Deliverable 2 – Dataset, Data Analysis and Mathematical Modeling

Determining the fluid flow in pipes and tubes is an important task in many areas of engineering and science. In fluid flow, friction loss takes place due to the viscosity of the fluid near the surface of the pipe. This loss in turn causes loss of pressure in pipe and ducts. In engineering, the Moody chart is used to represent the relation between the friction factor, Reynolds number and the surface roughness for a flow in a pipe. It can be used to predict pressure drop or flow rate in a pipe. This relation is called the Colebrook equation which is given as follows:-

$$\frac{1}{\sqrt{f_D}} = -2.0 \log_{10}\left(\frac{\epsilon/D}{3.7} + \frac{2.51}{\mathrm{Re}\sqrt{f_D}}\right), \text{for turbulent flow.}$$

where ε = the roughness (m), D = diameter (m), and Re = the *Reynolds number*

The Reynolds number is calculated as follows:-

$$\mathrm{Re} = \frac{\rho V D}{\mu}$$

where ρ = the fluid's density (kg/m^3), V = its velocity (m/s), and μ = dynamic viscosity (N $\cdot$ s/m^2).

In this project, I am determining the value of f i.e friction factor for air flow through a smooth, thin tube. The parameters for this case are as follows:-

$$\rho = 1.23 \text{ kg/m}^3$$

$$\mu = 1.79 \times 10^{-5} \text{ N} \cdot \text{s/m}^2$$

$$D = 0.005 \text{ m}$$

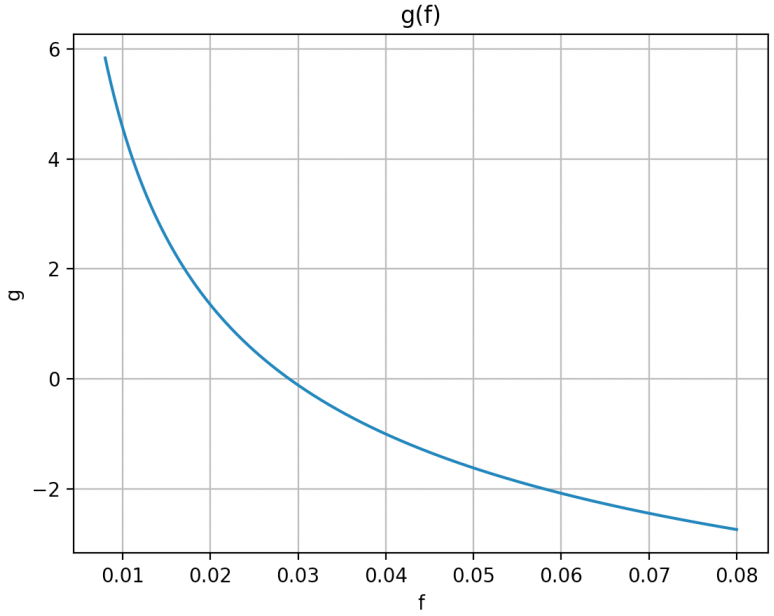
$$V = 40 \text{ m/s, and } \varepsilon = 0.0015 \text{ mm}$$

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

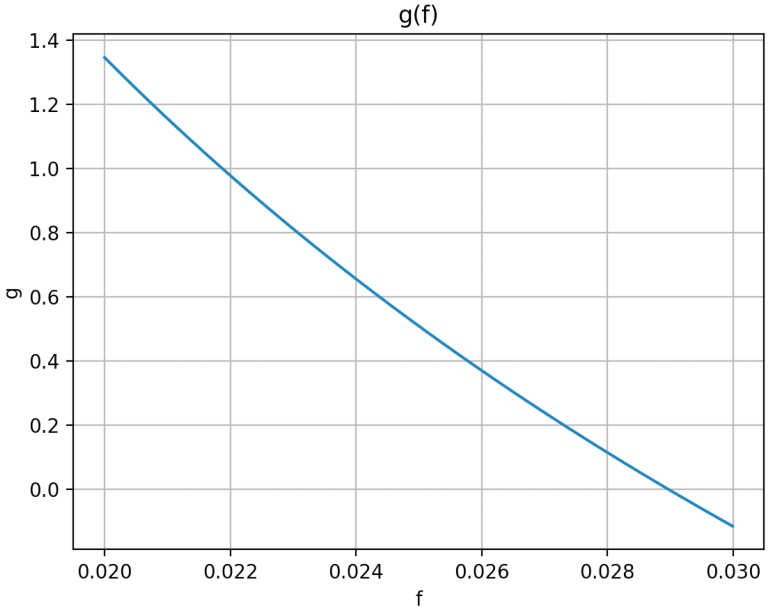
$$g(f) = \frac{1}{\sqrt{f}} + 2.0 \log\left(\frac{0.0000015}{3.7(0.005)} + \frac{2.51}{13{,}743\sqrt{f}}\right)$$

Graphical Method

My first approach to solve this problem is to estimate the root by plotting the function using the initial guesses of 0.008 & 0.08.



Visual inspection of the graph reveals the root to be between 0.02 and 0.03. To narrow it down further, I plotted another graph with the guesses as 0.02 and 0.03. This led me to the following graph:-



This graph clearly reveals the root between 0.028 and 0.03.

