Università degli Studi di Padova

SCUOLA GALILEIANA DI STUDI SUPERIORI

Università degli Studi di Padova

Scuola Galileiana di Studi Superiori

Classe di Scienze Naturali

Tesi di Diploma Galileiano

Best practices in scientific coding: a case study

Relatore

Prof. Luca Heltai

Diplomando

Jacopo Tissino

Abstract

Scientific software is often experimental and thus messy, undocumented, not reusable. As a given piece of software grows in user count, though, it can benefit in applying development best practices which are well-known in the industry setting.
This thesis will discuss a selection of these practices and show their application for a `python` project named `GWFish`, a Fisher matrix code used for gravitational wave data analysis.

Table of contents

Introduction

It is an opinion held by many that "scientists write bad code"; as evidence for this claim I will provide the large number of positive votes on this academia StackExchange post titled "Why do many talented scientists write horrible software?".

This is, however, a thorny statement which should be qualified: code may be "bad" according to the standards of software companies, but it may serve the purpose it is written to accomplish perfectly well, especially if this purpose is to act as a proof of concept, or as an exploratory step. However, the *industry best practices* that are missing from scientific code do have a reason to exist, and that reason often becomes apparent when projects become large enough, and/or used by enough people. Ignoring some of them is acceptable, and perhaps even advisable, for small projects, but for larger ones they start to matter more and more, and code not using them starts to accumulate *technical debt*, becoming difficult to read, maintain, modify, extend.

Resources on these best practices are plentiful, but are often focused on giving "industry" examples, which may not be too relevant to the scientific setting. When making an attempt to implement them in my own code, it was always a mental strain to "map" them to the kinds of issues I was interested in.

This work is an attempt to ease this process for others, providing some examples of the practical application of these concepts in a practical, scientific context. Using `GWFish` (Harms et al. 2022) to this end is something of a perfect storm. It is a young piece of software (its development started in earnest in early 2021, although the ideas it implements are quite a bit older) which serves a conceptually simple purpose, and which has recently started to be useful to a large amount of people.

Because of this, it is worth spending time on refactoring it, and adopting some development best practices. As of the writing of this thesis this process is still underway, therefore unfortunately I will not be able to discuss as many completed modifications as I would like.

Also, `GWFish` is written in `python`, which is also the programming language I know best. All discussions in this work will be limited to this language, and although some may apply for others as well I cannot guarantee this. The discussions will feature several suggestions of `python`-specific tools, and will assume a degree of familiarity with the language and its more common scientific packages.

All the code snippets discussed in this thesis, as well as the source code for this document, are provided in full in the repository github.com/jacopok/clean-coding-thesis.

`GWFish` in short

What follows is a short explanation of what this piece of software does, aimed at non-physicists. I will first introduce the concept of gravitational wave detection and the purpose of `GWFish` at a general level, without the mathematical details. Then, I will go into more detail into the things it needs to compute, including some math, but still at a level which an undergraduate in a scientific field should be able to understand.

The first direct measurement of gravitational waves was accomplished in 2015 by the LIGO interferometers in the United States (LIGO Scientific Collaboration and Virgo Collaboration et al. 2016). In 2017, they were joined by the Virgo interferometer in Italy, and the network has since detected almost 100 distinct signals. Most of these signals were generated by pairs of black holes orbiting each other, with the remaining few corresponding to pairs of neutron stars or one neutron star and one black hole.

By lowering the noise in the interferometer, we went from no detection to about one signal per week during detector operation. This has already been called the birth of a new era of *gravitational wave astronomy*, which can complement "regular" electromagnetic astronomy.

A lot of interest is going towards the question: how can we do better? Which kinds of new detectors are best if we want to detect even more gravitational waves? These instruments are big, expensive projects, therefore a careful scientific evaluation of what we think a new detector concept might be able to accomplish is crucial when outlining a funding proposal.

The basic questions we wish to answer for any new detector concept are:

- Which kinds of gravitational wave sources will it be able to detect? How many of them?
- How well will it be able to estimate the properties of these sources?

The answers depend on two basic aspects:

- Given a specific astrophysical source, can *new planned detector X* measure it? If so, how well?
  - For example, consider a pair of black holes, with masses so-and-so, at a distance of such-and-such…
- How many of that astrophysical source kind are there? How far away are they?
  - For example, what is the distribution of black hole binaries? What are their typical masses, how often do they occur in any given universe volume?

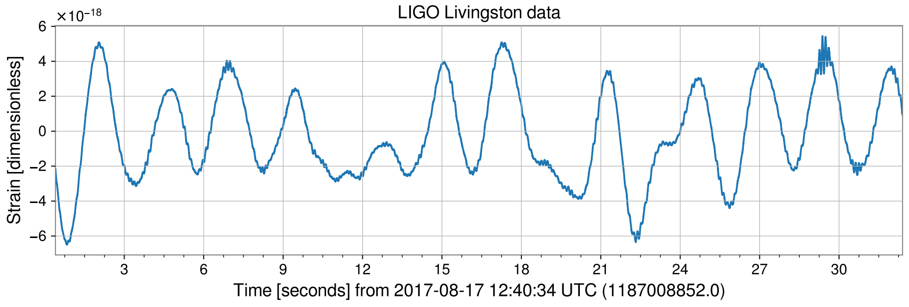
`GWFish` is a piece of software built for the purpose of giving an approximate answer to the first of these questions, while it relies on other tools to answer the second. Typically, the workflow consists in the generation of a theoretical *population* of, for example, binaries of black holes, which will contain several thousands of them, listing for each their masses, distances, spin and so on. This will then be fed into a tool like `GWFish` to get detection statistics.

Matched filtering

This section and the following will go in some detail about the mathematics of the approximation used in order to quickly get the required detection statistics.

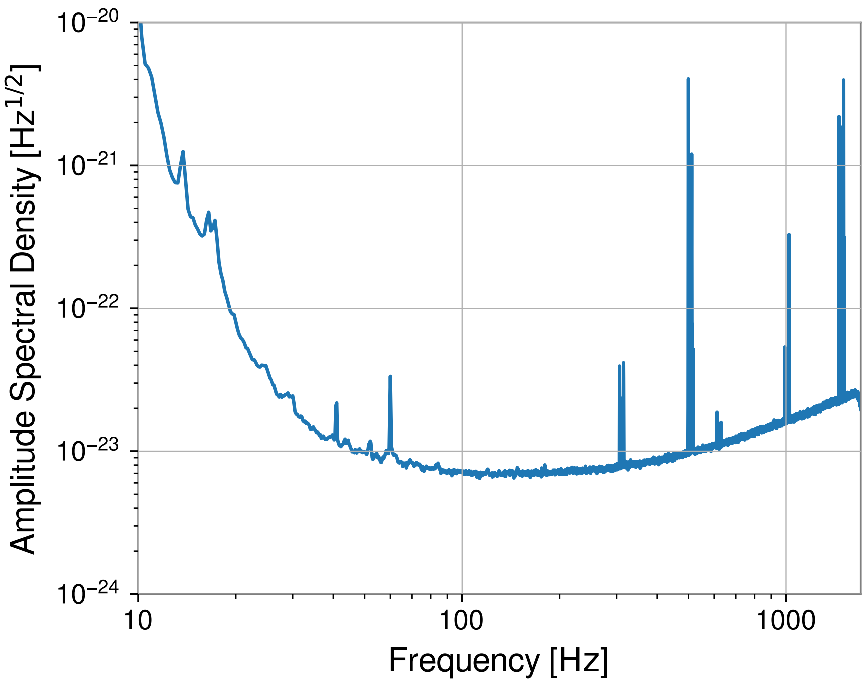
Our starting point is matched filtering, a fundamental technique used for all modern gravitational data analysis. The idea is that we want to extract a very weak signal which is submerged in noise. A complete overview of the method can be found in (Maggiore 2007), here I will give a very brief one.

Our detectors are measuring *strain*, which roughly speaking is the fluctuation in the length of the detector arms normalized to the arm length itself, $h(t) = \Delta L(t)/L$. Gravitational waves distort space itself, and we can therefore measure them by looking at this quantity. The raw strain data from a GW detector typically looks like this:



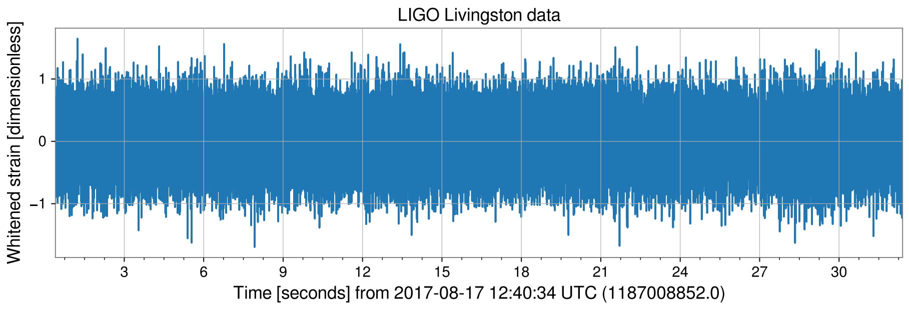
It is clear that the largest contribution in terms of amplitude is an oscillation with period on the order of a couple of seconds, and amplitude on the order of a few times 10^{-18}.

