Deliverable 2 – Dataset, Data Analysis and Mathematical Modeling

In this project, I am determining the value of t which is the time required for a rocket to reach the given upward velocity.

This can be computed by the following formula:

$$v = u \ln \frac{m_0}{m_0 - qt} - gt$$

The parameters in this case have the following values:-

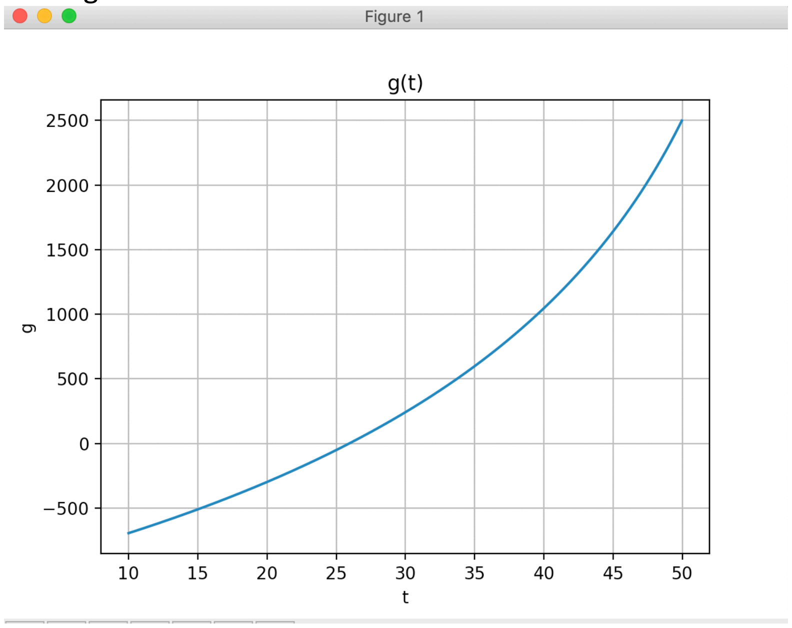
where v = upward velocity, u = the velocity at which fuel is expelled relative to the rocket, m_0 = the initial mass of the rocket at time $t = 0$, q = the fuel consumption rate, and g = the downward acceleration of gravity (assumed constant = 9.81 m/s^2). If $u = 2200$ m/s, $m_0 = 160{,}000$ kg, and $q = 2680$ kg/s,

After substituting these values, the final equation is as follows:-

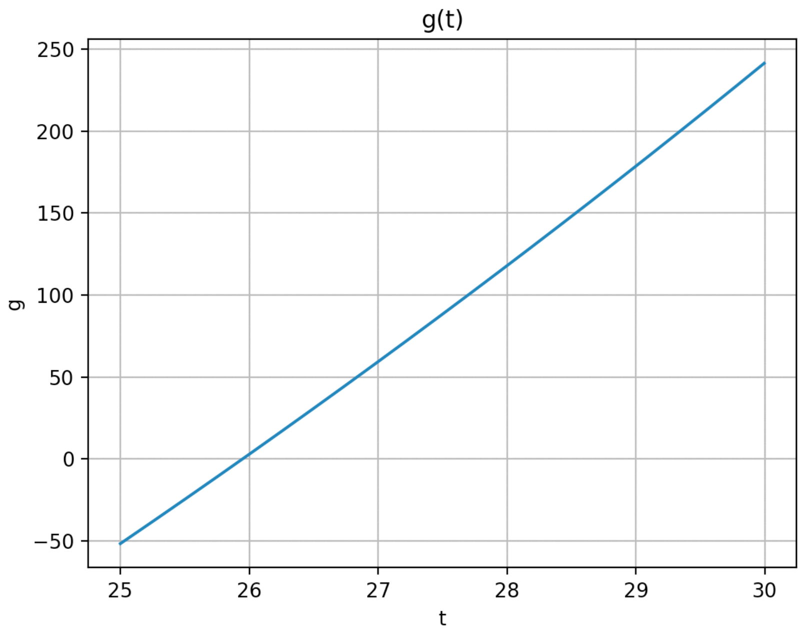
g(t) = 2200 * ln (160000/(160000 – 2680 t)) – 9.81 t - 1000

Graphical Method

My first approach to solve this problem is to estimate the root by plotting the function using the initial guesses of 10 & 50.



