



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica  
Universitat Politècnica de València

# Simulation of Orbiting Solar Array

Final Degree Project


**Bachelor's Degree in Informatics Engineering**

**Author:** Manuel Martínez López-Sáez

**Co-Tutors:** Brian Vinter

Isabel Galiano Ronda

2017-2018



I would like to dedicate this thesis to my loving parents ...



## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Manuel Martínez López-Sáez

May 2018



## **Acknowledgements**

I would like to thank my advisor, Brain Vinter for guiding and supporting me over the duration of this thesis. You have set an example of excellence as a researcher, mentor and instructor. I would like to thank my co-tutor Isabel Remedios Galiana as well for her support during this last semestre.



# Table of contents

<b>List of figures</b>	<b>xi</b>
<b>List of tables</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Objective . . . . .	2
1.3 Methodology . . . . .	3
1.3.1 Research Approach . . . . .	3
1.3.2 Data Collection . . . . .	3
1.3.3 Limitations . . . . .	4
1.4 State of the art . . . . .	5
<b>2 Context</b>	<b>7</b>
2.1 Requirements . . . . .	7
2.1.1 Javascript . . . . .	8
2.1.2 Python . . . . .	8
2.2 Design . . . . .	9
<b>3 Implementation</b>	<b>11</b>
3.1 Array Simulation . . . . .	11
3.1.1 Orbital Ephemerides . . . . .	12
3.1.2 Shape Factor . . . . .	14
3.1.3 Orbit Function . . . . .	19
3.1.4 Main Loop . . . . .	19
3.2 Web Development . . . . .	21
3.2.1 AngularJS App . . . . .	21
3.2.2 Mathbox Orbit Math . . . . .	24
3.2.3 HTML - Design . . . . .	28



3.2.4	Web Server Code . . . . .	29
<b>4</b>	<b>Results</b>	<b>33</b>
<b>5</b>	<b>Discussion</b>	<b>35</b>
5.1	Analysis of results . . . . .	35
5.2	Efficiency . . . . .	40
	<b>References</b>	<b>43</b>

# List of figures

2.1	JS Precision errors . . . . .	8
2.2	Traditional web architecture . . . . .	9
2.3	Project architecture . . . . .	10
3.1	Ephemerides Function . . . . .	12
3.2	Range-Kutta Function Pair . . . . .	13
3.3	Surface element in spherical coordinates . . . . .	14
3.4	Sphere implementation . . . . .	15
3.5	Sphere2 implementation . . . . .	16
3.6	Beta function implementation . . . . .	17
3.7	Shape factor implementation . . . . .	18
3.8	AngularJS App structure . . . . .	21
3.9	Mathbox initialization . . . . .	22
3.10	Mathbox set-up . . . . .	22
3.11	Mathbox view set-up . . . . .	23
3.12	Mathbox object set-up . . . . .	23
3.13	Quaternion animation . . . . .	24
3.14	Orbital Elements . . . . .	25
3.15	Orbital equation implementation . . . . .	26
3.16	Orbit Visual Implementation . . . . .	26
3.17	Example Orbit Mathbox Visualization . . . . .	27
3.18	App layout . . . . .	28
3.19	Orbit Parameters . . . . .	29
3.20	Orbit Info . . . . .	30
3.21	App Settings . . . . .	30
3.22	Solar simulation settings . . . . .	30
3.23	Server default app route . . . . .	31
3.24	Server simulation app route . . . . .	31

---

4.1	Web Application Result . . . . .	33
5.1	Parameters used for simulation results . . . . .	37
5.2	Results- Solar Irradiance . . . . .	37
5.3	Results- Efficiency versus Temperature . . . . .	38
5.4	Efficiency - No optimisation . . . . .	40
5.5	Python vs NumPy . . . . .	41
5.6	Python vs NumPy - 2 . . . . .	41
5.7	Numpy optimization . . . . .	42
5.8	Efficiency - Optimised . . . . .	42

# List of tables

- 4.1 Raw results from array simulation . . . . . 34
- 5.1  $P_{max}$  for orbit simulation . . . . . 39



# Chapter 1

## Introduction

CubeSats have served as teaching tools for thousands of students and researchers all across the globe. They are now also being considered as commercial assets thanks to the miniaturisation of hardware and thus thanks to the increased capabilities they can offer. The hostile environment in which they are designed to work requires careful analysis and testing of the systems on the ground before launching into space.

One of the main concerns and difficulties with designing and building CubeSats is the relatively low power requirements they have to meet due to their small size/weight and thus low solar panel surface area. Often times, calculating the available power on orbit for a CubeSat can be challenging, but necessary for mission assurance. Accurate knowledge of how much energy is received in orbit is required in order to correctly set the dimensions of the onboard batteries.

As satellites orbit around the Earth, for example, they have periods of darkness, where they do not receive any solar radiation and thus have to run on their internal batteries. Calculating these periods of darkness and the amount of solar irradiance received during the periods of sunlight is a challenge. There are many factors that can change the outcome of this calculation. Accurate simulations require not only sunlight to be taken into account, but also the irradiance reflected by earth's surface and atmosphere. This project will not calculate with absolute accuracy the albedo reflection as one can take into account many factors such as, terrain and cloud cover for which the satellite is passing over. Polar orbits, for example, have greater albedo values due to the fact that the poles reflect more light than other latitudes. However, orbital altitude will be taken into account.

This project has a profound impact on aerospace industry, because the uses of accurate simulation of solar irradiance of a solar array orbiting the Earth can have numerous practical applications. This accurate simulation of solar irradiance also allows for different orbited bodies such as Venus or Mars. As it will be explained in the following sections, we will

need to use a combination of view factor equations to calculate the solar incidence on the array, calculating before the orbit of both bodies and angles between the bodies in the simulation.

## 1.1 Motivation

This project is in part motivated by the creation of the CubeSat UPV group. A group of students at the Polytechnic University of Valencia looking to design a CubeSat from scratch. This project would therefore serve as a tool to calculate the required solar panel area needed to power the hardware they intend to fly.

## 1.2 Objective

The objective of this project is to design and develop a web application capable of calculating the available power received from the sun for an orbital solar array. This simulation will take into account the direct solar radiation received, the reflected radiation from the orbited body, and cosmic radiation. The user will be able to input their data for their specific satellite and receive the results of the simulation.

**Main Objective:** Accurately output the amount of solar energy received by a solar cell during a complete orbit taking into account the previously mentioned simulation. The application will allow the following parameters to be configurable for the simulation:

- Orbital parameters
- Array surface area
- Number of arrays
- Planet [Venus, Earth, Mars]
- Date

**Secondary Objectives:** The application will include the following features:

- Gravitational interaction between bodies in our simulation
- Science-ED friendly
- Allow results' to be downloaded in .CSV and .txt formats

## 1.3 Methodology

This section provides an outline of the research methodology used to develop the proposed project. This includes: the research approach, a description of primary data collection for the comparative techniques that ensure correct results and the limitations of the adopted research method.

### 1.3.1 Research Approach

The research approach influences design and provides an opportunity to consider benefits and limitations of various approaches available to the researcher. Due to the practical nature of this project and the various fields needed to solve it, fundamental research was chosen as the preferred method since applied research seeks generalisations and aims to develop basic processes at a more theoretical level.

A fundamental research approach aims to solve a problem by often using several disciplines. Due to the nature of the problem and the necessary solution, methods and theories needed to solve this problem are well known and accepted principles of science. It is therefore heavily based off of accepted laws of physics and mathematics.

In order to verify results, comparative research was used to compare results with real world test cases. Some generalisations were made with regards to certain data points.

### 1.3.2 Data Collection

Due to the nature of the project, processed data was deemed to be important due to the complexity of the mathematics involved. Obtaining processed data in the real world would allow comparative techniques to be used to ensure the correct implementation of the proposed solutions.

Various web tools were used to extract data manually. Data was also downloaded from trusted sources such as PDAS<sup>1</sup> in TXT file format. This information was saved for later use and would serve as a comparison for the results obtained during testing. Physics laws and mathematical equations were extracted from trusted sources as-is, in order to implement them in the programming language of choice. Apart from the aforementioned data, no other data was extracted as there was no need for it, taking into consideration the practical aspects of the project

---

<sup>1</sup>Public Domain Aeronautical Software



### **1.3.3 Limitations**

Currently most data regarding electrical power requirements and specific solar panel efficiencies in orbit for commercial satellites are proprietary information and thus not disclosed to the public. Therefore, meaningful and useful information for modern systems was not available.

Some general and estimated values for specific efficiencies do exist and thus these were interpolated. Data was also found for older systems dating back to the 1980s and 1970s, which required certain generalisations for its later use.

## 1.4 State of the art

This section presents a short summary of the current state-of-the-art techniques available for use to calculate and simulate the radiation received by an orbital array during a full orbit. As of the publication date of this project there is no software, commercial or non-commercial, that solves the problem related with the simulation. It is however a possibility that major satellite manufacturers have proprietary software developed in order to calculate this important data.

There is however relevant literature that does tackle this problem in some way. Most investigation on satellites and solar arrays are focused on the challenges of materials and efficiencies. For example, in the journal for Proceedings of the XIII Space Photovoltaic Research and Technology Conference, a paper was published that explored solar cell temperature coefficients in space (Landis, 1994)[4]. Solar cells had to be modeled in orbit taking into account solar incidence, distances and black body calculations. These findings present some important results with which we can validate the resulting efficiencies and  $P_{max}$  we obtain. This value is the maximum temperature coefficient. It indicates how much power a certain panel will lose or gain when the temperature changes  $1\text{C}^\circ$  above or below  $25\text{C}^\circ$ .

Other literature also studies the lifespan of orbital solar arrays. The harsh environment has adverse affects on the hardware as it degrades over time. In the study 'Thermal distortion analysis of orbiting solar array' (Shin, 2001)[3], the thermal environment the solar array is subjected to is calculated. The deemed maximum solar flux is  $1393\text{W}/\text{m}^2$  and minimum is  $1305\text{W}/\text{m}^2$ . They make an important point that specifies the dependability of the outcome with the solar array solar vector. Additionally the calculated values of solar flux will be considered for comparison.

More research in this field looks at the thermal analysis of solar arrays during orbits. The paper 'Thermal analysis of composite solar array subjected to space heat flux' (Junlin Li, 2012)[1] calculated using view factor equations all sources of radiation received by a satellite in orbit. It includes the results for both LEO<sup>2</sup> satellites and GEO<sup>3</sup> satellites. The results show very different thermal environments depending on the chosen orbit. Earth reflected radiation was found to have a very big influence on overall temperatures and thus should be taken into account in our simulation. This paper proves to have a special significance for aerospace engineering.

One of the complexities in this field is the numerical analysis of the solutions for calculating the amount of radiation satellites receive while in orbit. Certain papers can be found for the possible approaches that can be taken, such as the paper 'Analytical and numerical

---

<sup>2</sup>Low Earth Orbit

<sup>3</sup>Geostationary orbit

approaches of a solar array thermal analysis in a low-earth orbit satellite' (Hui Kyung Kim, 2010)[2] that explores the numerical solutions for this problem. They also approached the problem from an analytical perspective which was quite useful for calculating the worst maximum and minimum temperatures of the solar array. The analytical approach showed reasonable results when compared with the numerical approach. This paper therefore showed that max. and min. temperatures can be easily calculated serving an important purpose for the aerospace industry.

