Master's Thesis

in Astrophysics

Visualization of Gravitational Waves Using a Computational Analogue Model

by

Martin Bernard Ryan

Student ID: **12-766-556**

Supervised by

Prof. Prasenjit Saha

Physik-Institut

University of Zurich

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Astrophysics

at the University of Zurich

Zurich, Switzerland

January 2025

Dedicated to my mother, 1937 – 2024

1 Contents

About this Document

Abstract / Summary

The goal of this project was the implementation, using the Python programming language, of a simple numerical, computer-based visualisation of gravitational waves emitted by two orbiting bodies. It forms part of my Master's degree in Astrophysics at the University of Zurich.

Overview

The primary goal of this project was to create a visual simulation using the elastic sheet model as a prototype, showing gravitational waves emitted by two massive, fast-moving orbiting bodies, such as closely orbiting neutron stars. This document introduces the essential background in physics, mathematics, and computer science necessary to understand the implementation of the simulation.

It discusses the rationale behind the choice of implementation language, the development process, and the key challenges encountered and resolved along the way.

The simulation and this documentation not only serve as my MSc submission but also aim to enhance conceptual understanding of certain aspects of general relativity in science education, perhaps serving as a resource for potential future improvements in outreach projects. This, at least, is my hope.

Main Body of the Text

This document is organized into several sections to guide the reader through the project's development. First, the *Elastic Sheet Model* is introduced as a pedagogical analogy for gravity, providing the foundation for understanding the simulation. After discussing the significance of models in general relativity, I explain the motivation for coding the application in Python.

The following sections introduce two well-known physical laws — Hooke's law and Newton's second law — and explain how they can be used to compute the movements of oscillators. These oscillators, in turn, represent a surface featuring waves that move along it, which is rendered on the screen. I briefly discuss three numerical methods that can be used to compute each rendered frame of the animation.

NOTE: NEEDS COMPLETION

Appendices

Appendix A – What can General Relativity Explain?

Our understanding of gravitational waves is based upon the framework of general relativity. I therefore include this section, where some of the historical development and key observational insights of the theory are outlined. I avoid focussing on the mathematical formalism, which lies largely beyond the scope of this work.

Appendix B – The Python Language

The aim of this section is to provide the reader with the minimal knowledge of the Python language required for understanding the application.

Appendix C – The GUI

Outlined here is the background to understanting the graphical user interface (GUI) implemented in the code, which enables control of the parameters required for the simulation. In addition to explaining the GUI syntax and coding approach, I provide a brief code example demonstrating the creation of a *slider object* (an appropriate name for something used in Python!), along with an explanation.

Thesis Style

Since this work is intended for a broader audience than is typical for a project of this kind, the style is occasionally slightly less formal than that of a standard Master's thesis. Nonetheless, I adhere to a recognized *style guide* to ensure consistency throughout. Specifically, I follow the guidelines of the *American Psychological Association (APA)* [1]*, Seventh Edition (2020),* as outlined at https://apastyle.apa.org/products/publication-manual-7th-edition.

In order to distinguish my comments and explanations from any Python code itself, I have formatted all statements that can from part of the Python language in the font: `Consolas 11`.

About the Author

My original motivation for pursuing an MSc in Astrophysics was as a prerequisite for obtaining a teaching diploma (Physics) at the University of Zurich. Prior to this, my professional background was in IT, as a developer on IBM mainframe systems (both in London and, later, in Switzerland). This work was entirely unrelated to teaching or physics.

However, I found the MSc programme both rewarding and intellectually stimulating in its own right – there are few things as fascinating as astrophysics, even if only a select few are fortunate and talented enough to pursue it as a career.

---

[1] Four of the other common styles for citing scientific papers, not used here, are MLA Style (Modern Language Association), Chicago Style, IEEE Style and Harvard Style.

2   Introduction

Certain aspects of gravity, particularly those related to the general theory of relativity, can be represented using various models. Certain aspects of gravity, particularly those related to the general theory of relativity, can be represented using various models. A simple physical model often used in demonstrations in an educational environment is the elastic sheet. A discussion of this model is also presented in Kersting & Steier, 2018. A slightly different perspective is provided in Bayraktar, Gudukbay, & Ozguc, 2007 which features a discussion about how to animate cloth. This also served as a source of inspiration for this project.

2.1   The Elastic Sheet Model

Figure 1

Elastic sheet demonstration of relativistic gravity by Dan Burns, Los Gatos High School, California, United States

Note. Author's screen shot from the video,
https://www.youtube.com/watch?v=MTY1Kje0yLg

In its physical form the elastic sheet model consists of a fabric surface, usually made of rubber or spandex[2], stretched upon a circular or rectangular frame. A metal sphere or an object like a billiard ball or snooker ball is then placed upon the surface and deforms the sheet because of its weight. If additional, smaller, spheres are placed on the warped surface and given an initial motion in any direction, their paths will become curved due to the distortion of the sheet from the heavier mass or masses. This often results in an (at least partial) "orbit" around the larger sphere.

The setup can be used, for example, to approximately demonstrate the gravitational effects of our Sun upon one or more orbiting planets (as the video still of the demonstration, above, shows). Another reference of interest is Middleton (2014), which examines the theoretical and experimental analysis of circular-like orbits formed by a marble rolling on a warped spandex fabric.

If the model is meant to refer, more specifically, to general relativity, the sheet is supposed to represent not space but *spacetime* (three dimensions of space and one of time), the spheres representing masses within that spacetime.[3] The larger the mass of spheres used in the model, the more pronounced the local sheet deformation (distortion) around them, hence the greater the masses being represented. The problem is that since the sheet is merely a two-dimensional object

---

[2] Spandex is a synthetic fabric prized for its elasticity. The term refers to polyether-polyurea copolymer fabrics that have been made with a variety of production processes. The terms *spandex, elastane* and *Lycra* are synonymous. The only one of these, slightly surprisingly, that is a brand name, is *Lycra*.

[3] The Earth's and the Sun's masses are, to two significant figures, $6.0 \cdot 10^{24}$ kg and $2.0 \cdot 10^{30}$ kg, respectively.

embedded in our three-dimensional space, it is limited to representing *only two of the four* spacetime dimensions.

Some physical elastic sheet models include features such as the gravitational wave production from two centrally orbiting bodies, for example, Mould (2016). This is a key inspiration for this project.

Figure 2

Mechanical simulation of gravitational waves

Note. Author's screen shot from the video,
https://youtu.be/dw7U3BYMs4U
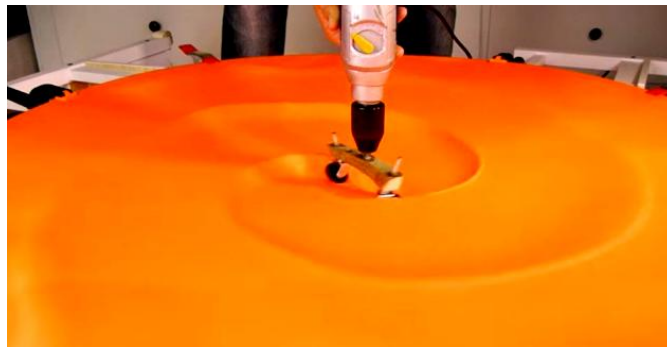
A similar video clip can be seen at *Demonstration of Gravitational Waves on a Spandex Universe - LIGO - Caltech* (2016), which uses similar materials for its demonstration.

My work extends the use of real-world materials to replicate gravitational phenomena, as demonstrated by Mould and others, by using these these concepts entirely in a software-based environment.

The additional motivation for this decision, apart from not needing to construct anything physical, was that my representation was intended to be
• Closer to physical reality in certain respects (not fully defined at the outset of the project).
• Designed to incorporate useful parameters that can be adjusted at the start or during runtime, making the simulation more versatile than physical models.

Since my simulation was intended to be a representation of reality, it is worthwhile briefly discussing the use of models as a pedagogical tool in science.

A simple animation of two orbiting bodies losing energy in the form of emitted gravitational waves can be found at Hilborn (2017). A slightly more sophisticated, aesthetically pleasing video clip (no date) is available at https://www.ligo.caltech.edu/video/gravitational-waves.

2.2   The Role of Models in Science Teaching

In physics, a model (or idealized model) is a simplified version of a physical situation that necessarily omits some characteristics that are present in reality. Writing about models used in science communication, Pössel (2018), elaborates: "A (teaching) model … is a concrete or abstract entity which represents *some* [my emphasis] of the structure … for the purpose of teaching…"

Every idealised model thus represents a *limited set of features* of a physical phenomenon. The skill is in knowing which aspects are to be omitted so that it

• is not a misleading representation
• represents the underlying reality in a useful way.

Models can exist in various forms, in several media. They can be
• physical (including static ones produced by modern 3D printing)
• mathematical
• phenomenological
• visualizations, in the form of a video or computer simulation.

Other examples include physical scenarios, such as people in swimming pools, to visualize concepts [(Cardiff University, n.d.)](#). This rather creative endeavor could also be regarded as a model.

There are several cognitive studies that attempt to reveal how we view models that represent scientific concepts. The age-range of participants in this research area spans that of school-age children (for example, [Baldy, 2007](#), [Postiglione, 2021](#)) to undergraduate students learning general relativity at university ([Bandyopadhyay, 2010](#), [Watkins, 2014](#)). One important result of the study by Postiglione, concerning general relativity, is that young people tend not always to understand that *all* masses deform spacetime, not just very large masses such as planets and stars. In principle, this includes everyday objects such as houses, cars, people, cats – and the coffee cup sitting on the desk at which I'm writing these words.

Another important point is that a more sophisticated target audience might make the judgement that a particular model's main features are always more important than those aspects of reality that are under- or unrepresented by it (provided they happen to know, additionally, what they are). So an educationalist needs to bear this in mind.

Some attempts have even been made to suggest how general relativity might be taught in the secondary school classroom setting, i.e. prior to tertiary education. One example is [Farr, Schelbert, and Trouille (2012)](#).

2.3   Why Simulate Gravitation and General Relativity?

> *"Space-time tells matter how to move; matter tells space-time how to curve."*
> – John Wheeler

The general theory of relativity was published by Albert Einstein in 1915 ([Einstein, 1915](#)) and is a refinement of Newton's law of universal gravitation ([Newton, 1687](#)). It provides a view of space and time that is conceptually and mathematically radically different, however, from the older *Newtonian* view. Contrary to our intuition, gravity is now regarded, at least by physicists, less as a force than as a *geometric property* of space and time[4] (more accurately, of four-dimensional spacetime). It is thus the curvature or deformation of *spacetime* that governs how matter moves within it, which the aphorism by Wheeler accurately encapsulates.

---

[4] Although this is hardly consoling for anyone falling to the ground from just a few metres!

General relativity is a term with which the public has become more and more familiar with over recent decades. One of its many predictions is that of waves – gravitational waves – moving, at the speed of light, through empty space. The first confirmed observation of such waves was made only relatively recently, however, in 2015. It thus took a century before technology caught up with one of the predictions contained in the theory and was able to test (confirm) it.

A book that I found very useful for initially learning about this theory is [Schutz (2009).](#) The text is suitable for advanced undergraduate physics and is written in an accessible style, unlike many other texts on this challenging subject. However, an understanding of general relativity (GR) at this level is not required to understand my project.

2.4   Understanding the Difficulties of General Relativity

"*General relativity has a reputation for being very difficult;*
I think the reason is... that it's very difficult".
– Leonard Susskind[5]

Unfortunately, in spite of many decades of effort by practitioners and educators in making the theory accessible to a general audience, general relativity's daunting reputation has hardly diminished since its inception. Susskind has a point.
 The obstacles that the theory presents are numerous and (mainly) still valid today:

- General relativity is *conceptually* challenging, featuring abstract concepts like the curvature of spacetime, *geodesics* and *tensors*. With the exception of the first term, these are not part of the vocabulary of most people.
- It uses *unfamiliar notation* (something called the *Einstein summation convention*), which has to be mastered early on.[6]
- It requires advanced undergraduate-level understanding of several branches of mathematics, in particular of *differential geometry*.
- Finally, once the theory has been grasped – by working through and understanding the derivations leading from a simple curved 2D surface all the way to the so-called *Einstein field equations* and to finally obtaining a useful equation called the *Schwarzschild metric* – there is a new learning curve. This occurs when we want to apply the theory to examine real astrophysical situations. At this final stage, one of the additional techniques to learn and apply is called *numerical relativity*, which makes use of modern computing techniques to solve these types of problems.

At first glance these issues would appear insurmountable for science communicators wishing to explain the theory to a broad segment of the public. However, many physical features of general relativity can indeed be presented in a simple and phenomenological (qualitative or semi-quantitative) way. This is achieved through the use of models, which is itself a justifying aspect of this project.

