Dipartimento di Fisica

CORSO DI LAUREA MAGISTRALE
INTERATENEO IN FISICA

Curriculum Astrofisica & Cosmologia

TESI DI LAUREA MAGISTRALE

Consistency test for gravitational–wave signals from compact binary coalescences

Test di consistenza delle forme d'onda gravitazionali prodotte da coalescenze di sistemi binari compatti

Laureanda:
Carolina Bresciani

Relatore:
Prof. Edoardo Milotti

Correlatore:
Dott.ssa Agata Trovato

ANNO ACCADEMICO 2023-2024

Abstract

La ricerca di segnali di orgine astrofisica nei dati degli interferometri per onde gravitazionali è un problema complesso sotto molti punti di vista: da un lato i segnali sono estremamente deboli, dall'altro il rumore di fondo ha una natura non stazionaria e non gaussiana. Per recuperare il segnale dal background sono stati sviluppati molti metodi, che possono essere divisi in due macrogruppi: i metodi modellati e quelli non modellati. I primi si avvalgono di segnali simulati per aiutarsi nella ricerca di quelli reali, i secondi no. Avere più di un metodo di ricerca del segnale e più di un metodo di simulazione dei modelli usati nelle ricostruzioni porta con sè la necessità di confrontare diverse ricostruzioni per assicurarsi che diano risultati compatibili. Se più ricostruzioni non sono compatibili è anche necessario capire quale sia quella migliore. In questa tesi viene presentato un *toy model* che ricrea i passaggi dell'analisi di onde gravitazionali e testa la consistenza dei risultati di tre metodi di ricostruzione, due modellati e uno non modellato, su segnali di tipo *sine-Gaussian*. Questi segnali, definiti da una funzione seno con un inviluppo Gaussiano, sono molto più semplici da parametrizzare rispetto ai segnali che derivano da coalescenze di sistemi binari compatti, ma condividono con questi alcune importanti caratteristiche. In particolare, entrambi i tipi di segnali oscillano, variano in ampiezza e hanno "code" che possono essere confuse con il background.

Contents

1 Introduction

After the first detection of a gravitational wave signal in 2015 (made public in 2016) [1], gravitational wave studies have been metaphorically thrust to the main stage of Physics — as evidenced by the awarding of the 2017 Nobel Prize in Physics to Weiss, Barish, and Thorne for their work at LIGO. Yet, this was hardly an overnight success. Einstein's theory of general relativity — which predicts the existence of gravitational waves — predates the first detection paper by around a hundred years; and a case can be made that Einstein was not the first to imply the existence of gravitational waves [2]. Already in Newton's time action at a distance forces raised doubts in scientists and philosophers . It was from these concerns and from the advances in fluid-dynamics and thermodynamics that over time the idea of a filling medium that transported information came about: it may not have been called a "field" right from the start, but the underlying concept is unequivocally similar [2].

The first overwhelming success of field theories was Maxwell's description of electromagnetism, which was used as a guide by Einstein - and, to be fair, to many who studied relativity with or after him - in the development of a field theory for gravity [2, 3]. What often gets overlooked is that Maxwell himself thought gravity may behave like a field and therefore propagate through waves — and he was not alone, with figures such as Heaviside, Poincaré, and Nordström working from the same assumption. The success of relativity, special first and general later, obscured the efforts of Einstein's "competitors", and made him the most prominent figure in matters of gravity.

For about fifty years after the publication of the first article on general relativity, the idea of gravitational waves was heavily discussed and doubted. The scientific community as a whole finally accepted the existence of gravitational waves when two experiments claimed to have observed an effect caused by them. For one, the ongoing observation of a binary system of neutron stars showed a change in their period consistent with the one predicted by loss of energy via gravitational waves [4]. Secondly, Joseph Weber claimed to have observed gravitational radiation directly [5]. While the latter experiment was later discredited, as attempts to replicate it failed again and again; the former result was confirmed, fueling interest towards the detection of gravitational waves. This interest brought along the necessity of devising both an experimental setup capable of detecting gravitational waves and a way of analyzing the signal received. These two efforts continued in parallel and independently for each collaboration. The results are that today, even though all collaborations work together, both the instrumentation and the pipelines developed by each team are not quite the same.

In particular, many pipelines have been developed through the years, and even though all can (and are) used to reconstruct and/or search for signals with data

from all detectors, they are different both in implementation and guiding principle. Most notably, some pipelines are "modeled" while others are "unmodeled": the former use theoretical waveforms to help in the search for the signals, while the latter do not. Obviously, if one were to analyze a signal with more than one algorithm, while they cannot expect all of them to give the exact same result, it is vital that they give compatible results.

In this thesis, I describe a toy model that emulates the basic steps of both the modeled and unmodeled pipelines, and utilizes their signal reconstructions to evaluate the consistency of the two methods, using several statistical tests. Since the waveforms used in the modeled reconstruction are of two types — an "exact" waveform that corresponds to the one in the simulated signal and a similar one with important differences in the low-SNR part of the signal — the toy model also helps assessing the sensitivity of the consistency tests in separating the two classes of signals.

1.1 The Wave Equation

The following section summarizes material taken from [6] and [7]. For a more in-depth discussion, see the original sources.

All current theories of gravity lay their foundations in the theory of general relativity proposed by Einstein in 1915. In general relativity space and time are treated as a four-dimensional manifold, aptly named space-time, and the gravitational field is described by the Einstein tensor G. This tensor is defined as

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R, \tag{1}$$

where $g_{\mu\nu}$ is the space-time metric, $R_{\mu\nu}$ is the Ricci tensor, and $R = g^{\mu\nu} R_{\mu\nu}$ is the Ricci scalar. The Ricci tensor, and therefore the Ricci scalar, are strictly related to the derivatives of the metric and in this way to the curvature of space-time.
The Einstein's equations connect the Einstein tensor to the energy–momentum tensor

$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}, \tag{2}$$

where the energy-momentum tensor $T_{\mu\nu}$ which describes the energy-matter distribution is the source term. In a vacuum, $T_{\mu\nu} = 0$, and Einstein's equations take the simpler form

$$G_{\mu\nu} = 0, \tag{3}$$

which can be shown to be equivalent to the even simpler equation

$$R_{\mu\nu} = 0. \tag{4}$$

What one usually does to get a wave equation is to perturb a stable solution: the simplest possible case for gravity is to perturb the Minkowski metric $\eta_{\mu\nu} = \text{diag}(-1, 1, 1, 1)$[1] and write the metric tensor as follows

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}, \tag{5}$$

where $h_{\mu\nu}$ is a small perturbation[2]. With the perturbed metric one can compute $G_{\mu\nu}$ at the first order in $h_{\mu\nu}$ to get

$$G_{\mu\nu} = \frac{1}{2}\left(-\Box h_{\mu\nu} - \partial^2_{\mu\nu} h + \partial_\lambda \partial_\nu h^\lambda_\mu + \partial_\lambda \partial^\mu h^\lambda_\nu\right) - \frac{1}{2}\eta_{\mu\nu}\left(-\Box h + \partial_\mu \partial_\nu h^{\mu\nu}\right) = 0 \tag{6}$$

where $h = \eta^{\mu\nu} h_{\mu\nu}$ is the trace of $h_{\mu\nu}$. To do away with the last term, everything is rewritten in terms of the trace reversed metric perturbation, defined as

$$\bar{h}_{\mu\nu} = h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu} h \tag{7}$$

and equation (6) becomes

$$\partial_\mu \partial_\nu \bar{h}^\mu_\sigma + \partial_\rho \partial_\gamma \bar{h}^\rho_\nu - \Box \bar{h}_{\sigma\nu} - \eta_{\nu\sigma} \partial_{\rho\lambda} \bar{h}^{\rho\lambda} = 0 \tag{8}$$

We can use some of the residual degrees of freedom in this equation to further simplify the equation. The first step consists in adopting the Lorentz gauge

$$\partial_\alpha \bar{h}^\alpha_\beta = 0 \tag{9}$$

so that equation (8) becomes

$$\Box \bar{h}_{\sigma\nu} = 0 \tag{10}$$

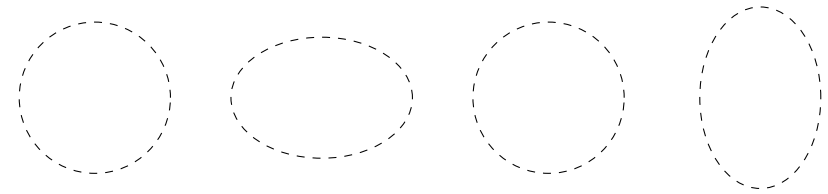
which is the wave equation that describes gravitational waves in vacuum. We still have additional degrees of freedom left that we can use to choose a special reference frame, comoving with the wave, such that there is no longitudinal component of the metric perturbation (the *transverse traceless gauge* or TT gauge). In the TT gauge, with the z-axis parallel to the direction of wave propagation, the amplitude tensor can be written in the form

$$\bar{h}^{\mu\nu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & h_+ & h_\times & 0 \\ 0 & h_\times & h_+ & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{11}$$

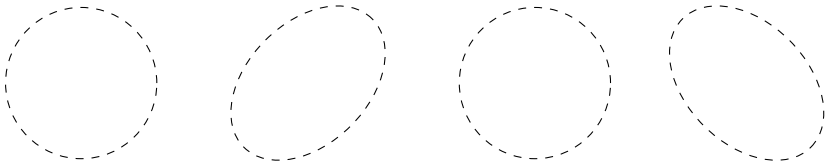
[1] Or $\eta_{\mu\nu} = \text{diag}(1, -1, -1, -1)$, there is no set standard. See [6] for a table of sign conventions.

[2] Here, "small" means $h_{\mu\nu} \ll 1$, so that we can neglect terms that are quadratic or higher order in $h_{\mu\nu}$.

Here, we identify the two polarizations, called *plus* (+) and *cross* (×). For a source like a binary system, the effect of the gravitational wave is an oscillating deformation of space-time, defined by the composition of the stretches and compressions in the plus and cross directions.



(a) Plus polarization oscillation



(b) Cross polarization oscillation

Figure 1: Schematic representation of the distortion caused by a gravitational wave.

1.2 The Quadrupole Formula

The wave equation points to the existence of gravitational waves propagating in vacuum, but it does not say anything about the other two relevant items, the generation of gravitational waves and their detection. The generation of gravitational waves is described by a famous formula, the *quadrupole equation*

$$\bar{h}^{ij}(ct, \mathbf{x}) \approx \frac{2G}{c^4 r} \ddot{Q}^{ij} \left(t - r/c\right), \tag{12}$$

where the quantity

$$Q^{ij} = \int_{\text{source}} \rho x^i x^j \, d^3\mathbf{x} \tag{13}$$

is the *quadrupole tensor* of the mass distribution. This formula was derived by Einstein in his 1918 paper [8] but has been the subject of a long-standing controversy [3] which finally ended when the formula was finally confirmed by the outstanding data describing the rotation of the periastron of the double system PSR 1916+13 [4]. This result is superficially similar to that found in classical electrodynamics (the power emitted by an oscillating dipole is proportional to the

second derivative of the dipole moment), but an analysis of the power emitted by a time-varying quadrupole tensor leads to the more complex formula

$$P_{\mathrm{GW}} = \frac{G}{5c^5} \left\langle \dddot{\mathcal{I}}^{mn} \dddot{\mathcal{I}}_{mn} \right\rangle \tag{14}$$

where $\mathcal{I}_{mn}$ is the *reduced quadrupole tensor*, a traceless version of the original quadrupole tensor. One outstanding feature of this formula is the presence of *third* derivatives of the reduced quadrupole tensor, hinting at the underlying complexity of general relativity.

1.3 Detection of Gravitational Waves from Astrophysical Sources

Although efforts to detect gravitational wave signal began in earnest in the early 1960s after the Chapel Hill conference [9], the technology was still inadequate. In the early 1970s, Rainer Weiss had the breakthrough idea of using interferometric detection instead of Weber's resonant bars [10]. Then, after a long development, the LIGO collaboration started taking the first scientific data at the dawn of the new century. Again, it took a few more years, and a second generation instrument, before gravitational waves could be finally detected, on Sept. 14th, 2015.

Currently, there are three instruments that have been capable of detecting gravitational waves:

- LIGO-Hanford;
- LIGO-Livingston;
- Virgo.

More instruments should join the search in the future: KAGRA, in Japan, had already started taking data when an earthquake hit and is currently being reconfigured, while LIGO India is still in the building stage.

They are all laser interferometers with the basic scheme of a Michelson interferometer, but incorporating many important improvements. These instruments provide direct measurements of the gravitational strain (i.e. the relative differential change in arm length $\Delta L/L$) caused by the incoming gravitational wave — see, e.g., [11] for a basic description of their structure and operation.

Unlike an ordinary laboratory Michelson interferometer, they have arms that range from 3 km in the case of Virgo [12] and KAGRA [13], to 4 km in the case of the two LIGO interferometers [14] and of LIGO India [15]. Their size alone, though, is not enough to reach the sensitivity needed for the detection of

gravitational waves, as noise dominates the data. What makes gravitational wave signal detection possible are the study, the minimization, the control, and — as far as possible — the removal of noise from the data. Noise in gravitational wave interferometers comes from — literally — any sort of vibration, but the main components of noise are:

seismic noise: all the noise associated with ground vibrations,

Newtonian noise: due to slow local changes in the mass distribution around the instrument,

thermal noise: due to thermal fluctuations,

quantum noise: photon pressure noise and shot noise.

These noise components have different power spectral densities (PSD) — and therefore different amplitude spectral densities (ASD). In particular, seismic, Newtonian, and photon pressure noise limit the sensitivity at low frequencies (below 10 Hz), while shot noise dominates at high frequencies (over 1000 Hz) [16]. See figure 2.

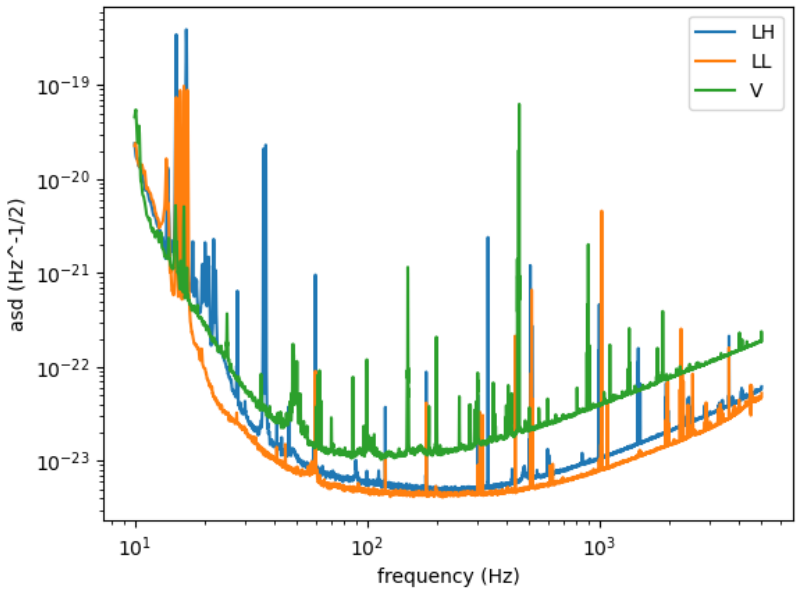
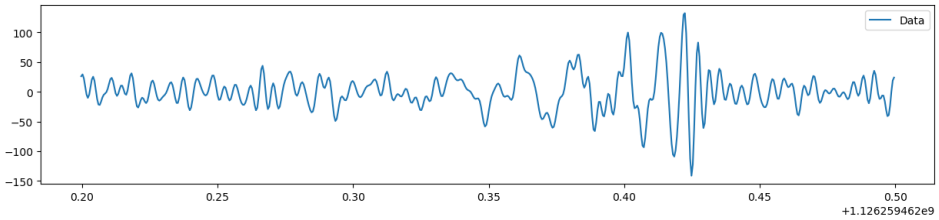


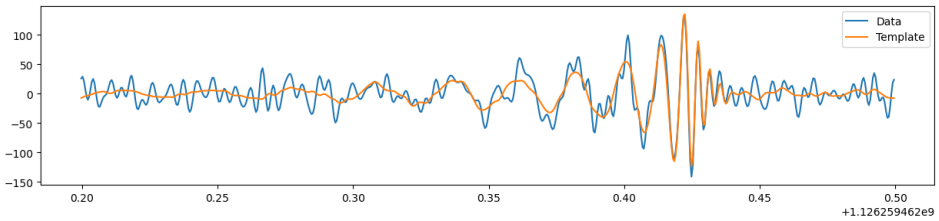
Figure 2: Plots of the mean ASD for LIGO Hanford (LH), LIGO Livingston (LL) and Virgo (V) during the O3 observing run. The data used for these plots were taken from [17].

The peak in sensitivity lies roughly in the 100 Hz − 500 Hz range, where noise is lowest, see figure 2. This is also the frequency range in which relevant astrophysical events are expected to emit gravitational waves. In particular, one would expect to detect binary neutron stars (BNS) mergers, binary black hole (BBH) mergers, neutron star–black hole (NSBH) mergers, isolated neutron stars with non–zero ellipticity, and core–collapse supernova explosions[3]. The first three categories, while different in many aspects, are similar enough to be often grouped together and simply called compact binary coalescences (CBCs). To date, all the detected gravitational waves originated from a CBC.

CBC gravitational waveforms have a characteristic shape in the time–frequency domain, called *chirp*[4], see figure 3 for a very relevant example.



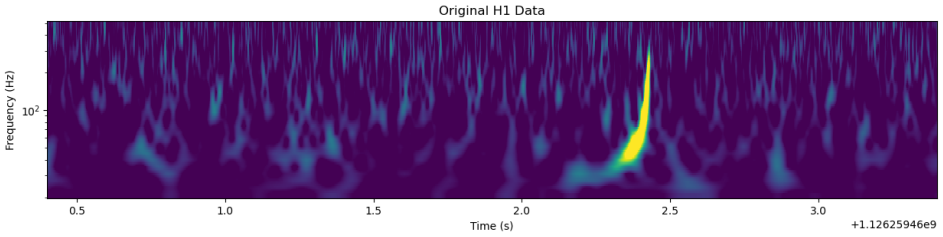
(a) Whitened strain of data from LIGO-Hanford around GW150914.



(b) Whitened strain of data from LIGO-Hanford around GW150914 and signal template. With the template as a guide, is easier to spot that the signal starts much earlier than its peak time.

[3]i.e. SNe type II, and possibly Ib and Ic [18, 19]. Type Ia supernovae could also be detected if generated by a binary white dwarf system as proposed by Iben and Tutukov [20]; but the emission would occur at much lower frequencies [21].

[4]As it turns out, acoustic waves of the same frequencies as those typical of CBCs signals are in the auditory range of humans, so coalescences can be "translated" into a sound [22], which is apparently best described as a high pitched "chirp", like the chirp of birds or bats.



(c) Q-transform of the whitened strain of data from LIGO-Hanford around GW150914: here it is much easier to distinguish the background from the signal - even without a template.

Figure 3: Three different representation of the same signal, GW150914. In looking at these - especially at 3a - it is important to note that this signal is one of the loudest ever recorded, and even in this case some features are hard to distinguish in the time domain. These plots were made using PyCBC [23] and data available through GWOSC [24].

In particular, BNS signals are difficult to recognize in the time domain as, even though they can have high SNRs, the signal is spread over a span of around 100 s; much longer compared to the few tenths of seconds covered by a BBH signal.
Despite the differences in length and shape, all CBC signals are characterized by the same three stages:

- inspiral;
- coalescence;
- ringdown.

The first of the three is the longest: it is characterized by a slight and slow increase in frequency and amplitude, and it corresponds (not surprisingly) to the in-spiral phase of the binary system - that is, when the two bodies are "falling" towards the center of mass of the system following spiral trajectories[5] [1]. The inspiral is difficult to reconstruct in its entirety, as it starts at very low amplitudes and frequencies. The observation of a complete inspiral phase is one of the objectives of space-based gravitational wave interferometers [25].
The second stage, the coalescence (which is emitted when the two bodies are merging), is the easiest to detect, as it is in this stage that the signal reaches maximum amplitude and frequency [1]. This part of the signal is characterized by a rapid increase in both frequency and amplitude, and is the part of the signal that determines the chirp shape.

[5]Much like marbles in a funnel.

The last part of the signal, the ringdown, drops quickly in amplitude, making it quite hard to distinguish it from the background noise. This section of the signal is interesting since it is the last emission that brings the result of the merger to a state of equilibrium. The shape of this last emission can vary when a different gravity model is assumed [25].

When a gravity model is assumed, the shape of a CBC signal can be used to infer physical quantities of the system that generated it. For example, one can obtain information about the masses of the two components of the system, about the mass of the resulting object, about the distance between Earth and the source, and about the spins of the two components [26]. If a detection is made using at least three different detectors, there is also the possibility of locating the source of the gravitational wave [27], thus opening up to the possibility of working in tandem with optical instruments to observe the result of the merger. This is exactly what happened in the case of GW170817, a BNS merger that was detected by gravitational wave interferometers and later connected to a gamma-ray burst (GRB) observed independently by the Fermi Gamma-ray Burst Monitor [28].

The possibility of probing the universe using gravitational waves, or a combination of electromagnetic and gravitational waves, is of immense importance to astrophysics and cosmology. This two branches of physics both suffer of the same ailment: by their own nature they cannot perform experiments, but only make observations, which limits them greatly. Any new way to observe the universe expands this limit a little bit. Moreover, gravitational waves promise to bring information precisely from those events or objects that cannot easily — or at all — be observed with traditional means; such as black holes, neutron stars, supernovae cores, and even the pre-recombination universe.

1.4 Signal Reconstruction

Signal reconstruction is not a problem only in the context of gravitational wave detection; in fact, it is a woe of researchers across many fields of study [29]. What sets apart the case of gravitational waves is that the signals are extremely weak, and especially the tails (i.e. start and the very end of the signal) can easily be completely covered by the noise. As luck would have it, some of the information that physicists are interested in today is found precisely in these hard to reconstruct segments [25], making signal reconstruction a forefront runner in the list of problems to tackle.
Moreover, the future generation of gravitational wave interferometers will be much more sensitive — LISA, for example, is expected to detect signals that last for

weeks at a time: with such signals, it is not only possible but almost certain to have superpositions [30]. In this case, too, the problem is one of signal reconstruction.

There are two main approaches to signal reconstruction: one can assume a model or not. In the former case, one is developing a modeled reconstruction algorithm, in the latter an unmodeled one. While both ways have their merits and downsides, modeled reconstructions are somewhat preferred in the field of gravitational waves — mainly for two reasons:

- a modeled search yields not only the signal shape, but also a parameter estimate;
- modeled searches have better accuracy.

Of course, unmodeled reconstruction algorithms have their place, too: they are mostly used for blind burst searches, targeting any kind of transient signals — not only those for which the shape is already known.

1.4.1 Modeled Reconstruction Methods

It is common knowledge that it is much easier to find something if you know what you are looking for. This almost tautological principle can also be applied in the study of gravitational waves, and it is the foundation on which modeled reconstruction pipelines are built.
The idea behind these methods is that, given a theory of gravity and a source, one can predict which shape the CBC gravitational wave signal should have. The predicted shape is often called a template, and with enough computational power one can generate many templates. Unfortunately for physicists, many is not an infinite amount, which means that there is not a template ready for every single possible signal that one could ever detect. Moreover, even if one did have an infinite amount of templates, it would be quite impractical to compare them one by one with the signal. Different modeled pipelines sample the parameter space in different ways, with the goal of maximizing precision while at the same time minimizing the number of templates to check.
One of the problems modeled methods of reconstruction run into is that some areas of the parameter space may be very sparsely covered. This means that some signals might be missed outright if there is no template close enough in the parameter space.
Another problem that is inherently embedded into modeled methods is the assumption of a model. The issue here is that the most well-established model for gravity — which is the one used to generate the templates — is general relativity

[31], which does not allow for the quantization of the gravitational field. To reconcile relativity and quantum mechanics, theoretical physicists have come up with a plethora of modified gravity models, but, since they have to be compatible with the observation that support general relativity, their predictions differ from those of relativity only in specific situations. In particular, some modified gravity models predict different shapes for CBC signals [25, 31], but as of today no deviations have been confirmed [31].
Lastly, there is more than one method to construct a template. The easiest method to implement is the Newtonian approach, such as the one developed in [32] and [33]. For this method the loss of energy via gravitational wave emission is computed using the quadrupole formula, but then it is applied to a non–relativistic binary system. As an example, if one considers a binary system of two unequal masses m_1 and m_2 on a circular orbits of radii r_1 and r_2 around their center of mass, the positions of the two components will be

$$x_1^i = \pm\left(r_1\cos(\omega t), r_1\sin(\omega t), 0\right) \tag{15}$$
$$x_2^i = \pm\left(r_2\cos(\omega t), r_2\sin(\omega t), 0\right) \tag{16}$$

if the origin of the coordinate system is set at the center of mass and $\hat{z}$ is perpendicular to the plane of orbit. Considering $m = {}^{(m_1 m_2)}/_{(m_1+m_2)}$ and $r = {}^{(r_1+r_2)}/_{2}$, the reduced quadrupole tensor[6] in this case will be

$$\mathcal{I}^{ij} = 2mr^2\begin{pmatrix} \cos^2(\omega t) - {}^1/_3 & \cos(\omega t)\sin(\omega t) & 0 \\ \cos(\omega t)\sin(\omega t) & \sin^2(\omega t) - {}^1/_3 & 0 \\ 0 & 0 & -{}^1/_3 \end{pmatrix} \tag{17}$$

From its third derivative with respect to time one can calculate the power emitted through gravitational waves using (12):

$$P_{GW} = \frac{128G}{5c^5}m^2r^4\omega^6 \tag{18}$$

If this is a non-relativistic binary system, its the total energy can be found (using momentum conservation, force balancing, and Kepler's third law) to be

$$E_{\text{tot}} = \frac{1}{2}I\omega^2 - \frac{Gm_1m_2}{r_1+r_2} = -\frac{1}{2}G^{2/3}\frac{m_1m_2}{(m_1+m_2)^{1/3}}\omega^{2/3} \tag{19}$$

From the quadrupole equation one can compute the strain tensor in terms of the orbital frequency ω, the distance from the source D, and the phase ϕ, which includes the contributions of retarded time and integration constants for the equations of motion. What one finds is

[6]The reduced quadrupole tensor is the nonspherical part of the quadrupole tensor.

$$\bar{h}^{ij}(ct,\vec{r}) = \frac{4c}{D}\left(\frac{GM_\odot}{c^3}\right)^{5/3}\frac{m_1 m_2}{(m_1+m_2)^{1/3}}\omega^{2/3}\begin{pmatrix}\cos(2\omega t+\phi) & \sin(2\omega t+\phi) & 0\\ \sin(2\omega t+\phi) & -\cos(2\omega t+\phi) & 0\\ 0 & 0 & 0\end{pmatrix} \tag{20}$$

It is easy to notice that the strain frequency — i.e. the gravitational wave frequency — is double the orbital frequency, while the amplitude is determined — up to a constant factor — by the masses of the components, the distance from the source and the orbital frequency. Since the rate of energy loss via gravitational emission is known (from equation 18), it can be used to find the instantaneous change in frequency as a function of the masses of the system. Once one knows the rate of change in frequency, the differential equation can be integrated to find the frequency as a function of time and, therefore, the strain as a function of time. This estimate, while useful, can only go as far as the coalescence time, which is defined as the point in time at which the sum of the orbital radii is equal to the sum of the Schwarzschild radii for the two masses. To go further than the coalescence one needs to go beyond a Newtonian approach.