Other features of this project, specially visually are available on online sources using an array of different tools and programming languages. The representation and manipulation of orbits have been implemented in several educational websites<sup>4 5 6</sup>. The methods used for these solutions are all heavily based in Javascript as they are all web-based.

---

<sup>4</sup>[http://lasp.colorado.edu/education/outerplanets/orbit\\_simulator/](http://lasp.colorado.edu/education/outerplanets/orbit_simulator/)

<sup>5</sup><https://theskylive.com/3dsolarsystem>

<sup>6</sup><http://orbitalimulator.com/gsim.html>

# Chapter 2

## Context

Before we begin to look at how we might solve the problem mathematically, we must figure out what are the best technologies and methods we might use for the solution. This particular task is essential as choosing one programming language over another can affect the real time aspect of the system. A preliminary analysis of the problem must be evaluated to know what kind of problem size we will have to work on. These decisions will be made depending on the requirements the project has and through testing.

### 2.1 Requirements

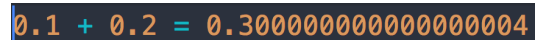
One key aspect of the project is accessibility. As previously mentioned, the motivation behind this project is to provide scholars', 'researchers' in the field' with a tool for an accurate simulation of an orbital array. However, it can also be used as a teaching tool both for orbital dynamics as well as for mechanics with solar energy. In order to have the most exposure, a web-based platform would be the most useful for both advanced and high level students. With this web requirement in mind, there are several considerations that must be taken into account. The technological context around this project is another issue that requires examination.

Web-based solutions offer accessibility to the application through a web browser, allowing it to be used by a larger target audience than any other solution. However web browsers, have limited performance and thus, running heavy complex tasks is not recommended as the client applications vary in performance depending on the host computer. A decision has to be made on how to confront this problem and which programming language can be used for the front and back end of the web application.

### 2.1.1 Javascript

Javascript is a lightweight compiled programming language, otherwise known as the scripting language for Web pages. There are also non browser uses for JS such as Node.JS and other server-side frameworks. It is a dynamic scripting language, prototype-based, object-oriented, imperative and declarative.

Regarding Javascript, we will go into a bit of detail into one important aspect. One of the main problems Javascript faces when operating with floats is rounding errors. The representation of floating points in JavaScript follows the format as specified in IEEE-754. Specifically, it is a double-precision format, meaning that 64 bits are allocated for each floating point. Due to inadequacies when representing numbers in base-2, as well as a finite machine, we are left with a format that is filled with rounding errors. Javascript therefore does not fit well with the Math-intensive simulation we wish to conduct.



```
0.1 + 0.2 = 0.30000000000000004
```

Fig. 2.1 Floating point sum precision error

There are some possible solutions to this problem; one of them is Math.js, a library capable of solving these rounding errors and adding functionality and functions to the suite of already available math functions javascript provides. This is a good simulation for the visual aspect of the application, rendering orbits with math functions, though we can explore other server-side programming languages to fill this need.

### 2.1.2 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for programmers. It has also grown in popularity among researchers thanks to modules such as numpy <sup>1</sup> and scipy <sup>2</sup>. Numpy is a very attractive package for scientific computing. It contains, among other elements, N-dimensional array objects, sophisticated broadcasting functions, linear algebra, Fourier transformations, and several random number capabilities. All very useful tools when developing a simulation with accuracy in excess of  $1 * 10e - 8$ . Regarding mathematical operations in python and more specifically Numpy, we have access to a default 64-bit precision float with the added option of using *long doubles*. Depending on the architecture of the system it will be

---

<sup>1</sup><http://www.numpy.org/>

<sup>2</sup><https://www.scipy.org/>

an 80-bit floating point representation or an 128-bit floating point representation. Later on we will explore why this long double is of special use for us. Nonetheless, with the default 64-bit floating point numbers used by default in Numpy, we can have an accuracy of up to  $1 * 10e - 15$ <sup>3</sup>.

Python can also be used as a server-side language with the help of modules like Flask<sup>4</sup>. Running python server-side allows us to efficiently run any python code and use REST calls for bidirectional communication between the browser and server-side code. Flask is a micro-framework for python based on Werkzeug<sup>5</sup> and Jinja 2<sup>6</sup>.

## 2.2 Design

After taking into consideration the technologies available to us for the implementation, we can make a decision on the architecture of the system. When designing and developing web-based applications a certain distinction has to be made on how the system works.<sup>7</sup>

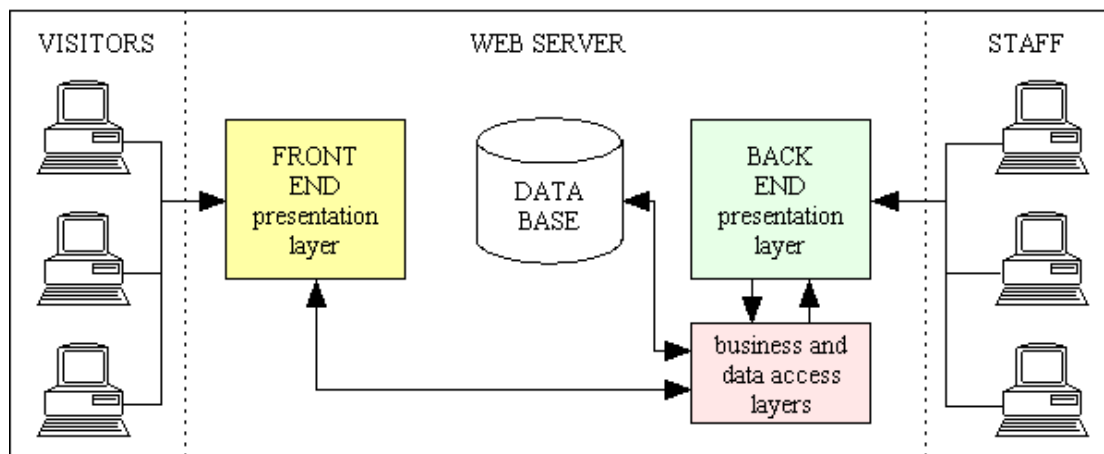


Fig. 2.2 Traditional web architecture

Traditional web architectures have a front-end and a back-end. Front-end is the presentation layer of the web; usually the HTML<sup>8</sup> and CSS<sup>9</sup> are considered the front-end. These are served by web servers upon request from client browsers when accessing a certain webpage.

<sup>3</sup>This precision was obtained after testing python operations within the environment in comparison with the same operations in Fortran90

<sup>4</sup><http://flask.pocoo.org/>

<sup>5</sup><http://werkzeug.pocoo.org/> - Advanced WSGI utility module

<sup>6</sup><http://jinja.pocoo.org/> -Jinja2 is a full featured template engine for Python

<sup>7</sup><https://www.tonymarston.net/php-mysql/an-end-to-end-ecommerce-solution-requires-more-than-a-fancy-website.html>

<sup>8</sup>Hypertext Markup Language

<sup>9</sup>Cascading Style Sheets

Back-end systems contain certain logic and data layers, which give functionality to the front-end side of the web-application.

The web-architecture for our system will be the following:

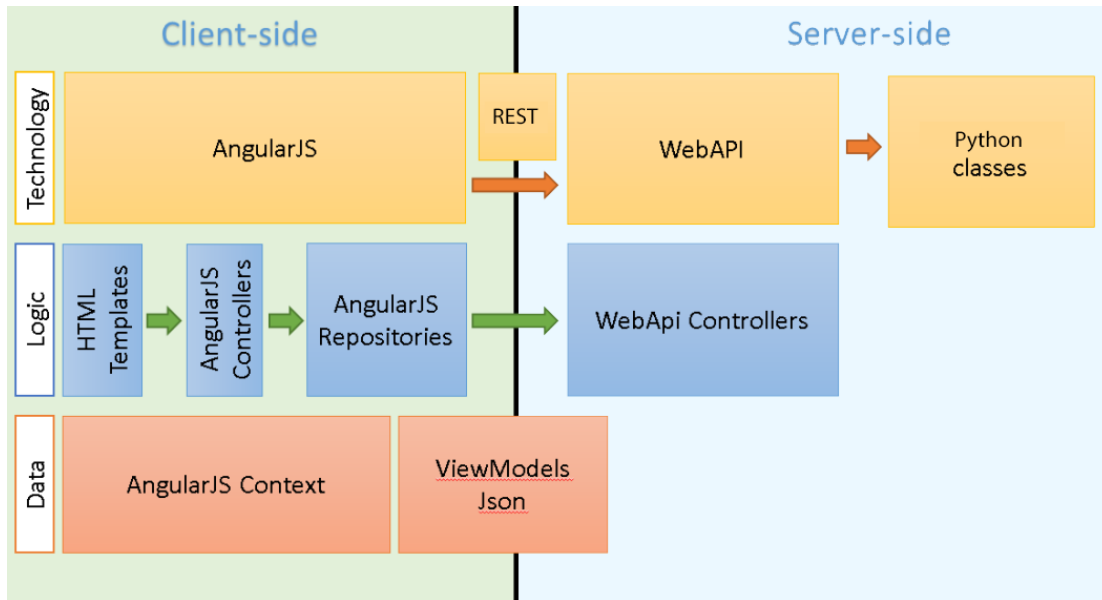


Fig. 2.3 Web architecture

The front-end of the web-application will utilise the Javascript framework AngularJS. This framework will take care of all the logic between the HTML views and the data and information displayed in them. AngularJS is widely considered one of the best libraries thanks to its ease of use and widely understood MVC<sup>10</sup> structure. The visual aspects, such as the representation of orbits, will be implemented using a myriad of JS libraries such as Three.js<sup>11</sup> and Mathbox.js<sup>12</sup>.

The back-end system will be based off of a Python web-server using the previously mentioned Flask library. This will allow us to run the CPU intensive simulation server-side and not bog down the client browser in the process. Having access to Numpy is also an added benefit as the implementation of the simulation will need the use of N-dimensional array objects. In the section on efficiency we will comment on the benefits of using python and specifically Numpy when operating over n-dimensional arrays.

<sup>10</sup>Model-view controller architecture

<sup>11</sup><https://threejs.org/>

<sup>12</sup><https://github.com/unconed/mathbox>

# Chapter 3

## Implementation

The implementation of the project has to be divided in two distinct development efforts: On the one hand the design and development of the front-end javascript code and on the other hand the python webAPI/array simulation. With this in mind we can proceed with the implementation.

### 3.1 Array Simulation

The array simulation is by far the most challenging development of the two previously mentioned efforts. After appropriate analysis and research the following techniques need to be developed in order to achieve the required results: techniques for constructing an ephemeris of an orbiting body, the calculation of black-body radiation from the sun with corrections for reflection and shadowing of the earth, and the radiation from a photoelectric array. The calculation of the temperature of the satellite can be done by numerical solutions of the differential equations of heat absorption and radiation.

As a general setup, the Earth ephemerides and solar array ephemerides need to be calculated, followed by the location of the Earth in heliocentric coordinates according to a specific date provided by the user. With these coordinates we can then calculate the surface elements on the orbital track of the solar array from which the generation of the shape factor parameters in heliocentric coordinates can be stipulated. The solar beta angle is then calculated for which a Solar model is generated. For the albedo calculation, element shape factors are required. Based on the information above we can then iteratively calculate time-step through time-step the necessary information.

The process of debugging and validating the results from this simulation will be talked about in a later section. Some problems were encountered with precision errors when comparing results by the math and numpy modules with a calculator for example. The necessity for



precision is paramount, as in some cases a difference of one hundred thousands could alter results quite considerably.