---

[5] https://www.youtube.com/watch?v=cyW0LWEACfI

[6] Other notational schemes exist, but do not necessarily make life easier for the student, as Susskind emphasises at the beginning of his recorded video lectures on general relativity.

3   On the Choice of Programming Language

Having decided on the *choice of model*, the next stage of planning was to determine *which programming language* (or languages) the simulation would be implemented in. My own area of expertise has been mainframe computer languages used in business. Mainframe programming languages such as COBOL and PL/1 are clearly unsuitable, however, not least because the implementation has to be able to run on modern devices such as laptops. The world of object-oriented programming seemed enticing but presented too steep a learning curve for an MSc project, especially for someone with no previous knowledge of such methodology.

3.1   Python

Python was designed in the late 1980s by Guido van Rossum at the CWI (Centrum Wiskunde & Informatica) and developed there by him and others. It was first released to the public in 1991. It is an open-source, high-level[7], object-oriented, programming language. However, as the previous paragraph makes clear, I do not explicitly use any of Python's Object-Oriented (OO) capabilities. Additionally, Python is an interpreted language, which means that there is no compilation step: The advantage of this is a reduction of the timescale for each iteration of any *edit-test-debug cycle.*

It was decided then, provisionally, to implement the software using Python. The initial cost-benefit consideration was:
Pros:
- straightforward to learn
- relatively uncomplicated to code and
- simple to read (bearing a closer similarity to written English than most alternatives)
Cons:
- perceived lack of speed.

At this stage, it was unclear whether the language alone would provide sufficient performance. Therefore, alternative programming languages – either in addition to or instead of Python – remained a possibility. For further details, see Appendix B for a discussion of Python.

4   Initial Development Ideas, Using Basic Physics

In secondary schools, one particular physical model sometimes used to visualise waves in physics lessons is called, simply, a *wave machine*. It consists of a series of connected rods, each of which is linked to a spring. The springs are coupled to allow the oscillations of one rod to influence adjacent rods, creating a chain reaction that illustrates wave propagation. Several modules can be linked serially, to produce a longer set of oscillators.

---

[7] A high level language is a programming language that enables a fairly straightforward implementation of an algorithm into computer code. High level languages allow the writing of programs which are almost independent of the type of computer used to run them. Examples, apart from Python, include Fortran and Pascal. In contrast, a low level language is *machine-oriented* because the code must be specified in terms of the capabilities and specifications of the processor. Examples include assembly language and machine code.

Figure 3

Wave machine used at some secondary schools.

Note. Source https://www.leybold-shop.com/wave-machine-basic-module-1-40120.html

I began development by creating a software implementation of such a system of coupled oscillators, producing waves in one dimension, as *proof of concept*. My toy model (Gedankenmodell) consisted of several rings or beads able to move freely and frictionlessly[8], in the *y*-direction (up and down). These were connected by identical springs. The forces, qualitatively, are represented by the yellow arrows.

Figure 4

First concept of coupled oscillators, 1D.

Immovable
end points
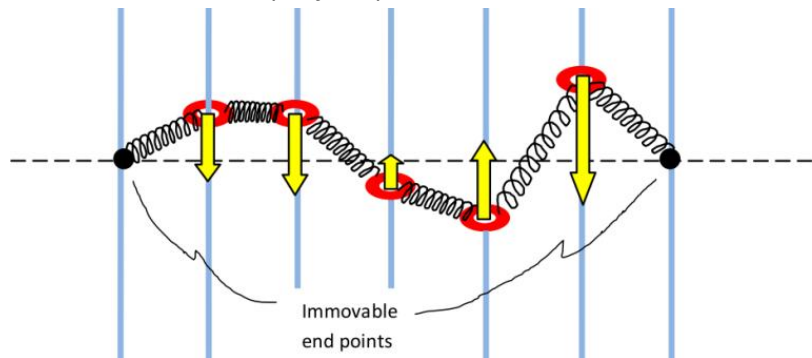
Note. Author's figure

A quantitative result is then obtained by calculating the force on each ring/bead, by considering the sum of the two adjacent spring forces via Hooke's law. The force (along the y axis) is given by

$$F_i = F_{i,i+1} + F_{i,i-1} \tag{1}$$

where $F_{i,i+1}$ represents the force on bead/ring *i* due to the next spring, and $F_{i,i-1}$ that of the previous one. Using Hooke's law then gives:

$$D \cdot \Delta\ell_{\,i,\,i+1} \cos (\theta_{i+1}) - \Delta\ell_{\,i,\,i+1} \cos (\theta_{i-1}) \tag{2}$$

where
- *D* is the spring constant,

---

[8] No other forces, including gravity!

- $\Delta\ell$ is the length extension or contraction of the spring between *i* and *i+1*, which is assumed to behave symmetrically in terms of the magnitude of force produced, regardless of whether the result of a contraction or expansion,
- θ is the angle between each spring and the vertical (*y*) axis.

Once this model had been coded (in Python) and tested, it was extended from one dimension to two. By creating a two-dimensional grid of oscillators, I had now achieved, however rudimentarily, a software representation of the elastic sheet mentioned in the introduction. Conceptually, when viewed *from diagonally above*, a small part of the elementary grid of coupled oscillators would resemble figure 5.

Figure 5

Concept of coupled oscillators, 2D

Note. Author's figure

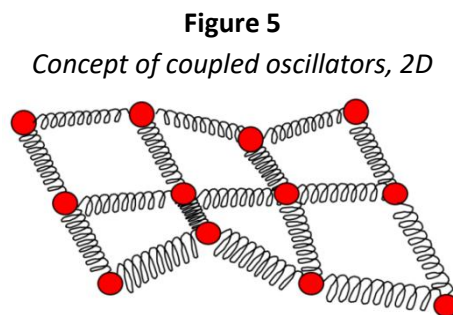The difference between the two was that my model was discretised (a mass concentrated at each coordinate point of the grid) rather than being a continuous, smooth entity. However, provided the sheet resolution (in effect, the grid size along each of the two axes) were sufficiently large, the elastic sheet would be well approximated in my representation.

Since the number of force calculations is $N_{grid}{}^2$ we would, for a 200 * 200 grid, obtain 40 000. If we wished to render the movement of waves along our surface at, say, 25 fps (frames per second), we would require one million force calculations each second. This is no challenge for any modern computer (even my laptop machine). The problem now lies with the implementation language, Python. As an interpreted language, it runs significantly slower than all lower-level, compiled languages (like C/C++), mainly because of the *overhead of its interpreter.*

Calculating forces alone, however, would not be sufficient to produce any movement of the oscillators making up the sheet surface, as additional information is required to be generated and iteratively used. For each oscillator, the following variables must be continually tracked and indexed:
- acceleration
- velocity
- displacement.

4.1   Hooke and Newton[9]

The extension and movement of the oscillators at any particular instant in time can be calculated by using a combination of Hooke's law and Newton's second law.

---

[9] Ironically for the title of this subsection, Newton and Hooke, who were contemporaries and knew each other, didn't get along at all.

Hooke's law states that the force due to the deformation (change of extension) of any ideal (perfect) spring is given by

$$F = -k\Delta s, \tag{3}$$

where k (not to be confused with the k used in the Runge-Kutta algorithm discussed in 5.3) is the spring constant and Δs is the distance the spring is stretched or compressed. The variable F here is called the restorative force. Its direction is always opposite to the direction of the spring's displacement Δs (hence the moniker, *restorative*).

If the sheet is required to be undulating in some fashion in order to produce transverse waves (like water waves), then the freedom of movement for the oscillators could be chosen to be, for example, exclusively in the vertical direction. The basic equation relating acceleration and displacement for any oscillator would then, if the z axis is taken as the vertical axis coordinate, be given by

$$dv_z/dt = -kz/m \tag{4}$$

The combination of equations (3) and (4) for our coupled oscillators leads to a first order differential equation (ODE). The vast majority of such equations cannot be analytically solved (solved by hand); their implementation has to be done numerically, on a computer, which, in effect, is the main point of this project.

The key idea for the calculations is to perform something called an *iterative temporal discretisation.* Unlike in pure mathematics, our computer code cannot truly use infinitesimals, so we replace the mathematical interval **dt** with **Δt**. This entity will be referred to as the *timestep*, although it does not directly correspond to either wall-clock time or CPU time: It refers instead to the integration step granularity, which determines the accuracy of the numerical solution. The aim is to apply as small a value for this interval as possible to maximise accurracy, whilst simulataneously ensuring that the rendering animation still runs at a visually convincing speed and framerate. As is often the case in computer science, it represents an optimal compromise.

5   Numerical Integration Methods

There are several possibilities for the choice of numerical intergration algorithm, since various methods can be applied to the scenario of coupled oscillators. Again, a decision had to be made between options. Each has its own strengths and weaknesses.

5.1   Euler (first order accuracy)

This method updates both position and velocity, with the updates considered simultaneous.

$$x_{n+1} = x_n + v_n\Delta t \tag{5}$$
$$v_{n+1} = v_n + a_n\Delta t \tag{6}$$

where x and v represent position and velocity at step n.

The Euler method is seen as simultaneous because, specifically, equation (5) uses the old (previous) velocity, rather than the newly computed value, ensuring that the two updates are not

interdependent within the same timestep. In reality the steps are indeed calculated sequentially on the computer.

5.1.1 Physical Conservation Laws

Two important considerations for all algorithms that represent a physical reality (recalling that we are simulating waves on a physical elastic surface consisting of oscillators) are the conservation laws of physics.

Energy Conservation

The Euler method generally does not conserve energy. It leads to a *drift* (over- or underestimation) in the computed total energy of the system after many iterations. This is due to the assumption of a constant acceleration over time (within each timestep discretisation) which produces inaccuracies in the new velocity and new position for each oscillator. The velocity inaccuracy leads, in turn, to incorrect values for the kinetic energy; the position error leads to incorrect values for the potential energy. This can be seen, specifically, from the equation for our mass-spring system of oscillators.

$$E = E_{KIN} + E_{POT} = \frac{1}{2}mv^2 + \frac{1}{2}kx^2 \tag{7}$$

Momentum Conservation

Due to the velocity drift/inaccuracy, the momentum, $p = mv$, is not conserved either.

Figure 6

Graphical representation of the Euler method
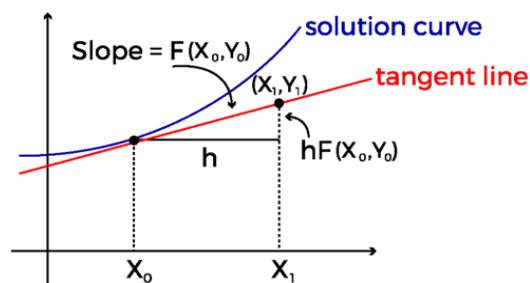
Note. Image source:
https://calcworkshop.com/first-order-differential-equations/eulers-method-table/

5.2 Leapfrog (second order accuracy)

This involves calculating the position and velocity at discrete time steps with a staggered pattern.

Figure 7

Graphical representation of the Leapfrog method

Note. Image source:
http://cvarin.github.io/CSci-Survival-Guide/leapfrog.html
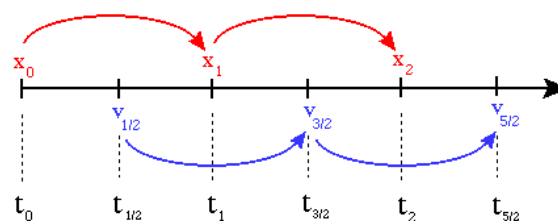
The steps are

- Update of position: The position is updated at integer time steps using the updated velocity from the previous step.

$$x_{n+1} = x_n + v_{n+1/2}\,\Delta t \tag{8}$$

- Update of velocity: The velocity is updated at half-integer time steps.

$$v_{n+3/2} = v_{n-1/2} + a_n\,\Delta t \tag{9}$$

Energy Conservation

Because position and velocity are symmetrically, Leapfrog integration maintains the time-reversal symmetry of the system. This helps in preserving the total energy over longer simulations, reducing the numerical drift in energy compared to the simple Euler method.

Momentum

This is also well conserved, for same reason that velocity is.

5.3 Leaping to the Runge-Kutta 4th-Order Method (RK4)

The Runge-Kutta 4th-order method (RK4) is a numerical technique for solving ordinary differential equations, providing a fourth-order approximation, as reflected in its name. (Intermediate-order methods, which are infrequently used, are not covered here.)

