Università degli Studi di Trieste

Dipartimento di Fisica

Corso di Laurea Magistrale in Fisica

Towards a measurement of gravitational wave polarization with LIGO/Virgo data

Author:
Andrea Virtuoso

Supervisor:
Prof. Edoardo Milotti

Academic Year 2019/2020

Università degli Studi di Trieste

Dipartimento di Fisica

Corso di Laurea Magistrale in Fisica

Verso una misurazione delle polarizzazioni associate alle onde gravitazionali con i dati LIGO-Virgo

Candidato:
Andrea Virtuoso

Relatore:
Prof. Edoardo Milotti

Anno Accademico 2019/2020

Risplende la tua luce nel buio della via
Non so di dove vieni e neppure chi tu sia
Sembri così vicina e sei tanto lontana
Non conosco il tuo nome, so solo che sei bella
e dovunque ti trovi e chiunque tu sia
scintilla scintilla piccola stella.

Da un'antica nenia irlandese

Abstract

Gravitational Waves (GWs) are one of the most interesting predictions of the Theory of General Relativity. Just as electromagnetic waves are produced by accelerated charges, GWs are produced by accelerated mass quadrupoles. They travel practically undisturbed by intervening objects (apart from gravitational lensing effects) and their detection gives us a new powerful channel to explore and understand our Universe. It takes huge masses and large quadrupole changes to generate measurable GWs, and since the objects that can do so are exceedingly rare, the GWs that reach Earth come from cosmological distances, and even when the energy dissipated in GWs is very large, the signals that we receive are extremely small. For this reason, GWs were not observed for a hundred years after their theoretical prediction, until 14 September 2015, when the LIGO (Laser Interferometer Gravitational Wave Observatory) interferometers detected a GW signal from a binary black hole (BBH), marking the beginning of the GW observations era.

General Relativity predicts that GWs propagate with two distinct polarizations: the interferometers detect a linear combination of these, which lives on top of the usual noise of the detectors. This Thesis consists in a detailed study of some techniques developed for extracting the GW polarizations from interferometer data.

The structure of the Thesis is as follows: Chapter 1 starts with a short derivation of GWs in General Relativity, followed by a description of how GWs affect the interferometer and so how GWs can be detected; finally, in this Chapter the main GW sources are discussed.

In Chapter 2 there is a brief description of the GW interferometers, which is followed by an analysis of the main noise sources that affect detection.

In Chapter 3 I give an overview of some important algorithms developed for GW polarization disentangling: this discussion starts from the first solutions (developed in 1980s) and ends with the description of the coherent WaveBurst (cWB) analysis pipeline and of the main methods implemented in it.

In Chapter 4 I test with simulations the polarization disentangling ability of the algorithms described in Chapter 3, introducing also a new data denoising technique involving wavelets; being the identification of the source location in the sky a key passage for reconstructing the polarization content of a detected GW signal, there is also a short Section in which I test the method implemented in cWB for finding the source location.

The Conclusions are discussed in Chapter 5.

Riassunto

Le onde gravitazionali rappresentano una delle predizioni più interessanti della Teoria della Relatività Generale. Così come le onde elettromagnetiche sono prodotte da cariche accelerate, le onde gravitazionali sono generate dall'accelerazione di quadrupoli di massa. Esse viaggiano praticamente senza essere disturbate da oggetti intermedi (eccetto per effetti di lente gravitazionale) e la loro rivelazione ci offre un nuovo, potente canale per esplorare e comprendere il nostro Universo. Ci vogliono masse enormi e grosse variazioni del momento di quadrupolo per generare onde gravitazionali che possano essere misurate, e visto che gli oggetti che possono fare ciò sono piuttosto rari, le onde gravitazionali che raggiungono la Terra provengono da distanze cosmologiche e anche se l'energia dissipata in radiazione gravitazionale è molto grande i segnali che riceviamo sono molto, molto piccoli. Questa è la ragione per cui le onde gravitazionali non sono state misurate fino al 14 Settembre 2015, circa cent'anni dopo la loro predizione teorica, quando gli interferometri LIGO (Laser Interferometer Gravitational Wave Observatory) hanno rivelato un segnale da un sistema binario di buchi neri, dando inizio all'era delle osservazioni tramite onde gravitazionali.

La Relatività Generale predice che le onde gravitazionali si propaghino con due distinte polarizzazioni: quello che viene rivelato negli interferometri è una combinazione lineare di queste, alla quale si aggiunge l'inevitabile rumore dei rivelatori. Questa Tesi consiste dunque in uno studio dettagliato di alcune tecniche sviluppate per estrarre le polarizzazioni associate ad un'onda gravitazionale il cui segnale sia stato rivelato dagli interferometri.

La Tesi si struttura come segue: il Capitolo 1 comincia con una breve derivazione delle onde gravitazionali a partire dalla Teoria della Relatività Generale, seguita da una descrizione di come queste onde agiscono sugli interferometri e dunque di come possano essere rivelate; alla fine del Capitolo vengono discusse le principali sorgenti delle onde.

Nel Capitolo 2 è riportata una breve descrizione degli interferometri, seguita da un'analisi delle principali sorgenti di rumore che influiscono sulla rivelazione.

Il Capitolo 3 consiste in una discussione di alcuni importanti algoritmi sviluppati nel corso del tempo per ricavare le polarizzazioni di un'onda gravitazionale a partire dal segnale rivelato dagli interferometri: questa trattazione parte dalle prime soluzioni sviluppate negli anni Ottanta per concludere con la descrizione del software di analisi coherent WaveBurst e delle principali tecniche implementate in questo.

Nel Capitolo 4 viene testata tramite l'uso di simulazioni l'abilità degli algoritmi precedentemente descritti di ricostruire le polarizzazioni di un segnale rivelato, introducendo inoltre una nuova tecnica di eliminazione del rumore dai dati tramite wavelets; essendo inoltre la ricostruzione delle polarizzazioni di un'onda gravitazionale dipendente anche dalla capacità di ricostruire correttamente la direzione nel cielo associata alla sorgente dell'onda, viene riporata una breve sezione in cui testo i metodi implementati all'interno di coherent WaveBurst riguardo all'individuazione della direzione associata alla sorgente.

Le Conclusioni vengono infine discusse nel Capitolo 5.

Contents

Chapter 1

Overview of Gravitational Waves

1.1 From the Einstein Equations to Gravitational Waves

In 1916, about one year after the publication of his Theory of General Relativity, Albert Einstein proposed a solution of his equations by linearizing them in the weak field limit: exploiting the similarity with the equations of electrodynamics, he found that in dynamical cases these equations have wave solutions, thus predicting the existence of GW about one hundred years before their first detection with ground-based detectors [1].
In this Section I provide a brief derivation, highlighting the main hypothesis and the key passages that lead to GWs.
The starting point are the well-known Einstein equations [2]:

$$R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R = \frac{8\pi G}{c^4} T_{\mu\nu}, \tag{1.1}$$

where $g_{\mu\nu}$ is the metric tensor (or, more simply, the metric), $R_{\mu\nu}$ is the Ricci tensor, R is the Ricci scalar and $T_{\mu\nu}$ is the energy momentum tensor. These equations link the space-time curvature (on the LHS) with the energy and momentum within this (on the RHS).
In General Relativity (GR), the description of space-time curvature starts with the Riemann curvature tensor, which can be expressed as a function of metric tensor derivatives with respect to space-time coordinates[1]

$$\{R^{\alpha}\}_{\beta\mu\nu} = \frac{1}{2} g^{\alpha\sigma} \left(g_{\sigma\nu,\beta\mu} - g_{\beta\nu,\sigma\mu} - g_{\sigma\mu,\beta\nu} + g_{\beta\mu,\sigma\nu} \right), \tag{1.2}$$

with

$$g_{\alpha\beta,\mu\nu} = \frac{\partial}{\partial x_{\mu}} \frac{\partial}{\partial x_{\nu}} g_{\alpha\beta}. \tag{1.3}$$

The Ricci tensor can be obtained from the Riemann curvature tensor as

$$R_{\alpha\beta} = \{R^{\mu}\}_{\alpha\mu\beta}, \tag{1.4}$$

next, the Ricci scalar is

$$R = g^{\alpha\beta} R_{\alpha\beta}. \tag{1.5}$$

[1] A more extensive derivation involving Christoffel symbols can be found, among the others, in [3]. From now on, the Einstein's convention on indexes is assumed.

The wave equations for the strain tensor $h_{\mu\nu}$ are obtained from the Einstein equations in the weak field limit. Then, the perturbation of the Minkowski metric (flat space) is

$$g_{\mu\nu} = \eta_{\mu\nu} + h_{\mu\nu} \quad \text{where} \quad \left|h_{\mu\nu}\right| << 1, \tag{1.6}$$

where the equation of motion is expanded to first order in $h_{\mu\nu}$.

It is useful to define the following tensor (*trace reverse*)

$$\bar{h}_{\mu\nu} = h_{\mu\nu} - \frac{1}{2}\eta_{\mu\nu}h \quad \text{where} \quad h = \eta^{\mu\nu}h_{\mu\nu}, \tag{1.7}$$

so that the Einstein equations can be written as

$$\Box\bar{h}_{\mu\nu} + \eta_{\mu\nu}\partial^\rho\partial^\sigma\bar{h}_{\rho\sigma} - \partial^\rho\partial_\nu\bar{h}_{\mu\rho} - \partial^\rho\partial_\mu\bar{h}_{\nu\rho} = -\frac{16\pi G}{c^4}T_{\mu\nu}, \tag{1.8}$$

where $\Box = \partial^\mu\partial_\mu$ is the d'Alambertian.

Equation (1.8) is simpler when we assume the *Lorenz gauge*[2] so that

$$\partial^\nu\bar{h}_{\mu\nu} = 0, \tag{1.9}$$

and eq. (1.8) assumes the usual form of a wave equation

$$\Box\bar{h}_{\mu\nu} = -\frac{16\pi G}{c^4}T_{\mu\nu}, \tag{1.10}$$

and the source-free version

$$\Box\bar{h}_{\mu\nu} = 0 \tag{1.11}$$

The metric can be further simplified since eq. (1.9) does not fix the gauge completely. After carrying out the coordinate transformation

$$x_\mu \to x_\mu + \xi_\mu \quad \text{with} \quad \Box\xi_\mu = 0, \tag{1.12}$$

we find that the $h_{\mu\nu}$ tensor transforms as

$$h_{\mu\nu}(x) \to h'_{\mu\nu}(x') = h_{\mu\nu}(x) - (\partial_\mu\xi^\nu + \partial_\nu\xi^\mu), \tag{1.13}$$

Now, from $\Box\xi^\mu = 0$ it follows that the functions ξ_μ can be chosen to set the conditions

$$h^{0\mu} = 0, \quad h^i_i = 0, \quad \partial^j h_{ij} = 0, \tag{1.14}$$

in order to fix the so called *transverse-traceless gauge*, or TT gauge[3].

In this gauge, with the assumption that waves propagate along the *z-axis*, the solution of eq. (1.11), takes the form

$$h^{\mathrm{TT}}_{ij}(t,z) = \begin{pmatrix} h_+ & h_\times & 0 \\ h_\times & -h_+ & 0 \\ 0 & 0 & 0 \end{pmatrix}_{ij} \cos\left[\omega(t - z/c)\right], \tag{1.15}$$

[2]The Lorenz gauge is defined by $\partial_\mu(g^{\mu\nu}\sqrt{-g}) = 0$: the name stems from the analogy with the Lorentz gauge of electromagnetism $\partial_\mu A^\mu = 0$.

[3]From this it also follows that $\bar{h} = 0$ and so $\bar{h}_{\mu\nu} = h_{\mu\nu}$.

or the even simpler form

$$h_{ab}^{TT}(t,z) = \begin{pmatrix} h_+ & h_\times \\ h_\times & -h_+ \end{pmatrix}_{ab} cos[\omega(t-z/c)], \tag{1.16}$$

with $a, b = \{1, 2\}$ indices in the plane (x, y).
The main result of this equation is that the wave solution displays two polarizations, denoted as *plus* and *cross*: this is a key result of GR which predicts only two tensor polarizations for GWs. Alternative theories (such as scalar theories, tensor theories, scalar-tensor theories, vector theories and others) admit up to six polarizations, namely (depending on the theory) two tensor modes, two vector modes and two scalar modes. However, observation gave no evidence of inconsistency between GR predictions and data, but further in-depth work is still ongoing [4].

An illustration of how the two tensor polarizations h_+ and $h_\times$ affect a ring of freely moving particles is shown in fig. 1.1.

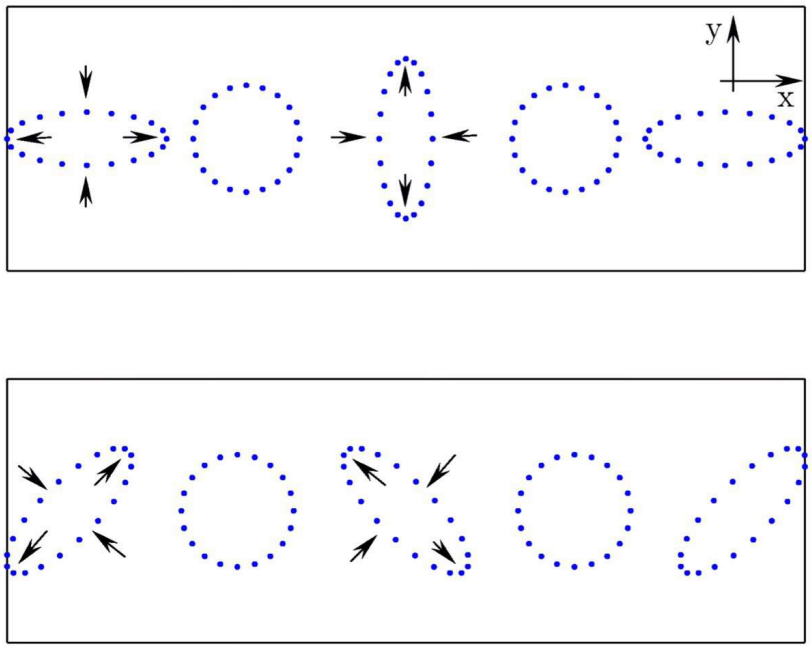


Figure 1.1: Demonstration of the effect of a GW propagating in the z direction on a ring of freely moving particles in the $x-y$ plane: the effects of the $+$ (upper panel) and $\times$ (lower panel) polarizations are shown (this figure is taken from [5]).

The wave solution can be written in a more general way by expanding it in plane waves. To this end I assume that the wave propagates in direction $\hat{\mathbf{n}}$; introducing two more unit vectors $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ orthogonal to $\hat{\mathbf{n}}$ the polarization tensor $e_{ij}^A(\hat{\mathbf{n}})$ $(A = +, \times)$ can be defined as

$$e_{ij}^+(\hat{\mathbf{n}}) = \hat{\mathbf{u}}_i\hat{\mathbf{u}}_j - \hat{\mathbf{v}}_i\hat{\mathbf{v}}_j; \qquad e_{ij}^\times(\hat{\mathbf{n}}) = \hat{\mathbf{u}}_i\hat{\mathbf{v}}_j + \hat{\mathbf{v}}_i\hat{\mathbf{u}}_j. \tag{1.17}$$

Then, the wave solution can be expanded into fixed-frequency plane waves as follows

$$h_{ij}(t,\mathbf{x}) = \sum_{A=+,\times} e_{ij}^A(\hat{\mathbf{n}}) \int_{-\infty}^{+\infty} df\, \tilde{h}_A(f) e^{-2\pi i f(t-\hat{\mathbf{n}}\cdot\mathbf{x}/c)}. \tag{1.18}$$

1.2 Antenna pattern functions

GWs are detected on the ground using a heavily modified type of Michelson interferometer. The detection is based on the differential strain induced by GWs on the interferometer arms which can be written as [6]

$$\tilde{\xi}(t) = [\delta L_x(t) - \delta L_y(t)]/L, \tag{1.19}$$

where L is the length of the interferometer arm and $\delta L_{x,y}$ are the length changes for the x and y arms (aligned with the coordinate axes). The definition of differential strain in equation (1.19) can be combined with the plane-wave expansion (1.18), and the resulting expression can be simplified making the following assumptions: first, the detector is located in $\mathbf{x} = 0$; second, the detector is sensitive to GWs with wavelength much longer than its size, that is $L \ll \lambda$. This implies that the GW amplitude doesn't vary significantly over the detector length, namely $h_{ij}(t, \mathbf{x}) \simeq h_{ij}(t, 0)$.
Then, eq. (1.18) can be simplified in the following way

$$h_{ij}(t) = \sum_{A=+,\times} e_{ij}^{A}(\hat{\mathbf{n}}) \int_{-\infty}^{+\infty} df\, \tilde{h}_A(f) e^{-2\pi i f t} = \sum_{A=+,\times} e_{ij}^{A}(\hat{\mathbf{n}}) h_A(t) \tag{1.20}$$

Assuming the detector to be a linear system, the differential strain is a linear function of the strain h, and is written as

$$\tilde{\xi}(t) = D^{ij} h_{ij}(t) = \sum_{A=+,\times} D^{ij} e_{ij}^{A}(\hat{\mathbf{n}}) h_A(t) \tag{1.21}$$

where D^{ij} is the detector tensor which depends on the directions $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ of the interferometer arms:

$$D_{ij} = \frac{1}{2}(\hat{\mathbf{x}}_i \hat{\mathbf{x}}_j - \hat{\mathbf{y}}_i \hat{\mathbf{y}}_j) \tag{1.22}$$

Now the detector antenna patterns can be defined as follows

$$F_A(\hat{\mathbf{n}}) = D^{ij} e_{ij}^{A}(\hat{\mathbf{n}}) \tag{1.23}$$

where $\hat{\mathbf{n}}$ is the unit vector associated with the direction given by the polar angles (θ, ϕ) in the detector frame (see fig. 1.2). With this definition, eq. (1.21) can be written as

$$\tilde{\xi}(t) = h_+(t) F_+(\theta, \phi) + h_\times(t) F_\times(\theta, \phi) \tag{1.24}$$

Eq. (1.28) is a key result: it describes how a GW interacts with an interferometer. The main purpose of this work is to find the unknowns h_+,$h_\times$,θ and ϕ starting from the detectors responses which are a combination of the strains induced by GWs and those produced by noise sources.
From geometrical considerations it can be shown that for an interferometer with arms along the x and y axes, and a wave propagating in the $\hat{\mathbf{n}}$ direction the antenna patterns are

$$F_+(\theta, \phi) = \frac{1}{2}(1 + \cos^2\theta)\cos 2\phi \tag{1.25}$$

$$F_\times(\theta, \phi) = \cos\theta \sin 2\phi \tag{1.26}$$

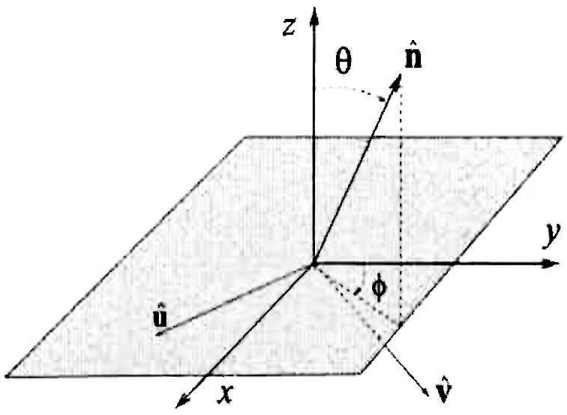


Figure 1.2: Angle convention for the single detector.

Note that the $(\hat{\mathbf{u}}, \hat{\mathbf{v}})$ axes in the transverse plane can be rotated by an angle ψ obtaining the axes $(\hat{\mathbf{u}}^{'}, \hat{\mathbf{v}}^{'})$ with the following linear transformation

$$\begin{aligned} \hat{\mathbf{u}}^{'} &= \hat{\mathbf{u}}\cos\psi - \hat{\mathbf{v}}\sin\psi \\ \hat{\mathbf{v}}^{'} &= \hat{\mathbf{u}}\sin\psi + \hat{\mathbf{v}}\cos\psi \end{aligned} \tag{1.27}$$

where ψ is the polarization angle.

This transformation acts on both the antenna patterns and the polarizations h_+ and $h_\times$

$$\begin{aligned} F_+^{'}(\theta,\phi,\psi) &= F_+(\theta,\phi)\cos 2\psi - F_\times(\theta,\phi)\sin 2\psi \\ F_\times^{'}(\theta,\phi,\psi) &= F_+(\theta,\phi)\sin 2\psi + F_\times(\theta,\phi)\cos 2\psi \end{aligned} \tag{1.28}$$

$$\begin{aligned} h_+^{'} &= h_+\cos 2\psi - h_\times \sin 2\psi \\ h_\times^{'} &= h_+\sin 2\psi + h_\times \cos 2\psi \end{aligned} \tag{1.29}$$

but the detector response, being the result of the scalar product $\tilde{\xi} = \mathbf{F}\cdot\mathbf{h}$ (where $\mathbf{F} = (F_+, F_\times)$, $\mathbf{h} = (h_+, h_\times)$) is invariant under the rotation in the transverse plane. This property will be used in Chapter 3 where the reconstruction of the polarization content of a detected signal $\mathbf{h}$ and $\mathbf{F}$ will be performed in a suitably rotated frame.

1.3 Gravitational wave sources

We expect to observe GWs from accelerating bodies without spherical or cylindrical symmetry [2]. Because of the very small strains induced in the space-time structure, the interferometers can detect only events emitting a huge amount of energy in GWs: e.g., in the first detected event GW150914[4] an energy of about $3M \odot c^2$ was dissipated as GWs [1].

[4]GW events are denoted with their detection date: e.g., GW150914 was detected on 14th September 2015.

Up to now only GWs from binary systems have been detected [7] [8], mainly from binary black holes (BBH). Some important detections were also made for binary neutron star (BNS) systems, such as the event GW170817 [9], while detections of GWs from BHNS during O3 still have to be confirmed [10].

There are several other sources that are expected to emit GWs. At the top of this list we predict to find the gravitational counterpart of a core-collapse supernovae (CCSN) [11] [12][13] as well as the stochastic GW background [14].

1.3.1 Compact binaries

The compact binaries studied by LIGO-Virgo include two-body bound systems where the components are either black holes or neutron stars. Initially, the components approach each other in spiralling orbits (inspiral phase). When the system reaches the last stable orbit (innermost stable circular orbit, ISCO), the components merge into a unique body (merger phase) [2] which corresponds to the peak of GW emission. In the last phase (ringdown) the merged body performs a few oscillations until it reaches a final equilibrium state.

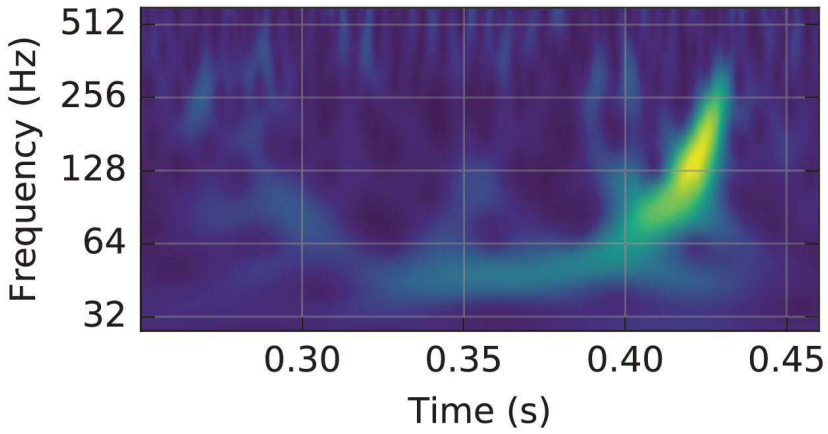


Figure 1.3: Spectrogram of the GW150914 event observed by the LIGO Hanford and LIGO Livingston detectors. The figure is taken from [1].

A simple and informative representation of this process is provided by spectrograms. In a spectrogram, a compact binary coalescence (CBC) takes the typical shape of a *chirp*[5] (fig. 1.3). In the inspiral phase, where relativistic effects are not dominant, the frequency behaviour can be described – to a good approximation – by the following equation derived

[5]Note that this representation is a close parallel of a musical score, with time on the x-axis and frequency in a log scale along the y-axis. Scientists have fun turning GWs into sounds: some examples can be found at `https://www.gw-openscience.org/audio/`

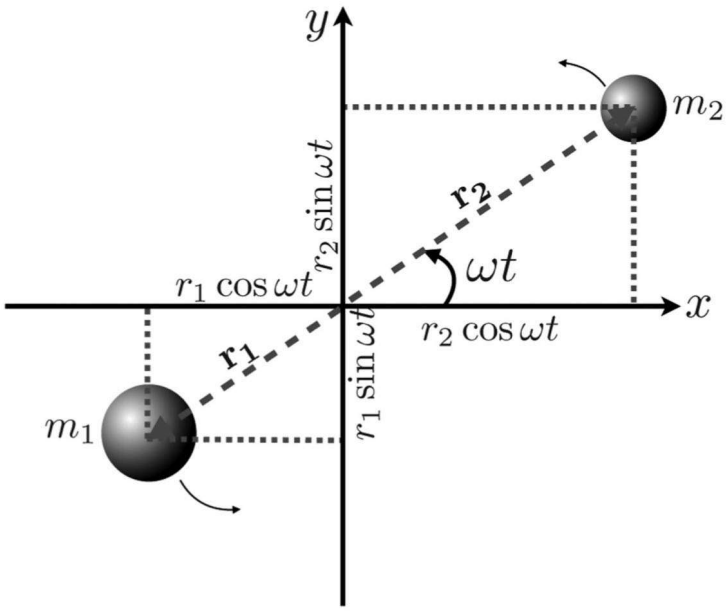


Figure 1.4: Geometry of a two-body system. The component masses are m_1 and m_2 and the orbit lies in the xy plane: the total mass $m_1 + m_2$ is denoted by M, while the reduced mass $m_1 m_2/M$ by μ. ω is equal to $2\pi f$ where f is the orbital frequency. The gravitational radiation is emitted with the twice the orbital frequency, namely $f_{GW} = 2f$.

from Newtonian dynamics [15]

$$\mathcal{M}_c = \frac{c^3}{G}\left(\left(\frac{5}{96}\right)^3 \pi^{-8} (f_{GW})^{-11} (\dot{f}_{GW})^3\right)^{1/5}, \tag{1.30}$$

where $\mathcal{M}_c$ is the chirp mass, a key parameter of the system, defined as

$$\mathcal{M}_c = \frac{(m_1 m_2)^{3/5}}{(m_1 + m_2)^{1/5}}. \tag{1.31}$$

The derivation of eq. (1.30) is straightforward, where the geometry of the system is as in fig. 1.4. The quadrupole moment Q_{ij} of the system's mass distribution is

$$Q_{ij} = \int d^3x \rho(\mathbf{x}) \left(x_i x_j - \frac{1}{3} r^2 \delta_{ij}\right) = \sum_{A \in (1,2)} m_A \begin{pmatrix} \frac{2}{3}x_A^2 - \frac{1}{3}y_A^2 & x_A y_A & 0 \\ x_A y_A & \frac{2}{3}x_A^2 - \frac{1}{3}y_A^2 & 0 \\ 0 & 0 & -\frac{1}{3}r_A^2 \end{pmatrix}, \tag{1.32}$$

where $\rho(\mathbf{x})$ is the mass density of the system and $r = r_1 + r_2$.
Using trigonometric considerations as in fig. 1.4, it is easy to find the connection between the quadrupole moment and the moment of inertia

$$Q_{ij}^A(t) = \frac{m_A r_A^2}{2} I_{ij}, \tag{1.33}$$

by combining the values $I_{xx} = m_A r_A^2[\cos(2\omega t)+1/3]$, $I_{yy} = m_A r_A^2[1/3-\sin(2\omega t)]$,$I_{xy} = I_{yx} = m_A r_A^2 \sin(2\omega t)$.
Using this formula we find the energy radiated as GWs

$$\frac{dE_{GW}}{dt} = \frac{1}{5}\frac{G}{c^5}\sum_{i,j=1}^{3}\frac{d^3Q_{ij}}{dt^3}\frac{d^3Q_{ij}}{dt^3} = \frac{32}{5}\frac{G}{c^5}\mu^2 r^4 \omega^6. \tag{1.34}$$

This radiated energy is compensated by the loss of orbital energy $E_{orb} = -GM\mu/2r$ and therefore[6]

$$\frac{dE_{GW}}{dt} = -\frac{dE_{orb}}{dt} = -\frac{GM\mu}{2r^2}\dot{r}. \tag{1.35}$$

By combining equations (1.34), (1.35) and using Kepler's third law $r^3 = GM/\omega^2$ we find the equation that links the orbital frequency ω and the chirp mass $\mathcal{M}_c$

$$\dot{\omega}^3 = \left(\frac{96}{5}\right)\omega^{11}\left(\frac{G\mathcal{M}_c}{c^3}\right)^5. \tag{1.36}$$

Eq. (1.36) is obviously the same for ω_{GW} and so with a simple rearrangement eq. (1.30) is obtained.
As mentioned above, the Newtonian approximation holds only when relativistic effects are negligible, i.e. in the first part of the inspiral phase. This is clearly shown in fig 1.5.

Moving on to a relativistic treatment, one finds the following expressions for h_+ and $h_\times$ during the inspiral phase[7] [2]:

$$h_+(t) = \frac{4}{r}\left(\frac{G\mathcal{M}_c}{c^2}\right)^{5/3}\left(\frac{\pi f_{GW}(t)}{c}\right)^{2/3}\left(\frac{1+\cos^2\iota}{2}\right)\cos[\Phi(t)] \tag{1.37}$$

$$h_\times(t) = \frac{4}{r}\left(\frac{G\mathcal{M}_c}{c^2}\right)^{5/3}\left(\frac{\pi f_{GW}(t)}{c}\right)^{2/3}\left(\frac{\cos\iota}{2}\right)\sin[\Phi(t)], \tag{1.38}$$

where r is the distance from the source while f_{GW} and Φ are

$$f_{GW}(t) = \frac{1}{\pi}\left(\frac{5}{256}\frac{1}{t_c-t}\right)^{3/8}\left(\frac{G\mathcal{M}_c}{c^3}\right)^{-5/8} \tag{1.39}$$

$$\Phi(t) = -2\left(\frac{5G\mathcal{M}_c}{c^3}\right)^{-5/8}(t_c-t)^{5/8} + \Phi_0, \tag{1.40}$$

and t_c is the coalescence time.

[6]Here it is assumed that each orbit is approximately Keplerian.

[7]The behaviour of the system during the merger phase is challenging, it is not possible to give an analytic solution and it requires numerical simulations. On the contrary, the ringdown process is well characterized by a sinusoidal exponential decay of the wave amplitude [6].

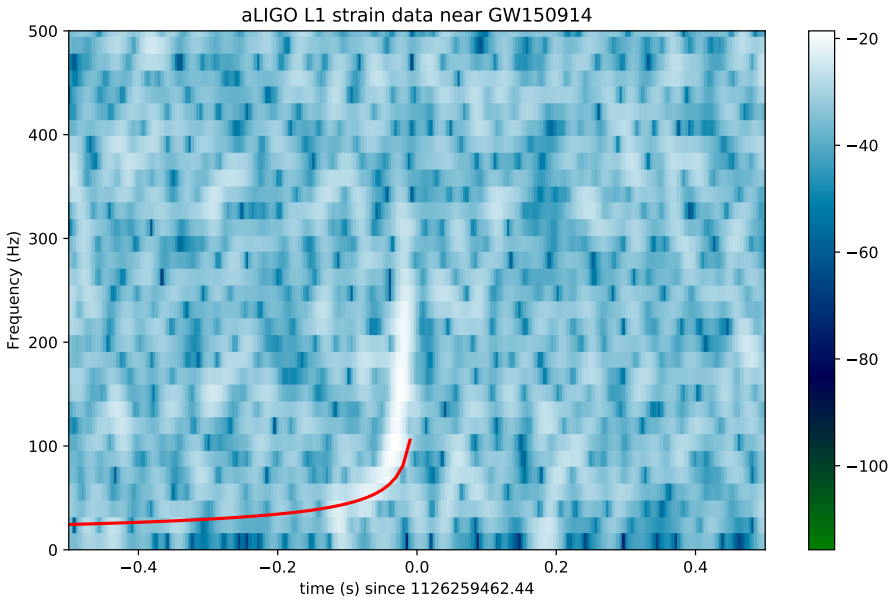


Figure 1.5: The Newtonian chirp (red line) of the GW150914 event. Here we see that the Newtonian formula properly describes the behaviour of the chirp only far from the merger, where relativistic effects are negligible.

Most observations of GWs are from binary black holes [7]. Their frequent detection is due to the high luminosity of these systems and to the well-known time-frequency dependence of the GW signals. While BBH detection is a key result of GW astrophysics, other detections may be even more impactful, like the one made on 17 August 2017, when the LIGO and Virgo interferometers detected a signal from a BNS merger [9]. The three-interferometers network was able to find the source location in the sky within a region of $31\,\mathrm{deg}^2$ and the position was promptly shared with the *Fermi* and *INTEGRAL* telescopes which in turn detected a short burst of γ rays associated with the event. This historic joint detection represents the beginning of the era of multimessenger astrophysics with GWs in which we will be able to study many phenomena with their gravitational counterpart: GWs are, and will be even more in the future with the new generation of detectors, a new channel for observing our Universe.

1.3.2 Core Collapse Supernovae

GWs are expected to be emitted from Core-collapse Supernovae (CCSNe) [11][12][13]. The rate for a galactic CCSN is expected to be about 3.2 per century and the models predict CCSNe to emit considerably less GW energy than compact binaries (a difference of about $7-8$ orders of magnitude [11]), so their detection is very challenging. Both modeled and minimally modeled searches [16] are carried out in this field. The problem of modeled searches is that there is not yet a well defined model for the gravitational emission from CCSNe, partly because CCSN simulations are computationally heavy. Thus, minimally

modeled searches could play a key role in this field [16]. The detection of a CCSN in this new era of multimessenger astronomy could greatly improve our knowledge on these phenomena: the simultaneous detection of electromagnetic waves, neutrinos and GWs will give us unprecedented information about the mechanisms underlying CCSNe and could lead us to better and more precise models.

1.3.3 Stochastic Gravitational Wave Background

The stochastic gravitational wave background (SGWB) is given by the superposition of a large number of independent sources [14]. The detection of GWs from compact binaries lends credence to an astrophysical background due to these sources: other sources of background could be supernovae and neutron stars.
SGWB is expected from cosmological sources also, as inflation and many others phenomena associated with the early Universe, including some "exotic" ones as cosmic strings. This kind of background is expected to be weaker than the astrophysical one, but its detection shall be one of the most important events in the history of science because it shall give us direct evidence of early Universe phenomena, which are inaccessible with EM radiation. Although the SGWB has not been detected yet, the detection of an astrophysical background is considered to be very likely in the coming years, while for the cosmological background we probably have to wait for the new generation of detectors, above all LISA.

1.4 Remarkable detections

Among the several detections made by LIGO-Virgo interferometers since the first detection in 2015, there are three events that must be highlighted for their importance.

The first one is of course the first GW detection, the GW150914 event. This a signal originated from a BBH involving two black holes with masses $35.6M_{\odot}$ and $30.6M_{\odot}$, and was detected by the two LIGO interferometers[8][1][7]. GW150914 is a landmark in the history of science: it is the first direct detection of GWs and the first demonstration of the existence of BBHs, and it marks the beginning of a new era for astronomical observations: GWs are a new window on our Universe, a novel channel to "see" astrophysical and cosmological phenomena.

This became evident when the LIGO and Virgo interferometers jointly detected a BNS system on 17th August 2017: this was an important achievement of the LIGO-Virgo Collaboration (LVC), because a network of three detectors is essential to localize the source direction[9]. In this milestone observation the interferometers detected GWs emitted from the merger of a binary neutron star system. As soon as the signal was analyzed by the online system, the source location was promptly shared with the astronomical community and the global scientific effort led to the first detection of the electromagnetic counterpart of the

[8]The presence of the signal was reported in real-time by the coherent WaveBurst pipeline. An overview of this tool is given in Section 3.4.

[9]As I discuss in Chapter 2, there are two LIGO interferometers, both located in the USA (more precisely at Hanford, Washington and Livingston, Louisiana), and the Virgo interferometer in Pisa, Italy.

gravitational wave event. This was the true beginning of multimessenger astronomy with GWs [9].

The third event was detected on May 21st 2019 and recently published (September 2020). In this event the source was a BBH with the highest masses detected until now, namely $85M_{\odot}$ and $66M_{\odot}$. These masses (above all the first) are in the mass gap ($65 - 120M_{\odot}$) predicted by the pair-instability supernova theory: this means that the origin of these BHs has to be searched elsewhere, and many hypotheses have been considered, such as second-generation BHs, stellar mergers in young star clusters and BH mergers in active galactic nuclei (AGN) [17]. The latter case was considered in [18], which claims to have detected the EM flare originating from the remnant of the merger, which is supposed to have traveled through the AGN accretion disk and heated the trailing gases. Studies on this and other hypotheses are still ongoing.

Chapter 2

Gravitational Wave detectors

2.1 Operating detectors and future perspectives

During the last twenty years of 20th century a large effort was devoted to the LIGO (Laser Interferometer Gravitational Wave Observatory) project, led by Rainer Weiss, Barry C. Barish and Kip S. Thorne[1]. The building of two interferometers in Hanford, Washington and in Livingston, Louisiana started in 1994 and 1995 [20]. The initial scientific runs began in 2002 and ended in 2010: in this period no signal from gravitational waves was detected, but in the following years (2010-2015) the interferometers were upgraded to Advanced LIGO, following important groundbreaking developments at the GEO600 detector in Hannover, Germany. In 2015, Advanced LIGO began its first observing run O1 (September 2015 - January 2016). The first detection of a gravitational wave signal was made, as described above, on 14 September 2015 and was announced to the scientific community on 11 February 2016: the era of GW observations had begun [1].
The Virgo project also started roughly at the same time as LIGO, under the push of the scientific enthusiasm of Adalberto Giazotto and Alain Brillet. Its construction ended in 2003 and data were taken from 2007 to 2011, but its sensitivity was too low and no signal was detected. A period for upgrading to "Advanced Virgo" began in 2011 and ended in 2016 [21]. In May 2017 Advanced Virgo joined the two Advanced LIGO detectors in the second observing run O2 (November 2016 - August 2017) and a signal was detected by the network of three detectors on August 14th, 2017 [22] which were able to correctly localize the source direction as well[2]. A few days later, on August 17th, LIGO-Virgo observed the BNS merger event GW170817, where the joint detection by three interferometers of the gravitational counterpart of the merger allowed for a much better identification of the source location, and for the successful detection of the EM counterpart by the astronomical observatories.

A third observing run (O3) started on April 1st, 2019 and ended on March 29th, 2020 (about a month before the planned end date, due to the sanitary emergency caused by COVID-19). In this run Advanced LIGO and Advanced Virgo increased their sensitivity as

[1] They were awarded with the Nobel Prize in 2017 "for decisive contributions to the LIGO detector and the observation of gravitational waves" [19].

[2] While a first localization made by Hanford and Livingston detector produced a wide 90% credible area on the sky of 1160 deg^2, the inclusion in the analysis of the Virgo data reduced this area to 60 deg^2.

shown in fig. 2.1: unfortunately, differently from previous planning, the japanese KAGRA interferometer did not reach a sufficient sensitivity, and its data could not contribute to the scientific output of the collaboration. There are high hopes that KAGRA shall effectively contribute during the fourth observing run (O4).

In the future, these instruments shall be joined by a new detector in India (LIGO India, essentially a copy of the American LIGO's), and by another interferometer that will operate in space (and so without noise from terrestrial sources, and with extremely long arms), the Laser Interferometer Space Antenna (LISA), whose launch is planned after 2034 [23]: LISA will cover a different frequency band with respect to the LIGO-Virgo interferometer, namely the low frequency band (in the mHz range), giving the possibility to detect yet unseen phenomena, such as those associated with supermassive black holes.