### 3.1.1 Orbital Ephemerides

The purpose of calculating the orbital ephemerides of both bodies is to create a table of orbital position as a function of time. We can analyse one complete period with theta going from 0 to  $2\pi$  in equal increments. The time required for each step is then computed by a numerical solution of the differential equation of orbital motion. The time for one complete period is computed and returned. The array is then normalised by this quantity and thus goes from 0 to 1.

```
def ephem(self, a, e, w, tp, theta):
    n = min(tp.shape, theta.shape)[0]
    dth = (np.pi * 2) / int(n - 1)
    c = ((1.0 - e * e) ** 1.5) / (np.pi * 2)
    st = 0.0
    th = 0.0
    ds = 0.0
    sts = 0.0

    for j in range(0, n):
        jrks = 1
        tp[j] = st
        theta[j] = th
        while 1 <= jrks < 4:
            f = 1.0 + e * np.cos(th)
            dst = c / (f * f)
            temp = self.rsfdx(st, dst, ds, dth, sts, jrks)
            st = temp.get('y')
            ds = temp.get('dy')
            sts = temp.get('ys')
            temp = self.rscon(th, dth, jrks)
            th = temp.get('xc')
            jrks = temp.get('kc')

    f = 1.0 / tp[n - 1]
    tp = np.multiply(tp, f)
    p = (math.pi * 2) * (a ** 1.5) / (3600. * np.sqrt(1.068E-9 * w))

    return {'p': p, 'tp': tp, 'theta': theta}
```

Fig. 3.1 Implementation of ephem function

For each iteration within the Ephem function rsfdx is called. It is the implementation of the Range-Kutta, a method of numerically integrating ordinary differential equations by using a trial step at the midpoint of an interval to cancel out lower-order error terms. Used

here is the fourth-order formula.

$$K_1 : K_1 = hf(x_n, y_n) \quad (3.1)$$

$$K_2 : K_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \quad (3.2)$$

$$K_3 : K_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \quad (3.3)$$

$$K_4 : K_4 = hf(x_n + h, y_n + k_3) \quad (3.4)$$

$$y_{n+1} \quad y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \quad (3.5)$$

```
def rsfdx(self, y, dydx, dy, dx, ys, kr):
    if kr == 1:
        ys = y
        dy = dydx * dx
        y = ys + 0.5 * dy
    elif kr == 2:
        dy = dy + 2.0 * dydx * dx
        y = ys + 0.5 * dydx * dx
    elif kr == 3:
        dy = dy + 2.0 * dydx * dx
        y = ys + dydx * dx
    elif kr == 4:
        dy = ((dy + dydx * dx) / 6.0)
        y = ys + dy
    return {'y': y, 'dy': dy, 'ys': ys}

def rscon(self, xc, dxc, kc):
    if kc == 1:
        xc = xc + 0.5 * dxc
        kc = 2
    elif kc == 2:
        kc = 3
    elif kc == 3:
        xc = xc + 0.5 * dxc
        kc = 4
    elif kc == 4:
        kc = 1
    return {'xc': xc, 'kc': kc}
```

Fig. 3.2 Implementation of rsfdx and rscon functions called by Ephem

The function is called iteratively through the main ephem function and therefore iterated through the values [1,4] of the rsfdx and rscon function. After every execution of the rsfdx function, the rscon method is also called. It is the control procedure of the Range-Kutta function.

### 3.1.2 Shape Factor

The shape factor equation is used various times during execution. It is more commonly known as the view factor and it is the proportion of the radiation which leaves a surface  $A$  that strikes a surface  $B$ . In a complex scenario there can be any number of different objects, which can be divided in turn into even more surfaces and surface segments.

In order to apply this shape factor equation we need to first calculate the heliocentric position of the Earth or Planet in question at a certain date. We then calculate the orbital surface track the orbital array passes over during its orbit. The portion of the surface that affects our solar array is calculated using the following formula:

$$dphi : \quad dphi = \arccos(RE/(a * (1.0 + e))) \quad (3.6)$$

$$phimax : \quad phimax = \frac{1}{2}\pi + dphi \quad (3.7)$$

$$phimin : \quad phimin = \frac{1}{2}\pi - dphi \quad (3.8)$$

$RE =$  Radius of earth,  $a =$  Array altitude,  $e =$  Array eccentricity

Both  $phimin$  and  $phimax$  are measured from the poles in radians. After obtaining these values, we need the surface position tensor, the surface area vector, the surface normal tensor and the actual number of surface elements for the orbited body and the sun. In order to calculate the spherical surface tensors in the general coordinate system we use functions `sphere2` and `sphere`. `Sphere2` is called to obtain the spherical surface tensors in the sphere's coordinate system as the solar array is orbiting this body. `Sphere` is called to obtain the spherical surface tensors in order to create a solar model.

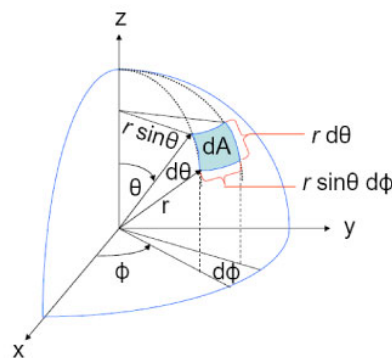


Fig. 3.3 Surface element in spherical coordinates

```
def sphere(self, xo, r, x, a, vn, phimin, phimax, n):
    na = 0
    phi = phimin
    dphi = (phimax - phimin) / n
    the = 0.0
    j = 0

    for k in range(0, n):
        sinp = np.sin(phi + dphi / 2.0)
        cosp = np.cos(phi + dphi / 2.0)
        da = np.float64(np.divide(dphi, sinp))
        xm = np.pi * (2 / da) + 0.5
        m = np.trunc(xm)
        da = np.pi * (2/m)
        the = the + da / 2.0
        area = r * r * da * (np.cos(phi) - np.cos(phi + dphi))
        na = na + m

        for i in range(0, np.int(m)):
            vn[0, j] = sinp * np.cos(the + da / 2.0)
            temp = the + np.divide(da, 2.0)
            vn[1, j] = sinp * np.sin(the + da / 2.0)
            vn[2, j] = cosp
            a[j] = area
            # x[0:2, j] = xo[0:2] + r * vn[0:2, j]
            x[0, j] = xo[0] + r * vn[0, j]
            x[1, j] = xo[1] + r * vn[1, j]
            x[2, j] = xo[2] + r * vn[2, j]
            the = the + da
            j = j + 1

        phi = phi + dphi
    return {'x': x, 'a': a, 'vn': vn, 'na': na}
```

Fig. 3.4 Implementation of sphere function

For the body orbited by the solar array we need to execute several additional operations. After calculating the spherical surface tensors in the sphere's coordinate system, we need to transform them to the general coordinate system. This is done by rotating about the x axis, rotating about the z axis and rotating for earth's inclination. We iteratively call methods Rtop and Ptor. Ptor executes a polar to rectangular conversion and Rtop executes a rectangular to polar conversion. An angle in  $[0, \pi]$  range is returned instead of  $[-\pi, \pi]$ .

```
def sphere2(self, xo, phi, ra, r, x, a, vn, phimin, phimax, n, na):
    xc = np.zeros(3)
    # Calculate spherical surface tensors in sphere's coordinate system
    result = self.sphere(xc, r, x, a, vn, phimin, phimax, n)
    x = result.get('x')
    a = result.get('a')
    vn = result.get('vn')
    na = result.get('na')
    # Iterate through returned surface elements
    for j in range(0, np.int(na)):
        result = self.rtop(x[1, j], x[2, j]) # Rotate about x axis
        rx = result.get('r')
        td = result.get('th')
        rn = np.sqrt(vn[1, j] * vn[1, j] + vn[2, j] * vn[2, j])
        td = td + phi
        result = self.ptor(rx, td)
        x[1, j] = result.get('x')
        x[2, j] = result.get('y')
        result = self.ptor(rn, td)
        vn[1, j] = result.get('x')
        vn[2, j] = result.get('y')

        result = self.rtop(x[0, j], x[1, j]) # Rotate about z axis
        rx = result.get('r')
        td = result.get('th')
        rn = np.sqrt(vn[0, j] * vn[0, j] + vn[1, j] * vn[1, j])
        td = td + ra
        result = self.ptor(rx, td)
        x[0, j] = result.get('x')
        x[1, j] = result.get('y')
        result = self.ptor(rn, td)
        vn[0, j] = result.get('x')
        vn[1, j] = result.get('y')

        result = self.rtop(x[1, j], x[2, j]) # Rotate for Earth's inclination
        rx = result.get('r')
        td = result.get('th')
        rn = np.sqrt(vn[1, j] * vn[1, j] + vn[2, j] * vn[2, j])
        td = td - 0.409274
        result = self.ptor(rx, td)
        x[1, j] = result.get('x')
        x[2, j] = result.get('y')
        result = self.ptor(rn, td)
        vn[1, j] = result.get('x')
        vn[2, j] = result.get('y')

        x[0:2, j] = x[0:2, j] + xo[0:2] # Translate
    return {'x': x, 'a': a, 'vn': vn, 'na': na}
```

Fig. 3.5 Implementation of sphere2 function

The next step is calculating the beta angle between the sun and the solar array. We input the location of the earth in x,y coordinates and then transform the coordinates, to take into account the inclination of the orbit of the satellite and the right ascension node of the orbit. A slight correction for the earth's inclination is also needed. After the transformations we calculate the beta angle with the following formula:

$$beta : \quad beta = \arccos \left( \frac{(x * xa) + (y * ya)}{\sqrt{(x^2 + y^2) - \frac{1}{2}\pi}} \right) \quad (3.9)$$

$$beta(deg) : \quad beta(deg) = \frac{beta * 180}{\pi} \quad (3.10)$$

```
def beta(self, re, phi, ra):
    xa = 0.0
    ya = 0.0
    za = 1.0

    result = self.rtop(ya, za) # Rotate for inclination
    theta = result.get('th')
    r = result.get('r')
    theta += phi
    result = self.ptor(r, theta)
    ya = result.get('x')
    za = result.get('y')

    result = self.rtop(xa, ya) # Rotate for right ascension node
    theta = result.get('th')
    r = result.get('r')
    theta += ra
    result = self.ptor(r, theta)
    xa = result.get('x')
    ya = result.get('y')

    result = self.rtop(ya, za) # Rotate for earth's inclination
    theta = result.get('th')
    r = result.get('r')
    theta -= -0.409274
    result = self.ptor(r, theta)
    ya = result.get('x')
    za = result.get('y')

    bet = np.arccos((re[0] * xa + re[1] * ya) / np.sqrt(re[0] * re[0] + re[1] * re[1])) - np.pi * 0.5
    bet = bet * 180 / math.pi # Radians to degrees

    return bet
```

Fig. 3.6 Beta function implementation

After obtaining the beta angle between the solar array and the sun we can finally proceed to the Shape Factor calculation mentioned above. This is done for the albedo radiation

calculation. Considering two finite surfaces, computing the view factor (shape factor) is a problem of mathematical integration.

$$F : \quad F = \frac{1}{A} \int_{A_1} \left( \int_{A_2} \frac{\cos \beta_1 \cos \beta_2}{\pi r^2} dA_2 \right) dA_1 \quad (3.11)$$

X =Coordinate array for each nodal plane

A =Area of each nodal plane

VN =Components of unit normal vector for each nodal plane

N =Number of nodal planes in each body

1,2 =Bodies for shape factor computation

B = All surface between Body 1 and Body 2 (if no intervening surfaces, XB=1, NB= 1)

