Rocket Lab Technical Test

Tim Holthuijsen

imports

First of all, we are going to need some Python libraries for this analysis. We import the necessary ones below

In [1]:
```
import scipy
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math as m
from scipy.optimize import curve_fit
from numpy import arange
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import explained_variance_score
```

The goal of this exercise is to accurately predict the ejection velocity of for the range of payload masses from 3kg to 15kg, as well as obtain the physical spring parameters k (spring constant) and b (damping coefficient) of a 2-spring nanosatelite dispenser (CSD). For this, we use the mass-spring-damper model, defined as follows:

$$m\ddot{x} = -b\dot{x} - kx$$

where m is the spacecraft mass in kg, b is the damping coefficient in Ns/m, k is the spring constant in N/m and x is the displacement of the spring in m

Additionally, we have been given a graph representing ejection velocity as a function of payload mass, as shown below. In this graph, we are specifically interested in the 3U dispenser containing 2 springs, or the green line in the figure

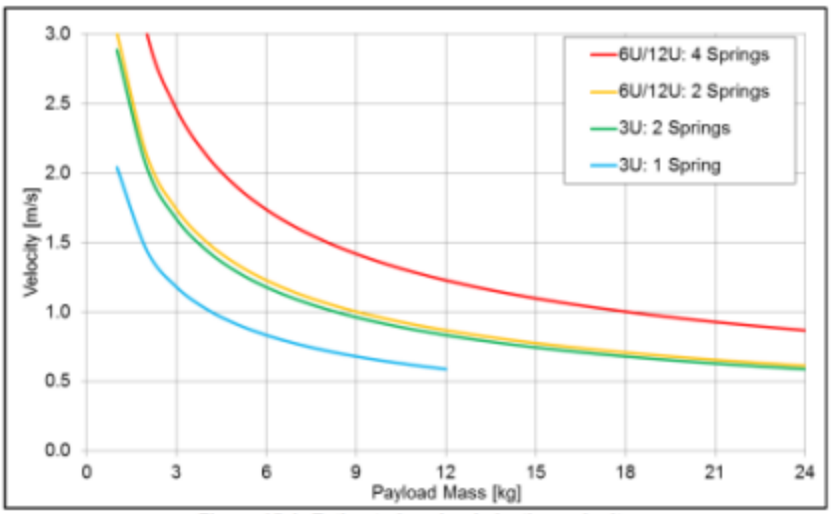


Figure 15-1: Estimated payload ejection velocity

Data extraction

In order to make this visual data useable, we visually extract datapoints from this graph using WebPlotDigitizer, as shown below:

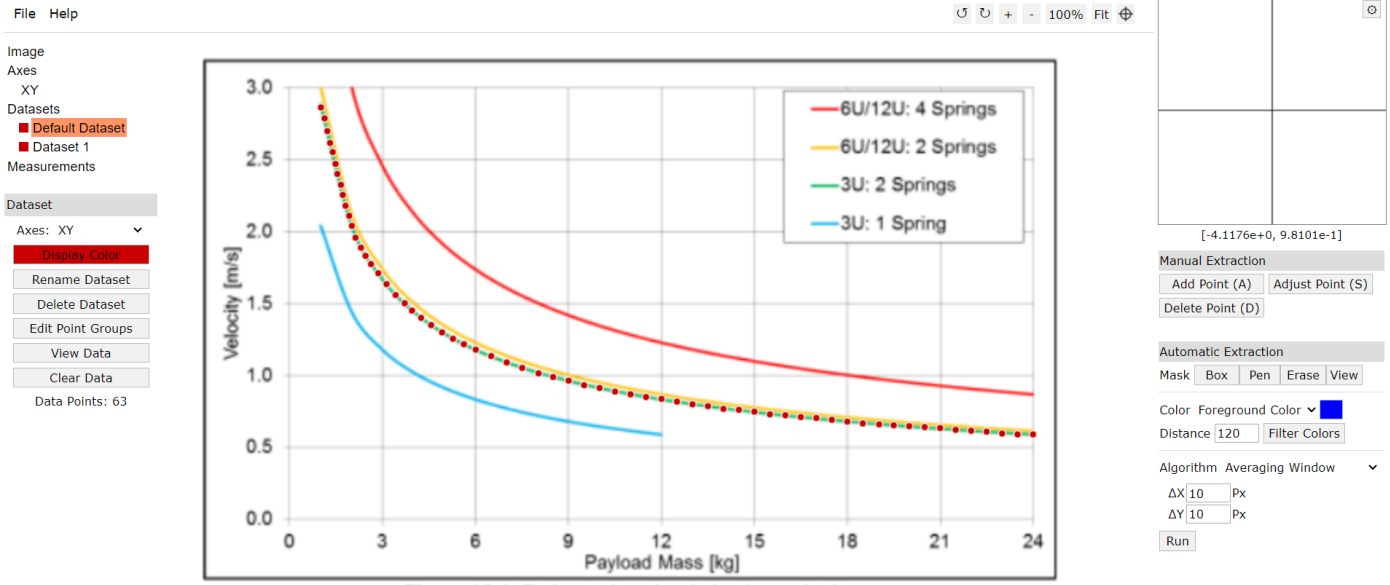


Figure 15-1: Estimated payload ejection velocity

Loading the data

Now that the data has been extracted (and saved as a csv), we can load it into Python as a pandas dataframe

In [2]:
```
#Load the data
data = pd.read_csv('TwoSpringsData.csv',names = ['mass', 'velocity'])
```

In [3]:
```
#have a quick look at the data
data
```

Out[3]:

	mass	velocity
0	1.000000	2.860759
1	1.117647	2.784810
2	1.205882	2.696203
3	1.294118	2.613924
4	1.382353	2.550633
...	...	...
58	22.000000	0.613924
59	22.500000	0.607595
60	23.000000	0.594937
61	23.500000	0.588608
62	24.000000	0.588608

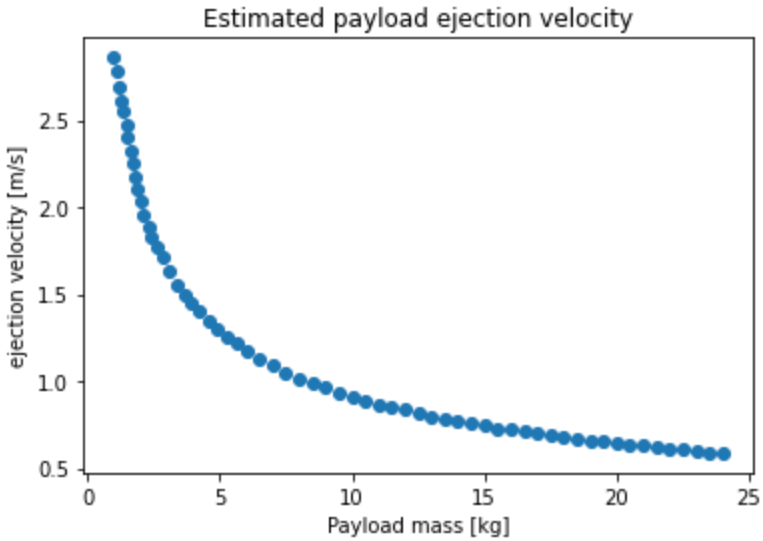
63 rows × 2 columns

Visual confirmation

To check that everything went right, we quickly plot our extracted data to confirm that it still has the same shape as the original CSD 2U Velocity/Mass line (it does)

In [4]:
```
#we store the individual data columns into some easily-accesible variables
mass_data = data['mass']
v_data = data['velocity']
```

In [5]:
```
#Start with plotting the original data
plt.scatter(mass_data, v_data)
plt.title("Estimated payload ejection velocity")
plt.xlabel("Payload mass [kg]")
plt.ylabel("ejection velocity [m/s]")
plt.show()
```



Parameter estimation

Now that we have our data in the right shape, we can start optimizing a function to fit our curve