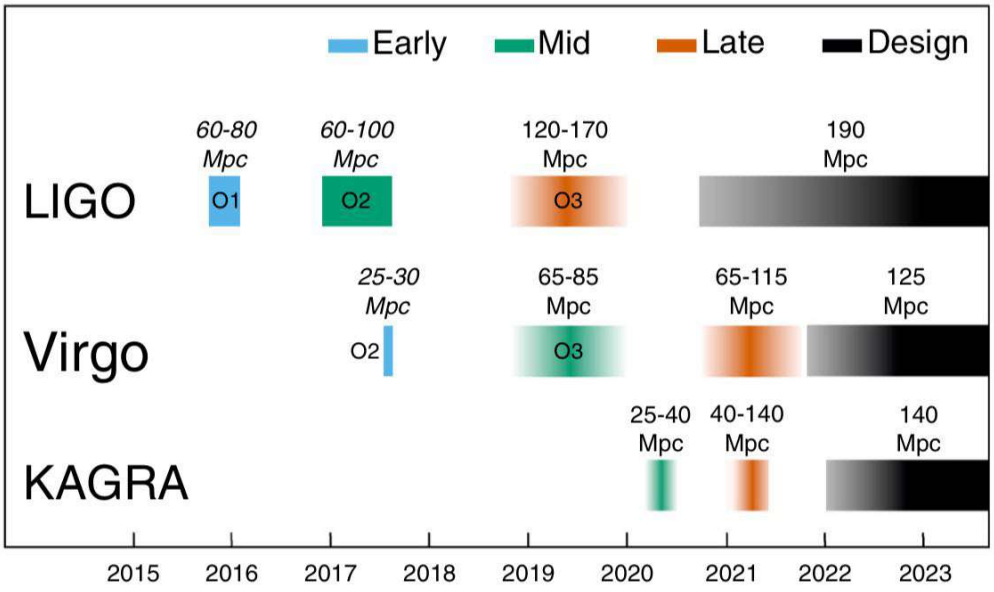


Figure 2.1: Planned sensitivity of the detectors for the completed observing runs (O1,O2,O3) and for O4. These plans may change because of the current COVID-19 emergency. The figure is taken from [24].

Detector name and label	Latitude	Longitude	a_1	a_2
LIGO Livingston (L1)	30° 33' 46.42" N	90° 46' 27.27" W	107° 42' 59.40"	197° 42' 59.40"
LIGO Hanford (H1)	46° 27' 18.53" N	119° 24' 27.57" W	35° 59' 57.84"	125° 59' 57.84"
Virgo Pisa (V1)	43° 37' 53.09" N	10° 30' 16.19" E	340° 34' 02.03"	70° 34' 02.03"

Table 2.1: Detector locations and arm orientations. The angles a_1 and a_2 describe the orientation of the first and the second arm: these are the angles through which one has to rotate the arm clockwise to point to the local North. This table is taken from [25].

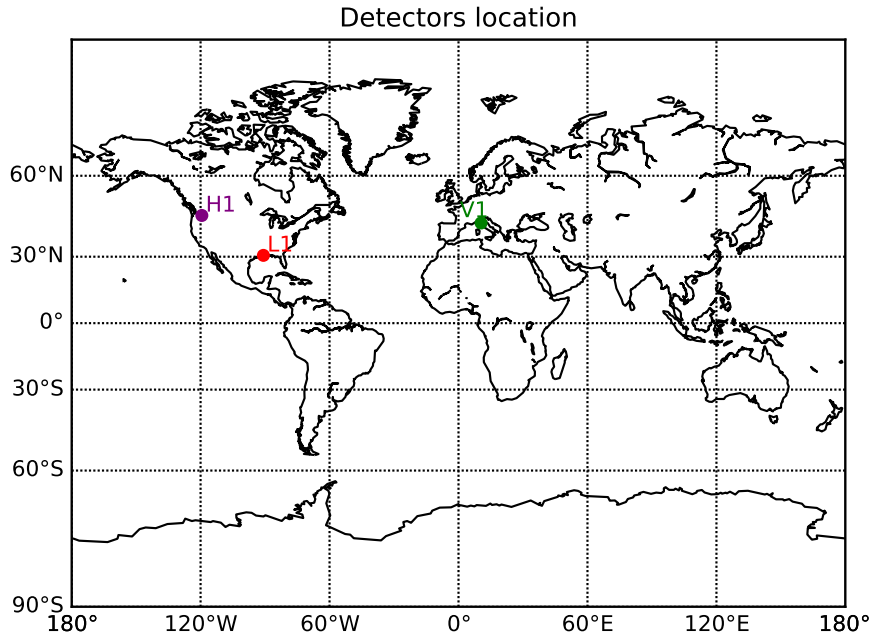


Figure 2.2: Location of the three detectors LIGO Livingston (L1), LIGO Hanford (H1) and Virgo (V1).

2.2 Brief overview of the GW interferometers

The detection of GWs is one of the most astonishing achievements in human history. Scientists predicted these waves and devised tools to measure displacements smaller than 10^{-20} m at the latest count, at least five orders of magnitude below the diameter of a proton.
To this end, the Michelson interferometer went through many important improvements [26]. In the basic configuration (fig. 2.3) a laser beam is split into two beams that travel along orthogonal arms: the beams are reflected by two mirrors and recombined to interfere beyond the beam splitter. If a GW traverses the instrument the arm size is modified, the beams travel different lengths along the two arms and inteference pattern is modified by the length change.

Starting from this basic idea additional methods had to be developed to make GW detection possible:

- large size: the interferometer arms must be large, of the order of a few kilometers
- high laser beam intensity: this produces a higher output power and reduces the relative intensity of shot noise; this is aided by the adoption of power recycling
- decouple of external noise sources: external noise source that do not directly perturb the mirrors (unlike ,e.g., seismic noise) can be decoupled from the mirrors with feedback systems

- mirror noise reduction: mirrors are directly affected by many types of noise, like mechanical thermal noise of the suspensions, the opto-thermal noise of the mirror substrates, etc.; reduction of this kind of noises is achieved by enclosing the mirrors (and the optical path) in high vacuum, in adopting silica glass fibers for the suspensions, in reducing the thermal stress of the mirrors by means of radiative heat-exchangers, etc.

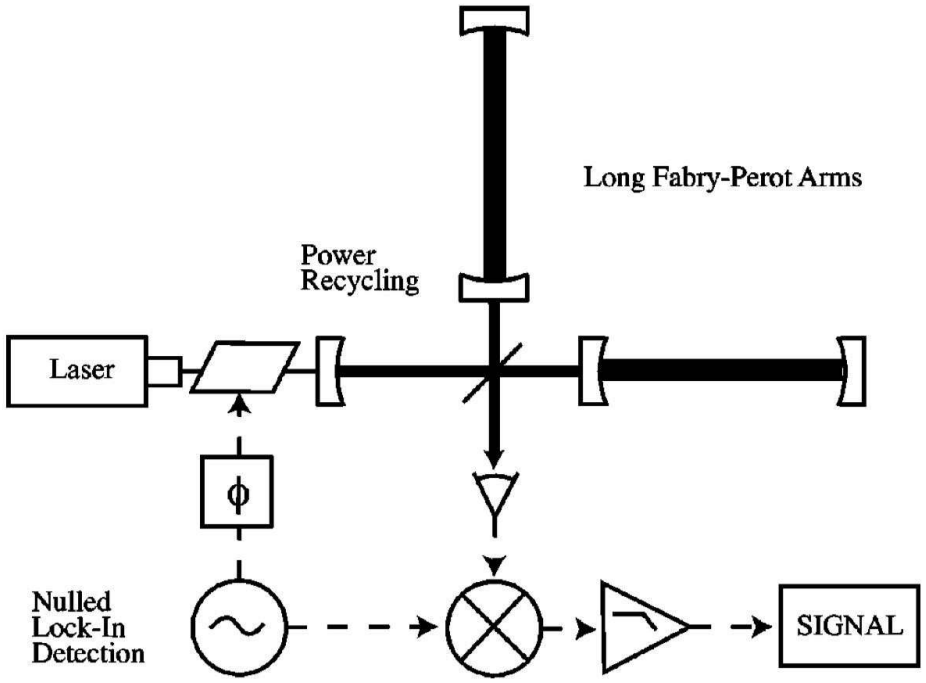


Figure 2.3: A simplified scheme of a GW detector, which represents the main detection techniques (nulled lock-in detection, Fabry-Perot cavities and power recycling) implementing for making a simple Michelson interferometer a GW one. The figure is taken from [26].

First of all, the null point of the interferometers corresponds to a dark fringe: this means that the arm lengths are set so that, after traversing the beam splitter on their way back, the two laser beams are 180° out of phase and interfere destructively at the output port when there is no GW. A complete description of the basics of the interferometer optics is given in [26].

The laser light is set to a stable wavelength of 1064 nm, while the input power is different between LIGO and Virgo detectors, anyway of the order of a few tens of W [27][28]. The interferometer incorporate long Fabry-Perot cavities, which are a clever way to increase the effective arm length[3]. Using the Fabry-Perot cavity the effective arm length is about 300 times larger than the actual arm length [29].

Shot noise can be reduced by using a larger laser beam intensity, and part of the job is done by a partially transmitting mirror – the recycling mirror – located between the laser and the beam splitter, as outlined in fig. 2.3.

[3]The real arm lengths are 3994.5 m for the LIGO detectors and 3000.0 m for Virgo.

2.3 Noise sources

In this section I discuss the main noise sources, along the lines of ref. [30] and [27]. I note at the very outset that every physical noise source is also filtered by the instrument itself (this is not the case with noise sources in the data acquisition chain), so, at each frequency f the frequency-dependent displacement $L(f)$ (length change) can be written as follows

$$L(f) = L_0 h(f) = T(f)N(f) \tag{2.1}$$

where $N(f)$ is the noise spectrum and $T(f)$ is the transfer function of the interferometer.

2.3.1 Seismic and Newtonian noise

Ground motion is one of the most important noise sources below 10 Hz, and in the current detectors (during run O3) at 10 Hz the displacement noise is about 3 orders of magnitude larger than at the freq. of maximum sensitivity. The interferometers are built with clever mirror suspension structures that minimize this noise as far as possible, so at the end of O3 the seismic noise contribution is quite small in the useful frequency range.
Another important noise contribution at low frequencies is given by the fluctuations of local gravity fields around the test masses: these couple with the arm length displacement in the following way

$$L(f) = 2\frac{N_{grav}(f)}{(2\pi f)^2} \tag{2.2}$$

with $N_{grav}(f) = \beta G \rho N_{sei}(f)$ where $\beta \simeq 10$ is a geometric factor, $\rho \simeq 1800\,\mathrm{kgm}^{-3}$ is the ground density near the mirror and $N_{sei}(f)$ is the seismic motion near the test mass. During O1 local gravity fluctuations were of the order of $N_{grav} \simeq 10^{-15}\mathrm{ms}^{-2}/\sqrt{\mathrm{Hz}}$ at 10 Hz so that the corresponding displacement was $L \simeq 5 \cdot 10^{-19}\mathrm{m}/\sqrt{\mathrm{Hz}}$. The O3 improvements gave a displacement of $L \simeq 2.28 \cdot 10^{-19}\mathrm{m}/\sqrt{\mathrm{Hz}}$ at the same frequency, about half the displacement of O1. These values can be different between the two LIGO detectors, and of course very different for the Virgo interferometer. For O3, I take the values reported for the LIGO Livingston detector.

2.3.2 Thermal noise

Thermal noise is due to thermal fluctuations of the apparatus. More specifically thermal noise drives the pendulum motion of the mirrors, the normal modes of the test masses and the violin modes which are the transverse resonant oscillations of the wires suspending the mirrors. Wires have fixed extremities and so can resonate as a violin string, generating peaks in noise spectrum corresponding to the string harmonics [31].
The power spectrum of thermal noise can be calculated with the fluctuation-dissipation theorem

$$F^2_{thermal}(f) = 4k_B T\ \mathrm{Re}(Z(f)) \tag{2.3}$$

where $\mathrm{Re}(Z(f))$ is the real part of the system impedance. The explicit form of the power spectral density is calculated in ref. [32]

$$S(f) = \frac{k_B T}{\pi^2 f^2}\frac{W_{diss}}{F_0^2} \tag{2.4}$$

where F_0 is the noise pressure caused by thermal fluctuations integrated over the test mass surface and W_{diss} is the time-averaged power dissipated in the test mass when this oscillating pressure is applied. Ref. [32] reports a value of $8.76 \cdot 10^{-40}\mathrm{m}^2/\mathrm{Hz}$ at 100 Hz for $S(f)$, while in O3 this takes the value $\sim 1.28 \cdot 10^{-40}\mathrm{m}^2/\mathrm{Hz}$ at the same frequency.

2.3.3 Quantum noise

Quantum noise originates from the quantum nature of the electromagnetic field. Shot noise is the most common kind of quantum noise, as it is a ubiquitous manifestation of the discrete nature of the electromagnetic field. Its effect on the arm length displacements is given by

$$L(f) = \frac{\lambda}{4\pi G_{\mathrm{arm}}} \left(\frac{2h\nu G_{\mathrm{src}}}{G_{\mathrm{prc}} P_{\mathrm{in}} \eta} \right)^{1/2} \frac{1}{K_-(f)} = 2 \cdot 10^{-20} \left(\frac{100kW}{P_{\mathrm{arm}}\eta} \right)^{1/2} \frac{1}{K_-(f)} \frac{\mathrm{m}}{\sqrt{\mathrm{Hz}}} \tag{2.5}$$

where G_{arm} is the arm cavity build-up, G_{src} is the inverse of the signal recycling attenuation, G_{prc} is the power recycling gain, P_{in} is the interferometer input power, P_{arm} is the power circulating in the arm cavities, η a factor that takes into account for the fact that only a fraction of the interferometer output power is transmitted to the photodiodes, and $K_- = f_-/(if + f_-)$ is the interferometer transfer function[4]. This noise matters at high frequencies (above 40 Hz): during O3 its contribution was about $3.02 \cdot 10^{-20}\mathrm{m}/\sqrt{\mathrm{Hz}}$ at 1000 Hz.

The interferometers are also affected by the quantum radiation pressure noise, which is due to vacuum fluctuations that modify the momentum of the interferometer mirrors. In this case, the effect on the arm length displacements is given by

$$L(f) = \frac{2}{cM\pi^2 f^2} (h\nu G_- P_{\mathrm{arm}})^{1/2} K_-(f) = \frac{1.38 \cdot 10^{-17}}{f^2} \left(\frac{P_{\mathrm{arm}}}{100\,\mathrm{kW}} \right)^{1/2} K_-(f) \frac{\mathrm{m\ s^{-2}}}{\sqrt{\mathrm{Hz}}} \tag{2.6}$$

where G_- is the differential coupled cavity build-up[5]. The contribution of this noise is very important below 40 Hz: during O3 it was about $8.38 \cdot 10^{-20}\mathrm{m}/\sqrt{\mathrm{Hz}}$ at 20 Hz. The observation of quantum radiation pressure noise, which has been elusive for many years, was recently performed by both LIGO and Virgo interferometers [28, 33].

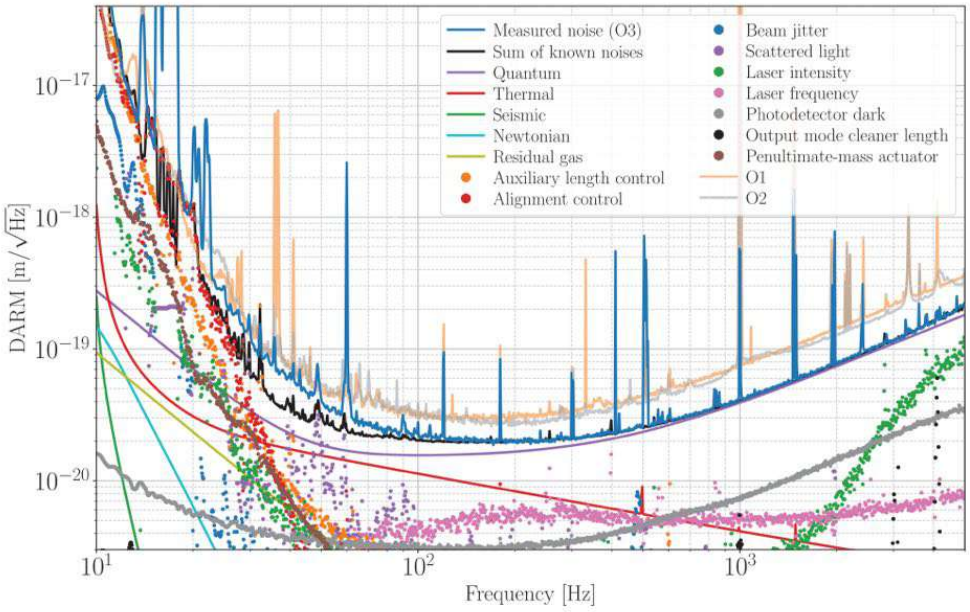
Note that the shot noise and the quantum radiation pressure noise are related by an uncertainty relation. Therefore, the performance of the interferometers can be improved by reducing the uncertainty associated with one of these noise sources (and, of course, increasing the one associated with the other): this is done substituting squeezed states[6] for the vacuum fluctuations that enter the dark port of the interferometer. The squeezing is set to reduce the shot noise and to increase the quantum radiation pressure noise: this is done to obtain a better sensitivity above 40 Hz, in the frequency range shared by the most likely GW sources

[4]The values of these parameters for the LIGO Livingston interferometer in O3 were $G_{\mathrm{arm}} = 265$, $G_{\mathrm{prc}} = 47$, $P_{\mathrm{in}} = 38\,\mathrm{W}$ and $P_{arm} = 240\,\mathrm{kW}$: note that in this last case the LIGO Hanford interferometer has different P_{arm} for the two arms, namely $P_{arm-x} = 190\,\mathrm{kW}$ and $P_{arm-y} = 200\,\mathrm{kW}$. The parameters $1/G_{\mathrm{src}}$, η and f_- (that is the differential coupled cavity pole) are reported to be in O1 as $1/G_{\mathrm{src}} = 0.11$, $\eta = 0.75$ and f_- in the range $335 - 390$Hz.

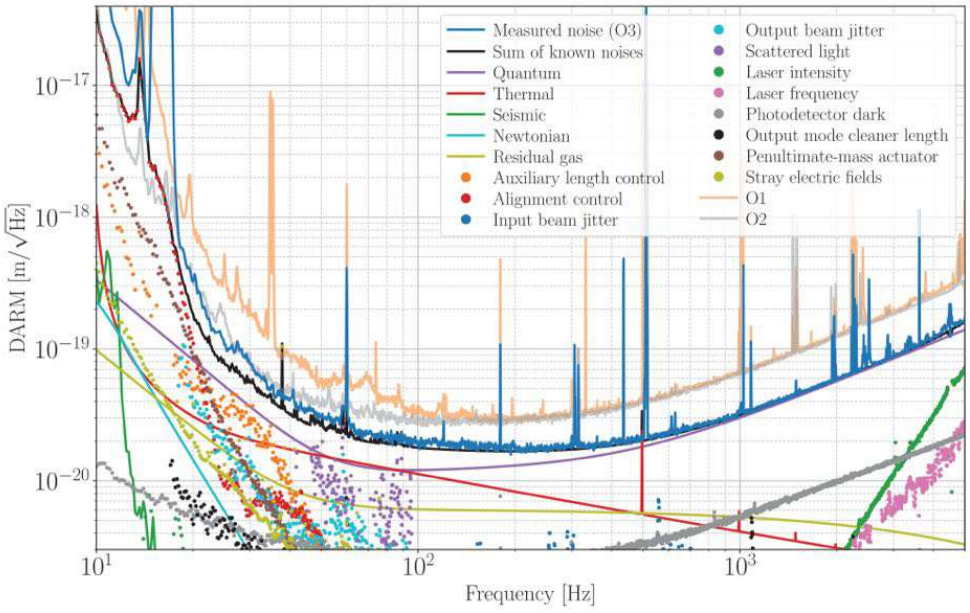
[5]Its value in O1 was $G_- = 31.4$.

[6]The squeezing is done creating correlations between normally independent fluctuations, e.g. converting one photon into two lower-frequency photons whose phases are correlated.

[34]. Several efforts were done to optimize this technique during O3, e.g. the laser power was increased and an in vacuum squeezer was installed at each LIGO site: with these upgrades the binary neutron star inspiral range (which is a useful metric for understanding the sensitivity of a detector) increased by about 14% at LIGO Livingston Observatory and 12% at LIGO Hanford Observatory.



(a) LHO



(b) LLO

Figure 2.4: The strain sensitivity (expressed as DARM, differential arm length) of LIGO Hanford Observatory (a) and LIGO Livingston Observatory (b). Calculated noise terms are given as solid lines, while measured contributions are given as dots. Note that are some notable differences between the two interferometers. The figure is taken from [27].

Chapter 3

Gravitational Wave polarization estimates

The purpose of this work is to study and compare methods to extract the polarization content of a detected signal. Along the years many different methods were developed, involving different hypotheses and/or assuming different theoretical bases. In this thesis I start my analysis from two algorithms with the following characteristics:

- they require (almost) no assumption on the detected signal waveform: for this reason they are called *minimally modeled* (sometimes also *unmodeled* or *agnostic*), they have already been instrumental in the first discovery of GWs and may again play a key role in future detections of GWs from CCSNe, pulsar glitches and others, where there is no established model for the emission of GWs. Such models are complementary to those that utilize precise models of coalescing binaries, which are extremely successful in finding the physical parameters of these exceedingly important astrophysical objects [1]
- they are *coherent*, namely the triggers are not generated individually for each detector but a unique list of triggers is made and the coherence of the data among the detectors is evaluated.

Before explaining the algorithms it is important to discuss the geometry used for the analysis.

3.1 Geometry of the detector network

As explained above (Chapter 1) the detection of a GW signal depends on the sky position of the source. Because of the different orientation of each interferometer, an effective combination of data from the global detector network requires a proper choice of reference frame.

The most natural choice is to take the Earth-centered reference frame where the origin is at the center of the Earth, the z-axis points to the North Pole, the x-axis points to the

Greenwich meridian, and the y-axis forms a right-handed Cartesian coordinate system with the x and the z axis (therefore it points 90° east of the Greenwich Meridian).

So the two angles θ (with respect to the z-axis, $\theta \in [0°, 180°]$) and ϕ (with respect to the x-axis, positive in the East direction, $\phi \in [-180°, 180°]$) are linked to the latitude λ and the longitude φ in the following way[1]

$$\theta = 90° - \lambda \tag{3.1}$$

$$\phi = \varphi. \tag{3.2}$$

Throughout this thesis I consider the network of three detectors (L1,H1,V1), with the positions and orientations listed in table 2.1.

In Section 1.2 we have seen how the antenna pattern can be expressed as a function of the detector reference frame, so it is important to define the transformation from the local reference frame to the global Earth frame and vice versa. The Euler angles that are required for this rotation can be derived following [25] from locations and arm orientation angles listed in table 2.1:

$$\alpha = \varphi + \pi/2 \tag{3.3}$$

$$\beta = \pi/2 - \lambda \tag{3.4}$$

$$\gamma = \frac{a_1 + a_2}{2} + \frac{3\pi}{2} \quad \text{if} \quad |a_1 - a_2| > \pi \quad \text{or} \tag{3.5}$$

$$\gamma = \frac{a_1 + a_2}{2} + \frac{\pi}{2} \quad \text{if} \quad |a_1 - a_2| \leq \pi. \tag{3.6}$$

$$\tag{3.7}$$

Detector name and label	α	β	γ
LIGO Livingston (L1)	-0° 46' 27.27"	59° 26' 13.58"	242° 42' 59.40"
LIGO Hanford (H1)	-29° 24' 27.57"	43° 32' 41.47"	170° 59' 57.84"
Virgo Pisa (V1)	100° 30' 16.19"	46° 22' 06.91"	115° 34' 02.03"

Table 3.1: Euler angles for the three detectors L1,H1 and V1. This table is taken from ref. [25].

Using the rotations defined by Euler's angles, we can transform a vector representation $\mathbf{x}_E$ in the global Earth frame to the representation $\mathbf{x}_d$ in detector frame as follows

$$\mathbf{x}_d = \mathcal{O}(\alpha, \beta, \gamma)\mathbf{x}_E, \tag{3.8}$$

where

$$\mathcal{O}(\alpha, \beta, \gamma) = R_\gamma R_\beta R_\alpha = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & \sin\beta \\ 0 & -\sin\beta & \cos\beta \end{pmatrix} \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.9}$$

[1]The θ angle is also known as co-latitude.

For example, if the source location of a GW signal is (θ_E, ϕ_E) then detector d detects the signal as in eq. (1.28)

$$\tilde{\xi}_d(t) = h_+(t)F_+(\theta_d, \phi_d) + h_\times(t)F_\times(\theta_d, \phi_d), \tag{3.10}$$

where (θ_d, ϕ_d) are the angles in the detector frame.

Note also that space-separated interferometers detect the same GW at different times, because GWs travel at a fixed speed, i.e, at the speed of light. The corresponding time delays times are referred to the center of the Earth, i.e., to the Earth frame (where $\mathbf{r}_E = (0, 0, 0)$) and take the simple form

$$\tau_d(\theta_E, \phi_E) = \mathbf{r}_d \cdot \frac{\mathbf{n}(\theta_E, \phi_E)}{c}. \tag{3.11}$$

Therefore, taking into account the noise in each detector, the detector response can be written as

$$\tilde{\xi}_d(t + \tau_d) = h_+(t)F_+(\theta_d, \phi_d) + h_\times(t)F_\times(\theta_d, \phi_d) + n_d(t + \tau_d). \tag{3.12}$$

3.2 The method of Gürsel and Tinto

A first rigorous approach to polarization disentangling and source direction determination was proposed in ref. [35] by Yekta Gürsel and Massimo Tinto in 1989. Even though many of their considerations have since been superseded, their paper is still an important starting point for further developments in the field.

3.2.1 Polarization disentangling: the simplest case of a three detector network

Gürsel and Tinto assume a fixed network of three detectors in their 1989 paper [35]. They define a global reference frame where a reference detector sits a the origin, and calculate the delay times relative to the reference detector. Here I replicate their analysis with one important change: I use the Earth reference frame described in the previous Section. This has the advantage of simplifying calculations, and of using the common concepts of latitude and longitude, with well-defined reference frame for each detector. With such an approach it is easier to generalize to a network with any number of detectors. This will prove to be useful also in comparisons with different algorithms for polarization disentangling.

Going back to eq. (3.12) we find that the detector responses can be written as follows[2]

$$R_{1\Lambda}(t + \tau_1) = F_{1+}(\theta_1, \phi_1)h_+(t) + F_{1\times}(\theta_1, \phi_1)h_\times(t) + \Lambda_1(t + \tau_1) \tag{3.13}$$

$$R_{2\Lambda}(t + \tau_2) = F_{2+}(\theta_2, \phi_2)h_+(t) + F_{2\times}(\theta_2, \phi_2)h_\times(t) + \Lambda_2(t + \tau_2) \tag{3.14}$$

$$R_{3\Lambda}(t + \tau_3) = F_{3+}(\theta_3, \phi_3)h_+(t) + F_{3\times}(\theta_3, \phi_3)h_\times(t) + \Lambda_3(t + \tau_3), \tag{3.15}$$

[2]In the original paper the time delays were considered from the reference detector (that was assumed to be the detector "1"), so the responses were written as $R_{1\Lambda}(t)$, $R_{2\Lambda}(t + \tau_{12})$ and $R_{3\Lambda}(t + \tau_{13})$. With respect to eq. (3.12) there is also a minor switch of notation because I choose to use here the original notation of Gürsel and Tinto.

where the noise contributions are now denoted by the Λs. We obtain the noiseless solution of this overdetermined system by combining pairs of equations and extracting three expressions for h_+ [3]

$$h_{23+}(t) = \frac{F_{3\times}R_{2\Lambda}(t+\tau_2) - F_{2\times}R_{3\Lambda}(t+\tau_3)}{K_{23}} \tag{3.16}$$

$$h_{31+}(t) = \frac{F_{1\times}R_{3\Lambda}(t+\tau_3) - F_{3\times}R_{1\Lambda}(t+\tau_1)}{K_{31}} \tag{3.17}$$

$$h_{12+}(t) = \frac{F_{2\times}R_{1\Lambda}(t+\tau_1) - F_{1\times}R_{2\Lambda}(t+\tau_2)}{K_{12}}, \tag{3.18}$$

with $K_{ij} = F_{i+}F_{j\times} - F_{j+}F_{i\times}$, $i \neq j$.
These three responses are combined as follows to reconstruct a unique response h_+

$$h_{opt+} = a_+(\theta,\phi)h_{23+}(t) + b_+(\theta,\phi)h_{31+}(t) + c_+(\theta,\phi)h_{12+}(t), \tag{3.19}$$

where the coefficients parameterize the effect of noise and are chosen so that $a_+(\theta,\phi) + b_+(\theta,\phi) + c_+(\theta,\phi) = 1$.
By minimizing the quantity

$$(\delta h_+)^2 = \frac{1}{\Delta t}\int_{t_0}^{t_1} [h_{opt+}(t) - h_+(t)]^2\, dt, \tag{3.20}$$

with respect to $a_+(\theta,\phi)$,$b_+(\theta,\phi)$ and $c_+(\theta,\phi)$ one finds three expressions for the coefficients[4]

$$a_+ = \frac{1}{\rho}\left(K_{23}^2\sigma_1^2 - \frac{K_{23}K_{12}F_{1\times}}{F_{3\times}}\sigma_3^2\right) + \frac{K_{23}F_{1\times}}{K_{12}F_{3\times}}c_+ \tag{3.21}$$

$$b_+ = \frac{1}{\rho}\left(K_{13}^2\sigma_2^2 - \frac{K_{13}K_{12}F_{2\times}}{F_{3\times}}\sigma_3^2\right) + \frac{K_{13}F_{2\times}}{K_{12}F_{3\times}}c_+ \tag{3.22}$$

$$c_+ = 1 - a_+ - b_+, \tag{3.23}$$

where $\rho = K_{23}^2\sigma_1^2 + K_{13}^2\sigma_2^2 + K_{12}^2\sigma_3^2$ and $\sigma_i = \frac{1}{\Delta t}\int_{t_0}^{t_1}\Lambda_i^2(t+\tau_i)dt$.

3.2.2 Generalization of the method of Gürsel and Tinto to larger detector networks

Here I extend the results of Gürsel and Tinto [35] to a network of N detectors. The response of the i-th detector is

$$R_{i\Lambda}(t+\tau_i) = F_{i+}(\theta_i,\phi_i)h_+(t) + F_{i\times}(\theta_i,\phi_i)h_\times(t) + \Lambda_i(t+\tau_i) \tag{3.24}$$

with $i = 1, ..., N$.

[3] The expressions for $h_\times$ are the same; from now, I will consider h_+ only. The indexes attached to the strains denote the equations used in obtaining them.

[4] For detailed calculations see Appendix C of ref. [35].

There are $\binom{N}{2}$ pairs of equations and for each pair (i,j) the partial solution is

$$h_{ij+}(t) = \frac{F_{j\times}R_{i\Lambda}(t+\tau_i) - F_{i\times}R_{j\Lambda}(t+\tau_j)}{K_{ij}} \quad \text{with} \quad K_{ij} = F_{i+}F_{j\times} - F_{i\times}F_{j+} \;\; i \neq j. \tag{3.25}$$

Now, the reconstructed waveform can be expressed as a linear combination of the partial solutions from individual detector pairs

$$h_{opt+}(t) = \sum_{j>i} a_{ij} h_{ij+}(t) = \frac{1}{2}\sum_{i\neq j} a_{ij} h_{ij+}(t) \quad \text{with} \quad \sum_{j>i} a_{ij} = 1. \tag{3.26}$$

Starting, just as before, from the expression

$$(\delta h_+)^2 = \frac{1}{\Delta t}\int_{t_0}^{t_1} [h_{opt+}(t) - h_+(t)]^2\, dt, \tag{3.27}$$

we define the constrained sum (with one Lagrange multiplier)

$$(\delta h_{new+})^2 = (\delta h_+)^2 - \lambda\left(\sum_{j>i} a_{ij} - 1\right) \tag{3.28}$$

so that using eq. (3.24) and (3.25), eq. (3.26) can be rewritten as

$$h_{opt+}(t) = \left(\sum_{j>i} a_{ij}\right) h_+(t) + \sum_{j>i} \frac{F_{j\times}\Lambda_i(t+\tau_i) - F_{i\times}\Lambda_j(t+\tau_j)}{K_{ij}} a_{ij} \tag{3.29}$$

and finally eq. (3.28) becomes

$$\begin{aligned}(\delta h_{new+})^2 =& \frac{1}{\Delta t}\int_{t_0}^{t_1}\left[\left(\sum_{j>i} a_{ij} - 1\right) h_+(t) + \sum_{j>i} \frac{F_{j\times}\Lambda_i(t+\tau_i) - F_{i\times}\Lambda_j(t+\tau_j)}{K_{ij}} a_{ij}\right]^2 dt \\ & - \lambda\left(\sum_{j>i} a_{ij} - 1\right).\end{aligned} \tag{3.30}$$

Eq. (3.30) is minimized with respect to the coefficients a_{ij}: the specific "ij" coefficients with respect to which the expression (3.30) is minimized are denoted as "$\hat{a}_{ij}$"[5]

[5]From now all the quantities related to the fixed indexes "ij" will be denoted with a hat ($\hat{a}_{ij}$, $\hat{F}_i$, $\hat{\Lambda}_i$ etc.).

$$\frac{\partial(\delta h_{new+})^2}{\partial \hat{a}_{ij}} = \frac{1}{\Delta t}\int_{t_0}^{t_1} 2\left(h_+(t) + \frac{\hat{F}_{j\times}\hat{\Lambda}_i(t+\hat{\tau}_i) - \hat{F}_{i\times}\hat{\Lambda}_j(t+\hat{\tau}_j)}{\hat{K}_{ij}}\right)\cdot$$
$$\cdot\frac{1}{2}\left[\underbrace{\left(\sum_{i\neq j} a_{ij} - 2\right)}_{=0} h_+(t) + \sum_{i\neq j}\frac{F_{j\times}\Lambda_i(t+\tau_i) - F_{i\times}\Lambda_j(t+\tau_j)}{K_{ij}}a_{ij}\right]dt - \lambda \tag{3.31}$$

with the constraint

$$\sum_{j>i} a_{ij} = 1 \tag{3.32}$$

Then, assuming that detector noises are uncorrelated, it follows that

$$\frac{1}{\Delta t}\int_{t_0}^{t_1} \hat{\Lambda}_i(t+\hat{\tau}_i)\Lambda_i(t+\tau_i)\,dt = 0 \quad i \neq \hat{i} \tag{3.33}$$

and therefore

$$\begin{aligned}\frac{\partial(\delta h_{new+})^2}{\partial \hat{a}_{ij}} &= \underbrace{\frac{1}{\Delta t}\int_{t_0}^{t_1} h_+(t)\sum_{i\neq j}\frac{F_{j\times}\Lambda_i(t+\tau_i) - F_{i\times}\Lambda_j(t+\tau_j)}{K_{ij}}a_{ij}dt}_{\equiv\,\Gamma} \\ &+ \underbrace{\left(\frac{1}{\Delta t}\int_{t_0}^{t_1}\hat{\Lambda}_i^2(t+\hat{\tau}_i)\right)}_{\hat{\sigma}_i^2}\frac{\hat{F}_{j\times}^2}{\hat{K}_{ij}^2}\hat{a}_{ij}dt + \underbrace{\left(\frac{1}{\Delta t}\int_{t_0}^{t_1}\hat{\Lambda}_j^2(t+\hat{\tau}_j)\right)}_{\hat{\sigma}_j^2}\frac{\hat{F}_{i\times}^2}{\hat{K}_{ij}^2}\hat{a}_{ij}dt - \lambda \\ &= \Gamma - \lambda + \left(\frac{\hat{\sigma}_i^2\hat{F}_{j\times}^2 + \hat{\sigma}_j^2\hat{F}_{i\times}^2}{\hat{K}_{ij}^2}\right)\hat{a}_{ij} = 0.\end{aligned} \tag{3.34}$$

Now, (with a minor switch of notation) the $\hat{a}_{ij}$ equation can be subtracted to the a_{ij} obtaining

$$a_{ij} = \frac{\hat{\sigma}_i^2\hat{F}_{j\times}^2 + \hat{\sigma}_j^2\hat{F}_{i\times}^2}{\hat{K}_{ij}^2}\cdot\frac{K_{ij}^2}{\sigma_i^2F_{j\times}^2 + \sigma_j^2F_{i\times}^2}a_{\tilde{i}\tilde{j}} \tag{3.35}$$

and using eq. (3.32) it follows that

$$\frac{1}{2}\hat{a}_{ij} = 1 - \frac{1}{2}\sum_{\substack{i\neq j\\ ij\neq\hat{i}\hat{j}}} a_{ij} = 1 - \frac{1}{2}\frac{\hat{\sigma}_i^2\hat{F}_{j\times}^2 + \hat{\sigma}_j^2\hat{F}_{i\times}^2}{\hat{K}_{ij}^2}\left(\sum_{\substack{i\neq j\\ ij\neq\hat{i}\hat{j}}}\frac{K_{ij}^2}{\sigma_i^2F_{j\times}^2 + \sigma_j^2F_{i\times}^2}\right)\hat{a}_{ij}, \tag{3.36}$$

so that the equation for the generic coefficient $\hat{a}_{ij}$ is

$$\hat{a}_{ij} = \left[\frac{1}{2} + \frac{1}{2} \frac{\hat{\sigma}_i^2 \hat{F}_{j\times}^2 + \hat{\sigma}_j^2 \hat{F}_{i\times}^2}{\hat{K}_{ij}^2} \left(\sum_{\substack{i \neq j \\ ij \neq \hat{ij}}} \frac{K_{ij}^2}{\sigma_i^2 F_{j\times}^2 + \sigma_j^2 F_{i\times}^2} \right) \right]^{-1} . \tag{3.37}$$

3.2.3 Shortcomings of the method

The paper by Gürsel and Tinto represents a milestone in techniques for polarization disentangling without any prior knowledge of the waveform but has some problems, mostly because the reconstruction is limited by the least sensitive detector, as will be shown in Chapter 4. For this reason, the method has been superseded by the more recent ones which I describe in the following Sections.
The techniques that I summarize next apply clever methods to reduce the contribution of noise, such as data whitening, and to combine data from different detectors; in particular, these algorithms are not limited in the signal reconstruction by the least sensitive detector and this leads to more powerful tools for extracting the polarizations from data.

3.3 Beyond the method of Gürsel and Tinto

The method of of Gürsel and Tinto was followed by several other proposals, see, e.g., [36]. Here, I consider in greater depth the method that is implemented within the coherent WaveBurst algorithm (Section. 3.4). The discussion in this section is adapted from [37].

The vector representing the response of a network of N detectors can be written as[6]

$$\tilde{\boldsymbol{\xi}}(\mathbf{t}) = \mathbf{F}_+(\boldsymbol{\theta}, \boldsymbol{\phi}) h_+(t) + \mathbf{F}_\times(\boldsymbol{\theta}, \boldsymbol{\phi}) h_\times(t), \tag{3.38}$$

where the vector components $\mathbf{t}_k = t + \tau_k$ and $(\boldsymbol{\theta}, \boldsymbol{\phi})_k = (\theta_k, \phi_k)$ belong to the k-th detector (as described in Section 3.1).
This set of equations is redundant for $N > 2$, but it can be reduced to a set of just two equations with the scalar products

$$\begin{aligned} \mathbf{F}_+ \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}) &= |\mathbf{F}_+|^2 h_+(t) + \mathbf{F}_+ \cdot \mathbf{F}_\times h_\times(t) \\ \mathbf{F}_\times \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}) &= \mathbf{F}_\times \cdot \mathbf{F}_+ h_+(t) + |\mathbf{F}_\times|^2 h_\times(t) \end{aligned} \tag{3.39}$$

The solution of the system of equations (3.39) is

$$h_+^{(r)}(t) = \frac{(\mathbf{F}_+ \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}))(|\mathbf{F}_\times|^2) - (\mathbf{F}_\times \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}))(\mathbf{F}_+ \cdot \mathbf{F}_\times)}{|\mathbf{F}_+|^2 |\mathbf{F}_\times|^2 - (\mathbf{F}_+ \cdot \mathbf{F}_\times)^2} \tag{3.40}$$

$$h_\times^{(r)}(t) = \frac{(\mathbf{F}_\times \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}))(|\mathbf{F}_+|^2) - (\mathbf{F}_+ \cdot \tilde{\boldsymbol{\xi}}(\mathbf{t}))(\mathbf{F}_+ \cdot \mathbf{F}_\times)}{|\mathbf{F}_+|^2 |\mathbf{F}_\times|^2 - (\mathbf{F}_+ \cdot \mathbf{F}_\times)^2} \tag{3.41}$$

[6]This is the simple case of noiseless data: the resulting equations (3.40) and (3.41) can be applied to noisy data as well, in particular with whitened data as shown in the following Sections.

