

# Python Notebook

Contents

**Installation and Setup** ..... 5

1. Anaconda ..... 6

1.1: Introduction to Anaconda ..... 6

1.2: Navigating Anaconda ..... 7

1.3: Installing Packages via PowerShell..... 7

1.4: Advanced Navigation ..... 7

2. Visual Studio Code ..... 8

2.1: Introduction to Visual Studio Code ..... 8

2.2: Managing and Installing Packages (\*) ..... 8

2.3: Working Directory and Creating Code (\*) ..... 8

2.4: Selecting Active Terminal for Running Code (\*) ..... 8

3. Git..... 9

3.1: Introduction to Git ..... 9

3.2: Initialising Git for a Working Directory (\*) ..... 9

3.3: Adding Files and Making a Commit (\*) ..... 9

3.4: Separate Branches and Switching Current Branch (\*) ..... 9

3.5: Branch Logs (\*) ..... 9

**Coding Project Section A: Vector Class** ..... 10

4. Theory ..... 11

4.1: Definition..... 11

4.2: Spherical Co-ordinates and Rotational Matrices ..... 14

5. Implementation ..... 16

**Coding Project Section B: Link Budgeting**..... 19

6. Theory ..... 20

6.1: Introduction ..... 20

6.2: Types of Link Budgeting ..... 20

6.3: The Terminal ..... 22

6.4: The Transmission Medium (Channel) ..... 24

6.5: The Satellite ..... 27

7. Implementation ..... 29

7.1: Terminal Class ..... 29

7.2: Satellite Class ..... 30

7.3: Channel Class ..... 30

**Coding Project Section C: Orbital Calculations** ..... 32

8. Theory ..... 33

**Coding Project Section D: Integration of Steps ..... 35**

9. 10. Theory ..... **Error! Bookmark not defined.**

**References ..... 37**

Table of Figures

Figure 1.1: Environments page with one ‘test’ virtual environment set up. .... 6

Figure 1.2: Environment prompt and ‘Home’ page ..... 7

Figure 1.3: PowerShell Terminal while installing a package ..... 7

Figure 4.1: Vector Add ..... 11

Figure 4.2: Vector Sub..... 11

Figure 4.3: Vector Scalar Multiplication/Division ..... 11

Figure 4.4: Cosine Rule Triangle..... 12

Figure 4.5: Vectors from Cross Product ..... 13

Figure 4.6: Spherical Co-ordinates ..... 14

Figure 5.1: Imports and Copy function ..... 16

Figure 5.2: Key arguments and Initialiser Functions ..... 16

Figure 5.3: Overrides on Basic Operators for Vectors ..... 17

Figure 5.4: Cartesian Conversion and Dot Product..... 17

Figure 5.5: Cross Product and Angle Functions ..... 17

Figure 5.6: Normalising Function and ‘get’ Functions ..... 18

Figure 5.7: Euler Rotation/ Rodriguez to Euler and Matrix ..... 18

Figure 6.1: Parabolic Dish (image source)..... 20

Figure 6.2: Flat Panel Antenna (image source) ..... 20

Figure 7.1: Imports, Initialisation and Private Calc functions ..... 29

Figure 7.2: Override, Setting, and Getting Functions ..... 30

Figure 7.3: Private Calc and Public Properties Function ..... 30

Figure 7.4: Imports, Init, and Setting Functions..... 30

Figure 7.5: Override, Getting, and Properties Functions ..... 30

## Installation and Setup

# 1. Anaconda

## 1.1: Introduction to Anaconda

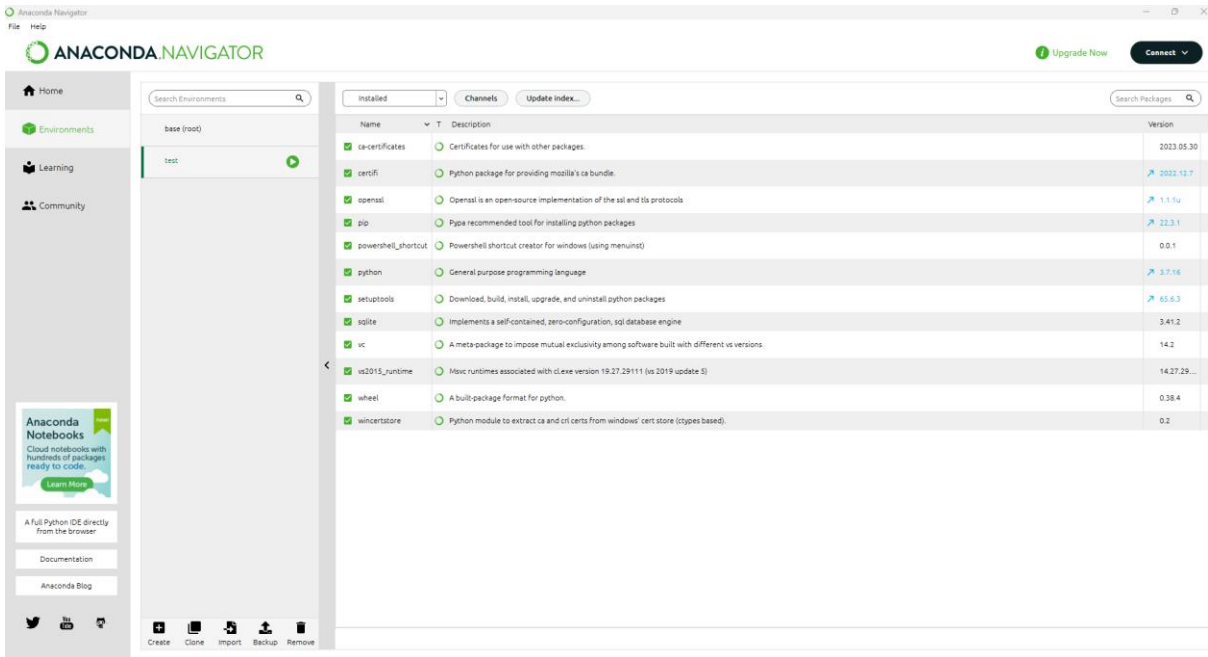


Figure 1.1: Environments page with one 'test' virtual environment set up.

When coding with Python, a massive strength is making use of the massive number of custom packages. However, these packages are not always coded on the most recent version of Python and sometimes depend on the unique way Python interacts with the machine in the older versions. This can cause version mismatches between packages and could otherwise make it impossible to use combinations of packages.

To deal with this, the software Anaconda is used. This allows multiple independent environments of Python to be created. Now we can split packages into their corresponding version on Python e.g., Python 3.7 and Python 3.11. Furthermore, intelligent division of working tasks allows a programmer to run independent tasks in different versions of Python, then use the results of them (could store in a file or alternative) for a final execution of their task. This makes Python a very capable and powerful language. Using Anaconda at this date is new to me but using packages is not.

## 1.2: Navigating Anaconda

Usually when you open Anaconda you will start on the Home Page. This will be discussed a bit later, but it is best to navigate to the Environments page first to create your first environment. To do so click the create button on the bottom on the environments page show in **Figure 1.1**. You will then be prompted with a create new environment window. Anaconda also allows virtual environments of R.

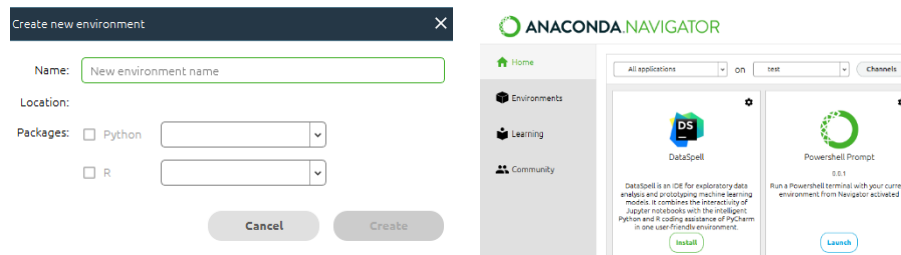


Figure 1.2: Environment prompt and 'Home' page

After naming and choosing your version of Python, you want to navigate to the Home Page, both shown in **Figure 1.2**. Here you can now see all applications installed and available for that virtual environment. You will want to install the PowerShell Prompt as this will be your best friend.

## 1.3: Installing Packages via PowerShell

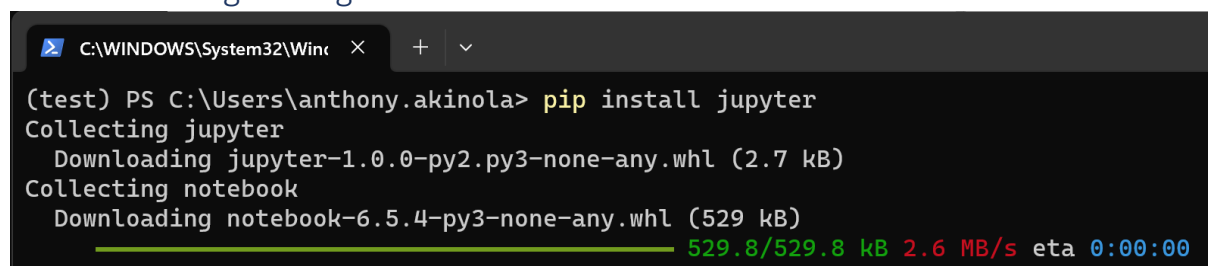


Figure 1.3: PowerShell Terminal while installing a package

To begin installing packages in your Python environment, you want to use the PowerShell Terminal and begin installing by using the command 'pip install' or 'conda install'. It is important to know both commands as some packages are only available on either pip or conda. It is important to note the installed PowerShell Terminal from the Home page will be auto set for the Environment you installed it for!

