# **Python Bootcamp – Programming Exercise**

Sean Morrison, Alex Fisher, Fabio Zambetta

In this exercise, we’ll go through a worked (and relevant) example: quadrotor simulation. Go to the UAS Team Google Drive, download the file exercise\_1.zip, and unzip it into a project folder. Make this folder your working directory in Spyder. You should see the python scripts **animation.py**, **config.py**, **quadrotor.py**, and **validate\_sim.py** in your directory.

This simulation consists of several components:

1. **A config file containing aircraft parameters**;
2. **Motor thrusts and torques**;
3. **Aerodynamic forces and moments acting on the aircraft**;
4. **The 6DOF equations of motion**;
5. **Rotation matrices to go from the inertial frame to the body frame, and back again**;
6. **An integration technique**; and finally,
7. **A main loop**, and **a plotting routine so that we can see what the aircraft is doing.**

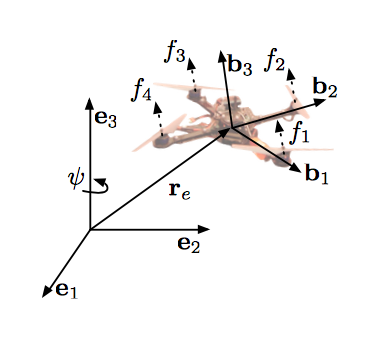
You will be implementing the components written in green; with the exception of the config file, these tasks can all be found in **quadrotor.py**. The components written in red are implemented for you, and can be found in the animation and validation scripts respectively (mainly for learning purposes). All equations are provided, along with the basic structure of the simulation. For this exercise, you will be using numpy for its matrix operations; a brief overview of numpy is provided at the end of this document.

## **Table of Nomenclature:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Description** | **Frame** | **Shape** |
|  | Linear velocity vector | Body | 3x1 |
|  | Angular velocity vector | Body | 3x1 |
|  | Position vector | Inertial | 3x1 |
|  | Attitude vector (roll, pitch, yaw) | Inertial | 3x1 |
|  | Linear velocity time derivative | Body | 3x1 |
|  | Angular velocity time derivative | Body | 3x1 |
|  | Position time derivate | Inertial | 3x1 |
|  | Attitude time derivative | Inertial | 3x1 |
|  | Thrust force | Body | 3x1 |
|  | Aerodynamic force | Body | 3x1 |
|  | Moment due to thrust force | Body | 3x1 |
|  | Aerodynamic moment | Body | 3x1 |
|  | Mass | N/A | Scalar |
|  | Inertia tensor | N/A | 3x3 |
|  | Gravity vector | Inertial | 3x1 |
|  | Euler rotation matrix | Body to inertial | 3x3 |
|  | Euler rates matrix | Body to inertial | 3x3 |
|  | Propeller thrust coefficient | N/A | Scalar |
|  | Propeller torque coefficient | N/A | Scalar |
|  | Aerodynamic force coefficient (drag coefficient) | N/A | Scalar |
|  | Aerodynamic moment coefficient | N/A | Scalar |
|  | Arm length (assume symmetrical aircraft) | N/A | Scalar |
|  | Rotor angular velocity | N/A | 1x4 |

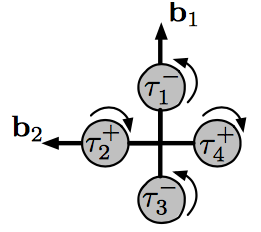
## **Aircraft Model**

We model the aircraft in an East-North-Up axis system, and assume an infinite flat Earth that also uses an East-North-Up convention. A diagram of our system is pictured below:



*Nathan Michael, CMU, 2014*

Where **b1** is our aircraft’s body x-axis, and **e1** is the same for the inertial frame. The convention we use for motor directions is given below:



*Nathan Michael, CMU, 2014*

The 6DOF, non-linear equations of motion are:

|  |  |
| --- | --- |
|  | **(1)** |
|  | **(2)** |
|  | **(3)** |
|  | **(4)** |