Figure 8

Graphical representation of the RK4 method.
Here, h has an identical meaning to Δt in the text.
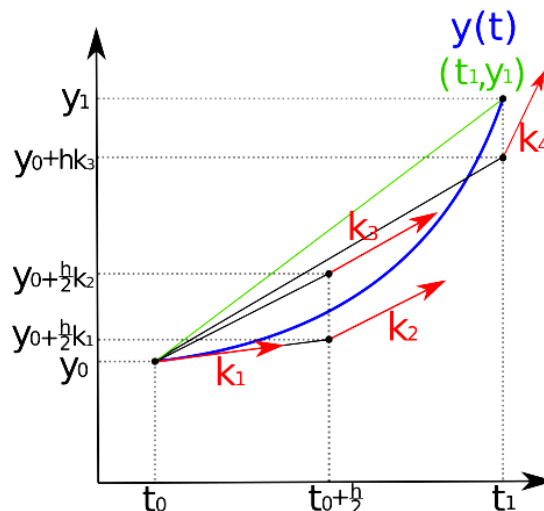
Note. Image source:
https://lowebms.readthedocs.io/en/latest/_images/RK4.png

It computes four slopes, that are used as intermediate values. Generally, an indexed letter k is used for these:

$$k_1 = f(t_n,\, y_n)$$
$$k_2 = f\left(t_n + \frac{\Delta t}{2},\ y_n + \frac{\Delta t}{2}\,k_1\right)$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, \quad y_n + \frac{\Delta t}{2} k_2\right)$$
$$k_4 = f\left(t_n + \Delta t, \quad y_n + \Delta t\, k_3\right)$$

The solution is then calculated using

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{10}$$

We can see that the computational requirements for the RK4 method are significant: four function evaluations per time step compared to Leapfrog's two. Accuracy comes at a cost.

Energy conservation

RK4 does not inherently conserve energy.

Momentum conservation

Not fulfilled either, but better than Leapfrog.

5.4   Comparison of the Three Methods

Method	Energy Conservation	Error Behaviour	Remarks
Euler	Poor. Error $\propto \Delta t$	Energy drifts exponentially or linearly.	Not suitable for large number of time steps or long-duration simulations.
Leapfrog	Very good. Error $\propto \Delta t^{\frac{1}{2}}$	Energy oscillates around the true value.	Balances simplicity with long-term accuracy, making it ideal for oscillators.
RK4	Moderate Error $\propto \Delta t^4$	Energy drifts slowly (cumulative).	Introduces slow energy drift over time. Preferred for systems where energy conservation is less important but high accuracy is needed in the solution.

6   Optimizing Rendering Speed (Part I): Numpy

As the number of coupled oscillators representing the simulated surface grew to approximately 10^4 elements, and one of the three numerical methods discussed was needed to compute their positions, accelerations, and velocities, improving the execution speed became essential due to increasingly unsatisfactory performance.

A useful technique, for large datasets (data aggregates), is to perform operations on *entire arrays* instead of working with one element at a time. Ways to improve efficiency and execution speed for such arrays include:
- Enforcing contiguous memory allocation: Ensuring that each array element is stored in adjacent memory locations to optimize access times.
- Standardising data types: Enforcing a single data type across the entire array, instead of allowing elements to have varying types, which can slow down processing.
- Performing array iteration in the background using precompiled code written in a faster language to boost processing speed.

This final point highlights a technique which avoids explicit looping and thus enables faster processing. It is known as vectorization.[10] NumPy is the extension module (library) for Python which provides just such functionality.

Implementing and running Python with Numpy did help. However, it soon became clear that other methods, libraries or interfaces to other (faster) languages would need to be used in the next stage of development in addition to Numpy. Although in the end, Numpy was not used in this implementation, I include the following references for interested readers: NumPy, n.d. and NumPy: The absolute basics for beginners, n.d.)

7 Optimizing Rendering Speed (Part II): Mayavi

Mayavi is an open-source, cross-platform, 3D scientific data visualization package. It was created in 2001 by P. Ramachandran.

Mayavi, working together with Python, seemed a good choice at this stage. This is because
- its interface can visualise the 3D data described by NumPy arrays[11] and
- since it uses openGL for its rendering[12].

However, although the results looked visually quite pleasing[13], as shown in a still from one of my video clips, the frame rates were too low for a smooth animation.

Figure 9
Early screenshot using Python with Numpy and the Mayavi package.
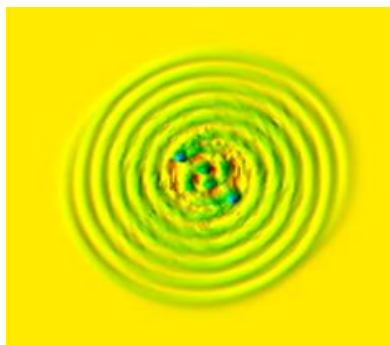
Note. Author's image

As NumPy and Mayavi were only an intermediate stage in the development of this application, and was soon to be abandoned in favour of another solution, detailed discussions about it them are omitted from this thesis.

---

[10] or vector computing, or array computing.

[11] As, incidentally, can Matplotlib. Unfortunately, Matplotlib cannot access GPUs to increase its performance.

[12] This is because it uses the *Visualization Toolkit (VTK)* for its graphics, which itself uses openGL.

[13] A particularly nice feature of Mayavi is its sophisticated GUI which allows rotation of viewing field of view in both directions, as well as a zoom option.

8 Reducing Reflections

8.1 Viscous Damping (Dashpot Damping)

In any numerical simulation of waves, the behavior of oscillators at and near the grid boundaries is problematic. Waves are reflected at such borders, even when there is no medium or grid on the other side, due to the *abrupt termination of the simulated domain.*

In this work, reflections are problematic for these reasons:

- **Energy increase**: As energy is continuously introduced into the oscillators representing the surface (by the two orbiting bodies), we require this energy input to be balanced by dissipation at the four borders.

- **Lack of realism**: In any case, it is physically unrealistic to expect wave reflections (gravitational or otherwise) occurring in empty space.

- **Appearance**: A visual simulation should look smooth and appealing.

A simple and effective method to reduce border reflections in simulations is to apply *viscous damping* near the boundaries. This technique absorbs energy from the oscillations of the surface elements, minimising – but almost never eliminating – unwanted reflections.

Additionally, we fix the layer at the border of the grid with the condition:

vel = 0 and

pos_z = 0, whilst pos_x and pos_y take the values of the current array value indices for the surface coordinates.

We extend the equation of motion with the term on the far right

$$m \frac{d^2x}{dt^2} = -kx - c\frac{dx}{dt} \tag{11}$$

where, as is typical,

- m is the mass,
- k is the spring constant,
- x is the displacement,

and now, additionally, c is the damping coefficient.

My implementation imposes a reduction of both the velocities (all three components) and, in addition to the above equation, the position vector (height) component. It does this for those oscillators lying within the specified boundary regions, the intensity of damping increasing as each of the four edge boundaries of the grid is approached (as seen from the grid centre). The damping increase can be linear or exponential. The latter requires more careful tuning, but since the linear version worked well, I adhered to this solution.

The mathematics for attempting to remove reflections at simulation boundaries can be more complicated than the above *Ansatz*. Examples that were examined during my research are to be found in Bao, Hatzor, & Huang (2012), Carcione, J. M. (1994) and Ehgquist, B., Majda, A. (1977). I found that my solution worked sufficiently well with my setup, making the content of these papers beyond the scope of this work.

The boundary damping factor found to be optimal in the implementation, assuming a number of damping layers proportional to the grid size (integer division: grid_size//20) turned out to be around 0.05. Clearly, a larger fraction of the grid outer layers could always be used for this, but the value chosen (in effect, one-tenth of the grid extent along each of the two axes) seems appropriate. Incidentally, utilising too many layers to reduce the reflected waves would negatively impinge upon the model's overall appearance.

The same damping factor and layer depth were applied to reduce both the velocity and position vectors of the oscillators near the borders. Varying these factors independently and refining them further was deemed an unnecessary use of resources.

9   Implementing a Graphical User Interface (GUI)

To enhance the application and provide more control over the animation, it was now necessary to implement a Graphical User Interface (GUI). This would allow some parameters to be set by the user either at the beginning and/or during the run as well as to start, stop and pause the run. My goal was to incorporate significantly more functionality than that offered by simpler mechanical models, such as those developed by Mould (2016) and others.

9.1   GUI Library Selection - Tkinter

Some of the more popular Python GUI libraries, along with their official homepages, include:
*   Tkinter
    https://wiki.python.org/moin/TkInter
*   PyQt
    https://riverbankcomputing.com/software/pyqt/intro
*   Kivy
    https://kivy.org/
*   wxPython
    https://wxpython.org/
*   Dear PyGui
    https://hoffstadt.github.io/DearPyGui/

I opted for Tkinter[14], because it is probably the simplest way to develop visual elements for Python[15]. It is already included with standard Linux, Microsoft Windows and macOS installs of the language, which is clearly another advantage. Although the four other libraries mentioned offer more advanced features and capabilities, these were secondary to the project's needs. Simplicity and avoiding what seemed to be a steep learning curve were the primary considerations.

GUI components are often referred to as *widgets*.[16] Those used in this project were
*   windows,
*   sliders,
*   buttons,

---

[14] *Tkinter* comes from *Tk interface*. It was written by Steen Lumholt and Guido van Rossum (once again!).

[15] It is cross-platform, so the same code works on Windows, macOS, and Linux.

[16] A widget is also a device placed in an aluminium container of beer to manage the characteristics of the beer's head. The original widget was patented in Ireland by Guinness.

- labels
- text boxes.

The user interface is displayed at the start of the program and remains visible throughout the entire animation sequence. Certain parameters were designed to be modifiable not only before the animation begins but also during the rendering process.

The development of the GUI window in its current form was somewhat laborious, as several features evolved during development and testing, new widgets and functionality were added, and some features were removed: Testing had to be extensive, as some parameters influenced the computation of the rendered surface, while others affected its visual appearance.

Thoughtful decisions had to be made regarding the positions and relative sizes of the elements within the window, as well as the proportion of the total screen space the whole GUI would occupy (final choice: 1/5 of the screen space). One early decision was to keep the GUI fixed (non-movable) and non-resizable to avoid overcomplicating the user experience.

9.2   Functionality of the User Interface

Sliders 1 to 4 influence the computation of the simulation:

1) **First Sphere Orbital Radius** (in metres)
   This slider selection allows an initial orbital radius for one of the two spheres to be selected. Units are the same as grid spacing (not necessarily in screen pixels) with a scaling factor under the hood to reasonable astrophysical distances for such a compact system as two closely orbiting neutron stars, of the order 100 km or so. The second radius cannot be set directly but is computed using the values of the two masses together with the first radius.

2) **Number of rendered spheres**
   This option allows one or both rendered spheres to be removed from the simulation in order to examine the underlying surface structure at the position of the perturbation. The computations for the oscillators representing the surface remain unaffected.

3) **Mass of the first sphere** (in Solar mass units).
   The masses can be either the same or different. If they are different, the radius and depth of the perturbations are scaled accordingly.

4) **Mass of the second sphere** in Solar mass units.

Sliders 5, 6, and 10 adjust the appearance of the surface:

5) **Vertical Scale** slider. Scaling the vertical aspect is useful for examining finer details of the rendered surface that may not otherwise be visible. This option was also employed during testing of the boundary damping to exaggerate reflection artifacts, which were then reduced in a subsequent code iteration. The default value is arbitrary, chosen to balance an aesthetically pleasing rendering with patterns that resemble those produced by mechanical elastic sheet models. The computations for the oscillators representing the surface remain unaffected.

6) The **Smoothing** slider removes high frequency artifacts, so that the main waves are more clearly visible.

Figure 10

The Graphical User Interface.
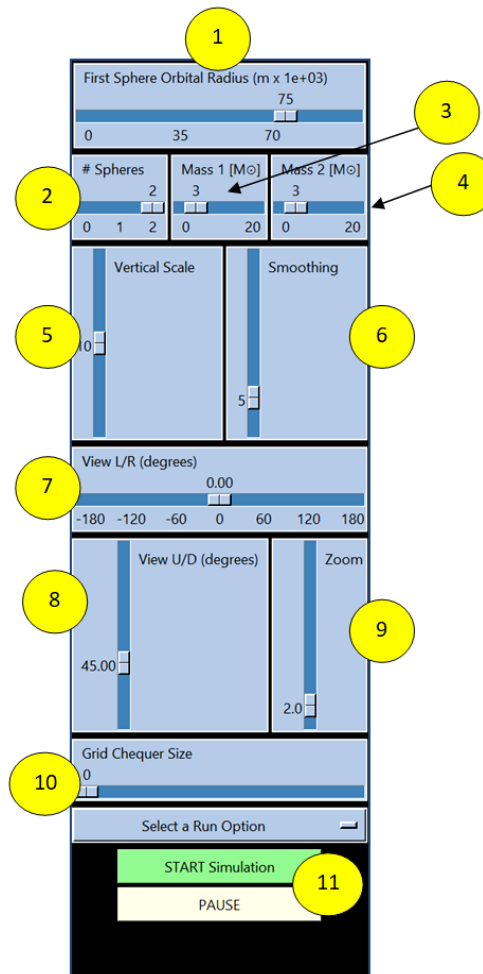
Note. Author's image

10) The rendered surface can be overlaid with a **two-colour grid**, creating a chequerboard pattern if desired.

