to the same format of the interferometer data output, represented by complex data. The details on the implementation of the Matlab-Python interface and of the developed Python codes for the pipeline are given in Appendix B and Appendix C.

Once we had our data, we injected the signals in the interferometer output by considering a factor α as a factor for the signals amplitude. Through this factor we could decide the intensity of the signal with respect to the noise (see Listing 7 in Appendix C):

$$s(t) = \alpha h(t) + n(t) \tag{10}$$

We decided to perform our simulation taking into account $\alpha = 10^2$, 10^{-1} and 10^{-4} to test the efficiency of our codes. In particular, the case for $\alpha = 10^{-4}$ is the most interesting since it should simulate the case of real observations. In order to consider real data, the total signal $s(t)$ has to be multiplied by a factor $10e-20$. Usually this step is done after the analysis to display the signals in order to reduce the computational cost and memory of the procedure.
The injection of the signals in the interferometer output data was made by only taking a sub-sample of the entire output data (in order to further reduce the computational cost of the simulation). Additionally, in order to resemble the randomness in the GW observations, we added a boundary to the injected data (called "*y_edge*", as shown in Listing 7 in Appendix C).

In order to perform the matched filtering we have built a database containing a series of Gaussian templates in the frequency domain (see Figure 2 and Listing 6). These templates are used in the convolution operation (in the Fourier domain) to match the signals frequencies, as shown in Figure 1. During the construction of the templates set we also considered a variable standard deviation σ: since the frequency signals we are looking for do not have linear variable time trends, we have to take into account the rate of frequency change in time and relate it to the templates. The main reason of this is that in real situations we have to deal with time-dependent signals embedded in noise. If we neglect the insertion of a variable σ for the Gaussian templates the risk is that of reconstructing wrongly the frequency signals during the convolution operation. For example, if we consider a certain time interval in which the frequency is rapidly variable and we use templates with

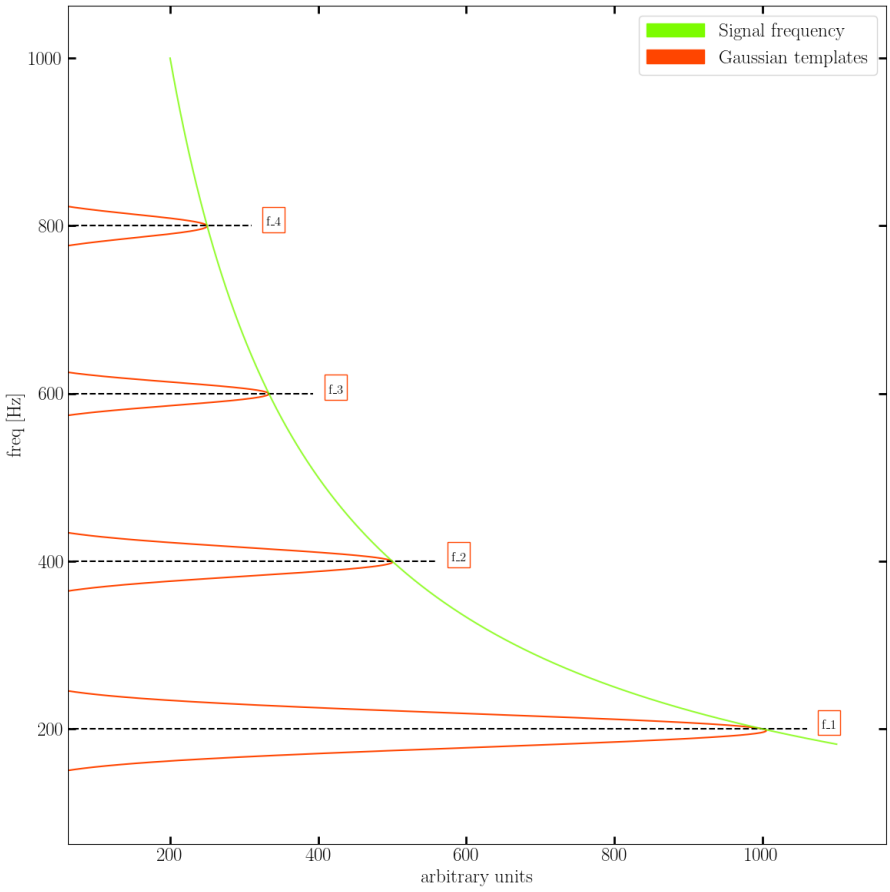


Figure 1: Reconstrution of the (ideal) signal frequency by the convolution between frequency and Gaussian templates.

broad σ we would measure mainly the noise in which the signal is embedded (note that in Figure 1 we neglect the noise). So, in order to precisely reconstruct the signals, a variable σ is crucial as we have to tune the width of each Gaussian template with the speed of variation of the signal frequency. We took into account this aspect in the simulation by considering the ideal signals (as shown in the Listing 6) and setting the template σ equal to the rate of frequency change in a TFFT length (which marks the Fourier transform time intervals of the frequencies and defined as TFFT = lfft$\times dt$, where lfft is the length of the frequency signals in the Fourier space and dt is the sampling time of the interferometer output data) divided by a factor "div" (as shown in the Listing 6). We choose a value of div = lfft/128 because we obtained that under this value (depending on the chosen lfft) the σ were too little to have defined templates, preventing the convolution from working.

The final steps of the frequencies reconstruction have been performed with the algorithms shown in Listings 10 and 11 respectively. In the first algorithm the matched filtering is performed by convolving the injected signals with the Gaussian templates. To easily do so we divided the input data (interferometer data sub-sample plus injected data) in chunks of length lfft (since the convolution is performed in the frequency domain) and we analyzed each one of them also taking into account

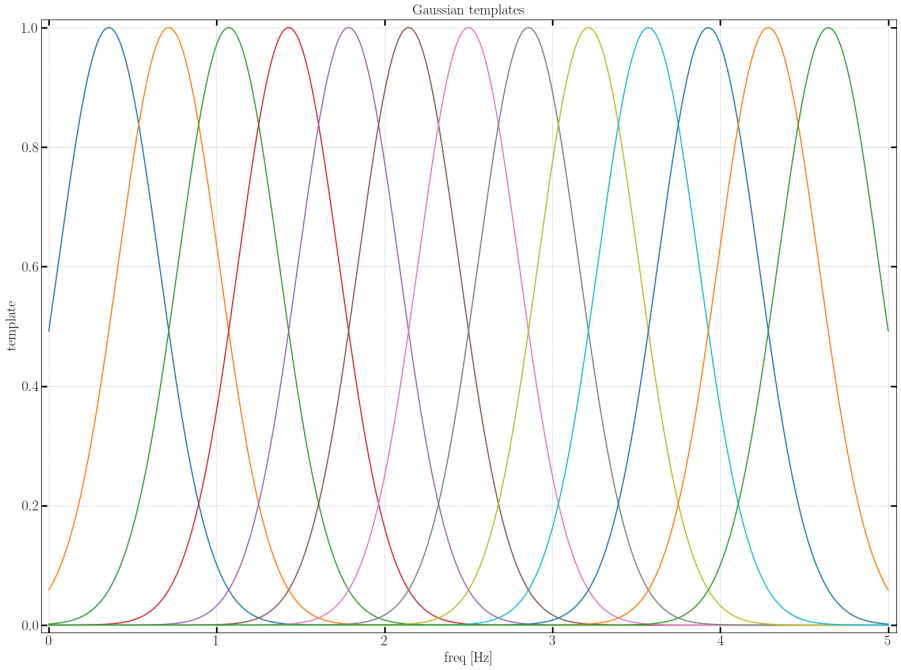


Figure 2: Gaussian templates.

the initial times at which each chunk occurred. All the chunks are interlaced as we advanced in the analysis by a step of lfft/2. After the convolution was performed (for each chunk), we went back in the time domain and we deleted the edge of the output arrays in order to avoid border effect from the Fast Fourier Transform operation (see Listing 10).
The output candidates selection (see Listings 9 and 8), where the candidates are represented by those templates that better match with the convolved data, was made by considering a detection statistic called Critical Ratio (CR), defined as (24):

$$CR_i = \frac{x_i - \overline{x}_i}{\sigma_i} \tag{11}$$

where x_i represents the data, $\overline{x}_i$ the median of the data and σ_i the standard deviation of the $i-th$ template. Each chunk was divided in a certain number of sub-chunks (*ncand*) for which those templates that had a critical ratio (CR) over a chosen threshold were extracted (one for each sub-chunk). The choice of the threshold influences (i) the search sensitivity, because the critical ratio is a measure of the false alarm probability and (ii) its computational weight since it reduces the number of total candidates that have to be analyzed. So the CR threshold is a compromise between the computational cost and the sensitivity of the search.
For the selection we considered the CR absolute value for each candidate |CR|. The $filter\_data\_chunk$ function (Listing 10) reports the analysis of all the chunks with the output candidates that satisfy the selection process.
In the second algorithm (Listing 11) the output candidates were extracted from the dataframe containing the whole chunks output. After the candidates extraction we performed a second selection to locate those candidates that at fixed frequency and fixed time have the higher CR values.

Finally, we performed a time correction on the selected candidates. First, we noted that in the transition from Matlab to Python the whole signals were translated in time for a period of one day, producing an offset in time between the frequency signals and the candidates. Then we corrected the initial times of each chunk taking into account the offset between the start of the sub-data sample in which the signals were injected and the beginning of the signals.

3.4 Fitting of the obtained data

After selecting the candidates, we proceeded with data fitting to verify the efficiency of our reconstruction as shown in the Listing 18 (see also Listings 14, 16, 15 and 17 for more details). Initially, we analyzed the obtained candidates to remove outliers that negatively affected the fit. We limited ourselves to removing candidates too far from the main clusters of data points identified as the proper reconstruction. We discuss potential improvements in Section 4.2.
After removing the outliers, we proceeded with fitting the output candidates. We attempted to fit the candidates using the *Scipy* and *Scikit-learn* libraries. The *scipy.optimize.curve_fit* method was unable to reconstruct the signal, therefore, we switched to a linear fit (linear regression) using a logarithmic base. Taking into account the models shown in Section 2 the expected values are:

$$ln(f(t)) = \ln(f_0) - \frac{1}{4}\ln\left(1 + \frac{t - t_{coes}}{\tau}\right) \tag{12}$$

$$ln(f(t)) = \ln(f_0) - \frac{t - t_{coes}}{\tau} \tag{13}$$

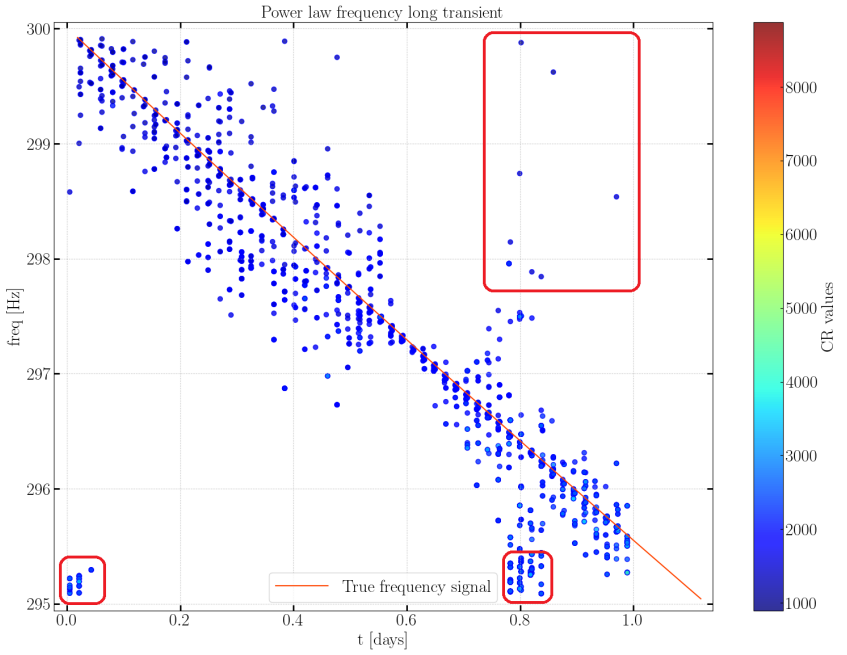
for the power law frequency model and exponential frequency model respectively.
We then examined the residuals of the fit, which are the differences between the predicted values of the model and the actual data points. Analyzing the residuals can help identify any remaining issues with the model, such as overfitting or underfitting, and improve the overall accuracy of the fit. We also analyzed the residuals between the fitted models and the true models, as well as the residuals between the selected output candidates and the true models.

4. Results

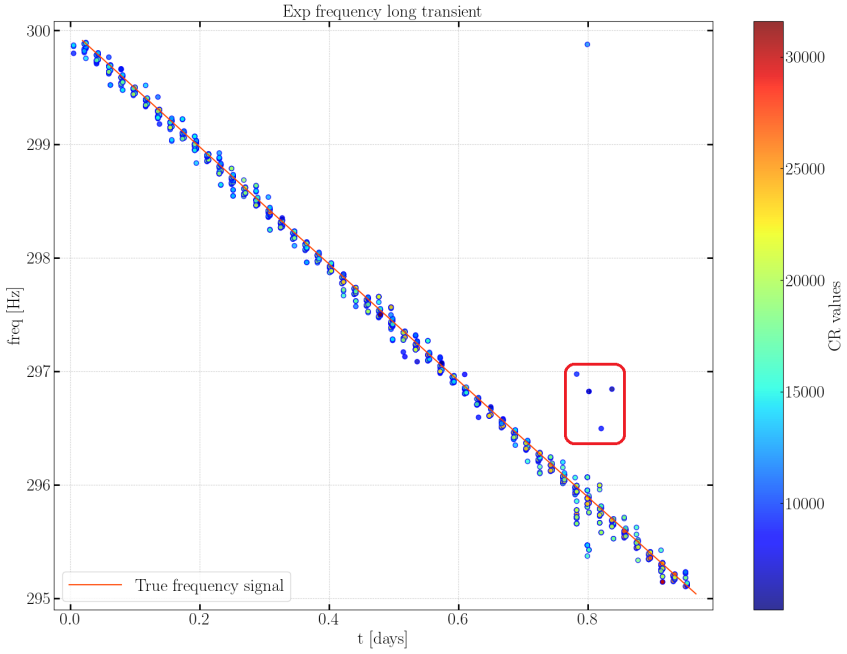
4.1 Analysis results

As previously discussed, we simulated the injection of signals into two frequency bands containing output data samples from LIGO-L: $[107 - 108]$ Hz and $[295 - 300]$ Hz.
In the simulation, we used a factor $\alpha = 10^2$, 10^{-1}, and 10^{-4} (as shown in equation (10) and as described in Section 3.3) to establish the signals amplitudes intensity relative to the noise.
In this Section, we present some of the obtained results from the analysis of the injected signals. Specifically, we focus on the frequency band $[295 - 300]$ Hz with $\alpha = 10^2$, and the frequency band $[107 - 108]$ Hz with $\alpha = 10^2$ and $\alpha = 10^{-4}$. The remaining results can be found in Appendix A.

Case study: $\Delta f = [295 - 300]$ Hz **and** $\alpha = 10^2$ The output candidates obtained from our analysis are shown in Figure 3, where we have also highlighted the outliers that we have removed following the discussion in Section 3.4 (the same for the other case studies in the Appendix). As shown, the outliers do not belong to the main cluster of data points which represents the frequency reconstruction. The output candidates have been obtained by considering a CR threshold of $\mathrm{CR}_{thr} = 8$ and a lfft = 8192. The final output candidates used in the fitting procedure are shown in Figure 4.