The signal we want to measure, unfortunately, is at least three orders of magnitude smaller: $h \sim 10^{-21}$! What can we do? The first step is to look at the "Fourier spectrum" of these data (more specifically, the amplitude spectral density, but thinking of it as the amplitude of the Fourier transform is not terrible):

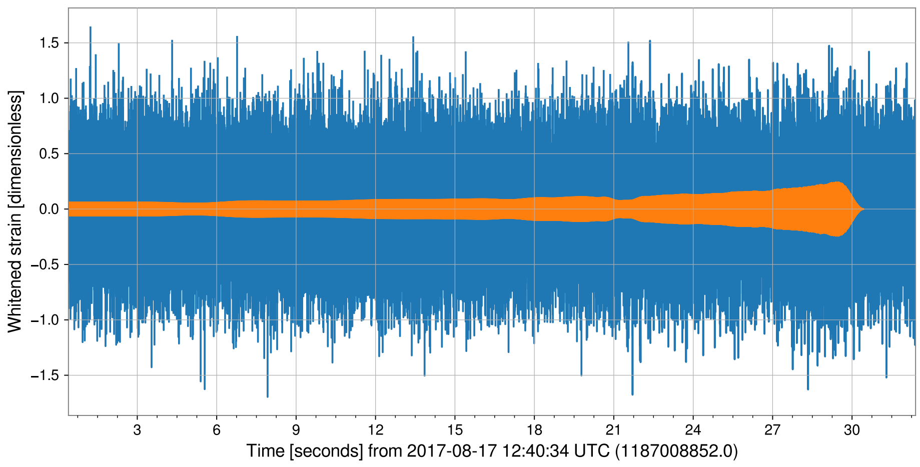


We are not even showing the part of the spectrum with period on the order a few seconds, the trend at low frequency continues and the amplitude for $f \sim 1\,\mathrm{Hz}$ there is enormous. This detector is *not sensitive* to signals with very low frequency, but it *is* sensitive to signals with frequencies in a band around 100Hz.

The first step towards actually measuring something lies in *whitening* the measured data, that is, dividing every Fourier component by its root-mean-square value. After this procedure, our data looks like this:



Still, no signal is visible! This is because the signal present in these data has high frequency and low amplitude; specifically, if we were to plot it subjected to the same procedure as the data, it would look like the orange curve:



So, how can we possibly detect something that is so far smaller in amplitude than our data?

The trick lies in a technique called *matched filtering*. Suppose our measured data is $d(t) = n(t) + s(t)$, with $n(t)$ being the noise and $s(t)$ being the astrophysical signal (assumed here to be monochromatic for simplicity), then we can look at a temporal integral in the form

$$I = \frac{1}{T}\int d(t)s(t)\mathrm{d}t = \underbrace{\frac{1}{T}\int n(t)s(t)\mathrm{d}t}_{I_n} + \underbrace{\frac{1}{T}\int s(t)s(t)\mathrm{d}t}_{I_s},$$

where T is the length of the integration period. There are two components: I_s is the average square magnitude of the signal, and it approaches a constant; on the other hand, I_n varies stochastically with n, but since n and s are not correlated the integral will be stochastic process with variance T (and therefore standard deviation $\sqrt{T}$); due to the division by T this term will then decay like $I_n \sim T^{-1/2}$. If we can observe the signal for long enough, then, the I_s term will dominate.

This is the essence of gravitational data analysis: if we know the expected signal ahead of time, we may extract its contribution from noisy data. The integrals above are optimal if the case of white noise, but going back to the actually-measured data the procedure is slightly more complicated.

Let us now jump ahead to the final result: first, we need to estimate the noise power spectral density (that is, the noise power per frequency bin) $S_n(f)$, and use to define with it the scalar product between timeseries $d(t)$ and $h(t)$ in terms of their Fourier transforms $\tilde{d}(f)$ and $\tilde{h}(f)$ as follows:

$$(d|h) = 4\Re\int_0^\infty \frac{\tilde{d}(f)\tilde{h}(f)}{S_n(f)}\mathrm{d}f$$

This product is the basis for signal searches, which are performed by looking for peaks in the following function of t:

$$(\text{observed strain data } d|h, \text{ theoretical signal, shifted by a time } t)$$

for a selection ("template bank") of plausible signals h we might see.

Similarly, parameter estimation for any observed signal d is performed by exploring the posterior distribution defined by the likelihood

$$\mathcal{L}(d|\theta) = \mathcal{N}\exp\left(-\frac{1}{2}(d-h(\theta)|d-h(\theta))\right)$$

where $\mathcal{N}$ is a constant normalization factor.

Fisher matrix error estimation

Up to now we discussed the analysis of current data; the question `GWFish` seeks to answer, on the other hand, pertains to data taken by detectors we have not built yet. We can, however, make estimates as to what their noise level will be and go from there.

A typical question it answers could be posed as:

> Given the estimated noise curve of the planned Einstein Telescope gravitational wave interferometer, suppose that two black holes with masses $M_1 = M_2 = 30 M_\odot$ (solar masses) merge at a distance of 10^9 light years from Earth.[1] How well could we measure their masses, distance, and position from the gravitational wave data?

The "proper" way to answer this question would be to simulate noise distributed according to the given noise curve, add the known signal to it, and analyze it as if it were real data. This can be done, and it *is* done to a certain extent, but it is very expensive: a single analysis of this kind takes several hours to days. This is what is done for the data of current detectors, where we know it to be worth it since it's *real* astrophysical data).

This prevents us from exploring things such as the dependence of the results on things such as the masses of the black holes, the distance and so on, as that requires re-doing the aforementioned analysis several (thousands of) times. The solution (or at least a partial one) is to make an approximation, called the *Fisher matrix approximation*: basically, we take the aforementioned likelihood, and approximate it as a multivariate Gaussian in the parameters θ, with mean given by the values we selected. Then, we may compute its covariance matrix by looking at the (negative expectation value of the) Hessian of $\log \mathcal{L}$ computed at that maximum-likelihood point, which is called the Fisher matrix:

$$\mathcal{F}_{ij} = -\mathbb{E}\left(\frac{\partial}{\partial \theta_i} \frac{\partial}{\partial \theta_j} \log \mathcal{L}(d|\theta) \right) = \mathbb{E} \frac{\partial}{\partial \theta_i} \frac{\partial}{\partial \theta_j} \left(\frac{1}{2} (d - h(\theta) | d - h(\theta)) \right)$$

Taking the expectation value amounts to looking at the case in which the noise equals zero, i.e. $d - h(\theta) = 0$; going through the derivatives we find that the only non-vanishing contribution is $\mathcal{F}_{ij} = (\partial_i h | \partial_j h)$, again evaluated at the maximum likelihood point.

This is the basic quantity `GWFish` is computing; we are approximating the likelihood as a Gaussian in the form $\log \mathcal{L} \sim -\Delta\theta_i \mathcal{F}_{ij} \Delta\theta_j / 2$ (with the Einstein summation convention), where $\Delta\theta = \theta - \overline{\theta}$ is the deviation of the parameter vector θ from its mean value $\overline{\theta}$. We may then use the properties of multivariate Gaussians, and state that our estimate for the variance of parameter i is given in terms of the diagonal components of the inverse of $\mathcal{F}$:

$$\sigma_i^2 \approx (\mathcal{F}^{-1})_{ii} \,.$$

This is a frequentist phrasing; alternatively, it is equivalent to a Bayesian one if we take a flat prior on the parameters, $p(\theta) = \text{const}$. This is surely not ultimately correct (for example, a flat prior on angular variables is not flat on the sphere), but the Fisher matrix approximation is ultimately quite rough itself, therefore including non-flat priors is likely not the primary concern.

Versioning

The first step towards successfully managing a software project is *version control*: having a proper system to track changes to the code made by many people.

[1]The problem as stated is under-specified, there are several other parameters to consider, but let us keep it simple here.

The most popular system for this purpose is by far and away `git`, which was developed by Linus Torvalds in 2005. It is a powerful and fast distributed version control system.

Typically, `git` is used in conjunction with a hosting server in which to store the software project: common choices include `github`, `gitlab`, `bitbucket`. `GWFish` is hosted on `github`, at the url github.com/janosch314/GWFish.

I will not introduce the basics of how to use `git` here, since there are many excellent resources for that; the discussion here is targeted at someone who is familiar with how to e.g. make commits, push to a remote repository and pull from it.

Recently, the group developing `GWFish` has started adopting a formal **process** for the development of new features, which quite closely matches Github flow. This is not the only possible way of handling a collaborative workflow, but it is a rather simple and effective one.

The idea is as follows: there is a `main` branch, which is expected to include finished code features and a usable version of the code. If I want to add a new feature, I will make a *feature branch* off the main one. The syntax is as follows:

```
git branch my-new-feature
git checkout my-new-feature
```

I can then work on this feature, making commits and pushing them to the repository. This way, my work can be public while not interfering with the main branch. While I will not voluntarily commit anything "wrong", building a new feature is necessarily experimental, and the branch is a safe place to possibly make mistakes.

When the feature seems to be good to go, I can then open a Pull Request (PR); the name is somewhat unfortunate since it's not in the perspective of the one making it. `gitlab` calls them "merge requests", which is more descriptive: I am asking for my code in the branch to be merged into the main one. Regardless of the name, the concept is rather simple: a PR is a structured process with the end goal of integrating code from a feature branch into the main one. This could also be accomplished with a simple git command (`git merge`), but the idea is that code from a feature branch should be evaluated and discussed.

As an example I will provide a pull request I made not for `GWFish` but for `pycbc`, an older and bigger piece of software for gravitational wave data analysis. There is no need to discuss the specifics of that PR, the point here is the *structure* of what was happening: I made a feature branch, proposing a new feature. I had previously outlined in an issue the reason why this feature was needed — a common pattern is to start with an issue and to solve it with a PR.

A maintainer of the project reviewed the PR; technical issues and slight modifications were discussed and implemented until a version of the feature which was satisfactory to everyone was reached. To this end, the automated testing pipeline was quite useful: for each commit I made to the branch, the pipeline could run to check that I had not accidentally created a *regression*, i.e. broken some previously-working functionality.