Adjustment of the viewing position:

7) The view can be rotated 180° in either direction, allowing a full 360° view of the surface.

8) The view can be tilted up or down from -45° to +90°. This adjustment can also be made using the mouse.

9) This slider allows zooming in and out, enabling a transition from a bird's-eye view to closer examination of the rendered details.

Adjustments 7) 8) and 9) can also be made using a mouse.

11) **Start/Stop and Pause**
The simulation can be started or stopped using these two buttons, with the option to pause. While paused, the height, smoothing, and viewing settings can still be adjusted.

10 Threading in Python

10.1 Motivation

The need to explicitly program threading arises from the fact that the Tkinter GUI, now being the interface of choice, operates in an infinite loop once started. After the interface becomes visible – and the user adjusts parameters if needed – the main part of the script must execute tasks such as:
- Repeatedly computing the oscillator positions over time, and
- Rendering the resulting surface as an animation on the computer screen.

10.2 Overview

In computer programs, threads allow multiple tasks to be performed concurrently within a single process, sharing resources such as memory. Each thread is a separate flow of execution[17].

Multiple threads within a process share the same *data space* with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes. Python threading enables different parts of the program to run concurrently: When the Python script is run, it starts an instance of the Python interpreter that runs the code in the main thread, which is the default thread of a Python process.

Python's threading library module allows thread creation and management. Here are some of the methods:
- `start()`   starts a thread by calling the run method.
- `join([time])`   waits for threads to terminate.
- `isAlive()`   checks whether a thread is still executing.
- `getName()`   returns the name of a thread.
- `setName()`   sets the name of a thread.

My application needed only the first of these.

In Python, stopping a thread is straightforward: there is no stop() method; the threading module does not provide a built-in way to forcefully stop a thread. Instead, a common way this is dealt with is by using a flag.

Three additional functions in Python's threading module, useful for testing purposes, are
- `threading.active_count():`   Returns the number of active thread objects.
- `threading.current_thread():`   Returns the current thread object executing the caller's code.
- `threading.enumerate():`   Returns a list of all currently active thread objects.

10.3 Daemon Threads

Mentioned here only for completeness and because of the intriguing name, *daemon threads* are threads that run in the background In Python, typically doing background tasks such as *listening* for

---

[17] However, the different threads do not actually execute at the same time: they merely appear to do so.

input. At the end of a run any daemon threads that are still active will shut down. This project does not use them, so they are not discussed further.

10.4 Sharing Data between Threads

An important feature of threads is that they can exchange data (but do not have to do so). Sharing of data between threads became a necessity once the decision had been made to utilise user input from the Tkinter GUI. This was because the GUI runs in an infinite, self-contained loop which needs to be in the main thread but the computation and rendering need to be separately controlled (started, paused, stopped, restarted), requiring a second thread.

When two or more threads access shared data concurrently, the outcome of their execution may be unpredictable, because the operating system can swap which thread is running at any time. Any updating of shared data resources is then not guaranteed to work because one thread could be reading a variable exactly at the same time as another is attempting to update it.[18] This undesired behaviour is know as a *race condition.* Python avoids this by means of a *Lock.* Many other programming languages also employ this concept, in part using different terminology. The figure below shows an example of using a lock on a Python data dictionary.

Figure 11

Example of the Python locking mechanism used to ensure integrity of data shared by more than one running thread.

```
# Extract the GUI values and place in the shared_slider_data dictionary.
shared_slider_data_from_gui(shared_slider_data)
# The use of the shared data dictionary independent of these processing
# variables ensures thread safety.
with shared_slider_data['lock']:
     first_orbital_radius  = shared_slider_data['first_orbital_radius']
```

Note. Author's own code.

11 Optimizing Rendering Speed (Part III): Taichi ☯

This programming language[19] provides a way for Python to execute high-performance numerical and/or graphical computations, which was precisely the requirement for improving the rendering performance – framerate – for the large system of coupled oscillators implemented in this project. Taichi was created by Yuanming Hu at Tsinghua University in Beijing in 2018. It uses almost the same syntax as Python but enforces stricter rules regarding what is permitted, in order to optimise performance (and parallelism).

---

[18] In fact, the situation is worse than that because even a simple operation (such as x += 1) takes the processor several steps (separate processor instructions) to execute, and a thread could be *swapped out* between the execution of any of these instructions!

[19] The ancient Chinese martial art in which practitioners perform a series of deliberate, flowing motions while focusing on deep, slow breaths is also called *tai chi.*

At the start of a Python program execution, Taichi performs a *just-in-time* (JIT) compilation step, translating the Taichi code blocks into optimized machine code. This compilation step was found to introduce negligible overhead at the beginning of the run and, therefore, does not negatively impact the user experience with the GUI during startup.

Taichi function blocks are differentiated from those of native Python by two types of decorator**:** a *kernel* and a *function* decorator (although both are really functions). The decorators are, respectively, `@ti.kernel` and `@ti.func.` Everything within these decorated functions has Taichi scope. Kernels are responsible for managing and allocating parallelism, while functions operate at a slightly lower level, providing a way to create modular and flexible code in Taichi: They can, for example, be *reused* within multiple kernels.

11.1 Taichi Kernels

These special functions are designed to allow much faster processing of data, either on CPUs or (if available) GPUs. Some of their characteristics are:
- Several kernels may exist in the code.
- They are not allowed to be nested.
- All kernels are mutually independent of each other and are compiled and executed in the same order as they are first called.
- They are called directly by Python code (i.e. from the Python scope) and serve as an *entry point* for Taichi.
- Kernels must take type-hinted arguments and return type-hinted values; this is in stark contrast to Python's dynamic typing system (which is more flexible, but much slower).

Figure 12

Example of a Taichi kernel function

```
@ti.kernel
def model_to_astro_scale(
        model_distance: ti.f64,
        astro_length_scaling: ti.f64
    ) -> ti.f64:
    return model_distance * astro_length_scaling
```

Note. Author's own code.

11.2 Taichi Functions

Key characteristics:
- They must be called by either kernels or other Taichi functions; they cannot be called from the Python scope.
- Nested functions are allowed.
- They do not have to be given take type-hinted arguments and return type-hinted values.
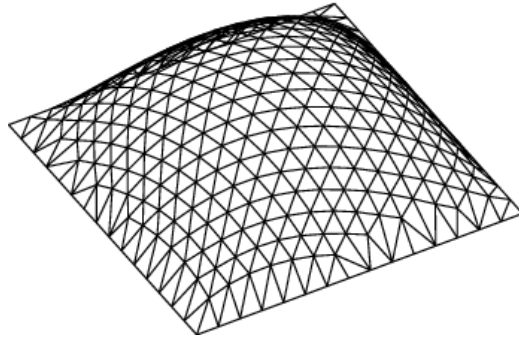
Example
```
@ti.func
def add(a, b):
    return a + b
```

11.3 Triangle Meshes

A mesh is a collection of *vertices* (points in 3D space) and *faces* (flat surfaces which connect these vertices). Usually, triangles are used for the faces, as any surface can be approximated using a large number of them.

Figure 7

A triangle mesh, representing an arbitrary 2D surface.

Note. Source

https://www.semanticscholar.org/paper/A-UNIFORM-TRIANGLE-MESH-GENERATION-OF-CURVED-Katoh-Ohsaki/350325e7f71d2d243317d9a2a30308588d4522e1

Each pair of adjacent rows and columns in the matrix represents the 3D position vectors of oscillators – *not* the vertices of the quadrilateral faces themselves. These 3D vectors define the points on the surface that will be (re-)rendered on the screen. The next step is to split the quadrilaterals, formed by these position vectors, into two triangles. This process converts the position vectors into the vertices of the mesh, which are then used for rendering.[20]

12 Optimizing Rendering Speed (Part IV): The Final Frontier – Taichi with GPUs

NOTE: Section still needs to be done.

---

[20] Taichi actually represents the mesh using two separate arrays: An Index Array (or Connectivity Array) which contains sets of three indices, where each set defines a triangle by specifying the indices of the vertices that form it; and a Vertices Array which holds the 3D coordinates of the vertices, representing their positions in space. By separating these connectivity and vertex data, Taichi can render the mesh more efficiently (faster) than by using an array holding all the information.

13 Some Further Physics – The Orbiting Bodies

13.1 Orbital Angular Speed of two Orbiting Masses

We know from Newton's universal law of gravitation, that the force between two masses M_1 and M_2 separated by a distance r is given by:

$$F_G = \frac{G\,M_1\,M_2}{r^2} \qquad (1)$$

where G is the gravitational constant (Newton's constant). The centripetal force between them is

$$F_C = \frac{M_1 v_1^2}{r_1} = M_1\,\omega_1^2\,r_1 = M_2\,\omega_2^2\,r_2 \qquad (2)$$

The centre of mass is defined as the point where the weighted relative position of the masses balances,

$$x = \frac{\sum (m_i\,x_i)}{\sum m_i}$$

So $x_{BARYCENTRE} = \dfrac{M_1 x_1 + M_2 x_2}{M_1 + M_2}$ becomes, on moving the pivot point to the centre of M_1,

$$r_1 = \frac{M_2 a}{M_1 + M_2}\ , \text{ where } a \text{ is now the distance between the two bodies.}$$

Using this, $M_1\,\omega^2\,r_1$ becomes:

$$M_1\,\omega^2\,\frac{M_2 a}{M_1 + M_2}\ , \text{ which we know from (1) and (2) also equals } \frac{GM_1M_2}{r^2}$$

Hence,

$$\omega^2 = \frac{GM_1M_2}{r^2}\,\frac{(M_1 + M_2)}{M_2 a\,M_1} = \frac{G(M_1 + M_2)}{a^3} \qquad (3)$$

We can, in our animation, now incorporate what I shall henceforth call the „astrophysical omega" in our information display window. This omega is based upon the scaled-up (from the grid „toy model universe" to reality) values of the two bodies' masses and their physical separation distance. The value of G used in the software, is, of course, the real value of Newton's constant as determined empirically.

13.2 Power of the Gravitational Waves radiated by a Binary System

$$P = \frac{G^4}{c^5} * \frac{32}{5} \left(\frac{M_1 M_2}{M_1 + M_2} \right)^2 a^4 \Omega^6 .$$

This can also be regarded as the "gravitational luminosity.[21]

13.3 Orbital Decay

The equation for this is

$$\frac{da}{dt} = -\frac{64}{5} \frac{G^3}{c^5} \frac{(m_1 m_2)(m_1 + m_2)}{a^3}$$

where
- r is the separation between the bodies,
- t time,
- G the gravitational constant,
- c the speed of light, and
- m_1 and m_2 the masses of the bodies.

Concluding Remarks

[NOTE: Summary of work,
goals achieved, not achieved,
future enhancements,
overall difficulty or ease of the project.]