Let's fit an accurate function to the velocity-mass graph

Using mathematical analysis (included further below) of the mass-springer-damper model, the exact format of the velocity-mass function is found to be of the shape:

$$v = \frac{a}{\sqrt{bm}}$$

With v being velocity in meters/second, m being payload mass in kg, and a and b being constants. Using this function template, we can fit function parameters for a and b to create a curve to exactly fit our extracted velocity-mass curve. This is done below:

In [6]:
```
# Define the shape of the function
def model(x, a, b):
    result = a / (b*x)**0.5 #This is the function listed above
    return result
```

In [7]:
```
#Optimize the parameters to fit the curve
parameters, covariance_matrix = curve_fit(model, mass_data, v_data)
```

In [8]:
```
parameters
```

Out[8]:
```
array([ 36.38281008, 156.09995452])
```

In [9]:
```
#Have a quick look at the parameters and the covariance matrix
print('  a is ', parameters[0], '\n ', 'b is ', parameters[1], '\n')
print(' covariance matrix is: ', '\n',covariance_matrix)
```

```
 a is  36.38281007546312
 b is  156.09995451828834

covariance matrix is:
[[5.25257081e+13 4.50721685e+14]
[4.50721685e+14 3.86763063e+15]]
```

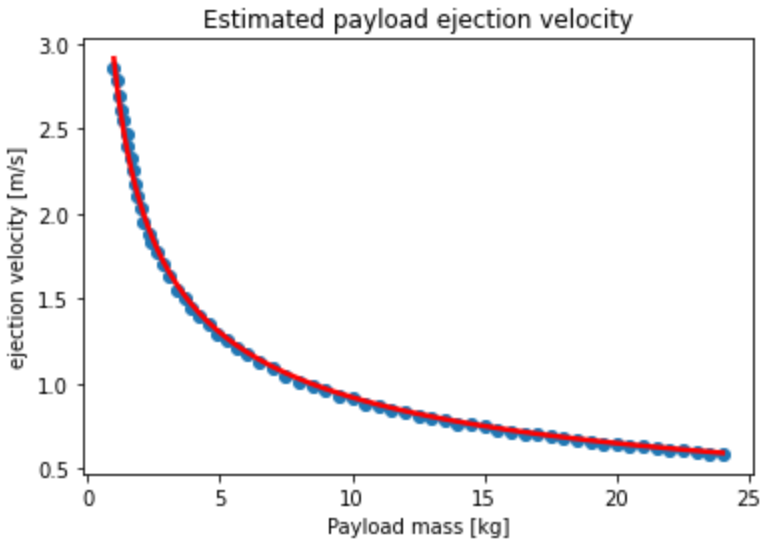
In [10]:
```
# display the final function with optimized parameters
a, b= parameters #store the parameters
print('v = %.5f / sqrt(%.5f m) ' % (a, b))
```

```
v = 36.38281 / sqrt(156.09995 m)
```

Now let's plot our prediction curve next to the original one

In [11]:
```
#Let's store the original data in a slightly more plot-like way
x = mass_data
y = v_data
```

In [12]:
```
#Start by plotting the original data
plt.scatter(x, y)
# define a sequence of inputs between the smallest and largest known inputs
#x_line = arange(min(x), max(x), 1)
# calculate the output of our fitted function for the range
v_predicted = model(x, a, b)
# create a line plot for the fitted function
plt.plot(x, v_predicted, color='red', linewidth = 2.5)

plt.title("Estimated payload ejection velocity")
plt.xlabel("Payload mass [kg]")
plt.ylabel("ejection velocity [m/s]")

plt.show()
```



A perfect fit!

As you can tell from the graph above, the red fitted curve fits the blue curve containing the original data almost exactly. This shows that we have acquired an accurate approximation of the velocity-mass function using our fitted curve, and, knowing either velocity or mass, we can now always calculate the other using the function:

$$v = \frac{36.38281}{\sqrt{156.09995m}}$$

With this, we have largely completed the task, stating as a goal to "accurately predict the ejection velocity of the dispenser for the range of spacecraft (payload) masses from 3kg to 15kg."