A step above the Newtonian approach is the realm of post-Newtonian approximations. These are a family of approximations that are valid for slow moving (i.e. $v \ll c$) objects in weak fields (i.e. $g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu}$). In a post-Newtonian approach, the wave equation is approximated iteratively by rewriting it in terms of

$$\tau^{\alpha\beta} = -gT^{\alpha\beta} + \frac{1}{16\pi}\Lambda^{\alpha\beta} \tag{21}$$

where Λ encompasses all the non linear terms of the equation. This object acts as an energy momentum tensor, and can be computed at order i in $h^{\alpha\beta}$ to get a correction of order $i+1$. Post-Newtonian approximations, while surprisingly good even in strong field and fast moving conditions, cannot simulate beyond the inspiral phase [34].

There are only three ways to simulate the later stages of a CBC: numerical relativity methods, effective one-body methods, and phenomenological methods. The most accurate among them is by far numerical relativity. This field of physics, which uses a Hamiltonian description of general relativity to solve Einstein's equation numerically, is relatively recent, as the extremely high computational cost of the calculations needed made it almost impossible to even attempt such an approach up until the mid 2000s [35]. Even today, numerical relativity templates are relatively few and cover only a small fraction of the parameter space, as their cost in time and computational resources is too high to make it feasible to cover it all, even sparsely. In particular, systems with high eccentricity or with components of highly different masses cannot be treated with numerical relativity [36, 37].

Effective one-body (often shortened to EOB) methods rely on post Newtonian expansions, but use them to define an effective Hamiltonian for a test particle, which can be used to find the energy and the loss of energy of the binary system from inspiral to ringdown [38]. Phenomenological models are more diverse in how they are implemented, but what they all try to do is find a way to fit existing templates through a series of empirical coefficients leaving physical quantities such as component masses and spins as parameters in the model [35]. Obviously phenomenological models can only be as good as the set they are "trained" on, so they need very accurate waveforms to start with.
It is important to test the consistency of each model, for multiple reasons. Firstly, one should know when it is appropriate to use one waveform or another: a phenomenological model tuned for aligned spins, for example, might give inconsistent result in case of non-aligned spins, and it should be used knowing its limitations. Secondly, since the computational cost of numerical relativity simulations is extremely high, sometimes only the last part of the signal is simulated, while the first is derived through some other method (most often post-Newtonian approximation). In order to patch the two together, it must be checked that they are equivalent in their overlapping range.

1.4.2 Unmodeled Reconstruction Methods

The alternative to modeled methods is not to assume a model at all — neither for gravity nor for the signal. This comes with the advantage of not having a bias towards some particular shapes.
In an unmodeled framework, what one looks for are bursts — which are short segments of data where the mean power exceeds the level that is expected form noise — and tries to reconstruct their shapes [39, 40]. One of the pros of this approach is that any kind of burst with enough power will be picked up. This means that even events with highly improbable parameters, for which there may not be a template, or events with a different astrophysical origin can be studied. Even glitches (i.e. short bursts of instrumental noise) could be treated this way — and in fact they have been [41]. The accuracy of unmodeled pipelines, though, is still not as high as that of modeled ones.
The actual workflow of unmodeled pipelines varies from one algorithm to another, but, broadly speaking, they all follow a similar pattern. The first step is to define a burst quantitatively, e.g. by setting a maximum duration and a minimum power level. Secondly, the raw data from all detectors are collected and whitened. At this point the whitened data from all detectors are analyzed (either separately or all together) to find if there is a coincident burst. The specific way this is done differs from pipeline to pipeline, but, in general, they are transformed — through variations of windowed Fourier transform or wavelet transforms — in order to lo-

calize energy peaks in both time and frequency. If a coincident burst is found, it is reconstructed and the source and parameters of the signal are defined, either using Bayesian methods or through comparison with different models [39, 40].

2 Data Analysis Methods

While most of the work in this thesis is focused on the application of data analysis techniques, it is important to have at least a general idea of the theory behind these methods. This section gives a brief introduction to the data analysis techniques adopted in the toy model, in the hope that even a reader unfamiliar with the topics might understand why and how they were used.

2.1 Bayesian Method

Bayesian methods are an alternative to frequentist methods to approach data analysis. The foundation of Bayesian statistics is Bayes' rule, which states that, given a data set y and model parameters θ, the conditional probability of θ given the data set y (i.e. the posterior, $p(\theta|y)$) is proportional to the product of the conditional probability of y given θ (i.e. the likelihood of the dataset $p(y|\theta)$) and the probability of the particular model parameter values (i.e. the prior, $p(\theta)$). The proportionality factor is the inverse of the probability of getting the dataset y (i.e. $p(y)$) [42]. In equation form this can be written as

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} \propto p(y|\theta)p(\theta) \tag{22}$$

While in a frequentist approach the parameters of a model are considered to be fixed and the data to be a random variable, in a Bayesian approach the parameters are treated as a random variable, with a distribution function defined by the prior, and the observed data are treated as a fixed set. If one already has a knowledge of how the prior should look like, they can set it accordingly (e.g. in the case of gravitational waves, one already knows that the instrument is sensitive only to a defined range of frequencies, so the prior for the frequency could be restricted to that range). If, on the other hand, one knows nothing about the parameter space, they can choose to use an uninformative prior; that is they can define the prior as a function that does not restrict the parameters to only a region of the parameter space.
The likelihood $p(y|\theta)$ is determined from the available data and is often the model that one wants to test. At this point one needs to find a way to sample the parameter space so that a posterior can be defined: Markov chain Monte Carlo (MCMC) is a family of methods that lend themselves extremely well to this endeavor.

2.1.1 Markov Chain Monte Carlo

In general, Markov chain Monte Carlo (MCMC) methods are algorithms that can approximate integrals numerically using randomly generated samples.
They work because given any integral

$$\int_a^b h(x)dx \tag{23}$$

one can always see the integrand function $h(x)$ as the product of some other function $f(x)$ (a likelihood, in Bayesian statistics terms) and a probability density function $p(x)$ (a prior for the parameters on which the likelihood depends). The integral can therefore be rewritten as

$$\int_a^b f(x)p(x)dx = E_{p(x)}[f(x)] \tag{24}$$

Given a sufficient amount of random values for x with distribution $p(x)$, the value of the initial integral can then be simply approximated as the mean value of the function $f(x)$ calculated over all the samples drawn. This is, in short, the Monte Carlo method of integration. A crucial point is that, given likelihood and prior, one also needs a "good" sample — i.e., a sample that represents the prior well enough to get the desired level of accuracy — on which to evaluate the integral [43].

Markov chains are a standard way of constructing a "good" sample, mainly for two reasons:

- the probability of transitioning to a certain point depends only on the state of the system at the point right before it;
- if the chain is irreducible (i.e. one can always go from any state to any other state) and aperiodic (i.e. the number of steps required to move between any two states is not required to be a multiple of some integer), it will, at some point, reach a stationary distribution — that is, a condition in which the probabilities of being in any particular state is independent of the initial condition [43].

Over the years countless algorithms have been developed to construct a Markov chain [44, 45], and the cost, efficiency, and availability of computers have grown, making the construction of a good sample much easier.

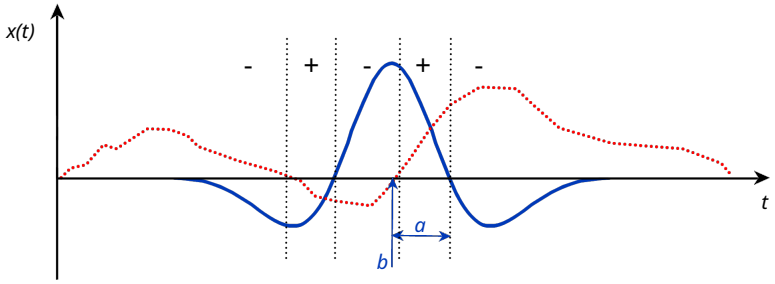
Together, Markov chains and Monte Carlo integration have become MCMC, a now ubiquitous tool (some would say an overused one) in scientific research, finding their way in parameter estimation programs of any kind [46].

2.2 Wavelets

It is beyond the scope of this thesis to give a detailed mathematical description of wavelets and wavelet transform techniques. For a much more in depth discussion of the mathematics of wavelets and their application in different contexts, see [47] and [29], respectively.

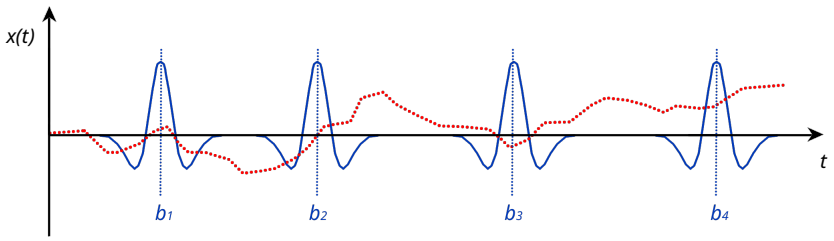
The idea of wavelet transform was developed independently by many authors starting from the late 1960s, but it was about fifteen years later that the field started to gain traction [29, 47], as evidenced by the fact that the term wavelet itself was coined in France in 1982 [47, 48]. Today, wavelet analysis is a well established method of signal processing in various fields [29], from medicine [49] to geology [50] to the topic of this dissertation — gravitational wave signal reconstruction [30, 40].

In order to have an intuitive grasp on what a wavelet transform is, one can think of a wavelet as an object that is wave-like and localized, and of a wavelet transform as a process that compares a wavelet to a signal: if the two match, then a high transform value is obtained; vice versa if the two do not match, a low transform value is obtained [29]. This process is then iterated shifting the wavelet along the time axis[7] and scaling the wavelet (i.e., squeezing it or stretching it while keeping the subtended area constant): this results in a matrix of coefficients that contain all the information of the original signal but presented in a different format [29]. A qualitative representation of this process is shown in figure 4.



(a) The wavelet at scale a is centred around the time b and compared to the signal. The regions where signal and wavelet have the same sign give a positive contribution, those in which the signs are opposite the contribution is negative.

[7] Here, and throughout this whole section, I assume that the signal one is analyzing is defined as the value of some variable with respect to time. If that is not the case, the general method of analysis remains the same.



(b) The process is repeated for different b values - i.e. for different times.

Figure 4: Simplified representation of the wavelet transform process. The wavelet is drawn in blue while the signal in red dotted.

While these ideas are fairly simple, their mathematical formalization is a bit more complex.
A wavelet, mathematically speaking, is any function $\psi(t)$ that satisfies the following three criteria:

- $\psi(t)$ has finite energy, where the energy is defined as $E = \int_{-\infty}^{\infty} |\psi(t)|^2 dt$
- if $\hat{\psi}(\nu)$ is the Fourier transform of $\psi(t)$, it must be true that $C_g < \infty$, where $C_g = \int_0^{\infty} \frac{|\hat{\psi}(\nu)|^2}{\nu} d\nu$
- if $\psi(t)$ is a complex function, then its Fourier transform must be real and must vanish for all negative frequencies.

From a single wavelet one can derive infinitely many other wavelets by simply including in the definition a dilation parameter a and a location parameter b, so that from $\psi(t)$ one gets an infinite number of $\psi\left(\frac{t-b}{a}\right)$. In literature one finds a plethora of different wavelets: while all of them have different properties and are better suited for different applications; the process of wavelet transform stays the same whichever is used.
Given a wavelet $\psi(t)$ and a continuous signal $x(t)$ the wavelet transform of x with respect to ψ is defined as:

$$T(a,b) = w(a) \int_{-\infty}^{\infty} x(t) \psi^* \left(\frac{t-b}{a} \right) dt \tag{25}$$

where $w(a)$ is a weight. Someone familiar with Fourier transforms might have already noticed that, apart from the weighting function, this is nothing but a convolution of the original signal and the wavelet. This particular definition of wavelet transform considers both a continuous wavelet and a continuous signal;

therefore is aptly named continuous wavelet transform (CWT).
In order to recover the signal from the wavelet transform one uses the inverse wavelet transform; which is defined as:

$$x(t) = \frac{1}{C_g} \int_{-\infty}^{+\infty} \int_{0}^{+\infty} T(a,b)\psi_{a,b}(t) \frac{da\, db}{a^2} \tag{26}$$

More often than not, though, one deals discrete quantities: in these cases the CWT cannot be used, but must be replaced with the so-called discrete wavelet transform (DWT).
The DWT uses a discrete set of dilation and location parameters, so that a continuous wavelet

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}} \psi \left(\frac{t-b}{a} \right) \tag{27}$$

becomes

$$\psi_{m,n}(t) = \frac{1}{\sqrt{a_0^m}} \psi \left(\frac{t - n b_0 a_0^m}{a_0^m} \right) \tag{28}$$

where m, n are integers that control scale and translation, while a_0, b_0 are fixed parameters. It must be that $a_0 > 1$ and $b_0 > 0$. If $a_0 = 2$ and $b_0 = 1$ one gets a log_2 scaling for both scale and translation: this is called a dyadic grid. Discrete dyadic grids can be chosen to be orthonormal, i.e. such that

$$\int_{\infty}^{\infty} \psi_{m,n}(t)\psi_{m',n'}(t)dt = \begin{cases} 1 \text{ if } m = m', n = n' \\ 0 \text{ otherwise} \end{cases} \tag{29}$$

This means that the information stored in one wavelet coefficient is not repeated in any other. With this discretization, the DWT is defined as

$$T_{m,n} = \int_{-\infty}^{\infty} x(t)\psi_{m,n}(t)dt \tag{30}$$

It is important to note that, for continuous signals, the DWT is an integral - not a sum: this makes it different from a simple discretization of the CWT.
In order to treat discrete signals one has to introduce more properties of dyadic grids: for these bases one can always define a scaling function ϕ which has the same form of the wavelet. The convolution of scaling function and signal gives an approximation coefficient

$$S_{m,n} = \int_{-\infty}^{\infty} x(t)\phi_{m,n}(t)dt \tag{31}$$

which is the weighted average of the signal (multiplied by $2^{m/2}$). The approximation coefficients at a given scale, as well as their sum over different ns, approximate

the continuous signal $x(t)$. In particular, one can write a continuous signal x(t) as:

$$x(t) = \sum_{n=-\infty}^{\infty} S_{m_0,n}\phi_{m_0,n}(t) + \sum_{m=-\infty}^{m_0} T_{m,n}\psi_{m,n}(t) = x_{m_0}(t) + \sum_{m=-\infty}^{m_0} d_m(t) \tag{32}$$

One can rewrite the scaling function $\phi(t)$ in terms of contracted and shifted version of itself as

$$\phi(t) = \sum_k c_k \phi(2t-k) \tag{33}$$

For wavelets with compact support with a finite number N_k of scaling coefficients, this makes it possible to rewrite the wavelet as

$$\psi(t) = \sum_k^{N_k-1} (-1)^k c_{N_k-1-k}\phi(2t-k) = \sum_k^{N_k-1} b_k \phi(2t-k) \tag{34}$$

From these equation one can see that scaling functions and wavelets at one scale can be derived using those at the next smaller scale. In the same way, one can derive approximation and wavelets coefficients at one scale from those at smaller ones.
When dealing with a discrete signal, this is treated as the signal approximation coefficients (equation 2.2) at scale $m = 0$: from it one can get, one by one, all other relevant detail coefficients.
The inverse transform for the DWT uses the wavelet coefficients at higer scales to define those at smaller ones; when one reaches the smallest scales one can find the approximation coefficients: those are exactly the discrete signal.

Wavelet analysis is particularly useful when one has to work with noisy signals, as it offers different ways to decrease the effect of noise.
For one, wavelet transform can be used to implement a pass-band filter of sorts: as lower scales (i.e. smaller values of the scale parameter a) correspond to higher frequencies, and vice-versa; setting to zero the contributions for all scales lower than a certain value of a effectively acts as a low-pass filter. Obviously, one could similarly construct a high-pass filter [29].
Secondly, one can deal with noise by "thresholding" the wavelet transform; that is by setting to zero or reducing by some (arbitrary) fraction the wavelet coefficients that are less than some (arbitrary) threshold λ. This way of reducing noise is interesting in that it does not blindly target all contributions at high frequency, but only those that are less relevant, which means that the high frequency part of the signal can be kept [29].
As wavelets are localized in both frequency and time, one can also visualize how a

signal is distributed in a frequency-time (or, equivalently, a scale-time) plot. This is useful to visualize the amount of information that is lost in the thresholding process and can be used as a guide in choosing the best wavelet-threshold combination for a given signal.

2.3 Tests for Comparing Reconstructions

A problem that arises from the fact that there is more than one way to reconstruct a signal is that, while the true signal is the same, the results from the different methods may differ. As a consequence, it is vital to find ways to compare different reconstructions. Comparison and consistency tests enable one to ascertain the limits of each method and to choose the most appropriate approach depending on their goals.

The simplest test to check how good a reconstruction is is to look at the distribution of the residuals. As a first approximation, one can think the detector noise to be Gaussian [16, 51]: if a good reconstruction is subtracted from the segment of data what is left should follow a Gaussian distribution. Vice-versa, if the reconstruction is bad either some of the signal is still in the data or some of the noise is in the reconstruction: in both cases the residuals will deviate from a Gaussian distribution.
However, real detector noise is not perfectly Gaussian; therefore a slight variation from a Normal distribution for the residuals may be due to the noise itself, not to the quality of the reconstruction.

A better way to compare reconstructions is to look at the match between the reconstruction and the original signal. Intuitively, the match is a number between -1 and 1 that quantifies the similarity between two signals: a match of 1 would indicate that the two signals compared are identical, a match of -1 would mean that they are exactly in antiphase, and a match of 0 would mean that there is no relation whatsoever. Mathematically the match calculated in the time domain at a point in time t is defined as:

$$m_{a,b}(t) = \frac{\int_{-\infty}^{\infty} a(\tau)b(t-\tau)d\tau}{\|a\|\|b\|} \tag{35}$$

integrating over the duration of the signal this becomes:

$$m(a,b) = \frac{a \cdot b}{\|a\|\|b\|} \tag{36}$$

The latter definition is the one used in the toy model.
Obviously, the higher the match between the whitened signal and the reconstruction, the better the reconstruction is.
To test which method of reconstruction is best in general, though, it is not wise to trust one single reconstruction, as the particular realisation of noise could by chance favour the worse method. Instead it is better to look at the distribution of the matches calculated on many reconstructions of the same signal but with different realisation of noise. Of course, this is much more straightforward when working with simulated signals, but a similar approach could be taken even with real signal by reinjecting the reconstructions into different - simulated or real - segments of noise.
Even the match distribution may not be enough to distinguish between two very similar reconstructions. A possible workaround to make the differences more noticeable is to compare the two reconstructions not to the signal but to a third reconstruction. In the case of this toy model the third reconstruction used as a comparison is an unmodeled reconstruction obtained through DWT of the original signal. The idea is that, even though the unmodeled reconstruction is not as "good" as the modeled ones, it should share more characteristics with the right model than with the wrong one. Therefore the match between unmodeled reconstruction and modeled reconstruction could give an indication of the best model to use.

3 Differences between modeled and unmodeled Reconstructions

3.1 Noise Spectrum

Noise models for gravitational waves interferometers were studied long before any such instruments were actually built [16]. As such, the general shape of the noise spectrum and the effects that limit sensitivity at different frequencies are quite well understood. However, detectors are also affected by transients of non-Gaussian noise, commonly referred to as glitches: these are less well-known.
Glitches can be caused by almost any form of disturbance: some are caused by instrumental noise coupling into the gravitational wave strain channel, some are caused by light scattering off of dust particles, and for some no-one knows why they are there [51, 52]. The usual way to treat glitches is not to use the data stretches affected by them. In some cases - notably in the case of GW170817[8] [53], shown in figure 5 — the signal is "cleaned" of the glitch, but this is not always

[8]This was the first ever detection of a gravitational wave produced by a BNS.

possible.

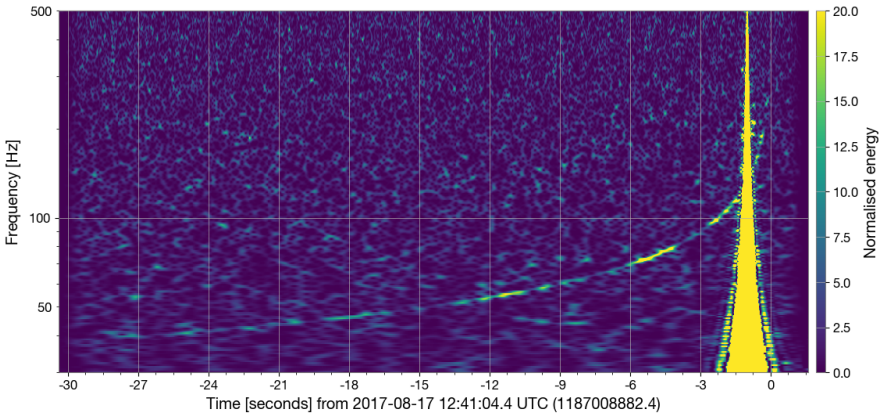


Figure 5: Q-transform of the whitened data strain from LIGO-Livingston around GW170817: a loud instrumental glitch covers part of the long BNS chirp. This plot was made using data available through GWOSC [24].

Given all the problems that glitches bring along, considering that even real pipelines often simply discard glitchy data, and to start with the simplest possible case, glitches were not considered in the toy model developed for this dissertation.

The raw data straight from the detectors are next to useless in terms of signal reconstruction, as noise covers most of the information. In order to be able to analyze the signal, the raw data must first be whitened and de-noised. In principle, after these processes, the stretch of data should be equivalent to the sum of a white Gaussian noise background and the signal. To mimic this situation in the toy model, at first, a clean "signal" — i.e. a sine-Gaussian waveform — was injected into white Gaussian noise. This procedure, though, left out an important step of the real data analysis process, that is the whitening. To account for this, too, the toy model was modified so that it could also inject the sine-Gaussian burst in a stretch of simulated detector background noise. To simulate this noise, the program follows a relatively simple algorithm described by Timmer and König [54]. In order, one must:

- define the target amplitude[9] spectral density (ASD);
- for each Fourier frequency draw two Gaussian distributed random numbers and multiply them by half of the ASD value at that frequency; these will be the real and complex part of the Fourier transform of the noise;

[9] or power spectral density, since one is the square of the other

- for an even number of data points, draw just one number for the Nyquist frequency;
- for a real-valued time series, make the Fourier components of the negative frequencies the complex conjugates of the Fourier components of the respective positive frequencies;
- inverse-Fourier transform the simulated data.

The reference ASDs used are the mean ASDs measured for LIGO-Hanford, LIGO-Livingston, and Virgo during o3 [17], which are plotted in figure 2.

3.2 Whitening of the Signal

The first step in signal analysis is to whiten the stretch of data. In the simplest of terms this means renormalizing the noise so that it has the same maximum amplitude at all frequencies. In practice this is done by dividing the stretch of data by the ASD of the noise. In order to do this one has to measure the ASD: in theory, this could be done just once, but in real life situations it is often preferred to repeat the ASD measurement for each signal, using a stretch of empty noise right before the detection, in order to account for the non-stationarity of the noise [31].
In the toy model, though, there are no variations in the noise ASD, so it needs to be defined just once. In particular, the noise ASD is needed in the process of noise simulation (described above in section 3.1), so it is defined once there and then used each subsequent time it is needed.
The whitening process obviously affects the amplitude of the signal. In particular, I found out the hard way that if one were to inject a signal with $h_{rss} \sim 1$ into a stretch of detector noise with amplitudes of order 10^{-21}; the resulting white signal would have oscillations of order 10^{-21}. While this seems obvious in hindsight, I had not thought of it from the start. In order to be able to set in the initialization file the h_{rss} or amplitude of the whitened signal, when simulated detector noise is used the input value is multiplied by a factor of 10^{-22} for the LIGO detectors and of 10^{21} for Virgo. These do not give exactly the input h_{rss} or amplitude for the white signal, but they get close enough without much computations. Moreover, even the library sine-Gaussian function specifically states that the actual h_{rss} of the signal might not be the one requested in input, so this value was already out of the complete control of the user.
If Gaussian white noise is used, no whitening is needed, so this step is skipped altogether.