Visual inspection of the graph reveals the root to be between 25 and 30. To narrow it down further, I plotted another graph with the guesses as 25 and 30. This led me to the following graph:-



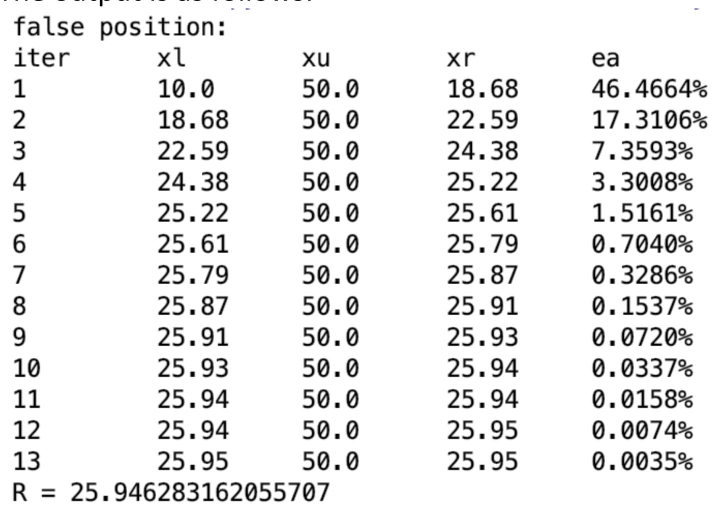
This graph clearly reveals the root between 25 and 26.

Next up, I solved the above function using the bracketing methods.
For each of the following methods described, the error tolerance was calculated using 4 significant digits ie $e_s = 0.005$.

False Position

I will describe the results obtained using False position below.

The false position method obtained the root of the equation in 13 iterations.
The output is as follows:-

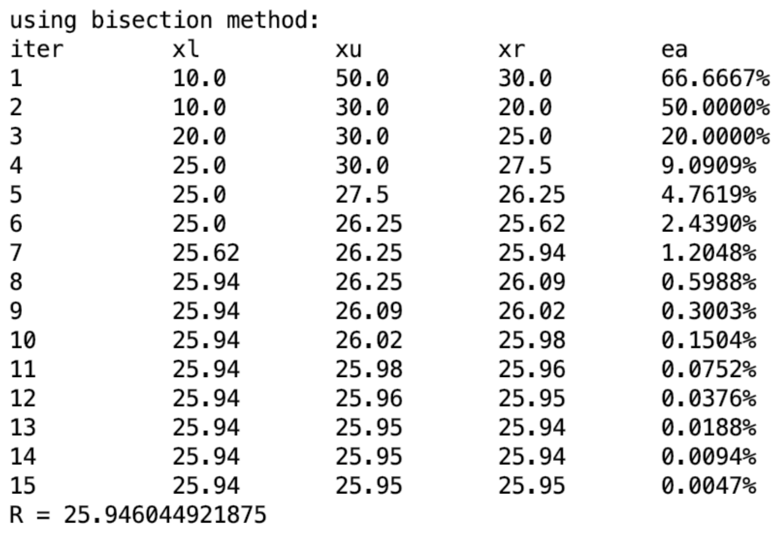


```
false position:
iter     xl       xu       xr       ea
1        10.0     50.0     18.68    46.4664%
2        18.68    50.0     22.59    17.3106%
3        22.59    50.0     24.38    7.3593%
4        24.38    50.0     25.22    3.3008%
5        25.22    50.0     25.61    1.5161%
6        25.61    50.0     25.79    0.7040%
7        25.79    50.0     25.87    0.3286%
8        25.87    50.0     25.91    0.1537%
9        25.91    50.0     25.93    0.0720%
10       25.93    50.0     25.94    0.0337%
11       25.94    50.0     25.94    0.0158%
12       25.94    50.0     25.95    0.0074%
13       25.95    50.0     25.95    0.0035%
R = 25.946283162055707
```

The false position method yields a root that is equal to 25.94628 with $e_s = 0.005$.
The false position is a bracketing method that uses graphical insights. It takes into consideration the closeness of $f(x_l)$ and $f(x_u)$ to zero. This gives an improved estimate of the root than blindly assuming equal halves in the interval between x_l and x_u. Because of the efficient scheme of root location in the false position method, the error approximation reduces quickly to below 1%. It takes only 6 iterations for the error approximation to go below 1%. Since the root estimate is continuously changing in every iteration, the root is known with great accuracy than the prescribed tolerance. However, it is possible for the false position to function poorly in case where the functions have significant curvature. This happens because the bracketing points that are used tend to stay fixed. This leads to poor convergence in determining the root value.