The basic recommended packages to install are 'pip install numpy pandas' to install 'numpy' and 'pandas', and 'conda install matplotlib jupyter' to install 'matplotlib' and 'jupyter'. A nice package from progress bars is 'tqdm'

## 1.4: Advanced Navigation

It is possible to skip interacting with the Anaconda GUI (Graphical-User Interface) and create virtual environments and additionally install packages via the command prompt. This is much faster and can also lead to faster installs due to, and requires, admin privileges being provided to Anaconda.

## 2. Visual Studio Code

### 2.1: Introduction to Visual Studio Code

Visual Studio Code is just like Notepad++, Atom, etc..., it is simply a Text Editor. However, it has some very useful features via extensions, to be installed allowing the Editor to automatically run code via a specified compiler/executable. This means it is compatible, albeit with the right extensions, with every programming language while requiring minimal adjustments from the user apart from the initial setup.

### 2.2: Managing and Installing Packages (\*)

Select the Extensions menu (3 blocks with 1 block being added icon). You can now search extensions in a search engine innate to VS Code called the Marketplace. This is a verified extension platform so don't worry about versions, etc. To start using Python you will want to install Python, Pylance, Jupyter, Jupyter Notebook as in



### 2.3: Working Directory and Creating Code (\*)

To open a directory, you want to select the open directory button under the Explorer menu (two pages icon). To open multiple directories, use the new directory button instead. You can also add folders by hovering over file on the top bar.

Under this working directory, you can now create files. These can be any type of files specified by their file extension. 'py' for Python script files and '.ipynb' for Python Notebook files.

### 2.4: Selecting Active Terminal for Running Code (\*)

This is not really needed for notebooks as they have an internal kernel which can be selected upon opening the file.

But for normal script files, one has to set the default terminal to run the script file on.



## 3. Git

### 3.1: Introduction to Git

Git is an essential tool to all software developers. It is used for tracking the development of code through iterations of work. It can store snapshots of files at specified instances and allowing a user to reload the state of the files at that date. This makes Git very powerful when it comes to error checking or loading back-ups of code.

### 3.2: Initialising Git for a Working Directory (\*)

Git first needs to be initialised before it can start tracking files. This is done by entering the command 'git init' in the current working directory's terminal shown in. Good practice to do whenever interacting with Git in a directory that has already been set up is to always start with 'git status'. This will provide you with all the files in the working directory and what Git is currently tracking. It will also tell you about the active branch and modifications in files that have previously been committed but that will be explained later.

### 3.3: Adding Files and Making a Commit (\*)

After 'git status' you should notice git will tell you it has x tracked files (first setup will be 0) and y untracked files. The untracked files will be any files, and folders, that are in the working directory, and are not highlighted in a '.gitignore' file which will be covered later.

### 3.4: Separate Branches and Switching Current Branch (\*)

Every git development project starts with a master branch.

### 3.5: Branch Logs (\*)

After making a series of commits on a branch, will become useful to see prior commits and the attached message to each commit. This allows a developer to track the progression of software development.

Coding Project Section A: Vector Class

## 4. Theory

Vectors are practically essential for any form of mathematical geometric modelling. They create Co-ordinate systems which describe the relative position of elements to each other in a space. This section explains the theory behind vectors and highlights why they might be useful to us.

### 4.1: Definition

#### 4.1.1: Basic Operators

Here, we want to define what a vector is or at least will be in this project. Due to the nature of our project dealing with the positions, velocity and point directions of satellites and antennas on the ground, it becomes important to have an efficient data type which allows us to perform common operations to. Mathematical vectors are used to represent most things in a coordinate system but is not a default datatype in Python. Hence, the introduction of a User Defined Class.

The class was downloaded off Git and adjusted for reasons which will be explained later in the 'Implementation' section. Here we will discuss the theory behind a mathematical vector, and its use case.

Starting from the definition, our vector will be a 3-Dimensional Column Vector representing a quantity in the X, Y and Z direction, respectively.  $\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$  shows the definition of a generic vector  $\vec{v}$ .

Again, most will know what a mathematical vector is, but it is worth to quickly sum what operations can be done with it. This lets us understand what such a mathematical structure may be useful to us.

It is useful to understand how addition and subtraction work with vectors and what they mean in a graphical sense. This helps build intuition which is very necessary for when complex manipulation is required.

For addition:  $\vec{a} + \vec{b} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix}$ , which graphically is the same as

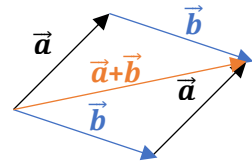


Figure 4.1: Vector Add

We can see addition is additively commutative, this means it results in the same final vector regardless of the order of addition.

Similarly for subtraction:  $\vec{a} - \vec{b} = \begin{pmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{pmatrix}$ , which looks like **Figure 4.2**.

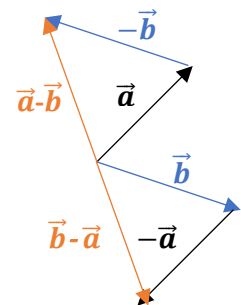


Figure 4.2: Vector Sub

Unlike addition, subtraction's order does matter and an inverse in ordering results in an inverse in direction of the resultant vector. Identifying where order matters will be very important to identify the 'correct positive' direction.

Our other basic operations are scalar multiplication and by extension division (simply multiply by the reciprocal of the scalar number).

For Scalar Multiplication:  $\lambda \vec{a} = \begin{pmatrix} \lambda a_x \\ \lambda a_y \\ \lambda a_z \end{pmatrix}$ , and is shown in **Figure 4.3**.

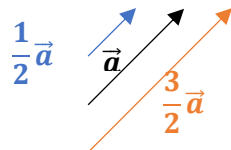


Figure 4.3: Vector Scalar Multiplication/Division

#### 4.1.2: Dot and Cross Product

There are two more commonly used operators for vectors. These are the Dot ( $\cdot$ ), and Cross ( $\times$ ) Product. They have many uses which will be exploited throughout the project. They are operations which involve two vectors (kind of like the addition and subtraction). **The Dot Product** is:

$$\vec{a} \cdot \vec{b} = (a_x \times b_x) + (a_y \times b_y) + (a_z \times b_z). \quad (1)$$

It is commutative meaning the order of vectors does not matter. Using ( 1 ) paired with the **Cosine Rule** [#] (referenced as the derivation is not necessary for this report) we can derive a power tool called the **Dot Product Rule**.

Consider two vectors,  $\vec{a}$  and  $\vec{b}$ , as shown in **Figure 4.2** the difference between the vectors creates a triangle. If we flatten this into a 2-D situation, we notice we can work out the angle between the vectors by using the **Cosine Rule** (also shown in **Figure 4.4**):

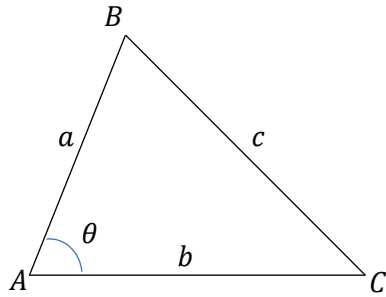


Figure 4.4: Cosine Rule Triangle

$$c^2 = a^2 + b^2 - 2ab \cos(\angle BAC). \quad (2)$$

As the sides of the triangle represent the length of the line, to apply our Vector triangle we must find the length of our vector which is also referred to as the magnitude of the vector. This is typically expressed as finding the two-norm (or norm) of a vector and is found by:

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}. \quad (3)$$

Subbing Equation ( 3 ) into ( 2 ) and assuming Vector  $\vec{AB} = \vec{a}$ ,  $\vec{AC} = \vec{b}$  yields...

$$\begin{aligned} \|\vec{a} - \vec{b}\|^2 &= \|\vec{a}\|^2 + \|\vec{b}\|^2 - 2\|\vec{a}\|\|\vec{b}\|\cos(\theta) \\ (a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2 &= a_x^2 + a_y^2 + a_z^2 + b_x^2 + b_y^2 + b_z^2 - 2\|\vec{a}\|\|\vec{b}\|\cos(\theta) \\ -2(a_x b_x + a_y b_y + a_z b_z) &= -2\|\vec{a}\|\|\vec{b}\|\cos(\theta). \end{aligned}$$

Which finally gives the **Dot Product Rule**:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\|\cos(\theta). \quad (4)$$

This will be very important for finding the angle between vectors. It is also important to note that  $\cos$  will represent the acute angle between vectors. In addition, the direction of the vectors does not matter.