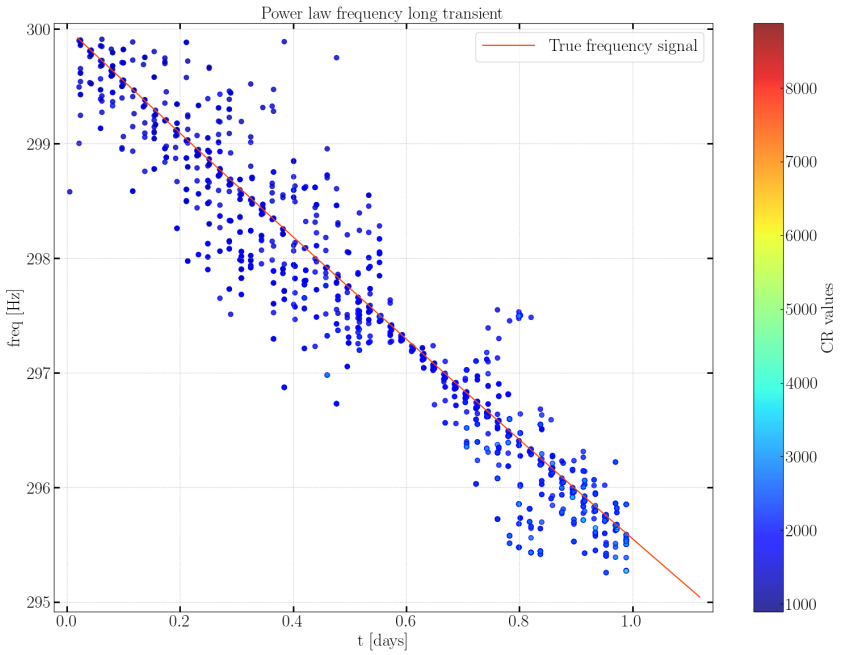


(a) Power law frequency

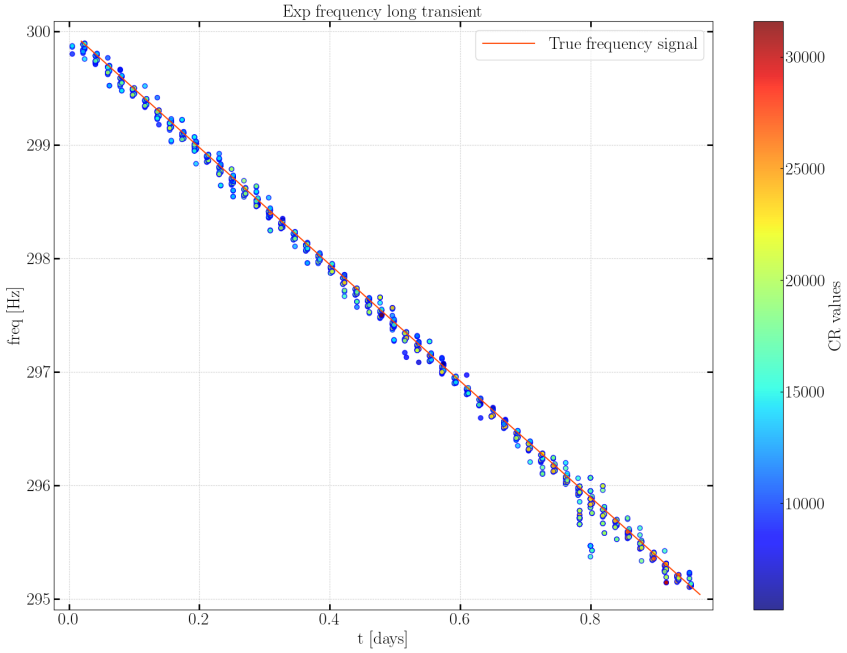


(b) Exponential frequency

Figure 3: Output candidates for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.



(a) Power law frequency



(b) Exponential frequency

Figure 4: Final output candidates after outliers removal for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.

From the linear regression in the logarithmic base, we obtained an r-squared value of $r^2 = 0.90$ for the power law frequency signal and $r^2 = 0.9976$ for the exponential frequency signal. The obtained regression coefficients are shown in Table 1, displaying a good agreement with respect to the expected values, while the comparison between the candidates, the fitted model and the true model are

Table 1: Comparison between linear regression coefficients and expected values (in the log base) for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.

Frequency	**Fit Coeff.**	**True Coeff.**	**Fit Intercept**	**True Intercept**
power law	-0.24 ± 0.01	-0.25	5.70325 ± 1e-5	5.70382
exponential	-1.0051 ± 5e-4	-1	5.703744 ± 4e-6	5.703816

shown in Figure 5.
During the analysis, we also studied the residuals between the final output candidates, the fitted models and the true models, as shown in Figure 6 and in Table 2. The residuals between the output

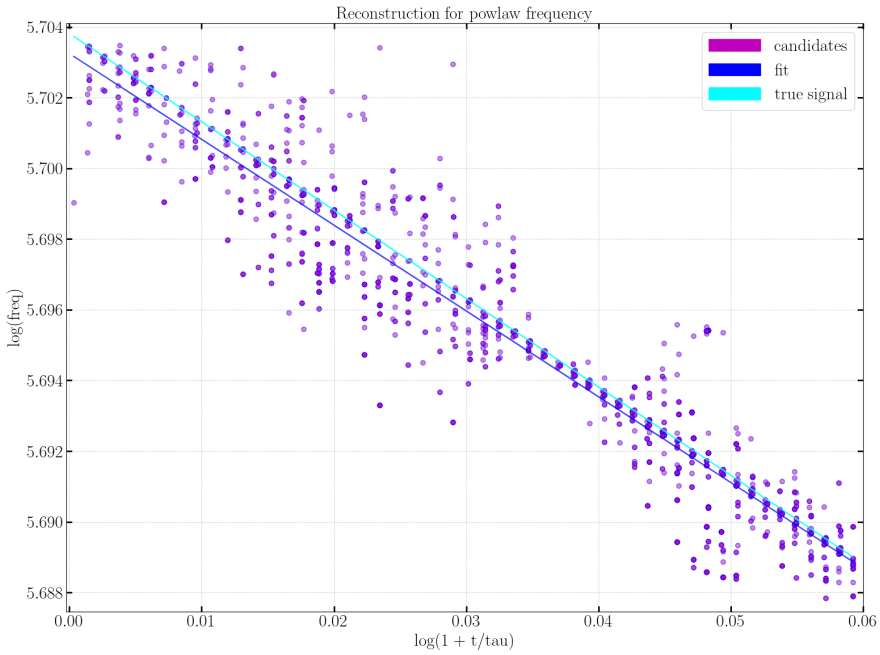
Table 2: Residuals between the final output candidates, the fitted models and the true models (in the log base) for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.

Frequency	**Fit vs Cand. median ± std**	**Fit vs True median ± std**	**Cand. vs True median ± std**
power law	-1e-4 ± 0.001	-3e-4 ± 1e-4	-2e-4 ± 0.001
exponential	-2e-5 ± 2e-4	-1.1e-4 ± 2e-5	-1e-4 ± 2e-4

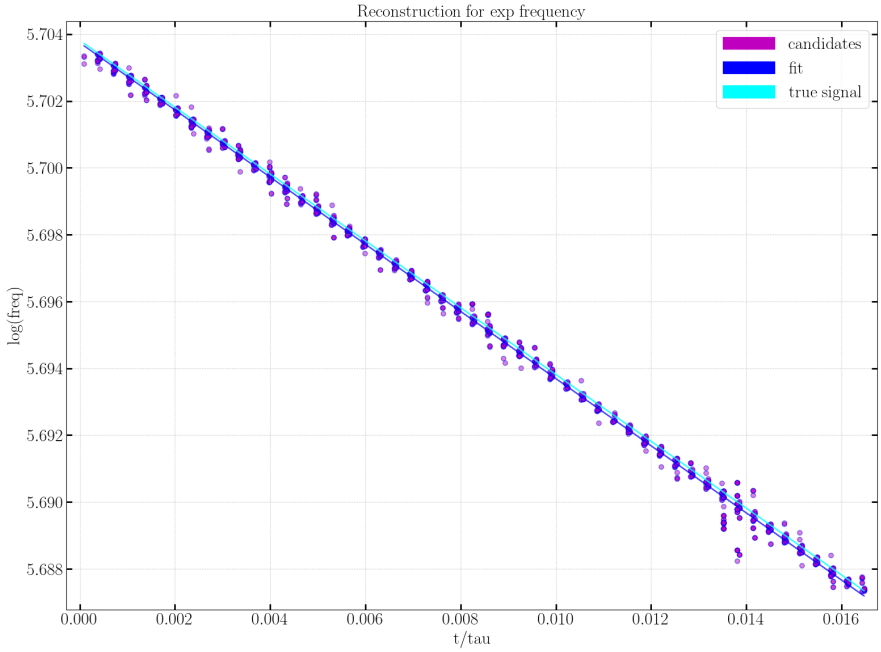
candidates and the fitted model demonstrate the accuracy of the fit procedure and that it has not been affected by underfitting or overfitting issues. They also show that no other underlying physical processes are present since they exhibit a uniform distribution, as well as the residuals between the output candidates and the true model. On the other hand, the residuals between the fitted and the true models show how the accuracy in the reconstruction changes with respect to the time coordinate.

Case study: $\Delta f = [107 - 108]$ Hz and $\alpha = 10^2$ Another important peculiarity of our analysis concerns the choice of the lfft value in the various case studies. As an example we show in Figure 7 the comparison between the reconstruction of the power law frequency signal in the case of lfft = 1024 and lfft = 512 (the results of the reconstruction for lfft = 1024 are shown in Appendix A). When dealing with signal reconstruction the lfft value characterize the resolution of the process since it determines the length of the chunks analyzed (see Listing 10) and it defines the Gaussian templates structure (see Listing 6). As shown, the two reconstruction appear different since in the lfft = 512 the output candidates display lower CR values but a more compact clustering around the true signal. Also in this case we choose a CR threshold of $\mathrm{CR}_{thr} = 8$.

Case study: $\alpha = 10^{-4}$ Finally, in Figures 8, 9 and 10 the results of the reconstruction for both frequency bands $[107 - 108]$ Hz and $[295 - 300]$ Hz for different lfft values are shown. These figures display that for a very low intensity of the signals with respect to the interferometer noise the simple matched filtering algorithm is not able to resolve the frequency signals, and no characteristic data points structures have been found since the candidates appear to be uniformly distributed. In

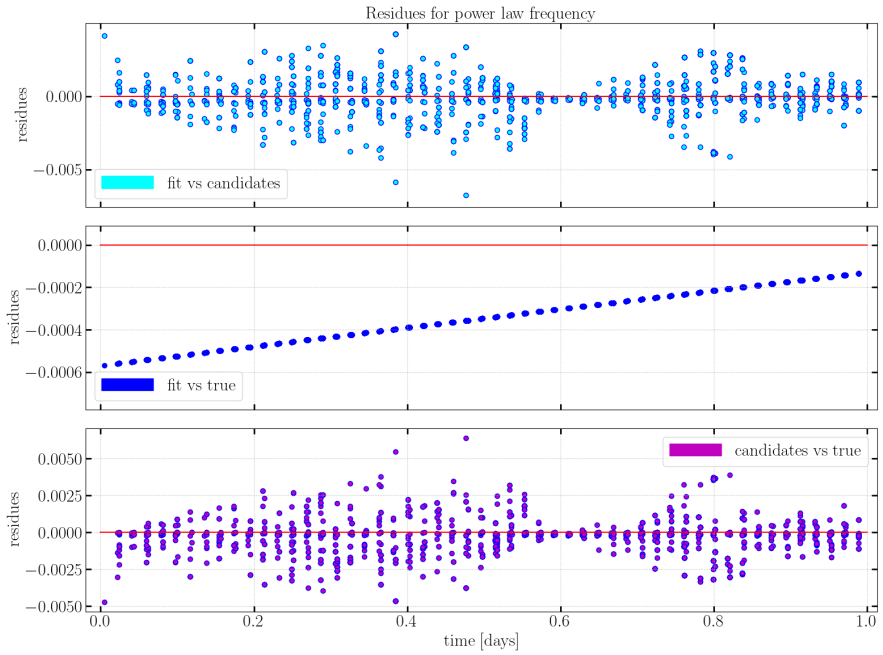


(a) Power law frequency

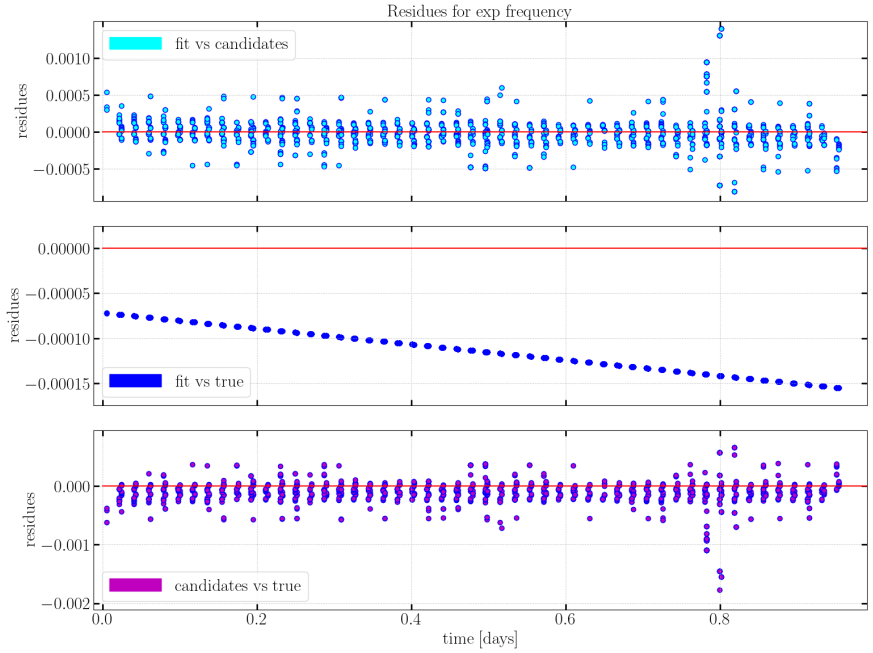


(b) Exponential frequency

Figure 5: Comparison between the output candidates, the fitted model and the true model (in the log base) for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.

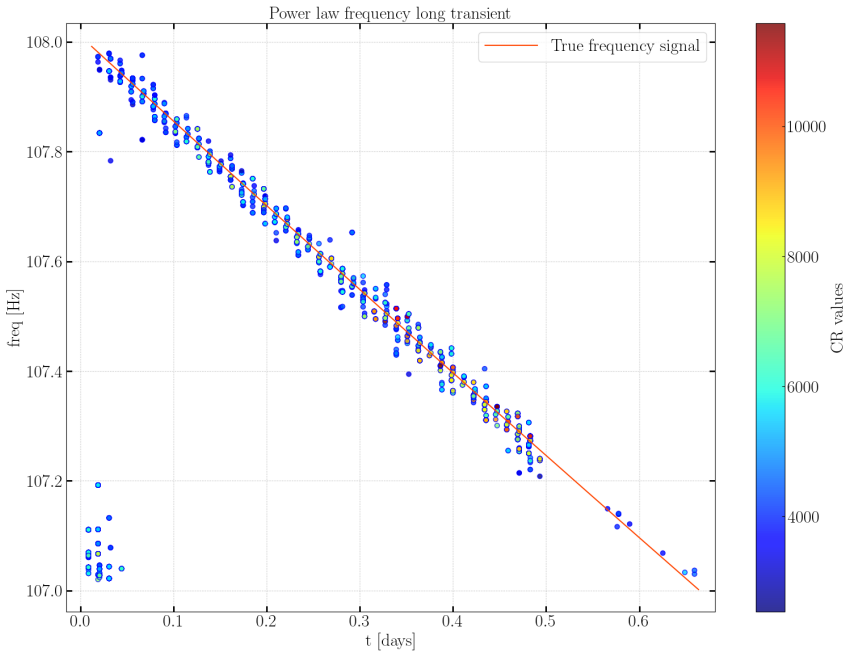


(a) Power law frequency

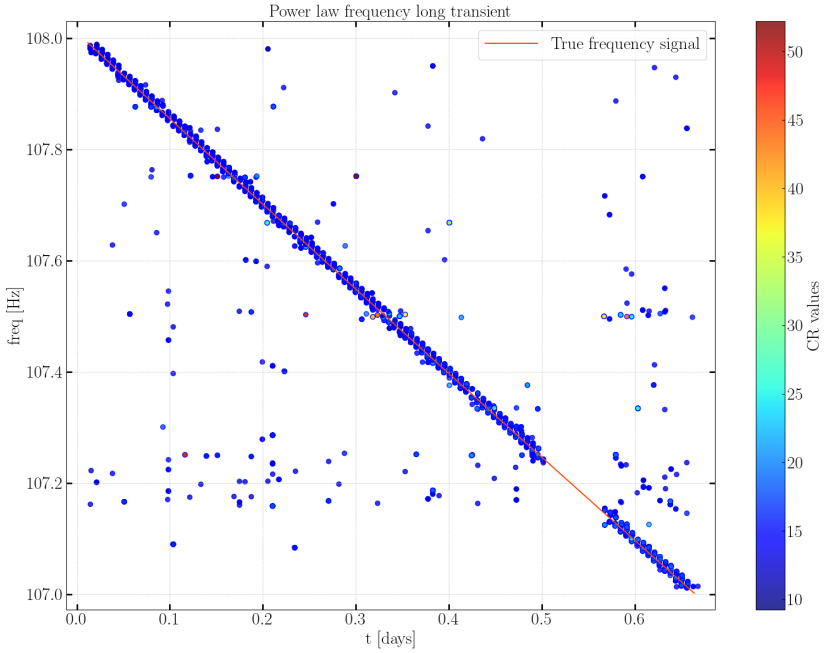


(b) Exponential frequency

Figure 6: Residuals between the final output candidates, the fitted models and the true models for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^2$.

