CS374
Numerical and Computational Methods

Dhirubhai Ambani Institute of Information and Communication Technology

Jigar Zanzarukiya 201801224
Hiren Chaudhary 201801438
Naman Dave 201801439
Deep Patel 201801443

Mentor: Prof. Madhukant Sharma

Abstract

This project aims to perform the analysis on some of the real world applications of some of the numerical and computational methods computationally using Matlab, python as well as different tools and HWs like google collab, GPUs, CUDA. These methods are very useful in getting an estimate and solving various engineering and physics problems. This project tries to solve some of this project along with some analytical results providing insight into the problem statement.

Content

Problem 1

Newton Raphson Method, Finding extreme points of a function, then methods to find maxima and minima explicitly, and then implementing Gradient Descent using Newton Raphson Method (Matlab)

Approach

Newton Raphson Method is used to find roots of a function by iterative manner, we can use the same technique to find maxima or minima of a function. Furthermore, we can also modify this method to find only the minima as well as the maxima. And then onwards we implement the most widely used optimizer for DL newralnets, i.e. Gradient Descent which is also a slight change in Newton Raphson Method with a few assumptions and some additional parameters which are called as hyperparameters.

1. find extreme point, modification in newton method to find an extreme point (f' = 0)
   Here, the extreme point could be a local/global maxima/minima
2. find local maxima and minima by modifying newton's method
3. Further modification of problem II to implement gradient descent

I) find extreme point, modification in newton method to find an extreme point (f' = 0)

- An extreme point is a point where f' vanishes.We take g = f' and find roots of g using newton's method,Implementing newton's method, we get

➢ $$X_n = X_{n-1} - \frac{g(X_{n-1})}{g'(X_{n-1})}$$

➢ $$X_n = X_{n-1} - \frac{f(X_{n-1})}{f''(X_{n-1})}$$

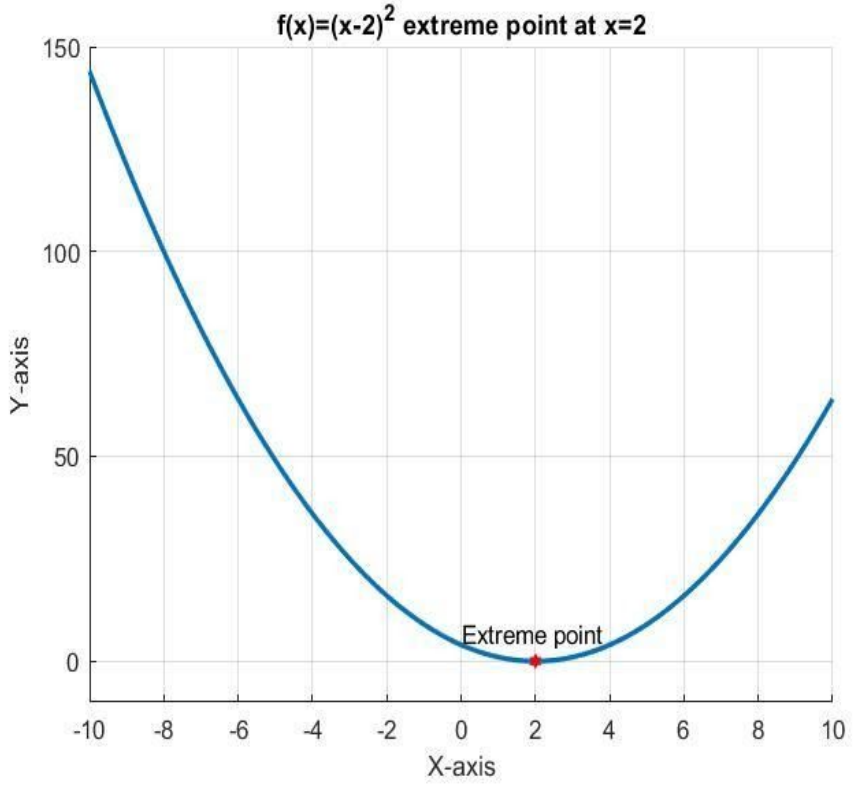
➢ $$e_{n+1} = e_n^2 \left| \frac{f'''(X_{n-1})}{f''(X_{n-1})} \right|$$

Here, the extreme point could be a local/global maxima/minima

Extreme Point Examples:

1. $f(x) = (x-2)^2$ where the range of x is [-10,10].

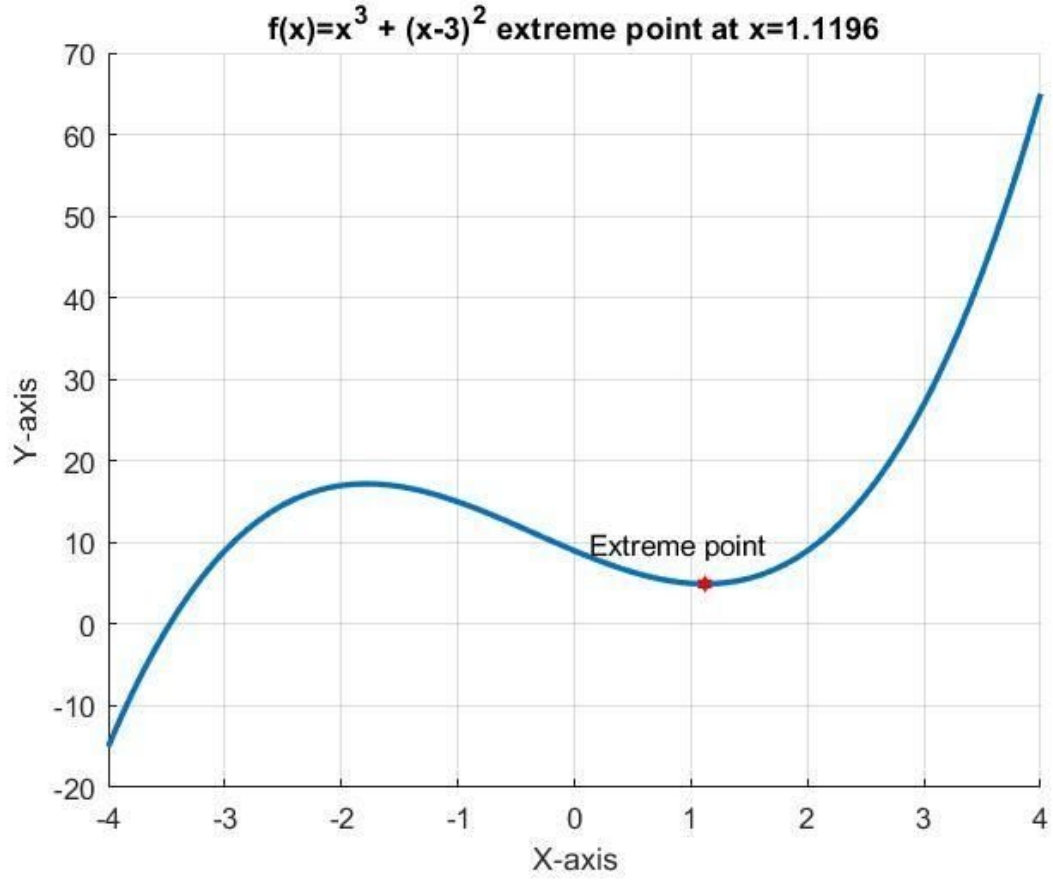
Result:



- Extreme point found at x=2.

2) $f(x) = x^3 + (x-3)^2$ where the range of x is [-4,4].

Result:



- Extreme point found at x=1.

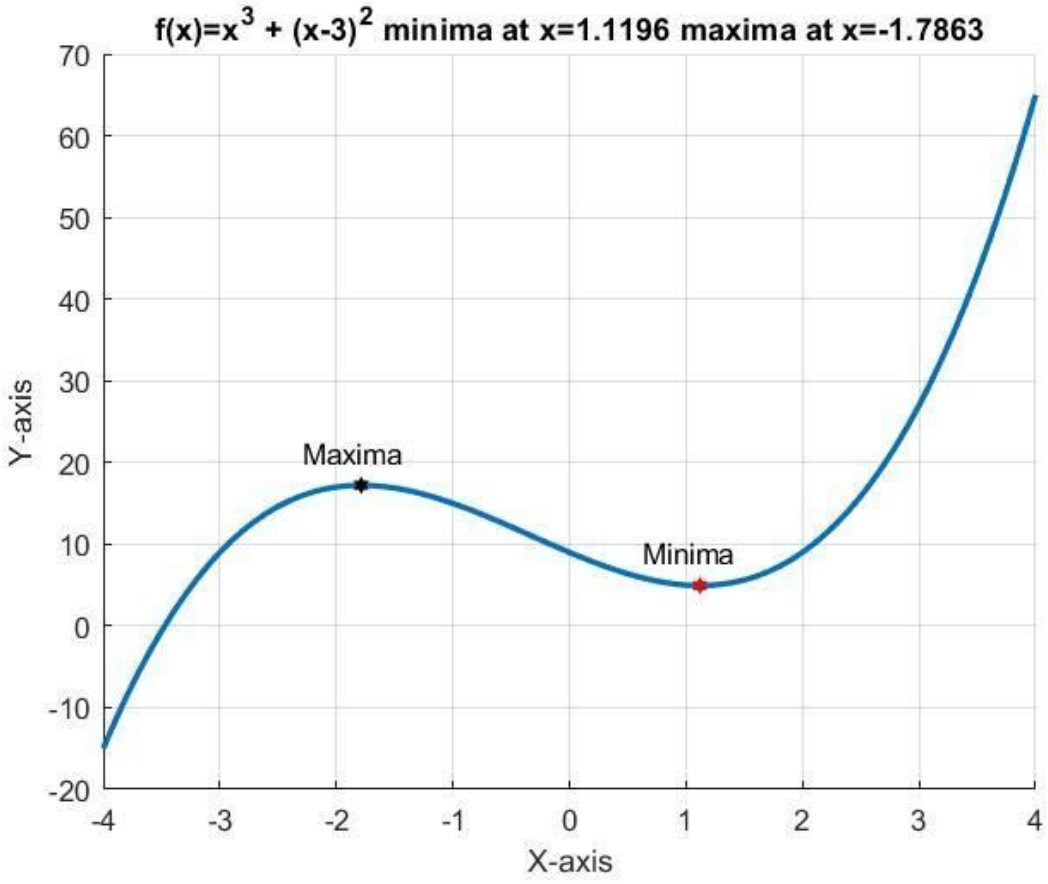
II) Find local maxima and minima by modifying newton's method.

- For local minima, we can use
  - $X_{n+1} = X_n - \frac{f(X_n)}{|f''(X_n)|}$
- For local maxima, we can use
  - $X_{n+1} = X_n - \frac{f(X_n)}{-|f''(X_n)|}$

Example: Maxima and Minima explicitly

$f(x) = x^3 + (x-3)^2$ where the range of x is [-4,4].

Result:



- Minima found at x = 1.1196
- Maxima found at x = -1.7863

III) Further modification of problem II to implement gradient descent

- For Gradient descent algorithm, the equation is
  ➢ $X_{n+1} = X_n - LR * f'(X_n)$