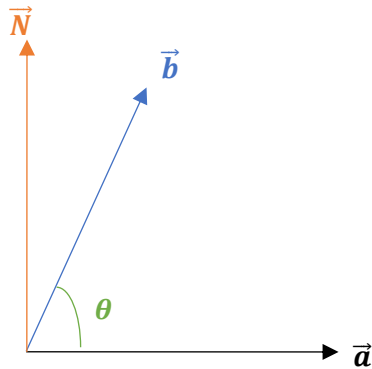
The **Cross Product** is more complex and has its own corresponding rule. The cross product between two vectors is defined as:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \vec{i}(a_y b_z - a_z b_y) - \vec{j}(a_x b_z - a_z b_x) + \vec{k}(a_x b_y - a_y b_x). \quad (5)$$

Where,  $\vec{i}$ ,  $\vec{j}$  and  $\vec{k}$  represent the **Unit Vectors** in directions  $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ ,  $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$  and  $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$  respectively. The **Cross Product Rule** will not be derived here (see [#]) as it is not necessary for this project, however, the definition will be brought up as it teaches something very useful to us:

$$\vec{a} \times \vec{b} = \|\vec{a}\| \|\vec{b}\| \sin(\theta) \vec{n}. \quad (6)$$

Where,  $\vec{n}$  is the **Unit Normal Vector**. This normal vector is important, and the direction of this vector depends on the order of the **Cross Product** hence it is not commutative. Generally,  $\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$ . The direction of the normal vector can be found using the '**Right-hand Rule**' where the first vector represents the direction of a thumb, second vector is the index finger and resultant normal vector is the middle finger, when positioned like **Figure 4.5** where,  $\vec{N} = \vec{a} \times \vec{b}$ , and is the **Normal Vector**.

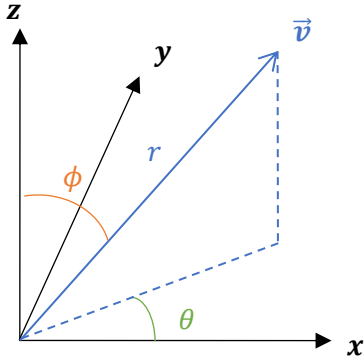


This Normal Vector will always be 90° to the defined **plane** in which both vectors move through. This can be kind of complex to understand but it is basically what we did when we squished our vectors into a 2-D situation during the Dot Product derivation. This plane is where our vectors appear 2-D!

Figure 4.5: Vectors from Cross Product

## 4.2: Spherical Co-ordinates and Rotational Matrices

Sometimes, it is convenient to define vectors using a different set of variables. This could be because of the application we are working with or trying to model, the standard **Cartesian** (x, y, z) are not as intuitive or sometimes not a helpful way of representing our geometry. A popular alternative to 3-D Cartesian Co-ordinates is **Spherical Co-ordinates**. They use the quantities: radius (**r**), polar angle (**θ**) and azimuthal angle (**φ**) to define the geometry as shown in Error! Reference source not found..



It is important to know how to convert between the systems as the methods we defined earlier only work with Cartesian representation. To convert from Spherical to Cartesian we use the series of transformations:

$$\begin{aligned}x &= r \cos(\theta) \sin(\phi), \\y &= r \sin(\theta) \sin(\phi), \\z &= r \cos(\phi).\end{aligned}$$

( 7 )

Figure 4.6: Spherical Co-ordinates

Sometimes the azimuthal angle (**φ**) is taken from the x-y plane to the vector as a kind of elevation angle.

Lastly, we want to be able to rotate our vector. This is typically done via a series of **Euler Rotation Matrices**. They are defined to apply a series of 3 rotations about each axis, A rotation about the **x-axis** (a.k.a. **Roll**), a rotation about the **y-axis** (a.k.a. **Pitch**), and a rotation about the **z-axis** (a.k.a. **Yaw**) each by an angle **φ**, **θ**, and **ψ** respectively. Their form is expressed as:

$$\begin{aligned}\text{Roll} \equiv \overline{\overline{R_x}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}, \\ \text{Pitch} \equiv \overline{\overline{R_y}} &= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}, \\ \text{Yaw} \equiv \overline{\overline{R_z}} &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}.\end{aligned}$$

( 8 )

The order of these rotations is important! A **Z-Y-X** (Means a Roll then Pitch and lastly Yaw) rotation is not the same as a **Y X Z**. rotation.

But suggest we want to redirect to a specific direction, but we do not know how much we need to rotate in each direction to get there. For applications involving some form of manual guidance, the Operator can simply turn until it 'looks right'. But for autonomous guidance, the system needs a mathematical way of determining the exact rotation angles required to reach the destination. This is quite difficult to implement but it is possible using the tools we derived earlier and one more equation.

We start by considering the 2-D situation of the vectors again. In this case we know the angle between the vectors from the Dot Product Rule, but our 'Normal Vector' is not the standard **z-axis**. We need a method to rotate about a given axis, which is defined by a vector direction.

This is done using **Rodrigues' rotation formula** [#], which provides the final vector from the rotation of a vector  $\vec{v}$  about a 'normalised axis vector'  $\vec{n}$  by angle  $\theta$  as:

$$\vec{v}_{rot} = \vec{v} \cos \theta + (\vec{n} \times \vec{v}) \sin \theta + \vec{n}(\vec{n} \cdot \vec{v})(1 - \cos \theta). \quad (9)$$

We are more concerned with the generalised rotational matrix which is found to be:

$$\overline{\overline{R}}_{\vec{n}} = \begin{bmatrix} \cos \theta + n_x^2(1 - \cos \theta) & n_x n_y(1 - \cos \theta) - n_z \sin \theta & n_x n_z(1 - \cos \theta) + n_y \sin \theta \\ n_x n_y(1 - \cos \theta) + n_z \sin \theta & \cos \theta + n_y^2(1 - \cos \theta) & n_y n_z(1 - \cos \theta) - n_x \sin \theta \\ n_x n_z(1 - \cos \theta) - n_y \sin \theta & n_y n_z(1 - \cos \theta) + n_x \sin \theta & \cos \theta + n_z^2(1 - \cos \theta) \end{bmatrix}. \quad (10)$$

The subscript denotes a Rotation about vector  $\vec{n}$

To get this normalised axis vector, we can obtain the normal vector from the **Cross Product** then divide it by its magnitude a.k.a. the **two-norm**:

$$\vec{n} = \frac{\vec{v} \times \vec{v}_{rot}}{\|\vec{v} \times \vec{v}_{rot}\|}. \quad (11)$$

The order of the Cross Product is very important here. It determines the direction of a positive rotation. The convention used here will be  $\overrightarrow{origin} \times \overrightarrow{destination}$ , which tells us how much the original vector needs to rotate to overlap the destination vector.

Finally, we might want to know what combination of **Euler rotations** would have put us at this destination. This could be because our mechanical steering system has 3 degrees of rotation and is designed in a way to take 3 Euler angle inputs. We can find the individual angles required by comparing matrix entries with the final matrix of the individual Euler rotations:

$$\overline{\overline{R}}_{zy'x''} = \begin{bmatrix} \cos \theta \cos \psi & -\cos \phi \sin \psi + \sin \phi \sin \theta \cos \psi & \sin \phi \sin \psi + \cos \phi \sin \theta \cos \psi \\ \cos \theta \sin \psi & \cos \phi \cos \psi + \sin \phi \sin \theta \sin \psi & -\sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi \\ -\sin \theta & \sin \phi \cos \theta & \cos \phi \cos \theta \end{bmatrix}. \quad (12)$$

Note the format of the subscript on the rotational matrix R. It is read from left to right with the right being the first transformation applied to the vector, the same as matrix multiplication. Therefore, it is read as:  $\overline{\overline{R}}_{zy'x''} = \overline{\overline{R}}_z \overline{\overline{R}}_{y'} \overline{\overline{R}}_{x''}$ . By comparing entries from *Equations (10), and (12)* we can determine the magnitude of the individual angles with:

$$\begin{aligned} \phi &= \arctan\left(\frac{R_{32}}{R_{33}}\right), \\ \theta &= \arcsin(-R_{31}), \\ \psi &= \arctan\left(\frac{R_{21}}{R_{11}}\right). \end{aligned} \quad (13)$$

## 5. Implementation

Here we will explain the coded implementation of our vector.

