Technical Report for BPHO 2023 Entry

Alex Arnold, Thomas Davey

Abstract

This report details the technical considerations and implementations that were involved in the development of our website submission for the British Physics Olympiad Computational Challenge 2023, with emphasis on development of models and compatibility considerations.

Introduction

Our aims

Due to the relative simplicity of the 7 tasks prescribed to us, we elected to dedicate most of our development time towards the suggested extension tasks and other models we felt were a natural fit for such a project. One notable omission is the suggested task of encoding a smartphone app. We felt that a website built with considerations for mobile devices would be more appropriate due to the difficulties of distribution for a smartphone app and its litany of cross-compatibility issues.

Languages used in development

The majority of our models were written in Python 3.10 with the exception of the System Simulator which was written in JavaScript with the Three.js library and tested using version 1.5. For all the code written for the models, an emphasis was placed on ensuring any data could be used, not just that which was required for completion of the task. The website was written in HTML and CSS in conjunction with a variety of open-source libraries, which were chosen for authenticity of model representation at the cost of a small degradation in responsiveness.

Deployment

Our website was deployed and hosted through GitHub Pages as it offers a free static site hosting service. We also purchased and setup the custom domain name "enter domain name here" to allow our website to be served from a domain other than the obtuse default domain.

1 Required Tasks

We do not seek to seek to explain or derive any of the methods used in the 7 required tasks. Any form of explanation or derivation is deemed unnecessary as prior knowledge of the methods presented by Dr Andrew French is assumed. Instead, we merely wish to explain our specific implementation of these methods such that they can be presented to the user graphically.

1.1 Python Libraries

We found that the choice of programming language to develop the models in was obvious due to us both being comfortable writing code in Python. Therefore, in order to display the results of the computations involved in the models, Matplotlib was the clear choice of library to use due to its convenient tools for generating both static and animated visualisations.

Other libraries used are the Python standard library math and the NumPy library due to the necessity of non-basic mathematical functions.

1.2 Planetary Data

In order to prevent bloating our code by repeating constants across every model, we decided to centralise all of our constants within a single file that would be imported into each model. This was trivial and done through the use of dictionaries: 1 containing every planet in our data set and individual dictionaries for each planetary system. Within each dictionary, the planets were numbered and their constants sorted into 1 dimensional arrays. This significantly reduced development time as it allowed for exoplanets to simply be added to the file and work immediately with all our pre-existing models. All planetary data was sourced from the NASA exoplanet database (citation here).

1.3 Animation

Tasks 3 and 4 respectively posed the unique challenge of demanding an animated visualisation. Using the method suggested in the briefing, orbital angle would increment such that $\theta_{n+1} = \frac{2\pi}{P}(t_n + \Delta t)$. Unfortunately, this produced unsatisfactory results as the angular velocity of the planets would remain constant, which is certainly not what would be observed in nature. Therefore, to solve this problem the code written for task 5 was adapted into a function that with an input of orbital time, would output orbital angle (see appendix a for code). After implementing this function, the expected "slingshot" around the sun was observed.

1.4 Task 5

Task 5 required evaluating the following equation using Simpson's numeric method:

$$t = P(1-\varepsilon^2)^{\frac{3}{2}}\frac{1}{2\pi}\int_{\theta_0}^{\theta}\frac{d\theta}{(1-\varepsilon\cos\theta)^2}$$

Simpson's numeric method (see appendix b for code):

$$\int_a^b f(x) \approx \frac{1}{3}h\{y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_N + y_N\}$$

Given

$$h = \frac{b-a}{N}$$

As suggested by Dr French in the briefing, we incremented time using this method to determine orbital angle at different orbital times. However, we encountered the issue that at a time increment of 0.1 years where N = 1000, an adequate graph was generated for Pluto despite Mercury generating a graph with a mere 2 points plotted. When the time increment was reduced to 0.001 years an adequate graph for Mercury was generated however the graph for Pluto took an unacceptable amount of time to generate. Our solution to this was to dynamically assign time increments per planet based on predefined values stored in our planetary data file. After implementing this solution, the quality for all planets remains indistinguishable whilst the time to plot also remains reasonably responsive.

1.5 Further Considerations

In task 7, we noted that the plot for when certain planets are assumed to be the centre of their respective system was cluttered to the point of difficult interpretation. To solve this problem, we devised a method to determine the total amount of orbits each planet in the system should undergo, which is as follows:

$$total\ number\ of\ orbits = -13\log(a_n) + 31$$

Where a_n is the semi-major axis of the second furthest planet, the result is rounded to the nearest integer. This significantly reduced the visual clutter and allowed for orbital paths to be observable for all planets.

2 Non-required Models

This section details the methods and implementations of the models programmed that were not suggested by the competition briefing. We do not seek to derive any of the methods used, as that is considered beyond the scope of this report. We decided on these 4 models as we felt that they produced interesting results that related to the nature of the challenge.