However, since k and b are asked, we will continue to analytically derive these.

First, however, we test the accuracy of our model a little more rigorously than a simple visual test. We compute some model evaluation parameters: Mean squared error, R squared, and the explained variance

Let's calculate our eveluation parmeters:

In [13]:
```
# given values are equal to v_data
real = v_data
# calculated values are equal to v_predicted
pred = v_predicted

#Calculation of Mean Squared Error (MSE)
mse = mean_squared_error(real, pred)
#R squared
r_squared = r2_score(real, pred)
#and finally, calculate explained variance
explained_variance = explained_variance_score(real,pred)
```

In [14]:
```
print('mse is: ', mse, ' :this should be as close to 0 as possible')
print('R squared is: ', r_squared,' :this should be as close to 1 as possible')
print('explained variance is: ', explained_variance,' :this should be as close to 1 as pos
```

```
mse is:  0.0005631950448297956  :this should be as close to 0 as possible
R squared is:  0.9987668111134123  :this should be as close to 1 as possible
explained variance is:  0.9988197605719936  :this should be as close to 1 as possible
```

Incredible fit

As you can tell from the values above, our model is an incredibly accurate fit with the real data.

However....

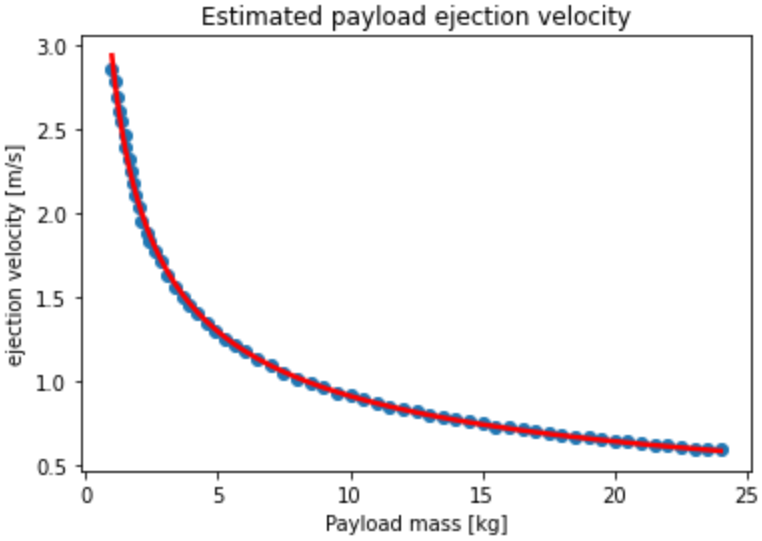
I think we might be able to do a bit better still. I have something specific in mind:

Rather than taking the square root of bm in our predicting function, let's optimize the power of the denominator as well and see if it improves things even further

In [15]:
```
# Define the shape of the new function
def model2(x, a, b, c):
    result = a / (b*x)**c # <---- This is what's different! rather than using the power o
    return result

#Optimize the new parameters to fit the curve
parameters2, covariance_matrix = curve_fit(model2, mass_data, v_data)

# display the final function with optimized parameters
```

```
a, b, c= parameters2 #store the parameters
print('formula is: v = %.10f / (%.10f m)**%.10f ' % (a, b,c))

#Start by plotting the original data
plt.scatter(x, y)

# calculate the output of our fitted function for the range
v_predicted2 = model2(x, a, b,c)
# create a line plot for the fitted function
plt.plot(x, v_predicted2, color='red', linewidth = 2.5)

plt.title("Estimated payload ejection velocity")
plt.xlabel("Payload mass [kg]")
plt.ylabel("ejection velocity [m/s]")

plt.show()
```

```
formula is: v = 0.7615416313 / (0.0707200048 m)**0.5099012268
```



As you can tell, the fit is still great. Now, let's see if it is even better than our previous model:

In [16]:
```
# calculated values are equal to v_predicted2 for the new model
pred = v_predicted2

#Calculation of Mean Squared Error (MSE)
mse2 = mean_squared_error(real, pred)
#R squared
r_squared2 = r2_score(real, pred)
#and finally, calculate explained variance
explained_variance2 = explained_variance_score(real,pred)
```

In [17]:
```
print('mse is: ', mse2, ' previous mse was: ', mse)
print('R squared is: ', r_squared2,' previous R squared was: ', r_squared)
print('explained variance is: ', explained_variance2,' previous explained variance was: ',
```

```
mse is:  0.00041797667569404436  previous mse was:  0.0005631950448297956
R squared is:  0.9990847856421134  previous R squared was:  0.9987668111134123
explained variance is:  0.9990854272900126  previous explained variance was:  0.9988197605
719936
```

Newest model is an even better fit

The evaluation parameters of this latest iteration of the model show that this version is even more accurate than the previous iteration. Our final formula comes to:

$$v = \frac{0.76154}{0.07072m^{0.50990}}$$

Analysis

We have the following equation:

$$m\ddot{x} = -b\dot{x} - kx$$

Additionally, the duration to fully extend for the spring is given by:

$$\text{Duration} = -\frac{m \log_e\left(\frac{2\sqrt{-km}}{bi + \sqrt{-b^2 + 4km}}\right) 2i}{\sqrt{4km - b^2}}$$

One could use this duration formula, in addition to the given displacement length of the spring (0.15312m), in order to calculate:

$$velocity = \frac{displacement}{duration}$$

I attempted this approach at length. However, I realized that using this formula to calculate velocity will only yield a *average* velocity value, rather than the velocity at ejection, and is therefore not of value (except for testing estimations). Therefore, we continue manipulating the spring-damper-model:

$$m\ddot{x} = -b\dot{x} - kx$$

$$\text{Divide by m to isolate } \ddot{x}$$

$$\ddot{x} = \frac{-b\dot{x}}{m} - \frac{kx}{m}$$

Since we are considering the moment of ejection, at this point we assume acceleration = $\ddot{x}$ = 0. Therefore, we can write:

$$\frac{b\dot{x}}{m} - \frac{kx}{m} = 0$$

We also know:

$$x(t) = A_o * e^{Dt} cos(wt)$$

With:

$$A_o = \text{initial displacement} = 0.15312m$$

$$\dot{x} = v$$

$$w = \sqrt{\frac{k}{m} - \frac{b}{2m}} \text{<-- Assuming damping}$$

And

$$w_n = \text{undamped natural frequency} = \sqrt{\frac{k}{m}}$$

We also know:

$$\frac{b}{m} = 2\zeta w_n$$

We do not know ζ. However, considering our conditions, I think it is fair to assume that our model is underdamped, and that ζ is thus:

$$0 > \zeta > 1$$

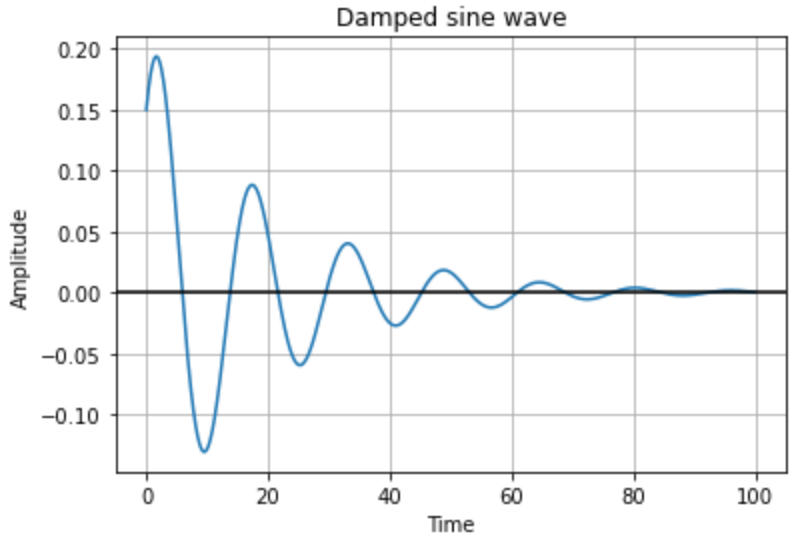
The exact value of ζ would be easily retrievable by observing the oscillating motion of the spring in a (preferably vacuum) laboratory. However, we do not have access to this at the moment. Therefore, we define a sinusoidal function, tweak ζ, and observe it until it looks like a realistic motion for a 15cm length spring:

We can write a **general equation** for an exponentially damped sinusoid as

$$y(t) = Ae^{-\lambda t} \cdot \left(cos(\omega t + \Phi) + sin(\omega t + \Phi)\right)$$

In [18]:

```
# Get x values of the sine wave
time        = np.arange(0, 100, 0.1);
# Amplitude of the sine wave is sine of a variable like time
amplitude   = 0.15 * m.e ** (-0.05*time) * (np.cos(0.4*time) + np.sin(0.4*time)) # Here we
# Plot a sine wave using time and amplitude obtained for the sine wave
plt.plot(time, amplitude)
# Give a title for the sine wave plot
plt.title('Damped sine wave with zeta = 0.05')
# Give x axis label for the sine wave plot
plt.xlabel('Time') # <--- Specific values don't matter because we are only interested in t
# Give y axis label for the sine wave plot
plt.ylabel('Amplitude')
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
# Display the sine wave
plt.show()
```



$\zeta = 0.05$ seems to yield a realistic motion for a damped oscilating spring. This could be further improved by observing the spring in question and fitting the motion data to a function as done previously in this notebook. But for now, $\zeta = 0.05$ will have to do

now we can write:

$$2\sqrt{\frac{k}{m}} * \zeta * \dot{x} = \frac{-k}{m}x = -w_n^2 x$$

We extract an arbitrary datapoint from our dataset so that we can plug in m and v:

In [19]:
```
data[38:39]
```

Out[19]:

	mass	velocity
38	12.0	0.835443

Thus:

$$m = 12$$

$$v = \dot{x} = 0.83$$

$$x = \text{displacement} = 0.15321$$

$$\zeta = 0.05$$

Let's plug this in in the formula

$$2\sqrt{\frac{k}{m}} * \zeta * \dot{x} = \frac{-k}{m}x$$

Which becomes:

$$2\sqrt{\frac{k}{12}} * 0.05 * 0.83 = \frac{-k}{12} * 0.15321$$

$$\text{solve for k:}$$

$$k = 3.522$$

Now we try to get b:

$$\text{we know:}$$

$$\frac{b}{m} = 2\zeta w_n$$

$$\frac{b}{m} = 2 * 0.05 * \sqrt{\frac{k}{m}}$$

$$\frac{b}{12} = 2 * 0.05 * \sqrt{\frac{3.522}{12}}$$

$$b = 12 * 2 * 0.05 * \sqrt{\frac{3.522}{12}}$$

$$b = 0.650$$

Finally, we show how we derived the general shape of the mass-velocity equation:

$$\frac{b}{m} = 2\zeta w_n$$

$$w_n = \sqrt{\frac{k}{m}}$$

$$\dot{x} = v$$

substitute all of that into the equation:

$$\frac{b}{m}\dot{x} = \frac{-k}{m}x$$

yields:

$$2\zeta\sqrt{\frac{k}{m}}v = -\frac{k}{m}x$$

A function expressing v in terms of m, from which we derived the general shape required to fit the curve t

Endnote

Wow, what an assignment. I don't think all of the analytical values above can be correct. They are dependent on too many seperate steps and assumptions to be flawless. However, they have yielded a real, tangible, and valueable result, and hopefully a thorough understanding of the processes involved in acquiring these. The Formula for predicting v in terms of m is an incredible fit to the data, and would have serious real-world applications; given a certain payload mass, it would be easy to accurately calculate the ejection velocity of the payload. Specifically for the 2-spring CSD at the moment, but this workflow could easily be scaled up to fit any and all nanosatellite ejection vehicles with high reliability.