3.3 Wavelet and Threshold Influence

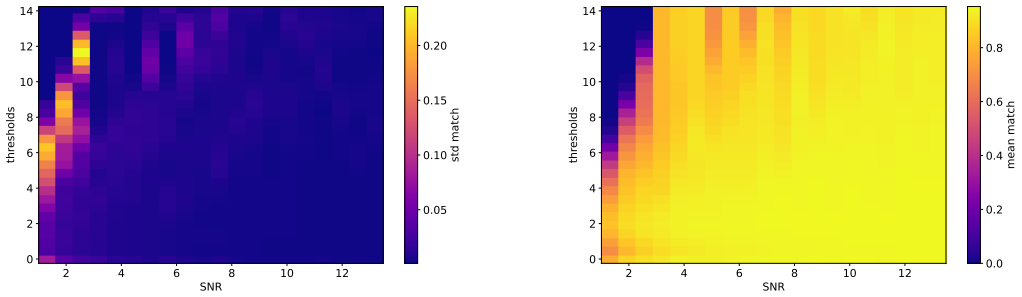
A key step in the unmodeled reconstruction algorithm is to threshold the DWT, that is, to set to zero all the coefficients lower than a set level in order to lower the noise components. There is a level of arbitrariness in this process, as both the choice of wavelet and the choice of threshold influence the result.
In order to limit this problem, I wrote a program to see, with a fixed wavelet, how the choice of threshold influenced the reconstruction at different SNRs. I then repeated this procedure with a number of wavelets, to see if changing that made an impact on the resulting reconstruction.
The program described here can be found in the file fvst.py in appendix A: it needs the same files that the toy model needs, as well as the same packages, and they are all listed in the same appendix.
The code evaluates the mean match and the match standard deviation on 1000 reconstructions of the same signals for each SNR and each threshold, then plots the results. The SNRs used range from around 1 to around 14[10], which is the same range used in the toy model described in section 3.4. The thresholds used in this test are the universal threshold, which in the plots is identified with index 0 and is defined as

$$\lambda = \frac{\sqrt{[2ln(N)]}\text{MAD}}{0.6745} \tag{37}$$

as suggested in The Illustrated Wavelet Transform Handbook [29], and the following set levels: 0.54, 1.08, 1.61, 2.15, 2.69, 3.23, 3.77, 4.31, 4.85, 5.38, 5.92, 6.46, and 7.00[11]. To choose the range of thresholds to test; I first did one run over a range I had reason to assume was too big, to see where it was that match and standard deviation were sure to be bad — that is, very low for the former and very high for the latter. The results of this first run are shown in figure 6, and it is clear that there really is no point in considering a threshold over 7.

[10] the SNR range changes slightly because what actually gets set is the h_{rss} or the amplitude of the signal, both of which are correlated with the SNR but not directly.

[11] these values were chosen by defining the threshold array as np.linspace[0,7,14]. One would think this would give a neat array with not too many decimal digits, but somehow it is not the case.



(a) (b)

Figure 6: Plots for the match standard deviation (6a) and mean (6b) evaluated with thresholds ranging from 0.5 to 14. The 0 row is evaluated using the universal threshold (37). In this case a db4 wavelet was used. The range of reasonable thresholds does not seem to change with the choice of wavelet, nor with the frequency of the signal.

3.4 Toy Model Description

This section describes in detail the code I wrote and used to produce the results discussed in section 3.5. The code itself can be found in appendix A and is written in Python. While most of the packages used in the program are standard (e.g. NumPy [55], matplotlib [56]), others are less common. To save anyone who might want to run this code the trouble of reading all the imports to find out which packages they are missing, a complete list of all the packages used can also be found in appendix A.

It is worth highlighting once more that in the context of this toy model, the word signal refers to a sine-Gaussian burst, not to a chirp. While this is clearly a simplification, this choice made it much easier to control the parameters of the signal (which are amplitude and frequency of the sine function, standard deviation of the envelope, peak time, phase and optionally a mean background noise level parameter), while keeping some of the problems one encounters in the reconstruction of a chirp — mainly the fact that the tails of the signal have amplitudes low enough to be covered by or confused with the noise background. An example of sine-Gaussian signal is shown in figure 7

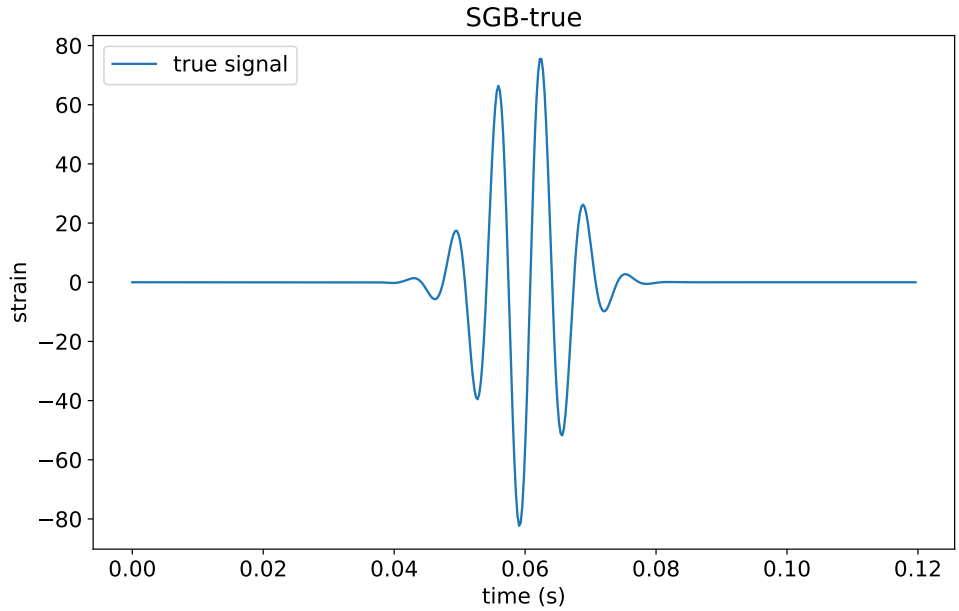


Figure 7: An example of sine-Gaussian burst. This particular signal is comparable in duration and peak strain with the chirp in 3b. The sine-Gaussian lacks a real tail at the start, but overall the shapes are similar.

In order to get started I was given a draft notebook by my supervisor. It quickly became apparent that keeping the program in notebook form was not ideal, and everything was rewritten as a simple python file. This change made it also possible to run the program on Bora; the new cluster of the Department of Physics.
To make the code easier to read and a bit more organized, I also decided to divide it into different files:

- toymod_func.py
- toymod_sig.py
- toymod_init.py
- toymod_cl_05.py

The first two files are used to define functions: toymod_func.py contains a variety of functions (from the definition of SNR to the instructions to plot and delete graphs), while toymod_sig.py specifically contains functions related to the generation of noise and signals.

All user initialized variables are defined in toymod_init.py: this means that in a perfect situation, one would only need to edit this file (in which I tried my best to comment everything) in order to adapt the program to their needs.

The last of these files, toymodel_cl_05.py, contains the main body of the code and is the one that gets called in order to run the program (from command line input "python3 toymodel_cl_05.py").
This file is quite long and can itself be thought as comprised of multiple sections and subsections:

- Proof of concept
  - signal generation
  - unmodeled reconstruction
  - modeled reconstruction with correct model
  - modeled reconstruction with wrong model
- Systematics
  - signals generation
  - unmodeled and modeled reconstructions
  - match distribution calculation
  - reconstructions comparison
  - SNR dependency

The program is somewhat heavy to run and requires a lot of memory, therefore a lot of the steps that are not strictly necessary - e.g. checks of MCMC convergence or plot of the signal shape - can be skipped by setting to False or zero the dedicated variables in the initialization file.

3.4.1 Proof of Concept

The first part of the program works on a single user-chosen signal that is then reconstructed three ways: first using a wavelet-transform and then by fitting the signal via Markov chain Monte Carlo with two different models, one known to be right and one similar but known to be wrong.
The three different reconstruction results are then compared in order to understand how similar or different they are, and to confirm that the unmodeled reconstruction gives a sound result.

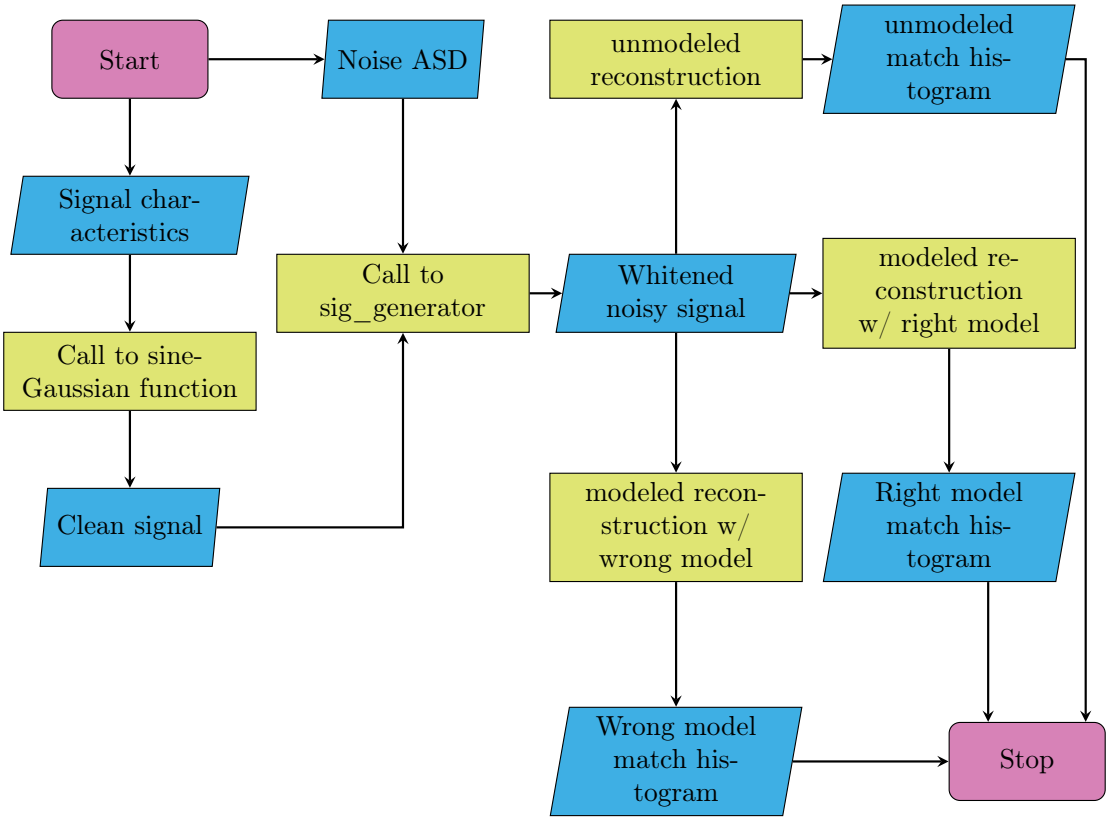


Figure 8: Flow chart of the proof of concept part of the program.

The signal to be reconstructed is generated — with the specifications given in the initialization file by the user — either using the sine-Gaussian burst function from the library LALSimulation [57] or using a sine-Gaussian function that I wrote. The main difference between the two is that the library function takes as input the root-sum-square strain, which is defined as

$$h_{\text{rss}} = \sqrt{\sum (h_+^2 + h_\times^2)\Delta t} \tag{38}$$

where h_+ is the plus polarization time series and $h_\times$ is the cross polarization time series; while the function I wrote takes the sine amplitude in input.
Once the clean signal has been defined, noise is added. At first, Gaussian white noise was used: this choice was made to simplify the signal processing, as it meant that there was no need for further whitening of the signal. At a later stage I implemented the option to add noise with a specific ASD; in particular the ASDs currently available are those measured during the o3 run for LIGO-Hanford, LIGO-Livingston, and Virgo. In order to generate the non-white Gaussian noise I followed the process described in section 3.1; in order to whiten the resulting data, the one

described in section 3.2. Once a white noisy signal is obtained, it is reconstructed.

In the draft I was given to start, the unmodeled reconstruction function was nothing more than a pass-band filter: this was the first thing that had to be changed in order to make the model more efficient.
To better the signal reconstruction process and make it effective on a broad range of frequencies, I wrote a function that, given the noisy signal and a user-chosen wavelet, would, in turn:

- wavelet-transform the signal,
- apply a hard threshold,
- inverse transform the resulting array of coefficients.

The output of the function is the unmodeled reconstruction.

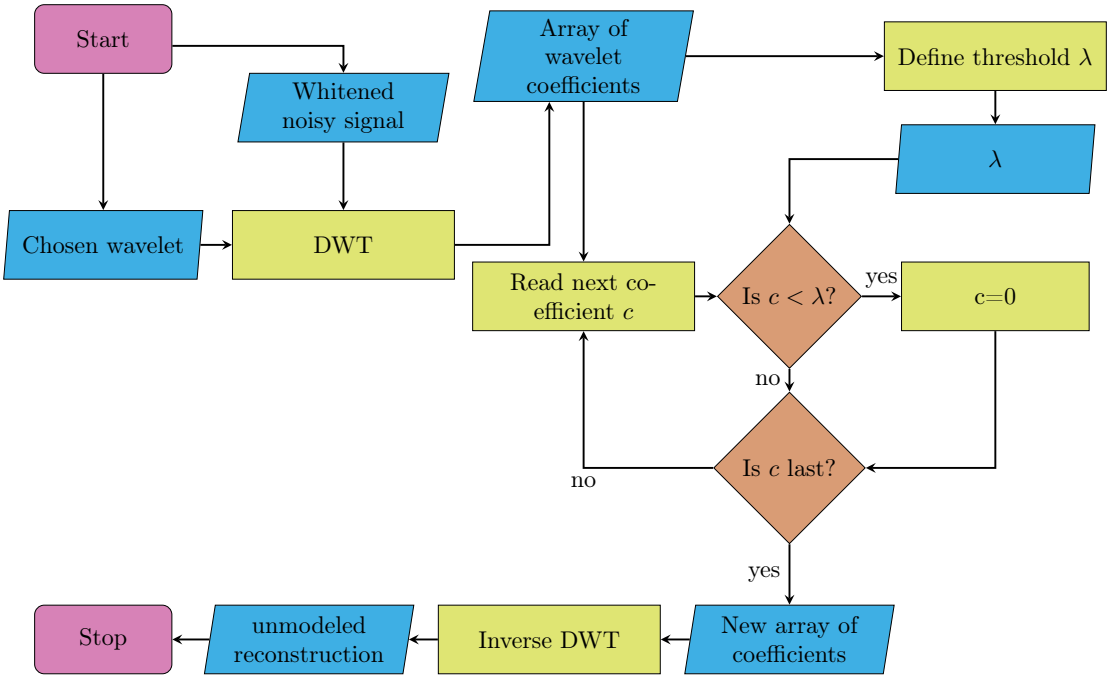


Figure 9: unmodeled reconstruction fuction flow chart

In this program the threshold is always set to 2 (see 3.3, 3.5.3), as the best threshold level at each SNR seemed to always be around this point.
Once obtained, the unmodeled reconstruction is subtracted from the noisy data and the residuals are plotted. The reconstruction then gets reinjected into a number (i.e. *nrepeat*) of different stretches of Gaussian white noise: these new signals

are then reconstructed using the unmodeled reconstruction function[12]. The new reconstructions are then matched to the first one: a match histogram is plotted,and mean, median, mode, and standard deviatin on the match are computed, in order to check if the unmodeled process gives consistent results.
All the wavelets, the functions for wavelet transform and inverse wavelet transform were taken from PyWavelets, a dedicated Python library [58].

The modeled reconstruction uses MCMC, implemented via emcee [59], to fit the signal.
The model functions used in the two modeled reconstructions are set by the user in the initialization file. In principle, one could use any model in the MCMC runs, however since the signals used in the toy model are sine Gaussian bursts, for now there are four options, two that can be considered correct and two that are wrong:

- a sine with a Gaussian envelope;
- a sine with a Gaussian envelope and a mean background noise level parameter;
- a sine with a Cauchy envelope;
- a sine with a Cauchy envelope and a mean background noise level parameter.

The choice of using a Cauchy envelope instead of a Gaussian one as the wrong model was made because the main difference between the two functions lies in the amplitude of their tails, which mirrors what happens with the models for real CBC signals.
The MCMC runs are divided into two, a shorter burn-in phase and a longer proper run phase. The first of the two is initialized using a parameter array defined by the user in the initialization file, while the latter is initialized using the most probable parameters found in the burn-in phase.
After each burn-in and each run the program can plot the residuals; after the MCMC runs there is the option to check if the chains converged, and to create a corner plot for all the parameters.
The modeled reconstructions are then compared to the unmodeled one: to do this, a number (i.e. *nrepeats*) samples are chosen randomly from the MCMC run, and their matches with the unmodeled reconstruction are computed and plotted.

The match between each reconstruction and the true signal is also computed.

[12]In principle one might argue that this procedure is correct only if the noise was Gaussian and white to begin with, while if the signal was whitened in order to be reconstructed one should de-whiten the reconstruction, add noise, whiten the result and then reconstruct again: while maybe more correct, this method seemed to be excessively involved for a first approximation.

3.4.2 Systematics

The systematics section is the heaviest part of the code and was also the least developed in the initial program draft; as such it was the one that required the most work.

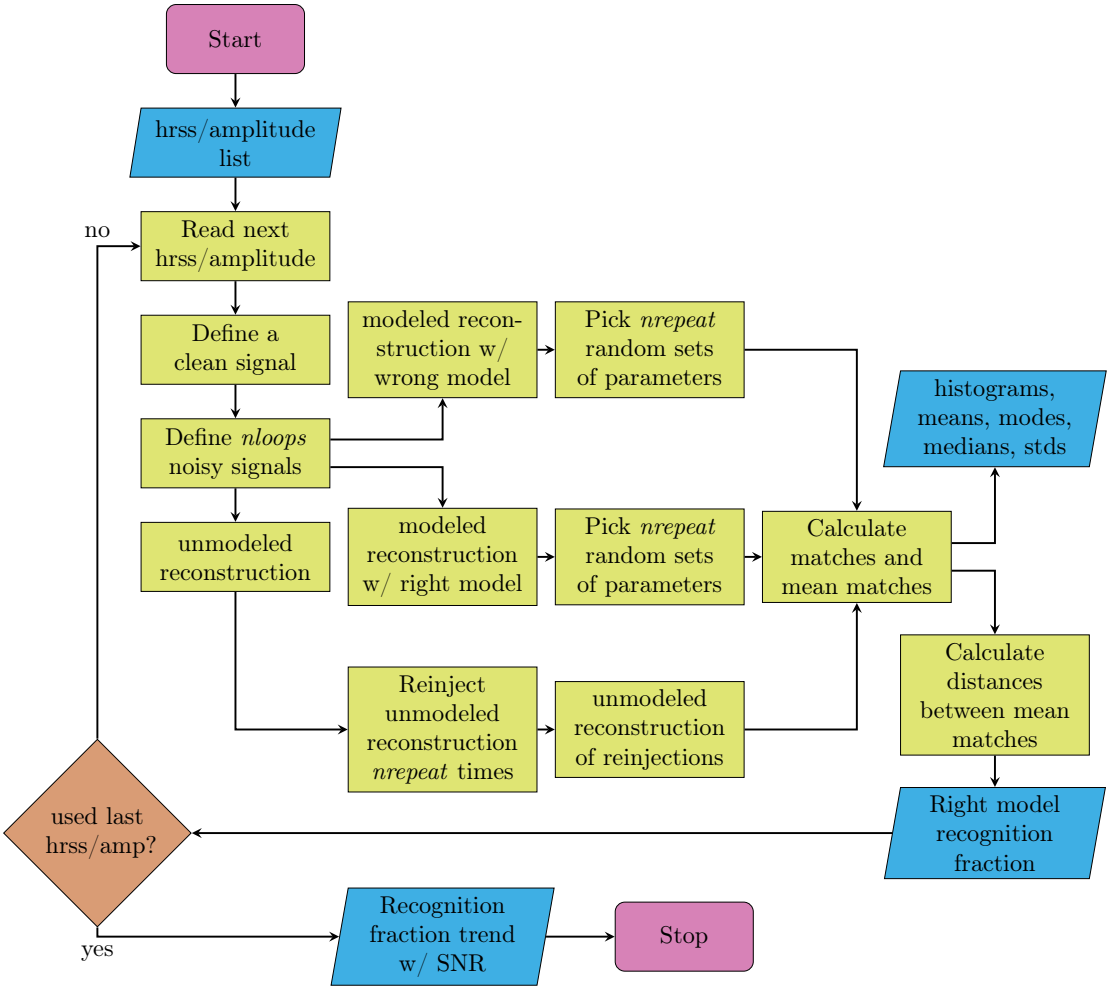


Figure 10: Systematics section flow chart

As shown in the flow chart in figure 10, this section starts with a loop over a number (i.e. *nSNRs*) of signal-to-noise ratios[13]. In each repetition of this cycle the program generates one clean signal and from it derives a number (i.e. *nloops*) of

[13]Technically, the loop is over h_{rss} or amplitude values, but since both of those are proportional to the SNR one can think in function of one or the others without much difference in this particular instance.

noisy signals, each one with a different realization of background noise. The noisy signals are then reconstructed in the same ways described in the previous section, with the difference that - for the sake of runtime - the MCMC runs can be set to be shorter.
Each unmodeled reconstruction is then injected in a number (i.e. *nrepeat*) of new stretches of white noise: these new signals are once again reconstructed, and the results are matched with the first reconstruction - that in a way can be thought as the "model signal".
For each modeled reconstruction nrepeat samples form the MCMC run are picked randomly and matched with the onsource unmodeled reconstruction.
The user can define *nSNRs*, *nloops*, and *nrepeat* in the initialization file, though suggested values for each are listed in the comments to the file.

The matches calculated for each reconstructed signal are then plotted to get a match distribution for each method of reconstruction. These distributions are then characterised by looking at their mean, median, mode, and standard deviation.
The match distribution - and the statistical values used to characterize it - is compiled for each one of the *nloops* signals defined at the start. To understand and visualize how each method of reconstruction fares statistically, the program plots histograms of the means, medians, modes, and standard deviations of each distribution.
The program also creates a series of auxiliary plots, such as a mean match vs standard deviation plot for each method of reconstruction, a mean match unmodeled vs mean match modeled plot, ect... These plots highlight possible correlations between the measured quantities, but can also be helpful as a tool to decide if some of the reconstructions are not to be considered - e.g. in account of a very low mean match or of a very high standard deviation.

Once all of the statistical quantities of interest are calculated the program has all that it needs to define what is maybe the most important parameter for the purposes of this dissertation, that is the distance between the mean match of the unmodeled reconstruction and the mean matches of the modeled reconstructions. These distances are defined as:

$$\frac{\text{mean}_{\text{mod}} - \text{mean}_{\text{unmod}}}{\sqrt{\sigma^2_{\text{mod}} + \sigma^2_{\text{unmod}}}} \tag{39}$$

In order to visualise the calculated distances, a true distance vs wrong distance plot is constructed. This plot is divided into distinct sections by four lines, two vertical at $td = \pm 2\sigma$ and two horizontal at $wd = \pm 2\sigma$. This 2σ threshold is arbitrarily set. Depending on the section in which each point is, it falls into a different category:

1. if both true and wrong distance are below the set threshold, then the test is inconclusive - it finds both models equally good at reconstructing the specific signal;

2. if the true distance is below threshold and the wrong distance is over the threshold, then the test has been successful in identifying the correct model as the better one for the reconstruction of the signal;

3. if the true distance is over threshold and the wrong distance is below the threshold, then the test has failed, as it finds the wrong model as the better one for the reconstruction of the signal;

4. lastly, if both distances are over threshold, the test has failed again - as this result would indicate that even the correct model is unfit to reconstruct the signal properly.

The result for each single point is relevant, but what is more important is that the test be accurate statistically. In practice this means that the metrics used to gauge how good the test is are the fraction of cases that fall into each category described above.

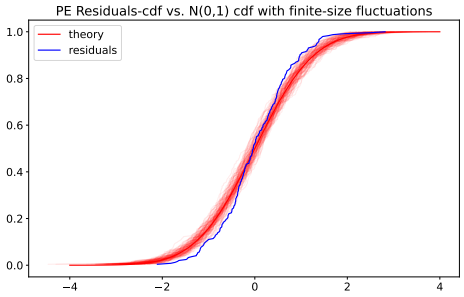
As one might expect, this test fares fairly differently with signals of different SNRs. For this reason the systematics section is looped over a number of SNR values, and at the end the program creates a graph with the various result fractions plotted against the SNR at which they have been obtained.

3.5 Computational Tests and Results

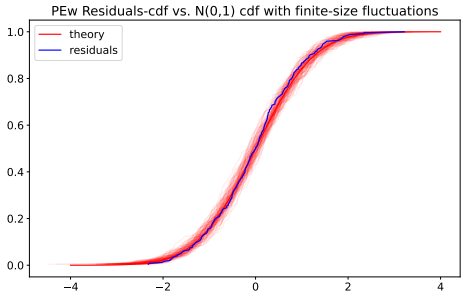
As stated in 3.4, all tests were carried out using a sine-Gaussian burst as signal. For most of the results presented in this section, the signals were defined as follows. The h_{rss} and amplitude values of the signal were chosen so that the optimal SNRs for the signals analyzed in the systematics section would cover the range between 1 and 14. Depending on the kind of noise the signals were injected into the exact values could vary, but that range is always covered. The frequency f_0 of the sine function was set to 400 Hz, the sampling frequency f_s to 4096 Hz. The *duration* parameter, which is used to determine the standard deviation of the Gaussian envelope, was set to 0.003. When the library function for the sine-Gaussian burst was used, the eccentricity and phase parameters were set to 0.
If any of the parameters were changed from those specified in this paragraph, it will be noted.