Where thrust and aerodynamic forces and moments are modelled as being:

|  |  |
| --- | --- |
|  | **(5)** |
|  | **(6)** |
|  | **(7)** |
|  | **(8)** |

Where the ^ symbol denotes a unit vector, and the symbol denotes the vector cross product. We calculate the thrust and aerodynamic effects in the body frame of the aircraft, but we want to track its position and attitude in the global (inertial) frame. This means we have to rotate linear and angular velocities from the body frame into the inertial frame using equations 2 and 3, where:

|  |  |
| --- | --- |
|  | **(9)** |
|  | **(10)** |

Once we have the velocities in the inertial frame, we step our simulation forward by a small time-step *dt* to get the new position and attitude.

Our program is therefore as follows:

**Inputs ():**

**1 Set ,**

**2 While t < T, repeat:**

**3 Generate control signal (already implemented)**

**4 Calculate thrust forces (Eq. 5)**

**5 Calculate thrust moments (Eq. 6)**

**6 Calculate aerodynamic forces (Eq. 7)**

**7 Calculate aerodynamic moments (Eq. 8)**

**8 Calculate (Eq. 9)**

**9 Rotate G into body frame using**

**10 Calculate body frame linear acceleration () (Eq. 1)**

**11 Calculate body frame angular acceleration () (Eq. 2)**

**12 Set (already implemented)**

**13 Set (already implemented)**

**14 Calculate (Eq. 10)**

**15 Calculate inertial frame velocity () (Eq. 3)**

**16 Calculate inertial frame Euler rates () (Eq. 4)**

**17 Set (already implemented)**

**18 Set (already implemented)**

**19 Set**

**20 Render graphics (already implemented)**

## **Exercises**

We can abstract the above code into a series of scripts and functions to make it more organized. Our first script – **validate\_sim.py** – contains the main while loop, and imports the simulation parameters from the file **config.py**,the Quadrotor class from **quadrotor.py**, and the visualization functions in **animation.py**. From there, it instantiates an aircraft by creating an instance of Quadrotor, and passing it the relevant aircraft parameters. Quadrotor includes all of the functions that step our simulation forward. It returns a new state to the main loop, which then calls the rendering function to update our graphics.

This sounds complicated at first, but it should become clearer after taking a look at the code. Your task is to implement the dictionary in the config file that contains all of our simulation parameters, and then finish the following functions (all of which are found in **quadrotor.py)**:

1. step(control\_signal) – a function that includes lines 4-19 above. This function takes an argument of control signal, and steps the simulation forward one time-step at a time. All of the functions below are defined in the Quadrotor class, and called from within the step() function;
2. thrust\_forces(rpm) – takes an argument of rotor speed, and returns a 3x1 thrust force vector using Eq. 5;
3. thrust\_moments(rpm) – takes an argument of rotor speed, and returns a 3x1 thrust moment vector using Eq. 6;
4. aero\_forces(uvw) – takes an argument of linear velocity (in the body frame) and returns a 3x1 aerodynamic force vector using Eq. 7;
5. aero\_moments(pqr) – takes an argument of angular velocity (in the body frame) and returns a 3x1 aerodynamic moment vector using Eq. 8.
6. R1(zeta) – takes an argument of aircraft attitude (inertial frame) and returns a 3x3 rotation matrix using Eq. 9; and,
7. R2(zeta) – takes an argument of aircraft attitude (inertial frame) and returns a 3x3 Euler rates matrix using Eq. 10.