As illustrated in simulations in Chapter 4, this method has the advantage of not being limited by the least sensitive detector. Remarkably, equations (3.40) and (3.41) are obtained with the likelihood approach implemented in coherent WaveBurst, which I describe in the following Section.

3.4 The Coherent WaveBurst analysis pipeline

3.4.1 A coherent method for minimally modeled gravitational wave analysis

Coherent WaveBurst (cWB) is a pipeline developed for gravitational waves analysis. Its main feature is that it is minimally modeled, i.e., it doesn't require prior assumptions on the waveform and therefore on the source characteristics [38]. Of course, this decreases the detection power of the pipeline with respect to the ones that assume some kind of model, but allows the unrestricted detection of a broad range of phenomena. For example, cWB is expected to play a key role in the future detection of the gravitational counterpart of core collapse supernovae (CCSNe), where there is not yet a general consensus on models of their gravitational wave emission [39]. cWB is *coherent* because the overall response is built up as a coherent sum over detector responses [40], and it generates a unique list of triggers combining data from several detectors [6].

Data are represented in the time-frequency (TF) domain via the Wilson-Daubechies-Meyer (WDM) wavelets [40], which provides a powerful tool to evaluate their coherence among detectors. An important feature of the WDM wavelets is that they are discrete, so that they can be computed with the fast wavelet transform, and add only a small computational load to the pipeline [41]. Thus, cWB can perform a fast data analysis and is used both for low-latency searches and for analyses on the archived data. The fast sky localizations of cWB can be promptly shared with telescopes to detect the electromagnetic counterpart of the observed GWs [38]. This takes us into the new era of the multimessenger astronomy, and provides an extraordinary tool to study and understand our Universe.
In the following sections I outline the main features of the pipeline, with particular reference to polarization reconstruction and source direction.

3.4.2 Likelihood analysis for a network of GW detectors

At its core, the algorithm combines coherently the detector data stream in a likelihood function [6]. From now on, I assume that there are K detectors, each one producing a set of I pixels in the TF map, that I denote as $x_k[i]$ (k representing the detector, i the TF pixel), so that for each i one can define a pixel vector $\mathbf{x}[i] = x_1[i], ..., x_K[i]$.

The likelihood for the hypothesis H_0 that data contain background noise only is

$$p(x \mid H_0) = \prod_{i=1}^{I} \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{x_k^2[i]}{2\sigma_k^2}\right), \tag{3.42}$$

while the likelihood for the hypothesis H_1 that data contain both signal and noise is

$$p(x \mid H_1) = \prod_{i=1}^{I} \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{(x_k[i] - \tilde{\xi}_k[i])^2}{2\sigma_k^2} \right), \tag{3.43}$$

where

$$\tilde{\boldsymbol{\xi}}[i] = \mathcal{F}\mathbf{h}[i] \quad \text{with} \quad \mathcal{F} = \begin{pmatrix} F_{1+}(\theta_1, \phi_1) & F_{1\times}(\theta_1, \phi_1) \\ ... & ... \\ F_{K+}(\theta_K, \phi_K) & F_{K\times}(\theta_K, \phi_K) \end{pmatrix} \quad \text{and} \quad \mathbf{h}[i] = (h_+[i], h_\times[i]). \tag{3.44}$$

To decide whether we have a detection or not, we choose the likelihood ratio, a choice supported by the (frequentist) Neyman-Pearson lemma stating that the likelihood ratio is the most efficient detection statistic, meaning that it has the lowest probability of wrong rejection of a detection for a fixed false detection probability

$$\Lambda(\mathbf{x}, \Omega) = \frac{p(\mathbf{x} \mid \mathbf{h}(\Omega))}{p(\mathbf{x} \mid 0)} \tag{3.45}$$

where Ω denotes the sky location in the case of H_1, so that the algorithm rejects H_0 when $\Lambda(\mathbf{x}, \Omega)$ is greater than a fixed threshold.

When we assume that the detectors have a quasi-stationary Gaussian noise with power spectral densities $S_1, ..., S_K$ it's useful to define new, whitened quantities as follows

- the whitened pixel vector

$$\mathbf{w}[i] = \frac{x_1[i]}{\sqrt{S_1[i]}}, \ ..., \frac{x_K[i]}{\sqrt{S_K[i]}} \tag{3.46}$$

- the whitened antenna pattern matrix[7]

$$\mathbf{F}[i] = \begin{pmatrix} \dfrac{F_{1+}(\theta_i, \phi_i)}{\sqrt{S_1[i]}} & \dfrac{F_{1\times}(\theta_i, \phi_i)}{\sqrt{S_1[i]}} \\ ... & ... \\ \dfrac{F_{K+}(\theta_K, \phi_K)}{\sqrt{S_K[i]}} & \dfrac{F_{K\times}(\theta_K, \phi_K)}{\sqrt{S_K[i]}} \end{pmatrix} \tag{3.47}$$

- the whitened network-response vector

$$\boldsymbol{\xi}[i] = \mathbf{F}[i]\mathbf{h}[i] \tag{3.48}$$

[7]From now I denote the original antenna patterns with $F_{+,\times}$ (italics), and the whitened antenna patterns with $\mathrm{F}_{+,\times}$.

cWB implements a useful transformation that further simplifies the equations.
This transformation consists in choosing a reference frame in which the vectors $\mathrm{f}_{+\mathrm{DPF}} = (\mathrm{f}_{1+\mathrm{DPF}}, ..., \mathrm{f}_{K+\mathrm{DPF}})$ and $\mathrm{f}_{\times\mathrm{DPF}} = (\mathrm{f}_{1\times\mathrm{DPF}}, ..., \mathrm{f}_{K\times\mathrm{DPF}})$ are orthogonal to each other. This is the so called Dominant Polarization Frame (DPF) and the linear transformation to this frame is defined by the rotation [6]

$$\mathrm{f}_{k+\mathrm{DPF}} = \mathrm{F}_{k+}\cos(\gamma) + \mathrm{F}_{k\times}\sin(\gamma) \tag{3.49}$$
$$\mathrm{f}_{k\times\mathrm{DPF}} = -\mathrm{F}_{k+}\sin(\gamma) + \mathrm{F}_{k\times}\cos(\gamma), \tag{3.50}$$

where $\cos(\gamma)$ and $\sin(\gamma)$ are

$$\cos(2\gamma) = \frac{\mathbf{F}_+^2 - \mathbf{F}_\times^2}{\sqrt{(\mathbf{F}_+^2 - \mathbf{F}_\times^2)^2 + (2\mathbf{F}_+\mathbf{F}_\times)^2}} \tag{3.51}$$
$$\sin(2\gamma) = \frac{2\mathbf{F}_+\mathbf{F}_\times}{\sqrt{(\mathbf{F}_+^2 - \mathbf{F}_\times^2)^2 + (2\mathbf{F}_+\mathbf{F}_\times)^2}}. \tag{3.52}$$

Another consequence of this transformation is that $|\mathbf{f}_{\times\mathrm{DPF}}| \leq |\mathbf{f}_{+\mathrm{DPF}}|$.
Note that as already explained in Chapter 1 ξ_k is the result of a scalar product and so it is invariant under a rotation of the polarization frame[8]

$$\tilde{\xi}_k = F_{k+}h_+ + F_{k\times}h_\times = \mathrm{f}_{k+\mathrm{DPF}}h_{+\mathrm{DPF}} + \mathrm{f}_{k\times\mathrm{DPF}}h_{\times\mathrm{DPF}}. \tag{3.53}$$

Next, one introduces the likelihood functional $\mathcal{L}$ that is defined as twice the logarithm of the likelihood ratio Λ

$$\mathcal{L} = 2\,\mathbf{w}\cdot\boldsymbol{\xi}^\dagger - \boldsymbol{\xi}\cdot\boldsymbol{\xi}^\dagger, \tag{3.54}$$

where all the quantities are assumed to be expressed in the DPF. We maximize the functional $\mathcal{L}$ by taking derivatives with respect to $h_{+\mathrm{DPF}}$ and $h_{\times\mathrm{DPF}}$

$$\frac{\delta\mathcal{L}}{\delta h_{+\mathrm{DPF}}} = \mathbf{w}\cdot\mathbf{f}_{+\mathrm{DPF}} - |\mathbf{f}_{+\mathrm{DPF}}|^2\, h_{+\mathrm{DPF}} = 0 \tag{3.55}$$
$$\frac{\delta\mathcal{L}}{\delta h_{\times\mathrm{DPF}}} = \mathbf{w}\cdot\mathbf{f}_{\times\mathrm{DPF}} - |\mathbf{f}_{\times\mathrm{DPF}}|^2\, h_{\times\mathrm{DPF}} = 0 \tag{3.56}$$

and by solving these equations for $h_{+\mathrm{DPF}}$ and $h_{\times\mathrm{DPF}}$ (and so, for h_+ and $h_\times$) we find

$$h_{+\mathrm{DPF}} = \frac{\mathbf{w}\cdot\mathbf{f}_{+\mathrm{DPF}}}{|\mathbf{f}_{+\mathrm{DPF}}|^2} \tag{3.57}$$
$$h_{\times\mathrm{DPF}} = \frac{\mathbf{w}\cdot\mathbf{f}_{\times\mathrm{DPF}}}{|\mathbf{f}_{\times\mathrm{DPF}}|^2} \tag{3.58}$$

$$h_+ = h_{+\mathrm{DPF}}\cos(\gamma) - h_{\times\mathrm{DPF}}\sin(\gamma) \tag{3.59}$$
$$h_\times = h_{+\mathrm{DPF}}\sin(\gamma) + h_{\times\mathrm{DPF}}\cos(\gamma). \tag{3.60}$$

[8]It is straightforward that this equality holds for whitened quantities also.

Note now that eqs. (3.57) are the same obtained in Section 3.3 (eqs. (3.40) and (3.41)) with $\mathbf{f}_{+\mathrm{DPF}} \cdot \mathbf{f}_{\times\mathrm{DPF}} = 0$: the DPF greatly simplifies reconstruction.
So the detector-response vector can be written as

$$\boldsymbol{\xi} = h_{+\mathrm{DPF}}\mathbf{f}_{+\mathrm{DPF}} + h_{\times\mathrm{DPF}}\mathbf{f}_{\times\mathrm{DPF}} = \frac{\mathbf{w}\cdot\mathbf{f}_{+\mathrm{DPF}}}{|\mathbf{f}_{+\mathrm{DPF}}|^2}\mathbf{f}_{+\mathrm{DPF}} + \frac{\mathbf{w}\cdot\mathbf{f}_{\times\mathrm{DPF}}}{|\mathbf{f}_{\times\mathrm{DPF}}|^2}\mathbf{f}_{\times\mathrm{DPF}}. \tag{3.61}$$

Note that this form of the detector response has a geometrical interpretation: it can be regarded as the projection of the data on the plane defined by the antenna pattern vectors $\mathbf{f}_{+\mathrm{DPF}}$ $\mathbf{f}_{\times\mathrm{DPF}}$, while noise is ideally in the sub-space orthogonal to the plane defined by $(\mathbf{f}_{+\mathrm{DPF}}, \mathbf{f}_{\times\mathrm{DPF}})$ [6]. Actually, part of the noise could leak into this plane and many techniques were developed to address this problem, like the so called regulators. The so-called *hard regulator* consists in assuming $|\mathbf{f}_{\times\mathrm{DPF}}| = 0$, i.e., that the GW signal has only an h_+ component, so that the detector network is insensitive to the $h_\times$ component. The hard regulator implies a lower-efficiency glitch detection but of course the non detection of $h_\times$ component would be worse, especially for sky regions in which $|\mathbf{f}_{\times\mathrm{DPF}}| \simeq |\mathbf{f}_{+\mathrm{DPF}}|$ [6][9].

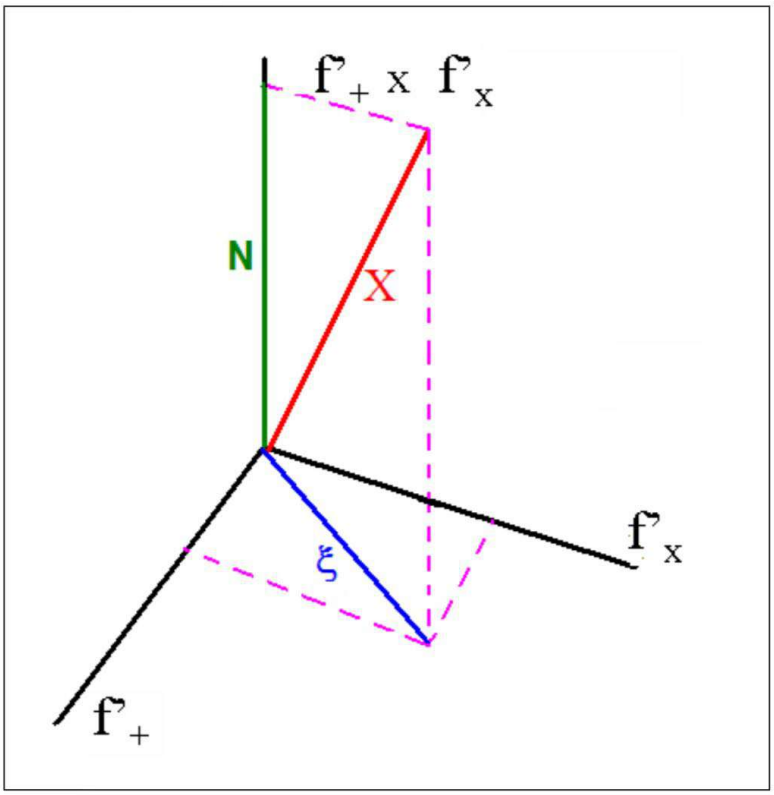


Figure 3.1: Dominant polarization frame. The detector response $\boldsymbol{\xi}$ is the projection of the data $\mathbf{X}$ on the plane defined by the DPF antenna patterns vectors $(\mathbf{f}_{+\mathrm{DPF}}; \mathbf{f}_{\times\mathrm{DPF}})$. The noise $\mathbf{N}$ is so the component of $\mathbf{X}$ in the subspace orthogonal to the antenna patterns plane.

The maximum likelihood ratio statistic can be calculated by inserting the solution (3.61)

[9]Details on regulators can be found in [6], [38] and [40].

into eq. (3.54), and we find

$$\mathcal{L}_{\max} = \sum_i \mathbf{w}[i]\mathcal{P}[i]\mathbf{w}^\dagger[i], \tag{3.62}$$

where $\mathcal{P}$ is the projector constructed with $\mathbf{e}_+$ and $\mathbf{e}_\times$ (namely the unit vectors along $\mathbf{f}_{+\mathrm{DPF}}$ and $\mathbf{f}_{\times\mathrm{DPF}}$)

$$\mathcal{P}_{nm}[i] = e_{n+}[i]e_{m+}[i] + e_{n\times}[i]e_{m\times}[i]. \tag{3.63}$$

Finally, the maximum likelihood ratio is a quadratic form representing the module of the projection of the data in the plane defined by the antenna pattern vectors in the DPF.

The likelihood approach also helps in reconstructing the source sky location that corresponds to the angles (θ, ϕ) (in the Earth frame) that maximize

$$\mathcal{L}_{\mathrm{sky}}(\theta, \phi) = c_c(\theta, \phi)\mathcal{L}_{\max}(\theta, \phi), \tag{3.64}$$

where c_c is the network correlation coefficient, defined as

$$c_c = \frac{E_c}{E_c + E_n}, \tag{3.65}$$

with

$$E_c = \sum_i \sum_{n \neq m} w_n[i]\mathcal{P}_{nm}[i]w_m^*[i] \tag{3.66}$$

and

$$E_n = \sum_i |\mathbf{w}[i] - \boldsymbol{\xi}[i]|^2 \tag{3.67}$$

and where E_c is the coherent energy of the event. The off-diagonal elements of the likelihood ($n \neq m$) indicate the strength of the coherence of data in different detectors. E_n is the *null stream energy*, namely the energy in the sub-space orthogonal to the plane $(\mathbf{f}_{+\mathrm{DPF}}; \mathbf{f}_{\times\mathrm{DPF}})$ (i.e., the noise energy). Note that, ideally, this energy should not contain any contribution from the signal at the correct source location[10] and so the correct source location corresponds to the minimum of E_n and to the maximum of c_c[11].
Thus, by combining this and $\mathcal{L}_{\max}$, the statistic $\mathcal{L}_{\mathrm{sky}}$ is used to evaluate the source location of a detected signal: its use is illustrated in the simulations of Chapter 4.

[10]The dependence from the angles (θ, ϕ) is in the antenna patterns in the expression for $\boldsymbol{\xi}$ (eq. (3.61)).

[11]The network correlation coefficient represents also a powerful parameter to evaluate the data coherence among the detectors and to distinguish genuine GW events ($c_c \simeq 1$) from spurious ones ($c_c << 1$)

Chapter 4

Simulations and tests

4.1 Overview

In this Chapter I discuss tests of the methods described above with simulated data.

Needless to say, 1) the previously described methods have already been widely tested and used in GW data analysis (in particular cWB), and 2) these methods are not the only ones, as several other approaches have been tried and tested in GW analysis. The aim of this thesis is to understand strengths and weaknesses of these tools: to do so, I have implemented the algorithms in Python, and carried out tests on simulated data.

4.1.1 Simulating the GW signals

As discussed above, the starting point for the simulation chain is the calculation of the detector response. It can be computed from the following equation

$$\tilde{\xi}_k(t+\tau_k) = h_+(t)F_+(\theta_k,\phi_k) + h_\times(t)F_\times(\theta_k,\phi_k) + n_k(t+\tau_k) \tag{4.1}$$

which requires the strains h_+ and $h_\times$, the source location with respect to each detector (θ_k, ϕ_k) and the simulation of noise for each detector.

Computing the strains

The strains are calculated with Python code using the PyCBC frame [42]: this is an open source software written in Python which implements many useful tools for GW data analysis, such as modules for signal processing, gravitational waveform generation and many others. In this thesis I use the PyCBC waveform library implementing the SEOBNR_v4 (Spinning Effective One Body-Numerical Relativity) model: this is an improved version of the effective-one-body (EOB) formalism which gives semi-analytical waveform models for BBHs representing very good approximation of GR [43].

Here I consider signals for BBH mergers with different masses[1]:

[1] n both cases the BHs are assumed to have zero spin.

- one in which the black holes are assumed to have masses of $11.0M_{\odot}$ and $7.6M_{\odot}$ at a luminosity distance of 320 Mpc, corresponding to those of the event GW170608 detected by the two Advanced LIGO detectors [7];
- one in which the black holes are assumed to have a mass of $30.7_{\odot}$ and $25.3M_{\odot}$ at a luminosity distance of 600 Mpc, corresponding to the masses of the event GW170814 detected by the two Advanced LIGO detectors by Advanced Virgo [7].

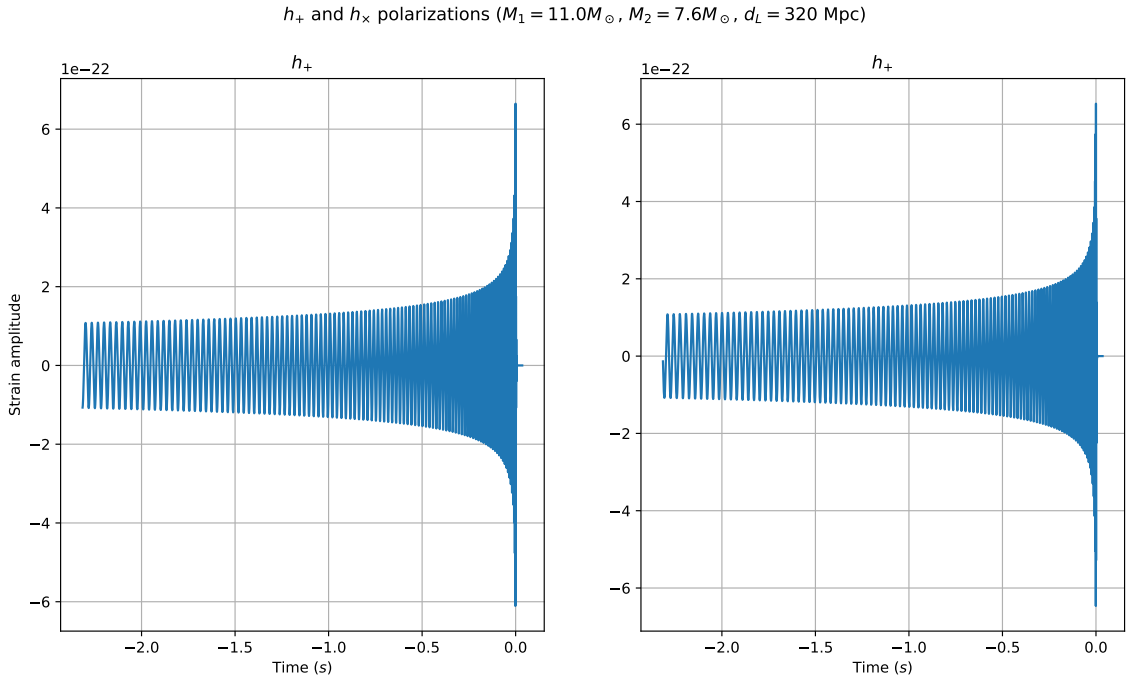


Figure 4.1: Strains for a BBH coalescence calculated with the SEOBNR_v4 model. The black holes are assumed to have masses of $11.0M_{\odot}$ and $7.6M_{\odot}$.

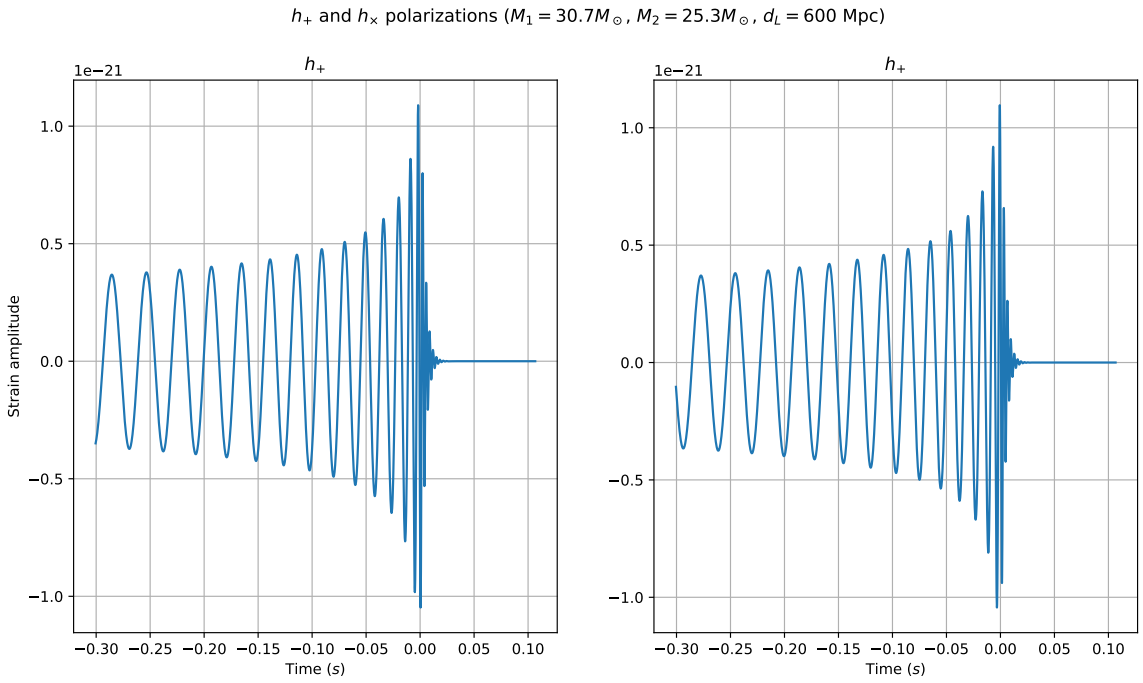


Figure 4.2: Strains for a BBH coalescence calculated with the SEOBNR_v4 model. The black holes are assumed to have masses of $30.7_\odot$ and $25.3M_\odot$.

Source direction

The source direction of the simulated signal is determined by choosing a pair of angles (θ_E, ϕ_E). The corresponding angles in the detector frames are obtained from the source direction in the Earth frame $\mathbf{n}_E = (\sin\theta_E \cos\phi_E, \sin\theta_E \sin\phi_E, \cos\theta_E)$ after performing the transformation $\mathbf{n}_k = (\sin\theta_k \cos\phi_k, \sin\theta_k \sin\phi_k, \cos\theta_k) = \mathcal{O}(\alpha_k, \beta_k, \gamma_k)\mathbf{n}_E$. Finally, we find the angles in the detector frame from the equations

$$\cos\theta_k = n_{kz} \tag{4.2}$$

$$\cos 2\phi_k = 2\frac{n_{kx}^2}{1 - n_{kz}^2} - 1 \tag{4.3}$$

$$\sin 2\phi_k = 2\frac{n_{kx} n_{ky}}{1 - n_{kz}^2} \tag{4.4}$$

Noise simulation

The noise is simulated for each detector using the equations of Section 2.3 which represent the main noise sources reported in [30] and [6]. I remark that all signals are implemented in Python as arrays and are therefore discrete, both in the time and in the frequency domain, e.g., noise in frequency domain $n(f)$ is represented by the vector $n(f_k)$, where f_k is a discrete set of frequencies.

For obtaining the noise strain sensitivity nss(f) I take the square root of the sum of the

squared module associated with each noise contribution[2]

$$L_{\text{tot}}(f) = \sqrt{\left|L_{\text{Newtonian}}(f)\right|^2 + \left|L_{\text{thermal}}(f)\right|^2 + \left|L_{\text{quantum radiation pressure}}(f)\right|^2 + \left|L_{\text{shot}}(f)\right|^2} \tag{4.5}$$

$$\text{nss}(f) = \frac{L_{\text{tot}}(f)}{L_0} \tag{4.6}$$

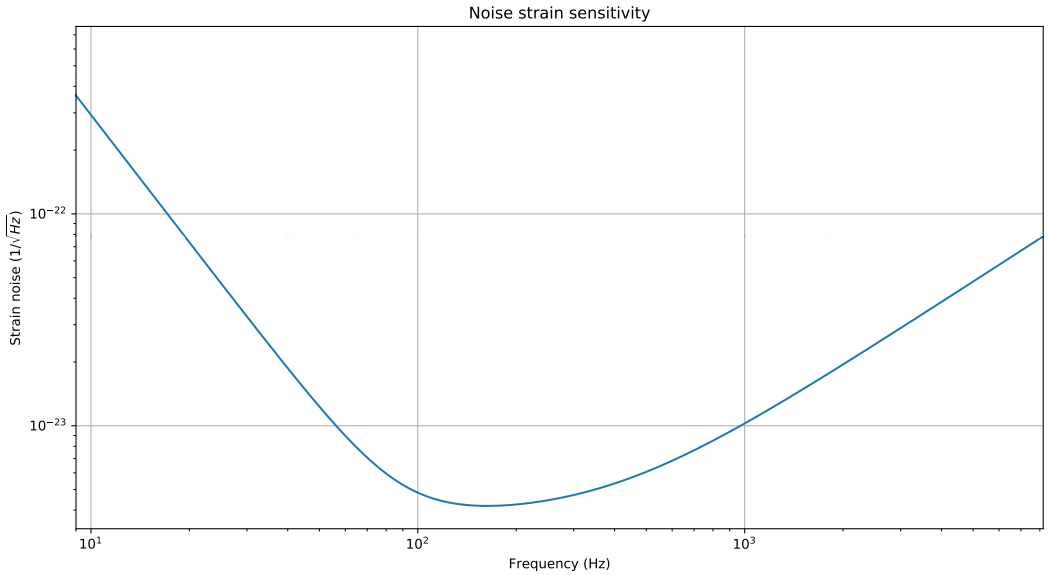


Figure 4.3: Calculated noise strain sensitivity. The equations and relative experimental values are the LIGO O3 ones reported in [27].

In the algorithms which require whitening the data will be divided by the square root of the noise power spectral density: this is calculated for each array's element k as

$$\sqrt{S(f_k)} = \sqrt{\frac{L_{\text{tot}}(f_k)^2}{N^2}} \tag{4.7}$$

where N is the array's length.

I calculate the frequency domain representation of the noise realization $n(f)$ with the algorithm proposed in [44], which allows for the generation of non deterministic noise with arbitrary PSDs[3]: using this the k^{th} element of the array is calculated as

[2]In this way I assume (which is straightforward) independent noise sources; note that the noise strain sensitivity has unit of measurement $1/\sqrt{\text{Hz}}$, so the $L(f)$ contributions (which unit of measurement is $m/\sqrt{Hz}$) have to be divided for the arm length L_0.

[3]Here I consider $n(f)$ dimensionless, so that it is obtained multiplying L_{tot} for $\sqrt{f_s}$, where f_s is the sampling frequency, i.e., 16 384 Hz, and dividing it for the arm length L_0.

$$n(f_k) = \sqrt{\frac{1}{2}}\mathcal{N}_1(0,1)\sqrt{S(f_k)} \cdot N \cdot \frac{\sqrt{f_k - f_{k-1}}}{L_0} + i\sqrt{\frac{1}{2}}\mathcal{N}_2(0,1)\sqrt{S(f_k)} \cdot N \cdot \frac{\sqrt{f_k - f_{k-1}}}{L_0} \tag{4.8}$$

$$= \sqrt{\frac{1}{2}}\mathcal{N}_1(0,1)L_{\mathrm{tot}}(f_k)\frac{\sqrt{f_s}}{L_0} + i\sqrt{\frac{1}{2}}\mathcal{N}_2(0,1)L_{\mathrm{tot}}(f_k)\frac{\sqrt{f_s}}{L_0}, \tag{4.9}$$

where $\mathcal{N}_i(0,1)$ is a normally distributed random number and $N \cdot \sqrt{f_k - f_{k-1}} = \sqrt{f_s}$.

To obtain a real-valued $n(t)$ the negative-frequency values are taken as $n(-f) = n(f)^*$. Next, $n(t)$ is obtained by taking the Discrete Inverse Fourier transform implemented in the Scipy package scipy.fftpack.ifft[4].

Even if eq. (4.5) implements all the relevant noise contributions fig. 4.3 is close but not exactly equal to fig. 2.4: in every case this is not a problem for our purposes, moreover the noise amplitude will be varied through a noise scaling factor for testing algorithms performances both for low and high noise signals (see Section 4.2). However it should be taken into account that while LIGO Livingston and LIGO Hanford detectors have comparable noise strain sensitivities the Virgo interferometer has higher one (namely higher noise): the noise spectra are a bit different also in shape, as it can be seen in fig. 4.4, but for simplicity (except in some specific cases) I decide to multiply the V1 noise for a factor 3.3 respect to the L1 and H1 one; the choice of this factor takes into account that in O2 Virgo had a sensitivity about 3.3 times lower than the LIGO detectors, as it can be seen in both figures 2.1 and 4.4.

4.1.2 Some notes on cWB equations usage

A main feature of the cWB pipeline is the usage of the WDM wavelets. These provide specific advantages, such as spectral leakage control and flexible structure of the frequency sub-bands[5] as well as the existence of analytic time-delay filters, which are a useful tool for aligning the detectors responses (subjected to different time delays) and then to perform data analysis, such as polarization reconstruction and localization of the gravitational-wave sources in the sky [6, 41]. Last but not least, WDM wavelets have a low computational cost and this gives speed and efficiency to the pipeline.

Still, in the framework of this work, where I mainly test the polarization reconstruction, a simpler analysis with discrete Fourier transforms suffices: therefore all the equations used by cWB are considered in frequency domain (and mostly for data whitening) and then only the final reconstructed polarizations are expressed in the time domain.

[4]The Discrete Inverse Fourier transform implemented in scipy.fftpack.ifft does not modify the dimensions of the data, so $n(t)$ is dimensionless too, as the calculated data $F_+h_+ + F_\times h_\times$ to which it will be summed.

[5]Note that the resolution in time and frequency can not be arbitrary small but is subject to a sort of uncertainty principle $\Delta t \Delta f \geqslant$ costant. The key idea of wavelets is to change the resolution by varying the frequency, e.g. thus obtaining a very good time resolution (poor frequency resolution) at high frequency and a very good frequency resolution (poor time resolution) at low frequency ($\Delta f \propto f$) [45].

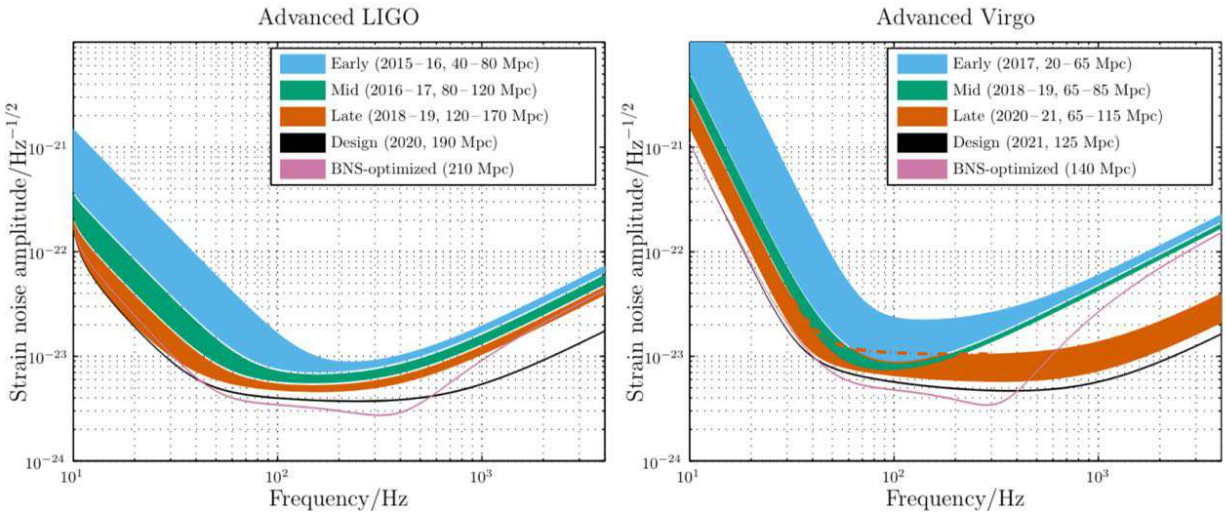


Figure 4.4: Noise strain sensitivity for LIGO-Virgo detectors. The figure is taken from [24].

4.1.3 My own implementation of the cWB methodology

I have developed a very simple implementation of cWB within the Python framework: it consists in the straightforward usage of the PyWavelets package, which provides a lot of software tools for both continuous and discrete wavelet transforms [46]. In my case I apply the single level discrete wavelet transform to noisy data with the following instruction

```
pywt.dwt(data, wavelet='dmey', mode='zero', axis=-1)
```

where I use the discrete Meyer wavelets (`wavelet='dmey'`), which are close to those actually implemented in cWB (the Wilson-Daubechies-Meyer wavelets). This command returns a pair of arrays of coefficients (cA,cD), where the cAs are the *approximation coefficients* and the cDs are the *details coefficients*. Setting the details coefficient to zero and taking the inverse wavelet transform one obtains a clearer signal is obtained (fig. 4.5): next, this is denoised and treated as in cWB. As I show in the analysis, this method returns a better reconstruction of the original polarizations.

Thus, in summary, the following methods are analyzed and tested with simulations

- Gürsel and Tinto equations,

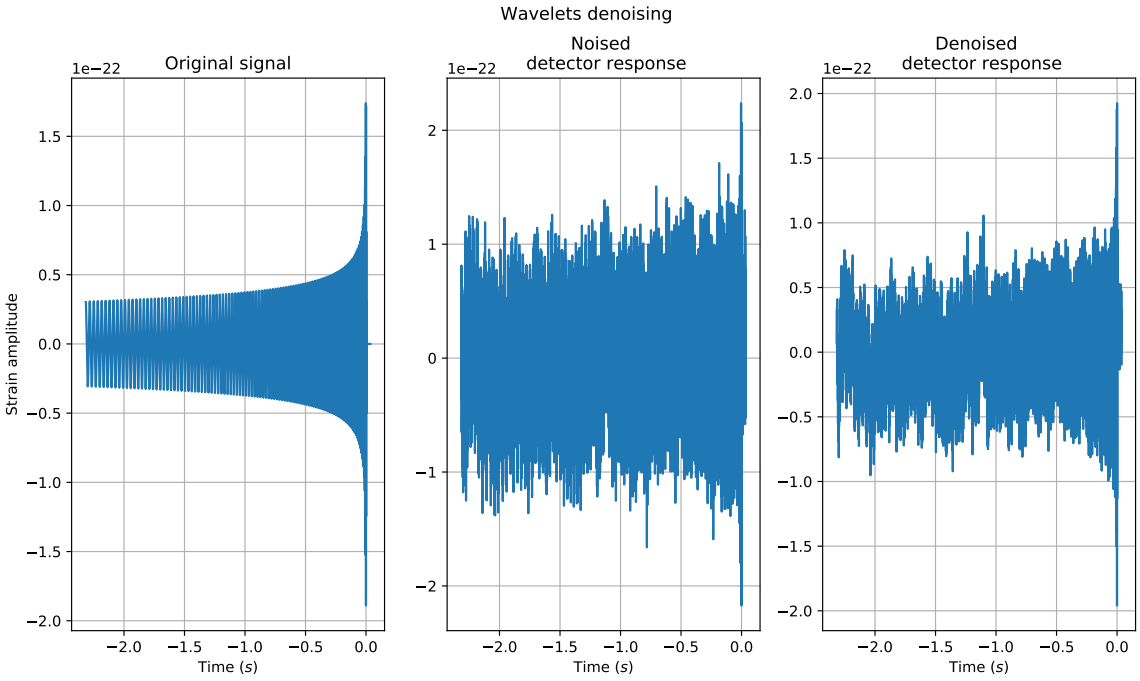


Figure 4.5: Example of wavelet denoising. The left panels shows the noiseless detector response, the centre panel shows the response plus noise, and the right panel shows the denoised response where the details coefficients have been set to zero.

- cWB equations,
- cWB equations with wavelets denoising.

4.2 Polarization reconstruction

4.2.1 Dependence on detector sensitivity

I consider first the algorithms in the particular case of two detectors with high sensitivity and one with low sensitivity. This is important to test a key feature for the algorithms, namely that they should not depend on the sensitivity of the least sensitive detector. This is crucial as new detectors are introduced into the network, since – initially at least – they could have a sensitivity much lower than the already operating ones (as was for Virgo in O2 and as it will be for KAGRA in O4, see fig. 2.1): so for having anyway the advantages of the data from a new (also if less sensitive) detector, that could improve the source location identification and others, an algorithm not depending on the less sensitive detector is required.

For this test I perform a simulation with known source location (chosen to be at the intersection of the Equator and the Greenwich meridian) taking a scaling factor of 0.1 (low noise) for L1 and H1 and of 3.3 (high noise) for V1. The polarizations are assumed to be the ones associated with a BBH source with masses $11.0M_{\odot}$ and $7.6M_{\odot}$ (fig. 4.1). The results of this simulation are reported in figures 4.6,4.7,4.8 and 4.9. While, as already known [40], the cWB reconstructions does not depend on the least sensitive detector, the Gürsel and Tinto reconstruction is heavily affected by its larger noise, as can be seen in the reconstruction of $h_{\times}$. In this case, the efficiency of the Gürsel and Tinto algorithm is significantly reduced, and by itself it justifies the continued development of several other methods after the publication of ref. [35] in 1989. However, as explained earlier, this algorithm represents a first key attempt in the development of a technique for polarizations reconstruction without any prior knowledge of the detected waveform and it was a milestone for future developments [36] [38].