Semantic versioning

Besides making it easier for developers to track their work, versioning is useful in order for users to be able to understand how and when the software they are using is changing, whether they should upgrade, whether the functionality they are using will be broken or discontinued.

The full extent of this information should be specified in long-form in a changelog, discussed in the next section, but a simple shorthand for it may be given by a *standardized* versioning system such as semantic versioning. It is a rather simple specification, with all versions[2] looking like X.Y.Z with integer X, Y and Z. An alternative, which is recommended for large projects, is calendar-based versioning.

[2]All full releases; prereleases are allowed to be in the form e.g. X.Y.Z-alpha.

The advantage of using a standardized (and very widespread) system is that the meaning may be clear without requiring explanation. If it is not, or a user wants to know more, having a changelog is useful.

Changelogs

A changelog should be human-readable, and communicate in short to a user what changed from a version to another. As with versions, it is good for it to follow some standard, and a good one may be found at keep a changelog.

If we write good git commit messages it may be tempting to use them to automatically construct a changelog. This is a bad idea! They contain way too much information compared to what the user needs to know about the new version.

Is it worth it to use semantic versioning and to maintain a changelog? As always, it depends on the size of the project at hand, but I would argue that if you want people to be able to use it without needing personal contact with you, it is time to have a changelog.

Even if this is not the case, however, a changelog may be very useful: if a project is being version-tracked, the act of writing out what has been modified can be helpful in clarifying what is happening with the code.

Poetry and dependency management

A common source of issues in software development lies in dependency management. Ideally, we would like the user to be able to install our package without having to worry about its dependencies, with everything being handled automatically.

This has been a long-standing issue in the `python` ecosystem, and a complete solution does not exist. However, a lot of work is going into it, and the most promising approach seems to be the management of a `pyproject.toml` file with `poetry`.

A usage guide for `poetry` is beyond the scope of this work, and the documentation for it is quite good. It allows for the automatic creation of virtual environments containing only the dependencies specified as required for the project, thus making it less likely that we will inadvertently use functionality from dependencies we are not specifying. It also automates checks for valid version combinations, as well as version control and publishing a project to `pypi`.

Overall, it is a very convenient system to manage the versions of our software and of its dependencies.

Documentation

A crucial aspect in software usability is the presence of good documentation.

In my experience, when documentation is brought up people's mind often goes to comments in the code, or line comments; but these are not proper *documentation*. Documentation is meant for the *user* of our code, while line comments are at best useful for future developers. As Clean Code (Martin 2008) puts it, line comments are a "necessary evil".

Sometimes, a simple-looking piece of code has a counterintuitive element, which may be clarified by a quick line comment; however, in most of their typical uses, line comments could be substituted by clearer code. The rest of this section, therefore, will not discuss line comments.

The diátaxis framework

The diátaxis framework (Procida 2022) allows us to structure our thinking about documentation according to the needs of the user, as opposed to our convenience when writing the code.

The website referenced above does an excellent job of explaining its categories, I will just give a quick summary here. The classification is along two axes: the first is based on whether the piece of documentation is meant to be used while studying or while working, while the second is based on whether the piece of documentation contains pratical steps or theoretical knowledge. Based on this, they distinguish the following categories:

1. **tutorials** show the user at study practical steps in a safe environment: they are meant to show them how to get started using the software;
2. **how-to guides** show the user at work practical steps in the real world: they are meant to show them how to accomplish some practical goal;
3. **reference material** describes software in a way that is helpful for a user at work, by listing its features, providing examples *etc.*;
4. **explanation** discusses the sofware broadly and theoretically, and is therefore meant for a user at study.

Reference material may be auto-generated in certain cases: for example, the documentation-formatting software for Python `sphinx` has an `autodoc` extension which can read function and class docstrings and extract them into HTML documentation. This, however, is not the be-all and end-all of documentation, and it is arguably not even the most important part of it.

Documentation for GWFish

Writing documentation may be a chore, and finding time to do it is difficult. This section is devoted to the thought process behind the documentation I wrote for `GWFish`, and how it grew organically from user requirements.

I started writing it when I was asked to give a short tutorial on the usage of this piece of software for people working on another gravitational wave detector proposal, LGWA (Harms et al. 2021). Having a spoken tutorial refer to written documentation is helpful, therefore I wrote it down as a documentation page. I used `sphinx` combined with `readthedocs` to make it so documentation could be version-tracked in the same repository as the code, as well as be deployed automatically whenever changes were made there.

Since the aim of a tutorial is to introduce new users to the software, I focused mine on a simple test case: computing the Fisher matrix errors for a single signal, with a specific combination of future planned detectors. The tutorial is in two parts, here and here; it is basic in the sense that it assumes no prior knowledge of the software itself, but it does assume familiarity with the task of Fisher matrix forecasting.

The tutorial provides runnable scripts and their expected output, with an explanation of what that output means. It is computationally cheap, since it is meant to be a learning tool, not to solve any concrete problems.

This was the starting point; after the architecture (`sphinx`+`readthedocs`) for the documentation was built it was easy to make small improvements and complement documentation pages with new information.

A few examples of things that were added are as follows:

- the conventions for the naming of relevant parameters are not uniform, therefore I added a page to the *reference material* section detailing the ones used by this code specifically;
- a piece of functionality for this code is the computation of a detection horizon (i.e. the maximum distance at which a certain signal can be detected), I discussed the line of reasonining behind this computation in the *explanation* section;
- users may want to add new, custom detectors to do their own experiments: this was a good candidate for a very brief *how-to* page, in which the steps to complete this real-world task are detailed; this page does not discuss the details of all the possible configurations for the detector, since that is material best deferred to a reference page.

Crucially, all of these arose from someone's need (often mine) of having a clear explanation of some aspect of the software readily available. Documentation grew organically, without any titanic one-time effort.

Testing

We should make sure the code we write works. That is fairly uncontroversial, but the way to practically test it is definitely nontrivial.

Often, in scientific code, what is tested are the end-to-end results of the code, but not the intermediate steps; these may only be tested informally, in an *ad hoc* way, if at all.

In the `GWFish` case, end-to-end testing was performed by way of a comparison with alternative software which did the same computations. Specifically, cross-checks were performed between `GWFish`, `GWFast` (Iacovelli et al. 2022) and `GWBench` (Borhanian 2021) as an activity within the Einstein Telescope Observational Science Board, which is an organization dedicated to developing the science case for the Einstein Telescope proposal. Since all these pieces of software are working with the same assumptions — Fisher matrix approximation, the same parametrization for the planned detectors, *etc.* — they should yield the same end result, and indeed, they did, at least for the situations considered. This effort can give us confidence that those versions of all codes were working correctly — reaching the same, wrong result with three independent approaches is of course possible but unlikely (or it is an intrisic feature of the assumptions made by all three).

This kind of testing is definitely useful, but it is not the main subject of this section. What we will discuss instead is how to make an *automated test suite*, which can be expanded as more features are added to the code, and which may be made complete enough that the fact it passes (i.e. runs without any errors) can give us a reasonable degree of confidence that our code is working sensibly.

We will not prove our code correct, but we can construct a series of checks that it is not failing in any silly way. This simplifies the development process significantly, since we can check at any time whether we have broken anything. Also, it is not too difficult an extension to run our test suite in an isolated environment with different software versions; this way we can make an informed claim about which ones our software supports and which it doesn't.

The golden standard in this regard is called *test driven development*, in which a workflow is adopted in which a test is written *before* the code which implements the feature it is testing. Before getting to that, however, we shall discuss the simpler task of how to test existing code: what paradigms and techniques we can use to construct good and convenient tests?

Unit testing for matrix inversion

This section showcases how unit tests can be added to a relatively simple section of code: a function within `GWFish` meant to invert matrices, with some extra restrictions. Really, while testing it we will see that it does not really compute the inverse of a matrix but a pseudoinverse; the tests here may be viewed as *exploratory*, I wrote them as I would when testing a "black box" function whose behaviour in edge cases is unknown.

Fisher matrix inversion and singularity issues

Within `GWFish`, an important step is the inversion of the Fisher matrix, which is required in order to provide estimates of the errors on the parameters.

This by itself does not seem like a difficult task computationally: after all, the matrices at hand are not very large (on the order of 10×10). However, issues do arise due to the differences in the magnitude between the various components: this can be quantified through the *conditioning number*, the ratio between the largest and smallest eigenvalues.

The code which inverts the Fisher matrix within `GWFish` looks like this:

```python
import numpy as np

def invertSVD(matrix):
    thresh = 1e-10
    dm = np.sqrt(np.diag(matrix))
    normalizer = np.outer(dm, dm)
    matrix_norm = matrix / normalizer

    [U, S, Vh] = np.linalg.svd(matrix_norm)

    kVal = sum(S > thresh)
    matrix_inverse_norm = U[:, 0:kVal] @ np.diag(1. / S[0:kVal]) @ Vh[0:kVal, :]

    return matrix_inverse_norm / normalizer
```

How can we investigate whether this code will correctly invert a matrix?

The simplest test

Let us start by building the simplest kind of test possible, which will already allow us to showcase some ideas about automated testing.

We start by adding a testing function to the same script as the one in which the `invertSVD` function is defined. The basic paradigm in testing is, of course, to run the code with some input and see whether it produces the correct result.