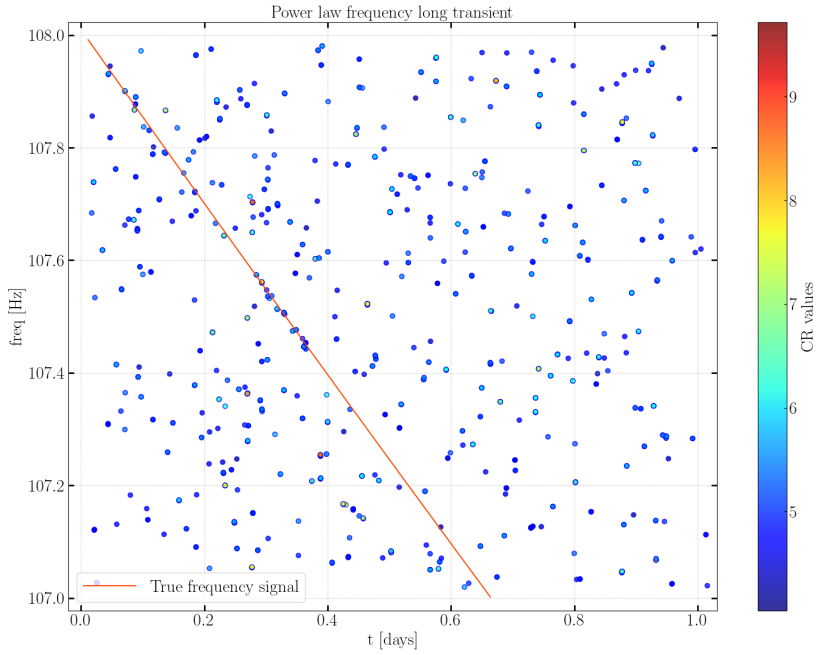


(a) Power law frequency, lfft = 1024

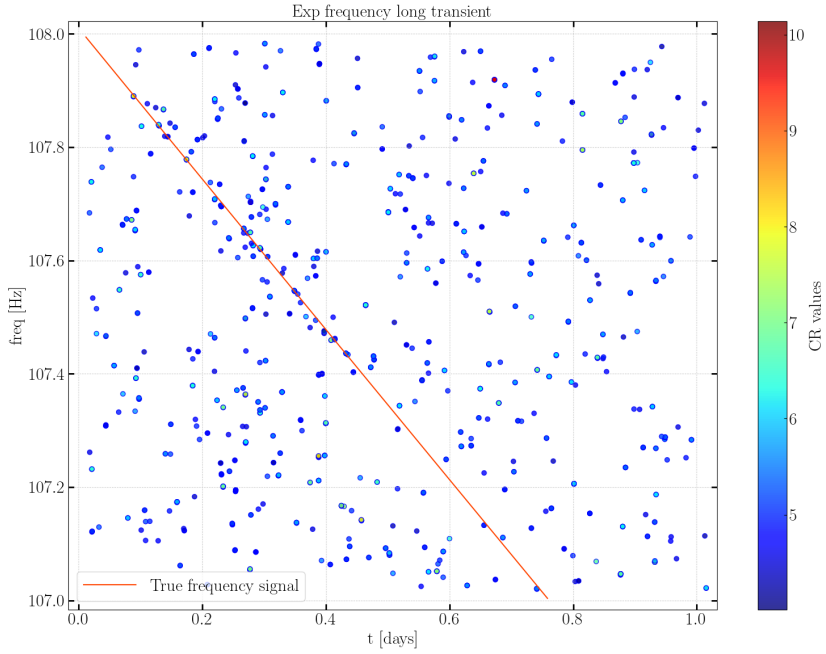


(b) Power law frequency, lfft = 512

Figure 7: Comparison between the reconstruction of the power law frequency signal in the frequency band $[107 - 108]$ Hz with lfft = 1024 and lfft = 512 for $\alpha = 10^2$.

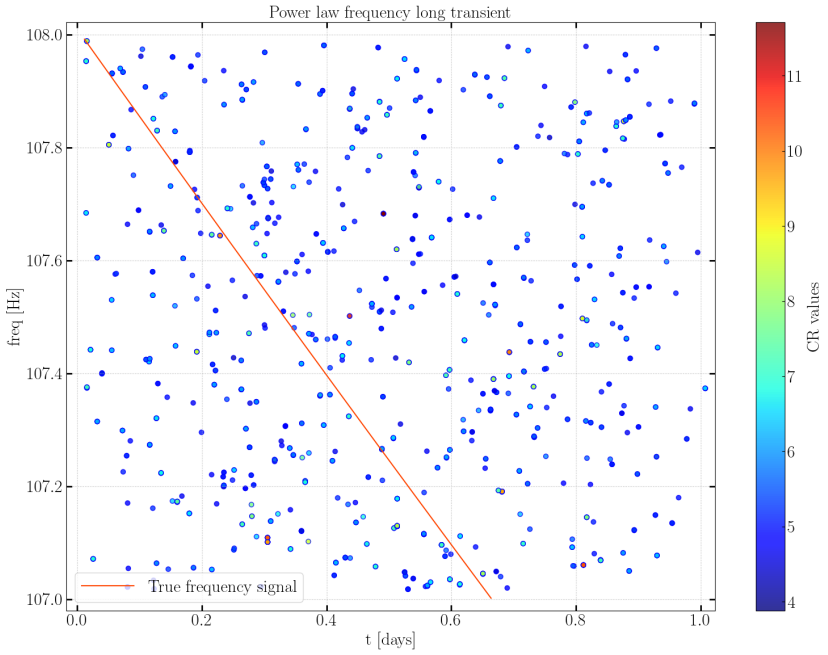


(a) Power law frequency

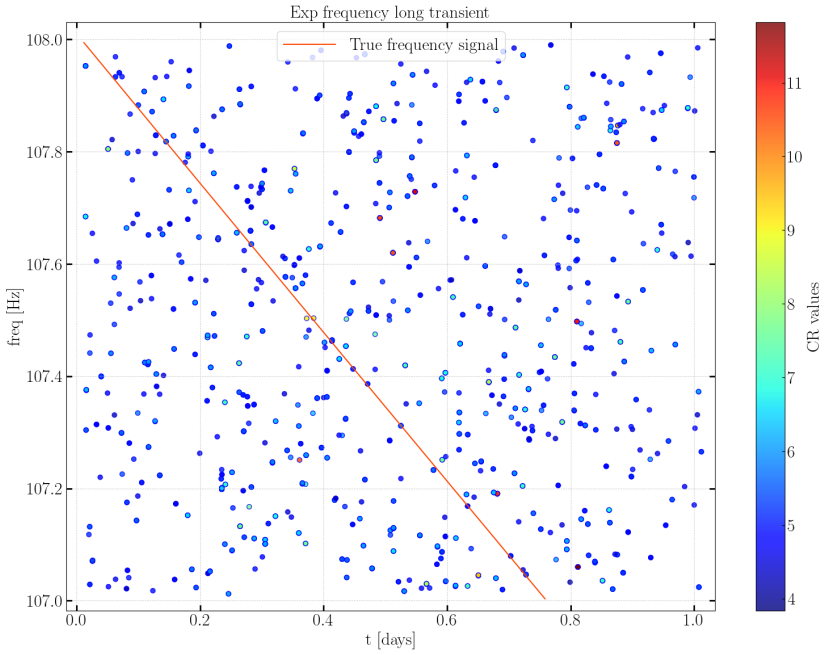


(b) Exponential frequency

Figure 8: Output candidates for the case study $\Delta f = [107 - 108]$ Hz, $\alpha = 10^{-4}$ and lfft = 1024.

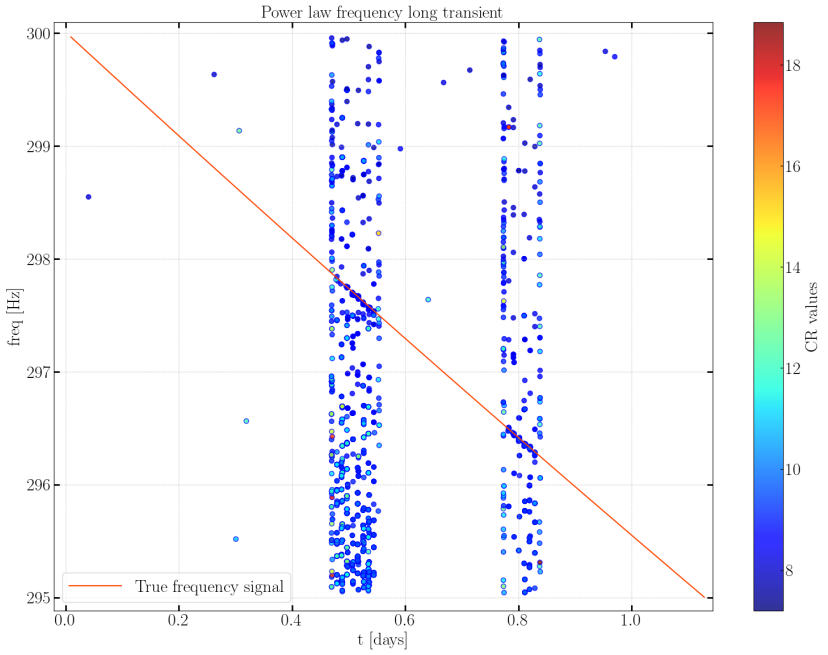


(a) Power law frequency

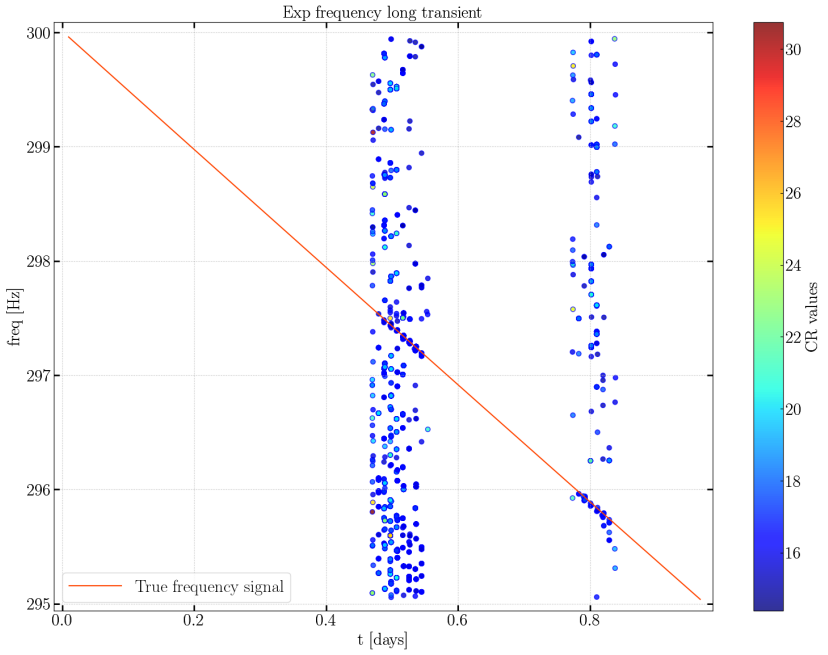


(b) Exponential frequency

Figure 9: Output candidates for the case study $\Delta f = [107 - 108]$ Hz, $\alpha = 10^{-4}$ and lfft = 512.



(a) Power law frequency



(b) Exponential frequency

Figure 10: Output candidates for the case study $\Delta f = [295 - 300]$ Hz, $\alpha = 10^2$ and lfft = 4096.

this case we set a CR threshold of $CR_{thr} = 0$ in order to consider all the output candidates from the matched filtering process. We tried to increase the threshold value after the reconstruction to see if any possible data point structures would emerge by deleting those candidates with lower CR values, but again, no data structures were found, setting a lower limit for the functioning of our simulation.

4.2 Further improvements

In the last part of this report we would like to discuss about some improvements that can be applied in possible future works. The main improvements, apart from a further optimization in the computational cost of the Python codes that have been built for this simulation, concern (i.) the selection of the output candidates which will be used in the fitting process; (ii.) a better identification and removal of the outliers; (iii.) a better organization and scanning of the dataframes containing the total output candidates of the matched filtering process (see Listings 10 and 11) and (iv.) the optimization of the filters transfer function for the matched filtering process (see Listing 6).

Output candidates examination When dealing with large dataframes which have to be loaded, read and analyzed the time costs can be long. To get around this problem one could think of filtering only some data contained in the dataframe or even customize it. Another possible solution is to divide the dataframe in smaller chunks and analyze them individually, diminishing the computational costs. This can be a possible solution to the output candidates extraction from the whole dataframe created after the matched filtering process, which might take a long time depending on the choice of the parameters that characterize the simulation such as the values "step" in Listing 6, "ncand" in Listing 10 and also lfft.

Final candidates selection The other important improvement concern the selection of the candidates for the frequency signals reconstruction. The total selection which gives the final output candidates for the fitting procedure can be thought as a two step process: (i) the first selection take place during the choice of those candidates that at fixed frequency and fixed time have the higher CR values (see Listing 11), while (ii) the second step is to remove the outliers in order to fit more accurately the data points.
However, a more accurate and robust way of selecting candidates could be the application of machine learning and deep learning techniques, as discussed in Section 1. One performing approach might be to use clustering algorithms, which group similar data points together based on some similarity metric and select less data points from each cluster that are representative of the cluster as a whole. An important step might be to choose a set of features that are relevant to the problem we are trying to solve. These features should capture the key characteristics of the data points that we want to cluster. Then, we can normalize the data so that all features have the same scale. This is important because clustering algorithms are sensitive to differences in scale between features.
Another example might be to use autoencoders, which are neural networks that can learn a compressed representation of the data. Autoencoders can be used to encode the data into a lower-dimensional space, and then select data points based on their proximity to a reference point in this lower-dimensional space. Again, we highlight the importance of data normalization since normalizing the data leads to features with the same scale; and also the neural network training on the normalized data to learn a compressed representation of the data.

Filter optimization The last improvement we discuss concerns the transfer function we have used for the filters in the simulation. In a matched filtering process, the transfer function of the filter plays

a crucial role in optimizing the SNR. The transfer function characterizes how the filter processes the input signal and separates the desired signal from noise and other unwanted components.
To improve the transfer function, one can use various techniques, such as adjusting the filter parameters (we already discussed the importance of a variable σ for the Gaussian templates), modifying the filter design, or implementing adaptive filtering algorithms. Our pipeline can be updated to consider different designs for the transfer function in order to optimize the SNR and to enhance the power excess detection when dealing with different signals waveform, leading to a better classification of the output candidates.
However, the choice of the most suitable design for improving the transfer functions efficiency depends on the specific application and the nature of the signals being processed.
Another important feature which characterizes the efficiency of the filters and which has to be accorded with their design concerns the choice of the external parameters for the simulation, e.g. the lfft value. As shown in Figure 7, different choices in the lfft value can lead to different outcomes, as discussed in the second paragraph of Section 4.1.

5. Conclusions

In this study, we simulated and characterized the observation of a long transient gravitational wave signal from a young isolated neutron star, with a focus on reconstructing its time-varying frequency. We generated the signals using the models discussed in Section 2 (see also Appendix B) and injected them into two different LIGO-L output data samples, which represent the interferometer's noise. The data samples covered the frequency bands $[107 - 108]$ Hz and $[295 - 300]$ Hz over a total observational period of one year and five days, respectively, during the $O3$ observing run.
Using the algorithms described in Appendix C, we reconstructed the frequency signals by employing a modified matched filtering technique. We then tested the accuracy of the reconstructions by fitting the obtained output candidates through linear regression.
The results of our study are presented in Section 4 and Appendix A, where we demonstrate the overall good agreement between the fitted values from the linear regression and the expected values from the models and we show the limits of the used algorithms.
Some general improvements about our simulation and analysis are discussed in Section 4, where we explore the optimization in the dataframe scanning, we talk about some possible machine learning and deep learning techniques which can be used to improve the candidates selection and we discuss a possible design optimization for the filters transfer function.
In summary, our study provides valuable insights into reconstructing time-varying gravitational wave frequency signals from isolated neutron stars which might be detected from the LIGO-Virgo (and in the near future KAGRA) network using a matched filtering based method in which signals power excess are detected in the interferometer output data and suggests avenues for further research to improve the accuracy and efficiency of the analysis, with the purpose of further exploring the (local) Universe, and beyond.

So long, and thanks for all the fish.

References

[1] J. Aasi, B. Abbott, R. Abbott, T. Abbott, M. Abernathy, K. Ackley, C. Adams, T. Adams, P. Addesso, R. Adhikari, et al. Advanced ligo. *Classical and quantum gravity*, 32(7):074001, 2015.