---

[21] An equivalent formulation is

$$P = \frac{G^4 (m_1 m_2)^2 (m_1 + m_2) \omega^6 r^6}{c^5}$$

References

- Abbott, B. P., Abbott, R., Abbott, T. D., Abernathy, M. R., Acernese, F., Ackley, K., Adams, C., Adams, T., Addesso, P., Adhikari, R. X., Adya, V. B., Affeldt, C., Agathos, M., Agatsuma, K., Aggarwal, N., Aguiar, O. D., Aiello, L., Ain, A., Ajith, P., … Zweizig, J. (2016). Binary Black Hole Mergers in the First Advanced LIGO Observing Run. *Physical Review* X, 6(4). https://doi.org/10.1103/physrevx.6.041015

- Baldy, E. (2007). A new educational perspective for teaching gravity. *International Journal of Science Education*, *29*(14), 1767–1788. https://doi.org/10.1080/09500690601083367

- Bandyopadhyay, A., & Kumar, A. (2010). Probing students' understanding of some conceptual themes in general relativity. *Physical Review Special Topics - Physics Education Research, 6(2).* https://doi.org/10.1103/physrevstper.6.020104

- Bao, H., Hatzor, Y. H., & Huang, X. (2012). A New Viscous Boundary Condition in the Two-Dimensional Discontinuous Deformation Analysis Method for Wave Propagation Problems. *Rock Mechanics and Rock Engineering*. https://doi.org/10.1007/s00603-012-0245-y

- Bayes, M., & Price, M. (1763). LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S. *Philosophical transactions of the Royal Society of London*, *53*(0), 370–418. https://doi.org/10.1098/rstl.1763.0053

- Bayraktar, S., Gudukbay, U., & Ozguc, B. (2007). Practical and Realistic Animation of Cloth. *CiteSeer X (the Pennsylvania State University)*. https://doi.org/10.1109/3dtv.2007.4379452

- Burns, D. (2012). *Gravity Visualized*. Youtube. https://www.youtube.com/watch?v=MTY1Kje0yLg

- Carcione, J. M. (1994). Boundary conditions for wave propagation problems. *Finite Elements in Analysis and Design*, *16*(3-4), 317–327. https://doi.org/10.1016/0168-874x(94)90074-4

- Cardiff University. *The first detection of Gravitational Waves.* An outreach video produced in and around a swimming pool. https://youtu.be/Lcxt097G4Ps

- Coles, P. (2001). Einstein, Eddington and the 1919 eclipse. *In arXiv [astro-ph].* http://arxiv.org/abs/astro-ph/0102462

- Dyson, F. W., Eddington, A. S., Davidson, C. (1920). A determination of the deflection of light by the Sun's gravitational field, from observations made at the total eclipse of May 29, 1919. Phil. Trans. R. Soc. Lond. A 220, 291 – 333.

- Einstein, A., 1915. Die Feldgleichungen der Gravitation. Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin, pp. 844-847.

- Ehgquist, B., Majda, A. (1977). Absorbing boundary conditions for the Numerical Solution of Waves, Math. Comput. 31, pp. 629-651. https://doi.org/10.2307/2005997

- Farr, B., Schelbert, G., & Trouille, L. (2012). Gravitational wave science in the high school classroom. American Journal of Physics, 80(10), 898–904. https://doi.org/10.1119/1.4738365

- Gilmore, G., & Tausch-Pebody, G. (2022). The 1919 eclipse results that verified general relativity and their later detractors: a story re-told. Notes and Records of the Royal Society of London, 76(1), 155–180. https://doi.org/10.1098/rsnr.2020.0040

- *Gravitational Waves*. (n.d.). LIGO Lab | Caltech. https://www.ligo.caltech.edu/video/gravitational-waves

- Hilborn, R. C. (2018). Gravitational waves from orbiting binaries without general relativity. *American Journal of Physics, 86(3),* 186–197. https://doi.org/10.1119/1.5020984

- Hilborn, R., *BinaryInSpiral using Python GlowScript IDE*. (2017). Glowscript.Org. https://www.glowscript.org/#/user/rhilborn/folder/Public/program/BinaryInSpiral

- Hawking, S. W. (1988). *A brief history of time : from the big bang to black holes*. Toronto : Bantam Books.

- Hubble, E. (1929). A relation between distance and radial velocity among extra-galactic nebulae. *Proceedings of the National Academy of Sciences of the United States of America, 15(3)*, 168–173.

- Kersting, M., & Steier, R. (2018). Understanding curved spacetime: The role of the rubber sheet analogy in learning general relativity. *Science & Education, 27*(7–8), 593–623. https://doi.org/10.1007/s11191-018-9997-4

- Klein, B (2021). *Intro to Python tutorial.* Python-Course.Eu. https://python-course.eu/python-tutorial/

- *Demonstration of Gravitational Waves on a Spandex Universe - LIGO - caltech*. (2016). Youtube. https://www.youtube.com/watch?v=YfSyhcFu_MM

- *NumPy: the absolute basics for beginners — NumPy v1.23 Manual*. (n.d.). Numpy.org. Retrieved July 29, 2022, from https://numpy.org/doc/stable/user/absolute_beginners.html

- McDonald, K. T. (2004). *What is the stiffness of spacetime?* https://arxiv.org/abs/gr-qc/0407036

- Middleton, C. A., & Langston, M. (2014). Circular orbits on a warped spandex fabric. American Journal of Physics, 82(4), 287–294. https://doi.org/10.1119/1.4848635

- Mould, Steve. Visualising gravitational waves. (2016). YouTube. https://youtu.be/dw7U3BYMs4U

- Newton, I. (1999). *The Principia: mathematical principles of natural philosophy*. (Cohen, I. B., Whitman, A. M., & Budenz, J. Berkeley, Trans.) University of California Press. (Original work published 1687)

- The NumPy community. *NumPy v1.21 Manual*. (2021). Numpy.Org. https://numpy.org/doc/stable/index.html

- O'Raifeartaigh, C., & Mitton, S. (2018). Interrogating the legend of Einstein's "biggest blunder." *Physics in Perspective*, *20*(4), 318–341. https://doi.org/10.1007/s00016-018-0228-9

- *PEP 8 – style guide for python code*. (2013). Python.org. https://peps.python.org/pep-0008/

- Postiglione, A., & De Angelis, I. (2021). Students' understanding of gravity using the rubber sheet analogy: an Italian experience. Physics Education, 56(2), 025020. https://doi.org/10.48550/arXiv.2102.04156

- Pössel, M. (2018). Relatively complicated? Using models to teach general relativity at different levels. In arXiv [gr-qc]. http://arxiv.org/abs/1812.11589

- Python Software Foundation (2021). 3.10.1 Documentation. Python.Org. https://docs.python.org/3/

- Ramachandran, P., Varoquaux, G. (2011). Mayavi: 3D visualization of scientific data. Computing in Science and Engineering, Institute of Electrical and Electronics Engineers, 2011, 13 (2), pp.40-51. https://dl.acm.org/doi/10.1109/MCSE.2011.35

- Schutz, B. F. (2009). A First Course in General Relativity (2nd ed.). Cambridge University Press.

- Watkins, T. (2014). Gravity & Einstein: Assessing the Rubber Sheet Analogy in Undergraduate Conceptual Physics. Boise State University Theses and Dissertations. https://scholarworks.boisestate.edu/td/862/

Appendix A – What can General Relativity Explain?

> In this section, I provide an outline of general relativity and an overview of some of the observations in astronomy and cosmology for which the theory provides explanations. The experimental verification of gravitational waves in 2015 has now led to a field called **gravitational wave astronomy**, providing a new "window of information" about our cosmos.

What is General Relativity

General relativity describes how mass concentrations distort the so-called spacetime (our three dimensions of space plus the single dimension of time) around them – and how this distortion affects other objects in their neighbourhood.

Due to the immense scales involved, far beyond those of our everyday experiences on Earth, general relativity excels at explaining large-scale astronomical phenomena. One example is an effect known as **gravitational lensing** by (and of) galaxies and groups of galaxies.

Even at the comparatively smaller scale of our Solar System, it outperforms its "competitor," Newton's theory of gravity, in explaining certain phenomena. An example is the deflection (bending) of starlight passing close to our Sun, as viewed from Earth. This was first confirmed by observations of a solar eclipse in 1919 at made at Sobral, Brazil, and Principe, West Africa[22] .

Figure A1

*An **Einstein ring** created by gravitational lens LRG 3-757*

Note. Source: https://apod.nasa.gov/apod/ap111221.html

Another observation within our Solar System is the precession of the elliptic orbital shape of the planet Mercury around our Sun and (some of) the physics of black holes, where a massive star is

---

[22] The 1919 expeditions (Dyson et al., 1920) are further discussed in Coles (2001) and Gilmore et al. (2020): The latter study reconfirms both the integrity and scientific objectivity of Eddington and the validity of the data obtained at that time. It is unfortunate that Stephen Hawking erroneously states in his 1988 bestseller *A Brief History of Time* (Hawking, 1988) that " … later examination of the photographs taken on [the 1919 solar eclipse] expedition showed the errors were as great as the effect they were trying to measure. Their measurement had been sheer luck … ". This statement is untrue.

collapsing "forever" inward and no longer allows anything, not even light, to escape from its "surface". Indeed, the very idea of a surface has to be carefully defined for such a system.

Figure A2

Cartoon depiction of the precession of the orbit of the planet Mercury around the Sun

*Note.* Not to scale; distances and angles highly exaggerated.

On the largest scales of all – cosmological scales – general relativity **predicts** the expansion of the Universe itself. This was not the prevailing view when Einstein published his paper in 1915, as there was no observational evidence to support it at the time. Furthermore, the incorrect static model would have been favoured by scientists on the additional usual and useful grounds of simplicity and elegance, compared to any "dynamical" alternative (expansion or contraction). This erroneous thinking led Einstein, in 1917, to insert an additional term into his theory called the **cosmological constant**, to allow for just such a static Universe. When, a few years later, it was observationally confirmed by Edwin Hubble (1929) that the Universe is expanding after all[23], Einstein called this addition his "biggest blunder". It appears that this story is not apocryphal but true (O'Raifeartaigh et al. 2018); one can easily imagine the great physicist's frustration upon realising that he had missed the opportunity of using the power of his own theory to correctly predict an as yet unconfirmed feature of the cosmos!

Since the invention of the telescope in the Netherlands, probably in 1608, and Galileo's famous use of his own improved versions in Italy over the following years (more primitive, by today's standards, than many children's telescopes), astronomers have been able to more and more effectively to make use of the information contained in the electromagnetic radiation emitted by sources in the sky. Indeed, the twentieth century saw the construction, for the first time, of telescopes which could produce images using wavelengths other than the visible part of the spectrum, beginning with the longer wavelengths like radio waves – in the 1930s – and subsequently using the infrared, ultraviolet, X-ray and gamma ray regions aswell. However, since the Earth's atmosphere prevents nearly all wavelengths shorter than visible light arriving at its surface, observations in these wavebands cannot be made from ground-based telescopes but only from space. For these observations the equipment is placed aboard rockets or on Earth-orbiting satellites.

---

[23] Swedish astronomer Knut Lundmark appears to have been the first person to find observational evidence for the expansion, in 1924. Three years later, Belgian priest and cosmologist Georges Lemaître presented somewhat more exact observational evidence, correctly stating that the relationship between distance to galaxies and their recessional velocity is a linear one. Edwin Hubble confirmed their findings in 1929 and it is generally he who is credited with the observations confirming the Universe's expansion. (Modified from Wikipedia.)

A well-known example of an astronomical object imaged at different wavelengths is the Crab nebula, the remains of a supernova explosion in our Galaxy recorded by Chinese astronomers in 1054.[24] It can no longer be seen with the naked eye, as its apparent brightness in the sky is around nine times too faint, but it can be seen as a small faint patch of light using a good pair of binoculars.[25] Each of the four images in the figure below contains different information about the nebula. Since humans cannot, in principle, perceive colours beyond the boundaries of wavelengths visible to our eyes, non-optical images in astronomy usually employ "false colours", to represent signal intensity (and to look aesthetically pleasing).

Figure A3

The Crab nebula in different energy regimes (wavelengths)

Note. Source https://ecuip.lib.uchicago.edu/multiwavelength-astronomy/x-ray/impact/07.html
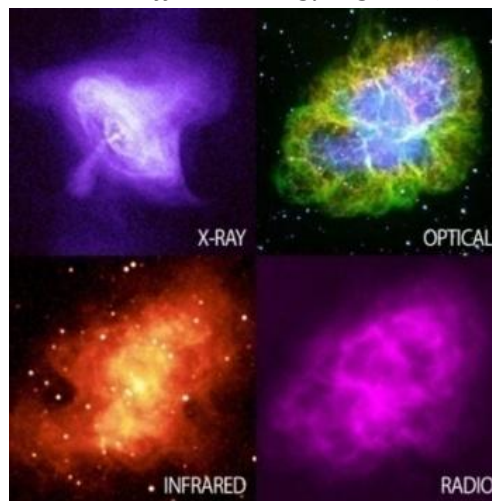
Gravitational Waves

All masses have a constant gravitational field in the region around them. According to general relativity, however, masses that are accelerating produce something in addition: the emission of something called gravitational waves.[26] Accelerated masses produce these waves in a similar way to **accelerating charges** producing **electromagnetic** waves. For example, electrons rapidly oscillating (moving back and forth) along the length of an antenna emit electromagnetic radiation – photons – with wavelengths in the radio band of the spectrum; a radio transmitter.

Both types of energy, electromagnetic and gravitational, can travel through the vacuum of empty space. They do so at the speed of light ($3,00 \cdot 10^8$ m/s).

---

[24] There is no recorded evidence that it was observed from other parts of the world, not even in Europe.