In [ ]:

Little bonus

Before this assignment was about second order differential equations, it was about something much more elegant

$$\Delta V$$

Reading about this in the assignment made me excited to work with it, so I was a little sad when it wasn't really necessary in the task. Therefore, I have decided to pose the following problem:

Given an Electron LV equipped with a 2-spring CSD in geostationary orbit, how much ΔV would it need to provide to the nanosatellite payload in order to de-orbit the payload with only that single ejection?

We assume a perfectly circular geosynchrounous orbit above Earth's equator with:

$$Aphelion = Perihelion = 35,786\text{km above sea level}$$

Let's say we want to cancel out all of the orbital velocity of the satellite, so that it will stand still and fall straight to earth. For this, we first calculate the orbital speed of a geostationary orbit:

$$\text{orbital velocity} = \sqrt{\frac{G * M_E}{\text{radius of orbit}}}$$

$$G = \text{Gravitational constant} = 6.67430 * 10^{11}$$

$$M_E = \text{mass of earth} = 5.972 * 10^{24} kg$$

$$u = G * M_E$$

We could also have reduced our orbit periapsis to around 100km above sea level to the Karman line and use aerobraking to de-orbit, but this would require multiple orbits to have the orbit decay completely, and we want to de-orbit right away. Therefore, we keep our approach of cancelling all orbital velocity with a single ejection

In [20]:
```
#Define a function to calculate the velocity with a given perigee and apogee
u = 3.98589 * 10 **14
def velocity(rp, ra):
    vel = m.sqrt(2*((u/rp)-(u/(rp + ra))))
    return vel
```

In [21]:
```
#geostationary height:
rp = 35786000
ra = rp # <-- circular orbit
speed = velocity(rp, ra)
```

In [22]:
```
print('the velocity we have to cancel is: ', speed, 'm/s')
```

```
the velocity we have to cancel is:  3337.3832027058957 m/s
```

We need to cancel 3337.38 m/s of orbital velocity in order to plunge straight to earth. In other words, we have to burn (or eject) with a ΔV of 3337 m/s in the retrograde direction

Assuming the ejector is facing retrograde, our CSD dispenser should be able to do this. All we need is for the payload mass to be very, very small. We use the model that we previously defined to get this required velocity:

$$v = \frac{0.76154}{0.07072m^{0.50990}}$$

$$3337.38 = \frac{0.76154}{0.07072m^{0.50990}}$$

$$\text{solve for m}$$

$$m = 1.4432 * 10^{-7} kg$$

$$= 1.4432 * 10^{-4} g$$

In other words, if the 2-spring CSD is used on a payload of $1.4432 * 10^{-4}$ grams, and manages to transfer all of its energy, then the ejection force is enough to de-orbit the payload straight away.

This could be a very efficient de-orbiting mechanism, as it does not require any fuel, and only a spring which can be re-compressed and re-used using electric energy

Of course, in reality, this scenario is not entirely realistic (yet).

Firstly, because our v-m model is not representative at such incredibly small m. v does not approach infinity as m approaches 0, as the spring would not reach such incredible speeds even when completely unloaded (m=0). Secondly, there are some design hitches related to the fact that with the force from our 2-spring CSD, we could only de-orbit a payload with the mass approximately that of a grain of sand. Using higher masses, we will also have to consider the ΔV effects on the dispensing spacecraft.

However, the concept itself is not impossible. Using sufficient spring parametres and payload mass according to orbit height, a payload could potentially be de-orbitted using CSD ejection alone.

Who knows. Maybe, one day in the future, instead of bringing retro-thrusters, we just bring a sufficiently strong spring in order to de-orbit. Wouldn't that be a day

Rocket Lab Technical Assessment

Written by Tim Holthuijsen

timholthuijsen@hotmail.com