Shfac=Black body shape factor referenced to body 1

```
def shfac(self, x1, a1, vn1, n1, x2, a2, vn2, n2, xb, ab, vnb, nb):
    r = np.zeros(3) # 3 dimensional array
    b = np.zeros(3)
    a = 0.0
    f1 = 0.0
    for j in range(0, np.int(n1)):
        a = a + a1[j]
        for k in range(0, np.int(n2)):
            r[0:2] = x2[0:2, k]-x1[0:2, j]
            rmag = np.sqrt(np.sum(r ** 2))

            cos1 = ((vn1[0, j] * r[0]) + (vn1[1, j] * r[1]) + (vn1[2, j] * r[2])) / rmag
            cos2 = 0. - (((vn2[0, k] * r[0]) + (vn2[1, k] * r[1]) + (vn2[2, k] * r[2])) / rmag)

            if cos1 <= 0.0 or cos2 <= 0.0:
                pass
            else:
                for i in range(0, np.int(nb)):
                    b[0:2] = xb[0:2, i] - x1[0:2, j]
                    xdotn = np.dot(b[0:2], vnb[0:2, i])
                    rdotn = np.dot(r[0:2], vnb[0:2, i])

                    if rdotn == 0.0:
                        pass
                    else:
                        c = xdotn / rdotn
                        if c <= 0.0 or c >= 1.0:
                            pass
                        else:
                            s = np.sum((c * r[0:2] - b[0:2]) ** 2)
                            if s <= ab[i] / 2.0:
                                f1 = f1
                                f = f1 / (np.pi * a)
                                return f

                f1 = f1 + cos1 * cos2 * a1[j] * a2[k] / (rmag * rmag)
    f = f1 / (np.pi * a)
    return f
```

Fig. 3.7 Shape factor implementation

### 3.1.3 Orbit Function

Regarding other functions used in the simulation, the orbit function is of special importance as it returns the coordinates of the solar array for a specific time from the perigee. We use the heliocentric coordinates of the earth, the time to perigee, the position of the arrays from ephemeris(calculated previously) and the orbital period.

The location is calculated in orbit coordinates after a linear interpolation of the solar array orbit ephemerides(result saved as variable  $\Theta$ ).

$$r : \quad r = \frac{a(1 - e^2)}{1 + e \cos(\Theta)} \quad (3.12)$$

A combination of rectangular to polar conversions are made to rotate for orbit inclination, ascension node, and earth inclination. With the result being outputted as a [x,y,z] coordinate.

### 3.1.4 Main Loop

As mentioned at the beginning of point 3.1, there is a general setup that must be calculated before extracting the necessary values from the simulation. After processing the elements' shape factors mentioned in the previous sub-section, we can proceed to tackle the main problem.

A variable dt is instantiated, which is equal to the period of the orbit divided by the number of time-steps for one orbit. Changing this number can reduce or increase the size of the problem. The loop will iterate through the points in the orbit, saving to several arrays the values of time, direct Kw, reflected Kw, total Kw, temperature of the panels, power generated per Kw and the perceived efficiency.

At the beginning of each iteration, the orbit coordinates are calculated with the above-mentioned orbit function. The direct solar radiation is calculated using the shape factor function( $f_{SUN}$ ). Direct solar radiation is then calculated:

$$qd[j] : \quad qd[j] = f_{SUN} * \zeta * area * t_{SUN}^4 \quad (3.13)$$

Next reflected radiation is calculated. If the direct radiation is equal to 0, then reflected radiation is not calculated as it too will be 0. The reflection received from both the cell side and back side are calculated using the Falbedo function (Shape factor of the sun as reflected off the earth). Terrestrial radiation and cosmic background radiation is also calculated at this point using once again the shfac function and crad function respectively.



With the effective direct solar radiation being received by the solar array, the effective heating rate and heating capacity can be calculated. At the end of each iteration the calculated values are saved and we increment the current time once for the next point. This is repeated until the whole orbit has been iterated through.

To calculate the operating temperature of the array we must equate the power incidence on the array  $P_{in}$  with the power produced, plus that radiated away  $P_{out}$ .

$$P_{in} = \alpha_{solar} + P_{sun} + P_{albedo} + \alpha_{thermal} + P_{thermal} \quad (3.14)$$

## 3.2 Web Development

The development of the web-application is made up of two parts: the AngularJS code and the python web server code. They function together to provide a fast and efficient system. The Python web server doubles as both the API<sup>1</sup> and as the web server. As mentioned in Chapter 2, the design of the python web server is based on what is called a micro-framework. Upon request it serves the static files to browsers and it also allows, in this case, to pull simulation data through REST<sup>2</sup> commands.

### 3.2.1 AngularJS App

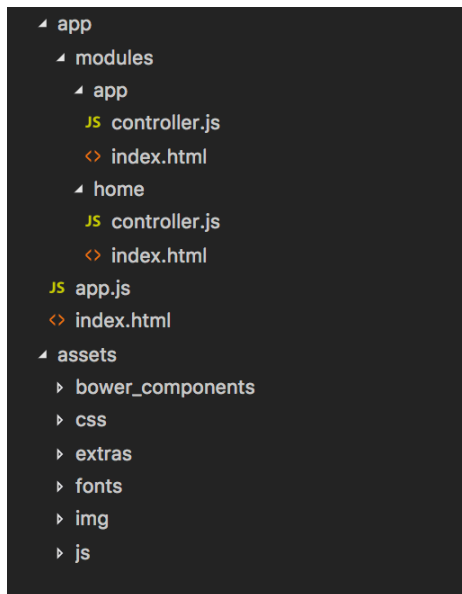


Fig. 3.8 AngularJS project structure

We start with an Angular skeleton. The app folder contains the modules folder. This folder contains all the different pages the application has to offer, each in their View-Controller pair. The *app.js* contains high-level code such as the router which is in charge of routing the user through the different webpages available inside the app. The *index.html* is the static file the web-server hosts which, in turn, loads all the dependencies, including the controllers, css and js files. The index file also contains meta-data for SEO and browser information.

As stated in the design portion of Chapter 2, we will be using two special dependencies to

---

<sup>1</sup>Application Program Interface

<sup>2</sup>Representational State Transfer

draw our simulation. *Three.js* is a cross-browser JavaScript library and Application Programming Interface (API) used to create and display animated 3D computer graphics in a web browser. We will use it in conjunction with *Mathbox.js*, which is a library for rendering presentation-quality math diagrams in a browser using WebGL. Built on top of *Three.js* and *ShaderGraph*, it provides a clean API to visualise mathematical relationships and animate them declaratively.

We start with *mathbox* and create a global variable called *Orbital* into which we will instantiate and append all of our information and parameters.

```
Orbital.mathbox = mathBox({
  plugins: ['core', 'controls', 'cursor', 'mathbox'],
  controls: {
    klass: THREE.OrbitControls,
  },
});
```

Fig. 3.9 Mathbox initialization

Following the documentation and examples available for *Mathbox* we obtain the reference for *Three.js* from the *Mathbox* object. We set the scene attributes, its lighting and create a renderer. We also set-up the view, in this case, a Cartesian view. This allows us to visualise the orbit trajectory as a math function on a grid.

```
//Reference Three.js from mathbox
Orbital.three = Orbital.mathbox.three;
//Set scene attributes
Orbital.three.renderer.setClearColor(new THREE.Color(0x000000), 1.0);
Orbital.light = new THREE.AmbientLight(0xcccccc);
Orbital.three.scene.add(Orbital.light);
Orbital.z_index = 0

//Set lighting
var directionallight = new THREE.DirectionalLight(0xffffff, 0.2);
directionallight.position.set(1000, 0, 0);
directionallight.name = "directional";
Orbital.three.scene.add(directionallight);

//Create Renderer
const renderer = new THREE.WebGLRenderer();
renderer.setClearColor(0x333333);
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(window.innerWidth, window.innerHeight);
```

Fig. 3.10 Mathbox set-up

```
//Camera
Orbital.camera = Orbital.mathbox.camera({
  proxy: true,
  fov: 45,
  position: [1.5, 1.5, 4]
});

//Set mathbox cartesian camera for orbit trajectory visualization
Orbital.view = Orbital.mathbox.cartesian({
  range: [[-1, 1], [-1, 1], [-1, 1]],
  scale: [1, 1, 1],
});
```

Fig. 3.11 Mathbox view set-up

The following steps require us to create 3D objects. Due to the nature of *three.js*, the values for sizes and speeds are all in base(REVISE) of arbitrary values. The size of Earth is an arbitrary unit of 1 in the scene and thus the Moon, for example, has a size of 0.366, as that is the relation between their sizes. All values seen in the configuration are in relation to the size of the Earth, that is, the number 1.

```
// Add Earth
Orbital.earth_geometry = new THREE.SphereGeometry(1, 128, 128);
Orbital.earth_material = new THREE.MeshPhongMaterial();
Orbital.earth_mesh = new THREE.Mesh(Orbital.earth_geometry, Orbital.earth_material);
Orbital.earth_material.map = THREE.ImageUtils.loadTexture('static/assets/img/earthmap_w_clouds.jpg', {});
Orbital.earth_material.specular = new THREE.Color('grey');
Orbital.three.scene.add(Orbital.earth_mesh);

// Add Moon
Orbital.moon_geometry = new THREE.SphereGeometry(0.3668, 64, 64);
Orbital.moon_material = new THREE.MeshPhongMaterial();
Orbital.moon_mesh = new THREE.Mesh(Orbital.moon_geometry, Orbital.moon_material);
Orbital.moon_material.map = THREE.ImageUtils.loadTexture('static/assets/img/moon.jpg', {});
Orbital.three.scene.add(Orbital.moon_mesh);
Orbital.moon_mesh.position.set(60/ $scope.scale, 0, 0);
```

Fig. 3.12 Mathbox object set-up

To implement rotation and animations, we require the use of Quaternion, a number system that extends complex numbers. It has special use in calculations involving three-dimensional rotations such as, as is this case, computer graphics. We instantiate Quaternion in our *three.js* attribute and create a render function which sets a rotation speed for the Earth mesh created beforehand and a Moon mesh position which will slowly move in an ellipsis around the Earth.

```
//Axis
const axis = new THREE.Vector3(0, 1, 0).normalize();

//Animation Earth -setup rotation
var quaternion = new THREE.Quaternion();
function render() {
    Orbital.earth_mesh.rotation.y += 0.00001 * $scope.timestep;

    quaternion.setFromAxisAngle(axis, 0.0000001 * $scope.timestep);
    Orbital.moon_mesh.position.applyQuaternion(quaternion);

    renderer.render(Orbital.three.scene, Orbital.camera);
}
```

Fig. 3.13 Quaternion animation set-up for Three.js

### 3.2.2 Mathbox Orbit Math

As previously mentioned, *Mathbox* allows us to draw functions on a 3D grid in realtime. We can use this to our advantage, as an orbit is really a third order equation.

In order to draw an orbit, we need to understand classical orbital mechanics. Orbital mechanics or astrodynamics is the application of ballistic and celestial mechanics. The motion of these objects can be calculated from Newton's laws of motion and Newton's law of universal gravitation. However, we must focus on what compromises an orbit.