**Exercise 1:**

Our quadrotor will use the following parameters:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description** |
| mass | 0.65 | Mass of the aircraft |
| prop\_radius | 0.1 | Propeller radius (for plotting only) |
| n\_motors | 4 | Number of motors |
| hov\_p | 0.5 | Hover thrust as percentage of max thrust (assume 50%) |
| l | 0.23 | Arm length L |
| Jxx | 7.5e-3 | Mass moment of inertia about body x-axis |
| Jyy | 7.5e-3 | Mass moment of inertia about body y-axis |
| Jzz | 1.3e-2 | Mass moment of inertia about body z-axis |
| kt | 3.13e-5 | Thrust force coefficient |
| kq | 7.5e-7 | Thrust moment coefficient |
| kd | 9e-3 | Aerodynamic force coefficient |
| km | 9e-4 | Aerodynamic moment coefficient |
| g | 9.81 | Gravitational acceleration |
| dt | 0.001 | Simulation time step |

Open the file called config.py and save these values into a dictionary called **params**. Use the names from the table above to store the values. Once you’ve done this, run simulation.py and ensure that they load correctly.

**Exercise 2:**

We have four functions to calculate thrust forces and moments, and aerodynamic forces and moments. The names of these functions are:

1. def thrust\_forces(self, rpm):
2. def thrust\_moments(self, rpm):
3. def aero\_forces(self, uvw):
4. def aero\_moments(self, pqr):

For each of these functions, implement the necessary equation and return a 3x1 numpy array. Once you’ve done this, run the code to ensure it is working correctly. If you want to transpose a vector or matrix using numpy, you can use **a.T** for **a**. The vector norm can be calculated using **np.linalg.norm(a)**.

**Exercise 3:**

We have two routines for rotating from the body frame to the inertial frame. The first routine converts linear velocities in the body frame to the inertial frame using Eq. 9. The second routine converts angular velocities in the body frame to Euler rates in the inertial frame using Eq. 10.

For each of these, implement the corresponding rotation matrix as outlined above. Each of these routines takes a 3x1 attitude vector containing the roll, pitch, and yaw of the vehicle as the argument, and returns a 3x3 numpy array.

**Exercise 4:**

Now that we have the rotation matrix, we need to calculate our accelerations in the body frame using:

Where denotes the transpose of the matrix . This is done in the function:

def step(self, control\_signal).

Remember the shapes of your arrays. **J** is a 3x3 array, **Q** and **F** will be 3x1 arrays, **ω** and **v** are 3x1 arrays, **G** is a 3x1 array, and *m* is a scalar value. You can use **np.cross(a, b, axis=0)** to get the vector cross product of two 3x1 vectors **a** and **b**. For matrix multiplication of a 3x3 matrix **a** and a 3x1 vector **b**, you can use **a.dot(b)**.

**Exercise 5:**

Once Exercise 4 is completed, the simulation will automatically update the velocities using the calculated acceleration. The final task is to take these velocity values, and rotate them into the inertial frame so that we can update the position and attitude of the vehicle. The equations to do this are:

This should be done in the step function of the script **quadrotor.py**. Once you’ve included this, you should be able to run the simulation and watch the aircraft take off (hopefully with no errors!).

## **Brief Notes on Numpy:**

Import numpy using:

import numpy as np

it is standard practice to use np to refer to numpy since it’s nice and short. Within numpy there are several functions that are useful:

array():

We can initialize a numpy array using:

x = np.array([[0.],[0.],[0.]])

This will create a 3x1 vector, and will let us use numpy’s inbuilt functions:

Transpose: y = x.T

Vector norm: z = np.linalg.norm(x)

Matrix multiplication: k = y.dot(x)

Cross product: a = np.cross(a, b)

This lets you do other tasks such as:

a = np.array([[3.],[2.],[1.]])

b = a.T.dot(a)

Which will calculate the squared magnitude of 3x1 vector **a**.

Numpy arrays can take any shape, and are excellent for linear algebra tasks. The underlying computation is all done in C, so numpy can be extremely fast when used correctly. It comes with many inbuilt functions for scientific computing, with useful examples including np.sum(), np.prod(), np.mean(), np.std(), etc. I’d recommend anyone interested in simulation and computation also get familiar with np.einsum(), since it is a nice, clean method for doing complex matrix operations (most MATLAB functions have an analogue in numpy).

For a more complete reference, go to <http://www.numpy.org>.