The class was originally downloaded off GitHub then adjusted to add or edit functions for our specific use case.

```
VectorClass.py > ...
1  import numpy as np
2  import math as m
3  from operator import add, sub
4  from numbers import Number
5  import math
6
7  # https://www.meccanismocomplesso.org/en/3d-rotations-and-euler-angles-
8  # Copied from above website
9
10
11 def copy(other):
12     """Returns a copy of vector other
13
14     Returns:
15     (Vector)
16     """
17     return Vector(other.magnitude(), other.theta(), other.phi())
```

Figure 5.1: Imports and Copy function

The packages required for the Vector Class involve 'numpy' for arrays and 'math' for the mathematical functions e.g., 'math.sin()', 'math.cos()' for the cosine and sine approximate functions. The copy function is here just for the ability to copy a Vector as normally, assignment of a variable space to an instance, results in the storage of the memory location of the instance rather than a new copy of the instance. This means, without a copy function you cannot make a duplicate vector.

```
VectorClass.py > ...
18
19
20 class Vector: # Vector class for satellite beam directions.
21     """ Create a 3D Vector.
22
23     Keyword arguments:
24     radius_x -- length in Spherical coordinates, x in Cartesian coordinates (default 0.0)
25     theta_y -- polar angle in Spherical coordinates, y in Cartesian coordinates (default 0.0)
26     phi_z=0 -- azimuthal angle in Spherical coordinates, z in Cartesian coordinates (default 0.0)
27     coords -- identifies coordinate system of arguments as either "spherical" or "cartesian" (default "spherical")
28     """
29
30     def __init__(self, radius_x=0.0, theta_y=0.0, phi_z=0.0, coords="spherical"):
31         if coords == "cartesian":
32             self.__init_cartesian(radius_x, theta_y, phi_z)
33         elif coords == "spherical":
34             self.__init_spherical(radius_x, theta_y, phi_z)
35         else:
36             raise AttributeError(
37                 "Incorrect parameter for coords keyword in Vector __init__")
38
39     def __init_spherical(self, radius, theta, phi):
40         self.__set_magnitude(radius)
41         self.__set_theta(theta)
42         self.__set_phi(phi)
43
44     def __init_cartesian(self, x, y, z):
45         temp_vector = self._to_spherical([x, y, z])
46         self.__radius = temp_vector.magnitude()
47         self.__theta = temp_vector.theta()
48         self.__phi = temp_vector.phi()
49
```

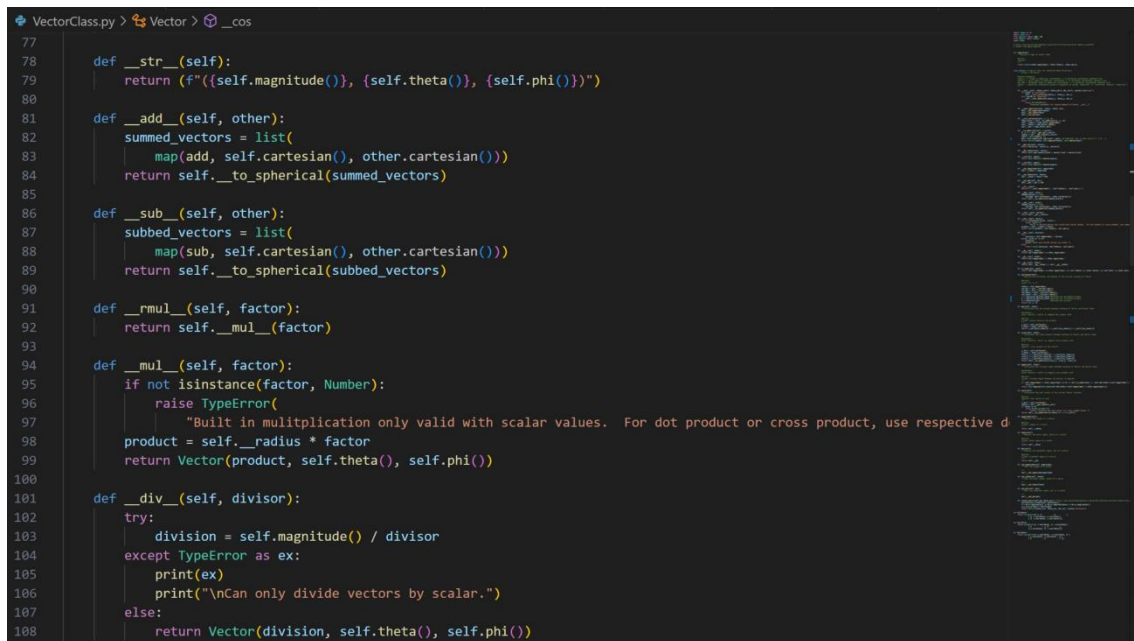
Figure 5.2: Key arguments and Initialiser Functions



The class, to remain consistent, stores vectors in a Spherical Coordinates form. This means the radius, polar angle (theta:  $\theta$ ) and azimuthal angle (phi:  $\phi$ ) are stored as the attributes of the instance. However, when initialising a vector, vectors can be defined with either Cartesian or Spherical Coordinates. This is defined by the 4<sup>th</sup> argument 'coords'. Remember Python does not support function overloads, so instead the initialiser calls different functions dedicated to performing different types of initialisations.

Notice the Cartesian function will call a conversion function to appropriately store the equivalent Spherical equivalents.

Not all functions will be covered here, just the general concepts and the major functions. For example, the conversion functions implementation uses quite a few functions, but the idea is the same as theory, and is mainly used for the result on initialisation.



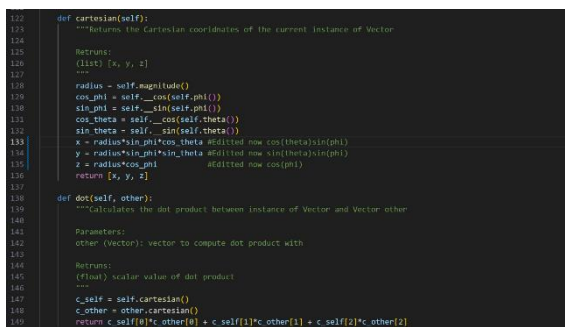
```

77
78 def __str__(self):
79     return f"({self.magnitude()}, {self.theta()}, {self.phi()})"
80
81 def __add__(self, other):
82     summed_vectors = list(
83         map(add, self.cartesian(), other.cartesian())
84     )
85     return self.__to_spherical(summed_vectors)
86
87 def __sub__(self, other):
88     subbed_vectors = list(
89         map(sub, self.cartesian(), other.cartesian())
90     )
91     return self.__to_spherical(subbed_vectors)
92
93 def __rmul__(self, factor):
94     return self.__mul__(factor)
95
96 def __mul__(self, factor):
97     if not isinstance(factor, Number):
98         raise TypeError(
99             "Built in multiplication only valid with scalar values. For dot product or cross product, use respective d
100
101     product = self.__radius * factor
102     return Vector(product, self.theta(), self.phi())
103
104 def __div__(self, divisor):
105     try:
106         division = self.magnitude() / divisor
107     except TypeError as ex:
108         print(ex)
109         print("\nCan only divide vectors by scalar.")
110     else:
111         return Vector(division, self.theta(), self.phi())

```

Figure 5.3: Overrides on Basic Operators for Vectors

These are operation overrides for the basic operations we highlighted in Theory. By default, Python does not know what a Vector instance + another Vector instance should look like. Due to this, an override is required to tell Python whenever it sees an addition between two Vector Class Instances, it should run this code.

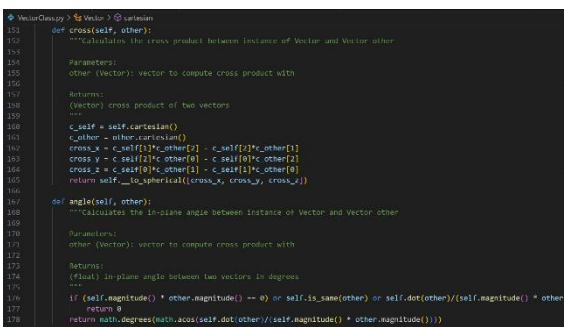


```

122 def cartesian(self):
123     """Returns the Cartesian coordinates of the current instance of Vector
124
125     Returns:
126     (list) [x, y, z]
127
128     """
129     radius = self.magnitude()
130     cos_phi = self.__cos(self.phi())
131     sin_phi = self.__sin(self.phi())
132     cos_theta = self.__cos(self.theta())
133     sin_theta = self.__sin(self.theta())
134     x = radius * sin_phi * cos_theta
135     y = radius * sin_phi * sin_theta
136     z = radius * cos_phi
137     return [x, y, z]
138
139 def dot(self, other):
140     """Calculates the dot product between instance of Vector and Vector other
141
142     Parameters:
143     other (Vector): vector to compute dot product with
144
145     Returns:
146     (float) scalar value of dot product
147
148     """
149     c_self = self.cartesian()
150     c_other = other.cartesian()
151     return c_self[0]*c_other[0] + c_self[1]*c_other[1] + c_self[2]*c_other[2]

```

Figure 5.4: Cartesian Conversion and Dot Product



```

152 def cross(self, other):
153     """Calculates the cross product between instance of Vector and Vector other
154
155     Parameters:
156     other (Vector): vector to compute cross product with
157
158     Returns:
159     (Vector) cross product of two vectors
160
161     """
162     c_self = self.cartesian()
163     c_other = other.cartesian()
164     cross_x = c_self[1]*c_other[2] - c_self[2]*c_other[1]
165     cross_y = c_self[2]*c_other[0] - c_self[0]*c_other[2]
166     cross_z = c_self[0]*c_other[1] - c_self[1]*c_other[0]
167     return self.__to_spherical(cross_x, cross_y, cross_z)
168
169 def angle(self, other):
170     """Calculates the in-plane angle between instance of Vector and Vector other
171
172     Parameters:
173     other (Vector): vector to compute cross product with
174
175     Returns:
176     (float) in-plane angle between two vectors in degrees
177
178     """
179     if (self.magnitude() * other.magnitude() == 0) or self.is_same(other) or self.dot(other)/(self.magnitude() * other.
180     return math.degrees(math.acos(self.dot(other)/(self.magnitude() * other.magnitude())))

```

Figure 5.5: Cross Product and Angle Functions

As all vectors are stored as Spherical Coordinates, for all manipulation and possibly just for viewing purposes, we need a way to represent the vector in Cartesian. This Cartesian function is very important for developing the Dot Product, Cross Product and Euler Rotation functions. Originally,  $\phi$  was defined to be the angle between the X-Y plane rather than Z axis. This can be beneficial when figuring out metrics like elevation angles relative to vectors and more but becomes impractical when dealing with rotations which are defined with Passive Convention in mind. Hence, they were changed to the conversions discussed in Theory.