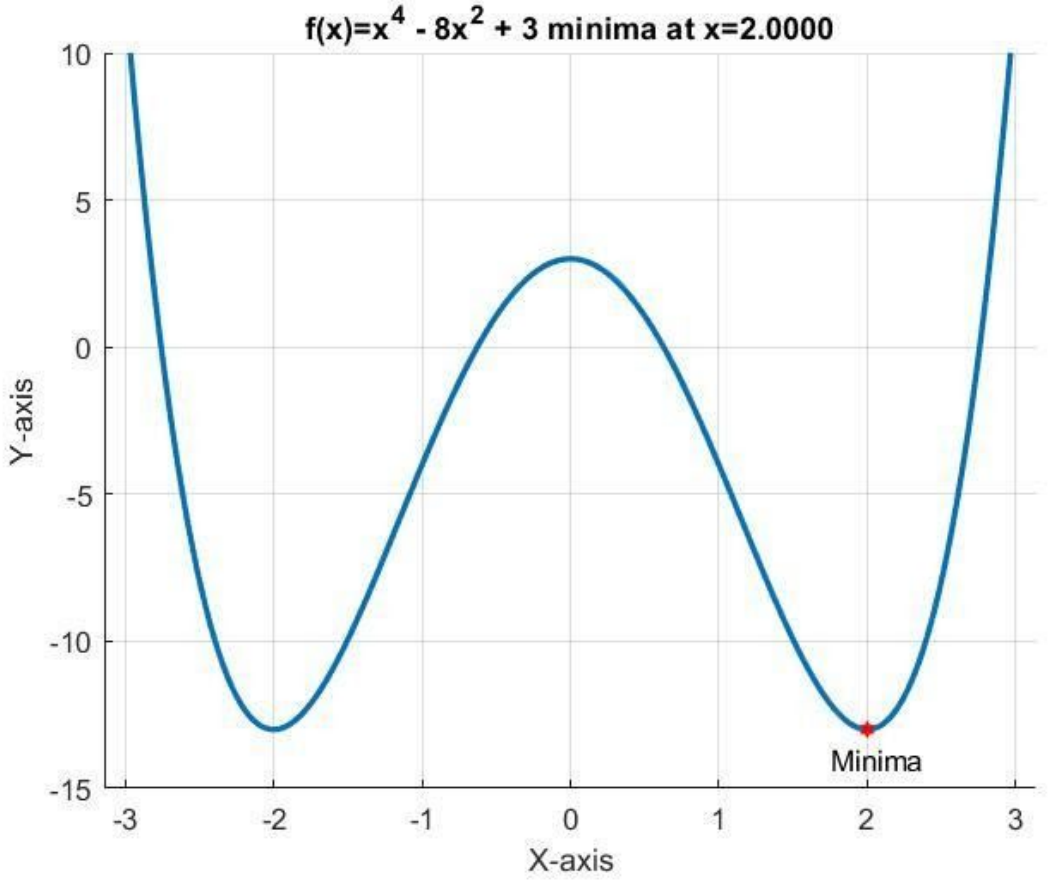
  Where LR is Learning Rate which controls step size.

Gradient Descent example:

$f(x) = x^4 - 8x^3 + 3$ where the range of x is [-3,3]

Initial value of x is 0.01

Result:



- Algorithms terminate at x = 2.0 which is a local minima.

Conclusion:
Here, we've seen different (modified) to calculate the specific outputs.
Modification to the basic newton-raphson's method will give us extrema, maxima, and minima. We can also change the sensitivity of the steps taken by further modifying methods mentioned above, from that intuition we can further achieve the desired results.

Future scopes:
Image processing, NLP and almost all the ML models nowadays use core optimizers as the gradient descent optimizers. In order to get higher accuracy, there are more advancements in the region of GDO/SGD such as SAGA (Stochastic Average Gradient Descent), Momentum (Nesterov), RMS Prop, Adam, RAdam, etc, which are the advancement of SGDs (Stochastic Gradient Descent). In all the ML scenarios like image processing and analyses, we first generate cost function and we find the

local minima of cost function in order to tune our weights and biases close to the ideal solution. Advancement in these algorithms also provides much more stability to the algorithm, the former algorithm may omit the nearmost minima but the advanced algorithms will make sure that it lies nearer to the exact/expected solution of the desired problem.

Problem 2

F1 race car tuning (Using Secant Method): Racing isn't just speed, however, it is the most important when the road is straight but in normal F1 races the roads are curvy. So the car's efficiency also depends upon two other factors; 1) Acceleration, 2) Handling. Now an F1 race car is finely tuned when the max-speed efficiency equals the handling efficiency times 10 plus acceleration times 0.23. Our goal is to find the point where the tuning happens (if multiple tuning points then find the optimum one) (Matlab). As the efficiency curves' function may not be differentiable at all the points, so we can not use methods involving derivatives, therefore we use the Secant Method

Approach

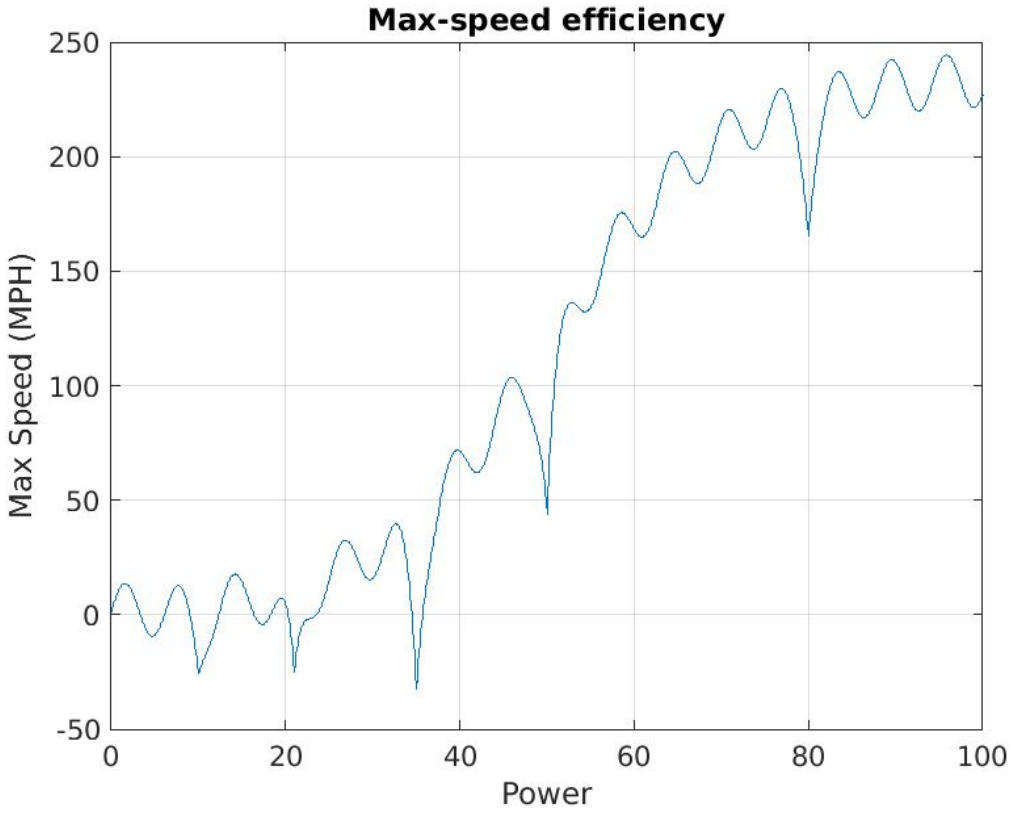
Here, we have functions for max-speed, acceleration and handling, the tuning point is where the maximum speed equals 10 times handling and 0.23 times acceleration, for that we need to find zeros of

$$f = max\,speed - 0.23 * acceleration - 10 * handling$$

The functions are mentioned in further slides. The optimal tuning point is the maximum of all the tuning points.