[2] J. Abadie, B. Abbott, R. Abbott, M. Abernathy, T. Accadia, F. Acernese, C. Adams, R. Adhikari, C. Affeldt, B. Allen, et al. Beating the spin-down limit on gravitational wave emission from the vela pulsar. *The Astrophysical Journal*, 737(2):93, 2011.

[3] B. Abbott, R. Abbott, T. Abbott, S. Abraham, F. Acernese, K. Ackley, C. Adams, R. Adhikari, V. Adya, C. Affeldt, et al. Gwtc-1: a gravitational-wave transient catalog of compact binary mergers observed by ligo and virgo during the first and second observing runs. *Physical Review X*, 9(3):031040, 2019.

[4] B. P. Abbott, R. Abbott, T. Abbott, M. Abernathy, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, R. Adhikari, et al. Observation of gravitational waves from a binary black hole merger. *Physical review letters*, 116(6):061102, 2016.

[5] R. Abbott, T. Abbott, S. Abraham, F. Acernese, K. Ackley, A. Adams, C. Adams, R. Adhikari, V. Adya, C. Affeldt, et al. Gwtc-2: compact binary coalescences observed by ligo and virgo during the first half of the third observing run. *Physical Review X*, 11(2):021053, 2021.

[6] R. Abbott, T. Abbott, F. Acernese, K. Ackley, C. Adams, N. Adhikari, R. Adhikari, V. Adya, C. Affeldt, D. Agarwal, et al. Gwtc-2.1: Deep extended catalog of compact binary coalescences observed by ligo and virgo during the first half of the third observing run. *arXiv preprint arXiv:2108.01045*, 2021.

[7] R. Abbott, T. Abbott, F. Acernese, K. Ackley, C. Adams, N. Adhikari, R. Adhikari, V. Adya, C. Affeldt, D. Agarwal, et al. Gwtc-3: compact binary coalescences observed by ligo and virgo during the second part of the third observing run. *arXiv preprint arXiv:2111.03606*, 2021.

[8] F. a. Acernese, M. Agathos, K. Agatsuma, D. Aisa, N. Allemandou, A. Allocca, J. Amarni, P. Astone, G. Balestri, G. Ballardin, et al. Advanced virgo: a second-generation interferometric gravitational wave detector. *Classical and Quantum Gravity*, 32(2):024001, 2014.

[9] R. Archibald, E. Gotthelf, R. Ferdman, V. Kaspi, S. Guillot, F. Harrison, E. Keane, M. Pivovaroff, D. Stern, S. Tendulkar, et al. A high braking index for a pulsar. *The Astrophysical Journal Letters*, 819(1):L16, 2016.

[10] M. Bailes, B. K. Berger, P. Brady, M. Branchesi, K. Danzmann, M. Evans, K. Holley-Bockelmann, B. Iyer, T. Kajita, S. Katsanevas, et al. Gravitational-wave physics and astronomy in the 2020s and 2030s. *Nature Reviews Physics*, 3(5):344–366, 2021.

[11] L. Baiotti and L. Rezzolla. Binary neutron star mergers: a review of einstein's richest laboratory. *Reports on Progress in Physics*, 80(9):096901, 2017.

[12] A. Bauswein and H.-T. Janka. Measuring neutron-star properties via gravitational waves from neutron-star mergers. *Physical review letters*, 108(1):011101, 2012.

[13] C. S. Burrus and T. Parks. Convolution algorithms. *Citeseer: New York, NY, USA*, 6, 1985.

[14] C. Clark, H. Pletsch, J. Wu, L. Guillemot, F. Camilo, T. Johnson, M. Kerr, B. Allen, C. Aulbert, C. Beer, et al. The braking index of a radio-quiet gamma-ray pulsar. *The Astrophysical journal letters*, 832(1):L15, 2016.

[15] T. Damour and A. Vilenkin. Gravitational wave bursts from cusps and kinks on cosmic strings. *Physical Review D*, 64(6):064008, 2001.

[16] K. Glampedakis and L. Gualtieri. Gravitational waves from single neutron stars: an advanced detector era survey. *The Physics and Astrophysics of Neutron Stars*, pages 673–736, 2018.

[17] O. Hamil, J. R. Stone, M. Urbanec, and G. Urbancova. Braking index of isolated pulsars. *Physical Review D*, 91(6):063007, 2015.

[18] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[19] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[20] P. Jaranowski, A. Krolak, and B. F. Schutz. Data analysis of gravitational-wave signals from spinning neutron stars: The signal and its detection. *Physical Review D*, 58(6):063001, 1998.

[21] D. Keitel, G. Woan, M. Pitkin, C. Schumacher, B. Pearlstone, K. Riles, A. G. Lyne, J. Palfreyman, B. Stappers, and P. Weltevrede. First search for long-duration transient gravitational waves after glitches in the vela and crab pulsars. *Physical Review D*, 100(6):064058, 2019.

[22] P. D. Lasky, C. Leris, A. Rowlinson, and K. Glampedakis. The braking index of millisecond magnetars. *The Astrophysical Journal Letters*, 843(1):L1, 2017.

[23] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[24] A. Miller, P. Astone, S. D'Antonio, S. Frasca, G. Intini, I. La Rosa, P. Leaci, S. Mastrogiovanni, F. Muciaccia, C. Palomba, et al. Method to search for long duration gravitational wave transients from isolated neutron stars using the generalized frequency-hough transform. *Physical Review D*, 98(10):102004, 2018.

[25] A. L. Miller, P. Astone, S. D'Antonio, S. Frasca, G. Intini, I. La Rosa, P. Leaci, S. Mastrogiovanni, F. Muciaccia, A. Mitidis, et al. How effective is machine learning to detect long transient gravitational waves from neutron stars in a real search? *Physical Review D*, 100(6):062005, 2019.

[26] L. M. Modafferi, J. Moragues, D. Keitel, L. Collaboration, V. Collaboration, and K. Collaboration. Search setup for long-duration transient gravitational waves from glitching pulsars during ligo-virgo third observing run. *arXiv preprint arXiv:2201.08785*, 2022.

[27] M. Oliver, D. Keitel, and A. M. Sintes. Adaptive transient hough method for long-duration gravitational wave transients. *Physical Review D*, 99(10):104067, 2019.

[28] B. J. Owen, L. Lindblom, C. Cutler, B. F. Schutz, A. Vecchio, and N. Andersson. Gravitational waves from hot young rapidly rotating neutron stars. *Physical Review D*, 58(8):084020, 1998.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[30] O. Piccinni, P. Astone, S. D'Antonio, S. Frasca, G. Intini, P. Leaci, S. Mastrogiovanni, A. Miller, C. Palomba, and A. Singhal. A new data analysis framework for the search of continuous gravitational wave signals. *Classical and Quantum Gravity*, 36(1):015008, 2018.

[31] R. Prix, S. Giampanis, and C. Messenger. Search method for long-duration gravitational-wave transients from neutron stars. *Physical Review D*, 84(2):023007, 2011.

[32] N. Sarin, P. D. Lasky, L. Sammut, and G. Ashton. X-ray guided gravitational-wave search for binary neutron star merger remnants. *Physical Review D*, 98(4):043011, 2018.

[33] B. F. Schutz. Gravitational wave astronomy. *Classical and Quantum Gravity*, 16(12A):A131, 1999.

[34] S. L. Shapiro and S. A. Teukolsky. *Black holes, white dwarfs, and neutron stars: The physics of compact objects*. John Wiley & Sons, 2008.

[35] M. Sieniawska and M. Bejger. Continuous gravitational waves from neutron stars: current status and prospects. *Universe*, 5(11):217, 2019.

[36] L. Sun and A. Melatos. Application of hidden markov model tracking to the search for long-duration transient gravitational waves from the remnant of the binary neutron star merger gw170817. *Physical Review D*, 99(12):123003, 2019.

[37] H. L. Van Trees. *Detection, estimation, and modulation theory, part I: detection, estimation, and linear modulation theory*. John Wiley & Sons, 2004.

[38] J. Vanderplas, A. Connolly, Ž. Ivezić, and A. Gray. Introduction to astroml: Machine learning for astrophysics. In *Conference on Intelligent Data Understanding (CIDU)*, pages 47 –54, oct. 2012.

[39] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

Appendix A. Complete analysis results

Here we show the other results that we have obtained in this work following the same pipeline discussed in Section 4.1. In the following figures and tables the comparison between the output candidates, the fitted models and the true models (in the log base) and the residuals between the output candidates, the fitted models and the true models (in the log base) are shown.

Case study: $\Delta f = [295 - 300]\,\text{Hz}$ **and** $\alpha = 10^{-1}$ This case study was characterized by a CR threshold of $\text{CR}_{thr} = 2$ and a lfft = 4096 value. We choose such lfft value since we observed that for a lfft = 8192 the candidates were more scattered around the main structures representing the true signals and also less candidates were correctly matched, for both power law and exponential frequency signals. The results are shown in Figures 11 and 12 and in Tables 3 and 4.

Table 3: Comparison between linear regression coefficients and expected values (in the log base) for the case study $\Delta f = [295 - 300]\,\text{Hz}$ and $\alpha = 10^{-1}$.

Frequency	r^2 **value**	**Fit Coeff.**	**True Coeff.**	**Fit Intercept**	**True Intercept**
power law	0.946	-0.252 ± 0.001	-0.25	5.70377 ± 5e-5	5.70382
exponential	0.976	-0.996 ± 0.004	-1	5.70371 ± 4e-5	5.70382

Table 4: Residuals between the final output candidates, the fitted models and the true models (in the log base) for the case study $\Delta f = [295 - 300]\,\text{Hz}$ and $\alpha = 10^{-1}$.

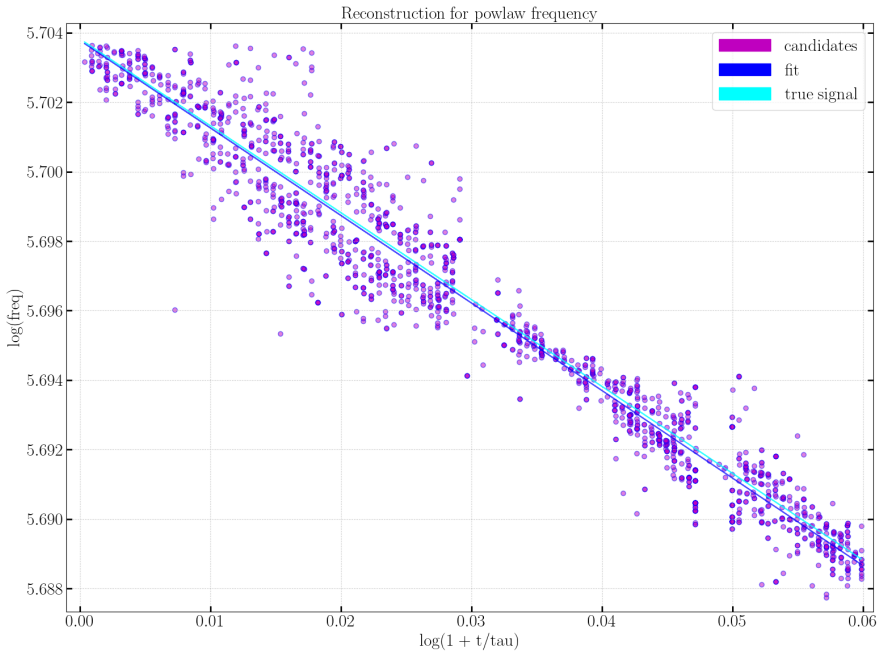
Frequency	**Fit vs Cand. median ± std**	**Fit vs True median ± std**	**Cand. vs True median ± std**
power law	5e-5 ± 0.001	-9.5e-5 ± 3e-5	-2e-4 ± 0.001
exponential	-3e-5 ± 7e-4	-8e-5 ± 2e-5	-1e-5 ± 7e-4

Case study: $\Delta f = [107 - 108]\,\text{Hz}$ **and** $\alpha = 10^{2}$ This case study was characterized by a CR threshold of $\text{CR}_{thr} = 8$ and a lfft = 1024 value. The results are shown in Figures 13 and 14 and in Tables 5 and 6.

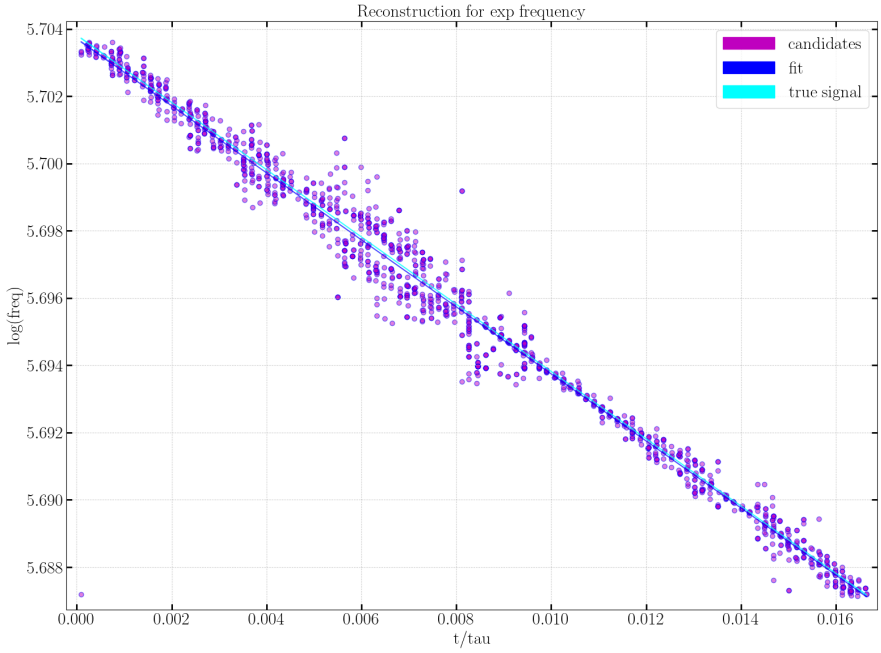
Table 5: Comparison between linear regression coefficients and expected values (in the log base) for the case study $\Delta f = [107 - 108]\,\text{Hz}$ and $\alpha = 10^{2}$.

Frequency	r^2 **value**	**Fit Coeff.**	**True Coeff.**	**Fit Intercept**	**True Intercept**
power law	0.9924	-0.2502 ± 6e-4	-0.25	4.68218 ± 1e-5	4.68222
exponential	0.967	-0.989 ± 0.005	-1	4.68210 ± 3e-5	4.68222

Case study: $\Delta f = [107 - 108]\,\text{Hz}$ **and** $\alpha = 10^{-1}$ This case study was characterized by a CR threshold of $\text{CR}_{thr} = 8$ and a lfft = 1024 value. The results are shown in Figures 15 and 16 and in Tables 7 and 8.