We will use a symmetric matrix, since all Fisher matrices are symmetric. We will not check exact equality, since when working with floating point numbers that cannot be guaranteed.

```
def test_matrix_inversion():

    matrix = np.array([[1, 3], [3, 4]])
    inverse = invertSVD(matrix)

    inverse_should_be = np.array([[-4/5, 3/5], [3/5, -1/5]])
    return np.allclose(inverse, inverse_should_be)

if __name__ == '__main__':
    print(test_matrix_inversion())
```

As expected, when running the script we get the result `True`; on the other hand, if we change one of the numbers in the `inverse_should_be` matrix we get `False`.

Several problems with this appear right away: do we really need to manually compute matrix inverses to test our code? We will get to that; first, though, let us *refactor* our test.

As is, we need to actively look at the output of the script in order to see whether our test has succeeded or failed. Also, the test and the actual code live in the same file, which is not great: as we add more tests, it will become a source of clutter.

Using `pytest`

Our first refactoring step lies in moving the test code to its own script. Also, as opposed to returning a boolean value, we will use an `assert` statement.

`assert` is a convenient tool for debugging and testing: a statement like `assert x` will not do anything if `x` is truthy,[3] while it will fail with an `AssertionError` if `x` is falsey.

So, our code will look like:

```
from gwfish_matrix_inverse import invertSVD
import numpy as np

def test_matrix_inversion():

    matrix = np.array([[1, 3], [3, 4]])
    inverse = invertSVD(matrix)

    inverse_should_be = np.array([[-4/5, 3/5], [3/5, -1/5]])
    assert np.allclose(inverse, inverse_should_be)
```

[3]"Truthy" in this context means that, when casted to a boolean value, it will be cast to `True`. For example, `0` is falsey, while all other numbers are truthy.

```
if __name__ == '__main__':
    test_matrix_inversion()
```

and, unlike before, it will now not output anything if everything is working correctly, and raise an error if not (try it!).

The next step is to try the same thing with the `pytest` library. We first need to install it (`pip install pytest`); after that, in the folder containing these files, we may simply run:

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.9.11, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/jacopo/Documents/clean-coding-thesis/scripts/testing_2
collected 1 item

test_matrix_inverse.py .                                                 [100%]

============================== 1 passed in 0.07s ===============================
```

`pytest` is able to go through the files in the folder, see that one of them has a name starting with `test_`, inside it there's a function starting with `test_`, run that function, find no errors, give us a success!

Note that now calling the function `test_matrix_inversion` in the script is not required anymore: we may safely remove those last two lines.

What happens if the test fails? Here, `pytest` really shines: let us change one of the numbers in the should-be inverse, and re-run the same command:

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.9.11, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/jacopo/Documents/clean-coding-thesis/scripts/testing_2
collected 1 item

test_matrix_inverse.py F                                                 [100%]

=================================== FAILURES ===================================
____________________________ test_matrix_inversion _____________________________

    def test_matrix_inversion():

        matrix = np.array([[1, 3], [3, 4]])
        inverse = invertSVD(matrix)

        inverse_should_be = np.array([[-4/5, 3/6], [3/5, -1/5]])
>       assert np.allclose(inverse, inverse_should_be)
E       assert False
E        +  where False = <function allclose at 0x7f25d5d27280>(array([[-0.8,
    0.6],
       [ 0.6, -0.2]]), array([[-0.8,  0.5],\n       [ 0.6, -0.2]]))
E        +    where <function allclose at 0x7f25d5d27280> = np.allclose

test_matrix_inverse.py:10: AssertionError
=========================== short test summary info ============================
FAILED test_matrix_inverse.py::test_matrix_inversion - assert False
============================== 1 failed in 0.12s ===============================
```

`pytest` ran our test code and found an error. It tells us exactly where it found it, and specifically how it came about: we used the `np.allclose` function to check for the equality of two matrices which it prints out, so we can see at a glance what has gone wrong.

Of course, we might not understand where the problem is right away in general, but this all is about convenience, and having the tools at hand to spot as many details as possible.

Debugging

Often, the information shown by `pytest` will still not be enough. A really convenient thing to be able to do, then, is to start from the failure and work interactively with the variables defined at that time: this way, we can do all sorts of manipulations, or even make plots if we want!

This is easily achieved by adding the `--pdb` flag to our call to `pytest`. `pdb`, “python debugger”, is a standard tool for debugging, and it has plenty of features worth exploring. Here I will give just a flavor of the possibilities.

The shell looks like this:

```
$ pytest --pdb
[... same output as before ...]
test_matrix_inverse.py:10: AssertionError
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> entering PDB >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

>>>>>>>>>>>>>>>>>> PDB post_mortem (IO-capturing turned off) >>>>>>>>>>>>>>>>>>>
> /home/jacopo/Documents/clean-coding-thesis/scripts/testing_2/
  test_matrix_inverse.py(10)test_matrix_inversion()
-> assert np.allclose(inverse, inverse_should_be)
(Pdb) matrix @ inverse
array([[ 1.00000000e+00, -1.11022302e-16],
       [ 4.44089210e-16,  1.00000000e+00]])
(Pdb) matrix @ inverse_should_be
array([[ 1.0000000e+00, -1.0000000e-01],
       [-4.4408921e-16,  7.0000000e-01]])
(Pdb) q


=========================== short test summary info ============================
FAILED test_matrix_inverse.py::test_matrix_inversion - assert False
!!!!!!!!!!!!!!!!!! _pytest.outcomes.Exit: Quitting debugger !!!!!!!!!!!!!!!!!!
```

It is hard to show in a fixed medium such as this, but after the test failure I was presented with a shell prompt `(Pdb)`, from which I could give arbitrary `python` commands.

I used it to compute the matrix product between the initial matrix and the computed inverse (with the numpy shortcut `A@B`$=AB$ in a matricial sense), and the same with the manually-written inverse, which showed that the computed inverse was indeed correct.

Property-based testing

So far, we have tested the output of our code against a manually computed “correct result”. This is OK as far as it goes, but is necessarily only checks a few examples which we hope will be relevant, but which might not cover all edge cases.

In many situations, we may be able to find an *invariant* in our code, which we expect to hold regardless of input. In this matrix inversion scenario, this is particularly simple: the defining property of the inverse A^{-1} of a matrix A is that $A^{-1}A = AA^{-1} = 1$.

The first step in this direction is to refactor our test so that it can accept any matrix: the following implementation uses the same matrix as before, but now we check the aforementioned property as opposed to the specific inverse.

```
MATRIX = np.array([[1, 3], [3, 4]])

def test_matrix_inversion_constant_matrix(matrix = MATRIX):

    inverse = invertSVD(matrix)

    assert np.allclose(inverse@matrix, np.eye(*matrix.shape))
    assert np.allclose(matrix@inverse, np.eye(*matrix.shape))
```

The method `np.eye` is a convenient way to generate an identity matrix with arbitrary shape.

This is the first step; what we could now do is to construct a method which generates random matrices and feed it to the algorithm.

This would already be quite good, but there is a better way, thanks to the `hypothesis` library.

We are starting out on a fairly complex example (a matrix full of floating point numbers) for it — if we were manipulating, say, strings, things could be quite a bit simpler —; however, it is the typical kind of task that we might need in a scientific context.

After installing `hypothesis` with the `numpy` extra (`pip install hypothesis[numpy]`), we may use it as follows:

```
from gwfish_matrix_inverse import invertSVD
import numpy as np
from hypothesis import given
from hypothesis import strategies as st
from hypothesis.extra.numpy import arrays

@given(arrays(np.float64, (2, 2)))
def test_matrix_inversion_hypothesis(matrix):

    inverse = invertSVD(matrix)

    assert np.allclose(inverse@matrix, np.eye(*matrix.shape))
    assert np.allclose(matrix@inverse, np.eye(*matrix.shape))
```

The `@given` decorator is what tells `hypothesis` to provide us with some test data, of the kind specified in its argument: for us, numpy `arrays`. We then specify the data type (floating point numbers), the shape (which for now we keep as 2x2, we will generalize this later), and any extra conditions. The data provided by `hypothesis` will not be *random* but *arbitrary*: it will purposefully try extreme examples, trying to get our code to break. We will then be able to constrain the parameter space we allow it to explore when attempting this if we wish.

If we run this code, we get an immediate failure: I will not clutter this document with the full output, but the command to run is still `pytest`, which fails by raising an error in the SVD step with the falsifying example:

$$A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Fair enough: the inverse of the zero matrix does not exist. One would now probably think of somehow restricting the examples to invertible matrices, but let us try to simply put a lower bound on the numbers allowed, so that they cannot be zero:

```
from gwfish_matrix_inverse import invertSVD
import numpy as np
from hypothesis import given
from hypothesis import strategies as st
from hypothesis.extra.numpy import arrays

@given(arrays(np.float64, (2, 2), elements=st.floats(min_value=1e-20)))
def test_matrix_inversion_hypothesis(matrix):

    inverse = invertSVD(matrix)

    assert np.allclose(inverse@matrix, np.eye(*matrix.shape))
    assert np.allclose(matrix@inverse, np.eye(*matrix.shape))
```

As expected, the code fails with a singular matrix,

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

but surprisingly it does not raise an error: instead, it returns the matrix

$$A^{-1} \overset{?}{=} \begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}.$$

The test then fails on the step of checking that this is the inverse (since it is not).

This is getting to the problem which really does occur in these computations: the Fisher matrix $\mathcal{F}$ is often singular or nearly-singular, and in order to deal with this the quantity this code is computing is actually not the matrix inverse, but the *Moore-Penrose pseudoinverse*, which is only the correct inverse in the subspace defined by the span of the matrix (to numerical precision, that is, with very small eigenvalues being approximated as zero). Formally, instead of $AA^{-1} = A^{-1}A = 1$, this pseudoinverse A^+ must satisfy $A^+AA^+ = A^+$ and $AA^+A = A$: let us therefore check these conditions.