I) Max-speed efficiency function

max_speed(x) = $235 * [(\frac{e^{0.1(x-50)}}{e^{0.1(x-50)}+1}) + 0.05sin(x) - (0.1e^{-|x-10|} + 0.2e^{-|x-21|} + 0.3e^{-|x-35|} + 0.3e^{-|x-50|} + 0.2e^{-|x-80|})]$

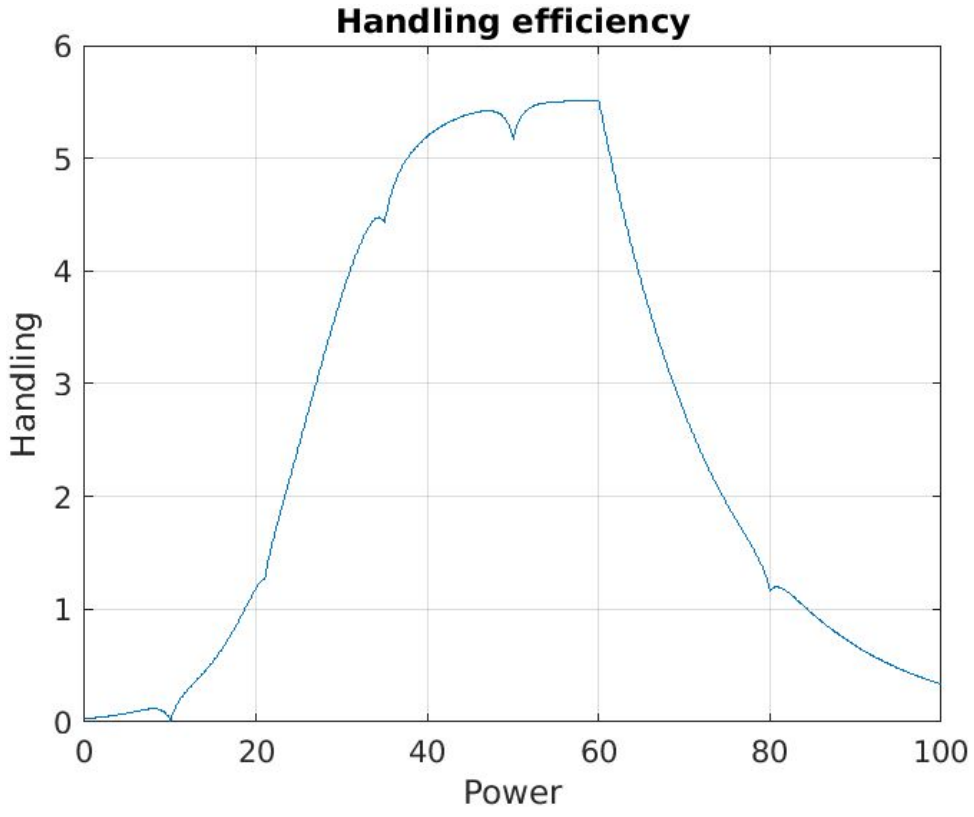


II)Handling efficiency function

handling(x) = $5.519[\{(\frac{e^{0.2(x-max_{handling})}}{e^{0.2(x-max_{handling})}+1})$ if $x \leq max_{handling}$

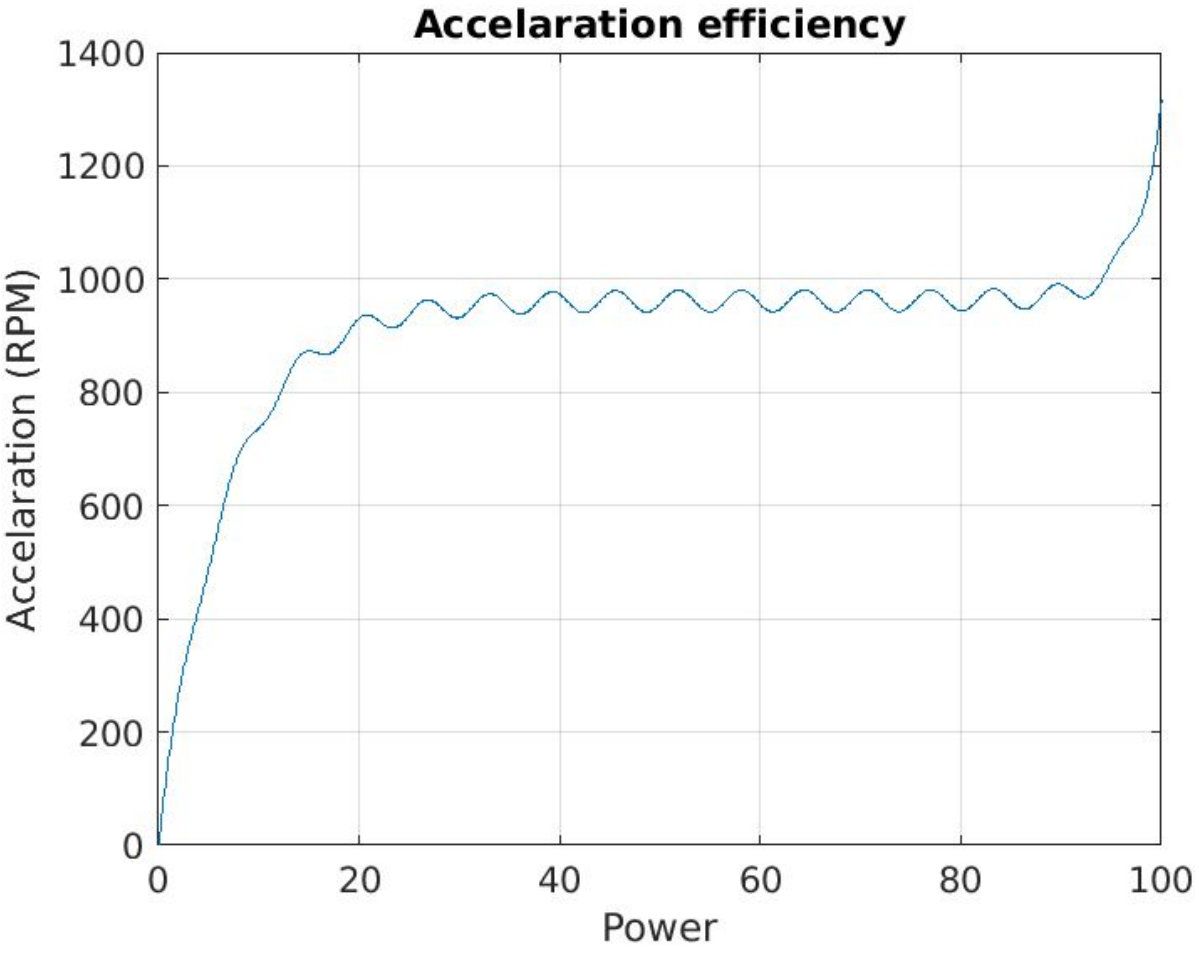
$e^{-0.07(x-max_{handling})}$ if $x > max_{handling}\}$