It is possible to specify an orbit entirely using a set of 5 parameters. With these 5 parameters, we can specify precisely where an orbit is, how it is oriented in 3-D space, and what size it is. We also have an optional sixth parameter that determines exactly where the satellite is in its orbit at any arbitrary time  $t$ . These 6 parameters are called the Keplerian Elements and they are the following:

- $T_p$  (Epoс): *Epoch time, number which specifies the time at which a 'snapshot' of the orbit is taken.*
- $i$  (Inclination): *Inclination is the angle between the orbital plane and the equatorial plane. By convention, inclination is a number between 0 and 180 degrees.*
- $\Omega$  (Longitude of Ascending Node): *is an angle, measured at the center of the earth, from the vernal equinox to the ascending node.*
- $\omega$  (Argument of perigee): *angle that specifies orientation of the orbit ellipse in the orbital plane.*

- $e$  (Eccentricity): *is the shape of the orbit, 0 being a circle.*
- $a$  (Semi-major axis): *is an Ellipsis longest diameter.*

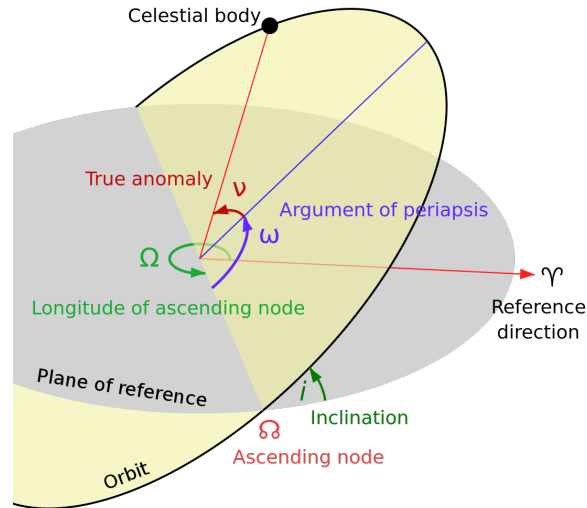


Fig. 3.14 Orbital Mechanics (Elements)

The next challenge is representing an orbit in a coordinate system  $[x,y,z]$ . We use the following equations derived from Newton's laws of motion.

$$r = \frac{a + (1 - e^2)}{1 - e \cos(\theta + \pi)} \quad (3.15)$$

$$x = r \cos(\theta) \quad (3.16)$$

$$y = r \sin(\theta) \quad (3.17)$$

$$x1 = x \cos(\omega) + y \sin(\omega) \quad (3.18)$$

$$y1 = -x \sin(\omega) + y \cos(\omega) \quad (3.19)$$

$$x2 = x1 \cos(i) \quad (3.20)$$

$$y2 = y1 \quad (3.21)$$

$$x3 = x2 \cos(\Omega) + y2 \sin(\Omega) \quad (3.22)$$

$$y3 = -x2 \sin(\Omega) + y2 \cos(\Omega) \quad (3.23)$$

$$\text{orbit Equation} = (x3, x1 \sin(i), y3) \quad (3.24)$$

```

Orbital.orbitEquation = function (a, e,  $\theta$ , inclination,  $\omega$ ,  $\Omega$ ) {
  var r = a * (1 - e * e) / (1 - e * Math.cos( $\theta$  +  $\pi$ ));
  var x = r * Math.cos( $\theta$ );
  var y = r * Math.sin( $\theta$ );
  var x1 = x * Math.cos( $\omega$ ) + y * Math.sin( $\omega$ );
  var y1 = -x * Math.sin( $\omega$ ) + y * Math.cos( $\omega$ );
  var x2 = x1 * Math.cos(inclination);
  var y2 = y1;
  var x3 = x2 * Math.cos( $\Omega$ ) + y2 * Math.sin( $\Omega$ );
  var y3 = -x2 * Math.sin( $\Omega$ ) + y2 * Math.cos( $\Omega$ );
  return ([x3, x1 * Math.sin(inclination), y3]);
}

```

Fig. 3.15 Orbital equation implementation

In order to draw the orbit, we need to create a view area inside the *Mathbox* module. We specify certain parameters such as the range and width/height of the area. We then append an array with which we draw a line on its path. The array expression emits the results from the orbital equation. Some parameters need to be passed to radians as the visual sliders use degrees as the unit of measure.

```

.array({
  width: Orbital.samples,
  channels: 3,
  classes: ['widget', 'orbit_' + Orbital.z_index],
  expr: function (emit, i, t, dt) {
    var a = $scope.slider_a * 1;
    var e = $scope.slider_e * 1;
    var inclination = Orbital.deg_to_rad * $scope.slider_i * 1;
    var  $\omega$  = Orbital.deg_to_rad * $scope.slider_w * 1 - Orbital.nhalf;
    var  $\Omega$  = Orbital.deg_to_rad * $scope.slider_omega * 1;
    var  $\theta$  = i * Orbital.sampling_scale;
    var coordinates = Orbital.orbitEquation(a, e,  $\theta$ , inclination,  $\omega$ ,  $\Omega$ );
    emit(coordinates[0], coordinates[1], coordinates[2]);
  },
})
.line({
  color: 0xffffffff,
  opacity: 0.8,
  points: '<',
  width: 6,
  depth: 0.5,
  zOrder: -1000 + Orbital.z_index,
  id: 'orbit_trajectory_' + Orbital.z_index,
  classes: ['widget', 'orbit_' + Orbital.z_index]
})

```

Fig. 3.16 Orbit Visual Implementation

As an additional visual aid we also add arrays with lines for the semi-major and minor axis of the ellipse.

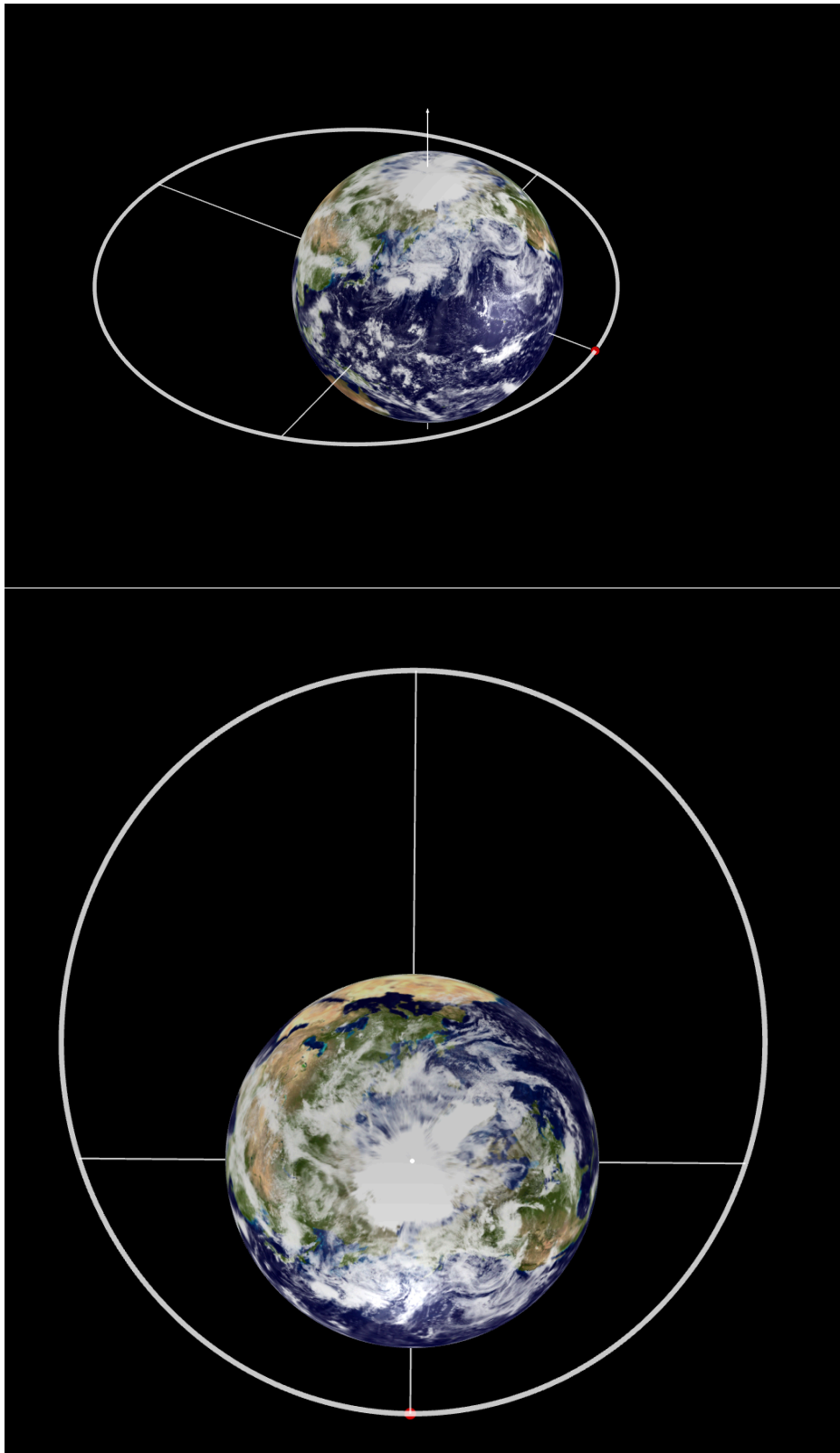


Fig. 3.17 Result of implementation



### 3.2.3 HTML - Design

We start by creating a layout for the App. We will have four main areas. The first, and largest, the visual representation of the space around the planet. This will be in the background. In the foreground there will be three windows, two of which will be toggleable.

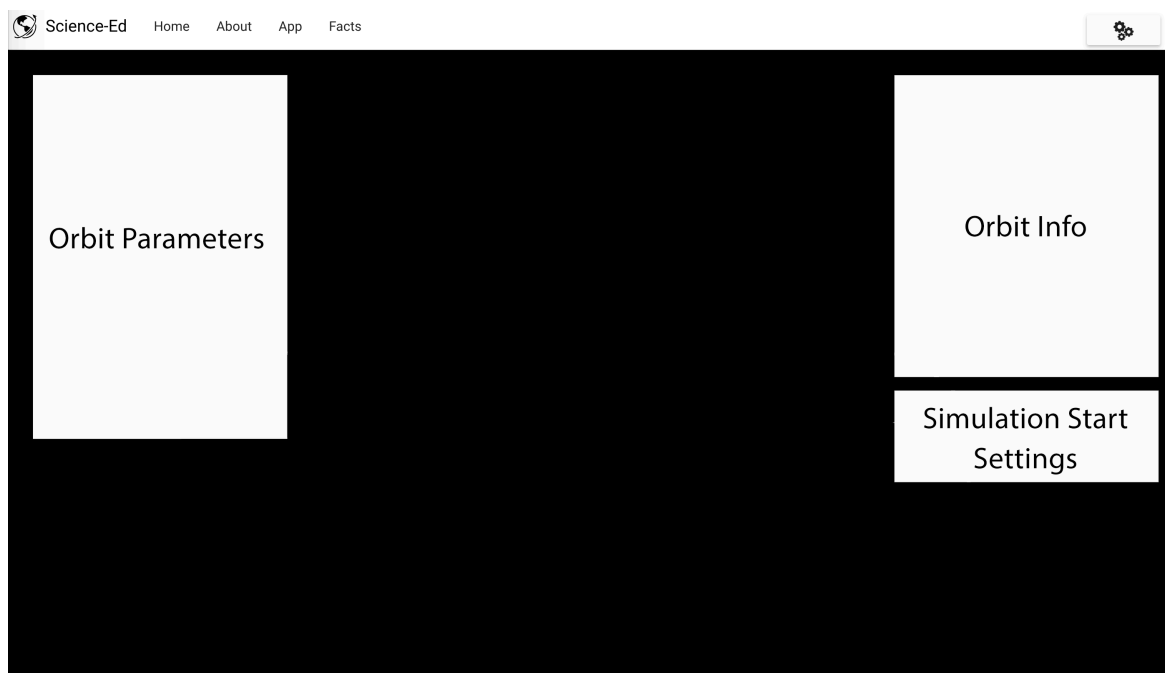


Fig. 3.18 Web app layout

The design of the windows will be mainly based on *angularjs* material design, using the components available from that library. There will be 5 main parameters, all the Keplerian elements except the Epoch, which will be the visual representation of the satellite in orbit. There are also 2 settings modals used in the app. One for the app general settings, and another one for the solar simulation settings. Regarding the calculation of the satellite orbital info:

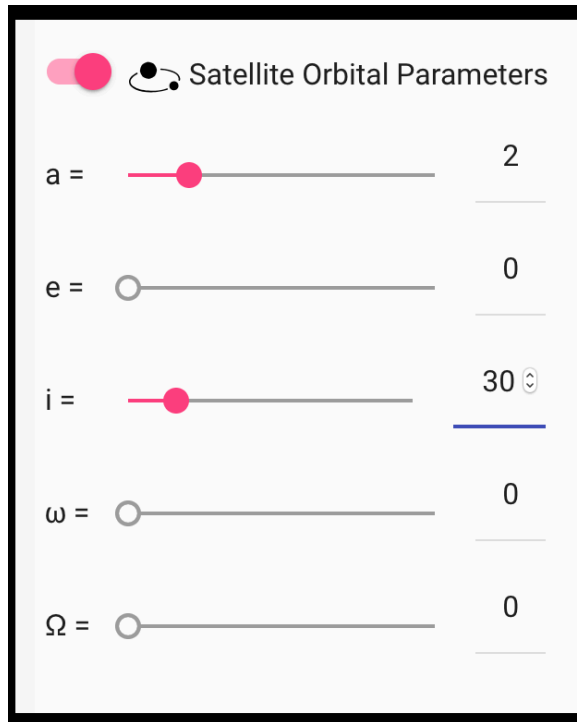


Fig. 3.19 Orbit Parameters view

$$\textit{Apogee} = d * (1 + e) \quad (3.25)$$

$$\textit{Perigee} = d * (1 - e) \quad (3.26)$$

$$\textit{OrbitalSpeed} = \sqrt{\frac{G * M}{r}} \quad (3.27)$$

$$\textit{OrbitalPeriod} = 2\pi \frac{d}{\textit{velocity}} \quad (3.28)$$

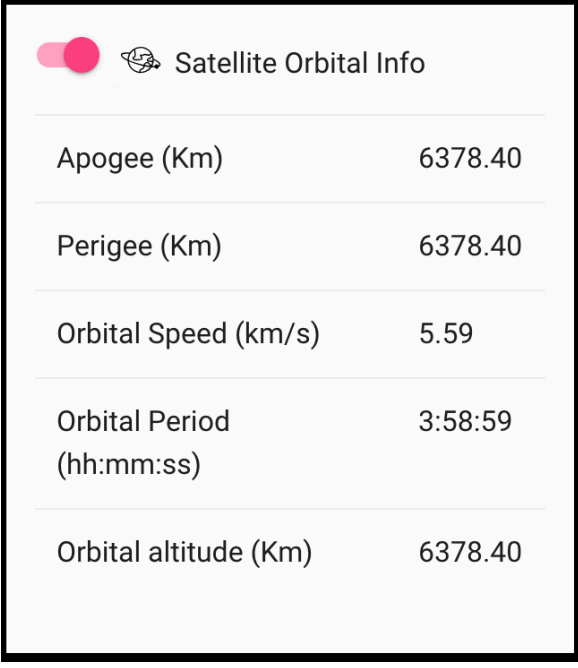
$$\textit{OrbitalAltitude} = d \quad (3.29)$$

$$(3.30)$$

It is important to take into account the radius of the Planet when calculating values. 1 Radius is always subtracted from the distance parameter in the equations.

### 3.2.4 Web Server Code

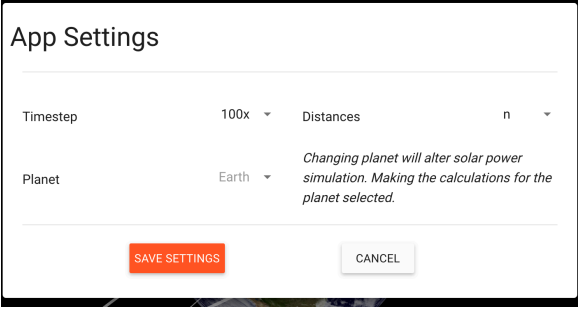
The server Python code is relatively simple. Inside the file run by the server (*server.py*) we import all necessary modules for Flask and we import the array simulation class file. We instantiate two routes for the app. The generic ( '/') returns the render of the index.html. We



The screenshot shows a mobile application interface titled "Satellite Orbital Info". It features a header with a red circular icon and a satellite icon. Below the header is a table with orbital parameters and their values.

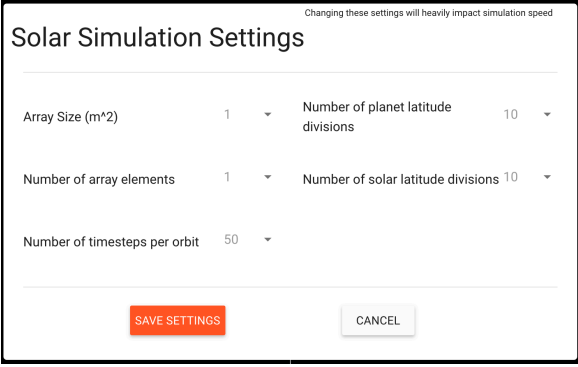
Parameter	Value
Apogee (Km)	6378.40
Perigee (Km)	6378.40
Orbital Speed (km/s)	5.59
Orbital Period (hh:mm:ss)	3:58:59
Orbital altitude (Km)	6378.40

Fig. 3.20 Orbit Info view



The screenshot shows the "App Settings" screen. It includes a "Timestep" dropdown set to "100x" and a "Distances" dropdown set to "n". The "Planet" dropdown is set to "Earth", with a warning message: "Changing planet will alter solar power simulation. Making the calculations for the planet selected." At the bottom, there are "SAVE SETTINGS" and "CANCEL" buttons.

Fig. 3.21 App settings view



The screenshot shows the "Solar Simulation Settings" screen. A warning at the top states: "Changing these settings will heavily impact simulation speed". The settings include: "Array Size (m<sup>2</sup>)" set to 1, "Number of planet latitude divisions" set to 10, "Number of array elements" set to 1, "Number of solar latitude divisions" set to 10, and "Number of timesteps per orbit" set to 50. At the bottom, there are "SAVE SETTINGS" and "CANCEL" buttons.

Fig. 3.22 Solar simulation settings view

then create a route that will be called when requesting the simulation through REST. With this method we save all the arguments sent in the request and call the initialise method of the class which returns the information in an array of objects.

```
@app.route('/')
def hello(name=None):
    return render_template('index.html', name=name)
```

Fig. 3.23 Server default app route implementation

```
@app.route('/simulation')
def simulation():
    phi = float(request.args['phi']) # inclination
    omega = float(request.args['omega']) # argument of perigee
    a = float(request.args['a']) # semi-major axis (km)
    e = float(request.args['e']) # inclination
    array_size = int(request.args['as'])
    date = float(request.args['date'])
    ra = float(request.args['ra']) # right ascension node

    # correct for feet
    a = (a * 6371) * 3280.84
    a = int(a)

    # Settings, parameters for simulation
    na = int(request.args['na']) # number of array elements
    ne = int(request.args['ne']) # number of earth latitude divisions
    ns = int(request.args['ns']) # number of solar latitude divisions
    nt = int(request.args['nt']) # number of timesteps per orbit

    print(phi, omega, a, e, array_size, date, ra, na, ne, ns, nt)

    v = Array_Simulation() # self
    result = v.initialize(a, phi, omega, e, array_size, date, ra, na, ne, ns, nt)
    return json.dumps(result)
```

Fig. 3.24 Server simulation app route implementation



# Chapter 4

## Results

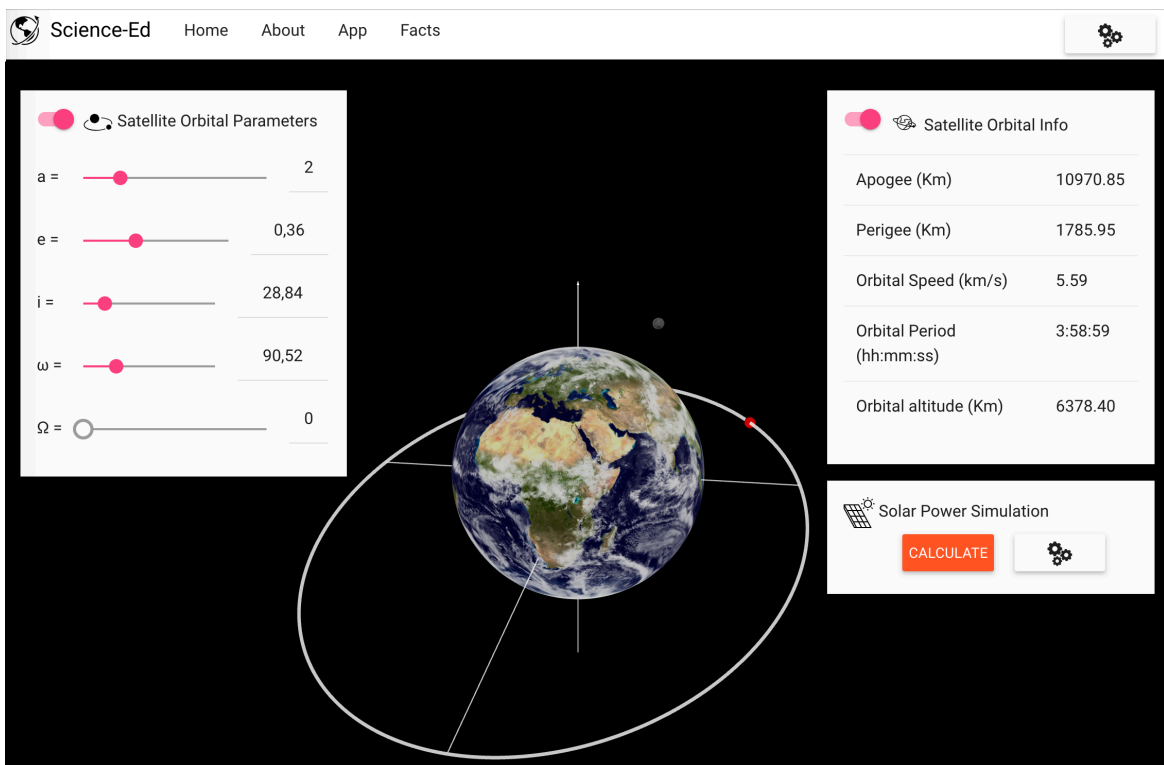


Fig. 4.1 Result of web application

The resulting web-application allows users to visualise orbits accurately and manipulate them as they wish. The orbital info updates accordingly and animations show the satellite accurately moving through the orbit with the planet rotation and moon movement. Clicking the solar simulation calculate button allows the user to visualise the data the simulation outputs. The information can be seen in raw format and graphically within the web-application.