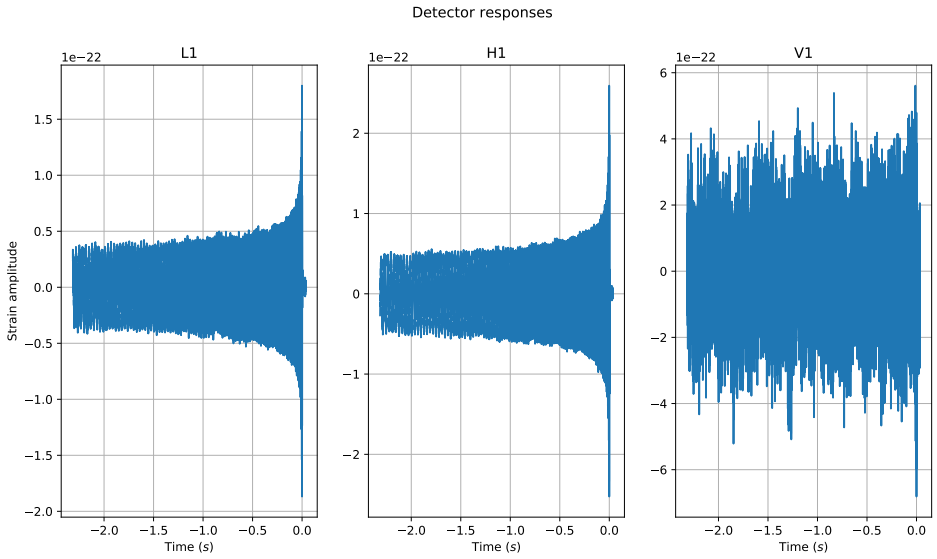


Figure 4.6: Detector responses for the simulated signal from the BBH with masses $11.0M_{\odot}$ and $7.6M_{\odot}$. The source location is $(\lambda, \varphi) = (0, 0)$ and scaling factors are 0.1 for L1 and H1 (low noise) and 3.3 for V1 (high noise).

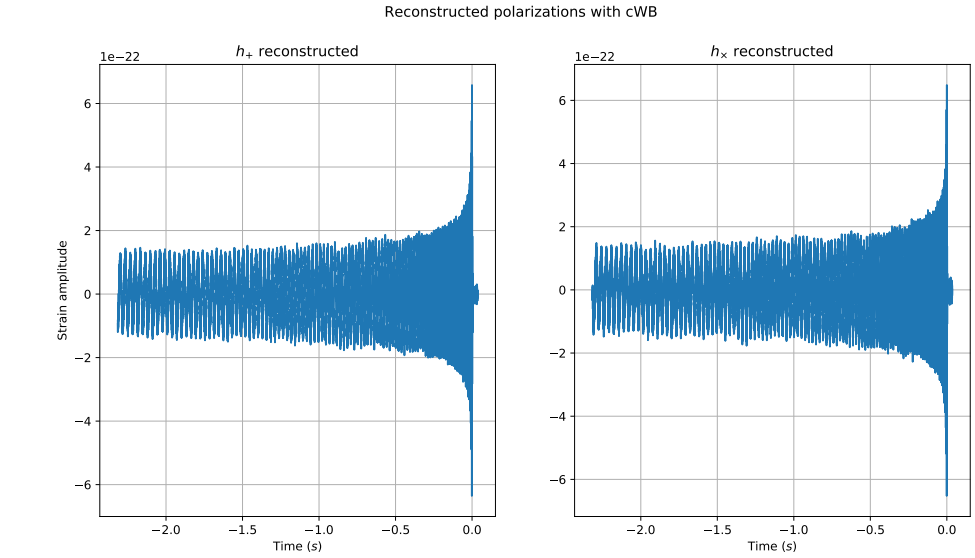


Figure 4.7: Reconstructed polarizations with cWB algorithm for the simulated signal from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. The source location is $(\lambda, \varphi) = (0, 0)$ and scaling factors are 0.1 for L1 and H1 (low noise) and 3.3 for V1 (high noise).

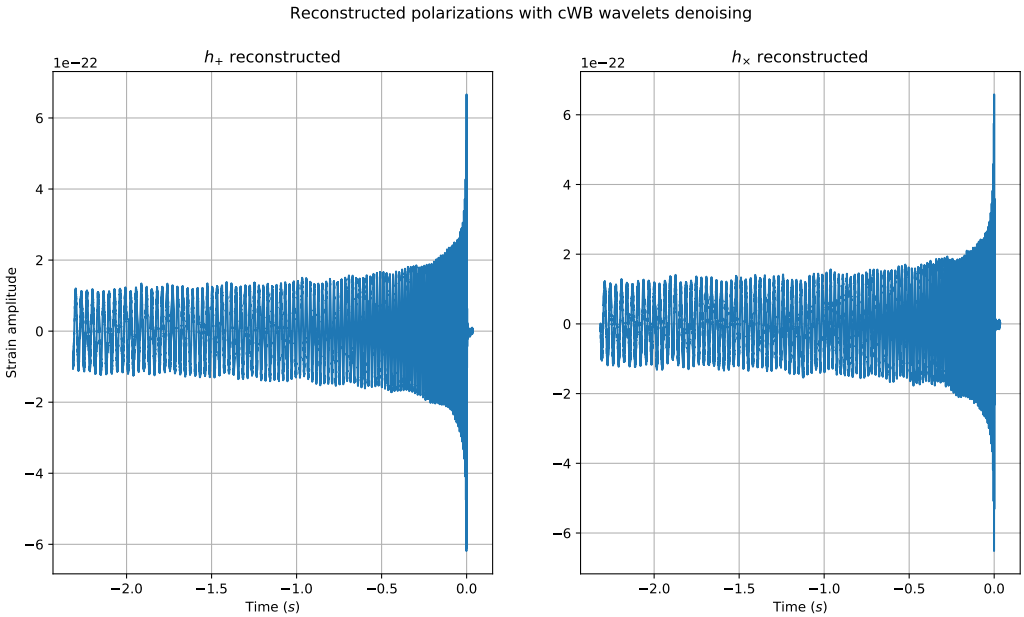


Figure 4.8: Reconstructed polarizations with cWB-wavelets denoising algorithm for the simulated signal from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. The source location is $(\lambda, \varphi) = (0, 0)$ and scaling factors are 0.1 for L1 and H1 (low noise) and 3.3 for V1 (high noise).

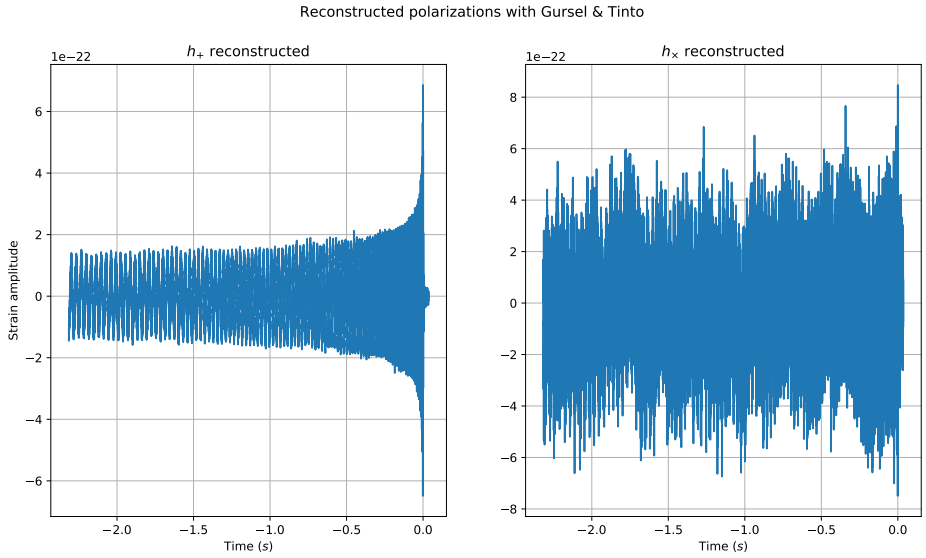


Figure 4.9: Reconstructed polarizations with Gürsel and Tinto algorithm for the simulated signal from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. The source location is $(\lambda, \varphi) = (0, 0)$ and scaling factors are 0.1 for L1 and H1 (low noise) and 3.3 for V1 (high noise).

4.2.2 Overlap analysis

The ability of an algorithm in reconstructing the original polarization content of a detected signal can be numerically characterized with the overlap between the injected waveform $h^{(i)}_{+,\times}$ and its reconstruction $h^{(r)}_{+,\times}$

$$O_{+,\times}(h^{(i)}_{+,\times}, h^{(r)}_{+,\times}) = \frac{(h^{(i)}_{+,\times}, h^{(r)}_{+,\times})}{\sqrt{(h^{(i)}_{+,\times}, h^{(i)}_{+,\times})(h^{(r)}_{+,\times}, h^{(r)}_{+,\times})}} \tag{4.10}$$

where the scalar product (a, b) is defined as $(a, b) = \int_{t_1}^{t_2} a(t)b(t)dt$. The overlap varies from -1 (signals in opposition) to $+1$ (perfect match) [37], and the closer the overlap is to $+1$ the better the reconstruction.

In my tests I evaluate the overlap for each algorithm while varying the noise scaling factor, and consider a wide range of factors – from low to high noise level. As explained above, the scaling factor is the same for L1 and H1, while in V1 it is larger by a factor 3.3.

Since the polarizations reconstruction depends on the antenna patterns and therefore on the source location [6] for each scaling factor I repeat the calculations for many uniformly distributed sky directions. The main purpose of this Section is to evaluate the polarization reconstruction error. I also isolate the polarization reconstruction uncertainty from that of the source direction, as I assume that the latter is known[6].

The results for each algorithm are reported in figures 4.10-4.12,4.15-4.17: in particular, they show that by fixing the scaling factor different source locations are associated to

[6]As shown in ref. [37], a wrong sky localization introduces a time shift between the the injected and the reconstructed waveforms, even though the distortions on waveform shape are negligible.

significantly different overlaps for the same algorithm.

To compare the algorithms for each scaling factor, the arithmetic mean of the overlaps for different source locations is evaluated: the results are reported in figures 4.13 and 4.18.

It is clear that the best performance is obtained by the cWB-wavelet denoising algorithm, while the Gürsel and Tinto algorithm has the worst performance: in particular, its mean overlap starts to deviate from 1 with a noise amplitude that is about one order of magnitude smaller than the corresponding noise amplitude for the cWB-wavelet denoising method.

A comparison is reported as a function of the SNR also (figures 4.14 and 4.19), calculated as

$$\mathrm{SNR} = \frac{\sum_k \sum_i \tilde{\xi}_k^2(t_i)}{\sum_k \sum_i n_k^2(t_i)} = \frac{\sum_k \sum_i \left|\tilde{\xi}_k(f_i)\right|^2}{\sum_k \sum_i \left|n_k(f_i)\right|^2} \tag{4.11}$$

where k is the detector index and i the time/frequency one as usual. Of course also in this representation the cWB-wavelet denoising performs better, having an overlap close to 1 at lower SNR values with respect to the other algorithms.

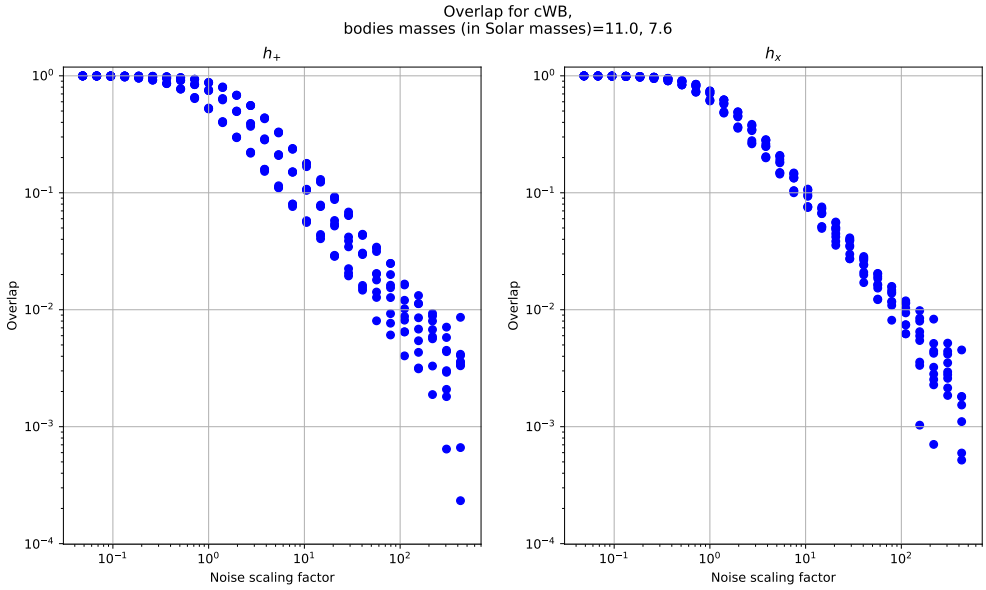


Figure 4.10: Overlap factor for cWB algorithm. The simulated polarizations are those from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

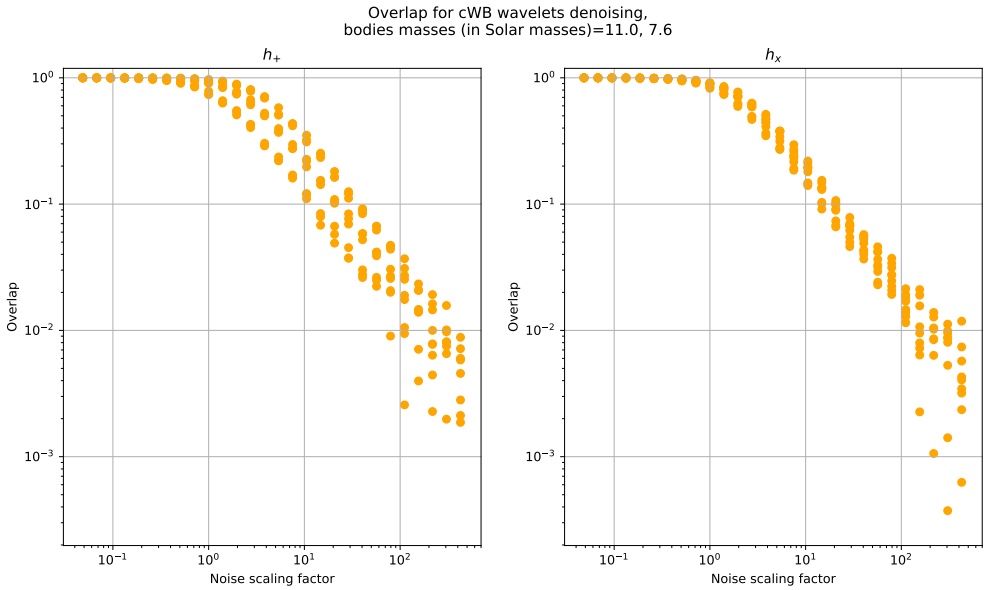


Figure 4.11: Overlap factor for cWB wavelet denoising algorithm. The simulated polarizations are those from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

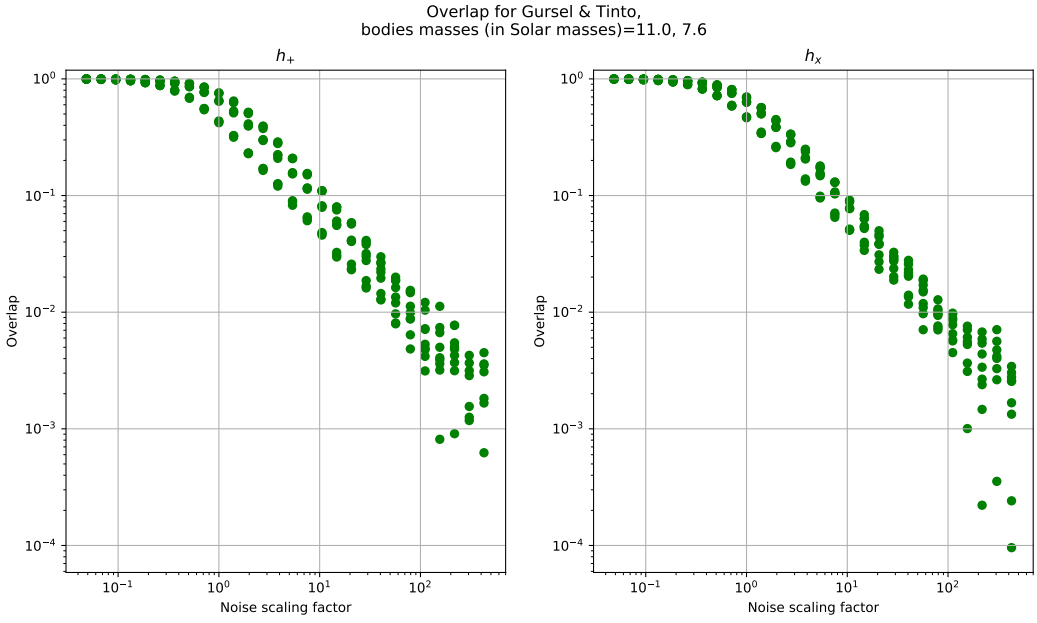


Figure 4.12: Overlap factor for Gürsel and Tinto algorithm. The simulated polarizations are those from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

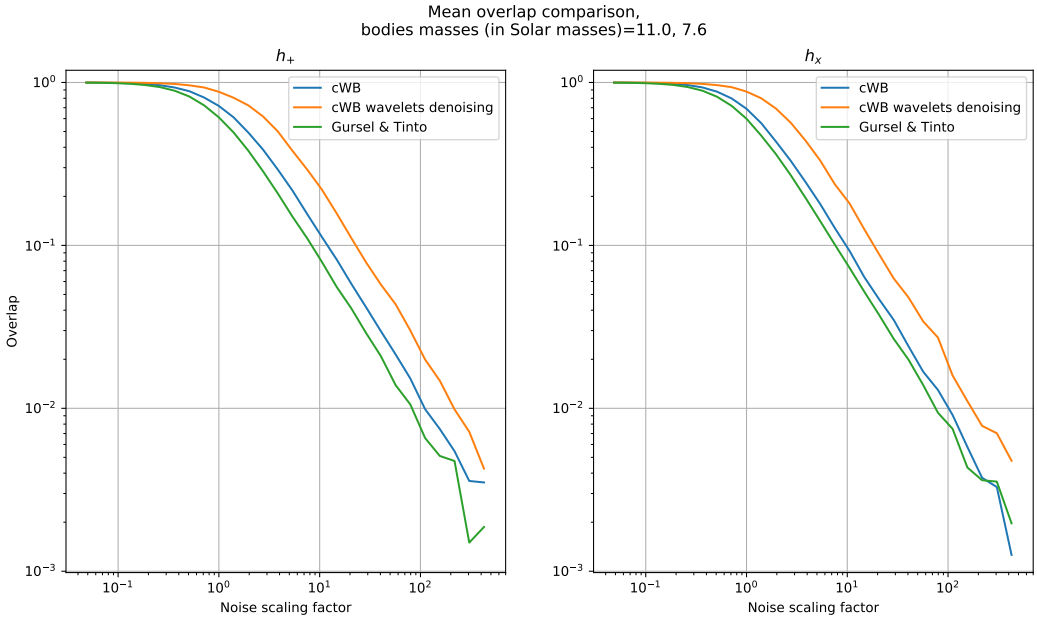


Figure 4.13: Mean overlap factor for the tested algorithms. The mean was calculated by taking the mean overlap value for the various sky directions, for each scaling factor. The simulated polarizations are those from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$.

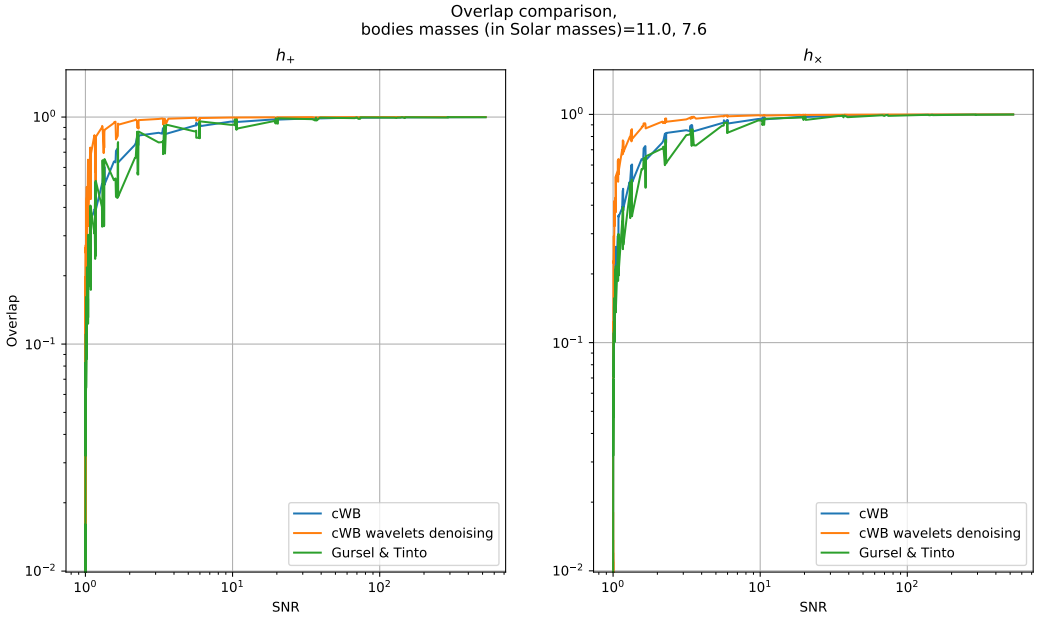


Figure 4.14: Overlap comparison in function of SNR. The simulated polarizations are those from the BBH with masses $11.0M_\odot$ and $7.6M_\odot$.

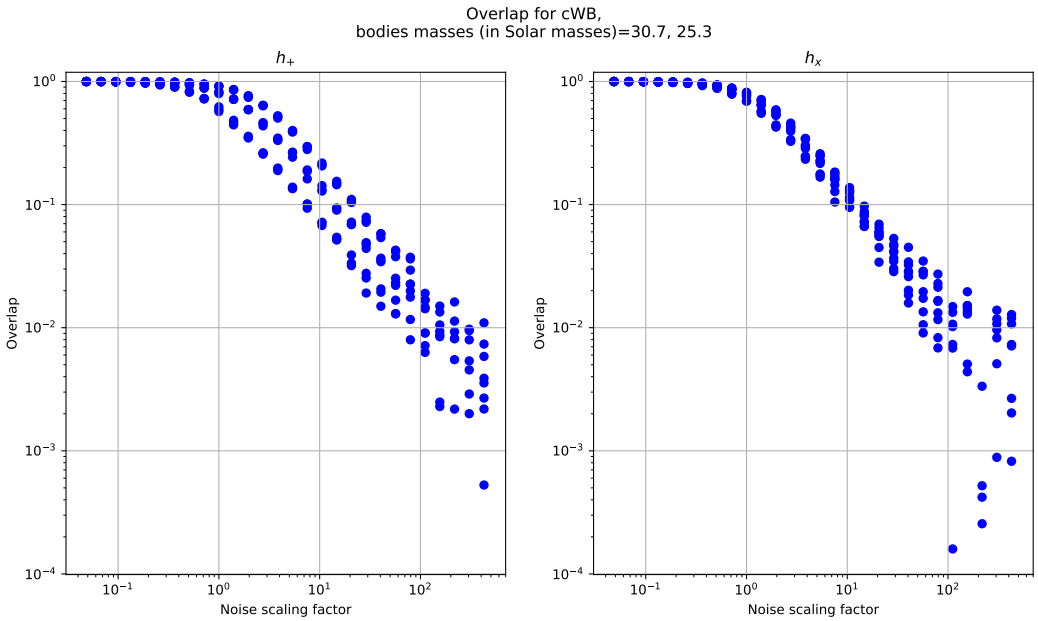


Figure 4.15: Overlap factor for cWB algorithm. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

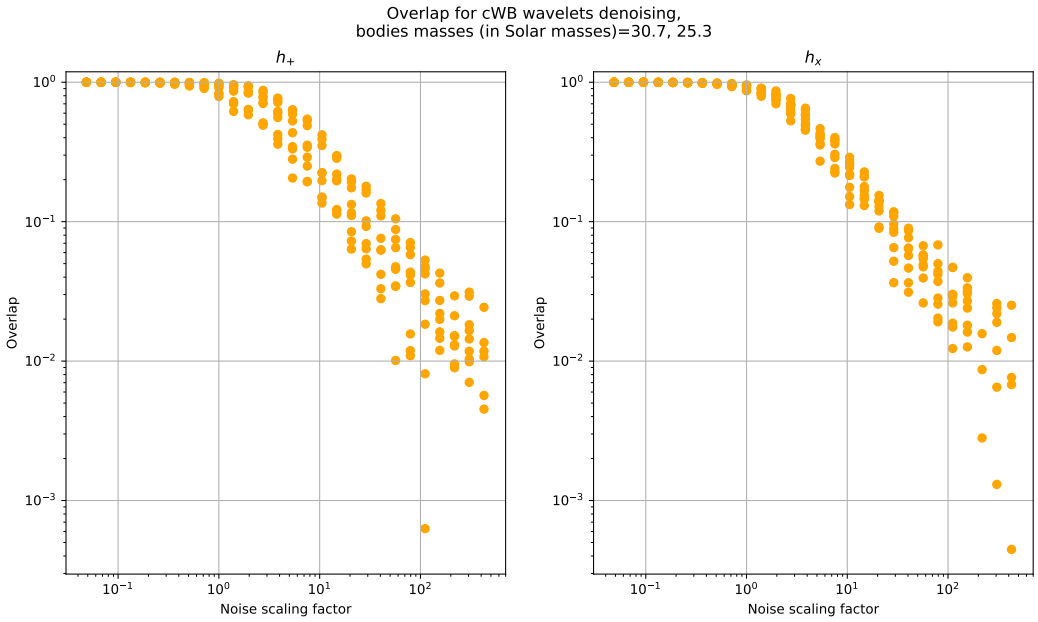


Figure 4.16: Overlap factor for cWB wavelet denoising algorithm. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

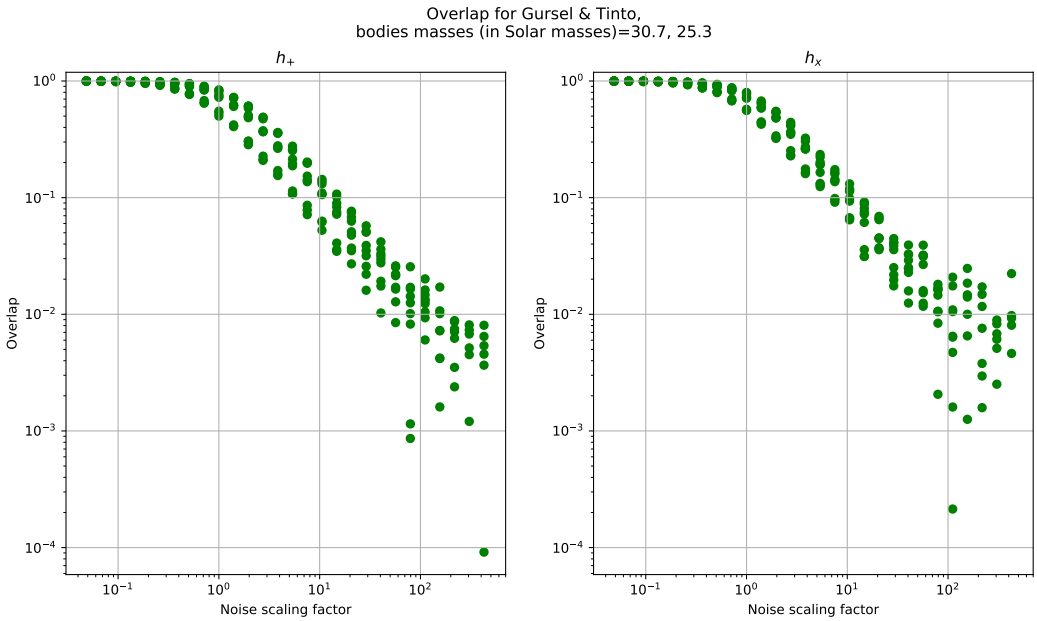


Figure 4.17: Overlap factor for Gürsel and Tinto algorithm. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$. For each scaling factor many directions uniformly distributed in the sky are evaluated.

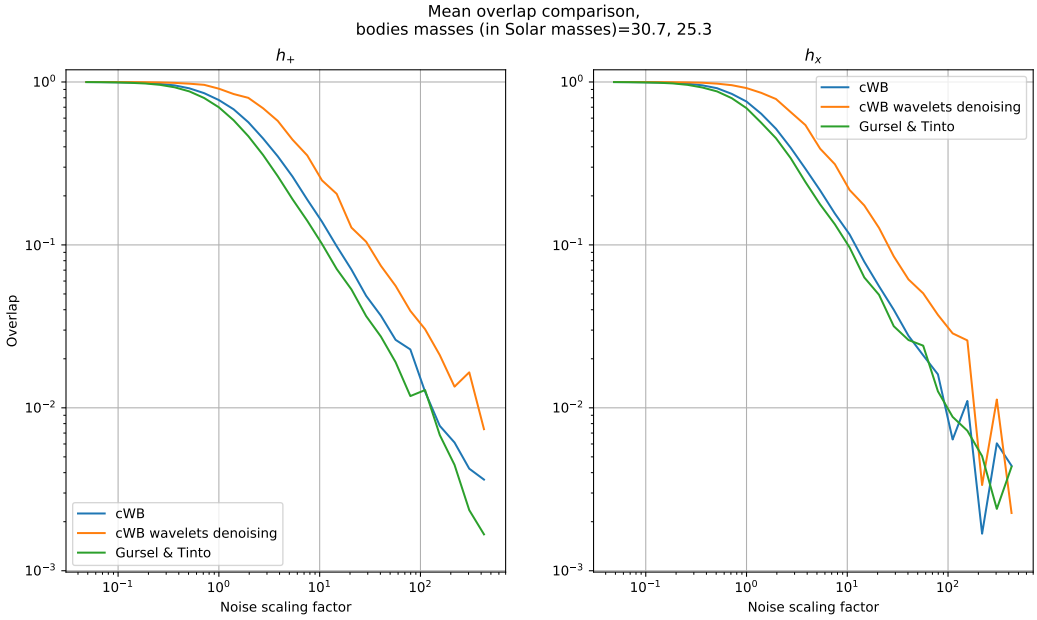


Figure 4.18: Mean overlap factor for the tested algorithms. The mean was computed by taking the mean overlap value for the various sky directions, for each scaling factor.
The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$.

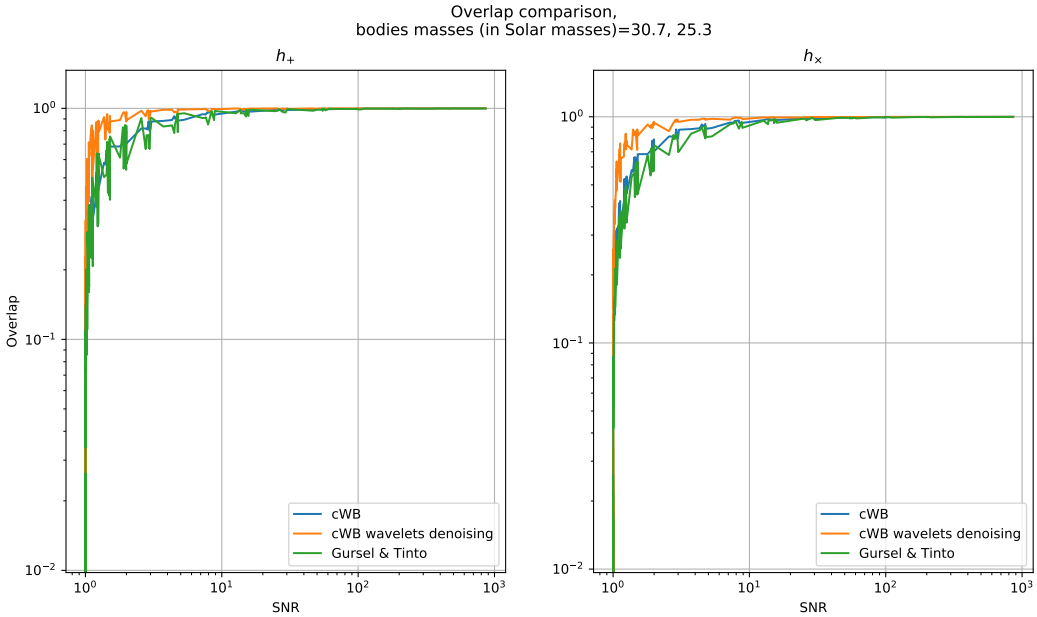


Figure 4.19: Overlap comparison in function of SNR. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$.

4.3 Source location identification

4.3.1 An intuitive introduction based on delay times

To obtain the source direction of a detected signal, a network of at least three detectors is needed [36]. This is because a network of N detectors provides N measurements and $N-1$ independent delay times[7] and therefore at least two independent time delays are needed to find the unknown θ_E and ϕ_E from the equation involving the relative time delay between detectors i and j and the source direction $\hat{\mathbf{n}}(\theta_E, \phi_E)$[8]:

$$\tau_{ij}(\theta_E, \phi_E) = (\mathbf{r}_i - \mathbf{r}_j) \cdot \frac{\hat{\mathbf{n}}(\theta_E, \phi_E)}{c}. \tag{4.12}$$

This can also be explained with a geometrical interpretation: for each delay time between two detectors, the locus of points with fixed delay time defines a circle on the celestial sphere (fig. 4.21): with three detectors – and two independent delay times – two possible source locations are found which are mirror images with respect to the plane defined by the three detectors. To further solve this ambiguity additional data are required; the obvious solution is to add a detector, so that with four detectors the mirror degeneracy is removed.

A method based on delay times was also described in the paper by Gürsel and Tinto [35], but this is highly affected by timing uncertainty [36], so that in this work I use the method implemented in the cWB pipeline.

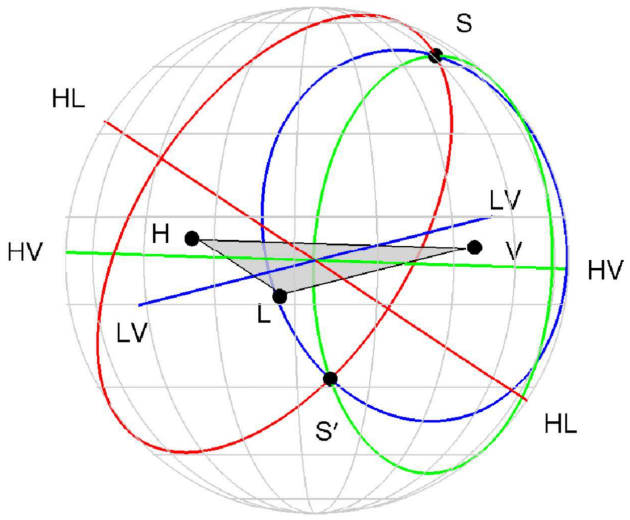


Figure 4.20: Geometrical interpretation of source location identification with delay times. The figure is taken from [36].

[7]With three detectors there are 5 constrains, and then the parameter set $h_+, h_\times, \theta, \phi$ can be correctly determined [38].

[8]The geometry is always assumed to be the one described in Section 3.1, with the origin corresponding to the Earth center.

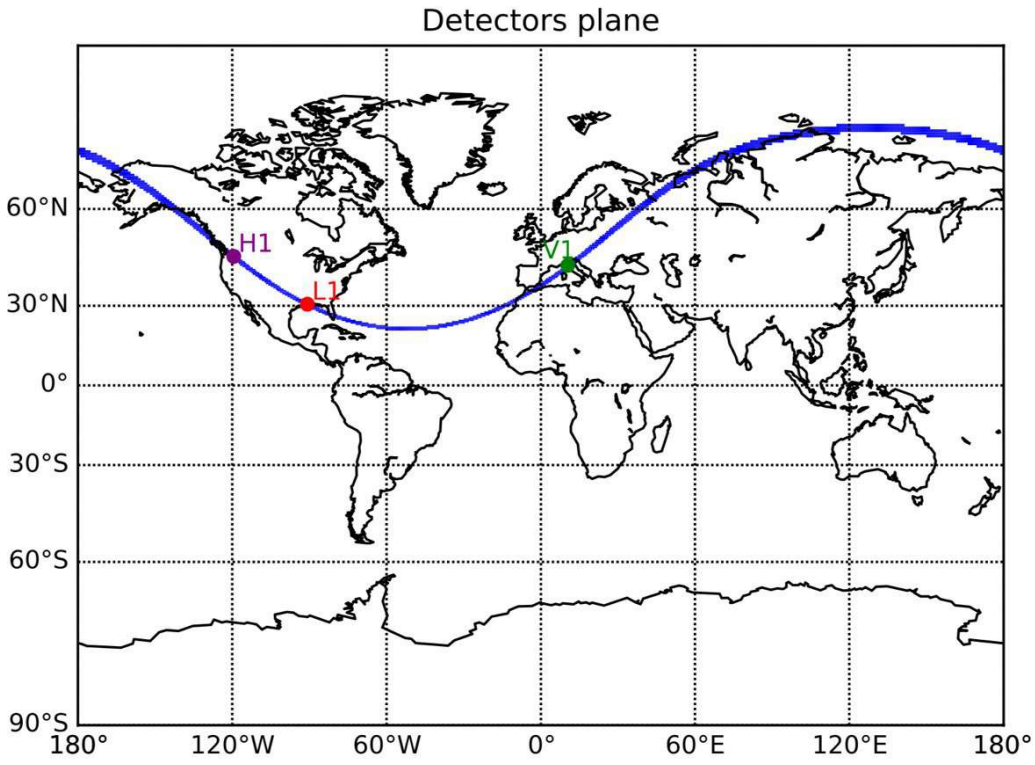


Figure 4.21: The plane (blue line) determined by the three interferometers H1, L1 and V1.

4.3.2 Sky location

As explained above (Section 3.4), the correct source location should correspond to the maximum of the sky statistic which takes into account both the maximum likelihood ratio statistic (eq. (3.62)) and the correlation coefficient[9] (eq. (3.65))

$$\mathcal{L}_{\rm sky}(\theta,\phi) = c_c(\theta,\phi)\mathcal{L}_{\rm max}(\theta,\phi) \tag{4.13}$$

It is important to note that $\mathcal{L}_{\rm sky}$ does not change under the transformation

$$\theta \to \pi - \theta \tag{4.14}$$

$$\phi \to \phi - \pi \quad \text{if} \quad \phi > 0, \quad \phi + \pi \quad \text{if} \quad \phi < 0. \tag{4.15}$$

In fact, this transformation affects the antenna patterns as follows[10]

$$\begin{aligned} F_+(\theta,\phi) &= \frac{1}{2}(1+\cos^2\theta)\cos 2\phi \to \frac{1}{2}(1+\cos^2(\pi-\theta))\cos(2\phi \pm 2\pi) = F_+(\theta,\phi) \\ F_\times(\theta,\phi) &= \cos\theta \sin 2\phi \to \cos(\pi-\theta)\sin(2\phi \pm 2\pi) = -F_\times(\theta,\phi) \end{aligned} \tag{4.16}$$

[9]To obtain quantities $\mathcal{L}_{\rm max}$ and $\mathcal{L}_{\rm sky}$ that are of order 1, I divide ("normalize") for the total energy associated to the whitened data, namely $E_t = \sum_i |\mathbf{w}[i]|^2$.

[10]Note that the antenna patterns involve the angles in each detector reference frame (θ_d, ϕ_d), but these are linked to the Earth frame angles (θ_E, ϕ_E) by the linear transformation involving the Euler angles, so the transformations $\theta \to \pi - \theta$,$\phi \to \phi \pm \pi$ are the same.

and so the projector $\mathcal{P}_{nm}(\theta,\phi)$ does not change

$$\mathcal{P}_{nm}(\theta,\phi) \to e_{n+}(\theta,\phi)e_{m+}(\theta,\phi) + (-e_{n\times}(\theta,\phi))(-e_{m\times}(\theta,\phi)) = \mathcal{P}_{nm}(\theta,\phi). \tag{4.17}$$

The main consequence of this is that $\mathcal{L}_{\rm sky}$ has two indistinguishable maxima: one at the correct source location (θ,ϕ) and the other one corresponding to the incorrect location $(\pi-\theta,\phi\pm\pi)$: to remove this degeneracy it is sufficient to analyze the delay times. This can be done starting from the source location vector $\hat{\mathbf{n}}(\theta,\phi)$, which transforms as

$$\hat{\mathbf{n}}(\theta,\phi) \to (-\sin\theta\cos\phi, -\sin\theta\sin\phi, -\cos\phi) = -\hat{\mathbf{n}}(\theta,\phi) \tag{4.18}$$

then comparing the measured delay times with the one calculated with the two coordinates (θ,ϕ) and $(\pi-\theta,\phi\pm\pi)$ the correct source location can be found.