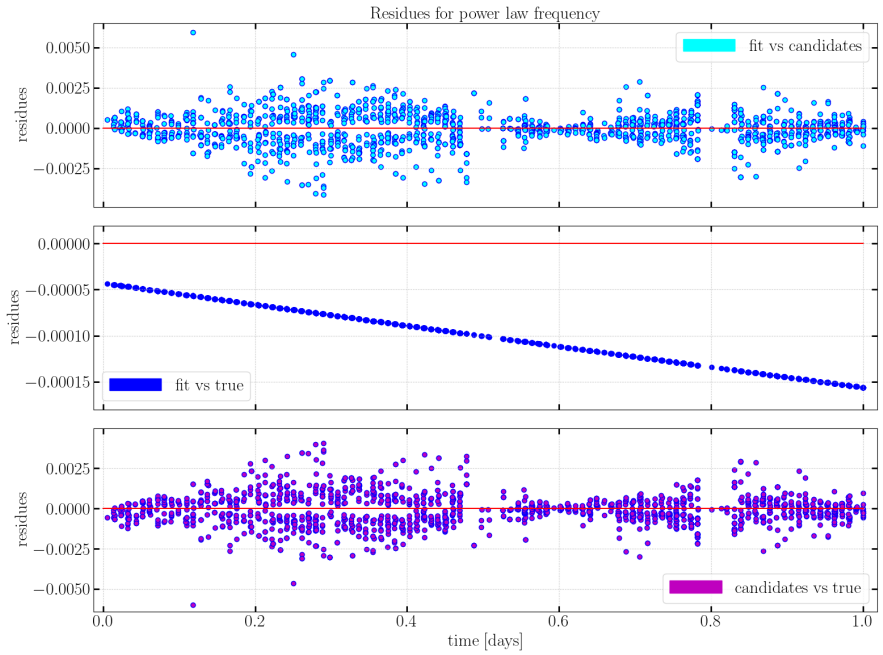


(a) Power law frequency

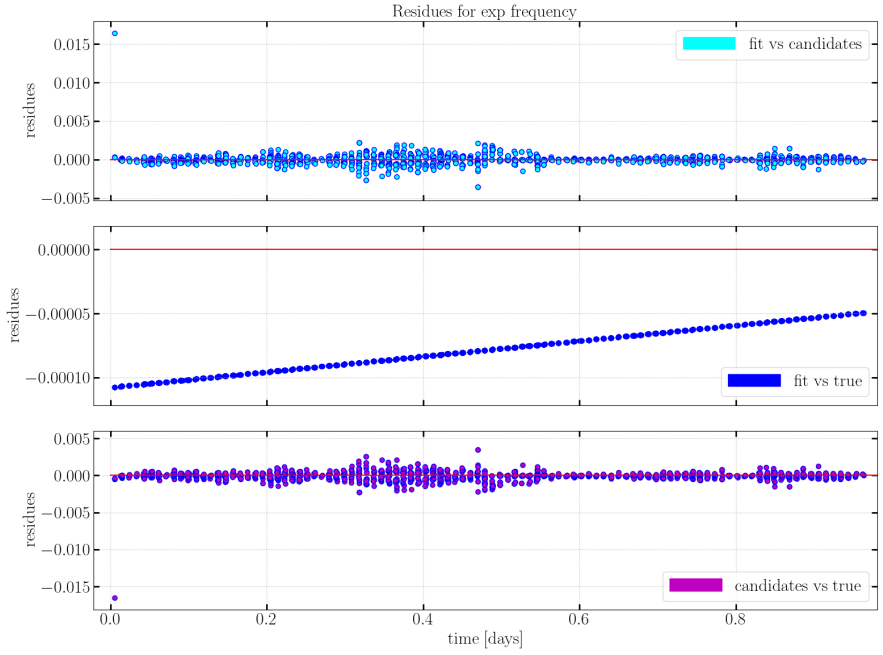


(b) Exponential frequency

Figure 11: Comparison between the output candidates, the fitted model and the true model (in the log base) for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^{-1}$.

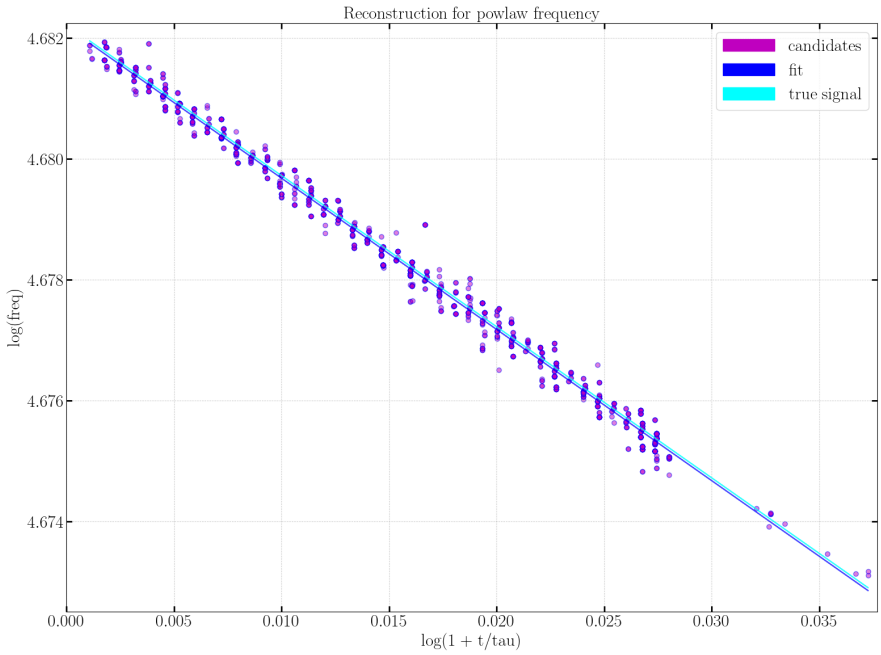


(a) Power law frequency

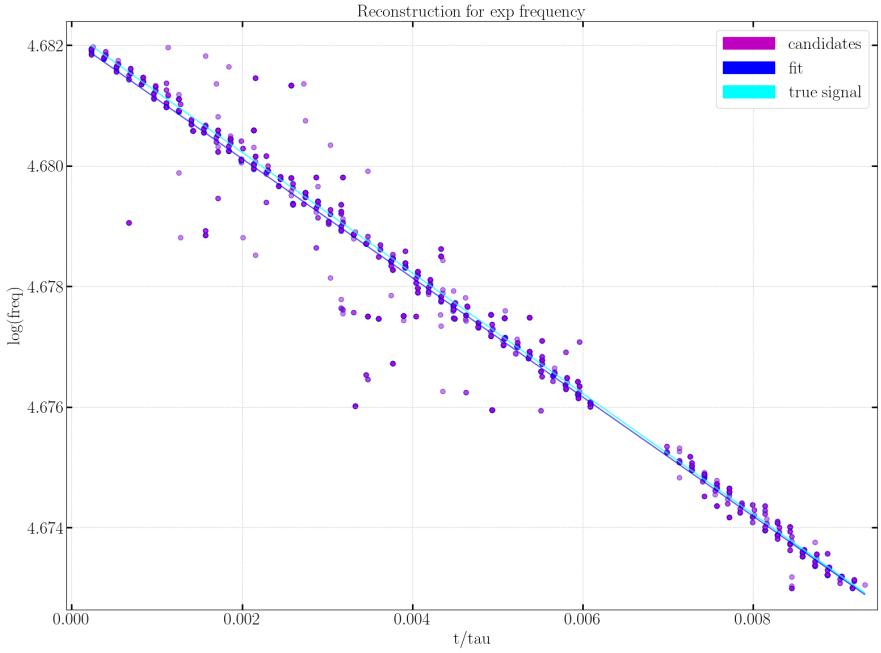


(b) Exponential frequency

Figure 12: Residuals between the final output candidates, the fitted models and the true models for the case study $\Delta f = [295 - 300]$ Hz and $\alpha = 10^{-1}$.

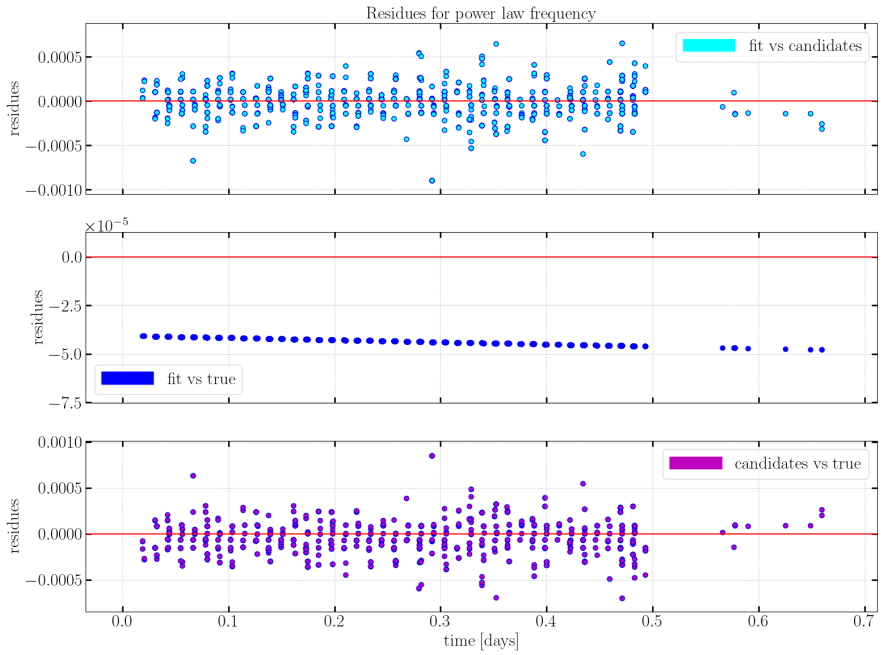


(a) Power law frequency

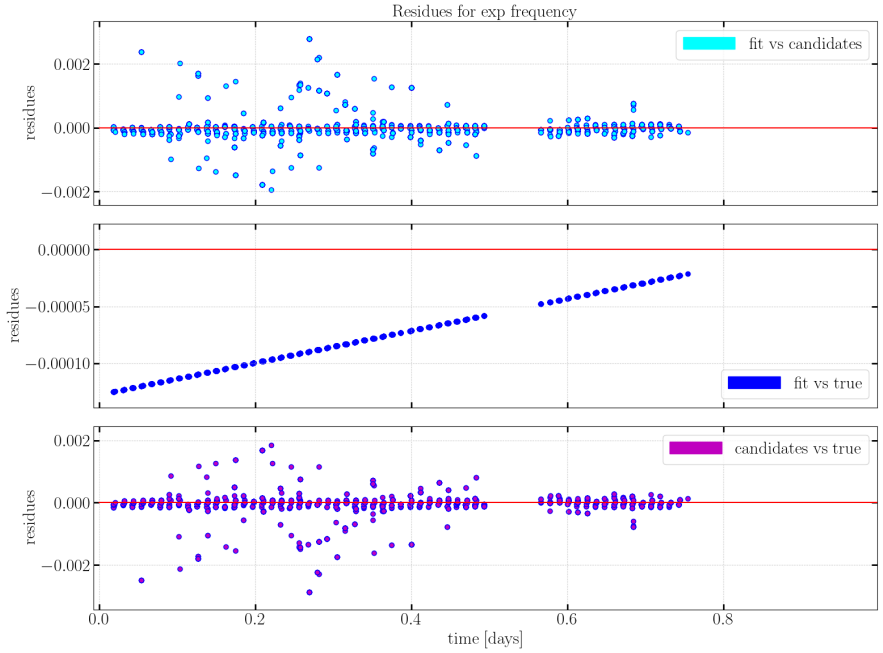


(b) Exponential frequency

Figure 13: Comparison between the output candidates, the fitted model and the true model (in the log base) for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^2$.



(a) Power law frequency



(b) Exponential frequency

Figure 14: Residuals between the final output candidates, the fitted models and the true models for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^2$.

Table 6: Residuals between the final output candidates, the fitted models and the true models (in the log base) for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^2$.

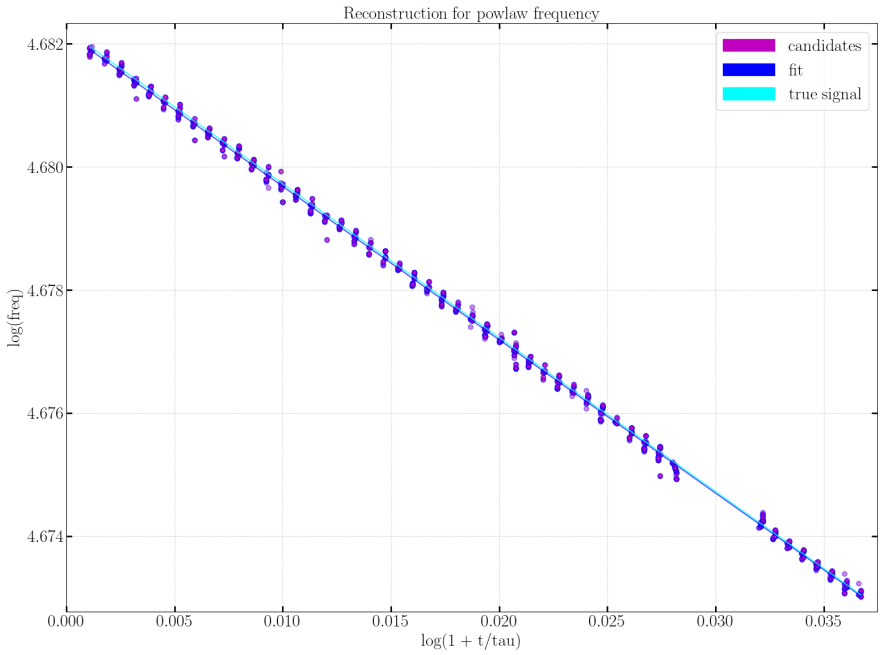
Frequency	**Fit vs Cand. median ± std**	**Fit vs True median ± std**	**Cand. vs True median ± std**
power law	2e-5 ± 2e-4	-4.4e-5 ± 2e-6	-1e-4 ± 2e-4
exponential	-5e-5 ± 5e-4	-8e-5 ± 3e-5	-1e-4 ± 5e-4

Table 7: Comparison between linear regression coefficients and expected values (in the log base) for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^{-1}$.

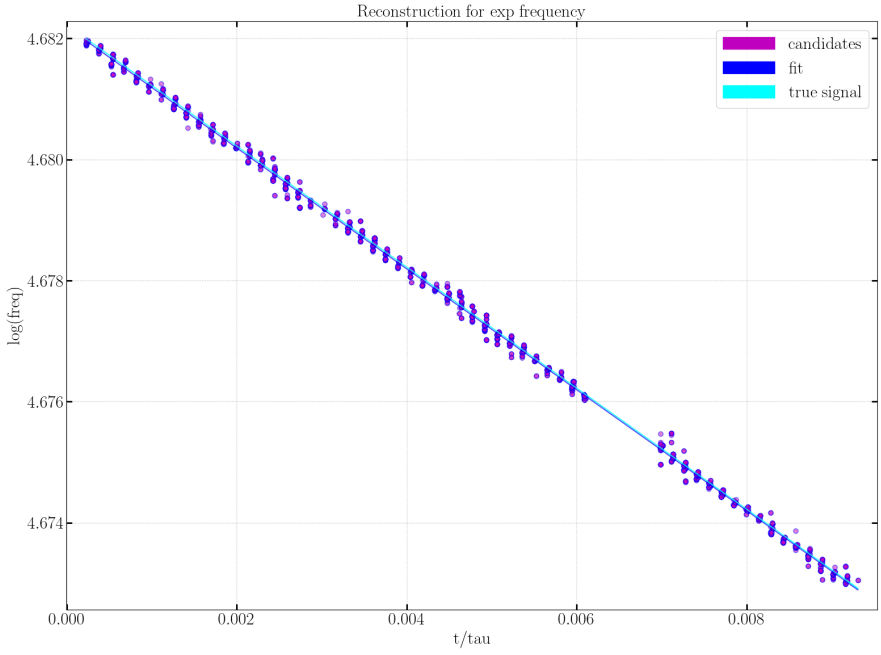
Frequency	**r^2 value**	**Fit Coeff.**	**True Coeff.**	**Fit Intercept**	**True Intercept**
power law	0.9988	-0.2492 ± 2e-4	-0.25	4.682174 ± 5e-6	4.682224
exponential	0.9988	-0.9987 ± 9e-4	-1	4.682189 ± 4e-6	4.682224

Table 8: Residuals between the final output candidates, the fitted models and the true models (in the log base) for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^{-1}$.