3.5.1 Residuals Distribution

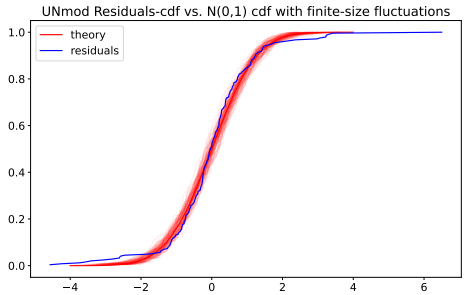
For low SNRs empirical cumulative distribution function (eCDF) plots for the residuals are not particularly useful. In this condition, looking at the eCDF for the residuals from a reconstruction gives little to no information on the goodness of the reconstruction itself; as in most cases the the eCDF is compatible with the one for a Gaussian distribution. If there are deviation from the theoretical curve, they are usually so small that they could easily be attributed to a particularly unlikely realisation of noise. Considering the plot of the residuals themselves - either directly in a strain vs time plot or in histogram form - gives the same information in a much clearer fashion.
For higher SNR values, even the eCDF plot starts to pick up on the deviation from a Normal distribution; but here, too, other forms of visualisation should be preferred.
The plots in figure 11 and 12 illustrate this point.



(a) Residuals for the modeled reconstruction with the correct model.

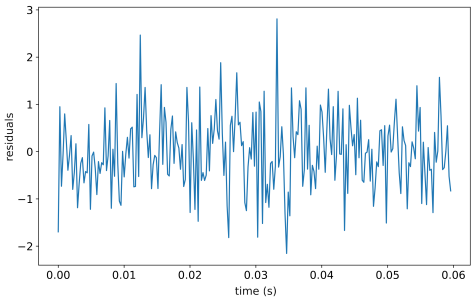


(b) Residuals for the modeled reconstruction with the wrong model.

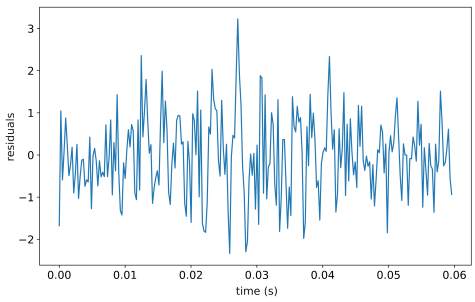


(c) Residuals for the unmodeled reconstruction

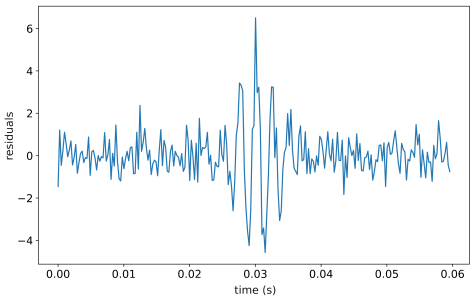
Figure 11: These are the eCDF plots for the three different reconstructions of a signal with SNR= 6.54. The plots for the modeled reconstructions, 11a and 11b, are perfectly compatible with the theoretical shape, while the one for the unmodeled reconstruction 11c shows some deviation in the tails.



(a) Residuals for the modeled reconstruction with the correct model.



(b) Residuals for the modeled reconstruction with the wrong model.



(c) Residuals for the unmodeled reconstruction

Figure 12: These are the same residuals used to derive the plots in 11: here it is much easier to spot that the unmodeled reconstruction has left part of the signal in the data.

In all cases, though, it is difficult to identify a best model between two similar ones from the residuals distribution alone.
All these behaviors were observed both in the case the signal was injected in a background of white Gaussian noise and in the case it was injected in a stretch of simulated detector noise.

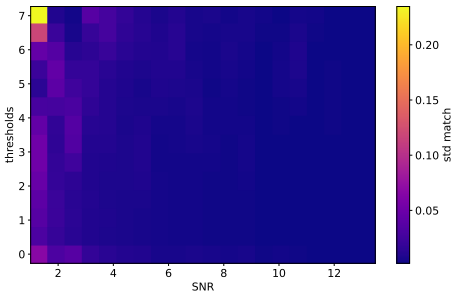
3.5.2 Unmodeled Reconstruction Behavior

At the risk of sounding unscientific, the first remark I want to make regarding the unmodeled reconstruction method is how convenient it is: compared to its modeled counterparts it was much faster and - while admittedly it never reaches their level of accuracy or precision - it is much harder to get an extremely bad result, even when choosing threshold and wavelet almost blindly.

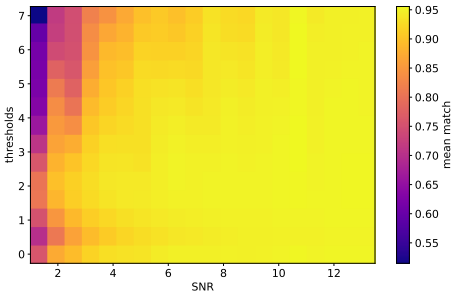
In terms of match with the injected signal, the unmodeled reconstruction fared better than its modeled counterparts for very low SNRs, but worse for intermediate and high SNRs. The mean match between the unmodeled reconstruction and the injected signal ranged from around 0.8 at SNR$\sim$ 1 to 0.94 for SNR$\sim$ 5. The mean match stays consistently around 0.94 for all higher values of SNR tested. These mean match values indicate that the unmodeled reconstruction, while "good", has some issues, which can be identified by looking at the residuals. From the plots in figure 12c it is clear that the unmodeled algorithm fails to perfectly reconstruct the peak of the signal. This behavior is observed with all the wavelets tested and with both a fixed threshold and the universal threshold. The results do not change with the change of noise ASD either.

3.5.3 Threshold Influence

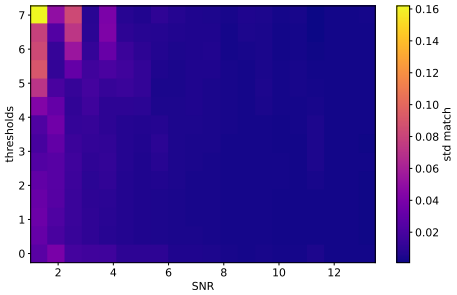
After running the code described in section 3.3 with multiple sets of parameters, a few trends became evident. First, at fixed wavelet and fixed signal frequency, it seems that for low SNRs (less than 4) there is indeed a best threshold value that provides a balance between a high mean match and a low standard deviation. This threshold oscillates around 2 - being slightly more or less depending on the wavelet used and the frequency of the signal. In all the situations considered, the best frequency was never below one or over three.
The universal threshold gives a consistently decent performance, independently of wavelet and frequency, however, it does seem that 2 is never far off. Moreover, the universal threshold has the disadvantage of having to be computed each time, which is the reason why, in later versions of the toy model, the threshold was switched to 2.
For high SNRs, the choice of threshold becomes less and less relevant, with both mean match and standard deviation tending to the same values for almost all thresholds in the considered range.



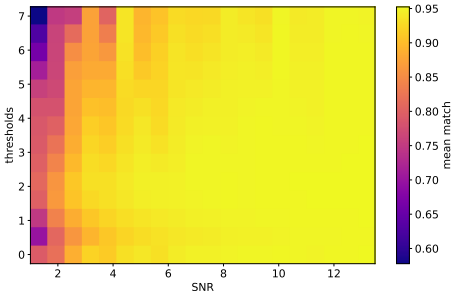
(a) Match standard deviation with db4 wavelet



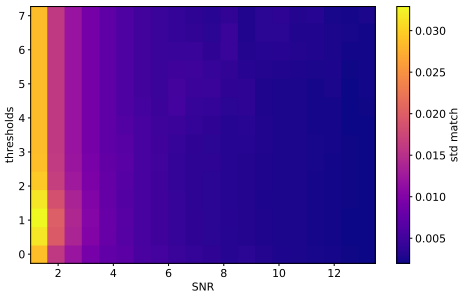
(b) Match mean with db4 wavelet



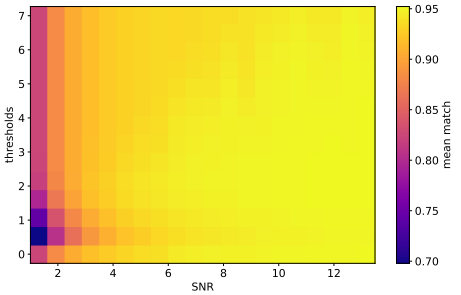
(c) Match standard deviation with db10 wavelet



(d) Match mean with db10 wavelet



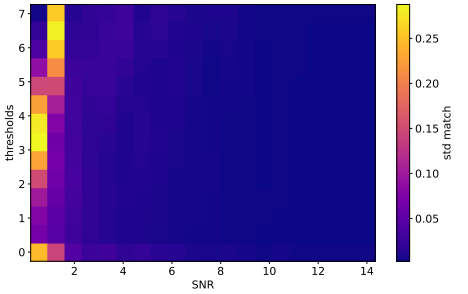
(e) Match standard deviation with db20 wavelet



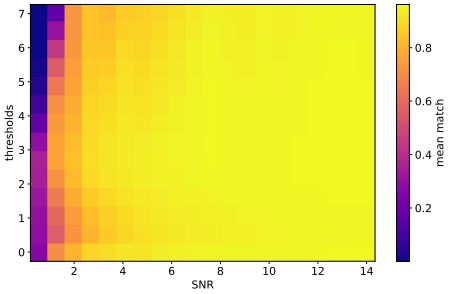
(f) Match mean with db20 wavelet

Figure 13: Plots evaluated for a signal with $f_0 = 400$ Hz with thresholds ranging from 0.5 to 7. The 0 row is evaluated using the universal threshold.

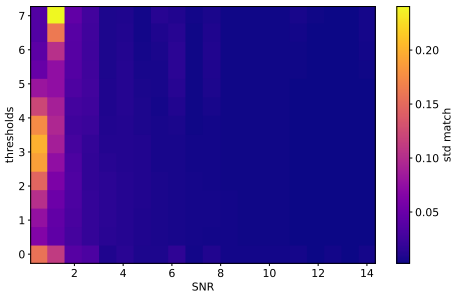
The choice of wavelet seems to be more relevant for signals with lower frequencies. Indeed, while for all frequencies higher-order Daubechies wavelets fare better than lower-order ones, the difference in performance is much greater for signals of lower frequencies. As one can see from the plots in figure 13, for a signal with frequency $f_0 = 400$ Hz the difference in performance between a Daubechies10 wavelet and a Daubechies20 wavelet lies only in the standard deviation of the mean match, while the mean match itself does not change. If instead one looks at the plots in figure 14, it is clear that for a signal of frequency $f_0 = 350$ Hz the situation differs, as both the standard deviation and the mean change with the choice of wavelet: the first gets lower and the latter grows. A similar trend was evident for a signal with frequency $f_0 = 375$ Hz, too.



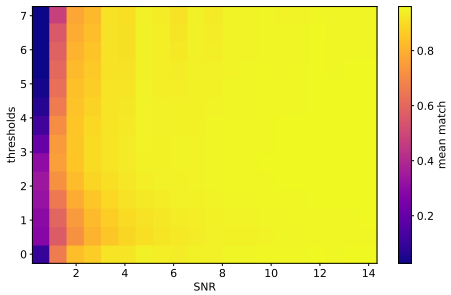
(a) Match standard deviation with db4 wavelet



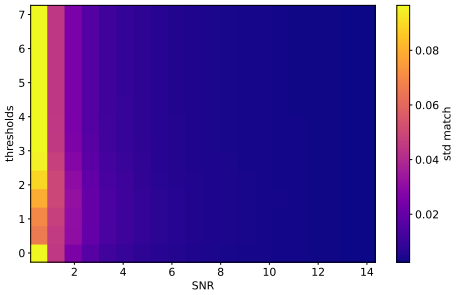
(b) Match mean with db4 wavelet



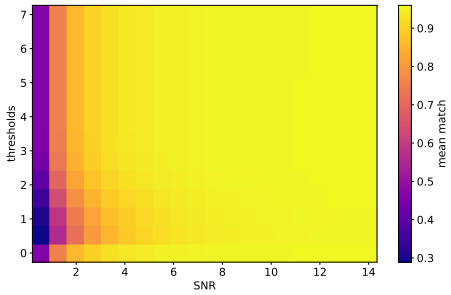
(c) Match standard deviation with db10 wavelet



(d) Match mean with db10 wavelet



(e) Match standard deviation with db20 wavelet



(f) Match mean with db20 wavelet

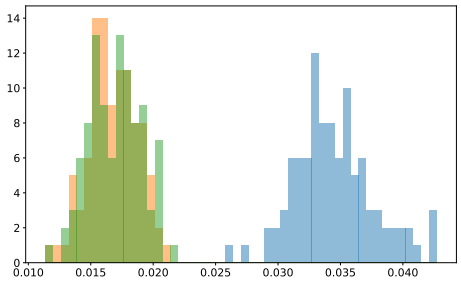
Figure 14: Plots evaluated for a signal with $f_0 = 350$ Hz with thresholds ranging from 0.5 to 7. The 0 row is evaluated using the universal threshold.

In all cases, though, both mean and standard deviation reach an extremal value that cannot be improved by using a higher-order wavelet.

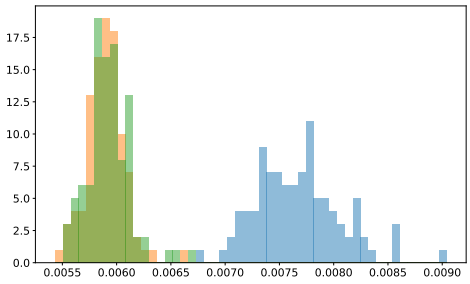
3.5.4 Modeled Reconstructions Behaviors

As expected, the accuracy of the modeled reconstructions is better than the one of the unmodeled reconstruction, both if white Gaussian noise or detector noise is used. The precision of the modeled reconstruction was worse than the one of the unmodeled reconstruction for very low SNR values (SNR$\sim$ 1), but in all other conditions both the right model and the wrong model gave more precise results. In particular, the mean match between the injected signal and the modeled reconstruction using the correct model went from around 0.6 for SNR$\sim$ 1 to 0.99 for

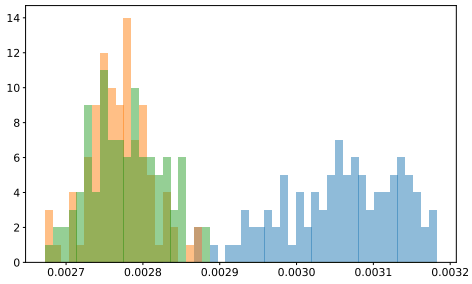
SNR$\gtrsim$ 5. For the reconstruction using the wrong model the situation is similar, with mean match values ranging from 0.6 to 0.98 in the same conditions. This similarity makes it difficult to discern which of the two models is best.



(a) SNR= 2.05



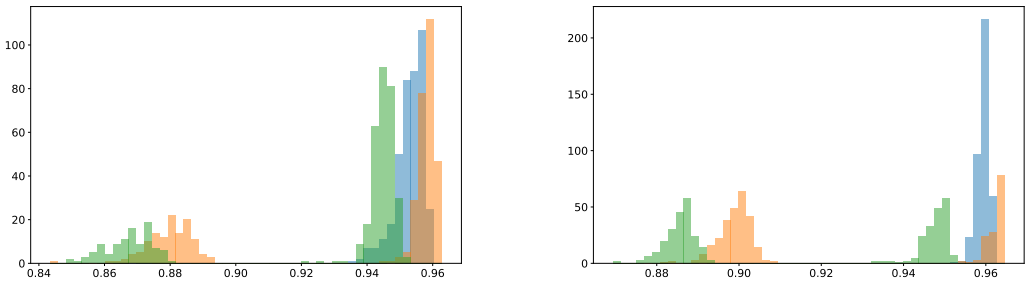
(b) SNR= 6.53



(c) SNR= 14.0

Figure 15: Examples of standard deviation distribution at different SNRs. In this case the background was white Gaussian noise. The right model is orange, the wrong model is green, and the unmodeled is blue.

One drawback of modeled methods is that - at least in this form - they are extremely sensitive to the starting point: even the right model if initialized incorrectly often finds what seems to be a local extrema for the parameters. This "solution" obviously has a worse match than the real one.



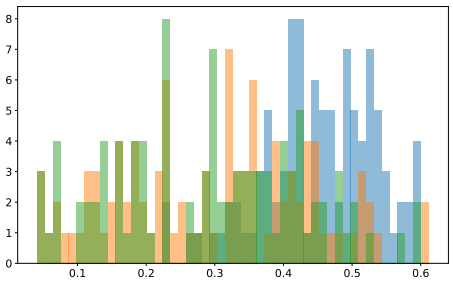
(a) SNR= 7.17 (b) SNR= 14.3

Figure 16: Examples of cases where a bad initialization of the amplitude caused the MCMC algorithms to find more than one peak. The right model is orange, the wrong model is green, and the unmodeled is blue. The x-axis is the mean match between the unmodeled onsource reconstruction and the MCMC results or offsource reinjections reconstructions.

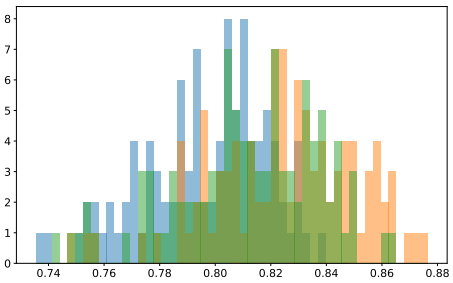
To test if the problem that caused this double peak was indeed one of bad initialization, I wrote my own sine-Gaussian function, so that I could know in advance the right parameters and set the initial point accordingly: sure enough the peak was just one. In a real-world scenario, though, one does not know the parameters exactly in advance, so this is not really a solution. Another possible solution to this issue could have been to use an optimizer to find a set of parameters to start from. Very early on in the project, though, the parameter estimate was done using an optimizer which had to be changed on account of the fact that it, too, required extremely specific parameters to find a minimum, so this did not look like a promising path. Different random walkers (in the form of a different "moves" parameter for the emcee sampler) and a much longer burn-in were tried, too, but without much improvement.

3.5.5 Model Distinction

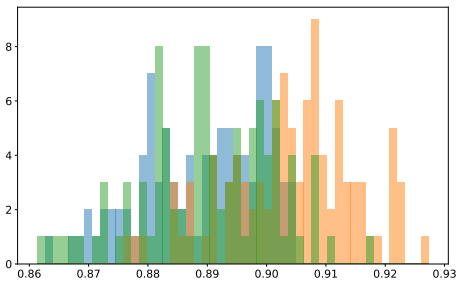
As stated in section 3.5.4, the mean match between signal and reconstruction is very similar for both modeled methods. In order to distinguish which is best between the two, their match with the unmodeled reconstruction was used as a comparison. As expected, the results of this test are highly dependent on the SNR of the signal.
For very low SNRs (below 4) all reconstructions have equivalently "bad" mean match distributions, all peaking below 0.9 as one can see in figure 17.



(a) SNR= 0.56



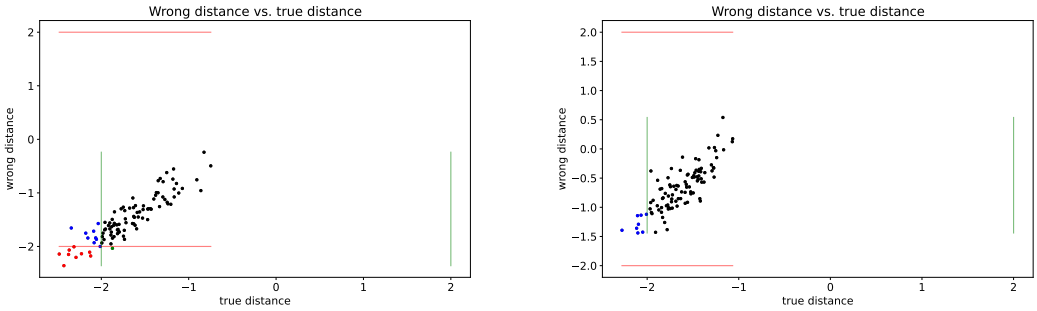
(b) SNR= 2.05



(c) SNR= 3.54

Figure 17: Example of mean match histograms for low SNRs. In this case the background was white Gaussian noise: the right model is orange, the wrong model is green, and the unmodeled is blue. As expected for a signal with SNR$<$ 1, the reconstruction for 17a is extremely bad with all methods.

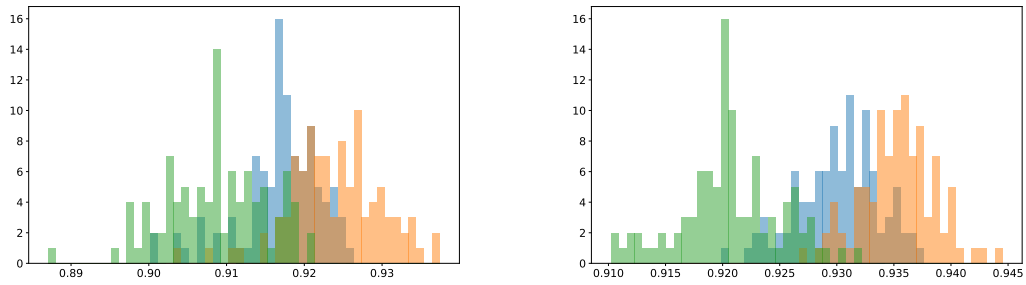
In this regime, the test is almost always inconclusive. When the result is not inconclusive, it is wrong — that is, it recognizes the wrong model as the better one to describe the signal. These results, shown in figures 23 and 18, are not wholly unexpected, as it is precisely in these cases that the noise could easily be misconstrued as a tail of the signal.



(a) SNR= 2.05 (b) SNR= 3.54

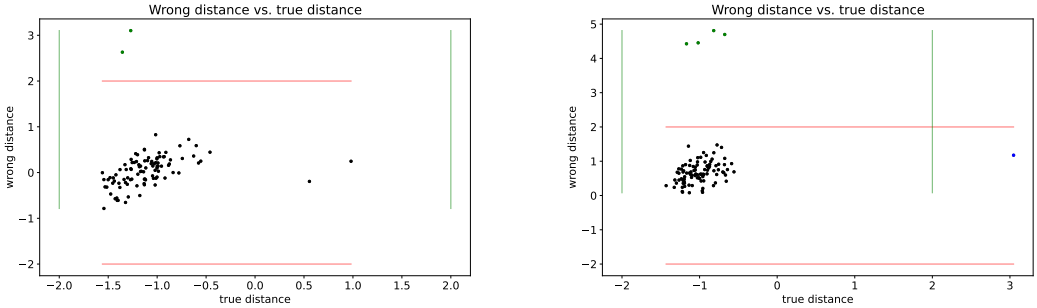
Figure 18: Wrong distance vs true distance scatter plots for low SNR values: in these instances the great majority of points falls in the 2σ range for both modeled reconstructions, but some do fall into the 2σ range only for the reconstruction obtained with the wrong model. The red and green lines mark the 2σ limits. Black points are cases in which the test is inconclusive, blue points represent cases in which the result is flipped (i.e. the wrong model is identified as the best one to represent the signal), red dots are cases in which no good model was found, and green dots are true alarms.

For SNRs$>$ 4 the mean match is higher for all methods of reconstruction, and the distributions for the two modeled reconstructions stop overlapping completely as can be seen in figure 19. It is at this point that there is a chance for the test to be successful. However, for intermediate SNRs - between 4 and 7 - the test is still inconclusive, as shown by 23 and 20.



(a) SNR= 5.04 (b) SNR= 6.53

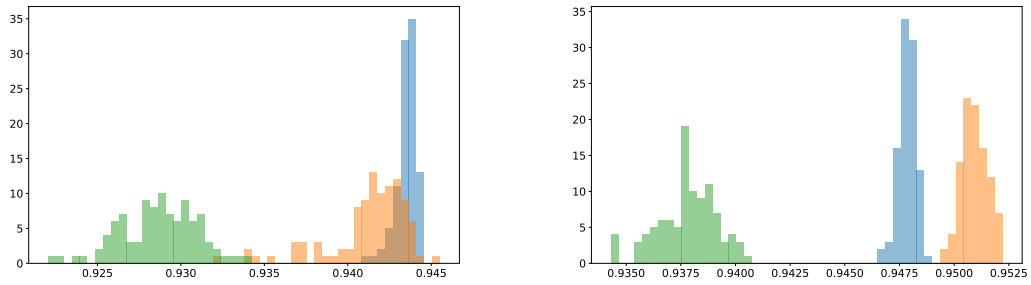
Figure 19: Example of mean match histograms for intermediate SNRs. In this case the background was white Gaussian noise: the right model is orange, the wrong model is green, and the unmodeled is blue. Around these SNR values the distributions begin to distance themselves.



(a) SNR= 5.04 (b) SNR= 6.53

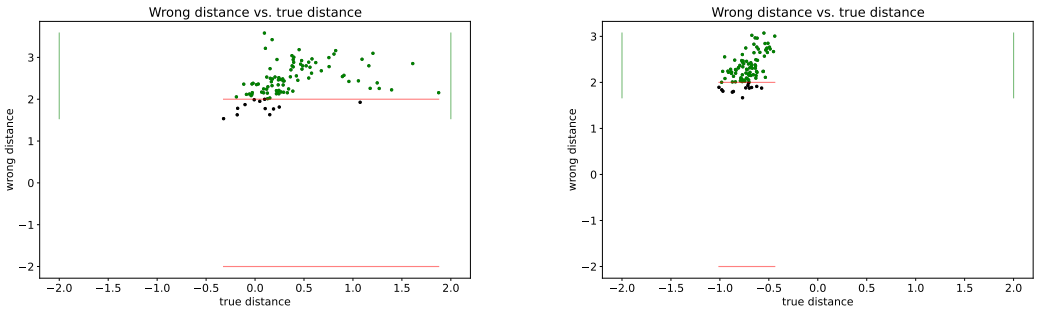
Figure 20: Wrong distance vs true distance scatter plots for intermediate SNR values: in these instances the great majority of points falls in the 2σ range for both modeled reconstructions, but some start to fall in the true alarm quadrant. The red and green lines mark the 2σ limits. Black points are cases in which the test is inconclusive, blue points represent cases in which the result is flipped (i.e. the wrong model is identified as the best one to represent the signal), red dots are cases in which no good model was found, and green dots are true alarms.