I obtain the correct source location of a simulated signal by calculating the sky statistic on a grid of uniformly distributed positions in the sky. This presents some problems because the calculation of the sky statistic for each position is computationally expensive and doing this for each pixel (e.g. for a resolution of 1° $180\times360 = 64800$ pixels are required) could take a very long time, so I make some reasonable assumptions to obtain an acceptable execution time.

First, I decide to sample the sky in $36 \times 72 = 2592$ pixels, and for each of these the corresponding value of $\mathcal{L}_{\rm sky}$ is calculated taking the angles corresponding to the centre of the corresponding pixel.

For the purposes of this work, i.e., tests of source location, I downsample the data to a sample rate of 2048 Hz[11]: so the Nyquist frequency is 1024 Hz, sufficiently high for compact binaries frequencies.

Moreover, the (real) data in the frequency domain can be represented by the positive frequencies only when calculating $\mathcal{L}_{\rm max}$: in fact, just as for any real signal, $w(-f) = w(f)^*$ and recalling that the likelihood is $\mathcal{L}_{\rm max} = \sum_i \sum_{nm} w_n[i]P_{nm}[i]w^*_m[i]$ I find that the following equality holds

$$\sum_{nm} w^*_n[i]\mathcal{P}_{nm}[i]w_m[i] = \sum_{nm} w_n[i]\mathcal{P}_{nm}[i]w^*_m[i]. \tag{4.19}$$

Again, to reduce computing time I consider in this analysis the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, only because the data array associated to this has fewer samples than the one of the BBH with masses $11.6M_\odot$ and $7.6M_\odot$ and so its analysis requires a shorter time (by about a factor of 5).

So, once the coordinates associated with the two pixels with the maximum value of $\mathcal{L}_{\rm sky}$ are found, a gradient ascent is implemented to correctly find the two maxima: then the correct source location between the two is chosen as explained above.

If the sky is not densely sampled, the gradient ascent can lead to a relative maximum of $\mathcal{L}_{\rm sky}$ and not to the correct one, so this algorithm requires a good starting sky partition: e.g., having 36×72 starting pixels sometimes is not sufficient, especially if the correct source location is not near the center of a pixel[12]. This can be seen in the following two examples:

[11] The usual sample rate of LIGO Virgo detectors is 16 384 Hz [47].

[12] With 36×72 pixels the evaluated latitudes are $+87.5^\circ, +82.5^\circ, +77.5^\circ, ..., -77.5^\circ, -82.5^\circ, -87.5^\circ$ while the longitudes are $-177.5^\circ, -172.5^\circ, -167.5^\circ, ..., +167.5^\circ, +172.5^\circ, +177.5^\circ$

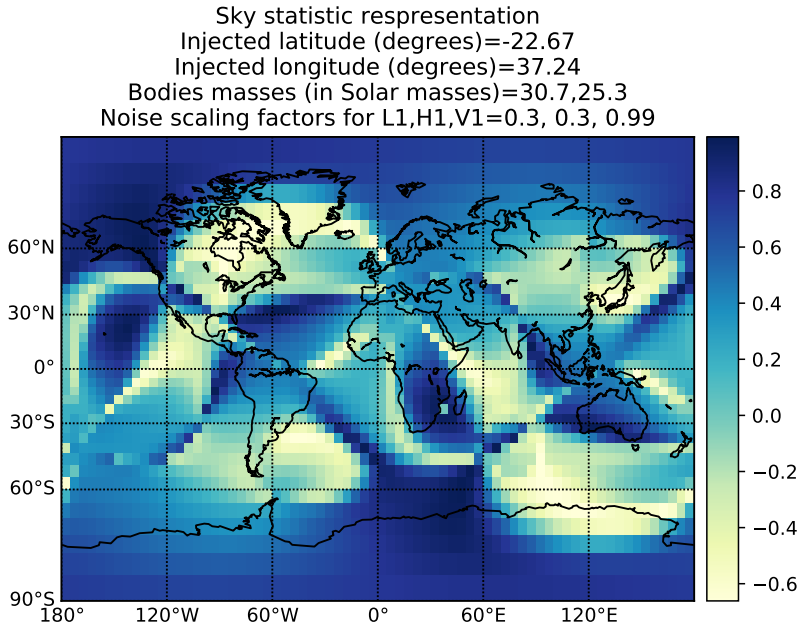


Figure 4.22: Sky statistic representation of the signal simulated with $(\lambda_i, \varphi_i) = (-22.67°, +37.24°)$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

in these examples to obtain a reasonable SNR (in the range 20-30) the noise scaling factors are 0.3 for LIGO detectors and 0.99 for Virgo[13].

In the first example a simulation is done assuming $(\lambda_i, \varphi_i) = (-22.67°, +37.24°)$ (and so near the pixel centered in $(-22.5°, +37.5°)$). The reconstructed coordinates are found to be $(\lambda_r, \varphi_r) = (-22.41°, +37.15°)$ and the algorithm performs quite well $((\lambda_r - \lambda_i, \varphi_r - \varphi_i) = (\Delta\lambda, \Delta\varphi) = (+0.26°, -0.09°))$. The original polarizations are correctly reconstructed $(O_+ = 0.998, \quad O_\times = 0.999)$. The results are reported in figures 4.22,4.23 and 4.24.

In the second example a simulation is done assuming $(\lambda_i, \varphi_i) = (+45.34°, -132.67°)$ (and so quite far from the nearest pixel center $(+47.5°, -132.5°)$). The reconstructed coordinates are found to be $(\lambda_r, \varphi_r) = (+34.34°, -161.37°)$ and so $((\Delta\lambda, \Delta\varphi) = (-10.99°, -28.70°))$. This incorrect source location identification of course affects the reconstruction of the original polarizations also $(O_+ = 0.902, O_\times = 0.995)$. The results are reported in figures 4.25,4.26 and 4.27.

[13]It is straightforward that the higher the noise the lower the algorithm ability to reconstruct the correct source location.

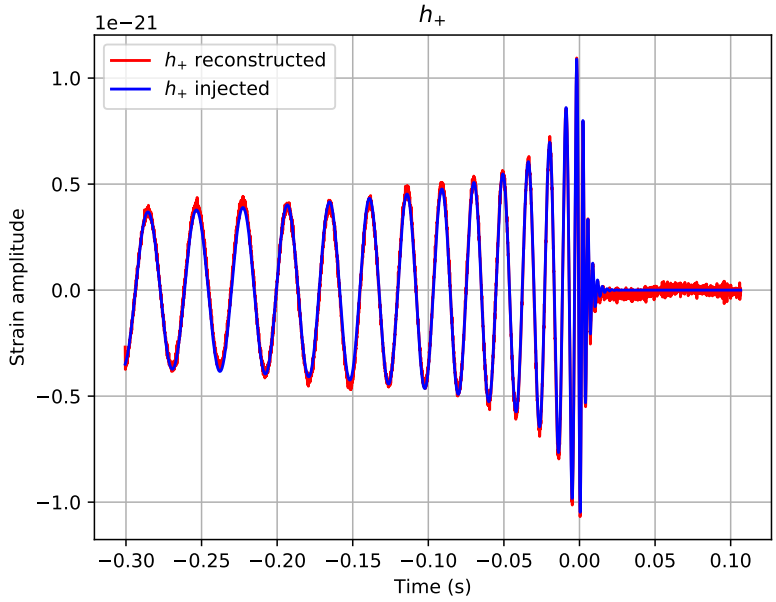


Figure 4.23: Reconstructed (red line) and injected (blue line) h_+ polarization of the signal simulated with $(\lambda_i, \varphi_i) = (-22.67°, +37.24°)$. The overlap is $O_+ = 0.998$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

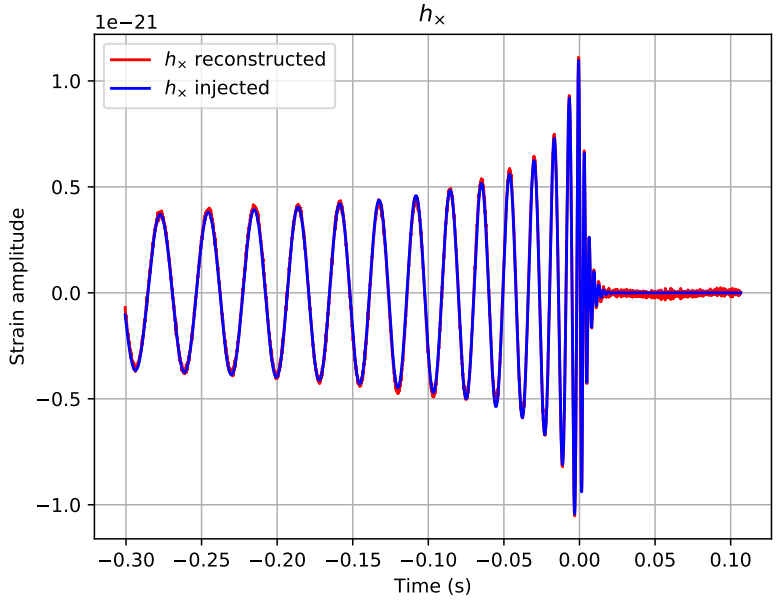


Figure 4.24: Reconstructed (red line) and injected (blue line) $h_\times$ polarization of the signal simulated with $(\lambda_i, \varphi_i) = (-22.67°, +37.24°)$. The overlap is $O_\times = 0.999$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

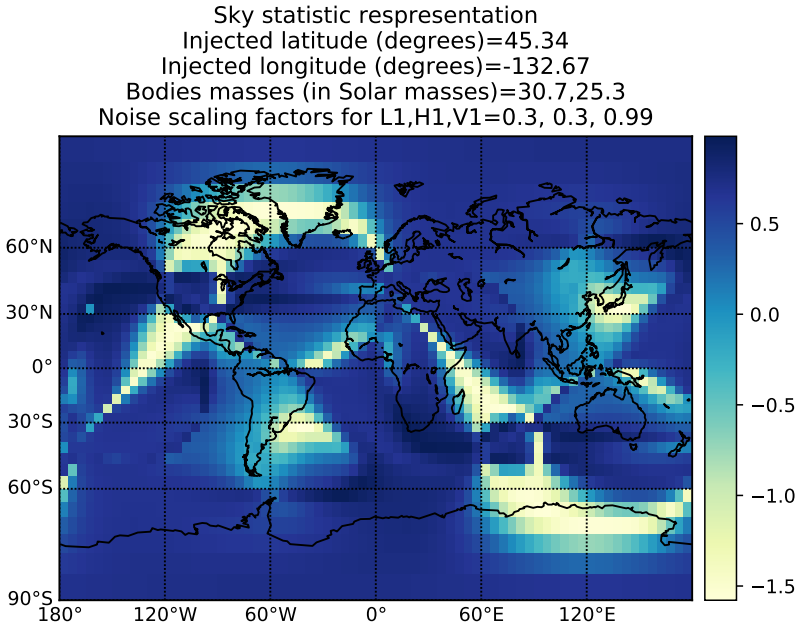


Figure 4.25: Sky statistic representation of the signal simulated with $(\lambda_i, \varphi_i) = (45.34°, -132.67°)$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

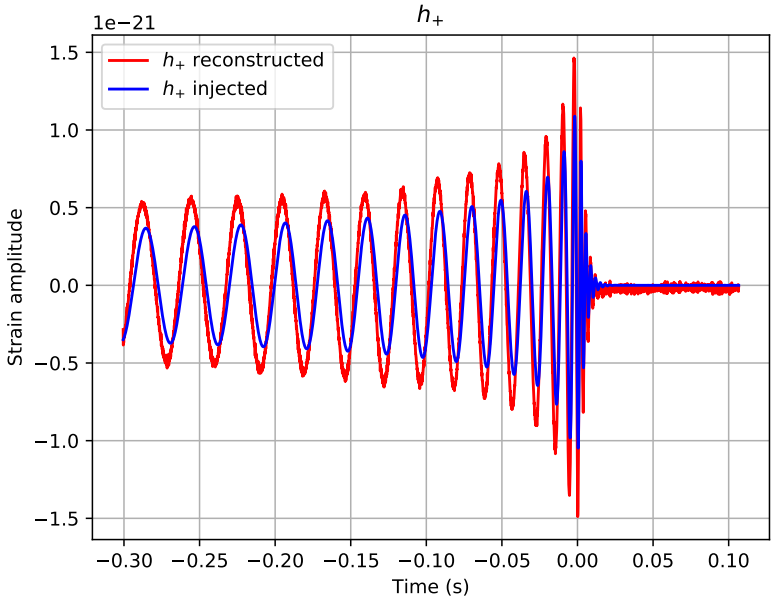


Figure 4.26: Reconstructed (red line) and injected (blue line) h_+ polarization of the signal simulated with $(\lambda_i, \varphi_i) = (+45.34°, -132.67°)$. The overlap is $O_+ = 0.902$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

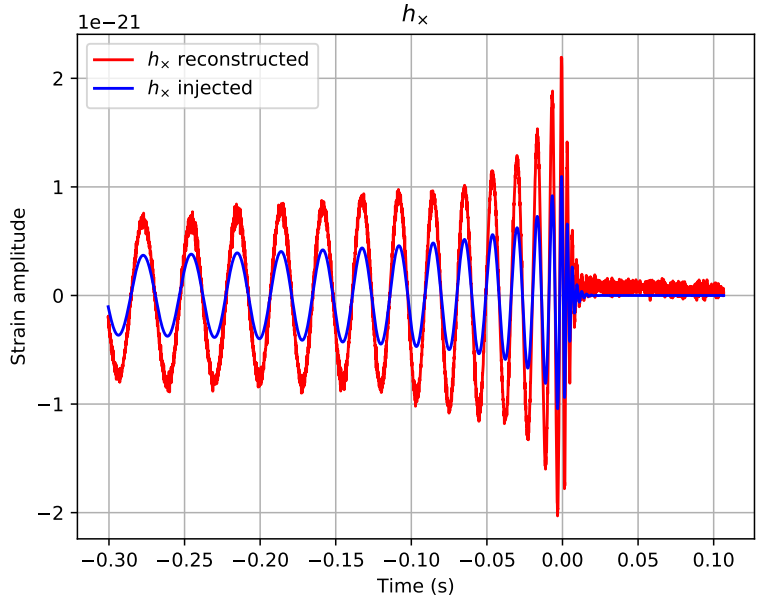


Figure 4.27: Reconstructed (red line) and injected (blue line) $h_\times$ polarization of the signal simulated with $(\lambda_i, \varphi_i) = (+45.34°, -132.67°)$. The overlap is $O_\times = 0.995$. The simulated polarizations are those from the BBH with masses $30.7M_\odot$ and $25.3M_\odot$, while the noise scaling factors are assumed to be 0.3 for L1, H1 and 0.99 for V1.

Chapter 5

Conclusions

The extraction of the gravitational waveform from a detected GW signal, has been studied for years, and many solutions have been developed. They can be divided in two main groups, namely modeled and unmodeled: modeled solutions infer the gravitational waveform parameters from numerical approximants of large sets of waveforms computed with numerical relativity codes, while the unmodeled methods estimate the signal shape without making no (or only very weak) assumptions. While unmodeled methods cannot directly provide information on intrinsic parameters like spins and masses, they can be used to estimate some of the extrinsic parameters of the detected waves, such as their polarization content.

In this work I have studied and tested unmodeled solutions to the problem of polarization extraction. The first one was developed by Gürsel and Tinto in 1989, at a time when the LIGO project still had to obtain funding on a regular basis. Even though this solution represents a milestone in the field of unmodeled methods it has shortcomings that have been discussed in this Thesis, above all the fact that the reconstruction ability of the algorithm is limited by the least sensitive detector.

After Gürsel and Tinto paper several other methods were developed, implementing new approaches and techniques. Above all, it was understood that an analysis in the frequency domain (and not in the time one only, as in Gürsel and Tinto paper) is more efficient and allows for the application of many useful and well known techniques of signal processing, such as data whitening.

An efficient implementation of these techniques was developed in the framework of the cWB pipeline, whose polarization disentangling algorithm is mainly based on a statistical approach involving the likelihood function. As shown in this work, the cWB method is able to reconstruct the GW polarizations better then the method of Gürsel and Tinto: above all, the final result does not depend on the least sensitive detector, which is a good feature of the method, because it allows for the introduction of new, less sensitive detectors (as it was for Virgo in 2017 and as it will be for KAGRA in 2021/2022) without being crippled by their lower sensitivity.

Still, the cWB algorithm loses its sensitivity to polarization for signals with low SNR: to increase its polarization sensitivity for low-SNR signals I implemented a new denoising method involving a wavelet transformation within the Python frame of the reconstruction program. This method appears to increase the polarization sensitivity, but the results are not

exhaustive and further studies are required: e.g., it must be checked how much this method reduces the effects of noise and above all how much it could distort the original polarization content of the signal, which would of course be a dangerous shortcoming. Eventually, if the method turns out to be efficient it could increase the cWB ability to reconstruct weak signals, such as the those expected from CCSNe.

In addition to denoising techniques, other solutions of the inverse problem can be developed assuming different theoretical frameworks: e.g., within cWB the polarization disentangling is mainly based on the previously described likelihood approach, which mainly consist in the maximization of a likelihood ratio. Of course other ideas could be developed, considering some quantities where maximization/minimization can give the correct solution for the polarizations. Further studies in this field will be done in a future research work, which hopefully will increase our ability to extract information from gravitational wave data, and maybe help a little in extending our knowledge of the Universe.

Appendix: Python code

Here I report the Python codes used in this Thesis: they can be divided in three main groups, namely modules, general simulations and overlap simulations.

There are two modules, used almost throughout the code, that implement general functions useful for signal analysis and noise generation

- `polarization_module.py`: this implements functions used in signal generation, e.g., those which calculate the antenna patterns, such as those used in polarization reconstruction and analysis, e.g. the Dominant Polarization Frame transformation, the cWB polarization reconstruction method, the overlap calculation and others; finally all the functions for the source direction representation are reported, including the auxiliary ones as signal downsampling and gradient ascent

- `noise_module.py`: this implements all the formulas reported in Section 2.3 and the algorithm described in [44] for noise generation; a function for calculating the amplitude spectral density (i.e. the square root of the power spectral density, used for data whitening) is included as well

The general simulation code implements the three algorithms described in this Thesis: the masses (in units of Solar masses) are a required input to generate the waveforms from coalescing binary system, along with the luminosity distance from the Earth (in Mpc) and with the source direction; moreover, the code allows for amplifying or reducing the noise in the detectors through a noise scaling factor

- `polarization_cWB.py` implements the reconstruction algorithm used in cWB pipeline

- `polarization_wavelets_denoising_and_source_location_identification.py` is a more general code, implementing the cWB method using wavelet denoising plus a part in which the code tries to find the correct source location (which is always assumed in the other pieces of code) and then starting from this it tries to reconstruct the polarizations of the detected signal

- `polarization_Gursel_Tinto.py` implements the reconstruction algorithm proposed by Yekta Gürsel and Massimo Tinto

The overlap methods `polarization_wavelets_denoising_overlap.py`, `polarization_cWB_overlap.py` and `polarization_Gursel_Tinto_overlap.py` calculate for each algorithm the overlaps between injected and reconstructed waveforms, for many

source directions and many noise scaling factors (and so, many SNRs). All the results are saved in `.txt` files which can be loaded by the `overlap_comparison.py` code that generates graphs in which the results of the three algorithms are compared.

polarization_module.py

```
import numpy as np
import scipy
import scipy.constants as constants
from scipy import fftpack

# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.98847e30                             # kg
pi=constants.pi


# note: the geometry is the one described in Chapter 3


#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.



#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.



# note: the Earth radii are taken considering the altitude at each site

#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
    phi_V1), 6368239.*np.cos(theta_V1)])



def Euler_matrix_alpha(alpha):
  return np.array([[np.cos(alpha), np.sin(alpha), 0.],[-np.sin(alpha), np.cos(alpha),
    0.],[0. , 0. , 1.]])
```

```
62
63
64 def Euler_matrix_beta(beta):
65   return np.array([[1. , 0. , 0.],[0. , np.cos(beta), np.sin(beta)],[0. , -np.sin(beta) , np
       .cos(beta)]])
66
67
68 def Euler_matrix_gamma(gamma):
69   return np.array([[np.cos(gamma), np.sin(gamma), 0.],[-np.sin(gamma), np.cos(gamma),
       0.],[0. , 0. , 1.]])
70
71
72 def Euler_transformation(alpha,beta,gamma):
73   R_alpha=np.array([[np.cos(alpha), np.sin(alpha), 0.],[-np.sin(alpha), np.cos(alpha),
       0.],[0. , 0. , 1.]])
74   R_beta=np.array([[1. , 0. , 0.],[0. , np.cos(beta), np.sin(beta)],[0. , -np.sin(beta) , np
       .cos(beta)]])
75   R_gamma=np.array([[np.cos(gamma), np.sin(gamma), 0.],[-np.sin(gamma), np.cos(gamma),
       0.],[0. , 0. , 1.]])
76   intermediate=np.matmul(R_gamma,R_beta)
77   return np.matmul(intermediate, R_alpha)
78
79
80 L1_Euler_transformation=Euler_transformation(alpha_L1,beta_L1,gamma_L1)
81 H1_Euler_transformation=Euler_transformation(alpha_H1,beta_H1,gamma_H1)
82 V1_Euler_transformation=Euler_transformation(alpha_V1,beta_V1,gamma_V1)
83
84
85 def time_delay(r_k,n):
86   return np.dot(r_k,n)/c
87
88
89 def F_plus_k(cos_theta_k,cos_2_phi_k):
90   return 0.5*(1.+(cos_theta_k**2))*cos_2_phi_k
91
92
93 def F_cross_k(cos_theta_k,sin_2_phi_k):
94   return cos_theta_k*sin_2_phi_k
95
96
97 def h_plus_function(t,omega_gw,amplitude_constant):
98   h_plus=amplitude_constant*np.cos(omega_gw*t)
99   return h_plus
100
101 def h_cross_function(t,omega_gw,amplitude_constant):
102   h_cross=amplitude_constant*np.sin(omega_gw*t)
103   return h_cross
104
105
106
107 # this function brings the antenna patterns in the Dominant Polarization Frame
108 def DPF_antenna_patterns(f_plus_L1,f_plus_H1,f_plus_V1,f_cross_L1,f_cross_H1,f_cross_V1):
109   DPF_denominator=np.zeros(len(f_plus_L1))
110   cos_2_psi_DPF=np.zeros(len(f_plus_L1))
111   sin_2_psi_DPF=np.zeros(len(f_plus_L1))
112   cos_psi_DPF=np.zeros(len(f_plus_L1))
113   sin_psi_DPF=np.zeros(len(f_plus_L1))
114
115   for i in range(len(f_plus_L1)):
116     DPF_denominator[i]=np.sqrt((f_plus_L1[i]*f_plus_L1[i]+f_plus_H1[i]*f_plus_H1[i]+
       f_plus_V1[i]*f_plus_V1[i]-(f_cross_L1[i]*f_cross_L1[i]+f_cross_H1[i]*f_cross_H1[i]+
       f_cross_V1[i]*f_cross_V1[i]))**2+(2.*(f_plus_L1[i]*f_cross_L1[i]+f_plus_H1[i]*f_cross_H1
       [i]+f_plus_V1[i]*f_cross_V1[i]))**2)
117     cos_2_psi_DPF[i]=((f_plus_L1[i]*f_plus_L1[i]+f_plus_H1[i]*f_plus_H1[i]+f_plus_V1[i]*
       f_plus_V1[i]-(f_cross_L1[i]*f_cross_L1[i]+f_cross_H1[i]*f_cross_H1[i]+f_cross_V1[i]*
       f_cross_V1[i])))/DPF_denominator[i]
118     sin_2_psi_DPF[i]=((2.*(f_plus_L1[i]*f_cross_L1[i]+f_plus_H1[i]*f_cross_H1[i]+f_plus_V1[i
       ]*f_cross_V1[i])))/DPF_denominator[i]
```

```
    cos_psi_DPF[i]=np.cos(0.5*np.arctan2(sin_2_psi_DPF[i],cos_2_psi_DPF[i]))
    sin_psi_DPF[i]=np.sin(0.5*np.arctan2(sin_2_psi_DPF[i],cos_2_psi_DPF[i]))

    f_plus_L1_DPF=np.zeros(len(f_plus_L1))
    f_plus_H1_DPF=np.zeros(len(f_plus_H1))
    f_plus_V1_DPF=np.zeros(len(f_plus_V1))
    f_cross_L1_DPF=np.zeros(len(f_cross_L1))
    f_cross_H1_DPF=np.zeros(len(f_cross_H1))
    f_cross_V1_DPF=np.zeros(len(f_cross_V1))


  for i in range(len(f_plus_L1)):
    f_plus_L1_DPF[i]=cos_psi_DPF[i]*f_plus_L1[i]+sin_psi_DPF[i]*f_cross_L1[i]
    f_cross_L1_DPF[i]=-sin_psi_DPF[i]*f_plus_L1[i]+cos_psi_DPF[i]*f_cross_L1[i]
    f_plus_H1_DPF[i]=cos_psi_DPF[i]*f_plus_H1[i]+sin_psi_DPF[i]*f_cross_H1[i]
    f_cross_H1_DPF[i]=-sin_psi_DPF[i]*f_plus_H1[i]+cos_psi_DPF[i]*f_cross_H1[i]
    f_plus_V1_DPF[i]=cos_psi_DPF[i]*f_plus_V1[i]+sin_psi_DPF[i]*f_cross_V1[i]
    f_cross_V1_DPF[i]=-sin_psi_DPF[i]*f_plus_V1[i]+cos_psi_DPF[i]*f_cross_V1[i]

  return cos_psi_DPF,sin_psi_DPF,f_plus_L1_DPF,f_plus_H1_DPF,f_plus_V1_DPF,f_cross_L1_DPF,
    f_cross_H1_DPF,f_cross_V1_DPF



# the following function downsamples the signal: this is important for the sky statistic
    calculation
# the downsampling is done in time domain; in frequency domain only positive frequencies are
     taken for calculating the likelihood

def downsampling_in_time_domain(data,t,downsampling_factor):
  downsampled_data=np.array([data[0]])
  downsampled_t=np.array([t[0]])
  downsampling_factor=int(downsampling_factor)


  for i in range(1,len(data),downsampling_factor):
    downsampled_data=np.append(downsampled_data,np.array([data[i]]))
    downsampled_t=np.append(downsampled_t,np.array([t[i]]))

  return downsampled_t,downsampled_data


def positive_frequencies_selection(downsampled_Fourier_data,downsampled_frequency):
  positive_frequencies_downsampled_Fourier_data=np.array([downsampled_Fourier_data[0]])
  positive_frequencies=np.array([downsampled_frequency[0]])


  for i in range(1,len(downsampled_frequency)):
    if(downsampled_frequency[i]>0.):
      positive_frequencies_downsampled_Fourier_data=np.append(
    positive_frequencies_downsampled_Fourier_data,np.array([downsampled_Fourier_data[i]]))
      positive_frequencies=np.append(positive_frequencies,np.array([downsampled_frequency[i
    ]]))

  return positive_frequencies,positive_frequencies_downsampled_Fourier_data




# this function calculates the sky statistic as described in the thesis

def sky_statistic_calculation(theta,phi,L1_whitened,H1_whitened,V1_whitened,
    L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt):


  n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta
    )]) #incoming direction
```

```
179
180 n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
181 n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
182 n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
183
184 cos_theta_L1=n_L1[2]
185 cos_theta_H1=n_H1[2]
186 cos_theta_V1=n_V1[2]
187
188 cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
189 cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
190 cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
191
192 sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
193 sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
194 sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
195
196
197 f_plus_L1=F_plus_k(cos_theta_L1,cos_2_phi_L1)/L1_noise_power_spectral_density_sqrt
198 f_cross_L1=F_cross_k(cos_theta_L1,sin_2_phi_L1)/L1_noise_power_spectral_density_sqrt
199 f_plus_H1=F_plus_k(cos_theta_H1,cos_2_phi_H1)/H1_noise_power_spectral_density_sqrt
200 f_cross_H1=F_cross_k(cos_theta_H1,sin_2_phi_H1)/H1_noise_power_spectral_density_sqrt
201 f_plus_V1=F_plus_k(cos_theta_V1,cos_2_phi_V1)/V1_noise_power_spectral_density_sqrt
202 f_cross_V1=F_cross_k(cos_theta_V1,sin_2_phi_V1)/V1_noise_power_spectral_density_sqrt
203
204 cos_psi_DPF,sin_psi_DPF,f_plus_L1_DPF,f_plus_H1_DPF,f_plus_V1_DPF,f_cross_L1_DPF,
    f_cross_H1_DPF,f_cross_V1_DPF=DPF_antenna_patterns(f_plus_L1,f_plus_H1,f_plus_V1,
    f_cross_L1,f_cross_H1,f_cross_V1)
205
206
207 L_max=0.
208 E_c=0.
209 E_n=0.
210 E_i=0.
211 for i in range(len(L1_whitened)):
212   f_plus_DPF=np.array([f_plus_L1_DPF[i],f_plus_H1_DPF[i],f_plus_V1_DPF[i]])
213   f_cross_DPF=np.array([f_cross_L1_DPF[i],f_cross_H1_DPF[i],f_cross_V1_DPF[i]])
214   data=np.array([L1_whitened[i],H1_whitened[i],V1_whitened[i]])
215
216   f_plus_DPF_norm=np.linalg.norm(f_plus_DPF)
217   f_cross_DPF_norm=np.linalg.norm(f_cross_DPF)
218
219   # the likelihood is calculated as E_c+E_i
220   if(f_plus_DPF_norm!=0. and f_cross_DPF_norm!=0.):
221     # these are the normalized antenna patterns
222     e_plus_DPF=f_plus_DPF/f_plus_DPF_norm
223     e_cross_DPF=f_cross_DPF/f_cross_DPF_norm
224
225     psi=(np.dot(data,e_plus_DPF))*e_plus_DPF+(np.dot(data,e_cross_DPF))*e_cross_DPF
226     E_n=E_n+abs(np.linalg.norm(data-psi))**2
227
228     for m in range(len(e_plus_DPF)):
229       for n in range(len(e_plus_DPF)):
230         P_nm=e_plus_DPF[n]*e_plus_DPF[m]+e_cross_DPF[n]*e_cross_DPF[m]
231         if(m!=n):
232           E_c=E_c+data[n]*P_nm*np.conj(data[m])
233         elif(m==n):
234           E_i=E_i+data[n]*P_nm*np.conj(data[m])
235
236
237 E_c=E_c.real
238 E_i=E_i.real
239 cc=E_c/(E_c+E_n)
240
241
242 # I decide to divide the likelihood for the total detected energy
243 E_t=((np.linalg.norm(L1_whitened)**2)+(np.linalg.norm(H1_whitened)**2)+(np.linalg.norm(
    V1_whitened)**2))
```

```
  L_max=E_c+E_i
  sky_statistic=(L_max/E_t)*cc
  return sky_statistic


# this function calculates the sky statistic for a uniform distributed set of sky positions

def sky_statistic_sky_grid(theta_pixels,phi_pixels,data_L1,data_H1,data_V1,
    L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt):
  sky_statistic=np.zeros((theta_pixels,phi_pixels))
  L1_whitened=data_L1/L1_noise_power_spectral_density_sqrt
  H1_whitened=data_H1/H1_noise_power_spectral_density_sqrt
  V1_whitened=data_V1/V1_noise_power_spectral_density_sqrt

  print('Sky statistic calculation (it might be a while...)')

  count=0

  theta=np.arange(+0.5*pi/theta_pixels,+pi+0.5*pi/theta_pixels,+pi/theta_pixels)
  phi=np.arange(-pi+0.5*2.*pi/phi_pixels,+pi+0.5*2.*pi/phi_pixels,+2.*pi/phi_pixels)


  for m in range(theta_pixels):
    for n in range(phi_pixels):

      sky_statistic[m][n]=sky_statistic_calculation(theta[m],phi[n],L1_whitened,H1_whitened,
    V1_whitened,
        L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt)


      count=count+1

      if(count%100==0):
        print(round(100.*(count/(theta_pixels*phi_pixels)),2),'%')
  print('End sky statistic calculation')

  # here I find the two maxima of the sky statistic
  sky_statistic_max_1=np.max(sky_statistic)
  max_1_indices=np.unravel_index(np.argmax(sky_statistic, axis=None), sky_statistic.shape)
  theta_max_1=+0.5*(pi/theta_pixels)+max_1_indices[0]*(pi/theta_pixels)
  phi_max_1=-pi+0.5*2.*(pi/phi_pixels)+max_1_indices[1]*(2.*pi/phi_pixels)
  sky_statistic_matrix_for_max_2=sky_statistic
  sky_statistic_matrix_for_max_2[max_1_indices[0]][max_1_indices[1]]=0.
  sky_statistic_max_2=np.max(sky_statistic_matrix_for_max_2)
  max_2_indices=np.unravel_index(np.argmax(sky_statistic_matrix_for_max_2, axis=None),
    sky_statistic_matrix_for_max_2.shape)
  theta_max_2=+0.5*(pi/theta_pixels)+max_2_indices[0]*(pi/theta_pixels)
  phi_max_2=-pi+0.5*2.*(pi/phi_pixels)+max_2_indices[1]*(2.*pi/phi_pixels)

  return sky_statistic,theta_max_1,phi_max_1,sky_statistic_max_1,theta_max_2,phi_max_2,
    sky_statistic_max_2


# this function calculates the gradient of the sky statistic
def sky_statistic_gradient(theta,phi,h_theta,h_phi,L1_whitened,H1_whitened,V1_whitened,
    L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt):
  gradient_sky_statistic=np.array([0.,0.])
  sky_statistic_theta_h=sky_statistic_calculation(theta+h_theta,phi,L1_whitened,H1_whitened,
    V1_whitened,L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt)
  sky_statistic_phi_h=sky_statistic_calculation(theta,phi+h_phi,L1_whitened,H1_whitened,
    V1_whitened,L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
    V1_noise_power_spectral_density_sqrt)
  sky_statistic=sky_statistic_calculation(theta,phi,L1_whitened,H1_whitened,V1_whitened,
    L1_noise_power_spectral_density_sqrt,H1_noise_power_spectral_density_sqrt,
```

```
    V1_noise_power_spectral_density_sqrt)
  gradient_sky_statistic[0]=(sky_statistic_theta_h-sky_statistic)/h_theta
  gradient_sky_statistic[1]=(sky_statistic_phi_h-sky_statistic)/h_phi
  return gradient_sky_statistic


# this function implements a gradient ascent on the sky statistic

def sky_statistic_gradient_sky_grid(theta_max,phi_max,theta_pixels,phi_pixels,n_max,
    threshold,data_L1,data_H1,data_V1,L1_noise_power_spectral_density_sqrt,
    H1_noise_power_spectral_density_sqrt,V1_noise_power_spectral_density_sqrt):
  L1_whitened=data_L1/L1_noise_power_spectral_density_sqrt
  H1_whitened=data_H1/H1_noise_power_spectral_density_sqrt
  V1_whitened=data_V1/V1_noise_power_spectral_density_sqrt
  x_2=np.array([theta_max,phi_max])
  x_1=x_2+np.array([0.5*pi/theta_pixels,pi/phi_pixels])
  iteration=np.array([0])
  theta_path=np.array([theta_max])
  phi_path=np.array([phi_max])

  for i in range(n_max):

    h_theta=(pi/180.)*1.e-4
    h_phi=(pi/180.)*1.e-4
    sky_statistic_gradient_x_2=sky_statistic_gradient(x_2[0],x_2[1],h_theta,h_phi,
    L1_whitened,H1_whitened,V1_whitened,L1_noise_power_spectral_density_sqrt,
    H1_noise_power_spectral_density_sqrt,V1_noise_power_spectral_density_sqrt)
    sky_statistic_gradient_x_1=sky_statistic_gradient(x_1[0],x_1[1],h_theta,h_phi,
    L1_whitened,H1_whitened,V1_whitened,L1_noise_power_spectral_density_sqrt,
    H1_noise_power_spectral_density_sqrt,V1_noise_power_spectral_density_sqrt)
    step=abs(np.dot(x_2-x_1,sky_statistic_gradient_x_2-sky_statistic_gradient_x_1))/((np.
    linalg.norm(sky_statistic_gradient_x_2-sky_statistic_gradient_x_1))**2)
    x_3=x_2+step*sky_statistic_gradient_x_2
    iteration=np.append(iteration,np.array([i]))
    theta_path=np.append(theta_path,np.array([x_3[0]]))
    phi_path=np.append(phi_path,np.array([x_3[1]]))
    x_1=x_2
    x_2=x_3

    if(np.linalg.norm(x_2-x_1)<threshold):
      break

  theta_max=x_2[0]
  phi_max=x_2[1]
  return theta_max,phi_max,iteration,theta_path,phi_path







# this function reconstruct the polarizations of a detected GW signal
def polarization_reconstruction(theta,phi,Fourier_L1_response_whitened,
    Fourier_H1_response_whitened,Fourier_V1_response_whitened,
  Fourier_L1_noise_power_spectral_density_sqrt,Fourier_H1_noise_power_spectral_density_sqrt,
    Fourier_V1_noise_power_spectral_density_sqrt):


  n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta
    )]) #incoming direction

  n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
  n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
  n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)

  cos_theta_L1=n_L1[2]
  cos_theta_H1=n_H1[2]
  cos_theta_V1=n_V1[2]
```

```
356
357   cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
358   cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
359   cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
360
361   sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
362   sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
363   sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
364
365
366   f_plus_L1=F_plus_k(cos_theta_L1,cos_2_phi_L1)/Fourier_L1_noise_power_spectral_density_sqrt
367   f_cross_L1=F_cross_k(cos_theta_L1,sin_2_phi_L1)/
        Fourier_L1_noise_power_spectral_density_sqrt
368   f_plus_H1=F_plus_k(cos_theta_H1,cos_2_phi_H1)/Fourier_H1_noise_power_spectral_density_sqrt
369   f_cross_H1=F_cross_k(cos_theta_H1,sin_2_phi_H1)/
        Fourier_H1_noise_power_spectral_density_sqrt
370   f_plus_V1=F_plus_k(cos_theta_V1,cos_2_phi_V1)/Fourier_V1_noise_power_spectral_density_sqrt
371   f_cross_V1=F_cross_k(cos_theta_V1,sin_2_phi_V1)/
        Fourier_V1_noise_power_spectral_density_sqrt
372
373   cos_psi_DPF,sin_psi_DPF,f_plus_L1_DPF,f_plus_H1_DPF,f_plus_V1_DPF,f_cross_L1_DPF,
        f_cross_H1_DPF,f_cross_V1_DPF=DPF_antenna_patterns(f_plus_L1,f_plus_H1,f_plus_V1,
        f_cross_L1,f_cross_H1,f_cross_V1)
374
375
376
377   h_plus_DPF_Fourier=np.zeros(len(Fourier_L1_response_whitened),dtype=complex)
378   h_cross_DPF_Fourier=np.zeros(len(Fourier_L1_response_whitened),dtype=complex)
379   h_plus_Fourier=np.zeros(len(Fourier_L1_response_whitened),dtype=complex)
380   h_cross_Fourier=np.zeros(len(Fourier_L1_response_whitened),dtype=complex)
381
382
383   for i in range(len(Fourier_L1_response_whitened)):
384     h_plus_DPF_Fourier[i]=((Fourier_L1_response_whitened[i]*f_plus_L1_DPF[i])+(
        Fourier_H1_response_whitened[i]*f_plus_H1_DPF[i])+(Fourier_V1_response_whitened[i]*
        f_plus_V1_DPF[i]))/(f_plus_L1_DPF[i]*f_plus_L1_DPF[i]+f_plus_H1_DPF[i]*f_plus_H1_DPF[i]+
        f_plus_V1_DPF[i]*f_plus_V1_DPF[i])
385     h_cross_DPF_Fourier[i]=((Fourier_L1_response_whitened[i]*f_cross_L1_DPF[i])+(
        Fourier_H1_response_whitened[i]*f_cross_H1_DPF[i])+(Fourier_V1_response_whitened[i]*
        f_cross_V1_DPF[i]))/(f_cross_L1_DPF[i]*f_cross_L1_DPF[i]+f_cross_H1_DPF[i]*
        f_cross_H1_DPF[i]+f_cross_V1_DPF[i]*f_cross_V1_DPF[i])
386     h_plus_Fourier[i]=cos_psi_DPF[i]*h_plus_DPF_Fourier[i]-sin_psi_DPF[i]*
        h_cross_DPF_Fourier[i]
387     h_cross_Fourier[i]=sin_psi_DPF[i]*h_plus_DPF_Fourier[i]+cos_psi_DPF[i]*
        h_cross_DPF_Fourier[i]
388
389
390
391
392   h_plus=(fftpack.ifft(h_plus_Fourier)).real
393   h_cross=(fftpack.ifft(h_cross_Fourier)).real
394
395   return h_plus,h_cross
396
397
398
399 # this function is used to choose the correct sky statistic maximum among the two equivalent
        ones
400 def time_delay_analysis(time_delay_L1,time_delay_H1,time_delay_V1,theta,phi):
401   n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta
        )]) #incoming direction
402   reconstructed_time_delay_L1=time_delay(r_L1,n_Earth_frame)
403   reconstructed_time_delay_H1=time_delay(r_H1,n_Earth_frame)
404   reconstructed_time_delay_V1=time_delay(r_V1,n_Earth_frame)
405
406   time_delay_parameter=abs(reconstructed_time_delay_L1-time_delay_L1)+abs(
        reconstructed_time_delay_H1-time_delay_H1)+abs(reconstructed_time_delay_V1-time_delay_V1
        )
```

```
407
408   return time_delay_parameter
409
410
411
412 # this function calculates the overlap between the injected polarization and the
      reconstructed one
413 def overlap(w_1,w_2):
414   return np.dot(w_1,w_2)/np.sqrt(np.dot(w_1,w_1)*np.dot(w_2,w_2))
415
```