[25] Its so-called **apparent magnitude** in the sky is 8.4. Since the astronomical magnitude scale is **reverse logarithmic** (the brighter an object is, the lower its magnitude number; a difference of 1.0 in magnitude corresponds to a brightness ratio of 2.512) and assuming that the limit of naked eye perception is (in best nighttime observing conditions) magnitude 6.0, the difference of 2.4 magnitudes works out as a brightness factor difference of around $2.512^{8.4 - 6} = 9.12$. This means that the Crab nebula is nine times too faint to be seen with the unaided eye.

[26] Provided that this acceleration is not perfectly spherically or cylindrically symmetric.

As it turns out, general relativity is not the only theory of gravity that predicts gravitational waves; any relativistic theory of gravity will predict them.[27]

On the difficulty of their detection

There are several reasons why gravitational waves (as opposed to gravity itself) are extremely difficult to detect (let alone, measure):

- Their interaction with matter is very weak. The gravitational waves emitted by the movement of everyday objects on the Earth (people, cars, planes) are far too small to be measurable. Indeed, the extreme weakness of gravitational waves makes detection difficult even for sources that are large and massive because the amount of energy emitted in this form by objects is generally extremely small. For example, the power of gravitational waves emitted by a pair of masses m_1 and m_2, orbiting around their common centre of gravity (barycentre) in circular paths of radius r is given (here without derivation) by the following equation:

$$P = \frac{-32}{5} \frac{G^4}{c^5} \frac{(m_1 m_2)^2 (m_1 + m_2)}{r^5} \tag{A.1}$$

Taking numerical values of Newton's constant G and the speed of light c, to three significant figures, we have the values:
$G^4 = (6.67 \cdot 10^{-11})^4 = 2.02 \cdot 10^{-41}$ and
$c^5 = 2.43 \cdot 10^{42}$.
Combining these gives us an order of magnitude of **10^{-83}** for the proportionality.
In order for any significant power to be produced by the emitted waves, the equation tells us that we need truly enormous masses, together with a relatively small orbital radius, in order to "compensate" for this extremely small parameter.

- Gravitational wave detectors measure the "distortion" of space (more precisely, spacetime), but significant energies in the sources are required to produce such distortions in their vicinity. This is illustrated as follows. The "stiffness of spacetime", expressed as the Young's modulus, can be determined, classically, using the equation

$$F = \frac{Y \Delta \ell}{\ell}$$

Again, using G and c, but this time using dimensional analysis rather than taking their numerical values, and additionally incorporating frequency, f, we obtain

$$[Y] = \frac{F}{A} = \frac{N}{m^2} = \frac{MLT^{-2}}{L^2} = ML^{-1}T^{-2} = \frac{(LT^{-1})^2 \cdot T^{-2}}{M^{-1}L^3T^{-2}}$$

It can be shown that the dimensionality of the expression $\frac{c^2 \cdot f^2}{G}$, for example, satisfies the above.
If we make the assumption that the Young's modulus is frequency-dependent, we can thus write

$$Y_{SPACETIME} \sim \frac{c^2 f^2}{G} = 4.5 \cdot 10^{27} f^2 .$$

---

[27] Several attempts were made, prior to 1915, of constructing such theories, notably by Heaviside, Poincaré and Nordstrom. However, none of these alternatives was as fully worked out as the version by Einstein.

For a gravitational wave of frequency 100 Hz, which is quite realistic, we finally obtain $Y_{SPACETIME} \sim 10^{20}\, Y_{STEEL}$ ($\sim 10^{31}$ Pa).

It therefore takes vast energies for an accelerating mass to distort spacetime.

- Third, as for all waves, their flux (power over area) is inversely proportional to the square of the distance to the wave source (r^{-2}). Since the energy of a wave is proportional to its amplitude squared, $E \propto A^2$, the measured amplitude of the waves will go as $1/r$. Thus, as expected, the farther out the event creating the waves, the smaller the observed effect on spacetime that can be measured locally, here on Earth.

Bearing in mind the above three points, we see that it would require events of great violence to produce measurable signals in the best (most sensitive) terrestrial detectors currently available.

Any system producing the gravitational waves would have to be

- very massive (by everyday Earthly standards): at least the mass of a Sun-sized star,
  $m \sim 10^{30}$ kg, and
- accelerating
  - o strongly and
  - o in some non-symmetrical fashion.

Events of this kind only occur in places far beyond our Solar system (our local region is "boringly average", luckily for life on Earth).

What are the candidates for producing measurable gravitational waves?

Supernovae

The supernova explosion[28] of a star, if perfectly symmetrical, would not produce gravitational waves, in spite of the violence of the event; but any detection of a gravitational wave associated with one might provide interesting information about the asymmetry, should it exist, of such an explosion. This information could then be "fed into" the supernova models we currently have.[29] (For example, such information might conceivably help with "producing the bang" in the first place, which is currently theoretically problematic).

Two orbiting bodies

Two stars orbiting one another (since all bodies in an orbit are being accelerated[30]) would also fulfill the above two requirements, at least in principle. Furthermore, the closer the objects to one another, the greater the gravitational radiation produced. Very dense bodies would allow closer proximity without touching, than less dense objects, so a high density is another desirable criterion.

---

[28] A supernova (plural: supernovae or supernovas) is an intense and brilliant stellar explosion. This phenomenon takes place either in the final stages of a massive star's life or when a white dwarf undergoes runaway nuclear fusion. The progenitor star—the original object—may collapse into a neutron star or black hole, or it may be entirely obliterated, leaving behind a diffuse nebula. At its brightest, a supernova can rival the luminosity of an entire galaxy, gradually dimming over a span of weeks or months.

[29] Unfortunately and obviously, there were no gravitational wave detectors on Earth in the year 1054.

[30] Although their speed (the magnitude of their velocity) may not be changing, their direction of movement is. This is sufficient, for only one of these two properties need change for acceleration to be occurring.

The densest objects in the Universe are believed to be neutron stars (or, sometimes denser still, black holes, provided these are not of the supermassive variety). Thus the most likely detectable sources of gravitational waves would be two closely orbiting black holes, or a black hole with a neutron star companion. These are indeed what has been observed since the experimental confirmation of gravitational waves in 2015.

In many cases one would expect a merger of the two bodies with one another very soon after first observing, here on Earth, the gravitational waves emitted by such a system. This is because our observations selectively favour those events that are already the most dramatic; and we know that the gravitational wave radiation produced by two orbiting bodies is at a maximum when those bodies are spiralling inward and are closest to one another, just prior to their collision and merger. It turns out that the duration of most observations ranges from around a minute to only a fraction of a second.

The reason for a spiral orbit is that the combined kinetic and gravitational potential energy of the binary is continually being converted into gravitational wave energy which is radiated outwards. This constant loss of energy means constantly shrinking orbits.

We can verify the claim of shrinking orbits by using simple classical (Newtonian) physics, together with the (not very realistic) assumption that the orbits are circular.

The force on body A due to B is given by Newton's law of universal gravitation,

$$F_A = \frac{Gm_A m_B}{r^2}, \qquad (A.2)$$

where G is Newton's constant and r is the distance between the two masses.
Newton's second law of motion, $F = ma$, then gives us

$$F_A = \frac{Gm_A m_B}{r^2} = \frac{m_A v_A^2}{r_A} \qquad (A.3)$$

Considering the kinetic energy of each body, we have

$$\tfrac{1}{2} m_A v_A^2 = \frac{r_A \, \tfrac{1}{2} m_A v_A^2}{r_A} = \frac{r_A \, \tfrac{1}{2} Gm_A m_B}{r^2}, \quad \text{using (A.2).}$$

Similarly,

$$\tfrac{1}{2} m_B v_B^2 = \frac{r_B \, \tfrac{1}{2} Gm_A m_B}{r^2}$$

Thus for the sum of the kinetic energy of both bodies, we have

$$\tfrac{1}{2} m_A v_A^2 + \tfrac{1}{2} m_B v_B^2 = \frac{(r_A + r_B) \, \tfrac{1}{2} Gm_A m_B}{r^2}$$

$$\tfrac{1}{2} m_A v_A^2 + \tfrac{1}{2} m_B v_B^2 = \frac{Gm_A m_B}{2r} \qquad (A.4)$$

The total mechanical energy for the system is then given by this kinetic energy plus the **negative** gravitational potential energy:

$$E = \tfrac{1}{2}\, m_A v_A^2 + \tfrac{1}{2}\, m_B v_B^2 - \frac{G m_A m_B}{r}$$

which, upon using (A.3) for the kinetic energy,
becomes:

$$E = \frac{G m_A m_B}{2r^2} - \frac{G\, m_A m_B}{r}$$

$$E = -\, \frac{G\, m_A m_B}{2r} \qquad\qquad\qquad (A.5)$$

This negative energy reflects the stability of the bound orbit and the fact that work would need to be done (energy added) to separate the objects "completely" (technically, separated at an infinite distance from one another).

The radiation of gravitational energy of the binary clearly implies a decrease in the total energy of the system of radiating bodies. A decrease on the right hand side of (A.4) therefore means that the energy term becomes even **more negative**. Thus *r* must be getting smaller, which in turn means that the distance between the two masses must decrease over time. This corresponds with our physical intuition.

Gravitational-Wave Astronomy

The first gravitational wave detection was an indirect one, via the **electromagnetic** emission, rather than any gravitational emission, of the source. The Hulse-Taylor binary neutron star system consists of two neutron stars, one of which is rotating on its axis 17 times per second and emitting regular pulses of radio waves at that frequency (rather like the sweeping light beam emitted by a lighthouse).

Figure A4

The Hulse/Taylor neutron star binary system. As they orbit one another, both bodies emit gravitational waves, causing a gradual shrinking of their *orbit.*
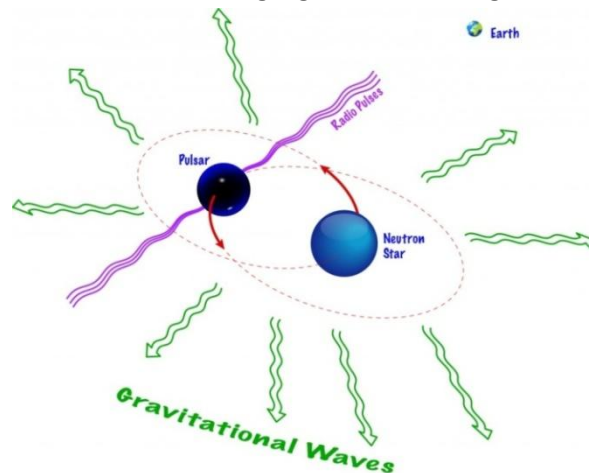
Note. Image: Shane L. Larson

These pulsed radio beams are, by chance, lined up with Earth and their period is known to be changing in a periodic way. From these observations it can be calculated that the separation between the two neutron stars is slowly decreasing: a shrinking orbit. This can only be explained by the binary continually losing energy[31] in the form of gravitational waves: There is no other mechnism to explain the energy loss. (The radio waves themselves cannot cannot carry away enough energy to account for the orbital shrinkage and friction/drag is nonexistent, since the system is not embedded in a gas or dust cloud).

The first **direct** detection of a gravitational wave took place in the autumn of 2015 when the two LIGO ground-based detectors (at Hanford, Washington and Livingston, Louisiana, United States) measured the signal of two coalescing (colliding) black holes (Abbott et al. 2016). The signal was named **GW**150914 – from 'Gravitational Wave' and the date of observation (in the format YYMMDD).

So far, the most massive and the least massive object detected in a merger were found in the events called, respectively, GW190521 (two black holes, 85 and 66 solar masses) and GW190814 (one black hole of 23 solar masses with an object of 2.6 solar masses). Both observations were made in 2019.

The data analysis for gravitational wave detection is complicated. Part of the problem is that of other, unwanted, terrestrial signals of comparable levels, such as seismic vibrations and road traffic. Some of the several techniques used are known as matched filtering, together with Bayesian inference (Bayes, 1764).