Only with SNRs higher than 7 the fraction of right identification of the good model grows. Plots representative of this situation are shown in 21 and 22.



(a) SNR= 8.02 (b) SNR= 11.0

Figure 21: Example of mean match histograms for high SNRs. In this case the background was white Gaussian noise: the right model is orange, the wrong model is green, and the unmodeled is blue.

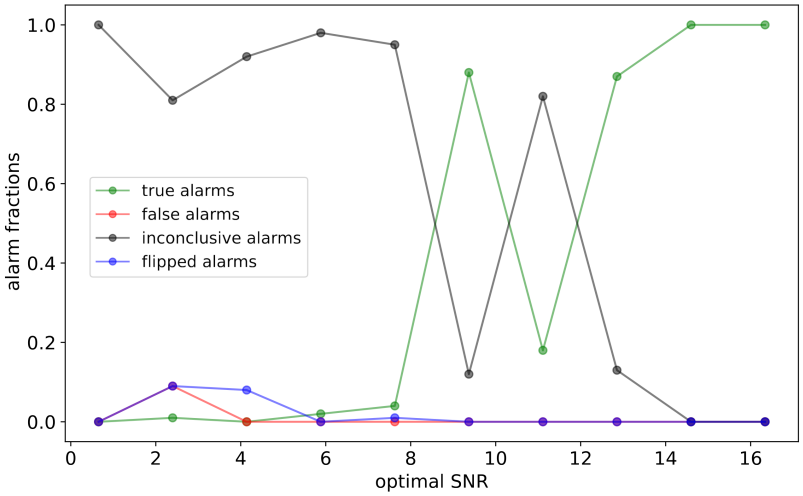


(a) SNR= 8.02 (b) SNR= 11.0

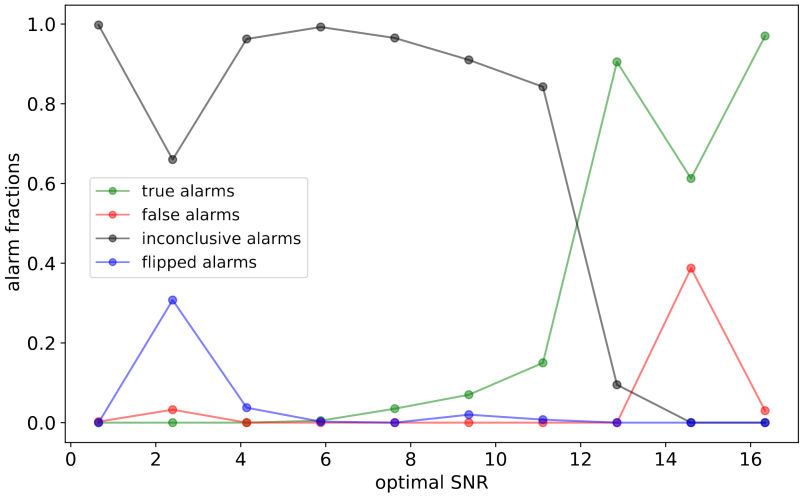
Figure 22: Wrong distance vs true distance scatter plots for high SNR values: in these instances the great majority of points falls in the 2σ range only for the reconstruction obtained with the correct model. The red and green lines mark the 2σ limits. Black points are cases in which the test is inconclusive, blue points represent cases in which the result is flipped (i.e. the wrong model is identified as the best one to represent the signal), red dots are cases in which no good model was found, and green dots are true alarms.

The growth of the fraction of true alarms, however, is not steady. I suppose this non smooth trend to be caused by using too small a sample, as when using 400 loops instead of 100 the curve smooths out, as can be seen by comparing figures 23a and 23b. Running the program with so many samples was computationally intensive (i.e. it took about double the time and four times the memory compared to running it with the smaller sample size) and all the other results seemed to not

change this drastically with sample size, therefore only a small number of tests were run this way.



(a) In this case the test was repeated 100 times for each SNR: the growth of the true alarm fraction is not smooth.



(b) In this case the test was repeated 400 times for each SNR: the growth of the true alarm fraction is slower but smoother.

Figure 23: Test results at different SNRs. The true alarm fraction is plotted in green, the fraction of inconclusive cases in grey, the fraction of flipped cases in blue, and the fraction of false alarms in red.

4 Discussion and Conclusions

In conclusion, the unmodeled reconstruction using DWT gives good results both when the signal is injected in white Gaussian noise and when it is injected in simulated detector noise and then whitened. The accuracy of the unmodeled reconstruction seems to be limited, and a fraction of the signal is always left in the noise. This effect is much more noticeable at higher SNR. The modeled reconstruction, too, gives good results with any noise tested — when given a good starting point for the MCMC. This highlights the need to find an analytical or empirical way to correlate the model parameters to a measurable quantity. Without such a relation, there is a chance of discarding the right model because of a bad reconstruction.

The unmodeled reconstruction with Daubechies wavelets is more accurate with higher-order wavelets than with lower-order ones. The threshold used in the unmodeled reconstruction does not matter much for signal with optimal SNR > 4, while there is clearly a best choice for lower SNR values. Although the universal threshold is never the worst performing threshold, it seems that for sine-Gaussian signals, the best choice was a fixed threshold of value $\lambda \sim 2$ Testing the two modeled reconstructions against the unmodeled one gives results that heavily depend on the SNR of the signal.

For low SNR values, any reconstruction method seems to be equivalent, as they all have mean match distributions that peak around the same value: in these conditions the test is not reliable. For intermediate SNR values, the mean matches of each reconstruction start to differ but the unmodeled reconstruction is still in range of 2σ from both the right and wrong modeled reconstruction mean match distributions. For SNR over 6, the precision of the unmodeled and of the correctly modeled reconstructions grows, and only the mean matches of the right model are inside the 2σ range from the unmodelled mean match.

Future developments for the project are manifold. First, one would need to input real signals (which is relatively simple) and develop a way to use real templates for the modeled reconstruction (which is very much less simple). Secondly, the model should be able to handle glitches either by identifying that there is a glitch and discarding the data or by identifying that there is a glitch and cleaning the data. Neither option is trivial to implement; mostly because from the point of view of the unmodeled algorithm, there really is no difference between a glitch and a signal. Comparing the data from more than one detector could help in differentiating some of the glitches from the signals, but not all.

An interesting future perspective would be to study the influence of below-threshold glitches: it is highly probable that detector noise is polluted with glitches that are too quiet to be seen, but loud enough to disturb the noise distribution. Since one controls the noise in the toy model, one could introduce a series of below-

threshold glitches and study how the whitening and the reconstruction are affected by them.

On a more technical note, the code would be easier to manage if it were more time and memory efficient: both could be achieved by "translating" the whole thing in a more low-level programming language; such as C or C++. This would also make it much easier to work with LALSuite if one did want to make the toy model and the library compatible. Another way to make the code more efficient would be to use an optimizer to get a parameter estimate, however this would not work well with the sine-Gaussian function from LALSuite without a good way to initialize the amplitude of the signal.

A Appendix

In order to run the code described in section 3.4, one needs a Python 3 environment with the following packages installed:

- NumPy [55];
- matplotlib [56];
- SciPy [60];
- emcee [59];
- PyWavelets [58];
- corner [61];
- LALSuite [62];
- multiprocessing;
- itertools;
- statsmodels [63].

Moreover, one needs to save all four files described in section 3.4 in a single directory, and in that same directory have a subdirectory called *curves_Jan_2020* containing the files *o3_h1.txt*, *o3_l1.txt*, and *o3_v1.txt*: these files describe the asd curves for LIGO-Hanford, LIGO-Livingston, and Virgo. The detectors ASDs can be found through [17].

```
#set these variables as 0/1 or False/True
test_char=1 #if true runs cells that are not strictly needed
corn_plot=1 #if true plots corner plots for the mcmc runs
traces=1 #if true plots traces for the mcmc runs
res_plot=1 #if true plots the residuals plots

#set these variables
save_path='./'#set folder in which to save output

ch_wlt='db20' # defines which wavelets to use, pywt.wavelist(kind='discrete')
    shows all available wavelets
n_th=4 #number of threads used in parallel processes
nburn=5000# # defines length of burn in in mcmc runs
niter=10000# defines length of mcmc runs
nburn_s=1500 # defines length of burn in in short mcmc runs
niter_s=2500 # defines length ofshort  mcmc runs
nrepeat=10000 # defines number of sample/draws in systematic checks -- 10000
nloops = 100 # defines number of repetitions of main loop -- 100

nwalkers=128
```

```

nSNRs=2 #10 defines how many amplitudes/hrss are checked in the systematic section

#choose which function to use to create the clean signal:
#-->'sg': sine-gaussian burst from lalsimulation
#-->'sg_amp': sine-gaussian burst written in house by yours truly
sig_gen='sg_amp' #'sg' 'sg_amp'

fs=4096. # sampling frequency
delta_t=1./fs #sampling interval
f0 = 400. # SGB central frequency
duration=0.003 # SGB duration

#if using sg these are hrss values, if using sg_amp these are amplitudes: define
    them accordingly

hrss = 1.25#e-22 #1

hrss_low=0.01#e-22 #1e-22 #0
hrss_high=0.25#e-22  #10e-22 #2

#define these if you're using sg as signal generator -- they can be left at 0
ecc=0 # SGB eccentricity
phase=0 # SGB phase

#possible noise ASDs, choose from:
#-->ll:LIGO-Livingston o3 asd
#-->lh:LIGO-Hanford o3 asd
#-->v:Virgo o3 asd
#-->g: white gaussian noise
noise_mod='ll' #'g' 'lh' 'll' 'v'

#right and wrong models to compare, choose:
#-->gb and cb for a gaussian and a cauchy envelope w/ a background level parameter
#-->g and c for a gaussian and a cauchy envelope w/OUT a background level
    parameter
model_func='gb'
model_func_W='cb'

# starting values for mcmc parameters, in order [ amplitude, frequency , sigma ,
    t0 ,  phi , background ]
initial=[1.5,380.,0.05,0.03,0.,0.]
initial_W=[1.5,380.,0.05,0.03,0.,0.]

seed=42 #123

#these are just variables that the program needs, don't worry about them
data=[]
asd_check=0
ASD=0
asd_freq=[]
asd_interpolated=[]
```

Listing 1: toymod_init.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.optimize import curve_fit
from scipy.fft import fft, ifft, irfft, rfft
from scipy.stats import norm, median_abs_deviation
```

```
7  from scipy.interpolate import PchipInterpolator
8  from scipy.signal.windows import tukey
9  import warnings
10 warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
11 import lal
12 import lalsimulation as ls
13 import sys
14 from numpy.random import default_rng
15 rng = default_rng()
16 #ad hoc functions and initialization
17 import toymod_func as tf
18 import toymod_init as tin
19
20 def sine_gaussian(duration, freq, delta_t,amp):
21     """
22     This function defines the x and + polarization strains for a sine-Gaussian
       burst. It mimics what the function ls.SimBurstSineGaussian does, but takes an
       amplitude instead of an hrss as input.
23     """
24     sigma2=duration**2
25     t_peak=duration*10.0
26     t_tot=duration*20.0
27     nsample=int(np.floor(t_tot/delta_t))
28     t_tot=delta_t*nsample
29     samp_times=np.linspace(0,t_tot,nsample)
30     hp=amp*np.sin(freq*2.0*np.pi*samp_times)*np.exp(-(samp_times-t_peak)**2/(2.0*
       sigma2))/(np.sqrt(2.0*sigma2*np.pi))
31     hc=amp*np.cos(freq*2.0*np.pi*samp_times)*np.exp(-(samp_times-t_peak)**2/(2.0*
       sigma2))/(np.sqrt(2.0*sigma2*np.pi))
32     return(hp,hc)
33
34 def sig_arg(gen_func,Q,hrss=None):
35     """
36     This function picks the right arguments for the chosen signal generation
       function and gives them as output in an array
37     """
38     if gen_func=='sg':
39         if hrss==None:
40             args=[Q,tin.f0,tin.hrss,tin.ecc,tin.phase,tin.delta_t]
41         else:
42             args=[Q,tin.f0,hrss,tin.ecc,tin.phase,tin.delta_t]
43     elif gen_func=='sg_amp':
44         if hrss==None:
45             args=[tin.duration,tin.f0,tin.delta_t, tin.hrss]
46         else:
47             args=[tin.duration,tin.f0,tin.delta_t,hrss]
48     else:
49         print('sorry, I am still working on this :(')
50     return args
51
52 def sig_form(gen_func,*args):
53     """
54     given the output of sig_arg as input, calls the chosen ls function and returns
        the clean signal components
55     """
56     if gen_func=='sg':
57         hp,hc=ls.SimBurstSineGaussian(*args)
58         return (hp.data.data,hc.data.data)
59     elif gen_func=='sg_amp':
60         return sine_gaussian(*args)
61     else:
62         print('unknown function')
```

```
63          return (NaN,NaN)
64
65  def sig_check(gen_func,*args):
66      """
67      given the chosen signal form performs some additional checks
68      """
69      if gen_func=='sg':
70          # check duration again (for test purposes)
71          duration=ls.SimBurstSineGaussianDuration(*args[:2])
72          print(duration)
73          # number of samples in each polarization (they should correspond; this
    number is variable and depends on different parameter choices)
74          nplus=len([*args[2]])
75          ncross=len([*args[3]])
76          print(f'Number of samples: {nplus}, {ncross}')
77          if (nplus!=ncross):
78              print("check your signal parameters, something must be wrong...")
79              raise SystemExit
80      else:
81          print('sorry, I am still working on this :(')
82
83  def sig_generator(noise_func,hp,hc, phi):
84      """
85      adds the chosen noise to the clean signal
86      """
87      alpha_p =  np.cos(phi)
88      alpha_c = -np.sin(phi)
89      signal_t = alpha_p*hp+alpha_c*hc
90
91      if noise_func=='g':
92          noise_p=rng.normal(0,1,len(hp))
93          signal0 = signal_t+noise_p # the signal to be analyzed is given by a white
     Gaussian noise background plus a SGB with polarization angle phi
94      else:
95          if tin.asd_check==0:
96              #getting the desired asd curve
97              if noise_func=='lh':
98                  asd_arr=np.loadtxt('./curves_Jan_2020/o3_h1.txt')
99              elif noise_func=='ll':
100                 asd_arr=np.loadtxt('./curves_Jan_2020/o3_l1.txt')
101             elif noise_func=='v':
102                 asd_arr=np.loadtxt('./curves_Jan_2020/o3_v1.txt')
103             else:
104                 print('sorry, I am still working on this :(')
105                 return(0,0)
106             asd_arr=asd_arr.T
107             #padding between 0 and the lowest frequency of the asd
108             low_freq_pad = 1*10**(int(np.ceil(np.log10(asd_arr[1,0])))+1)
109             tin.ASD=[]
110             tin.ASD = np.append([ [0,asd_arr[0,0]-1e-7], [low_freq_pad,
    low_freq_pad] ], asd_arr, axis=1)
111             #new frequencies
112             tin.asd_freqs = np.linspace(0., tin.fs, 64*(len(hp))+1 ) # the choice
    of multiplying by 64 is aribitrary, it gives a good enough interpolation
113             #interpolating the new points
114             pc = PchipInterpolator(tin.ASD[0], tin.ASD[1])
115             tin.asd_interpolated = pc(tin.asd_freqs)
116         #here the noise is generated following the method described in Timmer &
    Konig Astron '95
117         generated_freqs = tin.asd_interpolated * (1+0j) * np.sqrt((64*(len(hp))+1)
    *tin.fs)
118         generated_freqs[0] *= rng.normal(0,1,1)[0]
```

```
        generated_freqs[1:-1] *= (rng.normal(0,1, (64*len(hp))-1 ) + 1j*rng.normal
    (0,1, (64*len(hp))-1) ) #(len(hp)//2)-1
        generated_freqs[-1] *= rng.normal(0,1,1)[0]
        #noise_times = np.linspace(0, len(hp)/tin.fs, len(hp))
        noise_p = irfft(generated_freqs/np.sqrt(2.))
        noise_p[50:50+len(hp)]+=signal_t# from now noise_p is noise+signal
        white = irfft(rfft(noise_p)/tin.asd_interpolated/np.sqrt(tin.fs))
        signal0=white[50:50+len(hp)]
        true_sig_whiten=np.zeros(len(noise_p))
        true_sig_whiten[50:50+len(hp)]+=signal_t
        true_white= irfft(rfft(true_sig_whiten)/tin.asd_interpolated/np.sqrt(tin.
    fs))
        signal_t=true_white[50:50+len(hp)]
    return (signal_t, signal0)

def model_finder(x):
    """
    this function checks which model you have selected in the init file and points
     to the corresponding function
    """
    if x=='g':
        return tf.func
    elif x=='c':
        return tf.func4
    elif x=='gb':
        return tf.func2
    elif x=='cb':
        return tf.func3
    else:
        print('I am sorry, but you seem to have requested a function we still not
    have in stock')
        return 0
```

Listing 2: toymod_sig.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
from numpy.random import default_rng
rng = default_rng()
from scipy.stats import norm, median_abs_deviation
from statsmodels.distributions.empirical_distribution import ECDF
from scipy.interpolate import PchipInterpolator
import emcee
import pywt
import gc
import toymod_init as tin

"""-----------------------------------------------------------------"""
# definition of match function in the time domain
def match(a,b):
    return np.dot(a,b)/(np.linalg.norm(a) * np.linalg.norm(b))

"""-----------------------------------------------------------------"""
# this section contains the analytical formulas for the signals used in the PE
    step

# simple SNG
def func(t, a, f, sigma, t0, phi):
    return a * np.cos(2*np.pi*f*(t-t0)+phi) * np.exp(-(t-t0)**2 /(2*sigma**2))

```

```
# SNG + constant background
def func2(t, a, f, sigma, t0, phi, bkg):
    return a * np.cos(2*np.pi*f*(t-t0)+phi) * np.exp(-(t-t0)**2 /(2*sigma**2)) + bkg

# SNG with Cauchy instead of Gaussian + bkg (this is the "wrong waveform" in the comparison)
def func3(t, a, f, gamma, t0, phi, bkg):
    return a * np.cos(2*np.pi*f*(t-t0)+phi) * gamma**2/(gamma**2 + (t-t0)**2) + bkg

# SNG with Cauchy instead of Gaussian + bkg (this is the "wrong waveform" in the comparison)
def func4(t, a, f, gamma, t0, phi):
    return a * np.cos(2*np.pi*f*(t-t0)+phi) * gamma**2/(gamma**2 + (t-t0)**2)

# definition of amplitude SNR
def SNR(x,nplus):
    return 2.0*np.sqrt(np.dot(x,x)/nplus)
"""----------------------------------------------------------------"""
#these are the functions needed for the UNmodelled reconstruction
def burst_rec(x, th=None):
    if th:
        th_fact=th
    else:
        th_fact=(np.sqrt(2.0*np.log2(len(x)))/0.6745)
    dec=pywt.wavedec(x, tin.ch_wlt,'periodic')
    for i in range(1,len(dec)):
        if i==1:
            if th:
                thresh=th_fact
            else:
                thresh=th_fact*median_abs_deviation(dec[i])
        dec[i]=pywt.threshold(dec[i], thresh, mode='hard', substitute=0)
    res=pywt.waverec(dec, tin.ch_wlt)
    wl_dt=(len(x)*tin.delta_t)/len(res)
    wl_times=np.arange(0,len(res))*wl_dt
    pc = PchipInterpolator(wl_times,res)
    times=np.arange(0,len(x))*tin.delta_t
    res = pc(times)
    return (res)

"""----------------------------------------------------------------"""
#these functions are used for (intrinsically) parallelized calls of the emcee sampler
# definition of unnormalized log-prior
def log_prior(theta):
    a = theta[0]
    f = theta[1]
    sigma = theta[2]
    t0 = theta[3]
    phi = theta[4]
    if len(theta)==6:
        bkg = theta[5]
        if( 0.<a<200. and 100.<f<500. and 0.0025<sigma<0.1 and 0.025<t0<0.04 and -0.5*np.pi<phi<0.5*np.pi  and -.5<=bkg<0.5):#0-100 100-400
            return 0.0
    else:
        if( 0.<a<500. and 100.<f<500. and 0.0025<sigma<0.1 and 0.025<t0<0.04 and -0.5*np.pi<phi<0.5*np.pi):
            return 0.0
    return -np.inf
```

```
81
82 # definition of log-likelihood for the MCMC run
83 def log_likelihood(theta, model_func):#, x, y, yerr , model_func
84     sigma2=tin.data[2]
85     y=tin.data[1]
86     x=tin.data[0]
87     return -0.5 * np.sum((y - model_func(x, *theta)) ** 2 / sigma2 + np.log(sigma2
       ))#model_func[0]
88
89 # definition of log-posterior
90 def log_prob(theta, model_func ):#, x, y, yerr , model_func
91     if np.isfinite( log_prior(theta) ):
92        return log_prior(theta)+log_likelihood(theta, model_func)#, x, y, err, ,
       model_func
93     return -np.inf
94 """----------------------------------------------------------------"""
95 #these functions are needed in order to run mcmc in parallel w/out relying on
       emcee's internal parallelization.
96
97 # NEW and improved definition of Gaussian log-likelihood for the MCMC run
98 def log_likelihood_new(theta, new_data, model_func):#, x, y, yerr
99     #global data
100    sigma2=new_data[2]#data[2]
101    y=new_data[1]
102    x=new_data[0]
103    return -0.5 * np.sum((y - model_func(x, *theta)) ** 2 / sigma2 + np.log(sigma2
       ))
104
105 # definition of log-posterior
106 def log_prob_new(theta, new_data, model_func):#, x, y, yerr
107    if np.isfinite( log_prior(theta) ):
108       return log_prior(theta)+log_likelihood_new(theta, new_data, model_func)#, x
       , y, err
109    return -np.inf
110
111 def mcmc_func(x, model_func, initial_arr):
112    times=tin.data[0]
113    err = 1.
114    new_data = (times,x,err)
115    initial = initial_arr
116    ndim = len(initial)
117    p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-4 * np.
       random.randn(ndim) for i in range(tin.nwalkers)]
118    sampler = emcee.EnsembleSampler(tin.nwalkers, ndim,log_prob_new, moves=emcee.
       moves.StretchMove(), args=(new_data, model_func))#
119    p0, _, _ =sampler.run_mcmc(p0, tin.nburn, progress=False)
120    samples = sampler.flatchain
121    theta_max  = samples[np.argmax(sampler.flatlnprobability)]
122    del samples
123    initial = theta_max
124    p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-5 * np.
       random.randn(ndim) for i in range(tin.nwalkers)]
125    sampler.reset()
126    pos, prob, state = sampler.run_mcmc(p0, tin.niter_s, progress=False)
127    sample_list=sampler.get_chain(flat=True,discard=500)# sampler_list is an ARRAY
        and therefore behaves like an array - NOT like a list. why is it called list?
        I don't know, it wasn't my choice
128    return sample_list
129
130 """----------------------------------------------------------------"""
131 #these are functions that I just like to have so that each plot doesn't take ten
       lines of code each time
```

```
132
133 def graph_init(save_dpi,fig_dpi,font_sz,fig_sz):
134     rcParams["savefig.dpi"] = save_dpi#100
135     rcParams["figure.dpi"] = fig_dpi#100
136     rcParams["font.size"] = font_sz#14
137     rcParams["figure.figsize"] = fig_sz#(10,6)
138
139 def del_graph():
140     #this saves a bit of memory
141     curr_ax= plt.gca()
142     curr_ax.cla()
143     plt.clf()
144     plt.close('all')
145     gc.collect()
146
147 def my_plot(x,y,label_list,alpha, legend=None, save=None, title=None, show=None,
    **kwargs):
148     if len(alpha)!=1:
149         for i in range(0,len(alpha)):
150             if legend:
151                 plt.plot(x,y[i],label=label_list[i], alpha=alpha[i], **kwargs )
152                 c=i
153             else:
154                 plt.plot(x,y[i], alpha=alpha[i],**kwargs)
155                 c=-1
156     else:
157         if legend:
158             c=0
159             plt.plot(x,y,label=label_list[0], alpha=alpha[0],**kwargs)
160         else:
161             c=-1
162             plt.plot(x,y, alpha=alpha[0],**kwargs)
163     plt.xlabel(label_list[c+1])
164     plt.ylabel(label_list[c+2])
165     if legend:
166         plt.legend(loc='upper left')
167     if title:
168         print('plotting',title,'\n')
169         plt.title(title)
170     if save:
171         plt.savefig(tin.save_path+save)
172     if show:
173         plt.show()
174     del_graph()
175
176 def my_hist(x, n_bin, label_list, alpha, legend=None, save=None, title=None, show=
    None,**kwargs):
177     if (title==None and save):
178         title=save
179     if len(alpha)!=1:
180         for i in range(0,len(alpha)):
181             if legend:
182                 plt.hist(x,n_bin,label=label_list[i], alpha=alpha[i],**kwargs)
183                 c=i
184             else:
185                 plt.hist(x,n_bin, alpha=alpha[i],**kwargs)
186                 c=-1
187     else:
188         if legend:
189             c=0
190             plt.hist(x,n_bin,label=label_list[0], alpha=alpha[0],**kwargs)
191         else:
```

```
192             c=-1
193             plt.hist(x,n_bin, alpha=alpha[0],**kwargs)
194
195     plt.xlabel(label_list[c+1])
196     plt.ylabel(label_list[c+2])
197     if legend:
198         plt.legend(loc='upper left')
199     if title:
200         print('plotting',title,'\n')
201         plt.title(title)
202     if save:
203         plt.savefig(tin.save_path+save)
204     if show:
205         plt.show()
206     del_graph()
207
208 def res_plot(smt_res,title):
209     for k in range(100):
210         data_res = np.random.normal(0,1, size=len(smt_res))
211         ecdf = ECDF(data_res)
212         plt.plot(ecdf.x,ecdf.y,color='red',alpha=0.075)
213     xv=np.linspace(-4,4,100)
214     plt.plot(xv,norm.cdf(xv),color='red',label='theory')
215     ecdf = ECDF(smt_res)
216     plt.plot(ecdf.x,ecdf.y,color='blue',label='residuals')
217     plt.legend()
218     plt.title(title)
219     plt.savefig(tin.save_path+title+'.pdf')
220     del_graph()
221
222 #this function creates all the recap plots - it is truly a terrible function and
        should never have been born, but it saves like 50 lines in the main file so it
        's here to stay
223 def sys_plots(mean_burston2bursoff,mean_burston2PEsample,mean_burston2PEsampleW,
        std_burston2bursoff,std_burston2PEsample,std_burston2PEsampleW,
        median_burston2bursoff,median_burston2PEsample,median_burston2PEsampleW,
        mode_burston2bursoff,mode_burston2PEsample,mode_burston2PEsampleW, hrss):
224     mn=np.min(np.asarray([mean_burston2bursoff,mean_burston2PEsample,
        mean_burston2PEsampleW]))
225     mx=np.max(np.asarray([mean_burston2bursoff,mean_burston2PEsample,
        mean_burston2PEsampleW]))
226     #mean histogram
227     plt.hist(mean_burston2bursoff,50,range=(mn,mx),alpha=0.5)
228     plt.hist(mean_burston2PEsample,50,range=(mn,mx),alpha=0.5)
229     plt.hist(mean_burston2PEsampleW,50,range=(mn,mx),alpha=0.5)
230     plt.savefig(tin.save_path+f'systematic-mean-hrss{hrss}.pdf')
231     del_graph()
232     #std histogram
233     mn=np.min(np.asarray([std_burston2bursoff,std_burston2PEsample,
        std_burston2PEsampleW]))
234     mx=np.max(np.asarray([std_burston2bursoff,std_burston2PEsample,
        std_burston2PEsampleW]))
235     plt.hist(std_burston2bursoff,50,range=(mn,mx),alpha=0.5)
236     plt.hist(std_burston2PEsample,50,range=(mn,mx),alpha=0.5)
237     plt.hist(std_burston2PEsampleW,50,range=(mn,mx),alpha=0.5)
238     plt.savefig(tin.save_path+f'systematic-std-hrss{hrss}.pdf')
239     del_graph()
240     #median histogram
241     mn=np.min(np.asarray([median_burston2bursoff,median_burston2PEsample,
        median_burston2PEsampleW]))
242     mx=np.max(np.asarray([median_burston2bursoff,median_burston2PEsample,
        median_burston2PEsampleW]))
```

```
    plt.hist(median_burston2bursoff,50,range=(mn,mx),alpha=0.5)
    plt.hist(median_burston2PEsample,50,range=(mn,mx),alpha=0.5)
    plt.hist(median_burston2PEsampleW,50,range=(mn,mx),alpha=0.5)
    plt.savefig(tin.save_path+f'systematic-median-hrss{hrss}.pdf')
    del_graph()
    #mode histogram
    mn=np.min(np.asarray([mode_burston2bursoff,mode_burston2PEsample,
    mode_burston2PEsampleW]))
    mx=np.max(np.asarray([mode_burston2bursoff,mode_burston2PEsample,
    mode_burston2PEsampleW]))
    plt.hist(mode_burston2bursoff,50,range=(mn,mx),alpha=0.5)
    plt.hist(mode_burston2PEsample,50,range=(mn,mx),alpha=0.5)
    plt.hist(mode_burston2PEsampleW,50,range=(mn,mx),alpha=0.5)
    plt.savefig(tin.save_path+f'systematic-mode-hrss{hrss}.pdf')
    del_graph()
    #
    my_plot(mean_burston2bursoff,std_burston2bursoff,['mean match','std match'],
    alpha=[1],save=f'Std burst vs. mean burst-hrss{hrss}.pdf',marker='.', ls='')

    my_plot(mean_burston2PEsample,std_burston2PEsample,['mean match','std match'],
    alpha=[1],save=f'Std PE vs. mean PE-hrss{hrss}.pdf',marker='.', ls='')

    my_plot(mean_burston2PEsampleW,std_burston2PEsampleW,['mean match','std match'
    ],alpha=[1],save=f'Std PE W vs. mean PE W-hrss{hrss}.pdf',marker='.', ls='')
    #
    my_plot(mean_burston2bursoff,mean_burston2PEsample,['mean match burst','mean
    match PE'],alpha=[1],save=f'Mean PE vs. mean burst-hrss{hrss}.pdf',marker='.',
     ls='')
    #
    my_plot(mean_burston2PEsample,mean_burston2PEsampleW,['mean match PE','mean
    match PE W'],alpha=[1],save=f'Mean PE W vs. mean PE-hrss{hrss}.pdf',marker='.'
    , ls='')

#more recap plots
def trigger(tdist,wdist,hrss, h_count):
    tb=2.0
    wb=2.0
    good=[]
    bad=[]
    inconclusive=[]
    flipped=[]

    for i in range(len(wdist)):
        if np.abs(tdist[i])<tb and np.abs(wdist[i])>wb:
            good.append([tdist[i],wdist[i]])
        elif np.abs(tdist[i])>tb and np.abs(wdist[i])>wb:
            bad.append([tdist[i],wdist[i]])
        elif np.abs(tdist[i])<tb and np.abs(wdist[i])<wb:
            inconclusive.append([tdist[i],wdist[i]])
        elif np.abs(tdist[i])>tb and np.abs(wdist[i])<wb:
            flipped.append([tdist[i],wdist[i]])
    good = np.array(good)
    bad = np.array(bad)
    flipped = np.array(flipped)
    inconclusive = np.array(inconclusive)
    if len(tdist):
        print(f'Fraction of true alarms hrss={hrss:.2}: ',len(good)/len(tdist))
        print(f'Fraction of false alarms hrss={hrss:.2}: ',len(bad)/len(tdist))
        print(f'Fraction of inconclusive cases hrss={hrss:.2}: ',len(inconclusive)
    /len(tdist))
        print(f'Fraction of flipped cases hrss={hrss:.2}: ',len(flipped)/len(tdist
    ))
```

```
        print('\n')
        plt.plot(tdist,wdist,'.',color='gray')
        plt.xlabel('true distance')
        plt.ylabel('wrong distance')
        plt.title('Wrong distance vs. true distance')
        if len(good)!=0:
            plt.plot(good.T[0],good.T[1],'g.')
        if len(bad)!=0:
            plt.plot(bad.T[0],bad.T[1],'r.')
        if len(inconclusive)!=0:
            plt.plot(inconclusive.T[0],inconclusive.T[1],'k.')
        if len(flipped)!=0:
            plt.plot(flipped.T[0],flipped.T[1],'b.')
        xmin=min(tdist)
        xmax=max(tdist)
        ymin=min(wdist)
        ymax=max(wdist)
        plt.plot((xmin,xmax),(wb,wb),color='red',alpha=0.5)
        plt.plot((xmin,xmax),(-wb,-wb),color='red',alpha=0.5)
        plt.plot((tb,tb),(ymin,ymax),color='green',alpha=0.5)
        plt.plot((-tb,-tb),(ymin,ymax),color='green',alpha=0.5)
        plt.savefig(tin.save_path+f'w-vs-t-dist-hrss{h_count}.pdf')
        del_graph()
        return [len(good)/len(tdist),len(bad)/len(tdist),len(inconclusive)/len(
    tdist),len(flipped)/len(tdist)]
    else:
        print('all points were rejected \n')
        return [-1,-1,-1,-1]
"""-----------------------------------------------------------------"""
```