noise_module.py

```
import numpy as np
import scipy
import scipy.constants as constants
from scipy import fftpack


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.98847e30                           # kg
pi=constants.pi


# because the noise formulae diverge in zero I assume a cut frequency
f_cut=7. # Hz


# these values and equations are reported in Chapter 2 of the thesis, which are taken from
# A Buikema et al. 'Sensitivity and performance of the Advanced LIGO detectors in thethird observing run'. In:Physical Review D102.6 (2020), p. 062003.
# and
# Denis V Martynov et al. 'Sensitivity of the Advanced LIGO detectors at the beginning of gravitational wave astronomy'. In:Physical Review D93.11 (2016), p. 112004.


def K_minus_transfer_function_real(f,f_minus):
  function=(f_minus**2)/((f_minus**2)+(f**2))
  result=function
  return result

def K_minus_transfer_function_imaginary(f,f_minus):
  function=-(f_minus*f)/((f_minus**2)+(f**2))
  result=function
  return result



def fluctuations_of_local_gravity_noise(f): # m/sqrt(Hz)
  if f!=0.:
    return 2.28e-19*((10/f)**2)
  else:
    return 0.


def thermal_noise(f): # m/sqrt(Hz)
# this formula is taken from Yu Levin. 'Internal thermal noise in the LIGO test masses: A direct approach'. In:Physical Review D57.2 (1998), p. 659.

  if f!=0.:
    return 1.13e-20*((100./f)**2)
  else:
    return 0.



def quantum_radiation_pressure_noise_real(f,f_minus,P_arm): # m/sqrt(Hz)
  if f!=0.:
    function=(1.38e-17/(f**2))*np.sqrt(P_arm/1.e5)*K_minus_transfer_function_real(f,f_minus)
  else:
    function=0.
  result=function
  return result


def quantum_radiation_pressure_noise_imaginary(f,f_minus,P_arm): # m/sqrt(Hz)
```

```
  if f!=0.:
    function=(1.38e-17/(f**2))*np.sqrt(P_arm/1.e5)*K_minus_transfer_function_imaginary(f,
    f_minus)
  else:
    function=0.
  result=function
  return result


def shot_noise_real(f,f_minus,P_arm,eta): # m/sqrt(Hz)
  function=2.e-20*np.sqrt(1.e5/(P_arm*eta))
  result=function
  return result


def shot_noise_imaginary(f,f_minus,P_arm,eta): # m/sqrt(Hz)
  function=2.e-20*np.sqrt(1.e5/(P_arm*eta))*(f/f_minus)
  result=function
  return result



def noise_strain_sensitivity(f,f_minus,P_arm,eta,L): # 1/sqrt(Hz)
  if abs(f)>f_cut:
    newtonian_2=fluctuations_of_local_gravity_noise(f)**2
    thermal_2=thermal_noise(f)**2
    qrpn_2=(quantum_radiation_pressure_noise_real(f,f_minus,P_arm)**2)+(
    quantum_radiation_pressure_noise_imaginary(f,f_minus,P_arm)**2)
    sn_2=(shot_noise_real(f,f_minus,P_arm,eta)**2)+(shot_noise_imaginary(f,f_minus,P_arm,eta
    )**2)
    result=np.sqrt(newtonian_2+thermal_2+qrpn_2+sn_2)/L
  else:
    newtonian_2=fluctuations_of_local_gravity_noise(f_cut)**2
    thermal_2=thermal_noise(f_cut)**2
    qrpn_2=(quantum_radiation_pressure_noise_real(f_cut,f_minus,P_arm)**2)+(
    quantum_radiation_pressure_noise_imaginary(f_cut,f_minus,P_arm)**2)
    sn_2=(shot_noise_real(f_cut,f_minus,P_arm,eta)**2)+(shot_noise_imaginary(f_cut,f_minus,
    P_arm,eta)**2)
    result=np.sqrt(newtonian_2+thermal_2+qrpn_2+sn_2)/L
  return result


# this is an implementation of the Timmer & Koenig algorithm
# I consider n(-f)=n(f)*

def Timmer_Koenig(frequency,sampling,f_minus,P_arm,eta,L,mu,sigma,len_noise_array,
    scaling_factor):
  noise_array_real=np.zeros(len_noise_array)
  noise_array_imaginary=np.zeros(len_noise_array)
  noise_array_frequency=np.zeros(len_noise_array,dtype=complex)

  for i in range(len_noise_array):

    if i==0:
      noise_array_real[i]=np.sqrt(0.5)*np.random.normal(mu,sigma,1)*noise_strain_sensitivity
    (frequency[i],f_minus,P_arm,eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=0.
    elif 0<i and i<int(len_noise_array/2):
      noise_array_real[i]=np.sqrt(0.5)*np.random.normal(mu,sigma,1)*noise_strain_sensitivity
    (frequency[i],f_minus,P_arm,eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=np.sqrt(0.5)*np.random.normal(mu,sigma,1)*
    noise_strain_sensitivity(frequency[i],f_minus,P_arm,eta,L)*np.sqrt(sampling)
    elif i==int(len_noise_array/2):
      noise_array_real[i]=np.sqrt(0.5)*np.random.normal(mu,sigma,1)*noise_strain_sensitivity
    (frequency[i],f_minus,P_arm,eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=0.
    elif int(len_noise_array/2)<i:

```

```
      noise_array_real[i]=noise_array_real[int(len_noise_array)-i]
      noise_array_imaginary[i]=-noise_array_imaginary[int(len_noise_array)-i] # complex
    conjugate

  noise_array_frequency=(noise_array_real+1j*noise_array_imaginary)*scaling_factor
  noise_array_time=(fftpack.ifft(noise_array_frequency)).real

  return noise_array_frequency,noise_array_time


# here I calculate the amplitude spectral density
def power_spectral_density_sqrt(frequency,sampling,f_minus,P_arm,eta,L,len_power_array): #
    adimensional
  noise_array_real=np.zeros(len_power_array)
  noise_array_imaginary=np.zeros(len_power_array)
  power_spectral_density_sqrt=np.zeros(len_power_array)

  for i in range(len_power_array):


    if i==0:
      noise_array_real[i]=np.sqrt(0.5)*noise_strain_sensitivity(frequency[i],f_minus,P_arm,
    eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=0.
    elif 0<i and i<int(len_power_array/2):
      noise_array_real[i]=np.sqrt(0.5)*noise_strain_sensitivity(frequency[i],f_minus,P_arm,
    eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=np.sqrt(0.5)*noise_strain_sensitivity(frequency[i],f_minus,
    P_arm,eta,L)*np.sqrt(sampling)
    elif i==int(len_power_array/2):
      noise_array_real[i]=np.sqrt(0.5)*noise_strain_sensitivity(frequency[i],f_minus,P_arm,
    eta,L)*np.sqrt(sampling)
      noise_array_imaginary[i]=0.
    elif int(len_power_array/2)<i:

      noise_array_real[i]=noise_array_real[int(len_power_array)-i]
      noise_array_imaginary[i]=-noise_array_imaginary[int(len_power_array)-i]

  power_spectral_density_sqrt=np.sqrt(noise_array_real**2+noise_array_imaginary**2)/
    len_power_array

  return power_spectral_density_sqrt
```

polarization_cWB.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.98847e30                           # kg
pi=constants.pi


#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
    phi_V1), 6368239.*np.cos(theta_V1)])


```

```
63
64
65
66
67
68 L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
69 H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
70 V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)
71
72
73
74
75 print('Do you want to use default values for the waveforms? If so digit 1, if not other')
76 boolean_data=int(input())
77
78
79 f_s=16384
80
81 if(boolean_data==1):
82   print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
        another number for the BBH with masses 30.7 and 25.3')
83   boolean_0=int(input())
84
85   if(boolean_0==1):
86     mass_1=11.0
87     mass_2=7.6
88     d=320. # Mpc
89     h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
       mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
90     t_data=h_plus_data.sample_times
91   else:
92     mass_1=30.7
93     mass_2=25.3
94     d=600. # Mpc
95     h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
       mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
96     t_data=h_plus_data.sample_times
97
98
99 else:
100
101   print('Write the masses of the bodies (in solar masses)')
102   print('m_1=')
103   mass_1=float(input())
104   print('m_2=')
105   mass_2=float(input())
106   print('Write the luminosity distance of the bodies (in Mpc)')
107   d=float(input())
108   h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
        =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
109   t_data=h_plus_data.sample_times
110
111 mass_1_text=str(round(mass_1,1))
112 mass_2_text=str(round(mass_2,1))
113 d_text=str(round(d,0))
114 polarizations_figure, (h_plus, h_cross) = plt.subplots(1, 2)
115 polarizations_figure.suptitle('$h_{+}$ and $h_{x}$ polarizations ($M_{1}=$ %s $M_\odot$, $M_
      {2}=$ %s $M_\odot$, $d_{L}=$ %s Mpc)'%(mass_1_text,mass_2_text,d_text))
116 h_plus.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
117 h_plus.set_title('$h_{+}$')
118 h_plus.grid()
119 h_plus.plot(t_data,h_plus_data)
120 h_cross.set(xlabel='Time ($s$)', ylabel='')
121 h_cross.set_title('$h_{+}$')
122 h_cross.grid()
123 h_cross.plot(t_data,h_cross_data)
124
125
```

```
126
127 print('Write the latitude of the source (in degrees)')
128 latitude_degrees=float(input())
129 latitude=latitude_degrees*pi/180.
130 theta=pi/2.-latitude
131
132 print('Write the longitude of the source (in degrees)')
133 longitude_degrees=float(input())
134 longitude=longitude_degrees*pi/180.
135 phi=longitude
136
137
138
139 n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta)
        ]) #incoming direction
140
141 # here the source direction is considered in the detector frame
142 n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
143 n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
144 n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
145
146
147 cos_theta_L1=n_L1[2]
148 cos_theta_H1=n_H1[2]
149 cos_theta_V1=n_V1[2]
150
151 cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
152 cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
153 cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
154
155 sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
156 sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
157 sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
158
159
160
161 time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
162 time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
163 time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)
164
165
166 print('L1 time delay=',time_delay_L1,'s')
167 print('H1 time delay=',time_delay_H1,'s')
168 print('V1 time delay=',time_delay_V1,'s')
169
170
171
172
173 # this is the maximum delay time at which the intefrerometer can detect the signal
174 max_delay=0.022
175
176
177 # antenna patterns calculation
178 F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
179 F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
180 F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
181 F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
182 F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
183 F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)
184
185
186
187
188 t_min=min(t_data)
189 t_max=max(t_data)
190
191 t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)
192
```

```
193
194 # here I have to obtain an array with even number of elements: this will be important during
      the noise generation in frequency domain
195 if len(t)%2!=0:
196   t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)
197
198
199
200 # parameters for noise generation
201
202 L=4000.    # m
203 f_minus=390. # Hz
204 P_arm=240000.    # Watt
205 eta=0.75
206 f_min=10.    # Hz
207 f_max=8192. # Hz
208
209
210
211 frequency=fftpack.fftfreq(len(t)) * f_s
212
213
214 print('Now you can decide to multiply the noise for a scaling factor (for default values
      insert multiples of 1,1,3.3)')
215 print('Write the scaling factor for L1 noise')
216 scaling_factor_L1=float(input())
217 print('Write the scaling factor for H1 noise')
218 scaling_factor_H1=float(input())
219 print('Write the scaling factor for V1 noise')
220 scaling_factor_V1=float(input())
221
222
223
224
225 # noise generation
226 mu=0.
227 sigma=1.
228
229 Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
      ,scaling_factor_L1)
230 Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
      ,scaling_factor_H1)
231 Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
      ,scaling_factor_V1)
232
233
234
235
236 # this generate a graph of the noise strain sensitivity
237 strain_noise_L1=np.zeros(len(frequency))
238 for i in range(len(strain_noise_L1)):
239   strain_noise_L1[i]=scaling_factor_L1*nm.noise_strain_sensitivity(frequency[i],f_minus,
      P_arm,eta,L)
240
241 strain_noise_figure=plt.figure()
242 plt.xlabel('Frequency (Hz)')
243 plt.xlim(9., 8191.)
244 plt.ylabel('Strain noise ($1/\sqrt{Hz}$)')
245 plt.title('Noise strain sensitivity')
246 plt.grid()
247 plt.loglog(frequency,strain_noise_L1)
248
249
250
251 L1_response=L1_noise
252 H1_response=H1_noise
253 V1_response=V1_noise
254
```

```
255
256 i_trigger_L1=1e10
257 i_trigger_H1=1e10
258 i_trigger_V1=1e10
259
260
261 # now I generate the detector responses
262 for i in range(len(t)):
263
264
265   if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:
266
267     if i<i_trigger_L1:
268       i_trigger_L1=i
269
270     L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
      h_cross_data[i-i_trigger_L1]
271
272   if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:
273
274     if i<i_trigger_H1:
275       i_trigger_H1=i
276
277     H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*
      h_cross_data[i-i_trigger_H1]
278
279   if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:
280
281     if i<i_trigger_V1:
282       i_trigger_V1=i
283
284     V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*
      h_cross_data[i-i_trigger_V1]
285
286
287
288
289
290 # now the signals have to be aligned in time
291
292 t_abs=np.arange(t_min, t_max, 1./f_s)
293 t_abs=np.append(t_abs,t_max) # this is for including the last array element
294 # also here I have to obtain an array with even number of elements
295 if len(t_abs)%2!=0:
296   t_abs=np.append(t_abs,t_max+1./f_s)
297   h_plus_data=np.append(h_plus_data,0.)
298   h_cross_data=np.append(h_cross_data,0.)
299
300
301 L1_response_aligned=np.zeros(len(t_abs))
302 H1_response_aligned=np.zeros(len(t_abs))
303 V1_response_aligned=np.zeros(len(t_abs))
304
305 original_signal_L1=np.zeros(len(t_abs))
306
307 for i in range(len(t_abs)):
308   L1_response_aligned[i]=L1_response[i+i_trigger_L1]
309   H1_response_aligned[i]=H1_response[i+i_trigger_H1]
310   V1_response_aligned[i]=V1_response[i+i_trigger_V1]
311   original_signal_L1[i]=F_plus_L1*h_plus_data[i]+F_cross_L1*h_cross_data[i]
312
313
314
315
316
317 Fourier_L1_response_aligned=fftpack.fft(L1_response_aligned)
318 Fourier_H1_response_aligned=fftpack.fft(H1_response_aligned)
319 Fourier_V1_response_aligned=fftpack.fft(V1_response_aligned)
```

```
320
321
322
323 Signal=(np.linalg.norm(Fourier_L1_response_aligned)**2+np.linalg.norm(
        Fourier_H1_response_aligned)**2+np.linalg.norm(Fourier_V1_response_aligned)**2)
324 Noise=(np.linalg.norm(Fourier_noise_L1)**2+np.linalg.norm(Fourier_noise_H1)**2+np.linalg.
        norm(Fourier_noise_V1)**2)
325 SNR=Signal/Noise
326
327 print('SNR')
328 print(round(SNR,2))
329
330
331
332 detector_responses_figure, (L1, H1, V1) = plt.subplots(1, 3)
333 detector_responses_figure.suptitle('Detector responses')
334 L1.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
335 L1.set_title('L1')
336 L1.grid()
337 L1.plot(t_abs,L1_response_aligned)
338 H1.set(xlabel='Time ($s$)', ylabel='')
339 H1.set_title('H1')
340 H1.grid()
341 H1.plot(t_abs,H1_response_aligned)
342 V1.set(xlabel='Time ($s$)', ylabel='')
343 V1.set_title('V1')
344 V1.grid()
345 V1.plot(t_abs,V1_response_aligned)
346
347
348
349
350
351
352 frequency=fftpack.fftfreq(len(t_abs))*f_s
353
354
355
356 Fourier_L1_response=fftpack.fft(L1_response_aligned)
357 Fourier_H1_response=fftpack.fft(H1_response_aligned)
358 Fourier_V1_response=fftpack.fft(V1_response_aligned)
359
360 Fourier_L1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
        f_minus,P_arm,eta,L,len(frequency))*scaling_factor_L1
361 Fourier_H1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
        f_minus,P_arm,eta,L,len(frequency))*scaling_factor_H1
362 Fourier_V1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
        f_minus,P_arm,eta,L,len(frequency))*scaling_factor_V1
363
364
365 Fourier_L1_response_whitened=Fourier_L1_response/
        Fourier_L1_noise_power_spectral_density_sqrt
366 Fourier_H1_response_whitened=Fourier_H1_response/
        Fourier_H1_noise_power_spectral_density_sqrt
367 Fourier_V1_response_whitened=Fourier_V1_response/
        Fourier_V1_noise_power_spectral_density_sqrt
368
369
370
371 # reconstruction done assuming a known source location
372
373 h_plus,h_cross=pm.polarization_reconstruction(theta,phi,Fourier_L1_response_whitened,
        Fourier_H1_response_whitened,Fourier_V1_response_whitened,
374   Fourier_L1_noise_power_spectral_density_sqrt,Fourier_H1_noise_power_spectral_density_sqrt,
        Fourier_V1_noise_power_spectral_density_sqrt)
375
376
377 h_plus_figure=plt.figure()
```

```
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{+}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_plus,label='$h_{+}$ reconstructed',color='r')
plt.plot(t_abs,h_plus_data,label='$h_{+}$ injected',color='b')
plt.legend()

h_cross_figure=plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{x}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_cross,label='$h_{x}$ reconstructed',color='r')
plt.plot(t_abs,h_cross_data,label='$h_{x}$ injected',color='b')
plt.legend()


reconstructed_polarizations_figure, (h_plus_reconstructed,h_cross_reconstructed) = plt.
    subplots(1, 2)
reconstructed_polarizations_figure.suptitle('Reconstructed polarizations with cWB')
h_plus_reconstructed.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
h_plus_reconstructed.set_title('$h_{+}$ reconstructed')
h_plus_reconstructed.grid()
h_plus_reconstructed.plot(t_abs,h_plus)
h_cross_reconstructed.set(xlabel='Time ($s$)', ylabel='')
h_cross_reconstructed.set_title('$h_{x}$ reconstructed')
h_cross_reconstructed.grid()
h_cross_reconstructed.plot(t_abs,h_cross)


overlap_h_plus=pm.overlap(h_plus,h_plus_data)
overlap_h_cross=pm.overlap(h_cross,h_cross_data)

print('Overlap h plus')
print(overlap_h_plus)

print('Overlap h cross')
print(overlap_h_cross)

plt.show()
```

polarization_wavelets_denoising_and_source_location_identification.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
import pywt
from mpl_toolkits.basemap import Basemap
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.98847e30                           # kg
pi=constants.pi


#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
```

```
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
    phi_V1), 6368239.*np.cos(theta_V1)])
```

```
L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)
```

```
print('Do you want to use default values for the waveforms? If so digit 1, if not other')
boolean_data=int(input())


f_s=16384

if(boolean_data==1):
  print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
     another number for the BBH with masses 30.7 and 25.3')
  boolean_0=int(input())

  if(boolean_0==1):
    mass_1=11.0
    mass_2=7.6
    d=320. # Mpc
    h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
    mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
    t_data=h_plus_data.sample_times
  else:
    mass_1=30.7
    mass_2=25.3
    d=600. # Mpc
    h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
    mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
    t_data=h_plus_data.sample_times


else:

  print('Write the masses of the bodies (in solar masses)')
  print('m_1=')
  mass_1=float(input())
  print('m_2=')
  mass_2=float(input())
  print('Write the luminosity distance of the bodies (in Mpc)')
  d=float(input())
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times

mass_1_text=str(round(mass_1,1))
mass_2_text=str(round(mass_2,1))
d_text=str(round(d,0))
polarizations_figure, (h_plus, h_cross) = plt.subplots(1, 2)
polarizations_figure.suptitle('$h_{+}$ and $h_{x}$ polarizations ($M_{1}=$ %s $M_\odot$, $M_
    {2}=$ %s $M_\odot$, $d_{L}=$ %s Mpc)'%(mass_1_text,mass_2_text,d_text))
h_plus.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
h_plus.set_title('$h_{+}$')
h_plus.grid()
h_plus.plot(t_data,h_plus_data)
h_cross.set(xlabel='Time ($s$)', ylabel='')
h_cross.set_title('$h_{+}$')
```

```
h_cross.grid()
h_cross.plot(t_data,h_cross_data)



print('Write the latitude of the source (in degrees)')
latitude_degrees=float(input())
latitude=latitude_degrees*pi/180.
theta=pi/2.-latitude

print('Write the longitude of the source (in degrees)')
longitude_degrees=float(input())
longitude=longitude_degrees*pi/180.
phi=longitude



n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta)
    ]) #incoming direction

# here the source direction is considered in the detector frame
n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)


cos_theta_L1=n_L1[2]
cos_theta_H1=n_H1[2]
cos_theta_V1=n_V1[2]

cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.

sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))



time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)


print('L1 time delay=',time_delay_L1,'s')
print('H1 time delay=',time_delay_H1,'s')
print('V1 time delay=',time_delay_V1,'s')




# this is the maximum delay time at which the intefrerometer can detect the signal
max_delay=0.022


# antenna patterns calculation
F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)



t_min=min(t_data)
```

```
t_max=max(t_data)

t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)


# here I have to obtain an array with even number of elements: this will be important during the noise generation in frequency domain
if len(t)%2!=0:
  t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)



# parameters for noise generation

L=4000.    # m
f_minus=390. # Hz
P_arm=240000.    # Watt
eta=0.75
f_min=10.    # Hz
f_max=8192. # Hz



frequency=fftpack.fftfreq(len(t)) * f_s


print('Now you can decide to multiply the noise for a scaling factor (for default values insert multiples of 1,1,3.3)')
print('Write the scaling factor for L1 noise')
scaling_factor_L1=float(input())
print('Write the scaling factor for H1 noise')
scaling_factor_H1=float(input())
print('Write the scaling factor for V1 noise')
scaling_factor_V1=float(input())



# this is for writing the titles of the sky statistic figures
new_line='\n'
comma=','
space=' '
latitude_str='Injected latitude (degrees)='+str(latitude_degrees)
longitude_str='Injected longitude (degrees)='+str(longitude_degrees)
masses='Bodies masses (in Solar masses)='+str(mass_1)+comma+str(mass_2)
noise_scaling_factors='Noise scaling factors for L1,H1,V1='+str(scaling_factor_L1)+comma+space+str(scaling_factor_H1)+comma+space+str(scaling_factor_V1)
sky_statistic_title='Sky statistic respresentation'




# noise generation
mu=0.
sigma=1.

Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t),scaling_factor_L1)
Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t),scaling_factor_H1)
Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t),scaling_factor_V1)



# this generate a graph of the noise strain sensitivity
strain_noise_L1=np.zeros(len(frequency))
for i in range(len(strain_noise_L1)):
```

```
  strain_noise_L1[i]=scaling_factor_L1*nm.noise_strain_sensitivity(frequency[i],f_minus,
    P_arm,eta,L)

strain_noise_figure=plt.figure()
plt.xlabel('Frequency (Hz)')
plt.xlim(9., 8191.)
plt.ylabel('Strain noise ($1/\sqrt{Hz}$)')
plt.title('Noise strain sensitivity')
plt.grid()
plt.loglog(frequency,strain_noise_L1)



L1_response=L1_noise
H1_response=H1_noise
V1_response=V1_noise


i_trigger_L1=1e10
i_trigger_H1=1e10
i_trigger_V1=1e10


# now I generate the detector responses
for i in range(len(t)):


  if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:

    if i<i_trigger_L1:
      i_trigger_L1=i

    L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
    h_cross_data[i-i_trigger_L1]

  if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:

    if i<i_trigger_H1:
      i_trigger_H1=i

    H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*
    h_cross_data[i-i_trigger_H1]

  if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:

    if i<i_trigger_V1:
      i_trigger_V1=i

    V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*
    h_cross_data[i-i_trigger_V1]






# now the signals have to be aligned in time

t_abs=np.arange(t_min, t_max, 1./f_s)
t_abs=np.append(t_abs,t_max) # this is for including the last array element
# also here I have to obtain an array with even number of elements
if len(t_abs)%2!=0:
  t_abs=np.append(t_abs,t_max+1./f_s)
  h_plus_data=np.append(h_plus_data,0.)
  h_cross_data=np.append(h_cross_data,0.)


L1_response_aligned=np.zeros(len(t_abs))
H1_response_aligned=np.zeros(len(t_abs))
```

```
V1_response_aligned=np.zeros(len(t_abs))

original_signal_L1=np.zeros(len(t_abs))

for i in range(len(t_abs)):
  L1_response_aligned[i]=L1_response[i+i_trigger_L1]
  H1_response_aligned[i]=H1_response[i+i_trigger_H1]
  V1_response_aligned[i]=V1_response[i+i_trigger_V1]
  original_signal_L1[i]=F_plus_L1*h_plus_data[i]+F_cross_L1*h_cross_data[i]




Fourier_L1_response_aligned=fftpack.fft(L1_response_aligned)
Fourier_H1_response_aligned=fftpack.fft(H1_response_aligned)
Fourier_V1_response_aligned=fftpack.fft(V1_response_aligned)



Signal=(np.linalg.norm(Fourier_L1_response_aligned)**2+np.linalg.norm(
    Fourier_H1_response_aligned)**2+np.linalg.norm(Fourier_V1_response_aligned)**2)
Noise=(np.linalg.norm(Fourier_noise_L1)**2+np.linalg.norm(Fourier_noise_H1)**2+np.linalg.
    norm(Fourier_noise_V1)**2)
SNR=Signal/Noise

print('SNR')
print(round(SNR,2))


# wavelet denoising
# for each signal I take the approximation coefficients (cA) of the wavelet transofrm,
    discarding the details one (cD)

wavelet='dmey' # Meyer wavelets

(L1_response_aligned_cA, L1_response_aligned_cD)=pywt.dwt(L1_response_aligned, wavelet, mode
    ='zero', axis=-1)
(H1_response_aligned_cA, H1_response_aligned_cD)=pywt.dwt(H1_response_aligned, wavelet, mode
    ='zero', axis=-1)
(V1_response_aligned_cA, V1_response_aligned_cD)=pywt.dwt(V1_response_aligned, wavelet, mode
    ='zero', axis=-1)
(original_signal_L1_cA, original_signal_L1_cD)=pywt.dwt(original_signal_L1, wavelet, mode='
    zero', axis=-1)


zero=np.zeros(len(L1_response_aligned_cD))

L1_response_aligned_cA_reconstruction=pywt.idwt(L1_response_aligned_cA,zero,wavelet, mode='
    zero', axis=-1)
H1_response_aligned_cA_reconstruction=pywt.idwt(H1_response_aligned_cA,zero,wavelet, mode='
    zero', axis=-1)
V1_response_aligned_cA_reconstruction=pywt.idwt(V1_response_aligned_cA,zero,wavelet, mode='
    zero', axis=-1)
original_signal_L1_cA_reconstruction=pywt.idwt(original_signal_L1_cA,zero,wavelet, mode='
    zero', axis=-1)




detector_responses_figure, (L1, H1, V1) = plt.subplots(1, 3)
detector_responses_figure.suptitle('Detector responses')
L1.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
L1.set_title('L1')
L1.grid()
L1.plot(t_abs,L1_response_aligned)
H1.set(xlabel='Time ($s$)', ylabel='')
H1.set_title('H1')
H1.grid()
```

```
H1.plot(t_abs,H1_response_aligned)
V1.set(xlabel='Time ($s$)', ylabel='')
V1.set_title('V1')
V1.grid()
V1.plot(t_abs,V1_response_aligned)




wavelets_denoising_figure, (original_signal, noised_response, denoised_response) = plt.
    subplots(1, 3)
wavelets_denoising_figure.suptitle('Wavelets denoising')
original_signal.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
original_signal.set_title('Original signal')
original_signal.grid()
original_signal.plot(t_abs,original_signal_L1)
noised_response.set(xlabel='Time ($s$)', ylabel='')
noised_response.set_title('Noised \n detector response')
noised_response.grid()
noised_response.plot(t_abs,L1_response_aligned)
denoised_response.set(xlabel='Time ($s$)', ylabel='')
denoised_response.set_title('Denoised \n detector response')
denoised_response.grid()
denoised_response.plot(t_abs,L1_response_aligned_cA_reconstruction)



frequency=fftpack.fftfreq(len(t_abs))*f_s



Fourier_L1_response=fftpack.fft(L1_response_aligned_cA_reconstruction)
Fourier_H1_response=fftpack.fft(H1_response_aligned_cA_reconstruction)
Fourier_V1_response=fftpack.fft(V1_response_aligned_cA_reconstruction)

Fourier_L1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
    f_minus,P_arm,eta,L,len(frequency))*scaling_factor_L1
Fourier_H1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
    f_minus,P_arm,eta,L,len(frequency))*scaling_factor_H1
Fourier_V1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,
    f_minus,P_arm,eta,L,len(frequency))*scaling_factor_V1


Fourier_L1_response_whitened=Fourier_L1_response/
    Fourier_L1_noise_power_spectral_density_sqrt
Fourier_H1_response_whitened=Fourier_H1_response/
    Fourier_H1_noise_power_spectral_density_sqrt
Fourier_V1_response_whitened=Fourier_V1_response/
    Fourier_V1_noise_power_spectral_density_sqrt



# first reconstruction done assuming a known source location

h_plus,h_cross=pm.polarization_reconstruction(theta,phi,Fourier_L1_response_whitened,
    Fourier_H1_response_whitened,Fourier_V1_response_whitened,
  Fourier_L1_noise_power_spectral_density_sqrt,Fourier_H1_noise_power_spectral_density_sqrt,
    Fourier_V1_noise_power_spectral_density_sqrt)



h_plus_figure=plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{+}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_plus,label='$h_{+}$ reconstructed',color='r')
```

```
plt.plot(t_abs,h_plus_data,label='$h_{+}$ injected',color='b')
plt.legend()

h_cross_figure=plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{x}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_cross,label='$h_{x}$ reconstructed',color='r')
plt.plot(t_abs,h_cross_data,label='$h_{x}$ injected',color='b')
plt.legend()


reconstructed_polarizations_figure, (h_plus_reconstructed,h_cross_reconstructed) = plt.
    subplots(1, 2)
reconstructed_polarizations_figure.suptitle('Reconstructed polarizations with cWB wavelets
    denoising')
h_plus_reconstructed.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
h_plus_reconstructed.set_title('$h_{+}$ reconstructed')
h_plus_reconstructed.grid()
h_plus_reconstructed.plot(t_abs,h_plus)
h_cross_reconstructed.set(xlabel='Time ($s$)', ylabel='')
h_cross_reconstructed.set_title('$h_{x}$ reconstructed')
h_cross_reconstructed.grid()
h_cross_reconstructed.plot(t_abs,h_cross)



overlap_h_plus=pm.overlap(h_plus,h_plus_data)
overlap_h_cross=pm.overlap(h_cross,h_cross_data)

print('Overlap h plus')
print(overlap_h_plus)

print('Overlap h cross')
print(overlap_h_cross)


# source location identification part
# here I downsample the signal in time domain
# this is for making the calculation of the sky statistic easier and faster
downsampling_factor=8. # from 16384 Hz to 2048 Hz

downsampled_t_abs,downsampled_L1_response_aligned_cA_reconstruction=pm.
    downsampling_in_time_domain(L1_response_aligned_cA_reconstruction,t_abs,
    downsampling_factor)
downsampled_t_abs,downsampled_H1_response_aligned_cA_reconstruction=pm.
    downsampling_in_time_domain(H1_response_aligned_cA_reconstruction,t_abs,
    downsampling_factor)
downsampled_t_abs,downsampled_V1_response_aligned_cA_reconstruction=pm.
    downsampling_in_time_domain(V1_response_aligned_cA_reconstruction,t_abs,
    downsampling_factor)

downsampled_f_s=f_s/downsampling_factor
downsampled_frequency=fftpack.fftfreq(len(downsampled_t_abs))*downsampled_f_s



downsampled_Fourier_L1_response=fftpack.fft(
    downsampled_L1_response_aligned_cA_reconstruction)
downsampled_Fourier_H1_response=fftpack.fft(
    downsampled_H1_response_aligned_cA_reconstruction)
downsampled_Fourier_V1_response=fftpack.fft(
    downsampled_V1_response_aligned_cA_reconstruction)


downsampled_Fourier_L1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(
    downsampled_frequency,downsampled_f_s,f_minus,P_arm,eta,L,len(downsampled_frequency))*
```

```
    scaling_factor_L1/np.sqrt(downsampling_factor)
downsampled_Fourier_H1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(
    downsampled_frequency,downsampled_f_s,f_minus,P_arm,eta,L,len(downsampled_frequency))*
    scaling_factor_H1/np.sqrt(downsampling_factor)
downsampled_Fourier_V1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(
    downsampled_frequency,downsampled_f_s,f_minus,P_arm,eta,L,len(downsampled_frequency))*
    scaling_factor_V1/np.sqrt(downsampling_factor)

# here I use only positive frequencies (see Chapter 4)
positive_frequencies,downsampled_Fourier_L1_response=pm.positive_frequencies_selection(
    downsampled_Fourier_L1_response,downsampled_frequency)
positive_frequencies,downsampled_Fourier_H1_response=pm.positive_frequencies_selection(
    downsampled_Fourier_H1_response,downsampled_frequency)
positive_frequencies,downsampled_Fourier_V1_response=pm.positive_frequencies_selection(
    downsampled_Fourier_V1_response,downsampled_frequency)
positive_frequencies,downsampled_Fourier_L1_noise_power_spectral_density_sqrt=pm.
    positive_frequencies_selection(downsampled_Fourier_L1_noise_power_spectral_density_sqrt,
    downsampled_frequency)
positive_frequencies,downsampled_Fourier_H1_noise_power_spectral_density_sqrt=pm.
    positive_frequencies_selection(downsampled_Fourier_H1_noise_power_spectral_density_sqrt,
    downsampled_frequency)
positive_frequencies,downsampled_Fourier_V1_noise_power_spectral_density_sqrt=pm.
    positive_frequencies_selection(downsampled_Fourier_V1_noise_power_spectral_density_sqrt,
    downsampled_frequency)


# data whitening
downsampled_Fourier_L1_response_whitened=downsampled_Fourier_L1_response/
    downsampled_Fourier_L1_noise_power_spectral_density_sqrt
downsampled_Fourier_H1_response_whitened=downsampled_Fourier_H1_response/
    downsampled_Fourier_H1_noise_power_spectral_density_sqrt
downsampled_Fourier_V1_response_whitened=downsampled_Fourier_V1_response/
    downsampled_Fourier_V1_noise_power_spectral_density_sqrt









# here I calculate the sky statistic
theta_pixels=36
phi_pixels=72



sky_statistic,theta_max_1,phi_max_1,sky_statistic_max_1,theta_max_2,phi_max_2,
    sky_statistic_max_2=pm.sky_statistic_sky_grid(theta_pixels,phi_pixels,
  downsampled_Fourier_L1_response,downsampled_Fourier_H1_response,
    downsampled_Fourier_V1_response,
  downsampled_Fourier_L1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_H1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_V1_noise_power_spectral_density_sqrt)




# here I use a gradient ascent for the sky statistic
n_max=80
threshold=(pi/180.)*1.e-6 # threshold


print('Gradient ascent calculation (it might be a while...)')

theta_max_1,phi_max_1,iteration,theta_1_path,phi_1_path=pm.sky_statistic_gradient_sky_grid(
    theta_max_1,phi_max_1,
  theta_pixels,phi_pixels,n_max,threshold,
  downsampled_Fourier_L1_response,downsampled_Fourier_H1_response,
```

```
    downsampled_Fourier_V1_response,
  downsampled_Fourier_L1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_H1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_V1_noise_power_spectral_density_sqrt)


latitude_1_path=90.-(theta_1_path)*(180./pi)
longitude_1_path=(phi_1_path)*(180./pi)

latitude_1_path_figure=plt.figure()
plt.xlabel('Iteration')
plt.ylabel('Latitude')
plt.title('Latitude 1 path')
plt.plot(iteration,latitude_1_path)

longitude_1_path_figure=plt.figure()
plt.xlabel('Iteration')
plt.ylabel('phi')
plt.title('Longitude 1 path')
plt.plot(iteration,longitude_1_path)

print('Reconstructed latitude 1')
print((pi/2.-theta_max_1)*180./pi)
print('Reconstructed longitude 1')
print((phi_max_1)*180./pi)


theta_max_2,phi_max_2,iteration,theta_2_path,phi_2_path=pm.sky_statistic_gradient_sky_grid(
    theta_max_2,phi_max_2,
  theta_pixels,phi_pixels,n_max,threshold,
  downsampled_Fourier_L1_response,downsampled_Fourier_H1_response,
    downsampled_Fourier_V1_response,
  downsampled_Fourier_L1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_H1_noise_power_spectral_density_sqrt,
    downsampled_Fourier_V1_noise_power_spectral_density_sqrt)


latitude_2_path=90.-(theta_2_path)*(180./pi)
longitude_2_path=(phi_2_path)*(180./pi)

latitude_2_path_figure=plt.figure()
plt.xlabel('Iteration')
plt.ylabel('Latitude')
plt.title('Latitude 2 path')
plt.plot(iteration,latitude_2_path)

longitude_2_path_figure=plt.figure()
plt.xlabel('Iteration')
plt.ylabel('phi')
plt.title('Longitude 2 path')
plt.plot(iteration,longitude_2_path)

print('Reconstructed latitude 2')
print((pi/2.-theta_max_2)*180./pi)
print('Reconstructed longitude 2')
print((phi_max_2)*180./pi)









theta_grid=np.arange(0.,+pi+0.5*pi/theta_pixels,+pi/theta_pixels)
phi_grid=np.arange(-pi,+pi+0.5*2.*pi/phi_pixels,+2.*pi/phi_pixels)
latitude_grid=90.-(theta_grid)*(180./pi)
longitude_grid=phi_grid*180./pi

general_description=sky_statistic_title+new_line+latitude_str+new_line+longitude_str+
```

```
    new_line+masses+new_line+noise_scaling_factors

sky_statistic_figure = plt.figure()
m=Basemap(projection='mill',lat_ts=0,llcrnrlon=longitude_grid.min(), \
  urcrnrlon=longitude_grid.max(),llcrnrlat=latitude_grid.min(),urcrnrlat=latitude_grid.max()
    , \
  resolution='c')
longitude_grid, latitude_grid = m(*np.meshgrid(longitude_grid, latitude_grid))
plt.title(general_description)
m.pcolormesh(longitude_grid, latitude_grid,sky_statistic,shading='flat',cmap='YlGnBu')
m.colorbar(location='right')
m.drawcoastlines()
m.drawmapboundary()
m.drawparallels(np.arange(-90.,90.,30.),labels=[1,0,0,0])
m.drawmeridians(np.arange(-180.,180.,60.),labels=[0,0,0,1])




time_delay_parameter_1=pm.time_delay_analysis(time_delay_L1,time_delay_H1,time_delay_V1,
    theta_max_1,phi_max_1)
time_delay_parameter_2=pm.time_delay_analysis(time_delay_L1,time_delay_H1,time_delay_V1,
    theta_max_2,phi_max_2)

print('time delay parameter 1')
print(time_delay_parameter_1)

print('time delay parameter 2')
print(time_delay_parameter_2)



if(time_delay_parameter_1<time_delay_parameter_2):
  theta_reconstructed=theta_max_1
  phi_reconstructed=phi_max_1


elif(time_delay_parameter_2<time_delay_parameter_1):
  theta_reconstructed=theta_max_2
  phi_reconstructed=phi_max_2









print('Reconstructed latitude ')
print((pi/2.-theta_reconstructed)*180./pi)
print('Reconstructed longitude ')
print((phi_reconstructed)*180./pi)


print('Difference between reconstructed and injected latitude')
print(((pi/2.-theta_reconstructed)-(pi/2.-theta))*180./pi)
print('Difference between reconstructed and injected longitude ')
print(((pi/2.-phi_reconstructed)-(pi/2.-phi))*180./pi)




h_plus,h_cross=pm.polarization_reconstruction(theta_reconstructed,phi_reconstructed,
    Fourier_L1_response_whitened,Fourier_H1_response_whitened,Fourier_V1_response_whitened,
  Fourier_L1_noise_power_spectral_density_sqrt,Fourier_H1_noise_power_spectral_density_sqrt,
    Fourier_V1_noise_power_spectral_density_sqrt)