We really should test this only the kinds of inputs we expect to be possible. Doing so in this case turned out to be tricky but possible with `hypothesis`.

Gravitational-wave Fisher matrices can always be written as $M_{ij} = \vec{v}_i \cdot \vec{v}_j$ for vectors $\vec{v}$ lying in some high-dimensional vector space (specifically, the vector space is the Hilbert space of waveforms with the product $(\cdot|\cdot)$, and the vectors are the derivatives $\vec{v}_i = \partial_i h$). They can therefore be expressed as $M_{ij} = |v_i||v_j| \cos(\theta_{ij})$. This is definitely not true for all matrices!

The following test generates arbitrary vectors v_i and cosines $c_{ij} \in [-1, 1]$, and then the matrices as $M_{ij} = v_i v_j c_{ij}$. The set of matrices that can be generated this way is a superset of the one of valid Fisher matrices.

The conditions $c_{ii} = 1$ and $c_{ij} = c_{ji}$, which will always hold for angles amongst vectors, are enforced *a posteriori*.

```python
from gwfish_matrix_inverse import invertSVD
import numpy as np
from hypothesis import given, reject, target, seed
from hypothesis import strategies as st
from hypothesis.extra.numpy import arrays
import pytest

MATRIX_DIMENSION = 4
ABS_TOLERANCE = 1e-1
REL_TOLERANCE = 1e-2
MIN_NORM = 1e-5
MAX_NORM = 1e5


@seed(1)
@given(
    vector_norms=arrays(
        np.float64,
        (MATRIX_DIMENSION,),
        elements=st.floats(
            min_value=MIN_NORM,
            max_value=MAX_NORM,
        ),
        unique=True,
    ),
    cosines=arrays(
        np.float64,
        (MATRIX_DIMENSION, MATRIX_DIMENSION),
        elements=st.floats(
            min_value=-1.0,
            max_value=1.0,
        ),
        unique=True,
    ),
)
def test_matrix_pseudoinverse_hypothesis(vector_norms, cosines):

    cosines[np.arange(MATRIX_DIMENSION), np.arange(MATRIX_DIMENSION)] = 1
    cosines = np.maximum(cosines, cosines.T)

    matrix = np.outer(vector_norms, vector_norms) * cosines

    inverse = invertSVD(matrix)

    assert np.allclose(
        inverse @ matrix @ inverse, inverse, atol=ABS_TOLERANCE, rtol=
    REL_TOLERANCE
    )
    assert np.allclose(
        matrix @ inverse @ matrix, matrix, atol=ABS_TOLERANCE, rtol=REL_TOLERANCE
    )
```

This is not the be-all-and-end all for this kind of test: for one, the tolerances were hand-picked, and a better understanding of the numerical problem is desirable. Still, this showcases how powerful this property-based testing framework is. Not all tests can be formally specified in this way, but many can, and if possible writing a test of this sort is very powerful.

Testing against different versions with `tox`

Ideally, we would like to not rely on a specific version of our dependencies, but instead to support a range of versions for them. Testing every possible combination is unfeasible, but we can go to some length by at least checking every version of our "main" dependencies. For example, we can check that our software is installable with every currently supported version of `python`. This section will discuss a tool to automate this: `tox`.

It allows us to run our tests in a freshly created virtual environment with arbitrary package versions. In order to use it we need to structure our code snippet as a package, so we will use `poetry` with a `pyproject.toml` file as follows:

```
[tool.poetry]
name = "matrix_inverse"
version = "0.1.0"
description = ""
authors = ["jacopo <jacopo.tissino@gssi.it>"]
packages = [{include = "matrix_inverse"}]

[tool.poetry.dependencies]
python = ">=3.8, <3.12"
pytest = "^7.2.0"
hypothesis = "^6.56.4"
numpy = "^1.23.4"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Then, we need to move our code to a folder called `matrix_inverse`. Further, for consistency we will move the test code in a folder called `tests`. Finally, we can install `tox` and create a `tox.ini` file containing

```
[tox]
skipsdist = true
envlist = py{38,39,310,311}
isolated_build = true

[testenv]
deps =
    poetry
commands =
    poetry install
    poetry run pytest {posargs}
```

Then we are done! We can run `tox` from the shell, which will create virtual environments with our dependencies for python 3.8, 3.9, 3.10 and 3.11; it will then run the tests within these.

This takes a while the first time and the output is very long, but it ends with

```
__________________________________ summary ___________________________________
  py38: commands succeeded
  py39: commands succeeded
  py310: commands succeeded
  py311: commands succeeded
  congratulations :)
```

We can check that the system is working properly by trying to exploit some functionality that is available only in certain python releases. For example, the package `tomllib` was added to the standard python library in version 3.11; before, one would have had to import it manually. So, if without further changes we add a line `import tomllib` to our module or test code, we expect to get an error for versions 3.10 and below; indeed, the output is

```
__________________________________ summary ___________________________________
ERROR:   py38: commands failed
ERROR:   py39: commands failed
ERROR:   py310: commands failed
  py311: commands succeeded
```

Running such a combination of tests can get expensive as the test suite grows, but it may be worth it, especially if we require backward compatibility. For example, I often use the latest `python` release, but sometimes develop code that will be run on clusters whose installations are only updated irregularly; it is therefore important for me to be able to easily check that the code I am writing is compatible with the version installed there.

When to write tests

Writing tests is time-consuming, but once they have been written they are extremely useful. When is a good time to write them?

If we are developing a new feature, a good answer to this is: *before* writing the code that implements the feature. This practice is known as **test-driven development** (TDD), and it is more than twenty years old. The idea is that writing the test first forces the developer to think about what the thing they are implementing should accomplish.

If this seems hard, that's a feature, not a bug: writing code without being able to formulate a test case for it is indicative of the code being not well-thought-out, or perhaps not modular enough.

This may be applied to new code, but often the real situation is that we have a large module without any tests: what to do then? One approach, which will be showcased in the next chapter, is to start by creating some **end-to-end** tests to document the expected behavior of the code, and then refactor / simplify it as needed (since it will probably need refactoring).

Another opportunity to write tests is in the event of **bugs** or problems: if we encounter a failure for a certain input to our code, we can calcify that input as a test case. In `pytest`, the `@pytest.mark.xfail` decorator can help in clarifying: the test we just wrote is known to be currently failing. When the bug is fixed, the test should succeed, and it can be kept in our suite, allowing us to avoid regressing on that bug fix.

Finally, the kind of test development shown in the previous sections is a sort of "**exploratory testing**", which is yet another context in which tests can be written. If we have some functionality which we do not completely understand the behavior of, perhaps because it was written by somebody else, or maybe it even lies in a different package. As we are trying to make sense of it — as we would need to anyway — we can write tests cementing our understanding of its behavior. Besides allowing us to clarify our understanding in this exploratory phase, the tests will remain there acting as a safeguard against changes in the functionality, in the case that it is externally managed. For example, if we are relying on an external library which gets an update, we can quickly check whether it breaks our use case.

Refactoring

All previous sections have been *around* code more than *about* it: how to track its versions, document it, test it. Indeed, the aforementioned topics should typically have a higher priority than making changes to the code. Eventually, though, we will have to modify it: how and why should we?

In order to make the discussion here somewhat constrained, I will limit it to the refactoring of one single function, purposefully doing so without modifying anything else.

Refactoring `analyzeFisherErrors`

The computation of the errors from Fisher matrices in `GWFish` goes through a function called `analyzeFisherErrors`, which takes the following parameters:

- `network`, an object of type `Network`, which contains several `Detector` objects;
- `parameter_values`, a `pandas DataFrame` with each row representing a signal, and with each column corresponding to a different event parameter (masses, distance etc.);
- `fisher_parameters`, a list containing the aforementioned parameters in order;
- `population`, a string containing the name of the population being considered;
- `networks_ids`, a list of lists of integers, each between 0 and the number of detectors minus one.

The idea with the `networks_ids` parameter is the following: suppose we have a network of three detectors, A B and C, and that we want to be able to compute the capabilities they would have all together or as pairs of 2. We can accomplish this with `network_ids=[[0, 1], [1, 2], [0, 2], [0, 1, 2]]`: the numbers in each sub-list are indices used to select the detectors making up a subnetwork.

This function performs the following tasks for each of the subnetworks:

- compute the overall signal-to-noise ratio (SNR) as the root-mean-square of the individual SNRs;
- compute the network Fisher matrix as the sum of the individual Fisher matrices;
- compute the covariance matrix by inverting the network Fisher matrix;
- compute the sky localization ellipse size, which is determined by the errors on the sky position angles (right ascension and declination);
- write all the computed errors to a text file.

The function does not return anything. The code is reported here for reference, with minor changes to allow for correct printing. The full one can be found in this snapshot of the repository.