Listing 3: toymod_func.py

```
# library imports
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import default_rng
from scipy import signal
from scipy.optimize import curve_fit
from scipy.fft import fft, ifft
from scipy.stats import norm, median_abs_deviation
import emcee
import pywt
import corner
import warnings
warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
import lal
import lalsimulation as ls
import time
import sys
import os
from multiprocessing import Pool
from itertools import repeat
#ad hoc functions and initialization
import toymod_func as tf
import toymod_init as tin
from toymod_sig import *

"""-----------------------------------------------------------------"""
# getting the start time
init=time.time()

```

```
#graphics initialization
tf.graph_init(100,100,14,(10,6))

rng = default_rng(tin.seed)
"""-----------------------------------------------------------------"""
# definition of the waveform used in this test
fs=tin.fs # sampling frequency
delta_t=tin.delta_t # sampling interval
duration=tin.duration # SGB duration
f0 = tin.f0 # SGB central frequency
Q = 2.0 * np.pi * f0 * duration # SGB's Q-value

if tin.noise_mod=='ll' or tin.noise_mod=='lh':
    tin.hrss*=1e-22
    tin.hrss_low*=1e-22
    tin.hrss_high*=1e-22
elif tin.noise_mod=='v':
    tin.hrss*=1e-21
    tin.hrss_low*=1e-21
    tin.hrss_high*=1e-21

hrss = tin.hrss

if tin.test_char:
    print('Nyquist frequency: ',0.5*fs,' Hz')
    print(f'Q = {Q:.3}, duration = {duration:.3} s, f0 = {f0} Hz, hrss = {hrss}')
    print(f'f0 = {f0/fs:.3}*fs = {f0} Hz')

args_list=sig_arg(tin.sig_gen,Q)
hplus, hcross = sig_form(tin.sig_gen, *args_list)

sig_check(tin.sig_gen,Q,f0,hplus,hcross)
times=np.arange(0,len(hplus))*delta_t # sampling times

# plot of the model polarizations (for test purposes)
if tin.test_char:
    tf.my_plot(times,[hplus,hcross],['$h_+$','$h_x$','time (s)','strain'],[1,1],
    legend=1, save='SGB.pdf', show=0)

# physical signal: it is defined as the sum of the model signal plus a background

phi=rng.uniform(-0.5*np.pi,0.5*np.pi) # polarization angle
signal_t, signal0 = sig_generator(tin.noise_mod,hplus,hcross, phi)
nplus=len(hplus)

# plot of SGB ("true signal") and of SGB+noise ("onsource data") (for test
    purposes)
if tin.test_char:
    tf.my_plot(times,[signal0,signal_t],['onsource data','true signal','time (s)',
    'strain'],alpha=[1,1], legend=True, save='SGB-true.pdf', title='SGB-true')

del args_list
"""-----------------------------------------------------------------"""
#**Unmodelled reconstruction**
# frequency-domain representation of the injected signal
signal_f = fft(signal_t)
signal_psd = np.real(signal_f * np.conjugate(signal_f))
signal_psd1sided = 2*signal_psd[0:int(len(signal_psd)/2)] # simplified 1-sided psd
     with doubled DC component (it does not matter anyway)
freqs = np.linspace(0,fs,len(signal_f))
freqs1sided = freqs[0:int(len(signal_psd)/2)] # list of frequencies to be used
    with the 1-sided psd
```

```
87
88 #deleting some variables that won't be used anymore
89 del signal_f
90 del signal_psd
91 del freqs
92
93 # display of the 1-sided psd of the injected waveform
94 if tin.test_char:
95     plt.yscale('log')
96     tf.my_plot(freqs1sided,signal_psd1sided,['frequency (Hz)','psd (1/Hz)'],alpha
       =[1], save='SGB-true-freq.pdf')
97
98 # the "onsource unmodeled reconstruction"
99 burstrec0 = tf.burst_rec(signal0,0.0) # unmodeled reconstruction step
100
101 # plot of SGB ("true signal") and of SGB+noise ("onsource data") (for test
       purposes)
102 if tin.test_char:
103     tf.my_plot(times,[signal0,signal_t,burstrec0],['onsource data','true signal','
       rec signal','time (s)','strain'],alpha=[0.5,1,0.8], legend=True, save='SGB-
       recon.pdf')
104
105 # frequency-domain representation of the reconstructed signal
106 signal_f_b = fft(burstrec0)
107 signal_psd_b = np.real(signal_f_b * np.conjugate(signal_f_b))
108 signal_psd1sided_b = 2.0*signal_psd_b[0:int(len(signal_psd_b)/2)] # simplified 1-
       sided psd with doubled DC component (it does not matter anyway)
109 freqs_b = np.linspace(0,fs,len(signal_f_b))
110 freqs1sided_b = freqs_b[0:int(len(signal_psd_b)/2)] # list of frequencies to be
       used with the 1-sided psd
111
112 # frequency-domain representation of the noisy data
113 signal_f_0 = fft(signal0)
114 signal_psd_0 = np.real(signal_f_0 * np.conjugate(signal_f_0))
115 signal_psd1sided_0 = 2*signal_psd_0[0:int(len(signal_psd_0)/2)] # simplified 1-
       sided psd with doubled DC component (it does not matter anyway)
116 freqs_0 = np.linspace(0,fs,len(signal_f_0))
117 freqs1sided_0 = freqs_b[0:int(len(signal_psd_0)/2)] # list of frequencies to be
       used with the 1-sided psd
118
119 plt.plot(freqs1sided,signal_psd1sided,label='injected signal')
120 plt.plot(freqs1sided_0,signal_psd1sided_0,label='noisy data')
121 plt.plot(freqs1sided_b,signal_psd1sided_b,label='burst rec.')
122 plt.xlabel('frequency (Hz)')
123 plt.ylabel('psd (1/Hz)')
124 plt.yscale('log')
125 plt.legend()
126 plt.savefig(tin.save_path+'psd')
127 tf.del_graph()
128
129 #deleting some variables that won't be used anymore
130 del signal_f_b
131 del signal_psd_b
132 del freqs_b
133
134 del signal_f_0
135 del signal_psd_0
136 del freqs_0
137
138 # the "optimal unmodeled reconstruction"
139 opt = tf.burst_rec(signal_t,0.0)
140 # plot of "true signal" and of signal+noise ("onsource data")
```

```
141 if tin.test_char:
142     tf.my_plot(times,[signal0,signal_t,opt],['onsource data','true signal','
        optimal rec signal','time (s)','strain'],alpha=[0.5,1,0.8], legend=True, save=
        'SGB-opt.pdf')
143
144 # summary plots:
145 # - true signal: injected signal
146 # - onsource data: injected signal + noise
147 # - unmodeled reconstruction: reconstruction obtained from onsource data
148 # - optimal unmodeled rec.: reconstruction in the limit of vanishing noise
        background
149 tf.my_plot(times,[signal0,signal_t,burstrec0,opt],['onsource data','true signal','
        unmodelled rec','optimal unmodeled rec','time (s)','strain'],alpha
        =[0.5,1,0.8,0.5], legend=1, save='SGB-full.pdf')
150
151 print(f'optimal SNR: {tf.SNR(signal_t,nplus):.3}; SNRunmodeled: {tf.SNR(burstrec0,
        nplus):.3}')
152 print(f'optimal match: {tf.match(opt,signal_t):.3}, match: {tf.match(burstrec0,
        signal_t):.5}')
153 burst_res = burstrec0 - signal0
154 tf.my_plot(times,burst_res,['residuals','time (s)','residual'],alpha=[1],save='SGB
        -full-res.pdf')
155 tf.my_hist(burst_res, 50, ['residual','counts'], [1], save='SGB-hist-res.pdf',
        title='Residuals - whole range')
156
157 if tin.res_plot:
158     tf.res_plot(burst_res,'UNmod Residuals-cdf vs. N(0,1) cdf with finite-size
        fluctuations')
159
160 # sanity check: inject the true signal offsource and histogram the match (computed
        in the time domain)
161 signal_arr=np.zeros((tin.nrepeat,len(signal_t)))
162 for i in range(0,tin.nrepeat):
163     signal_arr[i,:]=signal_t+rng.normal(0,1,nplus)
164
165 with Pool() as temp_pool:
166      random_burstrec=np.asarray(temp_pool.starmap(tf.burst_rec, zip(signal_arr,
        repeat(0.0))))
167
168 bounds_br = np.percentile(random_burstrec, q=[5,95,50], axis=0) # find boundary
        lines for global confidence interval
169 m = [tf.match(random_burstrec[i],signal_t) for i in range(tin.nrepeat)] # compute
        matches with injected signal
170
171 # plot histogram of match values
172 mmin=min(m)
173 mmax=max(m)
174 m_mean=np.mean(m)
175 print(f'min(match): {mmin:.5}; max(match): {mmax:.5}; mean(match):{m_mean:.5}')
176 tf.my_hist(m,100, ['match','counts'], [1],save='off_inj_th.pdf', title='Offsource
        injections of theoretical waveform')
177
178 # plot the confidence band and compare it with the true signal
179 plt.fill_between(times,bounds_br[0],bounds_br[1], color='red', alpha=0.5,
        linewidth=0.3)
180 tf.my_plot(times, [signal_t,signal0,burstrec0],['true signal','onsource data','
        unmodelled rec','times (s)','strain'],alpha=[1,1,1],legend=1,save='SGB-th.pdf'
        ,title='Offsource injections of theoretical waveform')
181
182 # reinject offsource the "onsource reconstruction"
183 signal_arr=np.zeros((tin.nrepeat,len(signal_t)))
184 for i in range(0,tin.nrepeat):
```

```
185     signal_arr[i,:]=burstrec0+rng.normal(0,1,nplus)
186
187 with Pool(tin.n_th) as temp_pool:
188    random_burstrec=np.asarray(temp_pool.starmap(tf.burst_rec, zip(signal_arr,
       repeat(0.0))))    #
189
190 bounds_br = np.percentile(random_burstrec, q=[5,95,50], axis=0)
191 m1 = [tf.match(random_burstrec[i],burstrec0) for i in range(tin.nrepeat)]
192
193 m1min=min(m1)
194 m1max=max(m1)
195 m1_mean=np.mean(m1)
196 print(f'min(match): {m1min:.5}; max(match): {m1max:.5}; mean(match): {m1_mean:.5}'
       )
197 n1, bins1, patches1 = plt.hist(m1,100)
198 mode_idx1=np.argmax(n1)
199 mode1=0.5*(bins1[mode_idx1]+bins1[mode_idx1+1])
200 plt.xlabel('match')
201 plt.ylabel('counts')
202 plt.title('Offsource injections of reconstructed waveform')
203 tf.del_graph()
204
205 # plot the confidence band and compare it with the onsource reconstruction
206 plt.fill_between(times,bounds_br[0],bounds_br[1], color='red', alpha=0.5,
       linewidth=0.3,label='Unmodeled reconstruction')
207 tf.my_plot(times, [signal0,burstrec0],['onsource data','unmodelled rec','times (s)
       ','strain'],alpha=[0.5,1],legend=1,save='SGB-burst.pdf',title='Offsource
       injections of  burst reconstructed waveform')
208
209 tf.my_hist([m,m1],100, ['match','counts'], [1], title='Offsource injections of
       theoretical waveform')#h_range=( min(mmin,m1min), max(mmax,m1max) ),
210
211 plt.hist(m,100,range=( min(mmin,m1min), max(mmax,m1max) ),label='Match inj. th. &
       onsource rec.')
212 plt.hist(m1,100,range=( min(mmin,m1min), max(mmax,m1max) ),alpha=0.5,label='Match
       inj. onsource & onsource rec.')
213 plt.xlabel('match')
214 plt.ylabel('counts')
215 plt.title('Offsource injections of theoretical & reconstructed waveform')
216 plt.legend()
217 plt.savefig(tin.save_path+'Hist-th-burst.pdf')
218 tf.del_graph()
219 print('Offsource injection of exact waveform')
220 print(f'match mean: {np.mean(m):.4}; match std: {np.std(m):.2}; match median: {np.
       median(m):.4}\n')
221 print('Offsource injection of reconstructed waveform')
222 print(f'match mean: {np.mean(m1):.4}; match std: {np.std(m1):.2}; match median: {
       np.median(m1):.4}\n')
223 """-----------------------------------------------------------------"""
224 #Bayesian analisis --- carried out with emcee (https://emcee.readthedocs.io/en/
       stable/)
225
226 # MCMC initialization values
227 np.random.seed(tin.seed)
228 err = 1.
229 tin.data = (times,signal0,err)
230 initial = np.asarray(tin.initial) #np.array([10.,180.,0.01,0.03,0.,0.])
231 ndim = len(initial)
232 p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-4 * np.random.
       randn(ndim) for i in range(tin.nwalkers)] # vector of initial positions of
       random walkers
233
```

```
# check that initial values are all inside the acceptance region defined by the prior
isinregion = True
for theta in p0:
    isinregion = isinregion and np.isfinite(tf.log_prior(theta))
'initial values OK' if isinregion else 'initial values not OK'

# burn-in step
model_func=model_finder(tin.model_func)
with Pool(tin.n_th) as pool:
    sampler = emcee.EnsembleSampler(tin.nwalkers, ndim, tf.log_prob, moves=emcee.moves.DEMove(), pool=pool, args=[model_func] )
    print("Running burn-in...")
    p0, _, _ =sampler.run_mcmc(p0, tin.nburn, progress=False) #20000
    print("Mean acceptance fraction: {0:.3f}".format(np.mean(sampler.acceptance_fraction)))

    #quiet=True NON blocca il programma se tau e' stimato "male"
    tau=np.mean(sampler.get_autocorr_time(quiet=True))
    print("Mean autocorrelation time: {0:.3f} steps".format(tau))

    #show the MAP estimate after (lengthy) burn-in
    samples = sampler.flatchain
    lnprobs = sampler.flatlnprobability

    theta_max  = samples[np.argmax(sampler.flatlnprobability)]

    print('MAP estimate:',theta_max)
    tf.my_plot(times,[signal0,model_func(times, *theta_max)],['Onsource data','MAP estimate','times (s)','strain'], alpha=[1,1],legend=1)
    PE_res = model_func(times, *theta_max)-signal0
    tf.my_plot(times,PE_res,['residuals','time (s)', 'residual'],alpha=[1],save='PE-res.pdf')
    tf.my_hist(PE_res,50,['residual','counts'],[1], save='PE SGB-hist-res.pdf', title='PE Residuals - whole range')

    if tin.res_plot:
        tf.res_plot(PE_res,'PE burn-in Residuals-cdf vs. N(0,1) cdf with finite-size fluctuations')
    # set the new initial values at the position of the MAP estimate after burn-in
    initial = theta_max#np.array([a_max,f_max,sigma_max,t0_max,phi_max])#,bkg_max
    p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-5 * np.random.randn(ndim) for i in range(tin.nwalkers)]
    # production step
    sampler.reset()
    print("Running production...")
    pos, prob, state = sampler.run_mcmc(p0, tin.niter, progress=False)
    print("Mean acceptance fraction: {0:.3f}".format(np.mean(sampler.acceptance_fraction)))
    print("Mean autocorrelation time: {0:.3f} steps".format(np.mean(sampler.get_autocorr_time(quiet=True)) ))

if tin.traces:
    # check the stationarity of chains
    fig, axes = plt.subplots(ndim, figsize=(20, 7), sharex=True)
    schains = sampler.get_chain(thin=25)
    labels = np.linspace(1,len(initial),len(initial)) #["amp", "f", "width","t0","phi"]#,"bkg"
    for i in range(ndim):
        ax = axes[i]
        ax.plot(schains[:, :, i], "k", alpha=0.3)
        ax.set_xlim(0, len(schains))
```

```
285         ax.set_ylabel(labels[i])
286         ax.yaxis.set_label_coords(-0.1, 0.5)
287 
288     axes[-1].set_xlabel("step number");
289     fig.savefig(tin.save_path+'chains.pdf')
290     for i in range(ndim):
291         tf.del_graph()
292     del schains
293 
294 # show the MAP estimate after the production step
295 samples = sampler.flatchain
296 lnprobs = sampler.flatlnprobability
297 
298 theta_max  = samples[np.argmax(sampler.flatlnprobability)]
299 
300 print('MAP estimate:',theta_max)
301 tf.my_plot(times,[signal0,model_func(times, *theta_max)],['Onsource data','MAP
        estimate','time (s)','strain'],alpha=[1,1], legend=1,save='MCMC_MAP.pdf')
302 PE_res = model_func(times, *theta_max)-signal0
303 tf.my_plot(times,PE_res,["residuals",'time (s)','residual'],alpha=[1])
304 tf.my_hist(PE_res,50,['residual','counts'],[1],title='Residuals - whole range')
305 if tin.res_plot:
306     tf.res_plot(PE_res,'PE Residuals-cdf vs. N(0,1) cdf with finite-size
        fluctuations')
307 
308 # show the MAP estimate after the production step, including the uncertainty band,
       obtained with random draws from the empirical posterior distribution
309 plt.fill_between(times,bounds_br[0],bounds_br[1], color='red', alpha=0.5,
        linewidth=0.3,label='Unmodeled reconstruction')
310 samples = sampler.flatchain
311 lnprobs = sampler.flatlnprobability
312 first=True
313 for ind in np.random.randint(len(samples), size=500):
314     theta = samples[ind]
315     if first:
316         first = False
317         plt.plot(times, model_func(times, *theta), color="g", alpha=0.1,linewidth
        =0.5,label='MCMC')
318     else:
319         plt.plot(times, model_func(times, *theta), color="g", alpha=0.1,linewidth
        =0.5)
320 tf.my_plot(times,[signal0,burstrec0],['Onsource data','Onsource unmodeled','time (
        s)',r'$h$'],alpha=[0.8,0.5],legend=1,save='MCMC_band.pdf')
321 
322 sample_list=sampler.get_chain(flat=True,discard=1000)
323 
324 print( 'Sample list saved. Length is: ', len(sample_list))
325 
326 # plot the corner plot
327 if tin.corn_plot:
328     figure=corner.corner(sample_list,
329                          labels=np.linspace(1,len(initial),len(initial)),
330                          quantiles=[0.05, 0.5, 0.95],
331                          show_titles=True, title_kwargs={"fontsize": 14})
332     figure.savefig(tin.save_path+'corner.pdf')
333     tf.del_graph()
334 print('Burst-injected match: ', tf.match(signal_t,burstrec0) )
335 print('MCMC_MAP-injected match: ', tf.match(signal_t,model_func(times, *theta_max)
        ) )
336 print('MCMC_MAP-Burst match: ', tf.match(burstrec0,model_func(times, *theta_max))
        )
337 
```

```
tf.my_plot(times,[model_func(times, *theta_max),burstrec0],['MCMC_MAP','Unmodeled'
    ,'time (s)','strain'],alpha=[1,0.5], legend=1,save='SGB-MCMC-burst.pdf',title=
    'Onsource MCMC and unmodeled reconstruction')
# example of waveform extracted from 'posterior' distribution
if tin.test_char:
    pars=sample_list[np.random.randint(len(sample_list))]
    #plt.plot(times, func(times, *pars), 'r-')
    tf.my_plot(times,[model_func(times, *pars),signal0],['random MCMC waveform','
    Onsource data','time (s)','strain'],alpha=[1,0.5],legend=1)
# generation of large set of random waveforms from posterior pdf
posterior = sample_list[np.random.randint(len(sample_list), size=tin.nrepeat)] #
    extract a random parameter set from posterior distribution
random_waveform=np.array([model_func(times,*posterior[i]) for i in range(tin.
    nrepeat)]) # compute the corresponding waveforms
bounds = np.percentile(random_waveform, q=[5,95,50], axis=0) # find bounds defined
     by the distribution of waveforms
plt.fill_between(times,bounds[0],bounds[1], color='red', alpha=0.3, linewidth=0.5)
tf.my_plot(times, [bounds[2],signal0],['MCMC median','Onsource data','time (s)','
    strain'],alpha=[1,0.3], legend=1,save='SGB-MCMC.pdf')

plt.fill_between(times,bounds[0],bounds[1], color='red', alpha=0.4, linewidth=0.5)
plt.fill_between(times,bounds_br[0],bounds_br[1], color='green', alpha=0.3,
    linewidth=0.7)
tf.my_plot(times,[model_func(times, *theta_max),burstrec0,signal0],['MCMC','
    unmodelled','onsource data','time (s)','strain'], alpha=[1,1,0.3],legend=1,
    save='combined.pdf')

# match values from offsource injection of templated reconstructions with
    unmodeled onsource reconstruction
m2 = [tf.match(random_waveform[i],burstrec0) for i in range(tin.nrepeat)]
m2min=min(m2)
m2max=max(m2)
print(f'min(match): {m2min}; max(match): {m2max}')
n2, bins2, patches2 = plt.hist(m2,100)
mode_idx2=np.argmax(n2)
mode2=0.5*(bins2[mode_idx2]+bins2[mode_idx2+1])
plt.xlabel('match')
plt.ylabel('counts')
plt.title('Match between MCMC posterior and onsource unmodeled waveform')
tf.del_graph()

plt.hist(m2,100,range=( min(m1min,m2min), max(m1max,m2max) ),label='Match inj.
    MCMC & onsource rec.')
plt.hist(m1,100,range=( min(m1min,m2min), max(m1max,m2max) ),alpha=0.5,label='
    Match inj. onsource & onsource rec.')
plt.xlabel('match')
plt.ylabel('counts')
plt.title('Offsource inj. of rec. waveform, match with ons. rec. & MCMC')
plt.legend()
plt.savefig(tin.save_path+'Hist-MCMC-burst.pdf')
tf.del_graph()
print('Offsource injection of onsource unmodeled waveform with onsource unmodeled
    waveform')
print(f'match mean: {np.mean(m1):.4}; match std: {np.std(m1):.2}; match median: {
    np.median(m1):.4}; match mode: {mode1:.4}\n')
print('Waveform from MCMC posterior with onsource unmodeled waveform')
print(f'match mean: {np.mean(m2):.4}; match std: {np.std(m2):.2}; match median: {
    np.median(m2):.4}; match mode: {mode2:.4}\n')
print(f'Distance between means: {np.abs(np.mean(m1)-np.mean(m2)):.4} +- {np.sqrt(
    np.var(m1)+np.var(m2)):.4} ==> {np.abs(np.mean(m1)-np.mean(m2))/np.sqrt(np.var
    (m1)+np.var(m2)):.4} sigma')
print(f'Distance between modes: {np.abs(mode1-mode2):.4} ==> {np.abs(mode1-mode2)/
```

```
    np.sqrt(np.var(m1)+np.var(m2)):.4} sigma')
print('\n')
"""----------------------------------------------------------------"""
#**Bayesian reconstruction w/ wrong model**
np.random.seed(tin.seed)
err = 1.
initial = np.asarray(tin.initial_W)#np.array([10.,180.,0.01,0.03,0.,0.])
ndim = len(initial)
p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-4 * np.random.
    randn(ndim) for i in range(tin.nwalkers)]

# check that initial values are all inside the acceptance region defined by the
    prior
isinregion = True
for theta in p0:
    isinregion = isinregion and np.isfinite(tf.log_prior(theta))
'initial values OK' if isinregion else 'initial values not OK'

model_func_2=model_finder(tin.model_func_W)
with Pool(tin.n_th) as pool1:
    sampler = emcee.EnsembleSampler(tin.nwalkers, ndim, tf.log_prob, moves=emcee.
    moves.DEMove(), pool=pool1, args=[model_func_2])

    print("Running burn-in...")
    p0, _, _ = sampler.run_mcmc(p0, tin.nburn, progress=False)#20000

    print("Mean acceptance fraction: {0:.3f}".format(np.mean(sampler.
    acceptance_fraction)))

    print("Mean autocorrelation time: {0:.3f} steps".format(np.mean(sampler.
    get_autocorr_time(quiet=True)) ))

    #show the MAP estimate after (lengthy) burn-in
    samples = sampler.flatchain
    lnprobs = sampler.flatlnprobability

    theta_max  = samples[np.argmax(sampler.flatlnprobability)]