The angle function makes use of the Dot Product Rule, outlined in Theory, while also having an extra conditional check to avoids errors that may occur. An example of such a error would be if a vector has the magnitude of 0, so it does not really exist but is defined, this would result in a division by zero (see Equation ( 4 ),) after rearranging for  $\theta$ .

```
VectorClass.py > Vector > cartesian
180
181 def unit(self):
182     """Calculates the unit vector of the current Vector instance
183
184     Returns:
185     (Vector) unit vector of self
186
187     c_self = self.cartesian()
188     radius = self._get_radius(c_self)
189     if radius == 0:
190         raise ZeroDivisionError(
191             "Cannot calculate the unit vector of a zero length vector.")
192     return self._to_spherical([x/radius for x in c_self])
193
194 def magnitude(self):
195     """Returns the length of a vector
196
197     Returns:
198     (float) length of a vector
199     """
200     return self._radius
```

Figure 5.6: Normalising Function and 'get' Functions

```
def rotate_vector(self, theta, phi):
    """Rotate a vector by theta and phi angles in radians.
    theta is the angle around the z-axis (yaw) and phi is the angle
    around the x-axis (pitch). Both angles are in radians.
    Returns:
    (Vector) rotated vector
    """
    # Convert angles to radians
    theta = math.radians(theta)
    phi = math.radians(phi)

    # Calculate the unit normal vector
    u_norm = self._unit_vector()

    # Calculate the rotation matrices
    R_theta = math.cos(theta) * u_norm[0][0] + math.sin(theta) * u_norm[0][1]
    R_theta = math.cos(theta) * u_norm[1][0] + math.sin(theta) * u_norm[1][1]
    R_theta = math.cos(theta) * u_norm[2][0] + math.sin(theta) * u_norm[2][1]

    R_phi = math.cos(phi) * u_norm[0][0] + math.sin(phi) * u_norm[0][1]
    R_phi = math.cos(phi) * u_norm[1][0] + math.sin(phi) * u_norm[1][1]
    R_phi = math.cos(phi) * u_norm[2][0] + math.sin(phi) * u_norm[2][1]

    # Apply the rotation matrices
    r_rot = self._rotate_vector(u_norm, R_theta, R_phi)

    # Convert back to spherical coordinates
    r_rot = self._to_spherical(r_rot)

    return r_rot

def _rotate_vector(self, u_norm, R_theta, R_phi):
    """Rotate a unit vector by theta and phi angles in radians.
    Returns:
    (list) rotated unit vector
    """
    # Calculate the rotated unit vector
    r_rot = [0, 0, 0]

    # Apply the rotation matrices
    r_rot[0] = R_theta * u_norm[0][0] + R_phi * u_norm[0][1]
    r_rot[1] = R_theta * u_norm[1][0] + R_phi * u_norm[1][1]
    r_rot[2] = R_theta * u_norm[2][0] + R_phi * u_norm[2][1]

    return r_rot
```

Figure 5.7: Euler Rotation/ Rodriguez to Euler and Matrix

The Unit Function normalises a vector using the same concept of Equation ( 11 ) of dividing the vector by its overall magnitude/two-norm. This will especially be important for the following Euler Rotation functions as we outlined in Theory.

The magnitude and theta functions are simply 'get functions', they provide a method for a user to gain controlled access to the private attributes of radius, theta and phi.

The Euler Rotation function applies Euler rotation matrices in the order of **z-y-x**, The definitions of these matrices are outside of the class at the end.

The last, quite possibly most complex, function discussed is the Rotate to Function. The implementation of this is quite hard coded but, using the concepts explained earlier, can quite easily be adjusted. The function overall co-ordinates all the small steps needed to perform the final rotation of a vector. This consists of finding the Unit Normal Vector, the angle between Vectors, working out the Rodriguez Matrix, then comparing entries to find the equivalent Euler rotational angles, and finally rotating the matrix using the Euler rotation function defined slightly earlier.

Coding Project Section B: Link Budgeting

## 6. Theory

Link budgeting is an important process used to investigate the connection properties of a network topology. It investigates the process of a signal being sent and received by two nodes and how that signal may suffer from interference, noise or other forms of losses. It then calculates an important metric called the **SNR (Signal-to-Noise Ratio)** which is used to evaluate whether the signal is readable.

This section looks to explain/explore the methods of Link Budgeting and why they are important.

### 6.1: Introduction

A lot of this information will be condensed from a source provided by an Industry leading Antenna Company Kymeta Corp [#] and aims to briefly outline the process of the calculations and some of the origins of the losses. This will help us understand how we want to structure the implementation.

The last topic to cover in this Introduction will be discussing what an **FPA** is and how it compares to the conventional **Parabolic Dish**. The conventional antenna used for communications with, especially for geostationary, satellites is the **Parabolic Dish**. This is the usual antenna used for applications like Television and Radio Towers. A picture of one is shown in **Figure 6.1**.



Figure 6.1: Parabolic Dish (image [source](#))



Figure 6.2: Flat Panel Antenna (image [source](#))

They are best used for stationary applications and provide a high **Gain** (will be explained later) while having an easy setup. However, they are highly directional hence, require some form of mechanical steering or a setting angle which adds extra weight and bulk.

Another alternative antenna is the **Flat Panel Antenna (FPA)**, shown in **Figure 6.2**. They work as **Electronically Scanned Antennas (ESAs)**. This involves using an electronically steered beam to track satellites allowing them to be extremely useful in moving applications. They are very low profile, due to not needing an external mechanical antenna, and are very practical for installing on any moving platform. However, they have the drawback of lower **Gains** compared to the **Parabolic Dish**.

### 6.2: Types of Link Budgeting

Network Links are named corresponding to the direction of data flow through them. A forward (**FWD**) link would be one where data begins at the hub, then finds its way to the user terminal. A Return (**RTN**) would be the opposite direction, from user terminal to hub. The **FWD** link is also referred to as **downlink** (or **download link**) and is typically the easier of the two to compute. An example is shown in **Figure 6.3**, Teleport is a Hub and vSat Remote is a user Terminal.

#### 6.2.1: FWD Link. Hub to Terminal

This tends to be the simplest route to calculate due to the assumptions we can make. We usually assume the Hub will always have enough power to drive a FWD carrier up to the satellite while

ensuring constant power output. This means we only need to consider what happens the FWD route from Satellite to Terminal assuming the Satellite transmits with full power.

#### 6.2.2: Simple RTN Link. Terminal to Satellite

This part of the RTN Link is not too difficult to implement. This is because we know a lot about the Terminal's characteristics and just enough about the Satellite's to perform this.

#### 6.2.3: Complex RTN Link. Satellite to Hub

This is the most complex because we can no longer assume the Terminal had enough power to drive the Satellites transmit power to maximum. We would need a method to calculate what the Satellite transmit power would be, which would depend on orientation (angle), distance, weather, and more.

The Project will only cover the Simple Links as the Complex Links would require Confidential Information.

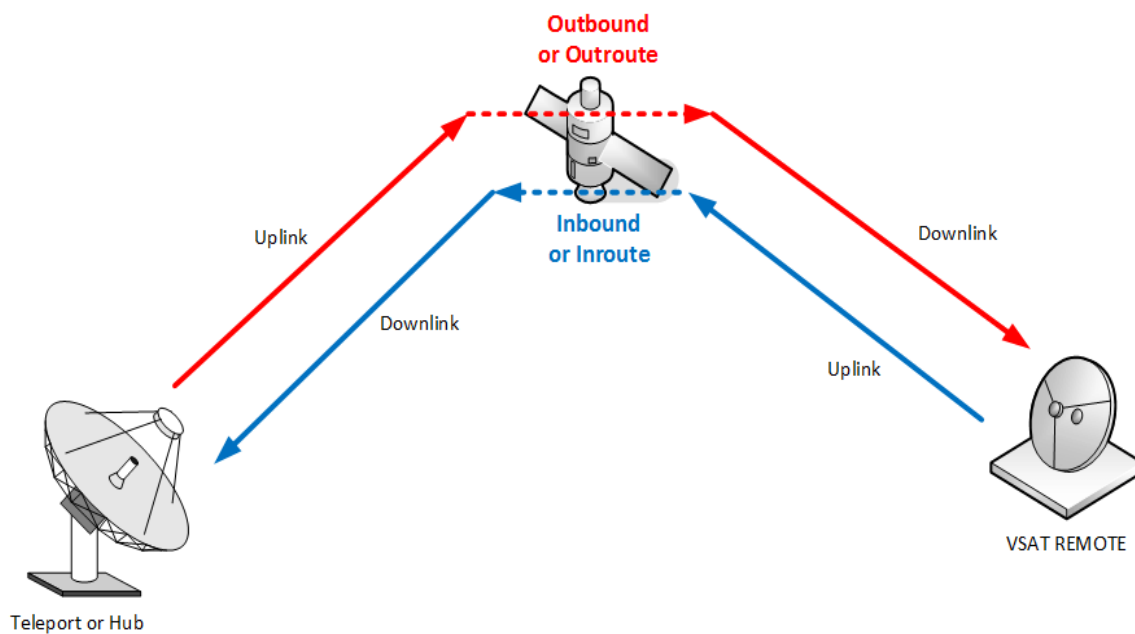


Figure 6.3: Downlink and Uplink Routes ([source](#)).