Bisection Method

Next, I solved the function using the Bisection method.
The bisection method obtained the root of the equation in 15 iterations which is longer than the false position method for the same error tolerance.



```
using bisection method:
iter      xl        xu        xr        ea
1         10.0      50.0      30.0      66.6667%
2         10.0      30.0      20.0      50.0000%
3         20.0      30.0      25.0      20.0000%
4         25.0      30.0      27.5      9.0909%
5         25.0      27.5      26.25     4.7619%
6         25.0      26.25     25.62     2.4390%
7         25.62     26.25     25.94     1.2048%
8         25.94     26.25     26.09     0.5988%
9         25.94     26.09     26.02     0.3003%
10        25.94     26.02     25.98     0.1504%
11        25.94     25.98     25.96     0.0752%
12        25.94     25.96     25.95     0.0376%
13        25.94     25.95     25.94     0.0188%
14        25.94     25.95     25.94     0.0094%
15        25.94     25.95     25.95     0.0047%
R = 25.946044921875
```

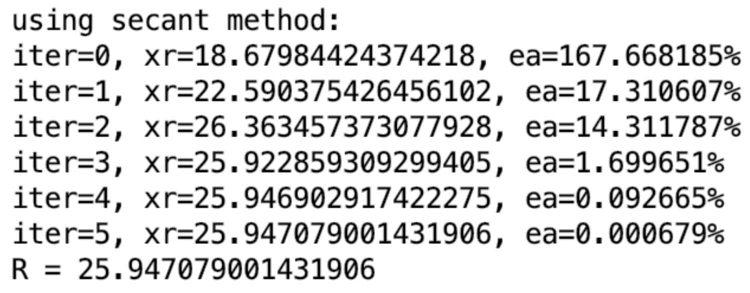
The root of the function by Bisection method turns out to be R = 25.94604. From the results, it can be concluded that the bisection method performs slightly worse than the false position method in this case. The bisection method obtains this root after 15 iterations with a percent relative error of 4.7×10^{-5}.
The bisection method obtains the root by the method of interval halving. If a function changes sign over an interval, the function value at the midpoint is evaluated. The location of the root is determined at the midpoint of the subinterval where the sign changes. This process is repeated multiple times to get better estimates of the root within the prescribed error tolerance.
The percent error approximation falls below 1% in the 8th iteration. This is slower than what we observed in the false position for the same error tolerance. In the bisection method the interval between the lower and upper estimate keeps decreasing during the course of a computation. Among the two bracketing methods that have been tested for this problem, I conclude that the false position method is slightly superior to the bisection method.

Open Methods

They do not employ the use of lower and upper estimates to figure out the root. They require a couple of starting values that may or may not bracket the root. However the open methods converge more quickly as compared to the bracketing methods. I solved this problem using the Secant, Modified Secant and Newton-Raphson methods. I have posted my observations below:-

Secant method

Secant method was the first one that was tested. I could obtain the root of the equation in 6 iterations.



```
using secant method:
iter=0, xr=18.67984424374218, ea=167.668185%
iter=1, xr=22.590375426456102, ea=17.310607%
iter=2, xr=26.363457373077928, ea=14.311787%
iter=3, xr=25.922859309299405, ea=1.699651%
iter=4, xr=25.946902917422275, ea=0.092665%
iter=5, xr=25.947079001431906, ea=0.000679%
R = 25.947079001431906
```

This was a markedly improved performance as compared to the bracketing methods of false position and bisection.

The error approximation goes below 1% in the 5th iteration. This is quicker than both the bracketing methods that we tried above. The secant method calculates the derivative by using a backward finite divided difference. I kept the lower and upper estimate for this function to be at the 10 and 50 to keep it consistent with the other methods and be able to compare between these methods. But this is not necessary for open methods. The secant method is very similar to the false position method. Both use two initial estimates to compute an approximation of the slope of the function to project to the x-axis for a new estimate of the root. The false-position method estimated the latest root to a value that yielded a function value with the same sign as the function value of the root in that iteration. However, the secant method replaces x_{i+1} with x_i and x_i replaces x_{i-1}. This could lead to divergence in this method. However, when the method converges, it does so much quickly than the false-position method.

The secant method converges twice as quickly as compared to the false-position method. The inferiority of the false-position method is due to one end staying fixed to maintain the bracketing of the root. This does avoid divergence, but it does not solve the problem of converging quickly to the root.

Modified Secant method

The modified secant method proves to be the most efficient so far in finding the upward velocity of the rocket. It returns the root for the equation in 5 iterations.

```
modified secant method:
iter=0, xr=30.13955218004831, ea=66.821007%
iter=1, xr=26.283023856746606, ea=14.673077%
iter=2, xr=25.950594251420487, ea=1.281010%
iter=3, xr=25.947095059866044, ea=0.013486%
iter=4, xr=25.94707900014371, ea=0.000062%
R = 25.94707900014371
```

The percentage error approximation falls below 1% in the 4th iteration. The modified secant method converges to the real root after 5 iterations which is almost thrice as quick as the false-position method.

The modified secant method employs the use of a perturbation factor instead of using two initial estimates as done in the secant method. This proves to be more effective since the value converges to the real root within 5 iterations. It is a significant improvement over the secant method which takes longer to converge and also requires two initial estimates to calculate the real root.