$-(0.1e^{-|x-10|} + 0.2e^{-|x-21|} + 0.3e^{-|x-35|} + 0.3e^{-|x-50|} + 0.2e^{-|x-80|})]$

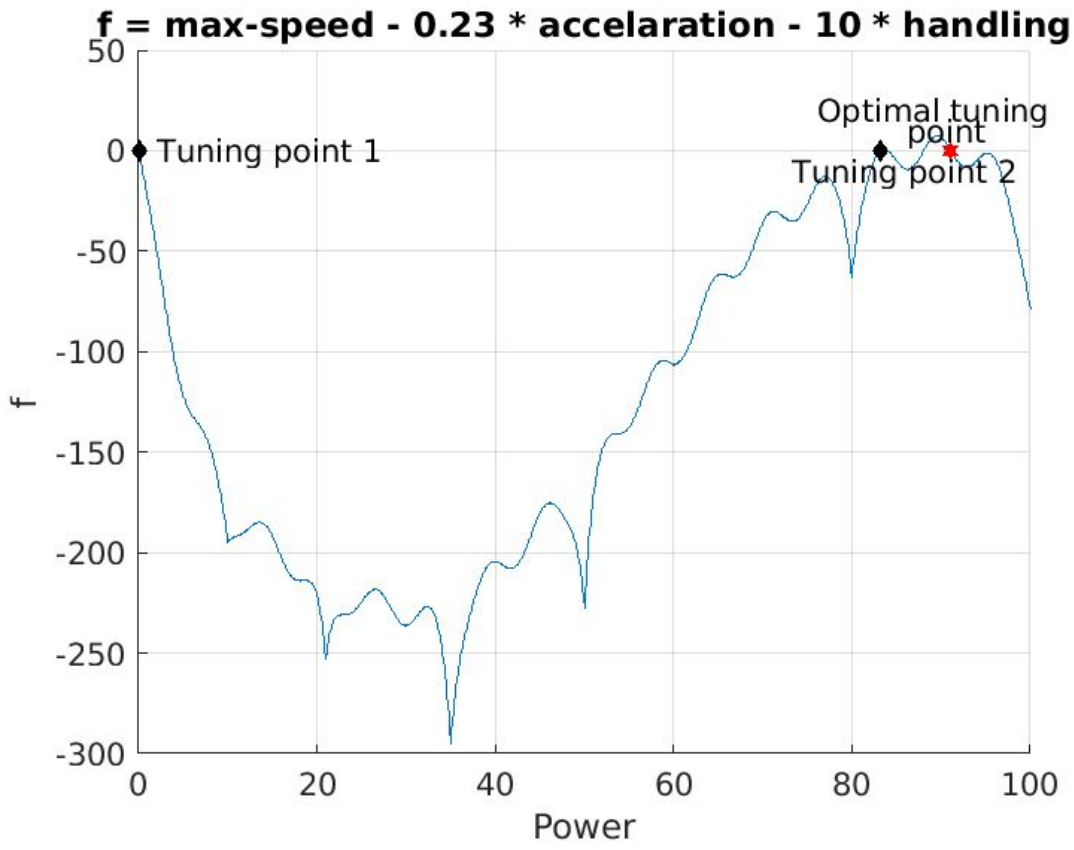


III)Acceleration function

acceleration(x) = $\frac{1}{1.04x10^{-3}}(1 - e^{-0.15x} + 10^{-7}e^{e^{e^{0.01x}}} + 0.02sin(x))$



$$f = max\,speed - 0.23 * acceleration - 10 * handling$$



Result:

Optimal Tuning Point	: 91.1207
Tuning Point 1	: 0.0497
Tuning Point 2	: 83.2633

Conclusion:
The secant method can be used for obtaining the solution of the equation $f(x) = \alpha g(x) + \beta h(x)$, secant solver does not include the derivation of functions because some functions may not be differentiable at all the points.

Future Scope:

This is the tuning point for an F1 car. However for economical use of automobiles, in order to get the optimal point in the desired region, the parts and machines are chosen s.t. they give perfect tuning in the desired power regime. Also for different vehicles we have different weights to max speed handling and acceleration. This also depends on the use of the automobile (car, truck, Tunnel Drilling Machine, etc.).

Problem 3

Comparison of Polynomial Interpolation (Lagrange and Newton's interpolation) and Polynomial Regression by accuracy, sensitivity and other comparison matrices Comparison of completely fitting curve vs. best fitting hyperplane (Python: ML).