### 6.3: The Terminal

Everything related to how the Terminal's **Gain (G)**, **G/T**, and **EIRP** are calculated will be explained here.

#### 6.3.1: Antenna Scan Angle and Cosine-Roll-Off

To find the gain of an ESA terminal, an important property is the **scan angle**. Like our prior explanation of Spherical Co-ordinates, this is one of the defining angles for the direction of transmitted beam.

Theta ( $\theta$ ) is the **scan angle**, which is the angle between the boresight vector and the broadside vector as shown in Fig.

Phi ( $\phi$ ) is the second angle which provides the direction of the boresight vector (notice, this is the opposite order of angles in our Spherical definition, but the concept is the same). It can either be measured from the X or Y axis but always from the antennas frame of reference.

For our calculations, we are only really concerned with  $\theta$ .

#### 6.3.2: Gain, Noise Temperature, and G/T

The **Gain (G)** of the Terminal's antenna is easily calculated for both fixed and mobile installations.

$$G(\text{dBi}) = \text{Peak Gain}(\text{dBi}) + \text{cosine roll off} \times 10 \log(\cos \theta).$$

( 14 )

Where:

- **Peak Gain (dBi)** = Peak **Gain** of the antenna and is what would be observed if the broadside and boresight vectors were perfectly aligned ( $\theta = 0$ ), hence no loss of gain due to a reduced effective area seen by a satellite. This metric is provided by the antenna manufacturer.
- **Cosine-Roll-Off** = An antenna coefficient that defines the empirical relation of loss in gain due to increase in scan angle. This is also provided by the antenna manufacturer.

This is an important quantity for calculating the SNR, but we require another quantity before it can be used directly. The quantity that can be used for calculating SNR is called the **gain-to-temperature ratio (G/T)** and to calculate it we need a quantity called the **Noise Temperature** of the terminal.

Three main factors contribute to the overall noise temperature of the terminal:

- Noise temperature of antenna aperture
- Loss of Passive components (such as a diplexer)
- Noise of active units (Such as the LNB)

Measuring the noise temperature of the antenna is difficult as you would need the characteristics of the antenna. However, manufacturers usually provide a value e.g., Kymeta u7 antenna  $\approx 170$  K.

The noise temperature of a passive component is calculated using the equation:

$$T_p = T_e \times (A - 1).$$

( 15 )

Where:

- $T_p$  = **Passive component** Noise Temperature.
- $T_e$  = **Ambient** temperature. For ground terminals, a good assumption would be **290 K**.
- $A$  = **Linear attenuation** of the component. You can infer from the *Equation ( 15 )*  $A \geq 1$

A typical expression of  $A$  would be  $A = 10^{\frac{NF}{10}}$ , where **NF** is the **Noise Figure** for an active, or sometimes the **intrinsic loss** of a passive, component. For example, a **diplexer** (passive component) may experience a loss of 0.15 dB resulting in,  $T_p = 290 \times \left(10^{\frac{0.15}{10}} - 1\right) = 11 K$ . Also, an **LNB** (active component) may experience a loss in SNR where its ratio of output-to-input can be expressed as a **Noise Figure** of 1.0 dB hence, for the active component,  $T_p = 290 \times \left(10^{\frac{1}{10}} - 1\right) = 75 K$

The overall **Noise Temperature** is just the sum of all components and is typically expressed in dB,  $T = 170 + 11 + 75 = 256 K \equiv 24.1 dBK$ . Expressing this in dB allows us to compute the **G/T** by subtracting **T**(dBK) from **G**(dBi):

$$\frac{G}{T} = G(dBi) - T(dBK).$$

( 16 )

### 6.3.3: EIRP and EIRP Density

The **Equivalent Isotropic Radiated Power (EIRP)** is a measure of transmitted power. For the Terminal, it is calculated using the **BUC (Block Upconverter)**, **operating power**, **passive losses** following the **BUC** and the **antenna Gain**.

The **BUC** is responsible for converting the **intermediate frequency (IF)** signal to a frequency signal ready for satellite band transmission. It raises all signals to its specified power output which is typically at the maximum output power where operation is linear. This is typically 1 dB lower than the saturated power.

The **passive losses** are assumed to be only the **diplexer** loss (as the **diplexer** is what allows a satellite to transmit and receive). The **EIRP** is simply the sum of these terms hence:

$$EIRP \text{ of terminal} = BUC \text{ operating power} + \text{passive losses} + \text{antenna gain}.$$

( 17 )

Another useful term can be **EIRP Density** which is used in RTN link budget to compare the **EIRP** to the noise floor in a 1 Hz Bandwidth (**BW**). This is found as:  $EIRP \text{ density} = EIRP - 10 \log\left(\frac{\text{channel BW}}{1 \text{ Hz}}\right)$ .



## 6.4: The Transmission Medium (Channel)

The Transmission Medium's effect on signals via **Free-Space-Path-Loss (FSPL)** and **Atmospheric Loss** will be explained here.

As a signal propagates either through space or any medium, the signal will incur some form of loss. These losses would be the same regardless of direction (FWD or RTN), if they used the same **transmission frequencies** however this usually is not the case hence, there are slight differences.

### 6.4.1: Free-Space-Path-Loss (FSPL)

Free space means a vacuum, space, or clear air (anything that would not significantly hinder the propagation of electromagnetic waves). **FSPL** represents the loss in strength a signal as it travels through a free space.

This is calculated considering two effects:

- Power density (**Intensity**) drop with increasing distance.
- Efficiency of receiving antenna when capturing an incoming signal.

The Intensity-Distance relationship for constant Power Output, can be equated by assuming the output of the signal (transmitter antenna) as a **point source**. This point source propagates energy uniformly in all directions (a.k.a. an **isotropic radiator**). This creates a '**sphere**' of propagation, and assuming no Energy is lost while transmitting, the energy across its surface area must be equal to the point source. This is also known as a form of the **inverse square law** and is shown in **Figure 6.4**.

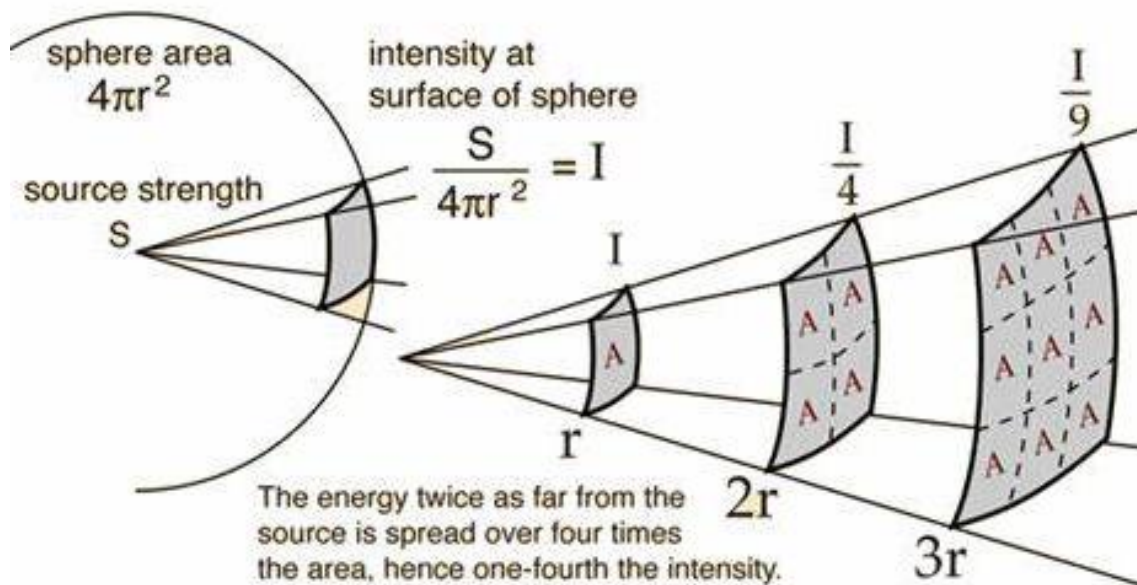


Figure 6.4: Inverse Square law (image [source](#)).

In most cases, applications do not extract all Power from a source, as this would require a complete encapsulation of the point source, so it is useful to know how much power a **unit meter square** would be able to capture from the source. This is referred to as the **Intensity** of the source and can be expressed by the relation:

$$I = \frac{P_t}{A},$$



where,  $I$  is the **Intensity** (or sometimes called the **Power Density  $S$** ) measured in  $W/m^2$ ,  $P_t$  is the Transmitted Power in W, and  $A$  is the **Area** the Power is distributed across.