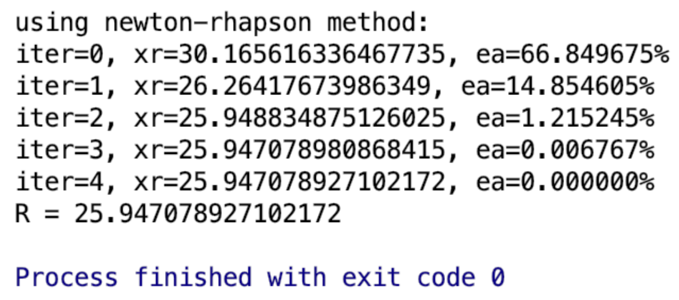
The choice of the perturbation factor is an important decision for the Modified Secant method. If the value is too small, the method will have round-off errors for subtractive cancelation in the denominator.

$$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)}$$

If the value is too big, the method can be divergent and inefficient. However, when a reasonably correct value is chosen, it provides a nice alternative for evaluating the function and does not need two initial estimates.

Newton-Raphson method

This method turns out to be as efficient as the Modified secant method. This method provides the root of the equation in 6 iterations.



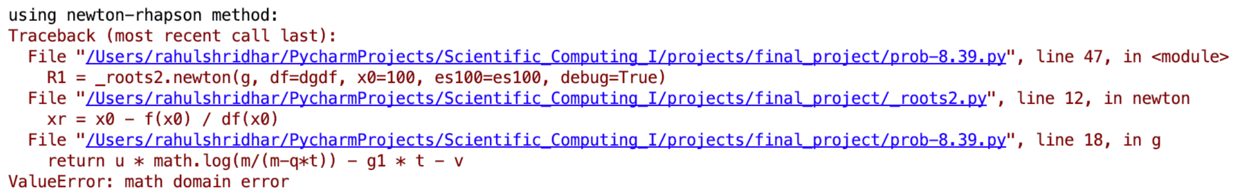
```
using newton-rhapson method:
iter=0, xr=30.165616336467735, ea=66.849675%
iter=1, xr=26.26417673986349, ea=14.854605%
iter=2, xr=25.948834875126025, ea=1.215245%
iter=3, xr=25.947078980868415, ea=0.006767%
iter=4, xr=25.947078927102172, ea=0.000000%
R = 25.947078927102172

Process finished with exit code 0
```

The root of the function is equal to R = 25.94707. The percentage error approximation goes below 1% in the 4th iteration. This performance is similar to the modified secant method and this is slightly better than the observation in the secant method.

The Newton-Raphson method is one of the most widely used root-location formula. It is observed that this method converges rapidly on the true root. The approximate percent relative error decreases very quickly as compared to the secant method. This is the most efficient method to find real roots among all the methods. If the initial guess at the root is xi, a tangent can be extended from the point [xi, f(xi)]. The point where this tangent crosses the x axis represents an improved estimate of the root. However, this method performs poorly in the case of multiple roots. The method can also converge really slowly due to the nature of the function. The convergence also depends on the accuracy of the initial root.

This method works on the fact that the initial guess is picked correctly. If the initial guess is set to the higher value i.e let's say 100, the routine diverges. This is the expected output during this computation.



```
using newton-rhapson method:
Traceback (most recent call last):
  File "/Users/rahulshridhar/PycharmProjects/Scientific_Computing_I/projects/final_project/prob-8.39.py", line 47, in <module>
    R1 = _roots2.newton(g, df=dgdf, x0=100, es100=es100, debug=True)
  File "/Users/rahulshridhar/PycharmProjects/Scientific_Computing_I/projects/final_project/_roots2.py", line 12, in newton
    xr = x0 - f(x0) / df(x0)
  File "/Users/rahulshridhar/PycharmProjects/Scientific_Computing_I/projects/final_project/prob-8.39.py", line 18, in g
    return u * math.log(m/(m-q*t)) - g1 * t - v
ValueError: math domain error
```

This occurs because the function's slope at the initial guess causes the iteration to jump to a negative value. So we can conclude the Newton-Raphson method requires good initial guesses to function efficiently.

Verifying my answer below:-

g(t) = 2200 * ln (160000 / (160000 – 2680 * 25.947)) – 9.81 * 25.947 = 999.9956 m/sec ~ 1000 m/sec.

Conclusion

The Newton-Raphson and the Modified secant method prove to be the best of all the methods used to calculate the friction factor. They converge to the correct value in 5 iterations which is superior to all the other methods. Both methods employ the use of a single guess but use a perturbation factor in the case of the modified secant method. Whereas the Newton-Raphson uses the derivative of the function to converge to the real roots.