Approach

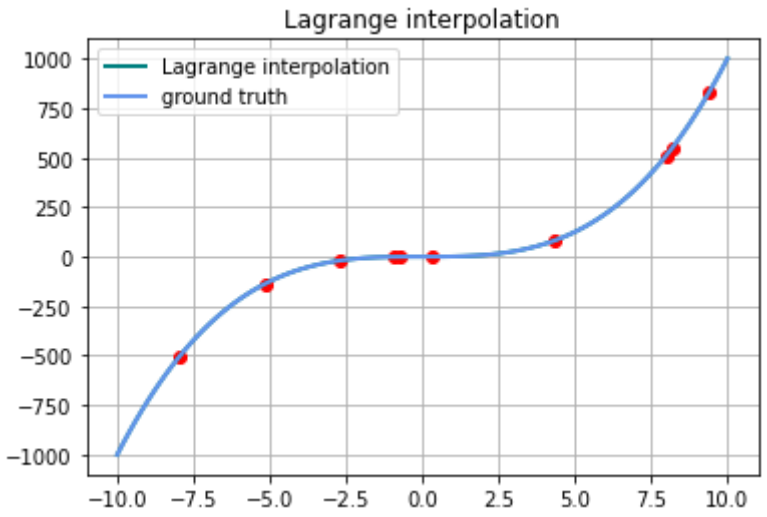
First we consider some data points and we fit our polynomial using these methods and then we predict the datapoint with a given set of attributes that will give us the accuracy. Also adding errors will give the approx idea about the sensitivity of methods.(Error Used: Summed Squared Error). In code we have used lagrange interpolation to predict polynomials and for ML model we have used over fitting polynomial features / kernel of linear regression and Stochastic Average Gradient Descent Optimizer for getting weights and biases of each kernel index / axis. In linear regression first random initialization polynomial coefficients then using gradient descent to fit data points on predicting polynomials.

Results:

I) Randomized coefficients of polynomial:

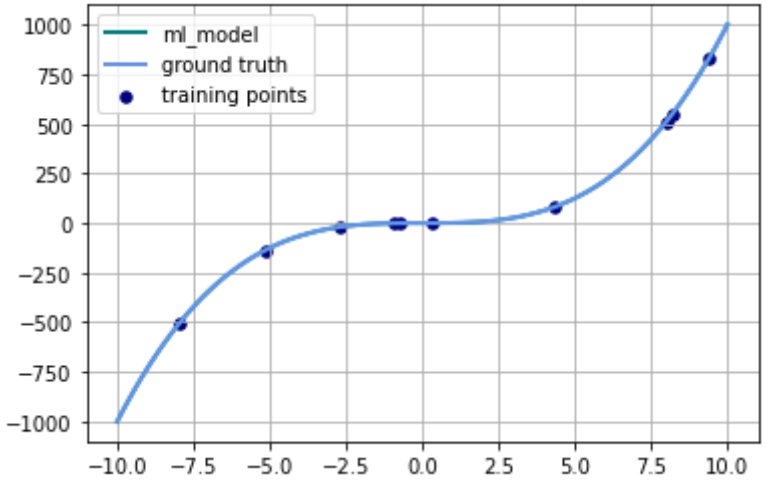
Random_coffs: 0.64442979 0.46303623 -1.1214624

- Lagrange interpolation



Lagrange interpolation error is: 3.4784069720843e-21

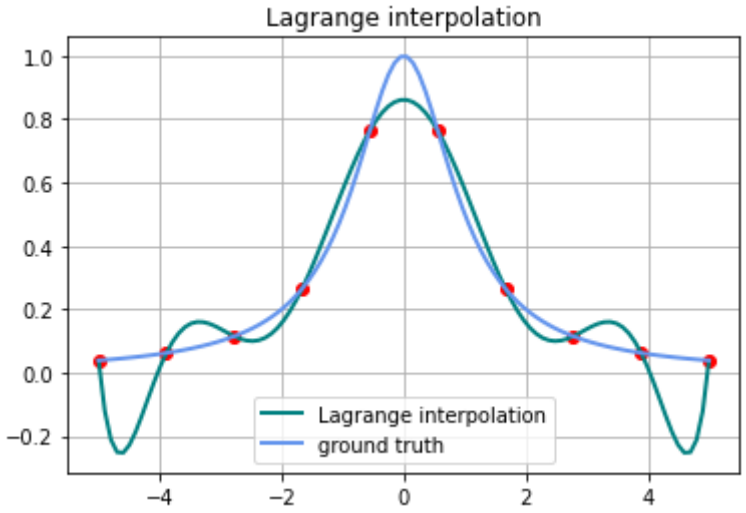
- ML_model



Polynomial regression error is :1.5406403549548525e-24

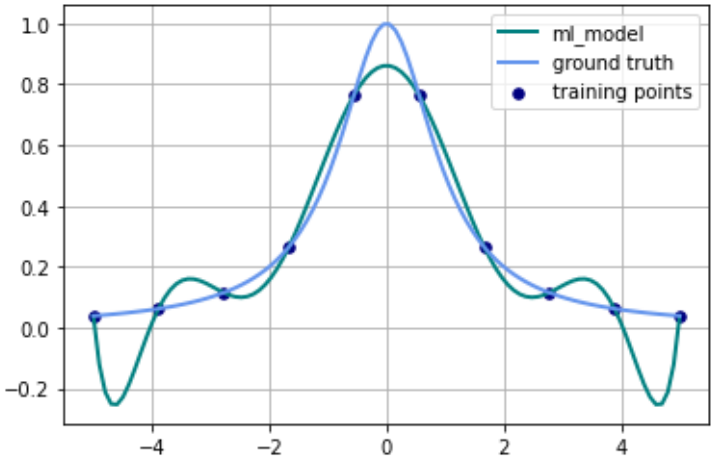
II) $f(x) = \frac{1}{1+x^2}$, It has infinite degrees