```python
def analyzeFisherErrors(network, parameter_values, fisher_parameters, population,
    networks_ids):
    """
    Analyze parameter errors.
    """

    # Check if sky-location parameters are part of Fisher analysis.
    # If yes, sky-location error will be calculated.
    signals_havesky = False
    if ('ra' in fisher_parameters) and ('dec' in fisher_parameters):
        signals_havesky = True
        i_ra = fisher_parameters.index('ra')
        i_dec = fisher_parameters.index('dec')
    signals_haveids = False
    if 'id' in parameter_values.columns:
        signals_haveids = True
        signal_ids = parameter_values['id']
        parameter_values.drop('id', inplace=True, axis=1)


    npar = len(fisher_parameters)
    ns = len(network.detectors[0].fisher_matrix[:, 0, 0])  # number of signals
    N = len(networks_ids)

    detect_SNR = network.detection_SNR

    network_names = []
    for n in np.arange(N):
        network_names.append('_'.join(
          [network.detectors[k].name for k in networks_ids[n]])
        )

    for n in np.arange(N):
        parameter_errors = np.zeros((ns, npar))
        sky_localization = np.zeros((ns,))
        networkSNR = np.zeros((ns,))
        for d in networks_ids[n]:
            networkSNR += network.detectors[d].SNR ** 2
        networkSNR = np.sqrt(networkSNR)

        for k in np.arange(ns):
            network_fisher_matrix = np.zeros((npar, npar))

            if networkSNR[k] > detect_SNR[1]:
                for d in networks_ids[n]:
                    if network.detectors[d].SNR[k] > detect_SNR[0]:
                        network_fisher_matrix += np.squeeze(
                          network.detectors[d].fisher_matrix[k, :, :])

                if npar > 0:
                    network_fisher_inverse = invertSVD(network_fisher_matrix)
                    parameter_errors[k, :] = np.sqrt(
                      np.diagonal(network_fisher_inverse))

                    if signals_havesky:
                        sky_localization[k] = (
                          np.pi * np.abs(np.cos(parameter_values['dec'].iloc[k]))
                          * np.sqrt(network_fisher_inverse[i_ra, i_ra]
                          * network_fisher_inverse[i_dec, i_dec]
                          - network_fisher_inverse[i_ra, i_dec]**2))
        delim = " "
        header = ('network_SNR '+delim.join(parameter_values.keys())+
          " "+delim.join(["err_" + x for x in fisher_parameters])
        )

        ii = np.where(networkSNR > detect_SNR[1])[0]
        save_data = np.c_[networkSNR[ii],
          parameter_values.iloc[ii],
          parameter_errors[ii, :]]
        if signals_havesky:
            header += " err_sky_location"
            save_data = np.c_[save_data, sky_localization[ii]]
        if signals_haveids:
            header = "signal "+header
            save_data = np.c_[signal_ids.iloc[ii], save_data]

        file_name = ('Errors_' + network_names[n] + '_'
          + population + '_SNR' + str(detect_SNR[1]) + '.txt'
          )

        if signals_haveids and (len(save_data) > 0):
            np.savetxt('Errors_' + network_names[n] + '_'
              + population + '_SNR' + str(detect_SNR[1]) + '.txt',
              save_data,
              delimiter=' ',
              fmt='%s ' + "%.3E " * (len(save_data[0, :]) - 1),
              header=header,
              comments=''
            )
        else:
            np.savetxt(
              'Errors_' + network_names[n] + '_' + population +
              '_SNR' + str(detect_SNR[1]) + '.txt',
              save_data,
              delimiter=' ',
              fmt='%s ' + "%.3E " * (len(save_data[0, :]) - 1),
              header=header,
              comments=''
            )
```

This function was not written from scratch like this: it shows the signs of a simple function being gradually extended to "work above its pay grade", which made it quite complex. It has many switches and several layers of loops. It features some duplication, which is a violation of the so-called "DRY principle" (Don't Repeat Yourself): similar but different parts of the code being activated in different situations are dangerous, since it is easy to modify one but not the other.

The next sections will discuss how these issues can be addressed, but one thing needs to be made explicit: if this was one-off code, it would be fine as-is. The necessity to refactor it came from actual user needs — specifically, a user required `hdf5` output for the errors, which would not be easily achievable with the function as written here. As opposed to adding another `if` clause, we can refactor the logic and make it more modular.

Writing the function in a "messy" way and *then* refactoring it *if needed* is a good process: the messy code is often easier to write quickly, it does not require us to think about which abstractions we should use and how each of the modular components we make maybe used in other contexts. Premature abstraction can create technical debt just like forests of for loops can. So, this section should not be interpreted as "fixing bad code", but as a natural part of the development process: starting with the easiest-to-write code that will get the job done, and then refining it.

Too many purposes

The first thing that catches the eye is that this function is doing two "big" things at once: calculations (combining Fisher matrices etc.) and input-output (I/O: writing out to a file). Combining them may seem natural in a first formulation, since once we have computed these values we want to save them somewhere, but this poses several issues.

- If a user wishes to use the computed errors, they have no way to access them directly, and if they do not modify the source code they must save to a file and then read from that file;
- in this case, loss of information: the values are being saved in scientific notation with three decimals in the mantissa, which means all information beyond these digits is lost;
- tests are really difficult to write for such a function: it has *side effects*, meaning that when it is run it saves things to disk somewhere, as opposed to being fully characterized by its output.

A simple solution to this is to split the function in two: one for the computation, one for the output.

We can do better: several sub-tasks in this function can be modularized, with the guiding principle of having each function perform a single conceptual task.

Type hinting

This function's call signature is a good example of how type hints can be useful. Without reading the function code, how could we be able to tell that, for example, `population` should be a string while `network` should be an object of type `Newtork`?

Documenting the function could be a solution, but is not the best one for this purpose. Despite our best efforts, it is not guaranteed to remain up-to-date; also, we may get it wrong, and there is no way to automatically check it.

In this context, since we are writing the code of a relatively large module, it makes sense to use type hinting: python is dynamically typed, so it allows us to not say anything about the types of the variables we are using, but we do have functionality to specify which types we expect.

The syntax to do it in this case is as follows:

```python
def analyzeFisherErrors(
    network: det.Network,
    parameter_values: pd.DataFrame,
    fisher_parameters: list[str],
    population: str,
    networks_ids: list[list[int]]
) -> None:
```

Python by itself will completely ignore these (as long as they can be evaluated without raising an error), which means they are only as good as documentation written in a comment; however, there are ways in which they can be better. For one, they are returned when calling `help(analyzeFisherErrors)` in a console. That is, however, also true for the docstring of the function.

The real power of type annotations in `python` is that they can be formally checked by a tool, called a *static type checker*, which will raise an error if, for example, a function is called with the wrong types, or if it uses its arguments in a way that is not compatible with their stated types. There are several of these available, but here we will focus on `mypy`.

This is a very useful tool to make spotting errors easier in a big codebase; since it is only used on the module code, it allows us to have clarity on the types within our module while retaining the flexibility of dynamic types when we use our code in an interactive console or a script.

After installing `mypy` (`pip install mypy`), we may use it from the command line with the command

```
mypy fishermatrix.py
```

where `fishermatrix.py` is the name of the module file containing the `analyzeFisherErrors` function. By itself this will raise several errors, of the sort

```
auxiliary.py:1: error: Skipping analyzing "scipy": module is installed,
  but missing library stubs or py.typed marker
```

for many packages beyond `scipy`. This reflects the fact that these imported packages are not adopting type hints themselves, which means calls to them cannot be checked. This reflects the fact that type hinting is a relatively recent addition to the `python` ecosystem, and many large libraries have not adopted it.

This does not prevent us from using `mypy` in our own code, however, it might just limit its usefulness; in order to get rid of the error we can run `mypy` with the option `--ignore-missing-imports`. With it, we get

```
$ mypy fishermatrix.py --ignore-missing-imports
Success: no issues found in 1 source file
```

This is good! Note that the file also contains other, non-type-hinted functions, which does not create any issue. We could enforce typing on *every* function with the option `--strict`.

We can make sure that `mypy` is indeed checking the body of the function by making the signature wrong in some way: for example, if we change the hint on the `network_ids` argument to `networks_ids: int` (as opposed to `networks_ids: list[list[int]]`) we get the error

```
$ mypy fishermatrix.py --ignore-missing-imports
fishermatrix.py:52: error: Argument 1 to "len" has incompatible type "int";
    expected "Sized"
fishermatrix.py:59: error: Value of type "int" is not indexable
fishermatrix.py:66: error: Value of type "int" is not indexable
fishermatrix.py:74: error: Value of type "int" is not indexable
Found 4 errors in 1 file (checked 1 source file)
```

`mypy` is able to understand that we are using this parameter as a list (for example, indexing it), so if it were an integer it would not be valid. Several errors can be caught this way.

Automatic formatting

Git commits often get polluted with meaningless whitespace changes, newlines added somewhere, and so on. This takes away from our ability to understand what was actually changed. Also, if different parts of the code are written by different people, the style will often look inconsistent.

This is a somewhat minor thing, but having automatic formatting as a part of our development routine ensures consistency and makes all code easier to read. There are several choices for this task, and I personally like `black` (of course, this is something that should be agreed on within a project). For `python`, standard style is defined by PEP8; while conforming to it is not a requirement, it is used widely enough that code formatted according to it will be readily understandable to a large amount of people.

After installing it with `pip install black`, we may format any file or folder with a command such as:

```
$ black fishermatrix.py
reformatted fishermatrix.py

All done!
1 file reformatted.
```

This is the output of the first run, while subsequent ones will show a message such as

```
All done!
1 file left unchanged.
```

Linting and PEP8

We can also use automated analysis tools to check our code for style and compliance to best practices: this is called *linting*.

When running `pylint`, a common linter, on this function, we get a few warnings, shortened here for brevity:

```
$ pylint fishermatrix.py
************* Module fishermatrix
fishermatrix.py:38:0: C0301: Line too long (114/100) (line-too-long)
fishermatrix.py:12:0: C0116: Missing function or method docstring (missing-
    function-docstring)
fishermatrix.py:15:4: C0103: Variable name "dm" doesn't conform to snake_case
    naming style (invalid-name)
fishermatrix.py:27:0: C0103: Function name "analyzeFisherErrors" doesn't conform
    to snake_case naming style (invalid-name)
fishermatrix.py:27:0: R0914: Too many local variables (29/15) (too-many-locals)
fishermatrix.py:27:0: R0912: Too many branches (15/12) (too-many-branches)
```

Some of these are rather generic warnings, which can be disabled or changed; however, they do point to some issues we discussed earlier: this function has too many tasks, and this naturally shows up in its number of local variables, branches (e.g. `if`s and `for`s).

The heuristics used by a given linter are not gospel, but they may give us a helpful indication about which parts of the code were not carefully written.

Git hooks

Things such as automatic formatting, linting, and even static type checking, are very useful if applied consistently. However, we are fallible and may forget to do so; also, typing `black <name.py>` often gets annoying.

There is a nice way, however, to ensure that badly-formatted code does not have the chance to get into our version tracking: git hooks.

Using them is not very difficult: we just need to create a file called `pre-commit` in the folder `.git/hooks/`. Its content, as a bash script, will be executed every time we make a commit; if it returns a non-zero code (which, for example, `black` will do if it needed to modify the files it found) it will abort the commit. However, it will have modified the relevant files; re-adding them will allow us to make a commit with only correctly-formatted files.

Premature optimizations

The code in `analyzeFisherErrors` is often using the syntax `for i in np.arange(N)` in order to loop over `N` numbers, as opposed to the native python `for i in range(N)`. The logic behind it was to improve performance: after all, `numpy` is typically faster than native `python` for vector and matrix operations.

However, using it in this context is a case of premature optimization. Benchmarking the whole function for this example is overkill, so let us use a simpler example: suppose we want to compute (but not store) the sums of the first 1000 integers. The native-python solution is

```
for x in range(1000):
    x+x
```

which takes about 20 µs on my machine (measured with the `ipython` magic `%timeit`). The alternative with `np.arange`,

```
for x in np.arange(1000):
    x+x
```

takes roughly double that: 43 µs. The way to really get a speed improvement in this context would be to ditch the `for` loop completely, and directly work with `numpy` vectors:

```
x_vec = np.arange(1000):
x_vec + x_vec
```

This takes roughly 1.4 µs.

Really, though, the `for` loops within `analyzeFisherErrors` are not the bottleneck in its evaluation, and this function is a rather fast component of the code. I would argue that in this case readability matters more than speed; even if `np.arange` was slightly faster than `range`, it would still be worth it to use the simpler native syntax in order to have less visual clutter.

Here performance is secondary to speed, but sometimes it may not be. In those cases, proper profiling is essential. There are several tools for this in the `python` ecosystem. For targeted optimization of a single function, I have had great success with `cProfile` combined with the visualization tool `snakeviz`: they can provide a breakdown of each function call happening inside the target function, allowing us to understand which the potentially slow parts are.

Another great tool which has recently been rising in popularity is `scalene` (Berger 2020). It allows for the profiling of memory usage, as well as distinguishing time spent in native `python` versus time spent running wrappers around `C` code.

This is useful since a common strategy to accelerate `python` code is indeed to take the numerical kernel of our computations and have it be computed by efficient, compiled `C` code. If we made a mistake and the `C` code is not being run, evaluation may be slow: `scalene` can aid in spotting such issues.

Test-aided refactoring and mocking

When refactoring, an important part of the job is to ensure that we are not breaking existing functionality.

So, before modifying anything substantial we should be covered by a test (or multiple). A minimal example of calling the `analyzeFisherErrors` function looks like the following. Note that, while it is formatted as a test, it is currently not performing any actual testing! As written, this is no more than a *smoke test*.[4]

```
import pytest
from fishermatrix import analyzeFisherErrors
from detection import Network, Detector
import pandas as pd
import numpy as np


def test_fisher_analysis_output():

    params = {
        "mass_1": 1.4,
        "mass_2": 1.4,
```

[4]The terms may originate in analog circuit design: this "test" is the equivalent of plugging in a soldered circuit board and seeing whether smoke is coming out of it. The absence of smoke is not a guarantee of the correctness of the results.

```
        "redshift": 0.01,
        "luminosity_distance": 40,
        "theta_jn": 5 / 6 * np.pi,
        "ra": 3.45,
        "dec": -0.41,
        "psi": 1.6,
        "phase": 0,
        "geocent_time": 1187008882,
    }

    parameter_values = pd.DataFrame()
    for key, item in params.items():
        parameter_values[key] = np.full((1,), item)

    fisher_parameters = list(params.keys())

    network = Network(
        detector_ids=["ET"],
        parameters=parameter_values,
        fisher_parameters=fisher_parameters,
        config="detectors.yaml",
    )

    network.detectors[0].fisher_matrix[0, :, :] = fishermatrix.FisherMatrix(
        "gwfish_TaylorF2",
        parameter_values.iloc[0],
        fisher_parameters,
        network.detectors[0],
    )

    network.detectors[0].SNR[0] = 100

    analyzeFisherErrors(
        network=network,
        parameter_values=parameter_values,
        fisher_parameters=fisher_parameters,
        population="test",
        networks_ids=[[0]],
    )
```

The fact that the minimum code required to run this function is so large is an indication of the high degree of *coupling* in the codebase: a part of it cannot function independently of the others.

In this case, testing the output seems to be hard to do since our code is writing out a file. Specifically, running this code leads to a file named `Errors_ET_test_SNR8.0.txt` being created, with the content:

```
network_SNR mass_1 mass_2 redshift luminosity_distance theta_jn ra dec psi phase
    geocent_time err_mass_1 err_mass_2 err_redshift err_luminosity_distance
    err_theta_jn err_ra err_dec err_psi err_phase err_geocent_time
    err_sky_location
100.0 1.400E+00 1.400E+00 1.000E-02 4.000E+01 2.618E+00 3.450E+00 -4.100E-01 1.600
    E+00 0.000E+00 1.187E+09 1.018E-07 1.018E-07 8.969E-08 2.322E+00 1.042E-01
    3.127E-03 2.694E-03 2.042E-01 4.093E-01 5.639E-05 2.423E-05
```

An option would be to read the file as a part of the test and then delete it, but that is somewhat risky: the deletion might have issues (especially if the test fails), and if the file is still there for the next test we have created non-independent test cases.

There are various ways around this, but I will use it to showcase the concept of *mocking*: substituting a complex function with a simplified version, which does not have its full functionality, but which allows us to check that the complex function would have been called correctly.

In our case, we can mock the `numpy.savetxt` function, so that our test does not actually save anything to a file, but we just check that we *would* save the correct thing. We can do so with the `pytest-mock` library, and it looks like this:

```
import pytest
from fishermatrix import analyzeFisherErrors
```

```python
import fishermatrix
import waveforms
import detection
from detection import Network, Detector
import pandas as pd
import numpy as np


def test_fisher_analysis_output(mocker):

    params = {
        "mass_1": 1.4,
        "mass_2": 1.4,
        "redshift": 0.01,
        "luminosity_distance": 40,
        "theta_jn": 5 / 6 * np.pi,
        "ra": 3.45,
        "dec": -0.41,
        "psi": 1.6,
        "phase": 0,
        "geocent_time": 1187008882,
    }

    parameter_values = pd.DataFrame()
    for key, item in params.items():
        parameter_values[key] = np.full((1,), item)

    fisher_parameters = list(params.keys())

    network = Network(
        detector_ids=["ET"],
        parameters=parameter_values,
        fisher_parameters=fisher_parameters,
        config="detectors.yaml",
    )

    network.detectors[0].fisher_matrix[0, :, :] = fishermatrix.FisherMatrix(
        "gwfish_TaylorF2",
        parameter_values.iloc[0],
        fisher_parameters,
        network.detectors[0],
    )

    network.detectors[0].SNR[0] = 100

    mocker.patch("numpy.savetxt")

    analyzeFisherErrors(
        network=network,
        parameter_values=parameter_values,
        fisher_parameters=fisher_parameters,
        population="test",
        networks_ids=[[0]],
    )

    header = (
        "network_SNR mass_1 mass_2 redshift luminosity_distance "
        "theta_jn ra dec psi phase geocent_time err_mass_1 err_mass_2 "
        "err_redshift err_luminosity_distance err_theta_jn err_ra "
        "err_dec err_psi err_phase err_geocent_time err_sky_location"
    )

    data = [
        1.00000000000e02,
        1.39999999999e00,
        1.39999999999e00,
        1.00000000000e-02,
        4.00000000000e01,
        2.61799387799e00,
        3.45000000000e00,
        -4.09999999999e-01,
        1.60000000000e00,
        0.00000000000e00,
        1.18700888200e09,
        1.01791427671e-07,
        1.01791427689e-07,
        8.96883449508e-08,
        2.32204133549e00,
        1.04213847237e-01,
        3.12695677565e-03,
        2.69412953826e-03,
        2.04240222976e-01,
        4.09349000642e-01,
        5.63911212310e-05,
        2.42285325663e-05,
    ]

    assert np.savetxt.call_args.args[0] == "Errors_ET_test_SNR8.0.txt"
    assert np.allclose(np.savetxt.call_args.args[1], data)

    assert np.savetxt.call_args.kwargs == {
        "delimiter": " ",
        "header": header,
        "comments": "",
    }
```

With this test code ready as a “safety net”, we can move on to the refactoring.

`analyzeFisherErrors` refactored

I refactored the function as follows:

```
def sky_localization_area(,
`
    network_fisher_inverse: np.ndarray,
    declination_angle: np.ndarray,
    right_ascension_index: int,
    declination_index: int,
) -> float:
    """
    Compute the 1-sigma sky localization ellipse area starting
    from the full network Fisher matrix inverse and the inclination.
    """
    return (
        np.pi
        * np.abs(np.cos(declination_angle))
        * np.sqrt(
            network_fisher_inverse[right_ascension_index, right_ascension_index]
            * network_fisher_inverse[declination_index, declination_index]
            - network_fisher_inverse[right_ascension_index, declination_index] **
    2
        )
    )


def compute_fisher_errors(
    network: det.Network,
    parameter_values: pd.DataFrame,
    fisher_parameters: list[str],
    sub_network_ids: list[int],
) -> tuple[np.ndarray, np.ndarray, Optional[np.ndarray]]:
    """
    Compute Fisher matrix errors for a network whose
    SNR and Fisher matrices have already been calculated.