Frequency	**Fit vs Cand. median ± std**	**Fit vs True median ± std**	**Cand. vs True median ± std**
power law	-1e-5 ± 9e-5	-3.5e-5 ± 8e-6	-3e-5 ± 9e-5
exponential	-1e-5 ± 9e-5	-2.8e-5 ± 3e-6	-3e-5 ± 9e-5

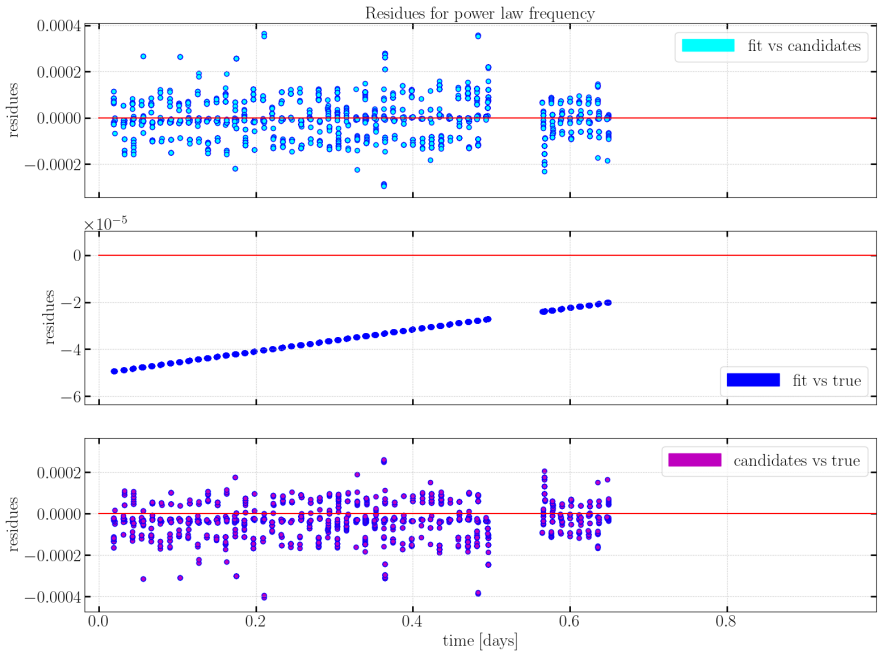


(a) Power law frequency

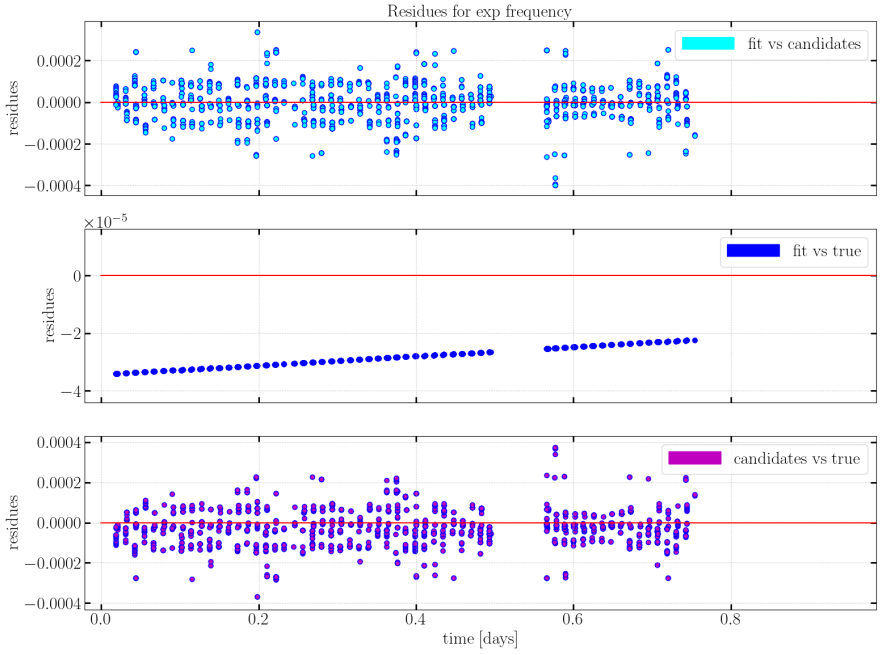


(b) Exponential frequency

Figure 15: Comparison between the output candidates, the fitted model and the true model (in the log base) for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^{-1}$.



(a) Power law frequency



(b) Exponential frequency

Figure 16: Residuals between the final output candidates, the fitted models and the true models for the case study $\Delta f = [107 - 108]$ Hz and $\alpha = 10^{-1}$.

Appendix B. Matlab codes for the signals generation

In this Appendix Section we show the two Matlab codes used to generate the signals following the discussion in Section 2. These algorithms have the purpose to model and to adapt the signals data to the interferometer output data format. All the information used to generate the signals are stored in different Matlab structured array and saved as a Matlab data file (*.mat* file).

```
function driver_injections_in_bsd_data_gw_transient(goutL)
iplot=1; % 1/0 plot frequency and amplitude
SD=86164.09053083288;
t0_O3=58574; % O3 starting time
tcoe=(t0_O3+60); % coalescing time [days]
tcoes=tcoe*86400; % coalescing time [s]
days = 1
t=tcoes:tcoes+days*86400; % time vector [s]
% signal parameters:
tau=1500000;
nbreak=5; % breaking index
fgw0=108.01 % gw frequency
fgw=fgw0*(1 +(t-tcoes)/tau).^(1/(1-nbreak));

H0=(fgw0^2)*1.0e-25; % max signal amplitude
K=1e-5
H=H0*(1+(t-tcoes)/tau).^(2/(1-nbreak));
if iplot==1
    figure,plot((t-min(t))/86400,fgw)
    hold on
    grid on
    xlabel('time [days]')
    ylabel('frequency [Hz]')
    plot((tcoes-min(t))/86400,max(fgw),'*')
    figure,plot(t/86400,H);
    hold on
    grid on
    xlabel('time [days]')
    ylabel('H')
    plot(tcoe,max(H),'*')
    figure,plot((t-min(t))/86400,H.*sin(2*pi*fgw))
end

% sour structure:
sour.type='transient'
sour.a=94.9690
sour.d=-17.3432
sour.v_a=0
sour.v_d=0
sour.pepoch=tcoe;
sour.tcoe=tcoe;%
sour.f0=fgw0
sour.df0=0
```

```
sour.ddf0=0
sour.fepoch=tcoe;
sour.ecl=[96.2585 -40.6746]
sour.t00=tcoe
sour.eps=1
sour.eta=-0.8530
sour.psi= 8.9378
sour.h=1.0000e-20 % not used
ssour.nr=1
sour.coor=0
sour.chphase= 0
sour.dfrsim= 0
sour.PH0=0
sour.k=K;
sour.n=nbreak
sour.tau=tau
sour.days=days
%

tini=tcoe-1; % starting time of BSD selected data
tfin=tini+2;   % stop time of BSD selected data
band=[107 108] % freq. band to be extracted
RUN='C01_GATED_SUB60HZ_O3' % O3 RUN name
goutLC=cut_bsd(goutL,[tini tfin]) % rigth time interval selection
% name of the bsd with injected signal
fileSinjL=strcat('bsd_softInj_LL_longtransient_powlaw',
                 RUN,'_',num2str(band(1)),
                 '_',num2str(band(2)),'_','.mat');
% LL injection
A=1; % to be changed befor ato dd the BSD signal to the BSD data
gSinjL=0*goutL;
[gsinjL]=bsd_softinj_re_mod_gen(goutLC,sour,A);
gsinjL=bsd_zeroholes(gsinjL); % zeroes in the bsd Sinj file
% save the BSD with the injected signal
save(fileSinjL,'gsinjL','sour');
% Amplitude factor factor to inject a signal of amplitude AMP
FACTOR=0.5*10^20;
AMP=0*5*10^-15*FACTOR;
bsd_data_plus_signal=goutL+AMP*gsinjL; % BSD data + BSD signal

lfft=740
%P=bsd_peakmap(bsd_data_plus_signal,lfft);
P=bsd_peakmap(gsinjL,lfft);
p=P.pt.peaks;
plot_triplets(P.pt.peaks(1,:)-tcoe,p(2,:),(p(3,:)),'.','jet')
title('peakmap')
ylabel('frequency[Hz]')
xlabel('time [days]')
end
```

```
function [out Hp Hc]=bsd_softinj_re_mod_gen(in,sour,A)
% Snag Version 2.0 - February 2019
% Part of Snag toolbox - Signal and Noise for Gravitational Antennas
% by O.J.Piccinni, S. Frasca, C.Palomba - Sabrina D'Antonio
% email: sergio.frasca@roma1.infn.it
% Department of Physics - Sapienza University - Rome

Tsid=86164.09053083288; % at epoch2000

frsid=1/Tsid;
inisid1=0;
N=n_gd(in);
dt=dx_gd(in);
cont=cont_gd(in);
y=y_gd(in);
t0=cont.t0;
inifr=cont.inifr;
ant1=cont.ant;
eval(['ant=' ant1 ';'])

sour=new_posfr(sour,t0);
fr=[sour.f0,sour.df0,sour.ddf0];
f0=sour.f0;
if exist('sdpar','var')
    if length(sdpar) == 1
        sdpar(2)=0;
    end
    fr(2)=sdpar(1);
    fr(3)=sdpar(2);
else
    sdpar(1)=fr(2);
    sdpar(2)=fr(3);
end

obs=check_nonzero(in,1);
if isfield(sour,'type')
    switch sour.type
        case 'transient'
            f0gw=sour.f0;
            PH0=sour.PH0;
            nbreak=sour.n;
            K=sour.k;
            tcoe=sour.tcoe; % sour.fepoch=coalescing time
            % N of sample for the coales.
            Ncoa=round((tcoe-t0)*86400/dt)+1;
            fgw0=sour.f0; % freq. sour at t0;
            alpha=(2-nbreak)/(1-nbreak);
            tau=-((fgw0)/2)^(1-nbreak)/(K*(1-nbreak));
            if isfield(sour,'tau')
                tau=sour.tau
```

```
            end
            % avoid alias.
            fgw(1:N)=0;
            fgw(Ncoa:N)=(f0gw)*(1 +(0:N-Ncoa)*dt/tau).
                        ^(1/(1-nbreak))-inifr;
            jck=find((fgw)>=cont.bandw |fgw <0 );
            fgw(jck)=0;
            %
            ph0=cumsum(fgw)*2*pi*dt;
            in=edit_gd(in,'y',exp(1j*ph0).*obs');
            VPstr=extr_velpos_gd(in);
            [p0 v0]=interp_VP(VPstr,sour);
            fr=diff(ph0)/(2*pi);
            fr(length(ph0))=fr(length(ph0)-1); %% CHECK
            % fr is a vector with the vring frequency
            out=vfs_subhet(in,-fr.*v0')
            einst=einst_effect(t0:dt/86400:t0+(N-0.5)*dt/86400);
            out=vfs_subhet(out,(einst-1)*(fgw0));
            % signal amplitude:
            sig0=y_gd(out);
            sig0(Ncoa:N)=sig0(Ncoa:N).*((1+(0:N-Ncoa)*dt/tau).
                        ^(2/(1-nbreak)))';
            sig0(1:Ncoa)=0;
            sig0(jck)=0*sig0(jck);
            sig=sig0*0;
        case 'other'
    end

else % for typical sour struct CW
    ph0=mod((0:N-1)*dt*(f0-inifr),1)*2*pi;
    in=edit_gd(in,'y',exp(1j*ph0).*obs');
    VPstr=extr_velpos_gd(in);
    p0=interp_VP(VPstr,sour);
    out=vfs_subhet_pos(in,fr,-p0);
    if sum(abs(sdpar)) > 0
        gdpar=[dt,N];
        sd=vfs_spindown(gdpar,sdpar,1);
        out=vfs_subhet(out,-sd);
    end
    einst=einst_effect(t0:dt/86400:t0+(N-0.5)*dt/86400);
    out=vfs_subhet(out,(einst-1)*fr(1));
    sig0=y_gd(out);
    sig=sig0*0;
end


% 5-vect

nsid=10000;
% running Greenwich mean sidereal time
stsub=gmst(t0)+dt*(0:N-1)*(86400/Tsid)/3600;
```

```
% time indexes at which the sidereal response is computed
isub=mod(round(stsub*(nsid-1)/24),nsid-1)+1;
% computation of the sidereal response
[~, ~ , ~, ~, sid1 sid2]=check_ps_lf(sour,ant,nsid);
gl0=sid1(isub).*sig0.';
gl45=sid2(isub).*sig0.';
% estimated + complex amplitude (Eq.2 of ref.1)
Hp=sqrt(1/(1+sour.eta.^2))*(cos(2*sour.psi/180*pi)
    -1j*sour.eta*sin(2*sour.psi/180*pi));
Hc=sqrt(1/(1+sour.eta.^2))*(sin(2*sour.psi/180*pi)
   +1j*sour.eta*cos(2*sour.psi/180*pi));
sig=Hp*gl0+Hc*gl45;
sig=sig.';
out=edit_gd(out,'y',A*sig);
out=bsd_zeroholes(out);
```

Appendix C. Python codes for data analysis

In this last Appendix Section we show the Python algorithms we used in our simulation and analysis (18), (23), (39), (29), (19), (38), [4].
Once we had our Matlab data files, we complete our simulation and the analysis with Python. The Python libraries we used are shown in Listing 1.
The first task of the analysis was to read the Matlab data files. After several attempts, we found that the best way to load the data files and read them with Python was to use Dictionaries. As shown in the Listing 2 in order to load both Matlab data files and version *v*7.3 Matlab data files we use the libraries *Scipy* (which contains the *loadmat* method) and *hdf5storage*.
These two libraries read Matlab structured arrays as structured NumPy arrays of dtype (data type) objects. The size of these arrays refers to the size of the whole structured arrays, and not to the number of elements in any particular field (which usually defines the length of arrays). When loading Matlab data files the shape is set to default as 1-dim and 2-dim for *hdf5storage* and *Scipy* respectively, so we have to correctly index the dictionary entries. The dictionaries are built by using the names from the Matlab data file dtypes.
The "key" argument refers to the name of the structured array contained in the Matlab data file with the relative information. If the Matlab data file that we are loading refers to the interferometer output, "key" must be equal to "gout" (which stands for g-out) plus the capital letter that represent the name of the interferometer (L for LIGO-Livingston, H for LIGO-Hanford and V for Virgo). If instead the Matlab data file that we are loading refers to the generated signals, "key" must be equal to "ginj" (which stands for g-injected) plus the capital letter that represent the name of the interferometer in which the signals have been injected. In the same Matlab data file also the structured array relative to the signals source is present. To load the information about the source we have to put "key" equals to "sour" (which stands for source).

Listing 1: List of the Python libraries we used for the simulation and the analysis.

```
import numpy as np                                  # operations
```

4. hdf5storage: `https://pypi.org/project/hdf5storage/`

```
import pandas as pd                                # dataframe

from scipy.io import loadmat                       # matlab datafile
import hdf5storage as hd                           # matlab datafile -v7.3

from scipy.optimize import curve_fit               # fitting
from sklearn import linear_model
from sklearn.metrics import r2_score

import matplotlib.pyplot as plt                    # plotting
import matplotlib.patches as mpatches
from matplotlib.ticker import AutoMinorLocator
from astroML.plotting import setup_text_plots
setup_text_plots(fontsize=12, usetex=True)
```

Listing 2: Conversion of the selected MATLAB data file structure to a Python dictionary.

```
def mat_to_dict(path, key = 'goutL', mat_v73 = False):

    """
    Conversion from MATLAB data file to dict.
    Parameters:
    ------------------------------------------------------------------
    path : (str) path of your MATLAB data file
    key : keyword with info from L, H or V interferometer (insert
           gout + interf. or gsinj + interf., default = goutL)
    mat_v73 : (bool) if the matlab datafile version is -v7.3 insert
                the 'True' value (default = 'False')
    ------------------------------------------------------------------
    return:
    data_dict: (dict) dict from MATLAB data file
    perczero: (float) percentage of total zero data in y
               (data from the Obs run)
    ------------------------------------------------------------------
    """

    if mat_v73 == True:

        mat = hd.loadmat(path)                  # load mat-file -v7.3
        # variable in mat file
        mdata = mat[key]
        # dtypes of structures are "unsized objects"
        mdtype = mdata.dtype
        # express mdata as a dict
        data_dict = {n: mdata[n][0] for n in mdtype.names}

        y = data_dict['y']
        y = y.reshape(len(y))
        data_dict['y'] = y
        y_zero = np.where(y == 0)[0]
```

```
        # perc of total zero data in y (data from the obs. run)
        perczero = len(y_zero)/len(y)

        cont = data_dict['cont']
        cont_dtype = cont.dtype
        # cont in data_dict remains a structured array (--> dict)
        cont_dict = {u: cont[str(u)] for u in cont_dtype.names}
        data_dict['cont'] = cont_dict

        return data_dict, perczero
    else:
        mat = loadmat(path)                 # load mat-file

        if key == 'sour':
            # variable in mat file
            mdata = mat['sour']
            # dtypes of structures are "unsized objects"
            mdtype = mdata.dtype
            # express mdata as a dict
            data_dict = {n: mdata[n][0, 0] for n in mdtype.names}

            return data_dict
        else:
            # variable in mat file
            mdata = mat[key]
            # dtypes of structures are "unsized objects"
            mdtype = mdata.dtype
            # express mdata as a dict
            data_dict = {n: mdata[n][0, 0] for n in mdtype.names}

            y = data_dict['y']
            y = y.reshape(len(y))
            data_dict['y'] = y
            y_zero = np.where(y == 0)[0]
            # perc of total zero data in y (data from the Obs run)
            perczero = len(y_zero)/len(y)

            cont = data_dict['cont']
            cont_dtype = cont.dtype
            # cont in data_dict is a structured array (--> dict)
            cont_dict = {u: cont[str(u)]
                         for u in cont_dtype.names}
            # now we have a fully accessible dict
            data_dict['cont'] = cont_dict

            return data_dict, perczero
```

Listing 3: Power law frequency for the long transient signal.

```
def freq_pow_law(fgw0, tau, nbreak, tcoes, t, log = False):
```

```
    """
    Power law frequency for the long transient signal.
    Parameters:
    ---------------------------------------------------------------
    fgw0 : (float) initial frequency [Hz]
    tau : (int or float) characteristic time [s]
          (or same S.I. units as t)
    nbreak : (int or float) breaking index (= 5 from NS studies)
    tcoes : (int) coalescing time [s] (or same S.I. units as t)
    t : (arrray) time vector [s]
    log : (bool) it gives the log of the frequency (default = False)
    ---------------------------------------------------------------
    return:
    freq: (array) freq signal [Hz]
    ---------------------------------------------------------------
    """

    if log == True:
        return np.log(fgw0*(1. + (t - tcoes)/tau)**(1./(1 - nbreak)))
    else:
        return fgw0*(1. + (t - tcoes)/tau)**(1./(1 - nbreak))
```