- Lagrange interpolation



Lagrange interpolation error is: 1.1888696081316732

- ML_model



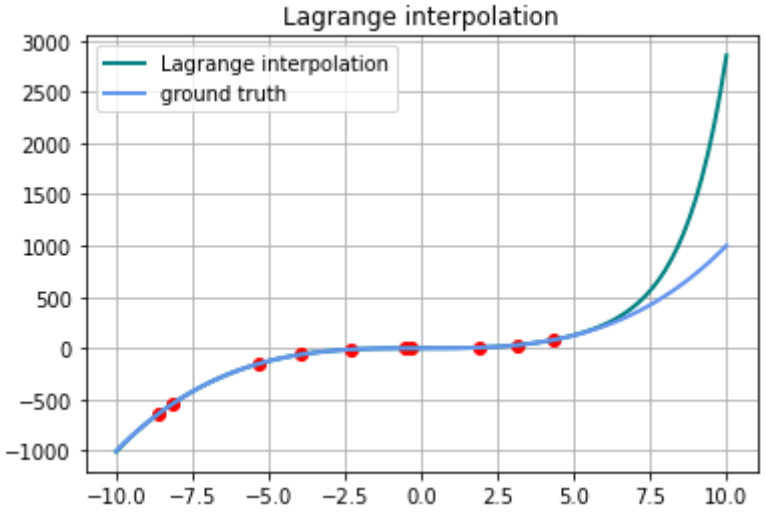
polynomial regression error is : 1.188869608122818

- Lagrange error is 8.855138844410249e-12 more than Poly-regression

III) Error Sensitivity:

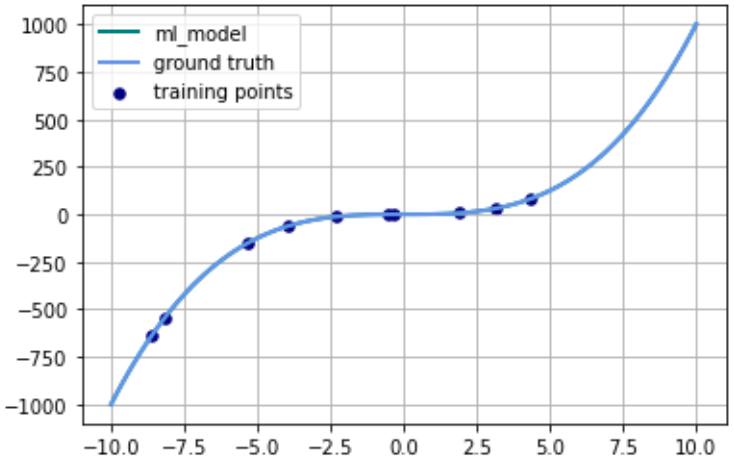
- We add some errors in y values. The interpolation gives more error than polynomial regression. Polynomial regression is more immune to error and interpolation is more prone to the slight error.

- Lagrange interpolation



Lagrange interpolation error is : 10922229.878232282

- ML_model



polynomial regression is : 0.0870671028771425

Conclusion:

Interpolation techniques tend to completely fit the data points, when the polynomial regression without regularization tends to overfit the data points and polynomial regression with regularization fits to the data points optimally. As the new points are added for testing, the interpolation may or may not give accurate results as compared to the Polynomial Regression. The polynomial regression is more immune to the error as it removes the errors and fits the best towards the actual result/polynomial.

CODE :
https://colab.research.google.com/drive/1QhFAYGEOPbPywXU0COiX-AaRpMt_BDZj?usp=sharing

Future Scope

In real life we have some value(data points) and we have to estimate another value then interpolation and polynomial regression work. Also in real life the rate of sensitivity is high so polynomial regression is better than interpolation technique so using regression we create polynomials then predict other value exams. housing price prediction. We have the size of the house and its value so create a polynomial using regression and then predict house price which size is not listed in our input data point.

Problem 4

Cholesky Decomposition Compression of a symmetric matrix vs. splitting compression of symmetric matrix. Computational time comparison using GPU operations (Matlab/Python: GPU):

Approach

Compression of a symmetric matrix can be done using two methods:

1. Cholesky Decomposition,
2. Consider only the lower triangle with diagonal (Splitting).

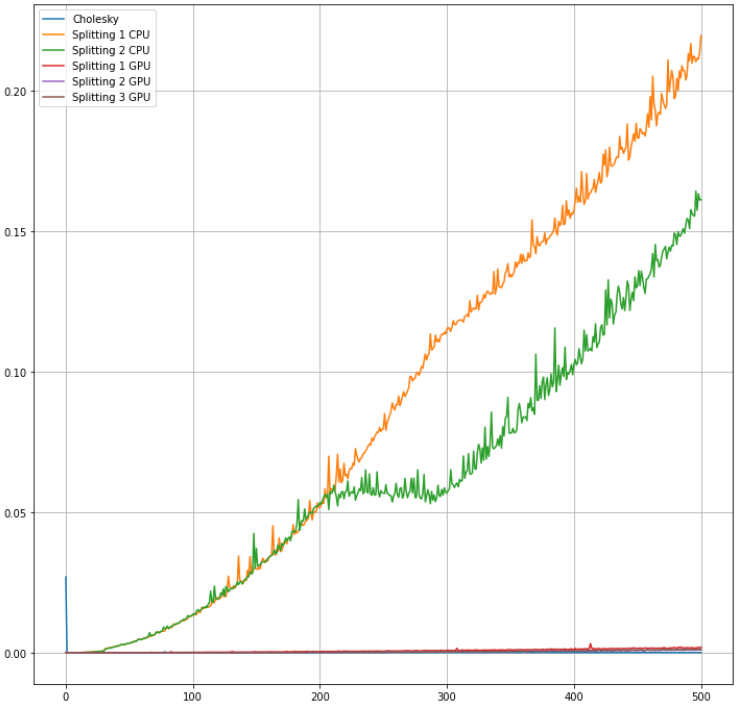
Now retrieving original matrices, we have one method for the Cholesky Decomposition which is to perform matrix multiplication.

