University of Auckland

Space Club – Project 1: Basic Avionics Package

# Aim

This project aims to complete a basic avionics program, run in Python 3.6, that can calculate atmospheric and orbital trajectories of a rocket.

# Key Tasks

The avionics program will include:

* A basic atmospheric model
* Engine thrust modelling (momentum and pressure) for an ideally expanded engine
* Rocket mass modelling
* Acceleration modelling
* Velocity modelling
* Position modelling
* Drag
* Staging

# Extra Tasks:

* Advanced atmospheric model (includes location and local weather)
* Engine thrust modelling for a single engine nozzle (over & under expanded flow)
* A nozzle design program
* Re-entry prediction for expendable hardware
* Interplanetary trajectory plotting

## Atmospheric Model:

**0-86 km**: Adapted version from *U.S. Standard Atmosphere 1976* (<https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770009539.pdf> ) written in Python (found at <http://www.pdas.com/atmos.html>). Equations are explained succinctly on <http://www.pdas.com/hydro.pdf>

**86-1000 km:**

Described here <http://www.braeunig.us/space/atmmodel.htm> - model and extension

NASA link: <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770003812.pdf>

Modern models: <http://spaceweather.usu.edu/files/uploads/PDF/COSPAR_INTERNATIONAL_REFERENCE_ATMOSPHERE-CHAPTER-1_3(rev-01-11-08-2012).pdf>

## Engine thrust modelling

## Solving 2nd Order Nonlinear Differential Equation

*# To solve a second-order ODE using scipy.integrate.odeint, you should write it as a system of first-order ODEs:  
#  
# I'll define z = [x', x], then z' = [x'', x'], and that's your system! Of course, you have to plug in your real relations:  
#  
# x'' = -(b\*x'(t) + k\*x(t) + a\*(x(t))^3 + m\*g) / m  
# becomes:  
#  
# z[0]' = -1/m \* (b\*z[0] + k\*z[1] + a\*z[1]\*\*3 + m\*g)  
# z[1]' = z[0]  
# Or, just call it d(z):  
#  
# def d(z, t):  
# return np.array((  
# -1/m \* (b\*z[0] + k\*z[1] + a\*z[1]\*\*3 + m\*g), # this is z[0]'  
# z[0] # this is z[1]'  
# ))  
# Now you can feed it to the odeint as such:  
#  
# \_, x = odeint(d, x0, t).T  
# (The \_ is a blank placeholder for the x' variable we made)  
#  
# In order to minimize b subject to the constraint that the maximum of x is always negative, you can use scipy.optimize.minimize. I'll implement it by actually maximizing the maximum of x, subject to the constraint that it remains negative, because I can't think of how to minimize a parameter without being able to invert the function.  
# #  
# from scipy.optimize import minimize  
# from scipy.integrate import odeint  
# import numpy as np  
#  
# m = 1220  
# k = 35600  
# g = 17.5  
# a = 450000  
# z0 = np.array([-.5, 0])  
# t = 5  
#  
# def d(z, t, m, k, g, a, b):  
# return np.array([-1/m \* (b\*z[0] + k\*z[1] + a\*z[1]\*\*3 + m\*g), z[0]])  
#  
# def func(b, z0, \*args):  
# \_, x = odeint(d, z0, t, args=args+(b,)).T  
# return -x.max() # minimize negative max  
#  
# cons = [{'type': 'ineq', 'fun': lambda b: b - 1000, 'jac': lambda b: 1}, # b > 1000  
# {'type': 'ineq', 'fun': lambda b: 10000 - b, 'jac': lambda b: -1}, # b < 10000  
# {'type': 'ineq', 'fun': lambda b: func(b, z0, m, k, g, a)}] # func(b) > 0 means x < 0  
#  
# b0 = 10000  
# b\_min = minimize(func, b0, args=(z0, m, k, g, a), constraints=cons)*

'''  
# The second order differential equation for the angle `theta` of a  
# pendulum acted on by gravity with friction can be written::  
  
# theta''(t) + b\*theta'(t) + c\*sin(theta(t)) = 0  
  
# where `b` and `c` are positive constants, and a prime (') denotes a  
# derivative. To solve this equation with `odeint`, we must first convert  
# it to a system of first order equations. By defining the angular  
# velocity ``omega(t) = theta'(t)``, we obtain the system::  
  
# theta'(t) = omega(t)  
# omega'(t) = -b\*omega(t) - c\*sin(theta(t))  
  
# Let `y` be the vector [`theta`, `omega`]. We implement this system  
# in python as:  
  
def pend(y, t, b, c):  
 theta, omega = y  
 dydt = [omega, -b\*omega - c\*np.sin(theta)]  
 return dydt  
# ...  
  
# We assume the constants are `b` = 0.25 and `c` = 5.0:  
  
b = 0.25  
c = 5.0  
  
# For initial conditions, we assume the pendulum is nearly vertical  
# with `theta(0)` = `pi` - 0.1, and it initially at rest, so  
# `omega(0)` = 0. Then the vector of initial conditions is  
  
y0 = [np.pi - 0.1, 0.0]  
  
# We generate a solution 101 evenly spaced samples in the interval  
# 0 <= `t` <= 10. So our array of times is:  
  
t = np.linspace(0, 10, 101)  
  
# Call `odeint` to generate the solution. To pass the parameters  
# `b` and `c` to `pend`, we give them to `odeint` using the `args`  
# argument.  
  
from scipy.integrate import odeint  
sol = odeint(pend, y0, t, args=(b, c))  
  
# The solution is an array with shape (101, 2). The first column  
# is `theta(t)`, and the second is `omega(t)`. The following code  
# plots both components.  
  
import matplotlib.pyplot as plt  
plt.plot(t, sol[:, 0], 'b', label='theta(t)')  
plt.plot(t, sol[:, 1], 'g', label='omega(t)')  
plt.legend(loc='best')  
plt.xlabel('t')  
plt.grid()  
plt.show()  
'''

## Validation