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 19 iterations.
The output is as follows:-

```
es100 =  0.005
false position:
iter      xl        xu        xr        ea
1         0.008     0.08      0.05698   85.9605%
2         0.008     0.05698   0.04468   27.5221%
3         0.008     0.04468   0.03792   17.8303%
4         0.008     0.03792   0.03412   11.1346%
5         0.008     0.03412   0.03196   6.7835%
6         0.008     0.03196   0.03071   4.0680%
7         0.008     0.03071   0.02998   2.4160%
8         0.008     0.02998   0.02956   1.4265%
9         0.008     0.02956   0.02931   0.8393%
10        0.008     0.02931   0.02917   0.4928%
11        0.008     0.02917   0.02909   0.2890%
12        0.008     0.02909   0.02904   0.1694%
13        0.008     0.02904   0.02901   0.0992%
14        0.008     0.02901   0.02899   0.0581%
15        0.008     0.02899   0.02898   0.0340%
16        0.008     0.02898   0.02898   0.0199%
17        0.008     0.02898   0.02897   0.0117%
18        0.008     0.02897   0.02897   0.0068%
19        0.008     0.02897   0.02897   0.0040%
R = 0.028969445362152145

Process finished with exit code 0
```

The false position method yields a root that is equal to 0.028969 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 9 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 16 iterations which is quicker than the false position method for the same error tolerance.

```
using bisection method:
iter        xl          xu          xr          ea
1           0.008       0.08        0.044       81.8182%
2           0.008       0.044       0.026       69.2308%
3           0.026       0.044       0.035       25.7143%
4           0.026       0.035       0.0305      14.7541%
5           0.026       0.0305      0.02825     7.9646%
6           0.02825     0.0305      0.02937     3.8298%
7           0.02825     0.02937     0.02881     1.9523%
8           0.02881     0.02937     0.02909     0.9667%
9           0.02881     0.02909     0.02895     0.4857%
10          0.02895     0.02909     0.02902     0.2423%
11          0.02895     0.02902     0.02899     0.1213%
12          0.02895     0.02899     0.02897     0.0607%
13          0.02895     0.02897     0.02896     0.0303%
14          0.02896     0.02897     0.02897     0.0152%
15          0.02897     0.02897     0.02897     0.0076%
16          0.02897     0.02897     0.02897     0.0038%
R = 0.0289674072265625

Process finished with exit code 0
```

The root of the function by Bisection method turns out to be R = 0.028967. From the results, it can be concluded that the bisection method performs slightly better than the false position method in this case. The bisection method obtains this root after 16 iterations with a percent relative error of 3.8×10^{-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 8^{th} iteration. This is quicker 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 bisection method is slightly superior to the false-position 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 9 iterations.

```
using secant method:
iter=0, xr=0.0516861151363558, ea=35.432891%
iter=1, xr=0.009283031064965336, ea=456.780590%
iter=2, xr=0.04084187508385286, ea=77.270801%
iter=3, xr=0.03527305545668735, ea=15.787744%
iter=4, xr=0.02720321223042653, ea=29.665038%
iter=5, xr=0.029241533551317502, ea=6.970638%
iter=6, xr=0.028979910268219045, ea=0.902775%
iter=7, xr=0.028967727709546903, ea=0.042056%
iter=8, xr=0.028967810196305854, ea=0.000285%
R = 0.028967810196305854

Process finished with exit code 0
```

This was a markedly improved performance as compared to the bracketing methods of false position and bisection. However, I had to modify the higher guess from 0.08 to 0.07 to obtain the results. The routine seems to diverge at the upper end of the range i.e at 0.08. This can be explained from the first graph. The graph jumps to a negative value at the upper end of the range. At values greater than 0.04, the graph tends to diverge.

The error approximation goes below 1% in the 7^{th} 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 0.08 and 0.008 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 friction factor using Colebrook's equation. It returns the root for the equation in 6 iterations.

```
modified secant method:
iter=0, xr=0.015825721673962213, ea=49.449383%
iter=1, xr=0.024258399755456468, ea=34.761889%
iter=2, xr=0.028426315866591587, ea=14.662175%
iter=3, xr=0.028964329522813567, ea=1.857504%
iter=4, xr=0.02896783497023163, ea=0.012101%
iter=5, xr=0.028967809992573312, ea=0.000086%
R = 0.028967809992573312

Process finished with exit code 0
```

The percentage error approximation falls below 1% in the 5th iteration. The modified secant method converges to the real root after 6 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 6 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=0.01576807188882761, ea=49.264564%
iter=1, xr=0.024152818030671278, ea=34.715395%
iter=2, xr=0.028369976227703122, ea=14.864863%
iter=3, xr=0.028958862727937896, ea=2.033528%
iter=4, xr=0.02896780811310703, ea=0.030880%
iter=5, xr=0.028967810171425943, ea=0.000007%
R = 0.028967810171425943

Process finished with exit code 0
```

The root of the function is equal to R = 0.028967. The percentage error approximation goes below 1% in the 5th 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 0.08, the routine diverges. This is the expected output during this computation.

```
Traceback (most recent call last):
es100 =  0.005
  File "/Users/rahulshridhar/PycharmProjects/Scientific_Computing_I/projects/final_project/project.py", line 57, in <module>
using newton-rhapson method:
iter=0, xr=-0.021842962868496754, ea=466.250680%
    R = _roots2.newton(g, df=dgdf, x0=0.08, 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/project.py", line 25, in g
    return 1/math.sqrt(f) + 2.0 * math.log10(E/(3.7*D) + 2.51 /(R * math.sqrt(f)))
ValueError: math domain error

Process finished with exit code 1
```

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.

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 6 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.