The orbit used for these results was one similar to that of the International Space Station, with a period of 1.52 hours and an orbital altitude of 442km.

Table 4.1 Raw results from array simulation

Time	Direct Kw	Reflected Kw	Total Kw	Temperature	Pwr Gen	Efficiency
0.0000	1.3859	0.0464	1.4323	4.6278	0.1741	27.9588
0.0311	1.3859	0.0813	1.4673	5.7508	0.1775	32.6665
0.0623	1.3860	0.0936	1.4795	6.9273	0.1781	32.5045
0.0934	1.3860	0.0852	1.4712	8.1752	0.1762	32.3326
0.1245	1.3860	0.0711	1.4571	9.4400	0.1735	32.1584
0.1557	1.3860	0.0514	1.4374	10.8082	0.1702	31.9700
0.1868	1.3860	0.0306	1.4166	12.4261	0.1666	31.7472
0.2179	1.3860	0.0138	1.3998	14.1328	0.1634	31.5121
0.2490	1.3860	0.0038	1.3898	15.9343	0.1609	31.2640
0.2802	1.3860	0.0002	1.3863	17.8268	0.1592	31.0033
0.3113	1.3860	0.0000	1.3860	19.7889	0.1578	30.7331
0.3424	1.3860	0.0000	1.3860	21.7822	0.1564	30.4586
0.3736	1.3860	0.0000	1.3860	23.7631	0.1550	30.1858
0.4047	1.3860	0.0000	1.3860	25.6897	0.1536	29.9204
0.4358	1.3860	0.0000	1.3860	27.5202	0.1523	29.6683
0.4670	1.3860	0.0000	1.3860	29.2205	0.1509	29.4003
0.4981	1.3860	0.0004	1.3864	30.7649	0.1499	29.1871
0.5292	1.3860	0.0047	1.3907	32.1393	0.1494	28.9973
0.5604	1.3860	0.0157	1.4017	33.3487	0.1497	28.8303
0.5915	1.3860	0.0332	1.4192	34.4147	0.1508	28.6832
0.6226	1.3860	0.0541	1.4401	35.3646	0.1523	28.5520
0.6538	1.3860	0.0722	1.4582	36.2292	0.1536	28.4326
0.6849	1.3860	0.0878	1.4738	37.0404	0.1546	28.3206
0.7160	1.3860	0.0927	1.4787	37.8419	0.1545	28.2100
0.7471	1.3859	0.0771	1.4631	38.6635	0.1522	28.0965
0.7783	1.3859	0.0396	1.4256	39.5385	0.1477	27.9757
0.8094	1.3859	0.0075	1.3934	40.3255	0.1438	27.8671
0.8405	1.3859	0.0009	1.3868	40.9118	0.1427	27.7861
0.8717	0.0000	0.0000	0.0000	41.3213	0.0000	27.7296

# Chapter 5

## Discussion

In this chapter we will discuss two main areas. First and foremost, the results have to be analysed to make sure they are accurate and secondly, the efficiency of the simulation has to be measured and, if necessary, adjusted. Before diving into the results, we will analyse possible factors that can change the results. The simulation only accounts for satellites which are orientated towards the sun to achieve the best possible efficiencies throughout their orbit. This means that they have active stabilisation either with reaction control thrusters or more commonly reaction control wheels. For larger satellites there are methods for passive stabilisation using the differences in gravity at one point of the satellite compared to another point further away from the centre of mass. In fact, knowing the optimum efficiencies and values, we can stipulate the curves for non stabilised satellites. Chitra Seshan's paper on 'Cell efficiency dependence on solar incidence angle' [5] gives useful insights into the cell efficiencies for different incidence angles. The calculated model from the article can be used to include a variation of options when visualising the solar simulation results.

### 5.1 Analysis of results

As discussed in the first chapter, there is little to no literature or practical data to cross-reference in order to make sure the results are valid. However, panel efficiencies from well known cubesat providers can be compared with the resulting efficiencies from the simulation. Additionally, we can use relevant literature to estimate the amount of solar irradiance that reaches the Earth's atmosphere.

In order to start with these comparisons, we have to understand how the results are returned from the simulation. These results should be divided into two important datasets. The first being the direct kW, reflected kW and total kW that hit the solar panels. This is effectively the amount of available solar energy the panel has at a specific moment during its orbit. A



solar cell with 100% efficiency would generate the total kW value outputted by the results. The second dataset: temperature, power generated and efficiency are all related to a specific solar panel with a specific efficiency. These values are dependant on the solar panel used for the simulation.

Fig. 5.1 Parameters used for simulation results

$$\begin{aligned}
 a &= 12756.8(\text{km}) \\
 e &= 0.0 \\
 i &= 0.0 \\
 \theta &= 0.0 \\
 \omega &= 90.0 \\
 \Omega &= 0.0
 \end{aligned}$$

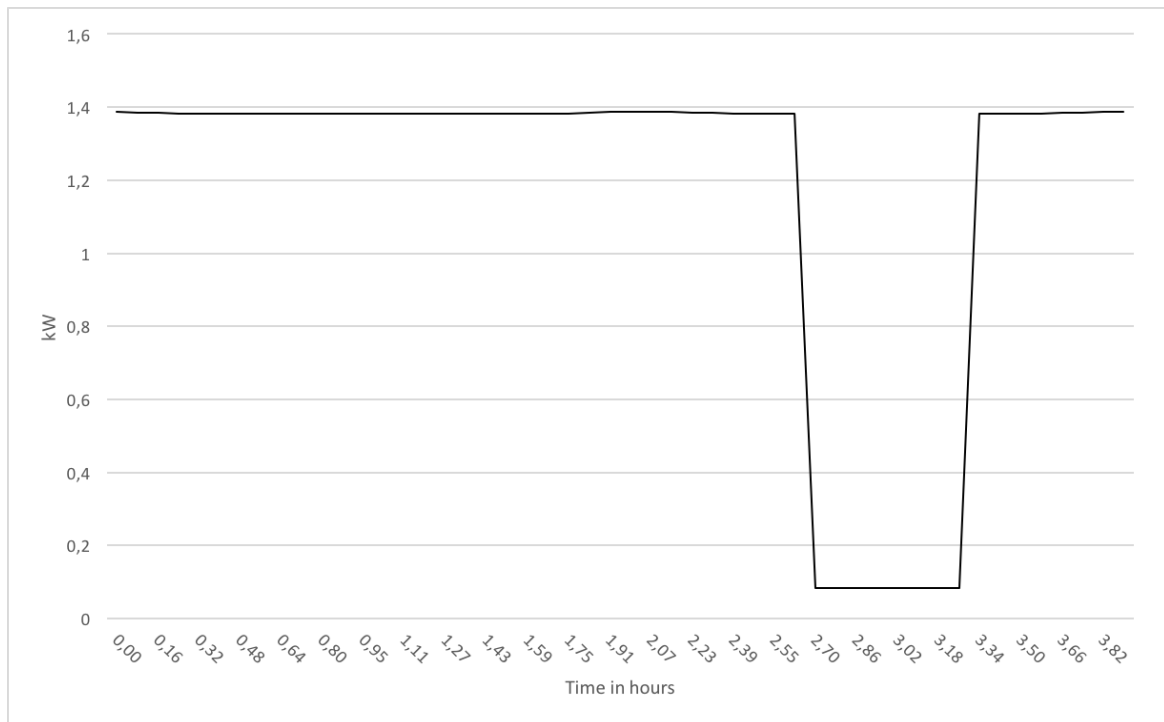


Fig. 5.2 Total solar irradiance received by orbiting satellite as a function of time

Regarding the first dataset, we obtain a direct kW value of approximately 1.38 for a solar panel with an area of  $1\text{m}^2$ . We can calculate the amount of solar energy the sun radiates based on its temperature. With this figure we can initially prove the accuracy of the results comparing the theoretical values with the simulation result. Furthermore, the amount of solar irradiance the Earth receives according to the NREL<sup>1</sup> is  $1.36\text{kW}/\text{m}^2$ . When comparing it to aforementioned 1.38kW, we observe a difference of around 0.014% with the results of the solar simulation. Comparing with the relevant literature mentioned in the state of the art section we observe a difference of around 0.02kW. Furthermore, the data for different orbits were calculated in order to have additional data-points, one of which has an average orbital distance of more than 2 million kilometres from the Earth. This orbit resulted in a

<sup>1</sup>National Renewable Energy Laboratory of the United States of America

max value of direct kW received of 1.42, accounting for an increase of 0.04 kW for a distance of 2 million kilometres. Using the radiation flux law we can calculate exactly the increase in radiation for a set distance and effectively compare results.

Flux,  $S$ , from a star drops off with increasing distance. In fact, it decreases with the square of the radial distance,  $r$ , from the star.

$$S : S = S_0 \left( \frac{r_0}{r} \right)^2 \quad (5.1)$$

The resulting solar irradiance with the flux equation is  $1.40 \text{ kW/m}^2$ , an increase of  $0.04 \text{ kW}$  for a distance of 2 million Km from Earth. This further validates the accuracy of the results obtained from the simulation.

Regarding the efficiency of the solar panels, it can be observed that there is a clear correlation between the temperature of the panels and their efficiency.

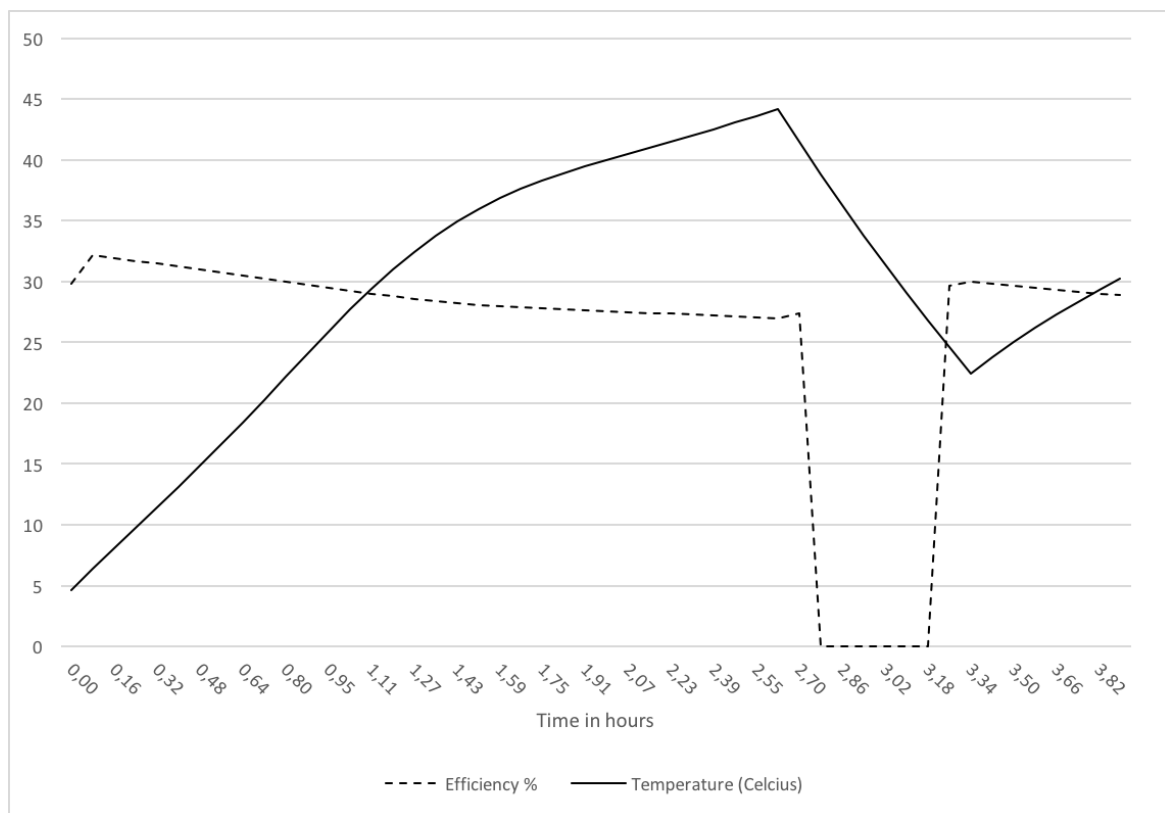


Fig. 5.3 Efficiency of solar panels and temperature as a function of time

The temperature coefficient ( $P_{max}$ ) is the maximum temperature coefficient. It indicates how much power a certain panel will lose when the temperature changes  $1C^\circ$  above or below  $25C^\circ$ . Further investigation yields temperature coefficients ranging from  $\pm 0.13$  to  $\pm 0.19$ [4](Nasa-study). Generally the efficiency change with temperature is non-linear; however, in ranges around  $-100C^\circ$  and  $+100C^\circ$ , efficiency is well-modeled as a linear function of temperature. The results obtained from our simulation yield temperature coefficients of  $\pm 0.1367\%$ , which align well with the literature for solar cell temperature coefficients for Space mentioned previously.