Figure A5

Signals of the first confirmed discovery of a gravitational wave (autumn 2015) observed by the LIGO detectors at Hanford and Livingston, United States.
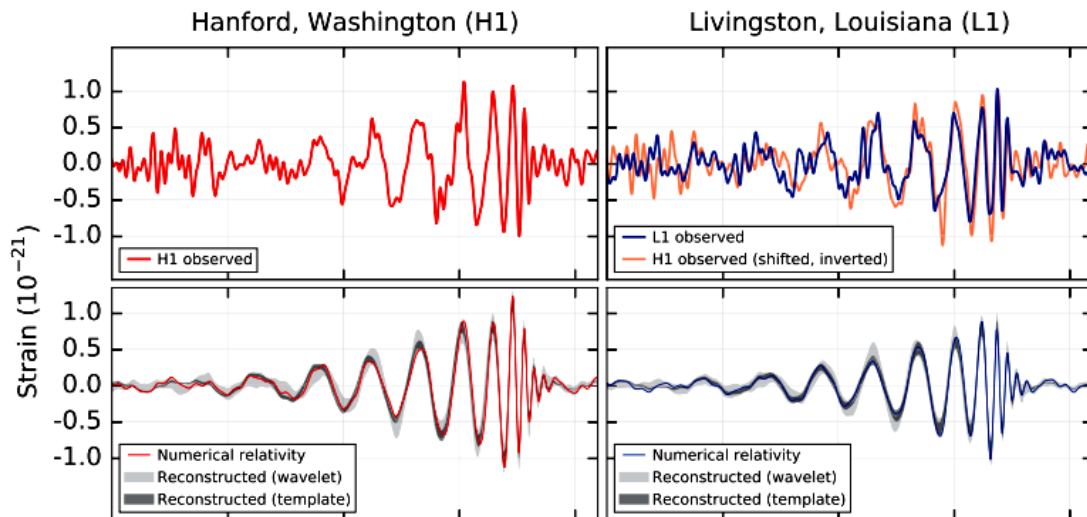
Note. Top row: observations at the two sites; bottom row: a simulation for a system with the parameters of GW150914 using numerical relativity. GW150914 arrived first at Livingston and 6.9 ms later at Hanford.

---

[31] The waves carry not only energy but **information** about their source, such as the mass and spin of the objects; hence the possibility of the discussion in this paragraph.

The principle of gravitational wave detectors is that of the **Michelson interferometer**, which uses the interference properties of light.

Figure A6

Schematic of a Michelson interferometer
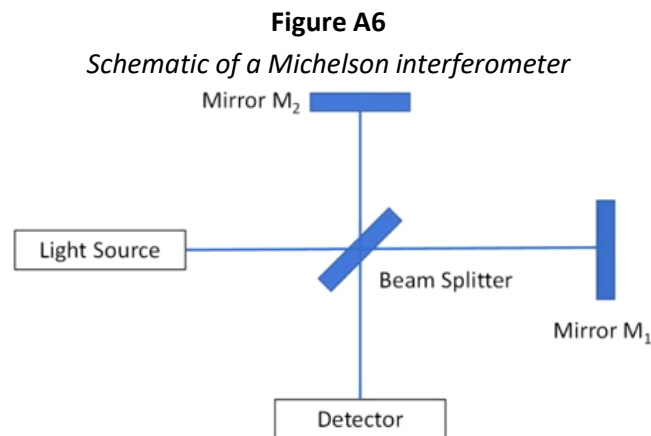
Note. Image and text modified from [Hilborn (2018)](#).

Light from a source (usually a laser) hits a so-called beam splitter (for example, a half-silvered mirror). Part of the light is reflected from the beam splitter (here, upward) and is later reflected by mirror M_2. The other part of the beam goes through the beam splitter (to the right), then reflects from mirror M_1.

After hitting the beam splitter again, parts of the two beams combine and hit the detector, where an interference pattern is observed. In the case of LIGO a laser beam is sent along each of two several-kilometre long "arms" oriented at 90° to one another, and reflected back by mirrors. If there is a phase shift of the two beams relative to one another, this may indicate a spatial distortion that could be evidence of a gravitational wave passing through the detector at that moment. The gravitational waves that can currently be detected by such instruments have frequencies between 10 Hz to a few kHz.

Figure A7

Aerial view of the LIGO detector in Livingston, Louisiana, United States. The vast length of one of the two arms is clearly visible.

Note. Image source: https://www.ligo.caltech.edu/image/ligo20150731c

Since 2017, there has been a collaboration between the United States-based LIGO detectors and the Virgo gravitational wave detector near Pisa, Italy. With the additional data from this third detector, researchers can better localize the sources of the gravitational waves using **triangulation techniques**.

Figure A8

Aerial view of the Virgo gravitational wave detector near Pisa, Italy

Note. Image source: https://www.aps.org/publications/apsnews/updates/ligo-virgo.cfm

Since the first discovery, the LIGO-Virgo collaboration has made over ninety confirmed detections of gravitational waves[32], mainly originating from the merger of two black holes. This direct observation of gravitational waves has thus opened a "new window" on the cosmos, since the information obtained from them about their sources is different from, and complementary to, that of electromagnetic waves. Gravitational-wave astronomy has thus become a new tool for observing the Universe.

---

[32] Results of the third Gravitational-Wave Transient Catalog (GWTC-3) on November 7, 2021, including events released in prior observing runs.

Appendix B – Uncoiling Python: A Quick Guide

"If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may[33] *be a good idea."*
– Tim Peters, *PEP 20 – The Zen of Python*

The homepage of the Python Software Foundation (PSF) provides *Beginner's Guides* for both programmers and non-programmers, serving as valuable reference material. These guides can be accessed at https://www.python.org/. Another useful resource for learning Python is the tutorial by Klein (2021).

Importing Modules to the Code

To use the functionality present in external modules, the Python keyword command `import` is required in the code, together with the external module name. When the interpreter parses the import statement, it *pulls* the module to the current program. The grouping of imports should be in the following order:

1. Standard library imports.
2. Related third party imports.
3. Local application/library specific imports.

Different Import types

1. Full Module Import with an alias
```
import taichi as ti
```

2. Direct Import (Selective Import)
```
from pyautogui import size
```

3. Multiple Import (Explicit Import)
```
import (
    Thread,
    Event,
    Lock
)
```

So-called *wildcard imports* (`from <module> import *`) should be avoided, as they make it unclear which names are present in the *namespace*.

Comments

Comments start either with a hash or begin and end with three quote marks. They should be complete sentences, the first word being capitalized (unless it is an identifier that begins with a lower case letter). A comment can be written entirely on its own line, next to a statement of code, or as a multi-line comment block.

Examples
```
# This line is a comment.
```

---

[33] My emphasis.

```
a = 6.1 # Everything after the hash is a comment.
""" This is a
multiple line block
of comment. """
```

Variable Naming

Variable names are case-sensitive.

Multiple-word variable names

Variable names consisting of more than a single word can be difficult to read. There are several options for improving their readability:

- **Pascal Case**
  Each word starts with a capital letter: `MyVariableName`
- **Camel(back) Case**
  Each word, except the first, starts with a capital letter: `myVariableName`
- **Snake Case**
  Each word is separated by an underscore character: `my_variable_name`

I use the third option in my Python code.

Dynamic Typing

Python is a *dynamically typed* language[34]: it doesn't *know* about its variable types until the code is run. Indeed, there is no command for declaring a variable. Consequently, variables, parameters and return values of a function can be any type and are set during the run of the program.

Python has the following data types built-in by default. The application uses all except complex numeric types, set types and binary types.

Text Type	str
Numeric Types	int, float, complex
Sequence Types	list, tuple, range
Mapping Type	dict
Set Types	set, frozenset
Boolean Type	bool
Binary Types	bytes, bytearray, memoryview
None Type	NoneType

Examples

Python statement:	Data Type:
`x = "Monty"`	String (str)
`x = 42`	Integer (int)
`x = 1.1`	Floating Point (float)

Furthermore, the types of variables can *change while the program is running.* The (current) data type is obtained with the Python `type()` function:

---

[34] Other such languages include: JavaScript, PHP, Lisp and Ruby.

```
print (type(x))
```

Evaluation Order of an Expression

When Python evaluates an assignment, the source (right-hand side) is evaluated before the target (left-hand side), each side, in left to right order. For any subscript in the target, the object (*container*) is evaluated before its subscript.

Data Collections (Aggregate Data Types)

There are four data types used to store collections of data (multiple items) inside a single variable (container):
- `list`
- `tuple`
- `set (and frozenset)`
- `dictionary.`

The differences between them have to do with whether they
• are *ordered* (and can therefore be indexed),
• *allow duplicate members* (multiple elements with identical contents),
• are *mutable* (their values are changeable during execution).

Lists

Lists can hold items of different data types, including other lists. They are defined by enclosing the items in square brackets [ ], with each item separated by a comma.

Examples

Assignment	Access	Output
`my_empty_list = []`	`my_empty_list`	`[]`
`my_list = [1, "Hello", 3.4]`	`my_list [0]` `my_list [-1]`	1 3.4

Tuple
The contents of a tuple are *immutable* (unchangeable during execution): This immutability allows faster execution; iteration through a tuple is faster than for a list (which is mutable). Referencing is with an integer index (which may be negative). Parentheses are optional.

Examples

Assignment	Access	Output
`this_tuple = 0`		
`that_tuple = "Monty", "Python's"`	`that_tuple[1]`	Python's

The various data types and the subset of them that are *aggregate* data types are listed in Tables **B1** & **B2**, respectively.

Table B1

The Python Built-in Data Types

Data type	Name	Example	Comments
Text	str	`a = "Norwegian Blue"`	`str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals.
Numeric	int	`x = 42`	`int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number).
	float	`y = 4.2`	Numeric literals containing a decimal point or an exponent sign yield floating point numbers, which are represented as 64-bit double-precision values in Python. The maximum value any such floating-point number can have is approx. 1.8×10^{308}. Any number greater than this will be indicated by the string inf. `float()` - constructs a floating point number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer).
	complex	`z = 1j`	Appending 'j' or 'J' to a numeric literal yields an imaginary number (a complex number with a zero real part). `complex()` can be used to produce complex numbers. Both the real and imaginary parts are a floating point number. z.real and z.imag allow the extraction of these from a complex number z.
Sequence	list	`x = ["Cleese", "Palin", "Jones"]`	
	tuple	`x = ("Gilliam", "Idle", "Chapman")`	
	range	`x = range(6)`	
Mapping	dict	`x = {"name" : "Eric", "age" : 82}`	
Set	set	`x = {"Dead", "Parrot", "Sketch"}`	
	frozenset	`x = frozenset({"I'm", "a", "lumberjack"})`	
Boolean	bool	`x = True`	Almost any value is evaluated to True if it has some sort of content. Any string is True, except empty strings. Any number is True, except 0. Any list, tuple, set, and dictionary are True, except empty ones.
Binary Types	bytes	`x = b"Hello"`	
	bytearray	`x = bytearray(5)`	
	memoryview	`x = memoryview(bytes(5))`	

Table B2

The Python Aggregate built-in Data Types

Data type	Name	Properties for collections of elements, where applicable
Text Type	str	
Numeric Types	int	
	float	
	complex	
Sequence Types	list	ordered, duplicates allowed, mutable
	tuple	ordered, duplicates allowed, immutable
	range	
Mapping Type	dict	
Set Types	set	unordered, duplicates not allowed, mutable
	frozenset	unordered, duplicates not allowed, immutable (hence the name!)
Boolean Type	bool	
Binary Types	bytes	
	bytearray	
	memoryview	

Constructors

For numeric types, the **constructors**

- `int()`
- `float()` and
- `complex()`

can be used to produce numbers of a specific type.

Operators

Python divides operators into the following groups:

- Arithmetic operators
- Assignment operators
- Bitwise operators
- Comparison operators
- Identity operators
- Logical operators
- Membership operators

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Logical Operators

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Comparison Operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Identity Operators

Operator	Description	Example
is	Returns True if both variables are the same object	`x is y`
is not	Returns True if both variables are not the same object	`x is not y`

Membership Operations

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	`x in y`
not in	Returns True if a sequence with the specified value is not present in the object	`x not in y`

Source: https://www.w3schools.com/python/python_operators.asp

Functions

A function allows the definition of a reusable block of code that can be executed many times, which is essential to *modular programming.*

The Python language passes arguments neither *by reference* nor *by value*, but *by assignment.* It uses a mechanism known as *Call-by-Object*, or Call by Object Reference (less commonly termed Call by Sharing). This is equivalent to passing a *pointer* in other programming languages.

Default argument passing

Sometimes the user cannot or does not wish to provide a value for one or more parameters that are taken by a function. For those arguments for which no value is passed during the function call, *default values* are used. These are provided in the parameters list, where the assignment operator '=' is used for them.

Example
```
def sum(a=5, b=7):  # function with default argument
  """ This function will print the sum of two numbers.
     If the arguments are not supplied
     it will add the default values to one another. """
  print (a + b)
```

The return statement

The Python *style guide, PEP 8 – Style Guide for Python Code (2013),* is noncommittal towards whether the return statement should contain one (or more) variables vs. returning a value without assignment in the function. I tend to use the latter provided the intention is clear.