In the case of a **point source**,  $A$  can be found as the **Surface Area** of a **Sphere** at a **radial distance ( $d$ )** from the source. Hence, the equation for  $A$ :

$$A = 4\pi d^2. \quad (19)$$

Taking the **distance** in metres and substituting Equation ( 19 ) into ( 18 ) yields:

$$I = \frac{P_t}{4\pi d^2}. \quad (20)$$

The efficiency of the receiver antenna is usually expressed as a function of the signal's wavelength ( $\lambda$ ):

$$P_r = \frac{I\lambda^2}{4\pi}, \quad (21)$$

where,  $P_r$  is **Power Received** measured in W, and **Wavelength ( $\lambda$ )** is measured in m. It can also be expressed in terms of **frequency ( $f$ )** using the speed of light relation  $c = \lambda f$ . Now combining Equations ( 20 ) and ( 21 ), and rearranging to express the **ratio** between transmitted and received power, the expression for **Free Space Path Loss (FSPL)** found as:

$$FSPL = \frac{P_t}{P_r} = \frac{(4\pi d)^2}{\lambda^2} = \frac{(4\pi df)^2}{c^2}, \quad (22)$$

where,  $c$  is the **speed of light** in a **vacuum** and is measured in m/s, and **frequency ( $f$ )** is measured in Hz (1/s). This is usually expressed in logarithmic (**decibel**) format for immediate use in SNR calculations like over metrics and for satellite communications in particular, express units in **GHz** for frequency and **km** for distance:

$$FSPL = 20 \log(d) + 20 \log(f) - 20 \log\left(\frac{4\pi}{c}\right),$$

$$FSPL = 20 \log(d) + 20 \log(f) + 92.45. \quad (23)$$

#### 6.4.2: Atmospheric Loss

Atmospheric Loss accounts for the effects on the signal as it travels through the atmosphere of the Earth. This is no longer Free Space, and the signal must travel through various mediums.

The atmosphere is made up of various molecules which collectively make up the definition of air. These molecules have different excitation frequencies, which are frequencies they absorb the most energy from in attempt to reach a higher energy state. **Figure 6.5** displays the relationship between absorption rate and frequency while outlining the key molecule responsible for each peak.

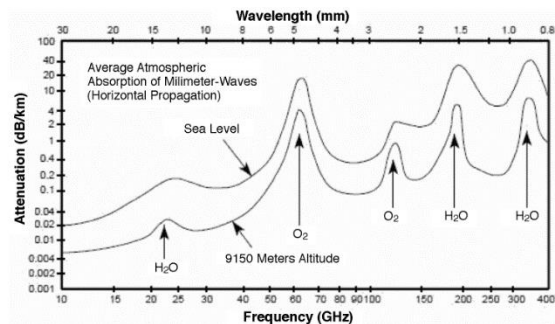


Figure 6.5: Atmospheric Absorption ([#]).

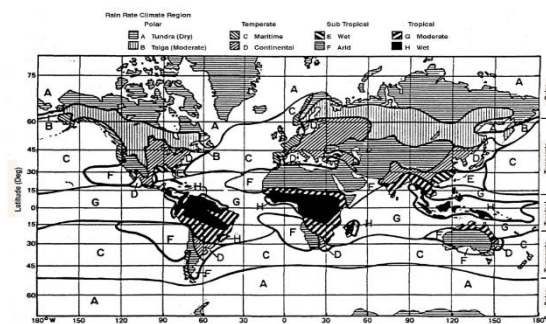


Figure 6.6: Rain Rate Climate regions ([#]).

Avoiding these peaks are important for minimising signal losses, so most communication happens in frequency ranges that avoid massive losses. The general standard is defined by **IEEE (Institute of Electrical and Electronics Engineers)** as the **K band** which is divided into three sub-bands:

- **Ku band (12-18 GHz)**: Primarily used for satellite communications, broadcast television and microwave-based communications.
- **K band (18-26.5 GHz)**: Used for mainly short-range applications due to the inclusion of the 22 GHz spike.
- **Ka band (26.5-40 GHz)**: Primarily used for radar and limited satellite communications applications.

Clouds present regions of high-water vapor concentration in the atmosphere. This affects the strength of a signal much larger than travelling through still air, even including the different layers of the atmosphere. This effect is also known as **rain fade**. It is affected by many factors but most notably rainfall rate and geographical location. The latter is due to different humidities and rainfall conditions in different climate zones around the world. A popular model for this is the **Crane model** which is often used for rain fade computations and is shown in **Figure 6.6**.

There are many equation models made and used to help calculate signal degradation from these atmospheric models and there is further explanation of the crane model in the **Kymeta Document** [#]. It is important to note a key parameter that is referred to by Kymeta as the **Availability**. This parameter, which will be referred to later as the **p** value (and in some other literature is the **Incidence** or **Avoidance Rate** depending on definition), refers to the percentage of time that a satellite link will close. To elaborate, an **Availability** of 99.5% would mean only 0.5% of the time will the rain fall be severe enough to break the connection link.

## 6.5: The Satellite

Everything related to how the Satellite works will be explained here. In practice it is like the Terminal but with a few more nuances.

### 6.5.1: Transponder EIRP

This will be the only addition discussed here as it is quite a significant factor even for the simple RTN Link calculation. The other factors have a significant role in the complex calculations but that is out of this projects scope.

Satellite's antennas are typically optimised in innovative by engineers to maximise coverage shape and strength over important regions. This is especially prevalent in GEO satellites due to their static nature with respect to Earth, but some LEO satellites also make use of this custom beam shape if the satellites orbit path allows it.

Due to this, satellites typically have a unique roll off (like cosine roll off) where their effective EIRP changes depending on the distance from the beam centre. Typically, coverage maps are drawn with contour lines indicating regions of constant EIRP, which are usually not circular shapes. An example of this is shown in **Figure 6.7**.

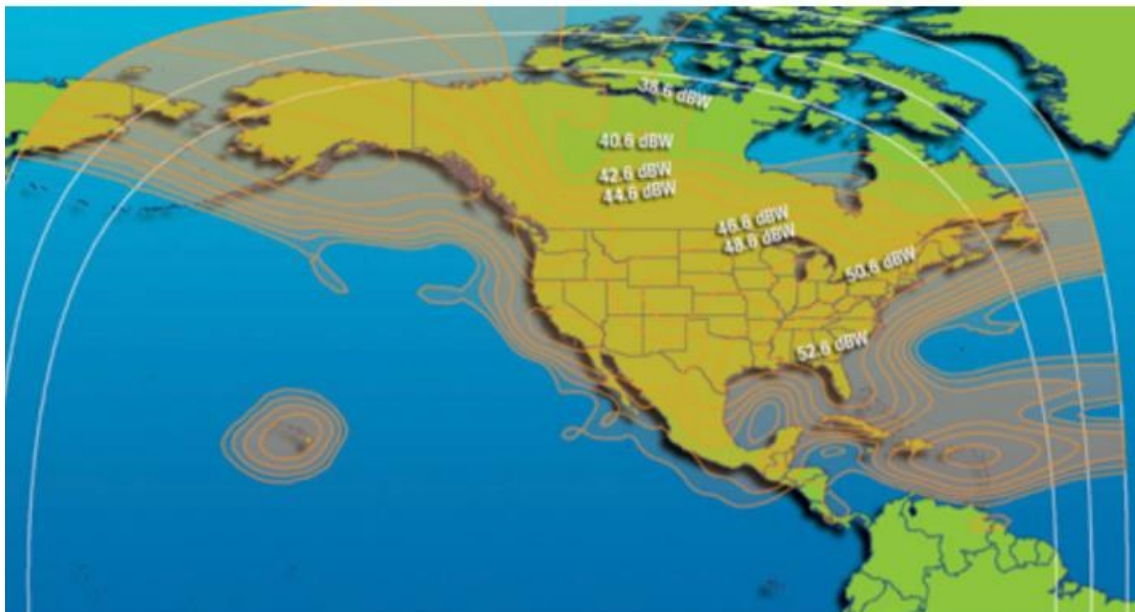


Figure 6.7: Coverage Map for the Galaxy 18 Satellite ([#])

Coverage maps are typically published to help keep sensitive information about the satellites private. Otherwise, equations for the beam shape and drop in Transponder EIRP with deviation in longitudinal and latitudinal positions would need to be provided.

## 6.6: Calculating SNR and Converting to Capacity

This is where all the individual components are added up to find the **Signal-to-Noise Ratio**.