2.1 System Simulator

Our System Simulator model provides a more interactive and immersive way of viewing planetary systems and their orbits. It was built from the ground up with ThreeJS, a JavaScript library that allows 3D rendering in browser, which python is unable to achieve. All planetary information used within this model was sourced from the Nasa exoplanet database (citation here).

Within this model, the orbits are rendered using a similar method to that in Task 4. Orbital angle is continuously iterated to determine a sufficient number of positions along the orbital path, which are connected to form a spline and rendered using ThreeJS. Animation of planetary movement demanded a rewrite of the function created in task 5 for JavaScript, where the current model runtime is inputted and orbital angle returned. Current orbital angle is divided by 2π to determine the proportion of the spline that the planet has travelled along and current position of the planet on spline is returned. This ensures that planetary position is correct according to Kepler's Second Law.

Foremost, a full 3D render allows for each planet to have a unique texture, unlike the monochromatic points of task 4. Each texture consists of a 4k image complete with normal map, in order to cast convincing shadows on its terrain. Textures for the exoplanets were generated using Textures for Planets (citation here) according to their identified planet types. Textures for planets within the Solar System were sourced from Solar system scope (citation here). The same methodology was used for cloud textures. Finally, the 6 images that form the skybox were sourced from Skybox generator (citation here). With the addition of textures, the rotation of planets is observable, as well as the tilt on their axis. These features were simple to include using ThreeJS's extensive library of 3D tools in conjunction with recorded values for rotational period and axis tilt.

Mobile compatibility was a major consideration during the development of this model. The user interface is primarily constructed using Bootstrap, a library heavily utilised throughout the website, that allows for dynamic UI scaling and a visual style consistent with the remainder of the website. Furthermore, the inclusion of ray casting within this model allows the user to tap on a planet in order to lock onto it, an input method that feels extremely natural to perform on mobile.

2.2 Binary Systems

Our binary systems model is a form of the restricted three body problem inspired by the Dynamic simulations of Gravity webinar from Dr French. The simulation consists of 2 stars and 1 planet for which the mass is assumed to be negligible. The initial separations from the stars' centre of mass are calculated as follows:

$$r_1 = a\frac{M1}{M1 + M2}$$

$$r_2 = a - r_1$$

Time is then incremented such that $\Delta t = 0.001\ years$ where at each increment, acceleration on each body is calculated using Newton's law of gravitation, which will not be shown here as it is considered common knowledge. The position of each object is then calculated using the verlet method which is shown below:

$$a_n = f(t_n, r_n, v_n)$$

$$t_{n+1} = t_n + \Delta t$$

$$r_{n+1} = r_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2$$

$$V = v_n + a_n \Delta t$$

$$A = f(t_{n+1}, r_{n+1}, V)$$

$$v_{n+1} = v_n + \frac{1}{2}(a_n + A)\Delta t$$

The orbital paths of each object are stored in their respective coordinate arrays and then outputted to the user as either a complete static plot or an animated model. Variables which are user changeable include the masses of the 2 stars, their mutual semi-major axis, the initial velocity of the stars, which star the planet orbits and its initial separation from the star. Due to the immense number of points being plotted, performance for the animated model suffers. One solution to this would be to simply increase Δt, however in order to ensure accuracy of the model, animation was simply limited to a maximum of 30 frames per second to avoid frame pacing irregularities.

2.3 Goldilocks Zones

Our goldilocks zone model computes and plots the region within a planetary system where temperature due to light intensity of a star allows for water to remain a liquid. This is done by calculating the upper and lower bounds of an annulus using the following formula:

$$Distance\ from\ star = \sqrt{\frac{Luminosity\ of\ star}{Luminosity\ of\ Sol}}$$

The lower bound of the annulus is 95% of the distance and the upper bound is 137% of the distance. This annulus is overlayed on top of the orbital model from task 2 to represent the respective system's goldilocks zone. This model is represented in 2 dimensions rather than 3, as in 3 dimensions a spherical shell is used to represent the goldilocks zone which makes the zone harder to perceive. Therefore, a cross-section is taken, represented by the annulus, and thus the model is represented in 2 dimensions.

2.4 Lagrange Points

3 The Website

3.1 Python Integration

Upon considering how best to implement our models into the website, we kept in mind our reasoning for choosing to encode our project as a website. This meant that in order to match the features an app would have, integrating python code into the website was essential. Therefore, our solution was to utilise the PyScript web framework, which utilises the Pyodide python distribution and web assembly to allow python code run within html. PyScript allowed us to adapt our models to return an image or animation that could be displayed on the website when the code is run. Unfortunately, this has a few limitations, the first of which being the loading time upon running a model for Pyodide to initiate, which is impossible to eliminate as it is a limitation of the framework. This was deemed acceptable as past the initial loading times, static models remain responsive.