    Will only return output for the n_above_thr signals
    for which the network SNR is above network.detection_SNR[1].

    Returns:
    network_snr: array with shape (n_above_thr,)
        Network SNR for the detected signals.
    parameter_errors: array with shape (n_above_thr, n_parameters)
        One-sigma Fisher errors for the parameters.
    sky_localization: array with shape (n_above_thr,) or None
        One-sigma sky localization area in steradians,
        returned if the signals have both right ascension and declination,
        None otherwise.
    """

    n_params = len(fisher_parameters)
    n_signals = len(parameter_values)

    assert n_params > 0
    assert n_signals > 0

    signals_havesky = False
    if ("ra" in fisher_parameters) and ("dec" in fisher_parameters):
        signals_havesky = True
        i_ra = fisher_parameters.index("ra")
        i_dec = fisher_parameters.index("dec")

    detector_snr_thr, network_snr_thr = network.detection_SNR

    parameter_errors = np.zeros((n_signals, n_params))
    if signals_havesky:
        sky_localization = np.zeros((n_signals,))
    network_snr = np.zeros((n_signals,))

    detectors = [network.detectors[d] for d in sub_network_ids]

    network_snr = np.sqrt(sum((detector.SNR**2 for detector in detectors)))

    for k in range(n_signals):
        network_fisher_matrix = np.zeros((n_params, n_params))

        for detector in detectors:
            if detector.SNR[k] > detector_snr_thr:
                network_fisher_matrix += detector.fisher_matrix[k, :, :]

        network_fisher_inverse = invertSVD(network_fisher_matrix)
        parameter_errors[k, :] = np.sqrt(np.diagonal(network_fisher_inverse))

        if signals_havesky:
            sky_localization[k] = sky_localization_area(,
            `
                network_fisher_inverse, parameter_values["dec"].iloc[k], i_ra,
     i_dec
            )

    detected = np.where(network_snr > network_snr_thr)[0]

    if signals_havesky:
        return (
            network_snr[detected],
            parameter_errors[detected, :],
            sky_localization[detected],
        )

    return network_snr[detected], parameter_errors[detected, :], None


def output_to_txt_file(
    parameter_values: pd.DataFrame,
    network_snr: np.ndarray,
    parameter_errors: np.ndarray,
    sky_localization: Optional[np.ndarray],
    fisher_parameters: list[str],
    filename: str,
) -> None:

    delim = " "
    header = (
        "network_SNR "
        + delim.join(parameter_values.keys())
        + " "
        + delim.join(["err_" + x for x in fisher_parameters])
    )
    save_data = np.c_[network_snr, parameter_values, parameter_errors]
    if sky_localization is not None:
```

```python
        header += " err_sky_location"
        save_data = np.c_[save_data, sky_localization]

    row_format = "%s " + " ".join(["%.3E" for _ in range(save_data.shape[1] - 1)])

    np.savetxt(
        filename + ".txt",
        save_data,
        delimiter=" ",
        header=header,
        comments="",
        fmt=row_format,
    )


def errors_file_name(
    network: det.Network, sub_network_ids: list[int], population_name: str
) -> str:

    sub_network = "_".join([network.detectors[k].name for k in sub_network_ids])

    return (
        "Errors_"
        + sub_network
        + "_"
        + population_name
        + "_SNR"
        + str(network.detection_SNR[1])
    )


def analyze_and_save_to_txt(
    network: det.Network,
    parameter_values: pd.DataFrame,
    fisher_parameters: list[str],
    sub_network_ids_list: list[list[int]],
    population_name: str,
) -> None:

    for sub_network_ids in sub_network_ids_list:

        network_snr, errors, sky_localization = compute_fisher_errors(
            network=network,
            parameter_values=parameter_values,
            fisher_parameters=fisher_parameters,
            sub_network_ids=sub_network_ids,
        )

        filename = errors_file_name(
            network=network,
            sub_network_ids=sub_network_ids,
            population_name=population_name,
        )

        output_to_txt_file(
            parameter_values=parameter_values,
            network_snr=network_snr,
            parameter_errors=errors,
            sky_localization=sky_localization,
            fisher_parameters=fisher_parameters,
            filename=filename,
        )
```

A summary of the changes made is as follows.

- The functionality of `analyzeFisherErrors` was split into five: `compute_fisher_errors`, `sky_localization_area`, `output_to_txt`, `errors_file_name`, `analyze_and_save_to_txt`. These all contain logically distinct sections of the code, which we may desire to modify independently of each other.
- The `compute_fisher_errors` function now only considers one subnetwork, and the looping over subnetworks is relegated to the higher-level function `analyze_and_save_to_txt`.
- The check for `npar>0` which nested the whole function by one layer was changed to an

`assert` statement at the beginning of the function — if the number of parameters is less than zero the whole function call is invalid, and something has gone wrong.
- A few computations were happening in different places, such as detector selection (evaluating `network.detectors[d]`) or a check that the network SNR was greater than a threshold value. They were unified.
- The computation of `newtorkSNR` was compacted.

The resulting code is still not perfect, of course. For one, too many parameters are being passed amongst these functions. This makes them coupled, long, complex.

However, it is now straightforward to use the `compute_fisher_errors` function within a script, if we do not need to output to a file. Also, it is equally straightforward to make a function analogous to `analyze_and_save_to_txt` but which saves to a format different from `txt`.

A way to ameliorate this issue will entail modifying more than just `analyzeFisherErrors`. Probably, the most convenient approach will be to restructure the classes in the whole package, probably with one representing the whole population analyzed: the functions in the refactored code are all working on the same data (a table of parameter values, a list of which parameters to consider for the Fisher analysis, a table of Fisher errors etc.).

Conclusions

This thesis discussed how some best practices can be applied in order to make software more maintainable, user-friendly, readable.

Several aspects have not been discussed here, one of the most important of which is large-scale module organization and object-oriented programming: the structure of classes, the possibility to use inheritance, interfaces, and so on. The main reason why this topic was omitted, besides its inherent complexity, is that concretely applying it to `GWFish` will take a large effort, and it has not been done yet. Several improvements are underway for `GWFish`, both in terms of new functionality (such as allowing the treatment of different kinds of signals which are difficult to model with the current infrastructure) and of the application of the concepts discussed in this thesis.

The main takeaway I hope a reader will have from this thesis is in terms of a sort of "Maslow hierarchy"[5] of software needs. The main topics discussed are in rough order of *urgency*: if a piece of software is not being properly *versioned* that is the first problem to be solved, since we could not reliably keep track of any changes applied otherwise. The relative importance of *testing* and *documentation* is debatable, but both are surely more urgent than cosmetic and even *structural* changes such as the ones discussed in the "Refactoring" chapter.

Beautiful, clean and clever code is the equivalent of "self-actualization" in the hierarchy of needs. It is wonderful to be able to reach it, but there are several steps before it which need to be fulfilled for the clever snippet of code to be useful.

Implementing a robust *process* for the development of a software project that is growing large is not simple, but it is definitely worth it, and an initial time investment can have a significant payoff.

Bibliography

Berger, Emery D. 2020. "Scalene: Scripting-Language Aware Profiling for Python." http://arxiv.org/abs/2006.03879.

Borhanian, Ssohrab. 2021. "Gwbench: A Novel Fisher Information Package for Gravitational-Wave Benchmarking." *Classical and Quantum Gravity* 38 (17): 175014. https://doi.org/10.1088/1361-6382/ac1618.

Harms, Jan, Filippo Ambrosino, Lorella Angelini, Valentina Braito, Marica Branchesi, Enzo Brocato, Enrico Cappellaro, et al. 2021. "Lunar Gravitational-wave Antenna." *The Astrophysical Journal* 910 (1): 1. https://doi.org/10.3847/1538-4357/abe5a7.

[5]This is a reference to the notorious Maslow hierarchy of needs for humans, starting with a need for food and shelter and going all the way to self-actualization and transcendence.

Harms, Jan, Ulyana Dupletsa, Biswajit Banerjee, Marica Branchesi, Boris Goncharov, Andrea Maselli, Ana Carolina Silva Oliveira, Samuele Ronchini, and Jacopo Tissino. 2022. "GWFish: A Simulation Software to Evaluate Parameter-Estimation Capabilities of Gravitational-Wave Detector Networks."

Iacovelli, Francesco, Michele Mancarella, Stefano Foffa, and Michele Maggiore. 2022. "Forecasting the Detection Capabilities of Third-Generation Gravitational-Wave Detectors Using $\Texttt{GWFAST}$." arXiv. https://doi.org/10.48550/arXiv.2207.02771.

LIGO Scientific Collaboration and Virgo Collaboration, B. P. Abbott, R. Abbott, T. D. Abbott, M. R. Abernathy, F. Acernese, K. Ackley, et al. 2016. "Observation of Gravitational Waves from a Binary Black Hole Merger." *Physical Review Letters* 116 (6): 061102. https://doi.org/10.1103/PhysRevLett.116.061102.

Maggiore, Michele. 2007. *Gravitational Waves: Volume 1: Theory and Experiments*. 1 edition. Oxford: Oxford University Press.

Martin, Robert C. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st edition. Upper Saddle River, NJ: Pearson.

Procida, Daniele. 2022. *Diátaxis Documentation Framework*. https://diataxis.fr/.