Table 5.1  $P_{max}$  for orbit simulation

Time	Temperature	$P_{max}$
0.79	30.01	0.1360
0.87	29.76	0.1359
0.95	29.51	0.1359
1.03	29.26	0.1359
1.11	29.00	0.1555
1.19	28.78	0.1362
1.27	28.58	0.1363

In conclusion, the results obtained from the simulation are accurate within  $\pm 0.014\%$  when compared with relevant scientific literature. As stated above, various data-points were taken into account for the analysis. Taking into account the objective of the project is to provide an accurate simulation of the amount of power a solar array receives while in orbit, it is important to assure a certain precision in the results. The objective of the project has been reached as the criteria have been met.

## 5.2 Efficiency

When developing web applications there are many important factors to take into consideration, in this section we will mainly focus on response time and in this case, execution time. Due to the interactive nature of the project, user wait time has to be taken seriously into account.

It is important to understand what simulations are in order to consider various factors about the efficiency and consequently program execution time. The very act of simulating something first requires that a model be developed; this model represents the key characteristics, behaviours and functions of the selected physical system. The simulation represents the operation of the system over time. This operation over time is the size of our problem. The bigger the size, the more we divide the time it takes to orbit in segments and therefore, the more data-points the simulation has. The more data-points we have, the more precise our simulation is considered. Accuracy is also important, but in this case we have already discussed the precision of the results the program outputs.

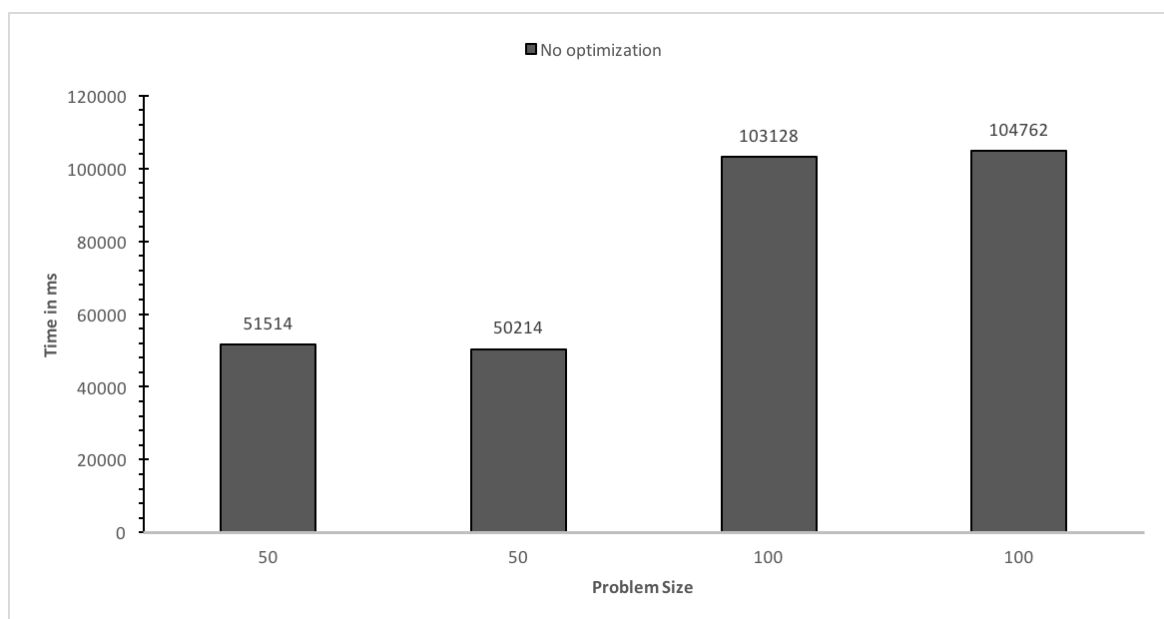


Fig. 5.4 Initial results for program execution times. No optimisation done.

The first results of program execution times were surprising due to various factors. A set of 4 initial runs were timed with two different problem sizes. The first 50 and the second 100. This means that we divided the amount of time it takes the orbital array to complete one orbit by 50 and 100 respectively. As can be observed it takes an average of 50.86 seconds for a problem size of 50. This was initially considered acceptable, however some improvements could still be made to improve execution times. Problem sizes of 50 were considered

accurate enough for LEO<sup>2</sup> and MEO<sup>3</sup>, but anything above GEO<sup>4</sup> would need slightly larger problem sizes to maintain sufficient data-points including orbits with high eccentricity. The code was revised taking a few details into consideration. The first being the use of NumPy functions for non ndarray<sup>5</sup> calculations. NumPy basic operations contain additional code that provides flexibility in order to handle NumPy arrays. Tests were carried out comparing native Python operations with NumPy operations.

```
python -m timeit 'abs(3.15)'
1000000 loops, best of 3: 0.146 usec per loop

python -m timeit -s 'from numpy import abs as nabs' 'nabs(3.15)'
100000 loops, best of 3: 3.92 usec per loop
```

Fig. 5.5 Basic math operation performance difference between NumPy and Python native math module

We observe a 25 fold improvement in performance when using a native python function rather than a NumPy function. However when testing operations for arrays we observe a 28 fold improvement with NumPy functions.

```
python -m timeit -s 'a = [3.15]*1000' '[abs(x) for x in a]'
10000 loops, best of 3: 186 usec per loop

% python -m timeit -s 'import numpy; a = numpy.empty(1000); a.fill(3.15)' 'numpy.abs(a)'
100000 loops, best of 3: 6.47 usec per loop
```

Fig. 5.6 Array math operation performance difference between NumPy and Python native math module

Worst performance usually occurs when mixing python built in math functions with NumPy due in part to type conversions. The code was reviewed to take this into account. For statements were also simplified using the builtin NumPy array functionalities which heavily lower execution times.

<sup>2</sup>Low Earth Orbit, 160km-2000km

<sup>3</sup>Medium Earth Orbit, 2000km-35786km

<sup>4</sup>Geostationary Orbit, 35786km

<sup>5</sup>N-dimensional arrays

```
x[0:2, j] = xo[0:2] + r * vn[0:2, j]
```

Fig. 5.7 'For' loop statements

Additionally, times were observed to be higher when running both the WebGL web application front-end on the same computer as the server. The decision was made to emulate the environment that users would find while using the webpage and therefore only the simulation would need to be run on the server. When accounting for all these changes the following was observed.

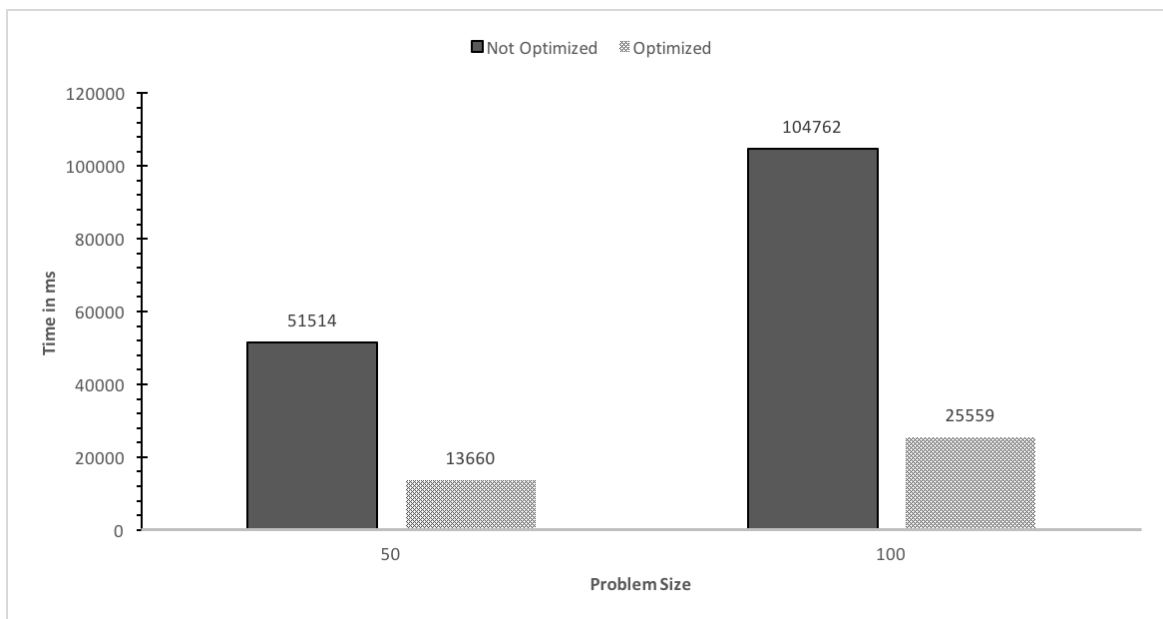


Fig. 5.8 Results for program execution times with optimisation

We can see a 3.77 fold reduction in execution times and like before a linear progression between problem sizes. A 2x increase in size has a 2x increase in execution time. Due to the accuracy of the simulation, a wait time of around 13 seconds was deemed usable. Even for problem sizes where the number of latitude and longitude divisions for the shape factor equations were increased would still yield reasonable execution times.

# References

- [1] Li Junlan and Yan Shaoze. *Aerospace Science and Technology*. Elsevier, 2013, pp. 84–94.
- [2] Hui Kyung Kim and Cho Young Han. *Advances in Space Research*. Elsevier, 2010, pp. 1427–1439.
- [3] Shin Kwang-Bok and Kim Chun-Gon. *Composites Part B: Engineering*. Elsevier, 2001, pp. 271–285.
- [4] G. A. Landis. “Review of Solar Cell Temperature Coefficients for Space”. In: *Proceedings of the XIII Space Photovoltaic Research and Technology Conference (SPRAT XIII) 13* (1994), pp. 385–440.
- [5] Chitra Seshan. “Cell efficiency dependence on solar incidence angle”. In: *Proc.35th IEEE Photovolt. Spec. Conf.* 35 (2010), pp. 2102–2105.