Listing 4: Exponential frequency for the long transient signal.

```
def freq_exp(fgw0, tau, tcoes, t, log = False):

    """
    Exponential frequency for the long transient signal.
    Parameters:
    ---------------------------------------------------------------
    fgw0 : (float) initial frequency [Hz]
    tau : (int or float) characteristic time [s]
          (or same S.I. units as t)
    tcoes : (int) coalescing time [s] (or same S.I. units as t)
    t : (array) time vector [s]
    log : (bool) it gives the log of the frequency (default = False)
    ---------------------------------------------------------------
    return:
    freq: (array) freq signal [Hz]
    ---------------------------------------------------------------
    """

    if log == True:
        return np.log(fgw0) - (t - tcoes)/tau
    else:
        return fgw0*np.exp(-(t - tcoes)/tau)
```

Listing 5: Frequency of the long transient signal.

```
def signal_freq(bsd, lfft, signal = 'power law', show_freq = False):

    """
    Frequency of the long transient signal.
    Parameters:
    ---------------------------------------------------------------
    bsd : (dict) bsd from get_data
    lfft : (int) fft length
    signal : (string) define the frequency signal
             ('power law' or 'exp', default = power law)
    show_freq : (bool) if show_freq is True, signal_freq return the
                 complete frequency and time array (default = False)
    ---------------------------------------------------------------
    return:
    freq: (array) freq signal in the chosen frequency interval [Hz]
    time: (array) time values of freq signal [s]
    ---------------------------------------------------------------
    """

    dx = bsd['dx']                          # sampling time [s]
    source = bsd['source']                  # source parameters
    TFFT = lfft*dx                          # FFT time duration
    inifr = bsd['inifr']                    # initial bin freq of data
    bandw = bsd['bandw']                    # bandwidth  of the bin
    finfr = inifr + bandw                   # final freq bin of data

    ##### signal parameters

    tcoe = source['tcoe'][0, 0]             # coalescing time [days]
    tcoe = np.int64(tcoe)
    # coalescing time [s] (86400 # s in one day)
    tcoes = tcoe*86400
    fgw0 = source['f0'][0, 0]               # initial frequency [Hz]

    ##### define frequency

    if signal == 'power law':               # power law frequency

        tau = source['tau'][0, 0]           # characteristic time [s]
        nbreak = source['n'][0, 0]          # breaking index
        days = source['days'][0, 0]         # N of days after tcoes
        days = np.int64(days)
        # time vector [s]
        time = np.arange(tcoes, tcoes + days*86400)

        freq = freq_pow_law(fgw0, tau, nbreak, tcoes, time)
        # first element of freq < finfr [Hz]
        f_ini = np.where(freq < finfr)[0][0]
        # last element of freq > inifr [Hz]
```

```
        f_fin = np.where(freq > inifr)[0][-1]
        # p, q | len(freq) = n*TFFT with n natural number
        p, q = int((f_ini//TFFT + 1)*TFFT), int((f_fin//TFFT)*TFFT)
        # freq_resh contains inifr < freq < finfr [Hz]
        freq_resh = freq[p:q]
        # time_resh array [s] with respect to freq in (inifr, finfr)
        time_resh = time[p:q]

    elif signal == 'exp':                # exp frequency

        tau = source['tau'][0, 0]        # characteristic time [s]
        days = source['days'][0, 0]      # N of days after tcoes
        days = np.int64(days)
        # time vector [s]
        time = np.arange(tcoes, tcoes + days*86400)

        freq = freq_exp(fgw0, tau, tcoes, time)
        # first element of freq < finfr [Hz]
        f_ini = np.where(freq < finfr)[0][0]
        # last element of freq > inifr [Hz]
        f_fin = np.where(freq > inifr)[0][-1]
        # p, q | len(freq) = n*TFFT with n natural number
        p, q = int((f_ini//TFFT + 1)*TFFT), int((f_fin//TFFT)*TFFT)
        # freq_resh contains inifr < freq < finfr [Hz]
        freq_resh = freq[p:q]
        # time_resh array [s] with respect to freq in (inifr, finfr)
        time_resh = time[p:q]

    if show_freq == True:
        return freq, time
    else:
        return freq_resh, time_resh
```

Listing 6: Database containing the Gaussian templates for the matched filtering.

```
def TFunc_gauss_DB(lfft, bsd, step = 1, signal = 'power law',
                   save_to_csv = False):

    """
    Filter Database (Gaussian Template).
    Parameters:
    ---------------------------------------------------------------
    lfft : (int) fft length
    bsd : (dict) dict from get_data with info
    step : (int) number of Gaussian template for bins of length
           TFFT = lfft*dx (default = 1)
    signal : (string) define the frequency signal ('power law' or
             'exp', default = power law)
    save_to_csv : (bool) if you want to save the dataframe insert
                  the 'True' value (default = 'False')
```

```
    ------------------------------------------------------------
    return:
    database: (pandas.core.frame.DataFrame) pandas dataframe with
              the Gaussian templates, Gaussian mean values and
              std values
    ------------------------------------------------------------
    """

    dx = bsd['dx']                          # sampling time [s]
    bandw = bsd['bandw']                    # bandwidth  of the bin
    dfr = 1./lfft*dx                        # freq resolution
    freq = np.linspace(0, bandw, lfft)      # freq interval [0, bandw)
    TFFT = lfft*dx                          # FFT time duration
    div = lfft//128                         # sigma dividend

    # define the transfer function (Gaussian template)
    def ES_filter(f, Mu, Sigma):
        return np.exp(-(f - Mu)**2/(2.0*Sigma**2))

    ##### define std for templates
    if signal == 'power law':               # power law frequency

        f_pow = signal_freq(bsd, lfft)[0]
        # std for Gaussian templates
        sigma = np.array([np.abs(f_pow[int(i*TFFT)] -
                         f_pow[int((i - 1)*TFFT)])/div
                         for i in range(1, int(len(f_pow)/TFFT))])

    elif signal == 'exp':                   # exp frequency

        f_exp = signal_freq(bsd, lfft, signal = signal)[0]
        # std for Gaussian templates
        sigma = np.array([np.abs(f_exp[int(i*TFFT)] -
                         f_exp[int((i - 1)*TFFT)])/div
                         for i in range(1, int(len(f_exp)/TFFT))])

    ##### generate database
    # mean values array
    mu = np.linspace(dfr, bandw - dfr, len(sigma)*step)
    # list with templates
    gauss_db = list(ES_filter(freq, mu[i], sigma[i//step])
                    for i in range(0, len(mu)))[step:-step]
    # creating empty dataframe with chosen dim
    database = pd.DataFrame(index = range(len(freq)),
                            columns = range(len(gauss_db)))
    # inserting template in df columns
    for i in range(len(gauss_db)):
        database[i] = gauss_db[i]
    # rename columns
    database.columns = list("mu = {value1}".format(value1 = v)
                            for v in mu[step:-step])
```

```
    std = []
    # modelling the sigma array (if step != 1 more mu have the same
    # sigma and the number of templates increases)
    for s in sigma[1:-1]:
        std += [s]*step

    par = pd.DataFrame([mu[step:-step], std], index = ['mu', 'std'],
                       columns = database.columns)
    # add mu and sigma to dataframe
    database = pd.concat([database, par])

    if save_to_csv == True:              # save the dataframe as csv
        name = 'MFGauss'

        if signal == 'power_law':
            p_law = name + '_power_law_database.csv'
            database.to_csv(p_law)

        elif signal == 'exp':
            exp = name + '_exp_database.csv'
            database.to_csv(exp)

    return database
```

Listing 7: Signal injection into the interferometer output data.

```
def get_data(path_bsd_gout, path_bsd_gsinj, y_edge,
             key = 'goutL', mat_v73 = False):

    """
    It takes some info from bsd_gout (dx, n, y, t0, inifr, bandw) and
    from bsd_gsinj (y) to obtain noise + signal for the filtering.
    Parameters:
    ---------------------------------------------------------------
    path_bsd_gout : (str) bsd_gout containing the interferometer's
                     noise
    path_bsd_gsinj : (str) _gsinj containing the injected signal
    y_edge : (int) number of y_gout elements which surround y_gsinj
             (recommended at least 1*lfft)
    key : (str) keyword with info from L, H or V interferometer
          (insert gout + interf. or gsinj + interf., default = goutL)
    mat_v73 : (bool) to load matlab datafile version is -v7.3
              (default = 'False')
    ---------------------------------------------------------------
    return:
    bsd_out: (dict) bsd with the info to use for filter data chunk
              and for the database
    ---------------------------------------------------------------
    """
```

```
    # gout and perczero of y_gout
    bsd_gout, perczero_gout = mat_to_dict(path_bsd_gout, key = key,
                                          mat_v73 = mat_v73)
    # gsinj and perczero of y_gsinj
    bsd_gsinj, perczero_gsinj = mat_to_dict(path_bsd_gsinj,
                                            key = 'gsinjL')
    source = mat_to_dict(path_bsd_gsinj, key = 'sour')     # source

    # sampling time of the input data
    dx = bsd_gout['dx'][0, 0]
    # data from the gout bsd
    y_gout = bsd_gout['y']
    # data from the gsinj bsd
    y_gsinj = bsd_gsinj['y']

    # starting time of the gout signal [days]
    try:
        t0_gout = bsd_gout['cont']['t0'][0, 0]
    except:
        t0_gout = bsd_gout['cont']['t0'][0, 0, 0]

    # starting time of the signal [days]
    tcoe = source['tcoe'][0, 0]
    t0_gout = np.int64(t0_gout)
    tcoe = np.int64(tcoe)
    # index of gout data with respect to injected signal
    t_ind = int(((tcoe - t0_gout)*86400)/dx)

    # we take a chunk of y_gout that contain y_gsinj
    y_gout = y_gout[t_ind - y_edge:t_ind + len(y_gsinj) + y_edge]
    # number of samples to consider for filter data chunk
    n_new = len(y_gout)

    # reshaping y_sinj to sum with y_gout
    y_gsinj_resh = np.hstack([np.zeros(y_edge, dtype = complex),
                              y_gsinj,
                              np.zeros(n_new - len(y_gsinj) - y_edge,
                              dtype = complex)])

    # amplitude factor to inject a signal of amplitude AMP
    amp = 1e2
    y_tot = amp*y_gsinj_resh + y_gout          # signal + noise

    y_zero = np.where(y_tot == 0)[0]
    # perc of total zero data in y (data from the Obs run)
    perczero = len(y_zero)/len(y_tot)
    # initial bin freq and bandwidth of the bin
    try:
        inifr = bsd_gout['cont']['inifr'][0, 0][0, 0]    # normal -v
        bandw = bsd_gout['cont']['bandw'][0, 0][0, 0]
```

```
    except:
        inifr = bsd_gout['cont']['inifr'][0, 0, 0]       # -v7.3
        bandw = bsd_gout['cont']['bandw'][0, 0, 0]

    # output dictionary with main information
    bsd_out = {'dx': dx,
               'n': n_new,
               'y': y_tot,
               'perczero': perczero,
               'inifr': inifr,
               'bandw': bandw,
               'y_edge' : y_edge,
               'source': source}

    return bsd_out
```

Listing 8: Definition of the Critical Ratio.

```
def CR(x):

    """
    It performs the Critical Ratio of the candidates using the
    median of input data.
    Parameters:
    ---------------------------------------------------------------
    x : (array) matched filtered data
    ---------------------------------------------------------------
    return:
    Crit Ratio : (array)
    ---------------------------------------------------------------
    """

    m_1 = np.median(x)
    m_2 = np.median(np.abs(x - m_1))/0.6745
    try:
        # ignoring indefinites values in y (i.e. 0/0 or inf)
        np.seterr(invalid = 'ignore')
        y = np.abs(x - m_1)/m_2
    except:
        y = -0.1
    return y
```

Listing 9: Candidates selection within the chunks during the matched filtering.

```
def sel_candidates(data, ncand, threshold, dx, t0, t_ini):

    """
    It lists the output candidates from matched filtered data.
```

```
    Parameters:
    ----------------------------------------------------------------
    data : (array) matched filtered data
    ncand : (int) number of searched candidates
    threshold : (float) threshold for the Critical Ratio
    dx : (float) sampling time
    t0 : (int) signal starting time
    t_ini : (float) starting time of data
    ----------------------------------------------------------------
    return:
    out_candidates : (list) output candidates
    ----------------------------------------------------------------
    """

    dfstep = len(data)//ncand      # divide data in ncand sub-chunk
    Crit_R = CR(data)              # CR of data
    out_candidates = []

    for i in range(0, len(data), dfstep):
        # max of CR in the sub-chunk
        cr_max = np.max(Crit_R[i:i + dfstep])
        # index of cr_max
        ind = np.argmax(Crit_R[i:i + dfstep])

        # time of the candidates (from t = 0, len(data) = lfft)
        t_cand = t_ini + (i + ind)*dx/86400

        # candidates selection
        if cr_max >= threshold:
            a = [t_cand, cr_max, "sub_chunk = {k}".format(k = i)]
        else:
            a = "CR below threshold"

        out_candidates.append(a)

    return out_candidates
```

Listing 10: Matched filtering process.

```
def filter_data_chunk(bsd, lfft, ncand, threshold, BPFilter):

    """
    It performs the Matched Filtering throught the input gout and
    returns the output candidates.
    Parameters:
    ----------------------------------------------------------------
    bsd : (dict) dict from MATLAB data file
    lfft : (int) fft length
    ncand : (int) number of searched candidates per chunk
    threshold : (float) threshold for the Critical Ratio
```

```
    BPFilter : (pandas.core.frame.DataFrame) dataframe with filter
               templates
    ------------------------------------------------------------
    return:
    database : (pandas.core.frame.DataFrame) output candidates with
               relative info: time, critical ratio, mean values and
               std of the Gaussian templates
    ------------------------------------------------------------
    """

    dx = bsd['dx']          # sampling time of the input data
    TFFT = lfft*dx          # FFT time duration
    n = bsd['n']            # total number of samples in the input BSD

    # percentage of zeros above which a chunk is not analyzed
    perczero = bsd['perczero']

    ##### init:
    C_out = []                # list with matched filtered candidates
    y = bsd['y']              # data from the bsd
    # signal starting time
    t0 = bsd['source']['tcoe'][0, 0]
    # number of samples in lfft unit
    n_lfft = (n//lfft)*lfft

    ##### loop for chunks
    for j in range(0, n_lfft - lfft//2, lfft//2):
        # select a chunck between [j, j + lfft] in y_gout
        y_chunk = y[j:j + lfft]
        # find the N of zeros in the chunk
        jzero = len(np.where(y_chunk == 0)[0])

        # FFTs removed if percentage of 0 > perczero
        if jzero <= perczero*lfft:
            # normalised FFT of the chunk
            fft_y = np.fft.fft(y_chunk, n = lfft, norm = "ortho")*dx
            # starting time of each data chunk (initial time = 0)
            t_ini = (j//lfft)*(TFFT + dx)/86400

            for column in BPFilter.columns:
                # select filter template
                T_Func = np.array(BPFilter[column][:-2],
                                  dtype = complex)
                # convolution (frequency domain)
                DataMatched_Freq = np.conjugate(T_Func)*fft_y
                DataMatched_bound = np.real(
                                    np.fft.ifft(DataMatched_Freq,
                                    n = lfft, norm = "ortho"))
                # remove boundary (affected by FFT transform)
                DataMatched = DataMatched_bound[lfft//16: -lfft//16]
                C_out.append(sel_candidates(DataMatched, ncand,
```

```
                              threshold, dx, t0, t_ini))

    # columns of BPFilter
    col = [column for column in BPFilter.columns]
    # N columns of BPFilter
    ncol = BPFilter.shape[1]
    # join filter for the same chunk
    cand_out_tot = [C_out[i:i + ncol]
                    for i in range(0, len(C_out), ncol)]

    # database with output
    C_database = pd.DataFrame(cand_out_tot,
                              index = ["chunk = {u}".format(u = i)
                                       for i in range(
                                       len(cand_out_tot))],
                              columns = col)

    # mu and std from BPFilter
    mu_db, std_db = np.array(BPFilter.iloc[-2]),
                    np.array(BPFilter.iloc[-1])
    par = pd.DataFrame([mu_db, std_db],
                       index = ['mu', 'std'], columns = col)
    # add mu and sigma to candidates dataframe
    database = pd.concat([C_database, par])

    return database
```