Examples
```
def function1 (argument1, ...):
    ...
    a = value_calculated_from_the_arguments
    return a
```

```
def function2 (argument1, ...):
    ...
    return value_calculated_from_the_arguments
```

Lint: Cleaning Up Code and Clothes

Lint is the common name for visible accumulations of textile fibres that can detach from clothing. A modern tumble dryer uses something called a lint screen to collect these remnants. Thus, linting, used as a verb, is the "cleaning of tiny bits of fluff". Python code requires such initial "linting", in the figurative sense, to help the programmer find initial simple errors in the source code (which is particularly important because of Python's dynamic typing and the "missing" compilation step which would normally do most of this work). The *linter* thus provides warnings about errors in syntax and formatting, as well as the use of locally unspecified variables. It performs *static analysis* of source code and can be regarded as a simple *debugger.*

Examples

- **Pyflakes.** The Spyder IDE (integrated development environment) uses the simple parser called Pyflakes, available at https://pypi.org/project/pyflakes/
  Pyflakes is faster than **Pylint** (primarily because the former only examines the syntax tree of each file individually). The cost/drawback is that Pyflakes is thus more limited in the types of things it can check.
  - **Pylint**, whose name explicitly refers to its function, is available via the following link, although *not used in my code*: https://pypi.org/project/pylint/

Figure B1

Example of a linting option for the Spyder IDE (integrated development environment)
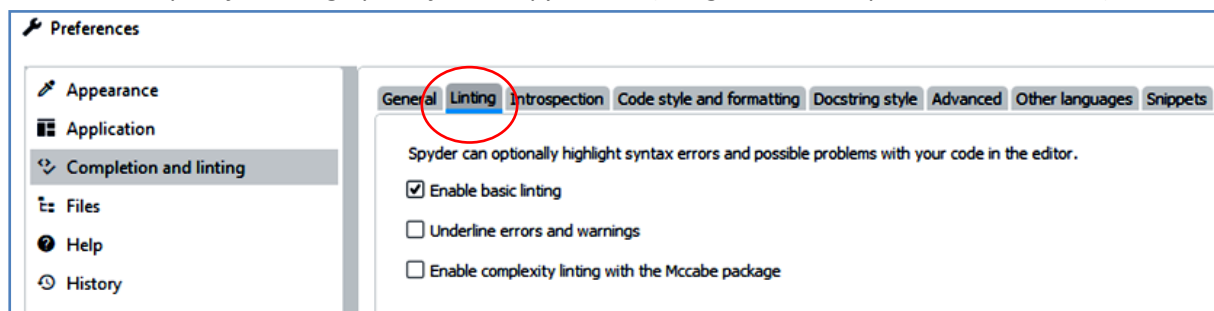
Note. This can be found in: *Spyder -> Tools -> Preferences -> Completion and Linting*

Figure B2

Another tool for linting

Appendix C – More on the Tkinter GUI

Initialising a GUI window

Tk() is a *class* in Tkinter that represents the main window of the GUI application.

The command root = Tk() initializes a Tkinter window, which serves as the main container for all other widgets (buttons, labels, frames, and so on).

Starting Tkinter

This is done by running the command root.mainloop(). This starts an infinite loop which is used to run the application and act upon user input, as and when it occurs. In computer science, something that indicates a user interacting with a program is called an *event*.

Creation of a Frame

Frame widgets are used to group/organise other widgets on the screen. The Frame controls the positions of the individual widgets that fit inside it using something called a *geometry manager*. Tkinter has three such manager methods for organizing layout:
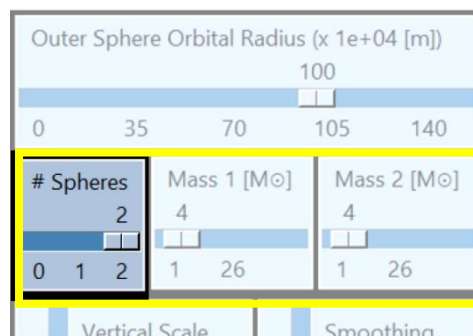- `pack(),`
- `grid() and`
- `place().`

After some experimentation with all three, I decided to use the first of these, as it appeared to be the most straightforward.

In the example shown, the *Frame* widget is instantiated with a black background (`bg="black"`), then placed at the top of the main GUI window. However, because another frame has already been specified earlier in the code listing, the new frame sits is placed directly underneath it, lower down from the top of the screen than the first one. This behaviour is correct, and intended.

Figure C1

A part of the implemented Graphical User Interface (GUI).

Note. Author's image

The yellow box highlights a single frame, while the non-faded section of the image emphasises a specific widget, making it stand out more clearly, for the purpose of illustration

The *pack* method places the frame using (side=TOP) and makes it expand horizontally (fill=X) to take up the full width of the GUI window:

```
padx, pady = 2, 2
horiz_slider_arguments = {
```

```
    "bg": "light steel blue",
    "troughcolor": "steel blue",
    "orient": "horizontal",
}
…
frame = Frame(root, bg="black")
frame.pack(side=TOP, fill=X, padx=padx, pady=pady)  # Expand frame to full
width but not height
```

The root variable is used as the parent window for all other widgets.

This frame will contain three widgets, placed in a row: all of them here are of the type **slider**.
The code provided shows only the first, left-hand-side one. This slider, labeled "# Spheres", is associated with a Tkinter variable called `tkinter_spheres_to_display`. The slider allows users to select a value, here from 0 to 2, with tick marks at intervals of 1. Some padding, defined by padx and pady is placed around each widget inside the frame, for aesthetic reasons:

```
slider_spheres_to_display = Scale(
    frame,
    label="# Spheres",
    variable=tkinter_spheres_to_display,
    from_=0, to=2,
    tickinterval=1,
    **horiz_slider_arguments
)
slider_spheres_to_display.pack(side=LEFT, padx=padx, pady=pady, fill=X)
slider_spheres_to_display.set(2)
```

13.3.1 Colours with Tkinter

These are represented in one of two ways:
- As a string specifiying the proportion of red, green and blue, in hexadecimal digits. For example, "#fff" (white), "#000000" (black), "#000fff000" (green).
- Using defined standard colour names. Examples are: "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta".

This project uses only the latter representation.

Examples
"bg": "light steel blue": light steel blue background
"troughcolor": "steel blue": steel blue slider trough colour

Partial Glossary

These terms are those that are referred to, but not explicitly expanded upon, in the main body of the text. I include those that may be unfamiliar to some readers, terms, particularly from computer science. Some entries have been taken and modified from the following sources:
https://en.wikipedia.org/wiki/Glossary_of_computer_science
https://www.computerhope.com/jargon.htm
https://www.ibm.com/ibm/history/documents/pdf/glossary.pdf

The remaining entries are my own.

Application Programming Interface (API)

A software interface that enables applications to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system.

Argument

A data value passed to a function. Arguments are assigned to the named local variables in a function body. There are two kinds of argument in a function call:

- **keyword** argument: an argument preceded by a name label (an identifier, e.g. *name=*)
- **positional** argument: an argument that is not preceded by an identifier and whose position in the call therefore matters (usually to be placed before all keyword arguments).

Array Slicing

Array slicing is an operation that extracts a subset of elements from an array and packages them as another array, possibly in a different dimensionality from the original.

Callback

A Python function that takes no arguments.

Class

In an object-oriented language, a construct that encapsulates a group of variables and methods.

Code Comment

An annotation or explanation in the source code of a program. Its purpose is to make the code easier to understand. Comments are ignored by interpreters and compilers.

Core

Synonym: **CPU core**

A core receives instructions, and performs calculations, or operations, to satisfy those instructions. A single CPU can have multiple cores inside it. Each core can perform operations independently of the others. The majority of consumer CPUs currently feature between about two and twelve cores.

CUDA

Originally: Compute Unified Device Architecture, acronym discontinued.

CUDA is an architecture for [GPUs](#) developed by NVIDIA, introduced in 2007. It improves the performance of those computing tasks which benefit from parallel processing. CUDA GPUs feature several thousand CUDA cores, integrated onto a single video card. Software must be written specifically for the architecture. Although the native programming language is C++, so-called [wrappers](#) are written for other languages.

Decorator

A decorator is a callable function (or, sometimes, a class) that accepts either a function or a class and returns a new one that "wraps" around the original one. A function or a class in Python is decorated with the @ symbol followed by the decorator name on the same line. The body of the function or class then starts on the following line.

Duck Typing

Duck Typing is a concept related to [dynamic typing](#), where the type or the class of an object is less important than the method it defines. Using duck typing, types are not checked at all. Instead, there is a check for the presence of a given method or attribute. (The reason for the name: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck".)

Dynamic Typing

The term means that a compiler or an interpreter assigns a type to all the variables at run-time, the types being decided based on their values.

Function

A reusable block of code designed to perform a specific task or computation. Functions take inputs, process them, and return an output.

Garbage Collection

The process of freeing memory when it is not used anymore. Garbage collection automates this process using a garbage collector to search for previously allocated memory that is no longer being used by a program. In some programming languages, this freeing of memory must be done manually. Python does it automatically and allows additional control via something called the *gc module*.

Generator Function

A function which returns a generator iterator. It looks like a normal function except that it contains **yield** expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time, using the **next()** function. Each yield action temporarily suspends processing, remembering the location execution state (including local variables).

Graphics Processing Unit (GPU)

The GPU is an electronic circuit used to speed up the creation of both 2D and 3D images. GPUs can either be

- **integrated** (built into the computer's CPU or motherboard), or
- **dedicated**, as a separate piece of hardware known as a video card.

By having its own, separate, processor, the GPU allows the computer's CPU resources to be used for other tasks.

Immutable Object

In Python, an object with a fixed value which cannot be altered during program execution. Immutable objects include numbers, strings and tuples. (If the object is immutable when we update the variable, we actually point it to another object, and Python's garbage collection will recycle the original object if it is no longer used.)

Interpreted Language

An interpreted computer language is a type of programming language in which most of the instructions are executed directly by an interpreter rather than being compiled into machine code beforehand. The interpreter processes the instructions on-the-fly.

Iterator

Synonym: **Iterable**

An iterable is any object that can return its members one at a time as a stream of data, by repeated calls.

Python has two commonly used types of iterables:

- Sequences (element access using integer indices)
- Generators

Just-in-time (JIT) compilation

Synonyms: **Dynamic translation**, **run-time compilation**.

This is a way of executing computer code that involves compilation during execution of a program (at run time) rather than as an independent step before execution.

Namespace

The place where a variable is stored. Namespaces support modularity by preventing naming conflicts and they aid readability and maintainability by making it clear which module implements a function.

Object

Member or instance of a particular class. It contains real values instead of variables.

Object-oriented programming

Synonyms: **OOP, OO programming.**

A programming language paradigm in which the code can be structured as reusable components, some of which may share properties or behaviours. Its basic "building blocks" are classes and objects.

Passing by reference

When passing by reference is used, the memory address of an argument is passed into the function. This means that the function can change the current value of the argument.

Passing by value

When passing by value is used, a copy of the argument's value is passed into the function. This means that the function cannot change the original value of the argument.

PEP (Python Enhancement Proposal)

A PEP is a design document providing information to the Python community, or describing a new feature for the Python language. PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone, and continue to go, into Python.

Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages.

Return Value

The value(s) that a function returns when it completes.

Scope

Synonym: **Visibility**

The reference to an entity (in Python, any object) may not be possible from all places in the code. If this is the case, then in some parts of the program the name may refer to either a different entity, or to nothing at all. The region(s) of the code where a correct and error-free reference can be made to the object is called that object's scope.

Static Typing

A **statically-typed** language is a language (such as Java, C, or C++) where variable types are known at compile time. Although Python is a **dynamically typed** language, it is possible to code so-called type hints, which make it possible to perform type checking of Python code. In contrast to genuinely statically-typed languages, however, type hinting does not cause Python to enforce the types given and can thus be regarded as code comments.

System Bus

Synonyms: **FSB** (front-side bus), **processor bus**, **memory bus**

Connects the CPU (chipset) with the main memory and cache.

Type Hinting

An indication, as part of the code, of the data type of a variable, in order to make the code more self-explanatory. This is usually done within the relevant function.

Example
```
def sum(num1: int, num2: int) -> int:
    return num1 + num2
```

Wrapper

A wrapper describes an intermediate set of functions that allow one piece of software to be accessed directly by other software, without additional computation. For instance, compiled

software written in the C++ programming language may offer a wrapper that allows it to be used directly by programs written in Python.

Yield

A keyword used to return from a function without destroying the states of its local variables. When the function is called, the execution starts from the last yield statement. Any function that contains a yield keyword is termed a **generator.**

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing, at the interactive prompt: `import this`