    print('MAP estimate:',theta_max)
    tf.my_plot(times,[signal0,model_func_2(times, *theta_max)],['Onsource data','
    MAP estimate W','times (s)','strain'], alpha=[1,1],legend=1)
    PE_res = model_func_2(times, *theta_max)-signal0
    tf.my_plot(times,PE_res,['residuals','time (s)', 'residual'],alpha=[1],save='
    PEw-res.pdf')
    tf.my_hist(PE_res,50,['residual','counts'],[1], save='PEw SGB-hist-res.pdf',
    title='PEw Residuals - whole range')

    if tin.res_plot:
        tf.res_plot(PE_res,'PEw burn-in Residuals-cdf vs. N(0,1) cdf with finite-
    size fluctuations')
    # set the new initial values at the position of the MAP estimate after burn-in
    initial = theta_max
    p0 = [ np.array(initial)*(1 + 0.0001 * np.random.randn(ndim)) + 1e-5 * np.
    random.randn(ndim) for i in range(tin.nwalkers)]

    sampler.reset()
    print("Running production...")
    pos, prob, state = sampler.run_mcmc(p0, tin.niter, progress=False)
    print("Mean acceptance fraction: {0:.3f}".format(np.mean(sampler.
    acceptance_fraction)))
    print("Mean autocorrelation time: {0:.3f} steps".format(np.mean(sampler.
    get_autocorr_time(quiet=True)) ))
```

```
431
432 if tin.traces:
433     # check the stationarity of chains
434     fig, axes = plt.subplots(ndim, figsize=(20, 7), sharex=True)
435     schains = sampler.get_chain(thin=25)
436     labels = np.linspace(1,len(initial),len(initial)) #["amp", "f", "width","t0","
        phi"]#,"bkg"
437     for i in range(ndim):
438         ax = axes[i]
439         ax.plot(schains[:, :, i], "k", alpha=0.3)
440         ax.set_xlim(0, len(schains))
441         ax.set_ylabel(labels[i])
442         ax.yaxis.set_label_coords(-0.1, 0.5)
443
444     axes[-1].set_xlabel("step number");
445     fig.savefig(tin.save_path+'chains_w.pdf')
446     for i in range(ndim):
447         tf.del_graph()
448     del schains
449
450 #show the MAP estimate after production
451 samples = sampler.flatchain
452 lnprobs = sampler.flatlnprobability
453 theta_max  = samples[np.argmax(sampler.flatlnprobability)]
454
455 print('MAP estimate:',theta_max)
456 tf.my_plot(times,[signal0,model_func_2(times, *theta_max)],['Onsource data','MAP
        estimate W','times (s)','strain'], alpha=[1,1],legend=1, save='SGB-MCMC-wrong.
        pdf')
457 PE_res = model_func_2(times, *theta_max)-signal0# cq func
458 tf.my_plot(times,PE_res,['residuals','time (s)', 'residual'],alpha=[1])
459 tf.my_hist(PE_res,50,['residual','counts'],[1],title='W Residuals - whole range')
460 if tin.res_plot:
461     tf.res_plot(PE_res,'PEw Residuals-cdf vs. N(0,1) cdf with finite-size
        fluctuations')
462
463 plt.plot(times,signal0,label='Onsource data')
464 samples = sampler.flatchain
465 lnprobs = sampler.flatlnprobability
466 first=True
467 for ind in np.random.randint(len(samples), size=500):
468     theta = samples[ind]
469     if first:
470         first = False
471         plt.plot(times, model_func_2(times, *theta), color="r", alpha=0.2,
        linewidth=0.5,label='MCMC')
472     else:
473         plt.plot(times, model_func_2(times, *theta), color="r", alpha=0.2,
        linewidth=0.5)
474 #plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
475 plt.xlabel('time (s)')
476 plt.ylabel(r'$h$')
477 plt.legend()
478 plt.savefig(tin.save_path+'MCMC_band-wrong.pdf')
479 tf.del_graph()
480
481 sample_listW=sampler.get_chain(flat=True,discard=1000)#,discard=10000,thin=100
482
483 print( 'Sample list saved. Length is: ', len(sample_listW))
484
485 if tin.corn_plot:
486     figure=corner.corner(sample_listW,
```

```
                    labels=np.linspace(1,len(tin.initial_W),len(tin.initial_W)), #[r"$A$", r"$f_0$", r"$\Gamma$", r"$t_0$", r"$\varphi$"],#,r'bkg'
                    quantiles=[0.05, 0.5, 0.95],
                    show_titles=True, title_kwargs={"fontsize": 14})
    figure.savefig(tin.save_path+'cornerW.pdf')
    tf.del_graph()

#
print('Burst-injected match: ', tf.match(signal_t,burstrec0) )
print('MCMC_MAP-injected match (wrong model): ', tf.match(signal_t,model_func_2(times, *theta_max)) )
print('MCMC_MAP-Burst match (wrong model):', tf.match(burstrec0,model_func_2(times, *theta_max)) )

tf.my_plot(times,[model_func_2(times, *theta_max),burstrec0],['MCMC_MAP','Unmodeled','time (s)','strain'],alpha=[1,0.5],title='Onsource MCMC and unmodeled reconstruction (wrong model)',legend=1,save='SGB-MCMC-burstW.pdf')
if tin.test_char:
    # example of waveform extracted from 'posterior' distribution
    pars=sample_listW[np.random.randint(len(sample_listW))]
    tf.my_plot(times, [model_func_2(times, *pars),signal0], ['random MCMC waveform','Onsource data','time (s)','strain'],alpha=[1,0.5], legend=1,)

# generation of large set of random waveforms from posterior pdf
posterior = sample_listW[np.random.randint(len(sample_listW), size=tin.nrepeat)]
random_waveform=np.array([model_func_2(times,*posterior[i]) for i in range(tin.nrepeat)])
bounds = np.percentile(random_waveform, q=[5,95,50], axis=0)

plt.fill_between(times,bounds[0],bounds[1], color='red', alpha=0.3, linewidth=0.5)
tf.my_plot(times, [bounds[2],signal0], ['MCMC median (wrong model)','Onsource data','time (s)','strain'], alpha=[1,0.3], legend=1,save='SGB-MCMC-wrong.pdf')
plt.fill_between(times,bounds[0],bounds[1], color='red', alpha=0.4, linewidth=0.5)
plt.fill_between(times,bounds_br[0],bounds_br[1], color='green', alpha=0.3, linewidth=0.7)
tf.my_plot(times,[model_func_2(times, *theta_max),burstrec0,signal0],['MCMC (wrong model)','Unmodeled','Onsource data','time (s)','strain'],alpha=[1,1,0.3], legend=1,save='combined-wrong.pdf')

# match values from offsource injection of templated reconstructions with unmodeled onsource reconstruction
m2 = [tf.match(random_waveform[i],burstrec0) for i in range(tin.nrepeat)]
m2min=min(m2)
m2max=max(m2)
print(f'min(match): {m2min}; max(match): {m2max}')
n2, bins2, patches2 = plt.hist(m2,100)
mode_idx2=np.argmax(n2)
mode2=0.5*(bins2[mode_idx2]+bins2[mode_idx2+1])
plt.xlabel('match')
plt.ylabel('counts')
plt.title('Match between MCMC posterior and onsource unmodeled waveform (wrong model)')
tf.del_graph()
plt.hist(m2,100,range=( min(m1min,m2min), max(m1max,m2max) ),label='Match inj. MCMC & onsource rec.')
plt.hist(m1,100,range=( min(m1min,m2min), max(m1max,m2max) ),alpha=0.5,label='Match inj. onsource & onsource rec.')
plt.xlabel('match')
plt.ylabel('counts')
plt.title('Offsource inj. of rec. waveform, match with ons. rec. & MCMC (wrong model)')
plt.legend()
```

```
533 plt.savefig(tin.save_path+'Hist-MCMC-burst-wrong.pdf')
534 tf.del_graph()
535 print('Offsource injection of onsource unmodeled waveform with onsource unmodeled
        waveform')
536 print(f'match mean: {np.mean(m1):.4}; match std: {np.std(m1):.2}; match median: {
        np.median(m1):.4}; match mode: {mode1:.4}\n')
537 print('Waveform from MCMC posterior with onsource unmodeled waveform')
538 print(f'match mean: {np.mean(m2):.4}; match std: {np.std(m2):.2}; match median: {
        np.median(m2):.4}; match mode: {mode2:.4}\n')
539 print(f'Distance between means: {np.abs(np.mean(m1)-np.mean(m2)):.4} +- {np.sqrt(
        np.var(m1)+np.var(m2)):.4} ==> {np.abs(np.mean(m1)-np.mean(m2))/np.sqrt(np.var
        (m1)+np.var(m2)):.4} sigma')
540 print(f'Distance between modes: {np.abs(mode1-mode2):.4} ==> {np.abs(mode1-mode2)/
        np.sqrt(np.var(m1)+np.var(m2)):.4} sigma')
541 print('\n')
542
543 #deleting some variables that won't be used anymore
544 del sample_list
545 del sample_listW
546
547 """-------------------------------------------------------------"""
548 #systematics
549
550 #defining hrss to loop over
551 print('def snrs \n')
552 #hrss_arr = np.logspace(np.log10(tin.hrss_low),np.log10(tin.hrss_high),num=tin.
        nSNRs)
553 hrss_arr = np.linspace(tin.hrss_low,tin.hrss_high,num=tin.nSNRs)
554
555 #defining empty arrays & lists
556 print('def empty arrays \n')
557 optSNR=[]
558
559 TA=[] # true alarms
560 FA=[] # false alarms
561 IC=[] # inconclusive cases
562 FC=[] # flipped cases
563
564 m1=np.zeros((tin.nloops, tin.nrepeat))
565 mean_burston2bursoff =np.zeros(tin.nloops) # []
566 std_burston2bursoff = np.zeros(tin.nloops) # []
567 median_burston2bursoff = np.zeros(tin.nloops) # []
568 mode_burston2bursoff = np.zeros(tin.nloops) # []
569
570 m2=np.zeros((tin.nloops, tin.nrepeat))
571 mean_burston2PEsample = np.zeros(tin.nloops) #[]
572 std_burston2PEsample = np.zeros(tin.nloops) #[]
573 median_burston2PEsample = np.zeros(tin.nloops) #[]
574 mode_burston2PEsample = np.zeros(tin.nloops) #[]
575
576 m3=np.zeros((tin.nloops, tin.nrepeat))
577 mean_burston2PEsampleW = np.zeros(tin.nloops) #[]
578 std_burston2PEsampleW = np.zeros(tin.nloops) #[]
579 median_burston2PEsampleW = np.zeros(tin.nloops) #[]
580 mode_burston2PEsampleW = np.zeros(tin.nloops) #[]
581
582 new_initial=np.zeros(len(tin.initial))
583 new_initial_w=np.zeros(len(tin.initial))
584
585 for i in range(1,len(tin.initial)):
586     new_initial[i]=tin.initial[i]
587 for i in range(1,len(tin.initial_W)):
```

```
588     new_initial_w[i]=tin.initial_W[i]
589
590 h_count=0
591 for new_hrss in hrss_arr:
592     h_count+=1
593     print('SNR #',h_count)
594
595     #definition of signal waveforms
596     args_list=sig_arg(tin.sig_gen,Q,new_hrss)
597     hplus,hcross=sig_form(tin.sig_gen,*args_list)
598     del args_list
599     signal0_s=np.zeros((tin.nloops,nplus))
600     phi=rng.uniform(-0.5*np.pi,0.5*np.pi) # polarization angle
601     print('def sigs')
602     for i in range(0, tin.nloops):
603         if i%10==0:
604             print( i, end=',')
605         if i==0:
606             signal_t,signal0_s[i,:] = sig_generator(tin.noise_mod,hplus,hcross,
    phi)
607             start_amp=np.max(signal0_s[i,:])
608         else:
609             _,signal0_s[i,:] = sig_generator(tin.noise_mod,hplus,hcross, phi)
610             start_amp+=np.max(signal0_s[i,:])
611
612     start_amp/=tin.nloops
613     print('\n')
614
615     opt_snr=tf.SNR(signal_t,nplus)
616     print(f'optimal SNR: {opt_snr:.3}')
617     optSNR.append(opt_snr)
618
619     print('\n')
620     if tin.sig_gen=='sg':
621         new_initial[0]=start_amp
622         new_initial_w[0]=start_amp
623     elif tin.sig_gen=='sg_amp':
624         new_initial[0]=new_hrss
625         new_initial_w[0]=new_hrss
626
627     if tin.test_char:
628         tf.my_plot(times,[signal0_s[0],(signal_t)],['onsource data','true signal',
    'time (s)','strain'], alpha=[1,0.8],legend=1,save=f'ex-SGB-true-s-{h_count}.
    pdf')
629
630     if h_count>1:
631         del burstrec0_s
632     with Pool(tin.n_th) as temp_pool:
633         burstrec0_s=np.asarray(temp_pool.starmap(tf.burst_rec, zip(signal0_s,
    repeat(0.0))))
634
635     #reinjecting reconstruction
636     print('reinjecting reconstruction')
637     np.random.seed(tin.seed)
638     if h_count>1:
639         del signal_arr_s
640     signal_arr_s=np.zeros((tin.nrepeat*np.shape(burstrec0_s)[0],np.shape(
    burstrec0_s)[1]))
641     count=0
642
643     for i in range(0,tin.nrepeat*np.shape(burstrec0_s)[0]):
644         if i%tin.nrepeat==0 and i!=0:
```

```
            count=count+1
        signal_arr_s[i,:]=burstrec0_s[count]+rng.normal(0,1,np.shape(burstrec0_s)
    [1])#attenta all'ordine dell'array

    if h_count>1:
        del random_burstrec_s
    with Pool(tin.n_th) as temp_pool:
        random_burstrec_s=np.asarray(temp_pool.starmap(tf.burst_rec, zip(
    signal_arr_s, repeat(0.0))))

    if tin.test_char:
        print('signal_arr shape=',np.shape(signal_arr_s))
        print('signal_recon_arr shape=',np.shape(burstrec0_s))
        print('recon_reinj_arr shape=',np.shape(random_burstrec_s))
        print('\n')
    m1min=+2.
    m1max=-2.
    for nl in range(0,tin.nloops):
        m1[nl] = [tf.match(random_burstrec_s[i],burstrec0_s[nl]) for i in range(0+
    tin.nrepeat*nl,tin.nrepeat+tin.nrepeat*nl)] # empirical match distribution
        if min(m1[nl])<m1min:
            m1min=min(m1[nl])
        if max(m1[nl])>m1max:
            m1max=max(m1[nl])
        mean_burston2bursoff[nl]=(np.mean(m1[nl]))
        std_burston2bursoff[nl]=(np.std(m1[nl]))
        median_burston2bursoff[nl]=(np.median(m1[nl]))

    if h_count>1:
        del sample_list_arr_gauss
    with Pool(tin.n_th) as temp_pool:
        sample_list_arr_gauss=np.asarray(temp_pool.starmap(tf.mcmc_func, zip(
    signal0_s, repeat(model_func), repeat(new_initial))))#pool.starmap(func, zip(
    a_args, repeat(second_arg)))
    print('done w/ mcmc \n')
    #np.savez_compressed(tin.save_path+f'samples_{h_count}',sample_list_arr_gauss)

    m2min=+2.
    m2max=-2.
    # empirical posterior distribution
    for nl in range(0,tin.nloops):
        if nl%10==0:
            print(nl,end=',')
        posterior_gauss = sample_list_arr_gauss[nl,:,:][np.random.randint(0,len(
    sample_list_arr_gauss[nl,:,:]), tin.nrepeat)] # extract a random parameter set
     from posterior distribution
        random_waveform_gauss=np.array([model_func(times,*posterior_gauss[i]) for
    i in range(tin.nrepeat)]) # compute the corresponding waveforms

        m2[nl] = [tf.match(random_waveform_gauss[i],burstrec0_s[nl]) for i in
    range(tin.nrepeat)]
        if min(m2[nl])<m2min:
            m2min=min(m2[nl])
        if max(m2[nl])>m2max:
            m2max=max(m2[nl])

        mean_burston2PEsample[nl]=(np.mean(m2[nl]))
        std_burston2PEsample[nl]=(np.std(m2[nl]))
        median_burston2PEsample[nl]=(np.median(m2[nl]))
    print('\n')

    if h_count>1:
```

```
698         del sample_list_arr_w
699     with Pool(tin.n_th) as temp_pool:
700         sample_list_arr_w=np.asarray(temp_pool.starmap(tf.mcmc_func, zip(signal0_s
        , repeat(model_func_2),repeat(new_initial_w))))
701     print('done w/ mcmc \n')
702     #np.savez_compressed(tin.save_path+f'samplesw_{h_count}',sample_list_arr_w)
703 
704     # empirical posterior distribution
705     m3min=+2.
706     m3max=-2.
707     for nl in range(0,tin.nloops):
708         if nl%10==0:
709             print(nl,end=',')
710         posterior_w = sample_list_arr_w[nl,:,:][np.random.randint(0,len(
        sample_list_arr_w[nl,:,:]), tin.nrepeat)] # extract a random parameter set
        from posterior distribution
711         random_waveform_w=np.array([model_func_2(times,*posterior_w[i]) for i in
        range(tin.nrepeat)]) # compute the corresponding waveforms
712 
713         m3[nl] = [tf.match(random_waveform_w[i],burstrec0_s[nl]) for i in range(
        tin.nrepeat)]
714         if min(m3[nl])<m3min:
715             m3min=min(m3[nl])
716         if max(m3[nl])<m3max:
717             m3max=max(m3[nl])
718 
719         mean_burston2PEsampleW[nl]=(np.mean(m3[nl]))
720         std_burston2PEsampleW[nl]=(np.std(m3[nl]))
721         median_burston2PEsampleW[nl]=(np.median(m3[nl]))
722     print('\n')
723 
724     #it's graph time
725     problem_list=[]
726     for nl in range(0,tin.nloops):
727         if nl%10==0:
728             print(nl,end=',')
729         lim_min=min(m1min,m2min,m3min)
730 
731         n1, bins1, patches1 = plt.hist(m1[nl],100,color='blue',alpha=0.15,range=(
        lim_min,1))
732         mode_idx1=np.argmax(n1)
733         mode1=0.5*(bins1[mode_idx1]+bins1[mode_idx1+1])
734 
735         n2, bins2, patches2 = plt.hist(m2[nl],100,color='red',alpha=0.15,range=(
        lim_min,1))
736         mode_idx2=np.argmax(n2)
737         mode2=0.5*(bins2[mode_idx2]+bins2[mode_idx2+1])
738 
739         n3, bins3, patches3  = plt.hist(m3[nl],100,color='green',alpha=0.15,range
        =(lim_min,1))
740         mode_idx3=np.argmax(n3)
741         mode3=0.5*(bins3[mode_idx3]+bins3[mode_idx3+1])
742 
743         mode_burston2bursoff[nl]=(mode1)
744         mode_burston2PEsample[nl]=(mode2)
745         mode_burston2PEsampleW[nl]=(mode3)
746 
747 # For now, this part does not serve any purpose, but it could be used to exclude
        points if they were "bad" for some reason (e.g. too low a match or too high a
        std)
748         #if (std_burston2bursoff[nl]>0.96) or (std_burston2PEsample[nl]>0.96) or (
        std_burston2PEsampleW[nl]>0.96):
```

```
749             #problem_list.append[nl]
750     plt.savefig(tin.save_path+f'graph_time{h_count}.pdf')#this plots the match
        distributions
751     tf.del_graph()
752     print('\n')
753 
754     if len(problem_list)!=0:
755         print('there are some problems')
756 
757     print(f'Burst on burst: mean: {np.mean(mean_burston2bursoff):.4} +- {np.std(
        mean_burston2bursoff):.4}; '
758         f'std: {np.mean(std_burston2bursoff):.4} +- {np.std(std_burston2bursoff)
        :.4}; '
759         f'median: {np.mean(median_burston2bursoff):.4} +- {np.std(
        median_burston2bursoff):.4}; '
760         f'mode: {np.mean(mode_burston2bursoff):.4} +- {np.std(mode_burston2bursoff
        ):.4} ')
761 
762     print(f'Burst on PE: mean: {np.mean(mean_burston2PEsample):.4} +- {np.std(
        mean_burston2PEsample):.4}; '
763         f'std: {np.mean(std_burston2PEsample):.4} +- {np.std(std_burston2PEsample)
        :.4}; '
764         f'median: {np.mean(median_burston2PEsample):.4} +- {np.std(
        median_burston2PEsample):.4}; '
765         f'mode: {np.mean(mode_burston2PEsample):.4} +- {np.std(
        mode_burston2PEsample):.4} ')
766 
767     print(f'Burst on PE (wrong): mean: {np.mean(mean_burston2PEsampleW):.4} +- {np
        .std(mean_burston2PEsampleW):.4}; '
768         f'std: {np.mean(std_burston2PEsampleW):.4} +- {np.std(
        std_burston2PEsampleW):.4}; '
769         f'median: {np.mean(median_burston2PEsampleW):.4} +- {np.std(
        median_burston2PEsampleW):.4}; '
770         f'mode: {np.mean(mode_burston2PEsampleW):.4} +- {np.std(
        mode_burston2PEsampleW):.4} ')
771 
772     #here we're plotting everything
773     tf.sys_plots(mean_burston2bursoff,mean_burston2PEsample,mean_burston2PEsampleW
        ,std_burston2bursoff,std_burston2PEsample,std_burston2PEsampleW,
        median_burston2bursoff,median_burston2PEsample,median_burston2PEsampleW,
        mode_burston2bursoff,mode_burston2PEsample,mode_burston2PEsampleW,h_count)
774 
775     # redefining arrays and lists in case some points were excluded
776     if h_count>1:
777         del new_mean_burston2bursoff
778         del new_mean_burston2PEsample
779         del new_mean_burston2PEsampleW
780         del new_std_burston2bursoff
781         del new_std_burston2PEsample
782         del new_std_burston2PEsampleW
783 
784     new_mean_burston2bursoff=[]
785     new_mean_burston2PEsample=[]
786     new_mean_burston2PEsampleW=[]
787     new_std_burston2bursoff=[]
788     new_std_burston2PEsample=[]
789     new_std_burston2PEsampleW=[]
790 
791     for nl in range(0,tin.nloops):
792         if nl not in problem_list:
793             new_mean_burston2bursoff.append(mean_burston2bursoff[nl])
794             new_mean_burston2PEsample.append(mean_burston2PEsample[nl])
```

```
795             new_mean_burston2PEsampleW.append(mean_burston2PEsampleW[nl])
796             new_std_burston2bursoff.append(std_burston2bursoff[nl])
797             new_std_burston2PEsample.append(std_burston2PEsample[nl])
798             new_std_burston2PEsampleW.append(std_burston2PEsampleW[nl])
799
800     print('rejected fraction=', len(problem_list)/len(mean_burston2PEsample))
801
802     tdist=(np.array(new_mean_burston2bursoff)-np.array(new_mean_burston2PEsample))
        /np.sqrt(np.array(new_std_burston2bursoff)**2+np.array(
        new_std_burston2PEsample)**2)
803     wdist=(np.array(new_mean_burston2bursoff)-np.array(new_mean_burston2PEsampleW)
        )/np.sqrt(np.array(new_std_burston2bursoff)**2+np.array(
        new_std_burston2PEsampleW)**2)
804     fracs=tf.trigger(tdist,wdist,new_hrss,h_count)
805
806     TA.append(fracs[0])
807     FA.append(fracs[1])
808     IC.append(fracs[2])
809     FC.append(fracs[3])
810
811 #plotting result fractions against snr/hrss
812 TA_err=[np.sqrt(tin.nloops*TA[i]*(1-TA[i])) for i in range(len(TA))]
813 plt.plot(hrss_arr,TA,'-o',color='green',alpha=0.5, label='true alarms')
814 plt.plot(hrss_arr,FA,'-o',color='red',alpha=0.5, label='false alarms')
815 plt.plot(hrss_arr,IC,'-o',color='black',alpha=0.5, label='inconclusive alarms')
816 plt.plot(hrss_arr,FC,'-o',color='blue',alpha=0.5, label='flipped alarms')
817 plt.xlabel('hrss')
818 plt.ylabel('alarm fractions')
819 plt.legend()
820 plt.savefig(tin.save_path+'TAplot.pdf')
821 tf.del_graph()
822
823 plt.plot(optSNR,TA,'-o',color='green',alpha=0.5, label='true alarms')
824 plt.plot(optSNR,FA,'-o',color='red',alpha=0.5, label='false alarms')
825 plt.plot(optSNR,IC,'-o',color='black',alpha=0.5, label='inconclusive alarms')
826 plt.plot(optSNR,FC,'-o',color='blue',alpha=0.5, label='flipped alarms')
827 plt.xlabel('optimal SNR')
828 plt.ylabel('alarm fractions')
829 plt.legend()
830 plt.savefig(tin.save_path+'TAplot-SNR.pdf')
831 tf.del_graph()
832
833 #how long everything took to run
834 end=time.time()
835 print('it took me ', (end-init)/60., 'minutes to run the program')
```