### 6.6.1: FWD Equation

This is a simple calculation as we assume the satellite has enough power to drive the signal at full power. At worse, the coverage maps provide a value that can be used for the Transponder EIRP, but this would normally be more significant when calculating the complex RTN (which as stated will not be covered).

$$\begin{aligned} SNR (dB) = & \text{satellite transponder EIRP (dBW)} - \text{transponder BW (dBHz)} - FSPL (dB) \\ & - \text{atmospheric loss (dB)} + \frac{G}{T} \text{ of terminal (dBK}^{-1}) - k (dBWK^{-1}Hz^{-1}). \end{aligned} \quad (24)$$

### 6.6.2: Simple RTN Equation

This is almost an identical equation to FWD's but simply reversing the order.

$$\begin{aligned} SNR_{up} (dB) = & BUC \text{ operating power (dBW)} + G (dBi) - \text{channel BW (dBHz)} - FSPL (dB) \\ & - \text{atmospheric loss (dB)} + \frac{G}{T} \text{ of satellite (dBK}^{-1}) - k (dBWK^{-1}Hz^{-1}). \end{aligned} \quad (25)$$

Simplifying the first two terms to be the terminal's EIRP yields:

$$\begin{aligned} SNR_{up} (dB) = & \text{terminal transponder EIRP (dBW)} - \text{channel BW (dBHz)} - FSPL (dB) \\ & - \text{atmospheric loss (dB)} + \frac{G}{T} \text{ of satellite (dBK}^{-1}) - k (dBWK^{-1}Hz^{-1}). \end{aligned} \quad (26)$$

It is interesting to spot here that increasing **Bandwidth** for both equations result in a decrease in **SNR**. However, an increase in Bandwidth usually results in an increase in connection capacity which can be shown from the **Shannon- Hartley theorem** [#].

### 6.6.3: Capacity

The theoretical limit for capacity is given by the **Shannon limit**. This limit, realistically, is never achieved and there are established data table standards which provide the recommended bandwidth and connection properties for a given SNR. For this project, the [DVB-S2X](#) standard was used. This also helps define the suitable **MODCOD (Modulation and Coding)** scheme.

The modulation and coding scheme dictate how the signal is sent/received and can control the phase angle between signals and how many signals per message. They provide some important advantages, but the main advantage is improving the reliability of a connection link as they have techniques, such as additional redundant information being sent, which allow the receiver, to an extent, to preform error correction in received data without needing to request a retransmission from the sender.

## 7. Implementation

The Implementation was done in an Object-Oriented Programming (OOP) architecture. This was done to increase readability, robustness and reusability.

The structure to perform a Link Budget Calculation involves the interaction between 3 Classes in a main function. The 3 Classes are Terminal, Channel, and Satellite Class following a similar structure/encapsulation as the information was presented in the Theory Section. The last function is called Link Budget and exists in a main file.

This section looks to discuss the structure of each class and the main function.

### 7.1: Terminal Class

The Terminal Class needs standard imports in addition with VectorClass, which we defined earlier.

**Figure 7.1**, displays the imports, initialisation, and private functions of Terminal.

```
import numpy as np
import math as m
import pandas as pd
import VectorClass as vc

class Terminal: #Terminal class for storing properties of a terminal.
    """ Define a User Terminal.

    Key Attributes:
    Geo - Stores the Longitude and Latitude (currently assumes Elevation is zero but can easily be changed)
    Position - Stores the Current position of the terminal as a VectorClass object
    Pointing - Stores the pointing direction of the terminal, it should point towards the most 'optimal' satellite but can be normal to earth

    Key Methods:
    Properties(angle) - Performs a few functions to calculate the G/T, Gain, and EIRP of the Terminal for the given angle (degrees)
    """

    #Class attributes
    __Antenna_Noise_Temperature = 170 #K
    __Ambient_Temperature = 290 #K
    __Diplexer_Loss = 0.15 #dB
    __NF = 1.0 #dB
    __Peak_Gain = 33 #dB
    __Cosine_Roll_Off = 1.2

    def __init__(self, User, geo):
        self.__User = User
        self.__Geo = geo
        self.__Calc_Passive_Noise()
        self.__Calc_Active_Noise()

    ## Private functions
    def __Calc_Passive_Noise(self):
        self.__Passive_Noise = self.__Ambient_Temperature * (10**(self.__Diplexer_Loss / 10) - 1)

    def __Calc_Active_Noise(self):
        self.__Active_Noise = self.__Ambient_Temperature * (10**(self.__NF / 10) - 1)
```

*Figure 7.1: Imports, Initialisation and Private Calc functions*

The Terminal has defining characteristics which affect how it behaves in a communication link. Hidden attributes such as Antenna Noise Temperature, Ambient Temperature, Diplexer Loss, Noise Factor, Peak Gain, and Cosine Roll Off, are properties of the Terminal which are in some way important for the overall calculation of SNR. Most, if not all, of these metrics should be known as they would either be directly measurable or provided by the manufacturer.

To ensure a level of robustness, default values have been used to initialise these attributes. However, methods to change them and retrieve them are still necessary to implement as shown in **Figure 7.2**.

The Terminal also uses these values to compute, more immediate quantities to be used for SNR calculation. This involves finding the Terminal's EIRP, Gain and by extensions, Gain to Noise Temperature Ratio. These are done by private functions with the final outputs needed for SNR calculations provided by the 'Properties' function in **Figure 7.3**.

[illegible]

Figure 7.2: Override, Setting, and Getting Functions

```

90 # Calculate Noise Temperature function: Takes no inputs and outputs the total Noise temperature in dBK
91 def _Calc_Noise_Temperature(self):
92     Kelvin = self._Antenna_Noise_Temperature + self._Passive_Noise + self._Active_Noise
93     Noise_Temperature = 10 * m.log10(Kelvin)
94     return Noise_Temperature
95
96 # Calculate Gain function: Takes one input, angle in degrees and outputs the terminal Gain in dBK
97 def _Calc_Gain(self, angle):
98     rad = m.radians(angle)
99     Gain = self._Peak_Gain + self._Cosine_Roll_Off * 10 * m.log10(abs(m.cos(rad)))
100     return Gain
101
102 def Properties(self, angle):
103     """Calculates the Terminal's Link Budget Properties.
104
105     Parameters: Beam Angle in degrees
106
107     Returns: (array) [G/T, G, EIRP]"""
108     T = self._Calc_Noise_Temperature()
109     G = self._Calc_Gain(angle)
110     BWC = 9.00
111     EIRP = BWC - self._Diplexer_Loss + G
112     return [G - T, G, EIRP]

```

Figure 7.3: Private Calc and Public Properties Function

## 7.2: Satellite Class

The Satellite Class operates, fundamentally, just like the Terminal Class. However, there are slight variations to deal with the specific nature of a satellite. Similar attributes such as Cosine Roll-Off and Peak Gain, remain the same, however, the Uplink and Downlink Frequencies are also stored as attributes in a Satellite Object as shown in **Figure 7.4**. This is because Satellites typically have a specified frequency they accept Uplinks from and a fixed frequency for sending Downlinks.

[illegible]

Figure 7.4: Imports, Init, and Setting Functions

```

10 def __init__(self):
11     return (self.__name__, (self.get_longitude(), self.get_latitude()))
12
13 # Getting functions
14 def get_longitude(self):
15     return self.__longitude
16
17 def get_latitude(self):
18     return self.__latitude
19
20 def get_Position(self):
21     return self.__Position
22
23 def get_longitude(self):
24     return self.__longitude
25
26 def get_lat(self):
27     return self.__lat
28
29 #
30 #
31 #
32 # calculate data function: Takes own input, angle in degrees and outputs the Terminal data in dB
33 def __calc_data(self, angle):
34     #  $\mu = \text{radius}(\text{angle})$ 
35      $\mu = r_{\text{radius}}(\text{angle})$ 
36      $\text{Gain} = \text{self} \cdot \text{Pmax\_Gain} + \text{self} \cdot \text{Const\_Link\_Off} * 10 * \text{w\_log}(10 \cdot \text{dbm}(\text{m}/\text{Pd}))$ 
37     return Gain
38
39 def Properties(self, angle):
40     """calculates the Satellite's Link Budget Properties.
41
42     Parameters: Beam angle in degrees
43
44     Returns: Array [Pr, G, LRP]"""
45     G = self.__G
46     G = self.__calc_gain(angle)
47     LRP = self.__LRP
48     return [G, L, LRP]

```

Figure 7.5: Override, Getting, and Properties Functions

**Figure 7.5** is almost the exact same as Terminal's Get and Properties definitions. The re use of the function naming and return type will be vital for the reusability in the Channel Class definition which will be explained next.

### 7.3: Channel Class

The Channel Class practically co-ordinates the interaction that would occur between a user terminal and satellite, and the medium the signal would travel through, during a communications link.



Coding Project Section C: Orbital Calculations



## 8. Theory

This involved getting familiar with the information available about current satellites on CelesTrak; and figuring out how to manipulate such information in a useful way using Python's Starfield package.

## 9. Implementation

The Implementation

Coding Project Section D: Integration of Steps

## 10. Theory

This mainly involves iterating through time, saving using Python databases, and experimenting with additional implementations to increase the accuracy of the simulation.

## References

- Kymeta Corporation. (2019, June 1). *Link Budget Calculations for Satellite Link with an Electronically Steerable Antenna Terminal*. Retrieved from KymetaCorp:  
<https://www.kymetacorp.com/wp-content/uploads/2020/09/Link-Budget-Calculations-2.pdf>
- Wikimedia Foundation, Inc. (2022, November 17). *Rodrigues' rotation formula*. Retrieved from Wikipedia: [https://en.wikipedia.org/wiki/Rodrigues%27\\_rotation\\_formula](https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula)
- Wikimedia Foundation, Inc. (2023, May 16). *Law of cosines*. Retrieved from Wikipedia: [https://en.wikipedia.org/wiki/Law\\_of\\_cosines](https://en.wikipedia.org/wiki/Law_of_cosines)