```

```
658
659 h_plus_figure=plt.figure()
660 plt.xlabel('Time (s)')
661 plt.ylabel('Strain amplitude')
662 plt.title('$h_{+}$')
663 plt.grid()
664 plt.plot(t_abs,h_plus,label='$h_{+}$ reconstructed',color='r')
665 plt.plot(t_abs,h_plus_data,label='$h_{+}$ injected',color='b')
666 plt.legend()
667
668 h_cross_figure=plt.figure()
669 plt.xlabel('Time (s)')
670 plt.ylabel('Strain amplitude')
671 plt.title('$h_{x}$')
672 plt.grid()
673 plt.plot(t_abs,h_cross,label='$h_{x}$ reconstructed',color='r')
674 plt.plot(t_abs,h_cross_data,label='$h_{x}$ injected',color='b')
675 plt.legend()
676
677
678 overlap_h_plus=pm.overlap(h_plus,h_plus_data)
679 overlap_h_cross=pm.overlap(h_cross,h_cross_data)
680
681 print('Overlap h plus for reconstructed source location')
682 print(overlap_h_plus)
683
684 print('Overlap h cross for reconstructed source location')
685 print(overlap_h_cross)
686
687
688
689
690 plt.show()
```

polarization_Gursel_Tinto.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.98847e30                           # kg
pi=constants.pi



#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
    phi_V1), 6368239.*np.cos(theta_V1)])

```

```
63
64
65
66
67
68
69 L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
70 H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
71 V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)
72
73
74
75
76 print('Do you want to use default values for the waveforms? If so digit 1, if not other')
77 boolean_data=int(input())
78
79
80 f_s=16384
81
82 if(boolean_data==1):
83   print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
       another number for the BBH with masses 30.7 and 25.3')
84   boolean_0=int(input())
85
86   if(boolean_0==1):
87     mass_1=11.0
88     mass_2=7.6
89     d=320. # Mpc
90     h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
       mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
91     t_data=h_plus_data.sample_times
92   else:
93     mass_1=30.7
94     mass_2=25.3
95     d=600. # Mpc
96     h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1,
       mass2=mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
97     t_data=h_plus_data.sample_times
98
99
100 else:
101
102   print('Write the masses of the bodies (in solar masses)')
103   print('m_1=')
104   mass_1=float(input())
105   print('m_2=')
106   mass_2=float(input())
107   print('Write the luminosity distance of the bodies (in Mpc)')
108   d=float(input())
109   h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
       =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
110   t_data=h_plus_data.sample_times
111
112 mass_1_text=str(round(mass_1,1))
113 mass_2_text=str(round(mass_2,1))
114 d_text=str(round(d,0))
115 polarizations_figure, (h_plus, h_cross) = plt.subplots(1, 2)
116 polarizations_figure.suptitle('$h_{+}$ and $h_{x}$ polarizations ($M_{1}=$ %s $M_\odot$, $M_
    {2}=$ %s $M_\odot$, $d_{L}=$ %s Mpc)'%(mass_1_text,mass_2_text,d_text))
117 h_plus.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
118 h_plus.set_title('$h_{+}$')
119 h_plus.grid()
120 h_plus.plot(t_data,h_plus_data)
121 h_cross.set(xlabel='Time ($s$)', ylabel='')
122 h_cross.set_title('$h_{+}$')
123 h_cross.grid()
124 h_cross.plot(t_data,h_cross_data)
125
```

```
126
127
128 print('Write the latitude of the source (in degrees)')
129 latitude_degrees=float(input())
130 latitude=latitude_degrees*pi/180.
131 theta=pi/2.-latitude
132
133 print('Write the longitude of the source (in degrees)')
134 longitude_degrees=float(input())
135 longitude=longitude_degrees*pi/180.
136 phi=longitude
137
138
139
140 n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(theta)
        ]) #incoming direction
141
142 # here the source direction is considered in the detector frame
143 n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
144 n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
145 n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
146
147
148 cos_theta_L1=n_L1[2]
149 cos_theta_H1=n_H1[2]
150 cos_theta_V1=n_V1[2]
151
152 cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
153 cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
154 cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
155
156 sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
157 sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
158 sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
159
160
161
162 time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
163 time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
164 time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)
165
166
167 print('L1 time delay=',time_delay_L1,'s')
168 print('H1 time delay=',time_delay_H1,'s')
169 print('V1 time delay=',time_delay_V1,'s')
170
171
172
173
174 # this is the maximum delay time at which the intefrerometer can detect the signal
175 max_delay=0.022
176
177
178 # antenna patterns calculation
179 F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
180 F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
181 F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
182 F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
183 F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
184 F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)
185
186
187
188
189 t_min=min(t_data)
190 t_max=max(t_data)
191
192 t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)
```

```
193
194
195 # here I have to obtain an array with even number of elements: this will be important during
        the noise generation in frequency domain
196 if len(t)%2!=0:
197   t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)
198
199
200
201 # parameters for noise generation
202
203 L=4000.    # m
204 f_minus=390. # Hz
205 P_arm=240000.    # Watt
206 eta=0.75
207 f_min=10.    # Hz
208 f_max=8192. # Hz
209
210
211
212 frequency=fftpack.fftfreq(len(t)) * f_s
213
214
215 print('Now you can decide to multiply the noise for a scaling factor (for default values
        insert multiples of 1,1,3.3)')
216 print('Write the scaling factor for L1 noise')
217 scaling_factor_L1=float(input())
218 print('Write the scaling factor for H1 noise')
219 scaling_factor_H1=float(input())
220 print('Write the scaling factor for V1 noise')
221 scaling_factor_V1=float(input())
222
223
224
225
226 # noise generation
227 mu=0.
228 sigma=1.
229
230 Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
        ,scaling_factor_L1)
231 Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
        ,scaling_factor_H1)
232 Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,len(t)
        ,scaling_factor_V1)
233
234
235
236
237 # this generate a graph of the noise strain sensitivity
238 strain_noise_L1=np.zeros(len(frequency))
239 for i in range(len(strain_noise_L1)):
240   strain_noise_L1[i]=scaling_factor_L1*nm.noise_strain_sensitivity(frequency[i],f_minus,
        P_arm,eta,L)
241
242 strain_noise_figure=plt.figure()
243 plt.xlabel('Frequency (Hz)')
244 plt.xlim(9., 8191.)
245 plt.ylabel('Strain noise ($1/\sqrt{Hz}$)')
246 plt.title('Noise strain sensitivity')
247 plt.grid()
248 plt.loglog(frequency,strain_noise_L1)
249
250
251
252 L1_response=L1_noise
253 H1_response=H1_noise
254 V1_response=V1_noise
```

```
255
256
257
258
259 i_trigger_L1=1e10
260 i_trigger_H1=1e10
261 i_trigger_V1=1e10
262
263
264 # now I generate the detector responses
265 for i in range(len(t)):
266
267
268   if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:
269
270     if i<i_trigger_L1:
271       i_trigger_L1=i
272
273     L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
      h_cross_data[i-i_trigger_L1]
274
275   if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:
276
277     if i<i_trigger_H1:
278       i_trigger_H1=i
279
280     H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*
      h_cross_data[i-i_trigger_H1]
281
282   if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:
283
284     if i<i_trigger_V1:
285       i_trigger_V1=i
286
287     V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*
      h_cross_data[i-i_trigger_V1]
288
289
290
291
292
293 # now the signals have to be aligned in time
294
295 t_abs=np.arange(t_min, t_max, 1./f_s)
296 t_abs=np.append(t_abs,t_max) # this is for including the last array element
297 # also here I have to obtain an array with even number of elements
298 if len(t_abs)%2!=0:
299   t_abs=np.append(t_abs,t_max+1./f_s)
300   h_plus_data=np.append(h_plus_data,0.)
301   h_cross_data=np.append(h_cross_data,0.)
302
303
304 frequency=fftpack.fftfreq(len(t_abs))*f_s
305
306 L1_response_aligned=np.zeros(len(t_abs))
307 H1_response_aligned=np.zeros(len(t_abs))
308 V1_response_aligned=np.zeros(len(t_abs))
309
310 sigma_2_L1=0.
311 sigma_2_H1=0.
312 sigma_2_V1=0.
313
314 for i in range(len(t_abs)):
315   L1_response_aligned[i]=L1_response[i+i_trigger_L1]
316   H1_response_aligned[i]=H1_response[i+i_trigger_H1]
317   V1_response_aligned[i]=V1_response[i+i_trigger_V1]
318   sigma_2_L1=sigma_2_L1+L1_noise[i+i_trigger_L1]**2
319   sigma_2_H1=sigma_2_H1+H1_noise[i+i_trigger_H1]**2
```

```
  sigma_2_L1=sigma_2_V1+V1_noise[i+i_trigger_V1]**2

sigma_2_L1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_L1
sigma_2_H1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_H1
sigma_2_V1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_V1



Fourier_L1_response_aligned=fftpack.fft(L1_response_aligned)
Fourier_H1_response_aligned=fftpack.fft(H1_response_aligned)
Fourier_V1_response_aligned=fftpack.fft(V1_response_aligned)



Signal=(np.linalg.norm(Fourier_L1_response_aligned)**2+np.linalg.norm(
    Fourier_H1_response_aligned)**2+np.linalg.norm(Fourier_V1_response_aligned)**2)
Noise=(np.linalg.norm(Fourier_noise_L1)**2+np.linalg.norm(Fourier_noise_H1)**2+np.linalg.
    norm(Fourier_noise_V1)**2)
SNR=Signal/Noise

print('SNR')
print(round(SNR,2))







detector_responses_figure, (L1, H1, V1) = plt.subplots(1, 3)
detector_responses_figure.suptitle('Detector responses')
L1.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
L1.set_title('L1')
L1.grid()
L1.plot(t_abs,L1_response_aligned)
H1.set(xlabel='Time ($s$)', ylabel='')
H1.set_title('H1')
H1.grid()
H1.plot(t_abs,H1_response_aligned)
V1.set(xlabel='Time ($s$)', ylabel='')
V1.set_title('V1')
V1.grid()
V1.plot(t_abs,V1_response_aligned)




# implementation of Gursel & Tinto method
F_plus=np.array([F_plus_L1,F_plus_H1,F_plus_V1])
F_cross=np.array([F_cross_L1,F_cross_H1,F_cross_V1])
sigma_2=np.array([sigma_2_L1,sigma_2_H1,sigma_2_V1])

N_detectors=3

a_plus=np.zeros((N_detectors,N_detectors))

for i in range(N_detectors):
  for j in range(N_detectors):
    if(i!=j):

      first_term=0.5*(sigma_2[i]*(F_cross[j]**2)+sigma_2[j]*(F_cross[i]**2))/((F_plus[i]*
    F_cross[j]-F_cross[i]*F_plus[j])**2)

      second_term=0.
      for l in range(N_detectors):
        for m in range(N_detectors):
          if(l!=m):
            if(l!=i and m!=j):
              second_term=second_term+((F_plus[l]*F_cross[m]-F_cross[l]*F_plus[m])**2)/(
```

```
    sigma_2[l]*(F_cross[m]**2)+sigma_2[m]*(F_cross[l]**2))

      a_plus[i][j]=1./(0.5+first_term*second_term)


a_cross=np.zeros((N_detectors,N_detectors))

for i in range(N_detectors):
  for j in range(N_detectors):
    if(i!=j):

      first_term=0.5*(sigma_2[i]*(F_plus[j]**2)+sigma_2[j]*(F_plus[i]**2))/((F_plus[i]*
    F_cross[j]-F_cross[i]*F_plus[j])**2)

      second_term=0.
      for l in range(N_detectors):
        for m in range(N_detectors):
          if(l!=m):
            if(l!=i and m!=j):
              second_term=second_term+((F_plus[l]*F_cross[m]-F_cross[l]*F_plus[m])**2)/(
    sigma_2[l]*(F_plus[m]**2)+sigma_2[m]*(F_plus[l]**2))

      a_cross[i][j]=1./(0.5+first_term*second_term)


h_plus_01=(F_cross[1]*L1_response_aligned-F_cross[0]*H1_response_aligned)/(F_plus[0]*F_cross
    [1]-F_cross[0]*F_plus[1])
h_plus_02=(F_cross[2]*L1_response_aligned-F_cross[0]*V1_response_aligned)/(F_plus[0]*F_cross
    [2]-F_cross[0]*F_plus[2])
h_plus_12=(F_cross[2]*H1_response_aligned-F_cross[1]*V1_response_aligned)/(F_plus[1]*F_cross
    [2]-F_cross[1]*F_plus[2])



h_cross_01=(F_plus[1]*L1_response_aligned-F_plus[0]*H1_response_aligned)/(F_cross[0]*F_plus
    [1]-F_plus[0]*F_cross[1])
h_cross_02=(F_plus[2]*L1_response_aligned-F_plus[0]*V1_response_aligned)/(F_cross[0]*F_plus
    [2]-F_plus[0]*F_cross[2])
h_cross_12=(F_plus[2]*H1_response_aligned-F_plus[1]*V1_response_aligned)/(F_cross[1]*F_plus
    [2]-F_plus[1]*F_cross[2])



h_plus=a_plus[0][1]*h_plus_01+a_plus[0][2]*h_plus_02+a_plus[1][2]*h_plus_12
h_cross=a_cross[0][1]*h_cross_01+a_cross[0][2]*h_cross_02+a_cross[1][2]*h_cross_12

h_plus_figure=plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{+}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_plus,label='$h_{+}$ reconstructed',color='r')
plt.plot(t_abs,h_plus_data,label='$h_{+}$ injected',color='b')
plt.legend()

h_cross_figure=plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Strain amplitude')
plt.title('$h_{x}$ reconstructed \n assuming known source location')
plt.grid()
plt.plot(t_abs,h_cross,label='$h_{x}$ reconstructed',color='r')
plt.plot(t_abs,h_cross_data,label='$h_{x}$ injected',color='b')
plt.legend()


reconstructed_polarizations_figure, (h_plus_reconstructed,h_cross_reconstructed) = plt.
    subplots(1, 2)
reconstructed_polarizations_figure.suptitle('Reconstructed polarizations with Gursel & Tinto
```

```
    ')
h_plus_reconstructed.set(xlabel='Time ($s$)', ylabel='Strain amplitude')
h_plus_reconstructed.set_title('$h_{+}$ reconstructed')
h_plus_reconstructed.grid()
h_plus_reconstructed.plot(t_abs,h_plus)
h_cross_reconstructed.set(xlabel='Time ($s$)', ylabel='')
h_cross_reconstructed.set_title('$h_{x}$ reconstructed')
h_cross_reconstructed.grid()
h_cross_reconstructed.plot(t_abs,h_cross)



overlap_h_plus=pm.overlap(h_plus,h_plus_data)
overlap_h_cross=pm.overlap(h_cross,h_cross_data)

print('Overlap h plus')
print(overlap_h_plus)

print('Overlap h cross')
print(overlap_h_cross)

plt.show()
```

polarization_cWB_overlap.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
import pywt
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.989e30                             # kg
pi=constants.pi




#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
```

```
    phi_V1), 6368239.*np.cos(theta_V1)])

L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)


f_s=16384 # Hz

print('This program calculate the overlap for many noise scaling factors')

print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
    another number for the BBH with masses 30.7 and 25.3')
boolean_0=int(input())

if(boolean_0==1):
  mass_1=11.0
  mass_2=7.6
  mass_1_text='11_0'
  mass_2_text='7_6'
  d=320. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times
else:
  mass_1=30.7
  mass_2=25.3
  mass_1_text='30_7'
  mass_2_text='25_3'
  d=600. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times




h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2=
    mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
t_data=h_plus_data.sample_times

t_min=min(t_data)
t_max=max(t_data)

# this is for aligning the signals in time

t_abs=np.arange(t_min, t_max, 1./f_s)
t_abs=np.append(t_abs,t_max) # this is for including the last array element
# here I have to obtain an array with even number of elements
if len(t_abs)%2!=0:
  t_abs=np.append(t_abs,t_max+1./f_s)
  h_plus_data=np.append(h_plus_data,0.)
  h_cross_data=np.append(h_cross_data,0.)


# here I define many noise scaling factors for calculating the signals
scale=1.4
start=-9
scaling_factor=np.array([(scale)**start])
len_scaling_factor=28

for i in range(1,len_scaling_factor):
  scaling_factor=np.append(scaling_factor,scale**(start+i))



theta_pixels=3
phi_pixels=3
```

```
127
128 SNR_total=np.array([0])
129 overlap_h_plus_total=np.array([0])
130 overlap_h_cross_total=np.array([0])
131
132 count=0
133
134
135
136
137 for lat_index in range(theta_pixels):
138   for long_index in range(phi_pixels):
139
140
141     latitude=(10.+160.*lat_index/(theta_pixels-1.))*pi/180.
142     longitude=(0.+360.*long_index/(phi_pixels-1.))*pi/180.
143
144     theta=pi/2.-latitude
145     phi=longitude
146
147     n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(
      theta)]) #incoming direction
148
149     n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
150     n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
151     n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
152
153
154     cos_theta_L1=n_L1[2]
155     cos_theta_H1=n_H1[2]
156     cos_theta_V1=n_V1[2]
157
158     cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
159     cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
160     cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
161
162     sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
163     sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
164     sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
165
166
167
168     time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
169     time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
170     time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)
171
172
173
174
175     max_delay=0.022 # this is the maximum delay time at which the intefrerometer can detect
      the signal
176     F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
177     F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
178     F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
179     F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
180     F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
181     F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)
182
183
184
185
186     t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)
187
188     # here I have to obtain an array with even number of elements
189     if len(t)%2!=0:
190       t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)
191
192
```

```
    #noise

    L=4000.    # m
    f_minus=390. # Hz
    P_arm=240000.    # Watt
    eta=0.75
    f_min=10.    # Hz
    f_max=8192. # Hz




    frequency=fftpack.fftfreq(len(t)) * f_s


    Virgo_factor=3.3

    overlap_h_plus=np.zeros(len(scaling_factor))
    overlap_h_cross=np.zeros(len(scaling_factor))


    for l in range(len(scaling_factor)):

      scaling_factor_L1=scaling_factor[l]
      scaling_factor_H1=scaling_factor[l]
      scaling_factor_V1=scaling_factor[l]*Virgo_factor


      mu=0.
      sigma=1.

      Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_L1)
      Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_H1)
      Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_V1)


      L1_noise_backup=(fftpack.ifft(Fourier_noise_L1)).real
      H1_noise_backup=(fftpack.ifft(Fourier_noise_H1)).real
      V1_noise_backup=(fftpack.ifft(Fourier_noise_V1)).real



      L1_response=L1_noise
      H1_response=H1_noise
      V1_response=V1_noise


      i_trigger_L1=1e10
      i_trigger_H1=1e10
      i_trigger_V1=1e10

      for i in range(len(t)):


        if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:

          if i<i_trigger_L1:
            i_trigger_L1=i

          L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
    h_cross_data[i-i_trigger_L1]

        if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:

          if i<i_trigger_H1:
```

```
            i_trigger_H1=i

        H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*h_cross_data[i-i_trigger_H1]

    if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:

        if i<i_trigger_V1:
            i_trigger_V1=i

        V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*h_cross_data[i-i_trigger_V1]



  L1_response_aligned=np.zeros(len(t_abs))
  H1_response_aligned=np.zeros(len(t_abs))
  V1_response_aligned=np.zeros(len(t_abs))
  L1_noise_aligned=np.zeros(len(t_abs))
  H1_noise_aligned=np.zeros(len(t_abs))
  V1_noise_aligned=np.zeros(len(t_abs))

  for i in range(len(t_abs)):
    L1_response_aligned[i]=L1_response[i+i_trigger_L1]
    H1_response_aligned[i]=H1_response[i+i_trigger_H1]
    V1_response_aligned[i]=V1_response[i+i_trigger_V1]
    L1_noise_aligned[i]=L1_noise_backup[i+i_trigger_L1]
    H1_noise_aligned[i]=H1_noise_backup[i+i_trigger_H1]
    V1_noise_aligned[i]=V1_noise_backup[i+i_trigger_V1]

  Signal=np.sum(L1_response_aligned**2)+np.sum(H1_response_aligned**2)+np.sum(V1_response_aligned**2)
  Noise=np.sum(L1_noise_aligned**2)+np.sum(H1_noise_aligned**2)+np.sum(V1_noise_aligned**2)
  SNR=Signal/Noise


  frequency=fftpack.fftfreq(len(t_abs)) * f_s


  Fourier_L1_response=fftpack.fft(L1_response_aligned)
  Fourier_H1_response=fftpack.fft(H1_response_aligned)
  Fourier_V1_response=fftpack.fft(V1_response_aligned)


  Fourier_L1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_L1
  Fourier_H1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_H1
  Fourier_V1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_V1


  Fourier_L1_response_whitened=Fourier_L1_response/(Fourier_L1_noise_power_spectral_density_sqrt)
  Fourier_H1_response_whitened=Fourier_H1_response/(Fourier_H1_noise_power_spectral_density_sqrt)
  Fourier_V1_response_whitened=Fourier_V1_response/(Fourier_V1_noise_power_spectral_density_sqrt)


  h_plus,h_cross=pm.polarization_reconstruction(theta,phi,Fourier_L1_response_whitened,Fourier_H1_response_whitened,Fourier_V1_response_whitened,
      Fourier_L1_noise_power_spectral_density_sqrt,Fourier_H1_noise_power_spectral_density_sqrt,Fourier_V1_noise_power_spectral_density_sqrt)


```

```
312
313       overlap_h_plus[l]=pm.overlap(h_plus,h_plus_data)
314       overlap_h_cross[l]=pm.overlap(h_cross,h_cross_data)
315
316       SNR_total=np.append(SNR_total,SNR)
317       overlap_h_plus_total=np.append(overlap_h_plus_total,overlap_h_plus[l])
318       overlap_h_cross_total=np.append(overlap_h_cross_total,overlap_h_cross[l])
319
320       count=count+1
321
322     print((100*count/(len(scaling_factor)*theta_pixels*phi_pixels)),'%')
323
324
325 SNR_total=np.delete(SNR_total,0)
326 overlap_h_plus_total=np.delete(overlap_h_plus_total,0)
327 overlap_h_cross_total=np.delete(overlap_h_cross_total,0)
328
329
330 new_line='\n'
331 space=' '
332 comma=','
333 masses='bodies masses (in Solar masses)='+str(mass_1)+comma+str(mass_2)
334 overlap_description='Overlap for cWB wavelets denoising algorithm '
335 title=overlap_description+comma+new_line+masses
336
337 # here I make a scatterplot of the overlap versus the SNR
338 overlap_scatterplot, (h_plus, h_cross) = plt.subplots(1, 2)
339 overlap_scatterplot.suptitle(title)
340 h_plus.set(xlabel='SNR', ylabel='Overlap')
341 h_plus.set_title('$h_{+}$')
342 h_plus.scatter(SNR_total,overlap_h_plus_total)
343 h_plus.set_xscale('log')
344 h_plus.set_yscale('log')
345 h_cross.set(xlabel='SNR', ylabel='')
346 h_cross.set_title('$h_{x}$')
347 h_cross.scatter(SNR_total,overlap_h_cross_total)
348 h_cross.set_xscale('log')
349 h_cross.set_yscale('log')
350
351 save_to_file=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
352 text_name_scatterplot='overlap_cWB_'+mass_1_text+'__'+mass_2_text+'_scatterplot.txt'
353 np.savetxt(text_name_scatterplot, save_to_file)
354
355
356 # this is for reordering the arrays in order to have increasing values for the SNR
357 for i in range(len(SNR_total)):
358   swap = i + np.argmin(SNR_total[i:])
359   (SNR_total[i], SNR_total[swap]) = (SNR_total[swap], SNR_total[i])
360   (overlap_h_plus_total[i], overlap_h_plus_total[swap]) = (overlap_h_plus_total[swap],
        overlap_h_plus_total[i])
361   (overlap_h_cross_total[i], overlap_h_cross_total[swap]) = (overlap_h_cross_total[swap],
        overlap_h_cross_total[i])
362
363 # here I graph the overlap versus the SNR as a line
364 overlap_figure, (h_plus, h_cross) = plt.subplots(1, 2)
365 overlap_figure.suptitle(title)
366 h_plus.set(xlabel='SNR', ylabel='Overlap')
367 h_plus.set_title('$h_{+}$')
368 h_plus.plot(SNR_total,overlap_h_plus_total)
369 h_plus.set_xscale('log')
370 h_plus.set_yscale('log')
371 h_cross.set(xlabel='SNR', ylabel='')
372 h_cross.set_title('$h_{x}$')
373 h_cross.plot(SNR_total,overlap_h_cross_total)
374 h_cross.set_xscale('log')
375 h_cross.set_yscale('log')
376
377
```

```
378
379 save_to_file_line=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
380 text_name_line='overlap_cWB_'+mass_1_text+'__'+mass_2_text+'_line.txt'
381 np.savetxt(text_name_line, save_to_file_line)
382
383
384
385
386
387 plt.show()
```

polarization_wavelets_denoising_overlap.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
import pywt
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.989e30                             # kg
pi=constants.pi




#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
```

```
    phi_V1), 6368239.*np.cos(theta_V1)])

L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)


f_s=16384 # Hz

print('This program calculate the overlap for many noise scaling factors')

print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
    another number for the BBH with masses 30.7 and 25.3')
boolean_0=int(input())

if(boolean_0==1):
  mass_1=11.0
  mass_2=7.6
  mass_1_text='11_0'
  mass_2_text='7_6'
  d=320. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times
else:
  mass_1=30.7
  mass_2=25.3
  mass_1_text='30_7'
  mass_2_text='25_3'
  d=600. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times




h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2=
    mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
t_data=h_plus_data.sample_times

t_min=min(t_data)
t_max=max(t_data)

# this is for aligning the signals in time

t_abs=np.arange(t_min, t_max, 1./f_s)
t_abs=np.append(t_abs,t_max) # this is for including the last array element
# here I have to obtain an array with even number of elements
if len(t_abs)%2!=0:
  t_abs=np.append(t_abs,t_max+1./f_s)
  h_plus_data=np.append(h_plus_data,0.)
  h_cross_data=np.append(h_cross_data,0.)


# here I define many noise scaling factors for calculating the signals
scale=1.4
start=-9
scaling_factor=np.array([(scale)**start])
len_scaling_factor=28

for i in range(1,len_scaling_factor):
  scaling_factor=np.append(scaling_factor,scale**(start+i))



theta_pixels=3
phi_pixels=3
```

```
127
128 SNR_total=np.array([0])
129 overlap_h_plus_total=np.array([0])
130 overlap_h_cross_total=np.array([0])
131
132 count=0
133
134
135 for lat_index in range(theta_pixels):
136   for long_index in range(phi_pixels):
137
138
139     latitude=(10.+160.*lat_index/(theta_pixels-1.))*pi/180.
140     longitude=(0.+360.*long_index/(phi_pixels-1.))*pi/180.
141
142     theta=pi/2.-latitude
143     phi=longitude
144
145     n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(
        theta)]) #incoming direction
146
147     n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
148     n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
149     n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
150
151
152     cos_theta_L1=n_L1[2]
153     cos_theta_H1=n_H1[2]
154     cos_theta_V1=n_V1[2]
155
156     cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
157     cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
158     cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
159
160     sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
161     sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
162     sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
163
164
165
166     time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
167     time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
168     time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)
169
170
171
172
173     max_delay=0.022 # this is the maximum delay time at which the intefrerometer can detect
        the signal
174
175     F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
176     F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
177     F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
178     F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
179     F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
180     F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)
181
182
183
184
185
186
187
188     t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)
189
190     # here I have to obtain an array with even number of elements
191     if len(t)%2!=0:
192       t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)
```

```
193
194
195     #noise
196
197     L=4000.     # m
198     f_minus=390. # Hz
199     P_arm=240000.     # Watt
200     eta=0.75
201     f_min=10.     # Hz
202     f_max=8192. # Hz
203
204
205     frequency=fftpack.fftfreq(len(t)) * f_s
206
207
208
209     Virgo_factor=3.3
210
211     overlap_h_plus=np.zeros(len(scaling_factor))
212     overlap_h_cross=np.zeros(len(scaling_factor))
213
214
215     for l in range(len(scaling_factor)):
216
217       scaling_factor_L1=scaling_factor[l]
218       scaling_factor_H1=scaling_factor[l]
219       scaling_factor_V1=scaling_factor[l]*Virgo_factor
220
221
222
223
224       mu=0.
225       sigma=1.
226
227       Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_L1)
228       Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_H1)
229       Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_V1)
230
231
232
233       L1_noise_backup=(fftpack.ifft(Fourier_noise_L1)).real
234       H1_noise_backup=(fftpack.ifft(Fourier_noise_H1)).real
235       V1_noise_backup=(fftpack.ifft(Fourier_noise_V1)).real
236
237
238
239       L1_response=L1_noise
240       H1_response=H1_noise
241       V1_response=V1_noise
242
243
244       i_trigger_L1=1e10
245       i_trigger_H1=1e10
246       i_trigger_V1=1e10
247
248       for i in range(len(t)):
249
250
251         if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:
252
253           if i<i_trigger_L1:
254             i_trigger_L1=i
255
256           L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
    h_cross_data[i-i_trigger_L1]
```

```
257
258         if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:
259
260           if i<i_trigger_H1:
261             i_trigger_H1=i
262
263           H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*
    h_cross_data[i-i_trigger_H1]
264
265         if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:
266
267           if i<i_trigger_V1:
268             i_trigger_V1=i
269
270           V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*
    h_cross_data[i-i_trigger_V1]
271
272
273
274
275
276
277
278       L1_response_aligned=np.zeros(len(t_abs))
279       H1_response_aligned=np.zeros(len(t_abs))
280       V1_response_aligned=np.zeros(len(t_abs))
281       L1_noise_aligned=np.zeros(len(t_abs))
282       H1_noise_aligned=np.zeros(len(t_abs))
283       V1_noise_aligned=np.zeros(len(t_abs))
284
285       for i in range(len(t_abs)):
286         L1_response_aligned[i]=L1_response[i+i_trigger_L1]
287         H1_response_aligned[i]=H1_response[i+i_trigger_H1]
288         V1_response_aligned[i]=V1_response[i+i_trigger_V1]
289         L1_noise_aligned[i]=L1_noise_backup[i+i_trigger_L1]
290         H1_noise_aligned[i]=H1_noise_backup[i+i_trigger_H1]
291         V1_noise_aligned[i]=V1_noise_backup[i+i_trigger_V1]
292
293       Signal=np.sum(L1_response_aligned**2)+np.sum(H1_response_aligned**2)+np.sum(
    V1_response_aligned**2)
294       Noise=np.sum(L1_noise_aligned**2)+np.sum(H1_noise_aligned**2)+np.sum(V1_noise_aligned
    **2)
295       SNR=Signal/Noise
296
297       # wavelet denoising
298       # for each signal I take the approximation coefficients (cA) of the wavelet transofrm,
     discarding the details one (cD)
299
300       wavelet='dmey' # Meyer wavelets
301
302       (L1_response_aligned_cA, L1_response_aligned_cD)=pywt.dwt(L1_response_aligned, wavelet
    , mode='zero', axis=-1)
303       (H1_response_aligned_cA, H1_response_aligned_cD)=pywt.dwt(H1_response_aligned, wavelet
    , mode='zero', axis=-1)
304       (V1_response_aligned_cA, V1_response_aligned_cD)=pywt.dwt(V1_response_aligned, wavelet
    , mode='zero', axis=-1)
305
306       zero=np.zeros(len(L1_response_aligned_cD))
307
308       L1_response_aligned_cA_reconstruction=pywt.idwt(L1_response_aligned_cA,zero,wavelet,
    mode='zero', axis=-1)
309       H1_response_aligned_cA_reconstruction=pywt.idwt(H1_response_aligned_cA,zero,wavelet,
    mode='zero', axis=-1)
310       V1_response_aligned_cA_reconstruction=pywt.idwt(V1_response_aligned_cA,zero,wavelet,
    mode='zero', axis=-1)
311
312
313
```

```
314
315       frequency=fftpack.fftfreq(len(t_abs))*f_s
316
317
318
319
320       Fourier_L1_response=fftpack.fft(L1_response_aligned_cA_reconstruction)
321       Fourier_H1_response=fftpack.fft(H1_response_aligned_cA_reconstruction)
322       Fourier_V1_response=fftpack.fft(V1_response_aligned_cA_reconstruction)
323
324
325
326
327
328       Fourier_L1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,
      f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_L1
329       Fourier_H1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,
      f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_H1
330       Fourier_V1_noise_power_spectral_density_sqrt=nm.power_spectral_density_sqrt(frequency,
      f_s,f_minus,P_arm,eta,L,len(frequency))*scaling_factor_V1
331
332
333       Fourier_L1_response_whitened=Fourier_L1_response/(
      Fourier_L1_noise_power_spectral_density_sqrt)
334       Fourier_H1_response_whitened=Fourier_H1_response/(
      Fourier_H1_noise_power_spectral_density_sqrt)
335       Fourier_V1_response_whitened=Fourier_V1_response/(
      Fourier_V1_noise_power_spectral_density_sqrt)
336
337
338       h_plus,h_cross=pm.polarization_reconstruction(theta,phi,Fourier_L1_response_whitened,
      Fourier_H1_response_whitened,Fourier_V1_response_whitened,
339           Fourier_L1_noise_power_spectral_density_sqrt,
      Fourier_H1_noise_power_spectral_density_sqrt,
      Fourier_V1_noise_power_spectral_density_sqrt)
340
341
342
343       overlap_h_plus[l]=pm.overlap(h_plus,h_plus_data)
344       overlap_h_cross[l]=pm.overlap(h_cross,h_cross_data)
345
346       SNR_total=np.append(SNR_total,SNR)
347       overlap_h_plus_total=np.append(overlap_h_plus_total,overlap_h_plus[l])
348       overlap_h_cross_total=np.append(overlap_h_cross_total,overlap_h_cross[l])
349
350       count=count+1
351
352
353     print((100*count/(len(scaling_factor)*theta_pixels*phi_pixels)),'%')
354
355
356 SNR_total=np.delete(SNR_total,0)
357 overlap_h_plus_total=np.delete(overlap_h_plus_total,0)
358 overlap_h_cross_total=np.delete(overlap_h_cross_total,0)
359
360
361 new_line='\n'
362 space=' '
363 comma=','
364 masses='bodies masses (in Solar masses)='+str(mass_1)+comma+str(mass_2)
365 overlap_description='Overlap for cWB wavelets denoising algorithm '
366 title=overlap_description+comma+new_line+masses
367
368 # here I make a scatterplot of the overlap versus the SNR
369 overlap_scatterplot, (h_plus, h_cross) = plt.subplots(1, 2)
370 overlap_scatterplot.suptitle(title)
371 h_plus.set(xlabel='SNR', ylabel='Overlap')
372 h_plus.set_title('$h_{+}$')
```

```
h_plus.scatter(SNR_total,overlap_h_plus_total)
h_plus.set_xscale('log')
h_plus.set_yscale('log')
h_cross.set(xlabel='SNR', ylabel='')
h_cross.set_title('$h_{x}$')
h_cross.scatter(SNR_total,overlap_h_cross_total)
h_cross.set_xscale('log')
h_cross.set_yscale('log')

save_to_file=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
text_name_scatterplot='overlap_cWB_wavelets_denoising_'+mass_1_text+'__'+mass_2_text+'
    _scatterplot.txt'
np.savetxt(text_name_scatterplot, save_to_file)


# this is for reordering the arrays in order to have increasing values for the SNR
for i in range(len(SNR_total)):
  swap = i + np.argmin(SNR_total[i:])
  (SNR_total[i], SNR_total[swap]) = (SNR_total[swap], SNR_total[i])
  (overlap_h_plus_total[i], overlap_h_plus_total[swap]) = (overlap_h_plus_total[swap],
    overlap_h_plus_total[i])
  (overlap_h_cross_total[i], overlap_h_cross_total[swap]) = (overlap_h_cross_total[swap],
    overlap_h_cross_total[i])