Listing 4: toymodel_cl_05.py

In order to run the threshold study code mentioned in 3.3 one needs all the same files mentioned above to run the toy model, as the code uses the same functions and the same initialization file as toymodel_cl_05.py.

```
1 # library imports
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from numpy.random import default_rng
5 from scipy import signal
6 from scipy.optimize import curve_fit
7 from scipy.fft import fft, ifft
```

```
from scipy.stats import norm, median_abs_deviation
import pywt
import corner
import warnings
warnings.filterwarnings("ignore", "Wswiglal-redir-stdio")
import lal
import lalsimulation as ls
import time
from multiprocessing import Pool
from itertools import repeat
#ad hoc functions and initialization
import toymod_func as tf
import toymod_init as tin
from toymod_sig import *

def fvst_func(ch_hrss, scalog=None):

    if tin.sig_gen=='ll' or tin.sig_gen=='lh':
        tin.hrss*=1e-22
        tin.hrss_low*=1e-22
        tin.hrss_high*=1e-22
    elif tin.sig_gen=='v':
        tin.hrss*=1e-21
        tin.hrss_low*=1e-21
        tin.hrss_high*=1e-21

    hrss_arr=np.linspace(tin.hrss_low,tin.hrss_high,num=tin.nSNRs)
    optSNR=np.zeros(len(hrss_arr))

    fs=tin.fs # sampling frequency
    delta_t=tin.delta_t # sampling interval
    duration=tin.duration # SGB duration
    f0 = tin.f0 # SGB central frequency
    Q = 2.0 * np.pi * f0 * duration # SGB's Q-value

    thr_arr=np.linspace(0,7,14)
    print(thr_arr)
    nsigs=1000#1000
    match_arr=np.zeros(nsigs)
    mean_arr=np.zeros((tin.nSNRs,len(thr_arr)))
    std_arr=np.zeros((tin.nSNRs,len(thr_arr)))

    h_count=0
    for hrss in hrss_arr:
        args_list=sig_arg(tin.sig_gen,Q,hrss)
        hplus, hcross = sig_form(tin.sig_gen, *args_list)
        nplus=len(hplus)
        del args_list
        noisy_sig_arr=np.zeros((nsigs,nplus))  #rng.normal(0,1,[tin.nloops,nplus])
        phi=rng.uniform(-0.5*np.pi,0.5*np.pi) # polarization angle
        print('def sigs')
        for i in range(0,nsigs):
            if i%10==0:
                print( i, end=',')
            if i==0:
                clean_sig,noisy_sig_arr[i,:] = sig_generator(tin.noise_mod,hplus,
    hcross, phi)
            else:
                _,noisy_sig_arr[i,:] = sig_generator(tin.noise_mod,hplus,hcross,
    phi)
        print('\n')
        opt_snr=tf.SNR(clean_sig,nplus)
```

```
68          optSNR[h_count]=(opt_snr)
69
70          for l in range(0,len(thr_arr)):
71              clean_rec=tf.burst_rec(clean_sig, thr_arr[l])
72              with Pool(tin.n_th) as pool:
73                  noisy_rec_arr=np.asarray(pool.starmap(tf.burst_rec, zip(
    noisy_sig_arr, repeat(thr_arr[l]))))
74              for i in range(0,nsigs):
75                  match_arr[i]=tf.match(clean_sig,noisy_rec_arr[i])
76              mean_arr[h_count,l]=np.mean(match_arr)
77              std_arr[h_count,l]=np.std(match_arr)
78
79          print('means=',mean_arr[h_count,:],'\n')
80          print('stds=',std_arr[h_count,:],'\n')
81          h_count+=1
82      mean_arr=mean_arr.T
83      std_arr=std_arr.T
84
85      dx2=(optSNR[1]-optSNR[0])/2.0
86      dy2=(thr_arr[1]-thr_arr[0])/2.0
87
88      me = plt.imshow(std_arr, vmin=np.min(std_arr), vmax=np.max(std_arr),origin='
    lower', extent=[optSNR[0]-dx2,optSNR[-1]+dx2,thr_arr[0]-dy2,thr_arr[-1]+dy2],
    aspect='auto',cmap='plasma')
89      plt.xlabel('SNR')
90      plt.ylabel('thresholds')
91      plt.colorbar(me,label='std match')
92      plt.savefig(tin.save_path+'th_sigs_'+tin.ch_wlt+'.pdf')
93      tf.del_graph()
94
95      me = plt.imshow(mean_arr, vmin=np.min(mean_arr), vmax=np.max(mean_arr),origin=
    'lower',extent=[optSNR[0]-dx2,optSNR[-1]+dx2,thr_arr[0]-dy2,thr_arr[-1]+dy2],
    aspect='auto',cmap='plasma')
96      plt.xlabel('SNR')
97      plt.ylabel('thresholds')
98      plt.colorbar(me,label='mean match')
99      plt.savefig(tin.save_path+'th_means_'+tin.ch_wlt+'.pdf')
100     tf.del_graph()
101     return(thr_arr)
102
103 """-----------------------------------------------------------------"""
104 # getting the start time
105 init=time.time()
106
107 #graphics initialization
108 tf.graph_init(100,100,14,(10,6))
109
110 best_th=fvst_func(tin.hrss)
111
112 #how long everything took to run
113 end=time.time()
114 print('it took me ', (end-init)/60., 'minutes to run the program')
```

Listing 5: fvst.py

B Acknowledgements

This research has made use of data or software obtained from the Gravitational Wave Open Science Center (gwosc.org), a service of the LIGO Scientific Collaboration, the Virgo Collaboration, and KAGRA. This material is based upon work supported by NSF's LIGO Laboratory which is a major facility fully funded by the National Science Foundation, as well as the Science and Technology Facilities Council (STFC) of the United Kingdom, the Max-Planck-Society (MPS), and the State of Niedersachsen/Germany for support of the construction of Advanced LIGO and construction and operation of the GEO600 detector. Additional support for Advanced LIGO was provided by the Australian Research Council. Virgo is funded, through the European Gravitational Observatory (EGO), by the French Centre National de Recherche Scientifique (CNRS), the Italian Istituto Nazionale di Fisica Nucleare (INFN) and the Dutch Nikhef, with contributions by institutions from Belgium, Germany, Greece, Hungary, Ireland, Japan, Monaco, Poland, Portugal, Spain. KAGRA is supported by Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan Society for the Promotion of Science (JSPS) in Japan; National Research Foundation (NRF) and Ministry of Science and ICT (MSIT) in Korea; Academia Sinica (AS) and National Science and Technology Council (NSTC) in Taiwan.

References

[1] B. P. Abbott et al. "Observation of Gravitational Waves from a Binary Black Hole Merger". In: *Phys. Rev. Lett.* 116 (6 2016), p. 061102. DOI: 10.1103/PhysRevLett.116.061102. URL: https://link.aps.org/doi/10.1103/PhysRevLett.116.061102.

[2] T. Rothman. "The Secret History of Gravitational Waves". In: *American Scientist* 106 (Jan. 2018), p. 96. DOI: 10.1511/2018.106.2.96.

[3] D. Kennefick. *Traveling at the speed of thought: Einstein and the quest for Gravitational waves*. Published by Princeton University Press, 2007, 319 pages, Jan. 2007.

[4] R. A. Hulse and J. H. Taylor. "Discovery of a pulsar in a binary system". In: *Astrophysical Journal, vol. 195, Jan. 15, 1975, pt. 2, p. L51-L53.* 195 (1975), pp. L51–L53.

[5] J. Weber. "Gravitational Radiation". In: *Phys. Rev. Lett.* 18 (13 1967), pp. 498–501. DOI: 10.1103/PhysRevLett.18.498. URL: https://link.aps.org/doi/10.1103/PhysRevLett.18.498.

[6] C. W. Misner et al. *Gravitation*. Macmillan, 1973.

[7] M. Maggiore. *Gravitational Waves: Volume 1: Theory and Experiments*. Oxford University Press, Oct. 2007. ISBN: 9780198570745. DOI: 10.1093/acprof:oso/9780198570745.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780198570745.001.0001.

[8] A. Einstein. "Über gravitationswellen". In: *Sitzungsber. Preuss. Akad. Wiss. Berlin* 154 (1918), p. 1918.

[9] J. Weber. "Detection and generation of gravitational waves". In: *Physical Review* 117.1 (1960), p. 306.

[10] R. Weiss and D. Muehlner. *Electronically coupled broadband gravitational antenna*. Citeseer, 1972.

[11] E. D. Black and R. N. Gutenkunst. "An introduction to signal extraction in interferometric gravitational wave detectors". In: *American Journal of Physics* 71.4 (2003), pp. 365–378.

[12] F. Acernese et al. "Advanced Virgo: a second-generation interferometric gravitational wave detector". In: *Classical and Quantum Gravity* 32.2 (2014), p. 024001.

[13] T Akutsu et al. "Overview of KAGRA: Detector design and construction history". In: *Progress of Theoretical and Experimental Physics* 2021.5 (2021), 05A101.

[14] J. Aasi et al. "Advanced ligo". In: *Classical and quantum gravity* 32.7 (2015), p. 074001.

[15] M Saleem et al. "The science case for LIGO-India". In: *Classical and Quantum Gravity* 39.2 (2021), p. 025004.

[16] P. R. Saulson. *Fundamentals of Interferometric Gravitational Wave Detectors*. 2nd. WORLD SCIENTIFIC, 2017. DOI: 10.1142/10116. eprint: https://www.worldscientific.com/doi/pdf/10.1142/10116. URL: https://www.worldscientific.com/doi/abs/10.1142/10116.

[17] M. Evans et al. *LIGO Document T1500293-v13 Unofficial sensitivity curves (ASD) for aLIGO, Kagra, Virgo, Voyager, Cosmic Explorer, and Einstein Telescope*. URL: https://dcc.ligo.org/LIGO-T1500293/public.

[18] M. Solar et al. "Binary progenitor systems for Type Ic supernovae". In: *Nature Communications* 15.1 (Sept. 2024). ISSN: 2041-1723. DOI: 10.1038/s41467-024-51863-z. URL: http://dx.doi.org/10.1038/s41467-024-51863-z.

[19] L. Dessart et al. "On the nature of supernovae Ib and Ic: Radiative transfer of SN Ib/Ic ejecta". In: *Monthly Notices of the Royal Astronomical Society* 424.3 (July 2012), 2139–2159. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2012.21374.x. URL: http://dx.doi.org/10.1111/j.1365-2966.2012.21374.x.

[20] I. Iben Jr. and A. Tutukov. "Supernovae of type I as end products of the evolution of binaries with components of moderate initial mass." In: *The Astrophysical Journal, Supplements* 54 (Feb. 1984), pp. 335–372. DOI: 10.1086/190932.

[21] Korol, Valeriya et al. "Expected insights into Type Ia supernovae from LISA's gravitational wave observations". In: *Astronomy & Astrophysics* 691 (2024), A44. DOI: 10.1051/0004-6361/202451380. URL: https://doi.org/10.1051/0004-6361/202451380.

[22] M. S. University. *SOUNDS OF SPACETIME*. 2016. URL: https://www.soundsofspacetime.org/about-gw-sounds.html (visited on 03/04/2025).

[23] A. Nitz et al. *gwastro/pycbc: v2.3.3 release of PyCBC*. Version v2.3.3. Jan. 2024. DOI: 10.5281/zenodo.10473621. URL: https://doi.org/10.5281/zenodo.10473621.

[24] Rich Abbott et al. "Open data from the first and second observing runs of Advanced LIGO and Advanced Virgo". In: *SoftwareX* 13 (Jan. 2021), p. 100658. ISSN: 2352-7110. DOI: 10.1016/j.softx.2021.100658. URL: http://dx.doi.org/10.1016/j.softx.2021.100658.

[25] K. Arun et al. “New horizons for fundamental physics with LISA”. In: *Living Reviews in Relativity* 25 (June 2022). DOI: 10.1007/s41114-022-00036-9.

[26] N. Christensen and R. Meyer. “Parameter estimation with gravitational waves”. In: *Reviews of Modern Physics* 94.2 (Apr. 2022). ISSN: 1539-0756. DOI: 10.1103/revmodphys.94.025001. URL: http://dx.doi.org/10.1103/RevModPhys.94.025001.

[27] S. Klimenko et al. “Localization of gravitational wave sources with networks of advanced detectors”. In: *Physical Review D* 83.10 (May 2011). ISSN: 1550-2368. DOI: 10.1103/physrevd.83.102001. URL: http://dx.doi.org/10.1103/PhysRevD.83.102001.

[28] R. Margutti and R. Chornock. “First Multimessenger Observations of a Neutron Star Merger”. In: *Annual Review of Astronomy and Astrophysics* 59.1 (Sept. 2021), 155–202. ISSN: 1545-4282. DOI: 10.1146/annurev-astro-112420-030742. URL: http://dx.doi.org/10.1146/annurev-astro-112420-030742.

[29] P. Addison. *The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science, Engineering, Medicine and Finance, Second Edition (2nd ed.)* CRC Press, 2016. DOI: 10.1201/9781315372556.

[30] L. C. W. W. Group et al. *Waveform Modelling for the Laser Interferometer Space Antenna*. 2023. arXiv: 2311.01300 [gr-qc]. URL: https://arxiv.org/abs/2311.01300.

[31] A. Gupta et al. “Possible causes of false general relativity violations in gravitational wave observations”. In: *SciPost Physics Community Reports* (Feb. 2025). DOI: 10.21468/scipostphyscommrep.5. URL: http://dx.doi.org/10.21468/SciPostPhysCommRep.5.

[32] H. Mathur et al. “An analysis of the LIGO discovery based on introductory physics”. In: *American Journal of Physics* 85.9 (Sept. 2017), 676–682. ISSN: 1943-2909. DOI: 10.1119/1.4985727. URL: http://dx.doi.org/10.1119/1.4985727.

[33] B. P. Abbott et al. “The basic physics of the binary black hole merger GW150914”. In: *Annalen der Physik* 529.1–2 (Oct. 2016). ISSN: 1521-3889. DOI: 10.1002/andp.201600209. URL: http://dx.doi.org/10.1002/andp.201600209.

[34] C. M. Will. “On the unreasonable effectiveness of the post-Newtonian approximation in gravitational physics”. In: *Proceedings of the National Academy of Sciences* 108.15 (Mar. 2011), 5938–5945. ISSN: 1091-6490. DOI: 10.1073/pnas.1103127108. URL: http://dx.doi.org/10.1073/pnas.1103127108.

[35] L. Santamaría et al. "Matching post-Newtonian and numerical relativity waveforms: Systematic errors and a new phenomenological model for non-precessing black hole binaries". In: *Physical Review D* 82.6 (Sept. 2010). ISSN: 1550-2368. DOI: 10.1103/physrevd.82.064016. URL: http://dx.doi.org/10.1103/PhysRevD.82.064016.

[36] A. H. Mroué et al. "Catalog of 174 Binary Black Hole Simulations for Gravitational Wave Astronomy". In: *Phys. Rev. Lett.* 111 (24 2013), p. 241104. DOI: 10.1103/PhysRevLett.111.241104. URL: https://link.aps.org/doi/10.1103/PhysRevLett.111.241104.

[37] M. Boyle et al. "The SXS collaboration catalog of binary black hole simulations". In: *Classical and Quantum Gravity* 36.19 (Sept. 2019), p. 195006. ISSN: 1361-6382. DOI: 10.1088/1361-6382/ab34e2. URL: http://dx.doi.org/10.1088/1361-6382/ab34e2.

[38] T. DAMOUR. "INTRODUCTORY LECTURES ON THE EFFECTIVE ONE BODY FORMALISM". In: *International Journal of Modern Physics A* 23.08 (Mar. 2008), 1130–1148. ISSN: 1793-656X. DOI: 10.1142/s0217751x08039992. URL: http://dx.doi.org/10.1142/S0217751X08039992.

[39] N. J. Cornish and T. B. Littenberg. "Bayeswave: Bayesian inference for gravitational wave bursts and instrument glitches". In: *Classical and Quantum Gravity* 32.13 (June 2015), p. 135012. ISSN: 1361-6382. DOI: 10.1088/0264-9381/32/13/135012. URL: http://dx.doi.org/10.1088/0264-9381/32/13/135012.

[40] S. Klimenko et al. "Method for detection and reconstruction of gravitational wave transients with networks of advanced detectors". In: *Physical Review D* 93.4 (Feb. 2016). ISSN: 2470-0029. DOI: 10.1103/physrevd.93.042004. URL: http://dx.doi.org/10.1103/PhysRevD.93.042004.

[41] S. Ghonge et al. *Assessing and Mitigating the Impact of Glitches on Gravitational-Wave Parameter Estimation: a Model Agnostic Approach*. 2024. arXiv: 2311.09159 [gr-qc]. URL: https://arxiv.org/abs/2311.09159.

[42] R. Schoot et al. "Bayesian statistics and modelling". In: *Nature Reviews Methods Primers* 1 (Dec. 2021). DOI: 10.1038/s43586-020-00001-2.

[43] B. Walsh. *Markov Chain Monte Carlo and Gibbs Sampling*. 2004. URL: http://nitro.biosci.arizona.edu/courses/EEB581-2004/handouts/Gibbs.pdf.

[44] D. B. Dunson and J. E. Johndrow. "The Hastings algorithm at fifty". In: *Biometrika* 107.1 (Dec. 2019), pp. 1–23. ISSN: 0006-3444. DOI: 10.1093/biomet/asz066. eprint: https://academic.oup.com/biomet/article-pdf/107/1/1/32450930/asz066.pdf. URL: https://doi.org/10.1093/biomet/asz066.

[45] C. P. Robert and W. Changye. *Markov Chain Monte Carlo Methods, a survey with some frequent misunderstandings*. 2020. arXiv: 2001.06249 [stat.CO]. URL: https://arxiv.org/abs/2001.06249.

[46] D. W. Hogg and D. Foreman-Mackey. "Data Analysis Recipes: Using Markov Chain Monte Carlo". In: *The Astrophysical Journal Supplement Series* 236.1 (May 2018), p. 11. ISSN: 1538-4365. DOI: 10.3847/1538-4365/aab76e. URL: http://dx.doi.org/10.3847/1538-4365/aab76e.

[47] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992. DOI: 10.1137/1.9781611970104. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970104. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970104.

[48] J. Morlet et al. "Wave propagation and sampling theory—Part II: Sampling theory and complex waves". In: *GEOPHYSICS* 47.2 (1982), pp. 222–236. DOI: 10.1190/1.1441329. URL: https://doi.org/10.1190/1.1441329.

[49] H. Yang et al. "Nonlinear adaptive wavelet analysis of electrocardiogram signals". In: *Phys. Rev. E* 76 (2 2007), p. 026214. DOI: 10.1103/PhysRevE.76.026214. URL: https://link.aps.org/doi/10.1103/PhysRevE.76.026214.

[50] E. Foufoula-Georgiou and P. Kumar. "Wavelet Analysis in Geophysics: An Introduction". In: *Wavelets in Geophysics*. Ed. by E. Foufoula-Georgiou and P. Kumar. Vol. 4. Wavelet Analysis and Its Applications. Academic Press, 1994, pp. 1–43. DOI: https://doi.org/10.1016/B978-0-08-052087-2.50007-4. URL: https://www.sciencedirect.com/science/article/pii/B9780080520872500074.

[51] D Davis et al. "LIGO detector characterization in the second and third observing runs". In: *Classical and Quantum Gravity* 38.13 (June 2021), p. 135014. ISSN: 1361-6382. DOI: 10.1088/1361-6382/abfd85. URL: http://dx.doi.org/10.1088/1361-6382/abfd85.

[52] B. K. Berger et al. "Searching for the causes of anomalous Advanced LIGO noise". In: *Applied Physics Letters* 122.18 (May 2023), p. 184101. ISSN: 0003-6951. DOI: 10.1063/5.0140766. eprint: https://pubs.aip.org/aip/apl/article-pdf/doi/10.1063/5.0140766/18009707/184101_1_5.0140766.pdf. URL: https://doi.org/10.1063/5.0140766.

[53] B. P. Abbott et al. “GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral”. In: *Phys. Rev. Lett.* 119 (16 2017), p. 161101. DOI: 10.1103/PhysRevLett.119.161101. URL: https://link.aps.org/doi/10.1103/PhysRevLett.119.161101.

[54] J. Timmer and M. König. “On generating power law noise.” In: *Astronomy and Astrophysics,* 300 (Aug. 1995), p. 707.

[55] C. R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[56] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[57] LIGO Scientific Collaboration et al. *LVK Algorithm Library - LALSuite*. Free software (GPL). 2018. DOI: 10.7935/GT1W-FZ16.

[58] G. R. Lee et al. “PyWavelets: A Python package for wavelet analysis”. In: *Journal of Open Source Software* 4.36 (2019), p. 1237. DOI: 10.21105/joss.01237. URL: https://doi.org/10.21105/joss.01237.

[59] D. Foreman-Mackey et al. “emcee: The MCMC Hammer”. In: *Publications of the Astronomical Society of the Pacific* 125.925 (Mar. 2013), p. 306. DOI: 10.1086/670067. arXiv: 1202.3665 [astro-ph.IM].

[60] P. Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[61] D. Foreman-Mackey. “corner.py: Scatterplot matrices in Python”. In: *The Journal of Open Source Software* 1.2 (2016), p. 24. DOI: 10.21105/joss.00024. URL: https://doi.org/10.21105/joss.00024.

[62] K. Wette. “SWIGLAL: Python and Octave interfaces to the LALSuite gravitational-wave data analysis libraries”. In: *SoftwareX* 12 (2020), p. 100634. DOI: 10.1016/j.softx.2020.100634.

[63] S. Seabold and J. Perktold. “statsmodels: Econometric and statistical modeling with python”. In: *9th Python in Science Conference*. 2010.