Listing 11: Extraction of the output candidates from the whole set of matched templates depending on the chosen critical ratio threshold.

```
def get_candidates(candidates, bsd, lfft, choosetem = True):

    """
    It selects the output candidates from the total candidates list
    based on the chosen threshold.
    Parameters:
    ---------------------------------------------------------------
    candidates : (pandas.core.frame.DataFrame) dataframe with total
                 candidates (also those with CR < threshold)
    bsd : (dict) dict from get_data with info
    lfft : (int) fft length
    choosetem : (bool) choose the template with higher CR at fixed
                frequency and time (default = False)
    ---------------------------------------------------------------
    return:
    output candidates : (pandas.core.frame.DataFrame) output
                        candidates with relative info: time (from
                        t = 0), time (in days from start), critical
                        ratio, mean value, std (Gauss templates)
```

```
    ------------------------------------------------------------------
    """

    output = []
    out_list = []
    out_list_2 = []
    # extract candidates with CR >= threshold from the input database
    for ind in candidates.index[:-2]:
        for column in candidates.columns:

            a = candidates[column][ind]

            for element in a:
                if not element == "CR below threshold":
                    output.append(element +
                                  [ind, candidates[column]['mu'],
                                  candidates[column]['std']])

    # candidates selection
    if choosetem == True:
        # extracted candidates dataframe
        df = pd.DataFrame(output,
                          columns = ['t_ini', 'CR', 'sub_chunk',
                          'chunk', 'mu', 'std'])

        # template mean values
        mu_db = np.array(candidates.iloc[-2], dtype = float)
        # choose the template with higher CR (at fixed frequency)
        for h in mu_db:
            try:
                t = df.loc[df['mu'] == h]
                r = t.loc[t['CR'] == t['CR'].max()]
                out_list.append(r)
            except:
                pass

        df_out = pd.concat([u for u in out_list])
        # choose the template with higher CR (at fixed time)
        for l in df_out['t_ini']:
            try:
                z = df_out.loc[df_out['t_ini'] == l]
                x = z.loc[z['CR'] == z['CR'].max()]
                out_list_2.append(x)
            except:
                pass

        df_out_2 = pd.concat([u for u in out_list_2])
    else:
        # candidates dataframe (no candidates screening)
        df_out_2 = pd.DataFrame(output, columns = ['t_ini', 'CR',
                                'sub_chunk', 'chunk', 'mu', 'std'])
```

```
    # time correction for y_edge and matlab to python conversion
    dx = bsd['dx']                  # sampling time of the input data
    TFFT = lfft*dx                  # FFT time duration
    # number of y_gout elements which surround y_gsinj
    y_edge = bsd['y_edge']
    # signal coalescing time [days]
    tcoe = bsd['source']['tcoe'][0, 0]
    time_correction = 1 + (y_edge//lfft)*(TFFT + dx)/86400
    df_out_2['t_ini'] = df_out_2['t_ini'] - time_correction
    # frequency and time assignment to the candidates
    df_out_2['mu'] = df_out_2['mu'] + bsd['inifr']
    df_out_2.insert(1, 't_days', df_out_2['t_ini'] + tcoe)

    return df_out_2
```

Listing 12: This algorithm group the main Python functions in order to load the data we need, to perform the signal injection and the matched filtering process.

```
def Matched_Filtering(path_gout, path_gsinj, lfft, y_edge, step,
                      ncand, threshold, signal = 'power law',
                      key = 'goutL', mat_v73 = False):

    """
    It performs the Matched Filtering process. Inside this function
    get_data, TFunc_gauss_DB and filter_data_chunk are performed.
    Parameters:
    ---------------------------------------------------------------
    path_gout : (str) bsd_gout containing the interferometer's noise
    path_gsinj : (str) _gsinj containing the injected signal
    lfft : (int) fft length
    y_edge : (int) number of y_gout elements which surround y_gsinj
             (recommended at least 1*lfft)
    step : (int) number of Gaussian template for bins of length
            TFFT = lfft*dx (default = 1)
    ncand : (int) number of searched candidates
    threshold : (float) threshold for the Critical Ratio
    signal : (string) frequency signal ('power law' or 'exp',
              default = power law)
    key : keyword with info from L, H or V interferometer (insert
          gout + interf. or gsinj + interf., default = goutL)
    mat_v73 : (bool) if the matlab datafile version is -v7.3 insert
               the 'True' value (default = 'False')
    ---------------------------------------------------------------
    return:
    candidates : (pandas.core.frame.DataFrame) output candidates with
                 relative info: time, critical ratio, mean values
                 and std of the Gauss templates
    database: (pandas.core.frame.DataFrame) pandas dataframe with the
```

```
                Gaussian templates, Gaussian mean values and std
    bsd_out: (dict) bsd with the info to use for filter data chunk
              and for the database
    ----------------------------------------------------------------
    """

    # get data for signal freq
    bsd_data = get_data(path_gout, path_gsinj, y_edge, key = key,
                        mat_v73 = mat_v73)

    # TFunc_gauss_DB output
    database = TFunc_gauss_DB(lfft, bsd_data, step, signal = signal)

    # filter data chunk
    candidates = filter_data_chunk(bsd_data, lfft, ncand, threshold,
                                   database)

    return candidates, database, bsd_data
```

Listing 13: Templates display.

```
def display_templates(mu, std, bandw, lfft):

    """
    It displays the Gaussian templates from the arrays mu
    (mean values) and sigma (std values).
    Parameters:
    ----------------------------------------------------------------
    mu : (array) dataframe with candidates (can be also the DF path)
    sigma : (array) initial bin freq of data
    bandw : (int) bandwidth of the considered frequency bin
    lfft : (int) fft length
    ----------------------------------------------------------------
    return:
    Plot with the templates
    ----------------------------------------------------------------
    """

    # define the transfer function (Gaussian template)
    def ES_filter(f, Mu, Sigma):
        return np.exp(-(f - Mu)**2/(2.0*Sigma**2))

    f = np.linspace(0, bandw, lfft)

    # templates plot
    plt.figure()
    for i, j in zip(mu, std):
        template = ES_filter(f, i, j)
        plt.plot(f, template)
    plt.grid(True)
```

```
    plt.xlabel('freq [Hz]')
    plt.ylabel('template')
    plt.title('Gaussian templates')
    plt.show()
```

Listing 14: Candidates display.

```
def display_candidates(df_input, fig_num = 1, thr = None,
                       signal = 'power law', bsd = None, lfft= None,
                       show_totfreq = False, save_plot = False,
                       save_to_csv = False, freqband = False):

    """
    It displays the output candidates and gives their info.
    Parameters:
    ------------------------------------------------------------
    df_input : (pandas.core.frame.DataFrame or str) dataframe with
               candidates (can be also the DF path)
    fig_num : (int) number of the figure (default = 1)
    thr : (float) threshold for the Critical Ratio
    signal : (string) frequency signal ('power law' or 'exp',
             default = power law)
    bsd : (dict) dict from get_data with info (default = None)
    lfft : (int) fft length (default = None)
    show_totfreq : (bool) if True, display_candidates shows the
                   complete frequency signal (default = False)
    save_plot : (bool) if you want to save the plot as PNG image
    save_to_csv : (bool) if you want to save the dataframe insert
                  the 'True' value (default = 'False')
    freqband : (list) frequency bandwidth (info for the file name)
    ------------------------------------------------------------
    return:
    mu, t, CR, std : (array) output info with candidates mean values,
                     time (in days from start), critical ratio,
                     templates std values
    ------------------------------------------------------------
    """

    # if df_input is the df path, this load the dataframe
    if type(df_input) == str:
        df_input = pd.read_csv(df_input)
    else:
        pass

    # get candidates with CR over threshold
    if not thr == None:

        df_thr = df_input.loc[df_input['CR'] >= thr]
        # info in the dataframe with CR over thr
        mu = np.array(df_thr['mu'])
```

```
        t = np.array(df_thr['t_ini'])
        CR = np.array(df_thr['CR'])
        std = np.array(df_thr['std'])

    else:
        mu = np.array(df_input['mu'])        # info in the dataframe
        t = np.array(df_input['t_ini'])
        CR = np.array(df_input['CR'])
        std = np.array(df_input['std'])

    # to display all the true freq signal
    if (bsd != None and lfft != None):

        if show_totfreq == True:
            freq, time = signal_freq(bsd, lfft, signal = signal,
                                     show_freq = True)
        else:
            freq, time = signal_freq(bsd, lfft, signal = signal)

    ### plot
    fig = plt.figure(num = fig_num)
    ax = fig.add_subplot(111)
    # scatter plot of candidates
    f1 = ax.scatter(t, mu, c = CR, cmap = 'jet', alpha=0.8)
    # plot true frequency signal (if keywords are given)
    try:
        tcoe = bsd['source']['tcoe'][0, 0]  # coalescing time [days]
        time_sc = time/86400 - tcoe
        ax.plot(time_sc, freq, color='OrangeRed',
                label='True frequency signal')
    except:
        pass

    plt.colorbar(f1, label = 'CR values')    # colorbar for CR values
    plt.legend(loc = 'best')                 # legend

    # title, labels and ticks options
    if signal == 'power law':
        plt.title('Power law frequency long transient')
    elif signal == 'exp':
        plt.title('Exp frequency long transient')
    plt.xlabel('t [days]')
    plt.ylabel('freq [Hz]')
    ax.grid(True)
    ax.label_outer()
    ax.tick_params(which='both', direction='in',width=2)
    ax.tick_params(which='major', direction='in',length=7)
    ax.tick_params(which='minor', direction='in',length=4)
    ax.xaxis.set_ticks_position('both')
    ax.yaxis.set_ticks_position('both')
    # save plot as PNG image
```

```
    if save_plot == True:
        plt.savefig("plot.png", format = 'png')

    plt.show()                                      ### end plot

    # save the dataframe as csv
    if save_to_csv == True:
        name = 'Output_candidates'
        name_lfft = 'lfft' + str(lfft)
        band = '_' + str(freqband[0]) + 'to' +
                str(freqband[1]) + 'Hz_'

        if signal == 'power law':
            db_powlaw = name + '_powerlaw_database' + band +
                        name_lfft + '.csv'
            if not thr == None:
                df_thr.to_csv(db_powlaw)
            else:
                df_input.to_csv(db_powlaw)

        elif signal == 'exp':
            db_exp = name + '_exp_database' + band +
                     name_lfft + '.csv'
            if not thr == None:
                df_thr.to_csv(db_exp)
            else:
                df_input.to_csv(db_exp)

    return mu, t, CR, std
```

Listing 15: Residues display.

```
def residues_plot(x, y1, y2, y3, signal = 'power law'):

    """
    It displays the residues y1, y2 and y3.
    Parameters:
    ---------------------------------------------------------------
    x : (array) x-axis data
    y1 : (array) y-axis data for 1st subplot
    y2 : (array) y-axis data for 2nd subplot
    y3 : (array) y-axis data for 3rd subplot
    signal : (string) frequency signal ('power law' or 'exp',
              default = power law)
    ---------------------------------------------------------------
    return:
    plot from matplotlib
    ---------------------------------------------------------------
    """
```

```
    # create the figure and three subplots with shared x-axis
    fig, axs = plt.subplots(3, 1, figsize=(6, 8), sharex=True)
    # plot the data on each subplot
    axs[0].scatter(x, y1, c = 'cyan')
    axs[1].scatter(x, y2, c = 'b')
    axs[2].scatter(x, y3, c = 'm')

    for ind in range(3):
        axs[ind].plot([0, 1], [0, 0], c = 'r')
        axs[ind].grid(True)
        axs[ind].label_outer()
        axs[ind].tick_params(which='both', direction='in',width=2)
        axs[ind].tick_params(which='major', direction='in',length=7)
        axs[ind].tick_params(which='minor', direction='in',length=4)
        axs[ind].xaxis.set_ticks_position('both')
        axs[ind].yaxis.set_ticks_position('both')

    # add titles to each subplot
    if signal == 'power law':
        axs[0].set_title('Residues for power law frequency')
    elif signal == 'exp':
        axs[0].set_title('Residues for exp frequency')
    axs[1].set_title('')
    axs[2].set_title('')
    # add y labels to the subplots
    axs[0].set_ylabel('residues')
    axs[1].set_ylabel('residues')
    axs[2].set_ylabel('residues')
    axs[2].set_xlabel('time [days]')
    # add a legend to the bottom subplot and adjust its position
    patch1 = mpatches.Patch(color='cyan', label='fit vs candidates')
    patch2 = mpatches.Patch(color='b', label='fit vs true')
    patch3 = mpatches.Patch(color='m', label='candidates vs true')

    for ind, patch in zip([0, 1, 2], [patch1, patch2, patch3]):
        axs[ind].legend(handles=[patch], loc='best')

    # adjust the layout of the subplots to avoid overlapping
    fig.tight_layout()
    plt.show()
```

Listing 16: Candidates outliers removal.

```
def candidates_remover(df, x_min, x_max, y_min, y_max):

    """
    It removes the candidates within the rectangle defined by the
    values of x_min, x_max, y_min, y_max.
    Parameters:
    ----------------------------------------------------------------
```

```
    df : (pandas.core.frame.DataFrame) dataframe with the candidates
    x_min : (float) min value along x-axis
    x_max : (float) max value along x-axis
    y_min : (float) min value along y-axis
    y_max : (float) max value along y-axis
    ------------------------------------------------------------
    return:
    df : (pandas.core.frame.DataFrame) dataframe with the candidates
          without those that were inside the rectangle
    ------------------------------------------------------------
    """

    # if df_input is the df path, this load the dataframe
    if isinstance(df, str):
        df = pd.read_csv(df)

    mask = (df['t_ini'] > x_min) & (df['t_ini'] < x_max) &
           (df['mu'] > y_min) & (df['mu'] < y_max)

    df = df.loc[~mask]
    return df
```

Listing 17: Standard errors of the fit regression coefficients.

```
def coeffs_errors(x, y, y_fit):

    """
    It computes the standard errors of the fit regression coefficients.
    Parameters:
    ------------------------------------------------------------
    x : (array) x-axis data for fit
    y : (array) y-axis data for fit
    y_fit : (array) fitted y-axis data
    ------------------------------------------------------------
    return:
    se_slope : (float)
    se_intercept : (float)
    ------------------------------------------------------------
    """

    # Calculate the sum of squared residuals
    ssr = np.sum((y - y_fit)**2)

    # Calculate the degrees of freedom
    n = len(x)
    p = 1
    dof = n - p - 1              # p = 1 because we have 1D data

    # Estimate the variance of the error terms
    mse = ssr/dof
```

```
    # Calculate the standard errors of the slope and intercept
    se_slope = np.sqrt(mse/np.sum((x - np.mean(x))**2))
    se_intercept = np.sqrt(mse*(1/n + np.mean(x)**2/
                           np.sum((x - np.mean(x))**2)))

    return se_slope, se_intercept
```

Listing 18: Fitting process of the output candidates and residuals computation for both power law and exponential frequency signals.

```
### path for interf. output data and sources
path_bsd_gout = "bsd␣gout␣-␣path␣(.mat)"
path_bsd_softInj = "bsd␣injected␣signal␣-␣path␣(.mat)"
### path for dataframe with candidates
path_df = "candidates␣dataframe␣-␣path␣(.csv)"
### get info from gout and injected signal
bsd = ltp.get_data(*arg)

### outliers remotion
for x1, x2, y1, y2 in zip(*coordinates):

    try:
        df = ltp.candidates_remover(*arg)
    except:
        df = ltp.candidates_remover(*arg)

parameters = ltp.display_candidates(*arg)

###  verify the matched filtering through scikit linear regression
## log values for fit
mu = parameters[0]                  # candidates freq
t = parameters[1]                   # candidates time
ln_mu = np.log(mu)                  # freq log
ln_t = np.log(1 + t/tau)            # time log (power law)
ln_t = t/tau                        # time log (exp)

## linear regression
regr = linear_model.LinearRegression()
# freq fit
regr.fit(ln_t.reshape(len(ln_t), 1), ln_mu)
fit_templ = regr.predict(ln_t.reshape(len(ln_t), 1))
# regression coefficients
a_fit, b_fit, c_fit = regr.coef_, regr.intercept_, r2_score(*arg)
# standard errors of the regression coefficients
se_a, se_b = ltp.coeffs_errors(ln_t, ln_mu, fit_templ)

###  residues
freq_signal = ltp.freq_pow_law(*arg)   # freq true signal
```

```
res1 = fit_templ - ln_mu                # fit-candidates res
res2 = fit_templ - freq_signal          # fit-true signal res
res3 = ln_mu - freq_signal              # candidates-true signal res
ltp.residues_plot(*arg)                 # plot residues
```