# here I graph the overlap versus the SNR as a line
overlap_figure, (h_plus, h_cross) = plt.subplots(1, 2)
overlap_figure.suptitle(title)
h_plus.set(xlabel='SNR', ylabel='Overlap')
h_plus.set_title('$h_{+}$')
h_plus.plot(SNR_total,overlap_h_plus_total)
h_plus.set_xscale('log')
h_plus.set_yscale('log')
h_cross.set(xlabel='SNR', ylabel='')
h_cross.set_title('$h_{x}$')
h_cross.plot(SNR_total,overlap_h_cross_total)
h_cross.set_xscale('log')
h_cross.set_yscale('log')



save_to_file_line=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
text_name_line='overlap_cWB_wavelets_denoising_'+mass_1_text+'__'+mass_2_text+'_line.txt'
np.savetxt(text_name_line, save_to_file_line)




plt.show()
```

polarization_Gursel_Tinto_overlap.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'
from matplotlib import colors
import scipy
import scipy.constants as constants
from scipy import fftpack
import polarization_module as pm
import noise_module as nm
import pywt
from pycbc.waveform import get_td_waveform


# constants:
c = constants.speed_of_light                    # m/s
G = constants.gravitational_constant            # m^3/kg/s^2
MSol = 1.989e30                             # kg
pi=constants.pi




#LIGO Livingston Euler angles
alpha_L1=(-46./60.-27.27/3600.)*pi/180.
beta_L1=(59.+26./60+13.58/3600.)*pi/180.
gamma_L1=(242.+42./60.+59.40/3600.)*pi/180.

#LIGO Hanford Euler angles
alpha_H1=(-29.-24./60.-27.57/3600.)*pi/180.
beta_H1=(43.+32./60.+41.47/3600.)*pi/180.
gamma_H1=(170.+59./60.+57.84/3600.)*pi/180.

#Virgo Euler angles
alpha_V1=(100+30./60.+16.19/3600.)*pi/180.
beta_V1=(46.+22./60.+6.91/3600.)*pi/180.
gamma_V1=(115.+34./60.+2.03/3600.)*pi/180.




#LIGO Livingston position angles
theta_L1=pi/2.-(pi/2.-beta_L1)
phi_L1=(alpha_L1-pi/2.)

#LIGO Hanford position angles
theta_H1=pi/2.-(pi/2.-beta_H1)
phi_H1=alpha_H1-pi/2.

#Virgo position angles
theta_V1=pi/2.-(pi/2.-beta_V1)
phi_V1=alpha_V1-pi/2.


#LIGO Livingston position
r_L1=np.array([6372843.*np.sin(theta_L1)*np.cos(phi_L1), 6372843.*np.sin(theta_L1)*np.sin(
    phi_L1), 6372843.*np.cos(theta_L1)])

#LIGO Hanford position
r_H1=np.array([6367275.*np.sin(theta_H1)*np.cos(phi_H1), 6367275.*np.sin(theta_H1)*np.sin(
    phi_H1), 6367275.*np.cos(theta_H1)])

#Virgo position
r_V1=np.array([6368239.*np.sin(theta_V1)*np.cos(phi_V1), 6368239.*np.sin(theta_V1)*np.sin(
```

```
    phi_V1), 6368239.*np.cos(theta_V1)])

L1_Euler_transformation=pm.Euler_transformation(alpha_L1,beta_L1,gamma_L1)
H1_Euler_transformation=pm.Euler_transformation(alpha_H1,beta_H1,gamma_H1)
V1_Euler_transformation=pm.Euler_transformation(alpha_V1,beta_V1,gamma_V1)


f_s=16384 # Hz

print('This program calculate the overlap for many noise scaling factors')

print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
    another number for the BBH with masses 30.7 and 25.3')
boolean_0=int(input())

if(boolean_0==1):
  mass_1=11.0
  mass_2=7.6
  mass_1_text='11_0'
  mass_2_text='7_6'
  d=320. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times
else:
  mass_1=30.7
  mass_2=25.3
  mass_1_text='30_7'
  mass_2_text='25_3'
  d=600. # Mpc
  h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2
    =mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
  t_data=h_plus_data.sample_times




h_plus_data, h_cross_data = get_td_waveform(approximant='SEOBNRv4_opt',mass1=mass_1, mass2=
    mass_2,delta_t=1.0/f_s,f_lower=30,distance=d)
t_data=h_plus_data.sample_times

t_min=min(t_data)
t_max=max(t_data)

# this is for aligning the signals in time

t_abs=np.arange(t_min, t_max, 1./f_s)
t_abs=np.append(t_abs,t_max) # this is for including the last array element
# here I have to obtain an array with even number of elements
if len(t_abs)%2!=0:
  t_abs=np.append(t_abs,t_max+1./f_s)
  h_plus_data=np.append(h_plus_data,0.)
  h_cross_data=np.append(h_cross_data,0.)


# here I define many noise scaling factors for calculating the signals
scale=1.4
start=-9
scaling_factor=np.array([(scale)**start])
len_scaling_factor=28

for i in range(1,len_scaling_factor):
  scaling_factor=np.append(scaling_factor,scale**(start+i))



theta_pixels=3
phi_pixels=3
```

```
127
128 SNR_total=np.array([0])
129 overlap_h_plus_total=np.array([0])
130 overlap_h_cross_total=np.array([0])
131
132 count=0
133
134
135
136 for lat_index in range(theta_pixels):
137   for long_index in range(phi_pixels):
138
139
140     latitude=(10.+160.*lat_index/(theta_pixels-1.))*pi/180.
141     longitude=(0.+360.*long_index/(phi_pixels-1.))*pi/180.
142
143     theta=pi/2.-latitude
144     phi=longitude
145
146     n_Earth_frame=np.array([np.sin(theta)*np.cos(phi), np.sin(theta)*np.sin(phi), np.cos(
      theta)]) #incoming direction
147
148     n_L1=np.matmul(L1_Euler_transformation,n_Earth_frame)
149     n_H1=np.matmul(H1_Euler_transformation,n_Earth_frame)
150     n_V1=np.matmul(V1_Euler_transformation,n_Earth_frame)
151
152
153     cos_theta_L1=n_L1[2]
154     cos_theta_H1=n_H1[2]
155     cos_theta_V1=n_V1[2]
156
157     cos_2_phi_L1=2.*((n_L1[0]**2)/(1.-(n_L1[2]**2)))-1.
158     cos_2_phi_H1=2.*((n_H1[0]**2)/(1.-(n_H1[2]**2)))-1.
159     cos_2_phi_V1=2.*((n_V1[0]**2)/(1.-(n_V1[2]**2)))-1.
160
161     sin_2_phi_L1=2.*((n_L1[0]*n_L1[1])/(1.-(n_L1[2]**2)))
162     sin_2_phi_H1=2.*((n_H1[0]*n_H1[1])/(1.-(n_H1[2]**2)))
163     sin_2_phi_V1=2.*((n_V1[0]*n_V1[1])/(1.-(n_V1[2]**2)))
164
165
166
167     time_delay_L1=pm.time_delay(r_L1,n_Earth_frame)
168     time_delay_H1=pm.time_delay(r_H1,n_Earth_frame)
169     time_delay_V1=pm.time_delay(r_V1,n_Earth_frame)
170
171
172
173
174     max_delay=0.022 # this is the maximum delay time at which the intefrerometer can detect
      the signal
175     F_plus_L1=pm.F_plus_k(cos_theta_L1,cos_2_phi_L1)
176     F_cross_L1=pm.F_cross_k(cos_theta_L1,sin_2_phi_L1)
177     F_plus_H1=pm.F_plus_k(cos_theta_H1,cos_2_phi_H1)
178     F_cross_H1=pm.F_cross_k(cos_theta_H1,sin_2_phi_H1)
179     F_plus_V1=pm.F_plus_k(cos_theta_V1,cos_2_phi_V1)
180     F_cross_V1=pm.F_cross_k(cos_theta_V1,sin_2_phi_V1)
181
182
183
184
185     t=np.arange(t_min-max_delay, t_max+max_delay, 1./f_s)
186
187     # here I have to obtain an array with even number of elements
188     if len(t)%2!=0:
189       t=np.arange(t_min-max_delay, t_max+max_delay+(1./f_s), 1./f_s)
190
191
192     #noise
```

```
193
194     L=4000.    # m
195     f_minus=390. # Hz
196     P_arm=240000.    # Watt
197     eta=0.75
198     f_min=10.    # Hz
199     f_max=8192. # Hz
200
201
202
203
204     frequency=fftpack.fftfreq(len(t)) * f_s
205
206
207     Virgo_factor=3.3
208
209     overlap_h_plus=np.zeros(len(scaling_factor))
210     overlap_h_cross=np.zeros(len(scaling_factor))
211
212
213     for index in range(len(scaling_factor)):
214
215       scaling_factor_L1=scaling_factor[index]
216       scaling_factor_H1=scaling_factor[index]
217       scaling_factor_V1=scaling_factor[index]*Virgo_factor
218
219
220       mu=0.
221       sigma=1.
222
223       Fourier_noise_L1,L1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_L1)
224       Fourier_noise_H1,H1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_H1)
225       Fourier_noise_V1,V1_noise=nm.Timmer_Koenig(frequency,f_s,f_minus,P_arm,eta,L,mu,sigma,
    len(t),scaling_factor_V1)
226
227
228       L1_noise_backup=(fftpack.ifft(Fourier_noise_L1)).real
229       H1_noise_backup=(fftpack.ifft(Fourier_noise_H1)).real
230       V1_noise_backup=(fftpack.ifft(Fourier_noise_V1)).real
231
232
233
234       L1_response=L1_noise
235       H1_response=H1_noise
236       V1_response=V1_noise
237
238
239       i_trigger_L1=1e10
240       i_trigger_H1=1e10
241       i_trigger_V1=1e10
242
243       for i in range(len(t)):
244
245
246         if t_min+time_delay_L1<t[i]<t_max+time_delay_L1:
247
248           if i<i_trigger_L1:
249             i_trigger_L1=i
250
251           L1_response[i]=L1_response[i]+F_plus_L1*h_plus_data[i-i_trigger_L1]+F_cross_L1*
    h_cross_data[i-i_trigger_L1]
252
253         if t_min+time_delay_H1<t[i]<t_max+time_delay_H1:
254
255           if i<i_trigger_H1:
256             i_trigger_H1=i
```

```
      H1_response[i]=H1_response[i]+F_plus_H1*h_plus_data[i-i_trigger_H1]+F_cross_H1*h_cross_data[i-i_trigger_H1]

    if t_min+time_delay_V1<t[i]<t_max+time_delay_V1:

      if i<i_trigger_V1:
        i_trigger_V1=i

      V1_response[i]=V1_response[i]+F_plus_V1*h_plus_data[i-i_trigger_V1]+F_cross_V1*h_cross_data[i-i_trigger_V1]



  L1_response_aligned=np.zeros(len(t_abs))
  H1_response_aligned=np.zeros(len(t_abs))
  V1_response_aligned=np.zeros(len(t_abs))
  L1_noise_aligned=np.zeros(len(t_abs))
  H1_noise_aligned=np.zeros(len(t_abs))
  V1_noise_aligned=np.zeros(len(t_abs))

  for i in range(len(t_abs)):
    L1_response_aligned[i]=L1_response[i+i_trigger_L1]
    H1_response_aligned[i]=H1_response[i+i_trigger_H1]
    V1_response_aligned[i]=V1_response[i+i_trigger_V1]
    L1_noise_aligned[i]=L1_noise_backup[i+i_trigger_L1]
    H1_noise_aligned[i]=H1_noise_backup[i+i_trigger_H1]
    V1_noise_aligned[i]=V1_noise_backup[i+i_trigger_V1]

  Signal=np.sum(L1_response_aligned**2)+np.sum(H1_response_aligned**2)+np.sum(V1_response_aligned**2)
  Noise=np.sum(L1_noise_aligned**2)+np.sum(H1_noise_aligned**2)+np.sum(V1_noise_aligned**2)
  SNR=Signal/Noise


  sigma_2_L1=0.
  sigma_2_H1=0.
  sigma_2_V1=0.

  for i in range(len(t_abs)):
    sigma_2_L1=sigma_2_L1+L1_noise_backup[i+i_trigger_L1]**2
    sigma_2_H1=sigma_2_H1+H1_noise_backup[i+i_trigger_H1]**2
    sigma_2_L1=sigma_2_V1+V1_noise_backup[i+i_trigger_V1]**2

  sigma_2_L1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_L1
  sigma_2_H1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_H1
  sigma_2_V1=((t_abs[1]-t_abs[0])/(t_max-t_min))*sigma_2_V1




  F_plus=np.array([F_plus_L1,F_plus_H1,F_plus_V1])
  F_cross=np.array([F_cross_L1,F_cross_H1,F_cross_V1])
  sigma_2=np.array([sigma_2_L1,sigma_2_H1,sigma_2_V1])

  N_detectors=3

  a_plus=np.zeros((N_detectors,N_detectors))

  for i in range(N_detectors):
    for j in range(N_detectors):
      if(i!=j):

        first_term=0.5*(sigma_2[i]*(F_cross[j]**2)+sigma_2[j]*(F_cross[i]**2))/((F_plus[i]*F_cross[j]-F_cross[i]*F_plus[j])**2)

        second_term=0.
```

```
            for l in range(N_detectors):
              for m in range(N_detectors):
                if(l!=m):
                  if(l!=i and m!=j):
                    second_term=second_term+((F_plus[l]*F_cross[m]-F_cross[l]*F_plus[m])**2)
    /(sigma_2[l]*(F_cross[m]**2)+sigma_2[m]*(F_cross[l]**2))

            a_plus[i][j]=1./(0.5+first_term*second_term)


      a_cross=np.zeros((N_detectors,N_detectors))

      for i in range(N_detectors):
        for j in range(N_detectors):
          if(i!=j):

            first_term=0.5*(sigma_2[i]*(F_plus[j]**2)+sigma_2[j]*(F_plus[i]**2))/((F_plus[i
    ]*F_cross[j]-F_cross[i]*F_plus[j])**2)

            second_term=0.
            for l in range(N_detectors):
              for m in range(N_detectors):
                if(l!=m):
                  if(l!=i and m!=j):
                    second_term=second_term+((F_plus[l]*F_cross[m]-F_cross[l]*F_plus[m])**2)
    /(sigma_2[l]*(F_plus[m]**2)+sigma_2[m]*(F_plus[l]**2))

            a_cross[i][j]=1./(0.5+first_term*second_term)


      h_plus_01=(F_cross[1]*L1_response_aligned-F_cross[0]*H1_response_aligned)/(F_plus[0]*
    F_cross[1]-F_cross[0]*F_plus[1])
      h_plus_02=(F_cross[2]*L1_response_aligned-F_cross[0]*V1_response_aligned)/(F_plus[0]*
    F_cross[2]-F_cross[0]*F_plus[2])
      h_plus_12=(F_cross[2]*H1_response_aligned-F_cross[1]*V1_response_aligned)/(F_plus[1]*
    F_cross[2]-F_cross[1]*F_plus[2])

      h_cross_01=(F_plus[1]*L1_response_aligned-F_plus[0]*H1_response_aligned)/(F_cross[0]*
    F_plus[1]-F_plus[0]*F_cross[1])
      h_cross_02=(F_plus[2]*L1_response_aligned-F_plus[0]*V1_response_aligned)/(F_cross[0]*
    F_plus[2]-F_plus[0]*F_cross[2])
      h_cross_12=(F_plus[2]*H1_response_aligned-F_plus[1]*V1_response_aligned)/(F_cross[1]*
    F_plus[2]-F_plus[1]*F_cross[2])



      h_plus=a_plus[0][1]*h_plus_01+a_plus[0][2]*h_plus_02+a_plus[1][2]*h_plus_12
      h_cross=a_cross[0][1]*h_cross_01+a_cross[0][2]*h_cross_02+a_cross[1][2]*h_cross_12


      overlap_h_plus[index]=pm.overlap(h_plus,h_plus_data)
      overlap_h_cross[index]=pm.overlap(h_cross,h_cross_data)



      SNR_total=np.append(SNR_total,SNR)
      overlap_h_plus_total=np.append(overlap_h_plus_total,overlap_h_plus[index])
      overlap_h_cross_total=np.append(overlap_h_cross_total,overlap_h_cross[index])

      count=count+1

    print((100*count/(len(scaling_factor)*theta_pixels*phi_pixels)),'%')


SNR_total=np.delete(SNR_total,0)
overlap_h_plus_total=np.delete(overlap_h_plus_total,0)
overlap_h_cross_total=np.delete(overlap_h_cross_total,0)

```

```
379
380 new_line='\n'
381 space=' '
382 comma=','
383 masses='bodies masses (in Solar masses)='+str(mass_1)+comma+str(mass_2)
384 overlap_description='Overlap for cWB wavelets denoising algorithm '
385 title=overlap_description+comma+new_line+masses
386
387 # here I make a scatterplot of the overlap versus the SNR
388 overlap_scatterplot, (h_plus, h_cross) = plt.subplots(1, 2)
389 overlap_scatterplot.suptitle(title)
390 h_plus.set(xlabel='SNR', ylabel='Overlap')
391 h_plus.set_title('$h_{+}$')
392 h_plus.scatter(SNR_total,overlap_h_plus_total)
393 h_plus.set_xscale('log')
394 h_plus.set_yscale('log')
395 h_cross.set(xlabel='SNR', ylabel='')
396 h_cross.set_title('$h_{x}$')
397 h_cross.scatter(SNR_total,overlap_h_cross_total)
398 h_cross.set_xscale('log')
399 h_cross.set_yscale('log')
400
401 save_to_file=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
402 text_name_scatterplot='overlap_Gursel_Tinto_'+mass_1_text+'__'+mass_2_text+'_scatterplot.txt
    '
403 np.savetxt(text_name_scatterplot, save_to_file)
404
405
406 # this is for reordering the arrays in order to have increasing values for the SNR
407 for i in range(len(SNR_total)):
408   swap = i + np.argmin(SNR_total[i:])
409   (SNR_total[i], SNR_total[swap]) = (SNR_total[swap], SNR_total[i])
410   (overlap_h_plus_total[i], overlap_h_plus_total[swap]) = (overlap_h_plus_total[swap],
        overlap_h_plus_total[i])
411   (overlap_h_cross_total[i], overlap_h_cross_total[swap]) = (overlap_h_cross_total[swap],
        overlap_h_cross_total[i])
412
413 # here I graph the overlap versus the SNR as a line
414 overlap_figure, (h_plus, h_cross) = plt.subplots(1, 2)
415 overlap_figure.suptitle(title)
416 h_plus.set(xlabel='SNR', ylabel='Overlap')
417 h_plus.set_title('$h_{+}$')
418 h_plus.plot(SNR_total,overlap_h_plus_total)
419 h_plus.set_xscale('log')
420 h_plus.set_yscale('log')
421 h_cross.set(xlabel='SNR', ylabel='')
422 h_cross.set_title('$h_{x}$')
423 h_cross.plot(SNR_total,overlap_h_cross_total)
424 h_cross.set_xscale('log')
425 h_cross.set_yscale('log')
426
427
428
429 save_to_file_line=np.column_stack((SNR_total,overlap_h_plus_total,overlap_h_cross_total))
430 text_name_line='overlap_Gursel_Tinto_'+mass_1_text+'__'+mass_2_text+'_line.txt'
431 np.savetxt(text_name_line, save_to_file_line)
432
433
434
435
436
437 plt.show()
```

overlap_comparison.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use( 'tkagg' )
plt.rcParams['backend'] = 'TkAgg'



print('This program graphs the overlaps calculated with the codes
    polarization_ALGORITHMNAME_overlap.py')


theta_pixels=3
phi_pixels=3

scale=1.4
start=-9
scaling_factor=np.array([(scale)**start])
len_scaling_factor=28

for i in range(1,len_scaling_factor):
  scaling_factor=np.append(scaling_factor,scale**(start+i))

overall_scaling_factor=scaling_factor
for i in range(1,theta_pixels*phi_pixels):
  overall_scaling_factor=np.append(overall_scaling_factor,scaling_factor)

scaling_factor=overall_scaling_factor


print('Which data file do you want to use? Press 1 for the BBH with masses 11.0 and 7.6 or
    another number for the BBH with masses 30.7 and 25.3')
boolean_0=int(input())

if(boolean_0==1):
  mass_1=11.0
  mass_2=7.6
  mass_1_text='11_0'
  mass_2_text='7_6'
  cWB_data=np.genfromtxt('overlap_cWB_11_0__7_6_scatterplot.txt',delimiter='\t')
  cWB_data_matrix= np.loadtxt('overlap_cWB_11_0__7_6_scatterplot.txt')
  SNR_total=cWB_data_matrix[:,0]
  overlap_h_plus_cWB=cWB_data_matrix[:,1]
  overlap_h_cross_cWB=cWB_data_matrix[:,2]

  cWB_wavelets_denoising_data=np.genfromtxt('
    overlap_cWB_wavelets_denoising_11_0__7_6_scatterplot.txt',delimiter='\t')
  cWB_wavelets_denoising_data_matrix= np.loadtxt('
    overlap_cWB_wavelets_denoising_11_0__7_6_scatterplot.txt')
  overlap_h_plus_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,1]
  overlap_h_cross_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,2]

  Gursel_Tinto_data=np.genfromtxt('overlap_Gursel_Tinto_11_0__7_6_scatterplot.txt',delimiter
    ='\t')
  Gursel_Tinto_data_matrix=np.loadtxt('overlap_Gursel_Tinto_11_0__7_6_scatterplot.txt')
  overlap_h_plus_Gursel_Tinto=Gursel_Tinto_data_matrix[:,1]
  overlap_h_cross_Gursel_Tinto=Gursel_Tinto_data_macWB_data=np.genfromtxt('
    overlap_cWB_11_0__7_6_scatterplot.txt',delimiter='\t')
  cWB_data_matrix= np.loadtxt('overlap_cWB_11_0__7_6_scatterplot.txt')
  SNR_total=cWB_data_matrix[:,0]
  overlap_h_plus_cWB=cWB_data_matrix[:,1]
  overlap_h_cross_cWB=cWB_data_matrix[:,2]

  cWB_wavelets_denoising_data=np.genfromtxt('
    overlap_cWB_wavelets_denoising_11_0__7_6_scatterplot.txt',delimiter='\t')
```

```
  cWB_wavelets_denoising_data_matrix= np.loadtxt('overlap_cWB_wavelets_denoising_11_0__7_6_scatterplot.txt')
  overlap_h_plus_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,1]
  overlap_h_cross_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,2]

  Gursel_Tinto_data=np.genfromtxt('overlap_Gursel_Tinto_11_0__7_6_scatterplot.txt',delimiter='\t')
  Gursel_Tinto_data_matrix=np.loadtxt('overlap_Gursel_Tinto_11_0__7_6_scatterplot.txt')
  overlap_h_plus_Gursel_Tinto=Gursel_Tinto_data_matrix[:,1]
  overlap_h_cross_Gursel_Tinto=Gursel_Tinto_data_matrix[:,2]

else:
  mass_1=30.7
  mass_2=25.3
  mass_1_text='30_7'
  mass_2_text='25_3'
  cWB_data=np.genfromtxt('overlap_cWB_30_7__25_3_scatterplot.txt',delimiter='\t')
  cWB_data_matrix= np.loadtxt('overlap_cWB_30_7__25_3_scatterplot.txt')
  SNR_total=cWB_data_matrix[:,0]
  overlap_h_plus_cWB=cWB_data_matrix[:,1]
  overlap_h_cross_cWB=cWB_data_matrix[:,2]

  cWB_wavelets_denoising_data=np.genfromtxt('overlap_cWB_wavelets_denoising_30_7__25_3_scatterplot.txt',delimiter='\t')
  cWB_wavelets_denoising_data_matrix= np.loadtxt('overlap_cWB_wavelets_denoising_30_7__25_3_scatterplot.txt')
  overlap_h_plus_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,1]
  overlap_h_cross_cWB_wavelets_denoising=cWB_wavelets_denoising_data_matrix[:,2]

  Gursel_Tinto_data=np.genfromtxt('overlap_Gursel_Tinto_30_7__25_3_scatterplot.txt',delimiter='\t')
  Gursel_Tinto_data_matrix=np.loadtxt('overlap_Gursel_Tinto_30_7__25_3_scatterplot.txt')
  overlap_h_plus_Gursel_Tinto=Gursel_Tinto_data_matrix[:,1]
  overlap_h_cross_Gursel_Tinto=Gursel_Tinto_data_matrix[:,2]



x_lim_left=3.e-2
x_lim_right=7.e2
y_lim_down=5.e-5
y_lim_up=1.2


new_line='\n'
space=' '
comma=','
masses='bodies masses (in Solar masses)='+str(mass_1)+comma+space+str(mass_2)


overlap_scatterplot_figure, (h_plus_scatterplot, h_cross_scatterplot) = plt.subplots(1, 2)
overlap_scatterplot_figure.suptitle('Overlap')
h_plus_scatterplot.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_plus_scatterplot.set_title('$h_{+}$')
h_plus_scatterplot.set_xscale('log')
h_plus_scatterplot.set_yscale('log')
h_plus_scatterplot.scatter(scaling_factor,overlap_h_plus_cWB,label='cWB')
h_plus_scatterplot.scatter(scaling_factor,overlap_h_plus_cWB_wavelets_denoising,label='cWB wavelets denoising')
h_plus_scatterplot.scatter(scaling_factor,overlap_h_plus_Gursel_Tinto,label='Gursel & Tinto')
h_plus_scatterplot.set_xlim([x_lim_left,x_lim_right])
h_plus_scatterplot.set_ylim([y_lim_down,y_lim_up])
h_plus_scatterplot.legend()
h_plus_scatterplot.grid()
h_cross_scatterplot.set(xlabel='Noise scaling factor', ylabel='')
h_cross_scatterplot.set_title('$h_{x}$')
h_cross_scatterplot.set_xscale('log')
h_cross_scatterplot.set_yscale('log')
```

```
h_cross_scatterplot.scatter(scaling_factor,overlap_h_cross_cWB,label='cWB')
h_cross_scatterplot.scatter(scaling_factor,overlap_h_cross_cWB_wavelets_denoising,label='cWB wavelets denoising')
h_cross_scatterplot.scatter(scaling_factor,overlap_h_cross_Gursel_Tinto,label='Gursel & Tinto')
h_cross_scatterplot.set_xlim([x_lim_left,x_lim_right])
h_cross_scatterplot.set_ylim([y_lim_down,y_lim_up])
h_cross_scatterplot.legend()
h_cross_scatterplot.grid()


array_dimension=len_scaling_factor
scaling_factor_single_array=np.zeros(array_dimension)
mean_cWB_h_plus=np.zeros(array_dimension)
mean_cWB_wavelets_denoising_h_plus=np.zeros(array_dimension)
mean_Gursel_Tinto_h_plus=np.zeros(array_dimension)
mean_cWB_h_cross=np.zeros(array_dimension)
mean_cWB_wavelets_denoising_h_cross=np.zeros(array_dimension)
mean_Gursel_Tinto_h_cross=np.zeros(array_dimension)


total_pixels=theta_pixels*phi_pixels

# here I calculate the mean overlap value for each noise scaling factor
for i in range(array_dimension):

  scaling_factor_single_array[i]=scaling_factor[i]


  for j in range(total_pixels):

    mean_cWB_h_plus[i]=mean_cWB_h_plus[i]+overlap_h_plus_cWB[i+j*array_dimension]
    mean_cWB_h_cross[i]=mean_cWB_h_cross[i]+overlap_h_cross_cWB[i+j*array_dimension]
    mean_cWB_wavelets_denoising_h_plus[i]=mean_cWB_wavelets_denoising_h_plus[i]+overlap_h_plus_cWB_wavelets_denoising[i+j*array_dimension]
    mean_cWB_wavelets_denoising_h_cross[i]=mean_cWB_wavelets_denoising_h_cross[i]+overlap_h_cross_cWB_wavelets_denoising[i+j*array_dimension]
    mean_Gursel_Tinto_h_plus[i]=mean_Gursel_Tinto_h_plus[i]+overlap_h_plus_Gursel_Tinto[i+j*array_dimension]
    mean_Gursel_Tinto_h_cross[i]=mean_Gursel_Tinto_h_cross[i]+overlap_h_cross_Gursel_Tinto[i+j*array_dimension]

  # the absolute value is for representing the overlaps in log log scale
  mean_cWB_h_plus[i]=abs(mean_cWB_h_plus[i])/total_pixels
  mean_cWB_h_cross[i]=abs(mean_cWB_h_cross[i])/total_pixels
  mean_cWB_wavelets_denoising_h_plus[i]=abs(mean_cWB_wavelets_denoising_h_plus[i])/total_pixels
  mean_cWB_wavelets_denoising_h_cross[i]=abs(mean_cWB_wavelets_denoising_h_cross[i])/total_pixels
  mean_Gursel_Tinto_h_plus[i]=abs(mean_Gursel_Tinto_h_plus[i])/total_pixels
  mean_Gursel_Tinto_h_cross[i]=abs(mean_Gursel_Tinto_h_cross[i])/total_pixels




overlap_figure_title='Mean overlap comparison'+comma+new_line+masses

overlap_figure, (h_plus, h_cross) = plt.subplots(1, 2)
overlap_figure.suptitle(overlap_figure_title)
h_plus.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_plus.set_title('$h_{+}$')
h_plus.loglog(scaling_factor_single_array,mean_cWB_h_plus,label='cWB')
h_plus.loglog(scaling_factor_single_array,mean_cWB_wavelets_denoising_h_plus,label='cWB wavelets denoising')
h_plus.loglog(scaling_factor_single_array,mean_Gursel_Tinto_h_plus,label='Gursel & Tinto')
h_plus.set_xlim([x_lim_left,x_lim_right])
h_plus.set_ylim([y_lim_down,y_lim_up])
h_plus.legend()
```

```
h_plus.grid()
h_cross.set(xlabel='Noise scaling factor', ylabel='')
h_cross.set_title('$h_{x}$')
h_cross.loglog(scaling_factor_single_array,mean_cWB_h_cross,label='cWB')
h_cross.loglog(scaling_factor_single_array,mean_cWB_wavelets_denoising_h_cross,label='cWB wavelets denoising')
h_cross.loglog(scaling_factor_single_array,mean_Gursel_Tinto_h_cross,label='Gursel & Tinto')
h_cross.set_xlim([x_lim_left,x_lim_right])
h_cross.set_ylim([y_lim_down,y_lim_up])
h_cross.legend()
h_cross.grid()


overlap_scatterplot_cWB_figure_title='Overlap for cWB'+comma+new_line+masses

overlap_scatterplot_cWB_figure, (h_plus_scatterplot_cWB, h_cross_scatterplot_cWB) = plt.subplots(1, 2)
overlap_scatterplot_cWB_figure.suptitle(overlap_scatterplot_cWB_figure_title)
h_plus_scatterplot_cWB.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_plus_scatterplot_cWB.set_title('$h_{+}$')
h_plus_scatterplot_cWB.set_xscale('log')
h_plus_scatterplot_cWB.set_yscale('log')
h_plus_scatterplot_cWB.scatter(scaling_factor,overlap_h_plus_cWB,c='b')
h_plus_scatterplot_cWB.set_xlim([x_lim_left,x_lim_right])
h_plus_scatterplot_cWB.set_ylim([y_lim_down,y_lim_up])
h_plus_scatterplot_cWB.grid()
h_cross_scatterplot_cWB.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_cross_scatterplot_cWB.set_title('$h_{x}$')
h_cross_scatterplot_cWB.set_xscale('log')
h_cross_scatterplot_cWB.set_yscale('log')
h_cross_scatterplot_cWB.scatter(scaling_factor,overlap_h_cross_cWB,c='b')
h_cross_scatterplot_cWB.set_xlim([x_lim_left,x_lim_right])
h_cross_scatterplot_cWB.set_ylim([y_lim_down,y_lim_up])
h_cross_scatterplot_cWB.grid()

overlap_scatterplot_cWB_wavelets_denoising_figure_title='Overlap for cWB wavelets denoising'+comma+new_line+masses

overlap_scatterplot_cWB_wavelets_denoising_figure, (h_plus_scatterplot_cWB_wavelets_denoising, h_cross_scatterplot_cWB_wavelets_denoising) = plt.subplots(1, 2)
overlap_scatterplot_cWB_wavelets_denoising_figure.suptitle(overlap_scatterplot_cWB_wavelets_denoising_figure_title)
h_plus_scatterplot_cWB_wavelets_denoising.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_plus_scatterplot_cWB_wavelets_denoising.set_title('$h_{+}$')
h_plus_scatterplot_cWB_wavelets_denoising.set_xscale('log')
h_plus_scatterplot_cWB_wavelets_denoising.set_yscale('log')
h_plus_scatterplot_cWB_wavelets_denoising.scatter(scaling_factor,overlap_h_plus_cWB_wavelets_denoising,c='orange')
h_plus_scatterplot_cWB_wavelets_denoising.set_xlim([x_lim_left,x_lim_right])
h_plus_scatterplot_cWB_wavelets_denoising.set_ylim([y_lim_down,y_lim_up])
h_plus_scatterplot_cWB_wavelets_denoising.grid()
h_cross_scatterplot_cWB_wavelets_denoising.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_cross_scatterplot_cWB_wavelets_denoising.set_title('$h_{x}$')
h_cross_scatterplot_cWB_wavelets_denoising.set_xscale('log')
h_cross_scatterplot_cWB_wavelets_denoising.set_yscale('log')
h_cross_scatterplot_cWB_wavelets_denoising.scatter(scaling_factor,overlap_h_cross_cWB_wavelets_denoising,c='orange')
h_cross_scatterplot_cWB_wavelets_denoising.set_xlim([x_lim_left,x_lim_right])
h_cross_scatterplot_cWB_wavelets_denoising.set_ylim([y_lim_down,y_lim_up])
h_cross_scatterplot_cWB_wavelets_denoising.grid()


overlap_scatterplot_Gursel_Tinto_figure_title='Overlap for Gursel & Tinto'+comma+new_line+masses

```

```
overlap_scatterplot_Gursel_Tinto_figure, (h_plus_scatterplot_Gursel_Tinto,
    h_cross_scatterplot_Gursel_Tinto) = plt.subplots(1, 2)
overlap_scatterplot_Gursel_Tinto_figure.suptitle(
    overlap_scatterplot_Gursel_Tinto_figure_title)
h_plus_scatterplot_Gursel_Tinto.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_plus_scatterplot_Gursel_Tinto.set_title('$h_{+}$')
h_plus_scatterplot_Gursel_Tinto.set_xscale('log')
h_plus_scatterplot_Gursel_Tinto.set_yscale('log')
h_plus_scatterplot_Gursel_Tinto.scatter(scaling_factor,overlap_h_plus_Gursel_Tinto,c='green'
    )
h_plus_scatterplot_Gursel_Tinto.set_xlim([x_lim_left,x_lim_right])
h_plus_scatterplot_Gursel_Tinto.set_ylim([y_lim_down,y_lim_up])
h_plus_scatterplot_Gursel_Tinto.grid()
h_cross_scatterplot_Gursel_Tinto.set(xlabel='Noise scaling factor', ylabel='Overlap')
h_cross_scatterplot_Gursel_Tinto.set_title('$h_{x}$')
h_cross_scatterplot_Gursel_Tinto.set_xscale('log')
h_cross_scatterplot_Gursel_Tinto.set_yscale('log')
h_cross_scatterplot_Gursel_Tinto.scatter(scaling_factor,overlap_h_cross_Gursel_Tinto,c='
    green')
h_cross_scatterplot_Gursel_Tinto.set_xlim([x_lim_left,x_lim_right])
h_cross_scatterplot_Gursel_Tinto.set_ylim([y_lim_down,y_lim_up])
h_cross_scatterplot_Gursel_Tinto.grid()

plt.show()
```

Bibliography

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

[2] Michele Maggiore. *Gravitational waves: Volume 1: Theory and experiments*. Vol. 1. Oxford university press, 2008.

[3] Bernard Schutz. *A first course in general relativity*. Cambridge university press, 2009.

[4] BP Abbott et al. "Tests of general relativity with the binary black hole signals from the LIGO-Virgo catalog GWTC-1". In: *Physical Review D* 100.10 (2019), p. 104036.

[5] Giles Hammond, Stefan Hild, and Matthew Pitkin. "Advanced technologies for future laser-interferometric gravitational wave detectors". In: *Journal of Modern Optics* 61 (Feb. 2014). DOI: 10.1080/09500340.2014.920934.

[6] Marco Drago. "Search for transient gravitational wave signals with unknown waveform in the LIGO-Virgo network of interferometric detectos using a fully coherent algorithm". PhD thesis. Università di Padova, 2010.

[7] BP Abbott et al. "GWTC-1: a gravitational-wave transient catalog of compact binary mergers observed by LIGO and Virgo during the first and second observing runs". In: *Physical Review X* 9.3 (2019), p. 031040.

[8] LIGO. *LIGO detections*. [Online; accessed 15-July-2020]. 2020. URL: https://www.ligo.org/detections.php.

[9] Benjamin P Abbott et al. "GW170817: observation of gravitational waves from a binary neutron star inspiral". In: *Physical Review Letters* 119.16 (2017), p. 161101.

[10] Wikipedia contributors. *List of gravitational wave observations — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-October-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=List_of_gravitational_wave_observations&oldid=965545398.

[11] Viktoriya Morozova et al. "The Gravitational Wave Signal from Core-collapse Supernovae". In: *The Astrophysical Journal* 861.1 (2018), p. 10.

[12] BP Abbott et al. "Optically targeted search for gravitational waves emitted by core-collapse supernovae during the first and second observing runs of advanced LIGO and advanced Virgo". In: *Physical Review D* 101.8 (2020), p. 084002.

[13] Maurice HPM van Putten et al. "Prospects for multi-messenger extended emission from core-collapse supernovae in the Local Universe". In: *The European Physical Journal Plus* 134.10 (2019), p. 537.

[14] Nelson Christensen. "Stochastic gravitational wave backgrounds". In: *Reports on Progress in Physics* 82.1 (2018), p. 016903.

[15] LIGO Scientific et al. "The basic physics of the binary black hole merger GW150914". In: *Annalen der Physik* 529.1-2 (2017), p. 1600209.

[16] M Drago et al. "Coherent WaveBurst, a pipeline for unmodeled gravitational-wave data analysis". In: *arXiv preprint arXiv:2006.12604* (2020).

[17] R Abbott et al. "Properties and astrophysical implications of the 150 $M_{\odot}$ binary black hole merger GW190521". In: *The Astrophysical Journal Letters* 900.1 (2020), p. L13.

[18] MJ Graham et al. "Candidate electromagnetic counterpart to the binary black hole merger gravitational-wave event s190521g". In: *Physical review letters* 124.25 (2020), p. 251102.

[19] *Nobel Prizes 2017*. `https://www.nobelprize.org/list-of-2017-nobel-laureates/`. Accessed 1 August 2020.

[20] Wikipedia contributors. *LIGO — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-August-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=LIGO&oldid=970248038`.

[21] Wikipedia contributors. *Virgo interferometer — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-August-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Virgo_interferometer&oldid=963631686`.

[22] Benjamin P Abbott et al. "GW170814: a three-detector observation of gravitational waves from a binary black hole coalescence". In: *Physical review letters* 119.14 (2017), p. 141101.

[23] *What Next for Gravitational Wave Detection?* `https://www.aps.org/publications/apsnews/201906/wave.cfm`. Accessed 1 August 2020.

[24] Benjamin P Abbott et al. "Prospects for observing and localizing gravitational-wave transients with Advanced LIGO, Advanced Virgo and KAGRA". In: *Living Reviews in Relativity* 21.1 (2018), p. 3.

[25] Archana Pai, Eric Chassande-Mottin, and Olivier Rabaste. "Best network chirplet chain: Near-optimal coherent detection of unmodeled gravitational wave chirps with a network of detectors". In: *Physical Review D* 77.6 (2008), p. 062005.

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

[27] A Buikema et al. "Sensitivity and performance of the Advanced LIGO detectors in the third observing run". In: *Physical Review D* 102.6 (2020), p. 062003.

[28] F Acernese et al. "Quantum backaction on kg-scale mirrors: Observation of radiation pressure noise in the Advanced Virgo detector". In: *Physical Review Letters* 125.13 (2020), p. 131101.

[29] *LIGO's Interferometer*. `https://www.ligo.caltech.edu/page/ligos-ifo`. Accessed 1 October 2020.

[30] Denis V Martynov et al. "Sensitivity of the Advanced LIGO detectors at the beginning of gravitational wave astronomy". In: *Physical Review D* 93.11 (2016), p. 112004.

[31] Marzia Colombini. "Thermal noise issue in the monolithic suspensions of the Virgo+ gravitational wave interferometer". In: (2013).

[32] Yu Levin. "Internal thermal noise in the LIGO test masses: A direct approach". In: *Physical Review D* 57.2 (1998), p. 659.

[33] Haocun Yu et al. "Quantum correlations between the light and kilogram-mass mirrors of LIGO". In: *arXiv preprint arXiv:2002.01519* (2020).

[34] Sheila Dwyer. "Squeezing quantum noise". In: *Physics Today* 67.11 (2014).

[35] Yekta Gürsel and Massimo Tinto. "Near optimal solution to the inverse problem for gravitational-wave bursts". In: *Physical Review D* 40.12 (1989), p. 3884.

[36] Shourov Chatterji et al. "Coherent network analysis technique for discriminating gravitational-wave bursts from instrumental noise". In: *Physical Review D* 74.8 (2006), p. 082005.

[37] Irene Di Palma and Marco Drago. "Estimation of the gravitational wave polarizations from a nontemplate search". In: *Physical Review D* 97.2 (2018), p. 023011.

[38] S Klimenko et al. "Method for detection and reconstruction of gravitational wave transients with networks of advanced detectors". In: *Physical Review D* 93.4 (2016), p. 042004.

[39] Hans-Thomas Janka, Tobias Melson, and Alexander Summa. "Physics of core-collapse supernovae in three dimensions: a sneak preview". In: *Annual Review of Nuclear and Particle Science* 66 (2016), pp. 341–375.

[40] Sergey Klimenko et al. "A coherent method for detection of gravitational wave bursts". In: *Classical and Quantum Gravity* 25.11 (2008), p. 114029.

[41] V Necula, S Klimenko, and G Mitselmakher. "Transient analysis with fast Wilson-Daubechies time-frequency transform". In: *Journal of Physics: Conference Series*. Vol. 363. 1. IOP Publishing. 2012, p. 012032.

[42] Alex Nitz et al. *gwastro/pycbc: PyCBC release v1.16.9*. Version v1.16.9. Aug. 2020. DOI: `10.5281/zenodo.3993665`. URL: `https://doi.org/10.5281/zenodo.3993665`.

[43] Alejandro Bohé et al. "Improved effective-one-body model of spinning, nonprecessing binary black holes for the era of gravitational-wave astrophysics with advanced detectors". In: *Physical Review D* 95.4 (2017), p. 044028.

[44] J Timmer and M Koenig. "On generating power law noise." In: *Astronomy and Astrophysics* 300 (1995), p. 707.

[45] Olivier Rioul and Martin Vetterli. "Wavelets and signal processing". In: *IEEE signal processing magazine* 8.4 (1991), pp. 14–38.

[46] Gregory 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.

[47] LIGO Scientific Collaboration. *Gravitational Wave Open Science Center*. https://www.gw-openscience.org/yellow_box/. Accessed 8 August 2020.