While retrieving splitted matrix, we have three methods;

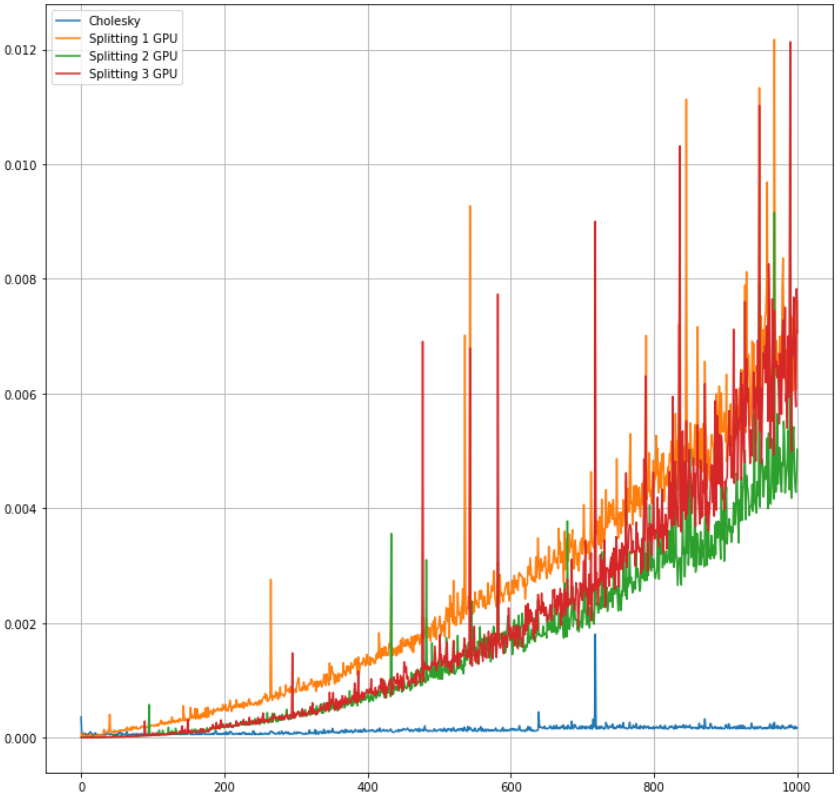
1. Two Loops
2. Add matrix with its transpose and half the diagonal
3. Decompose the compressed matrix with two matrices; a lower triangle matrix with zeros on diagonal and a diagonal matrix. Add the lower triangular matrix with its transpose and add the diagonal matrix.

Now the main performance criteria we consider here is the time taken by these methods, because the time taken also depends upon the hardware selection, as the Cholesky decomposition takes GPU/TPU as a primary hardware which is optimized for matrix multiplications and the rest may or may not use GPU, our goal is to compare all the methods on retrieving time and give the final results.

I) Cholesky reconstruction, CPU without time optimization, with Numpy GPU

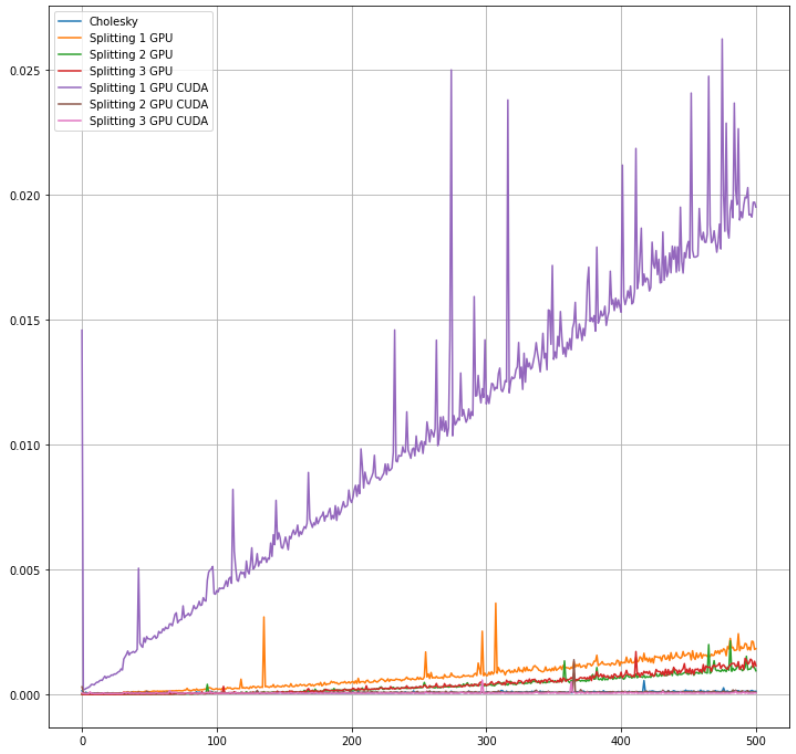


II) Cholesky and Numpy GPU optimization

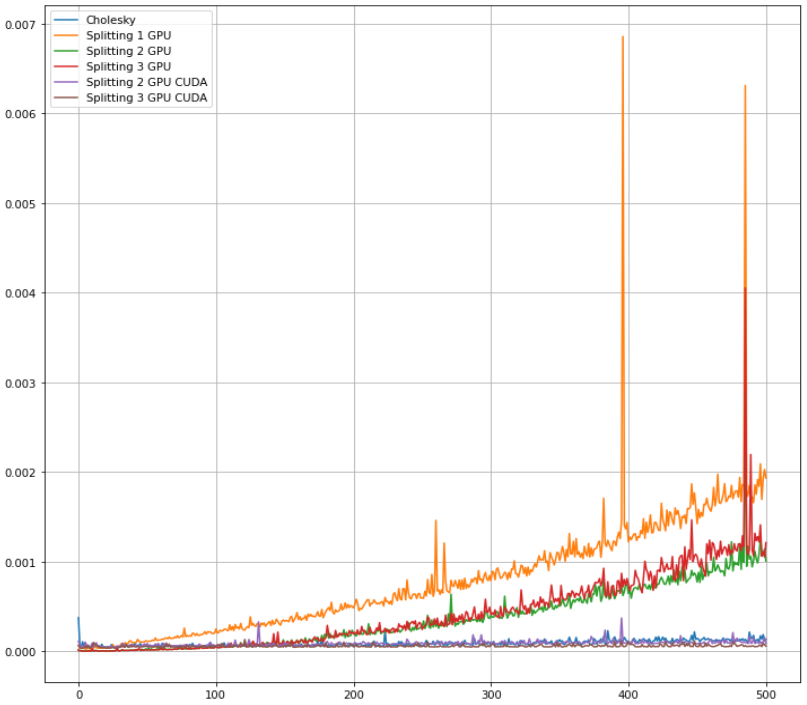


III) Cholesky, Partial CPU/GPU, Numpy GPU, and CUDA pytorch GPU

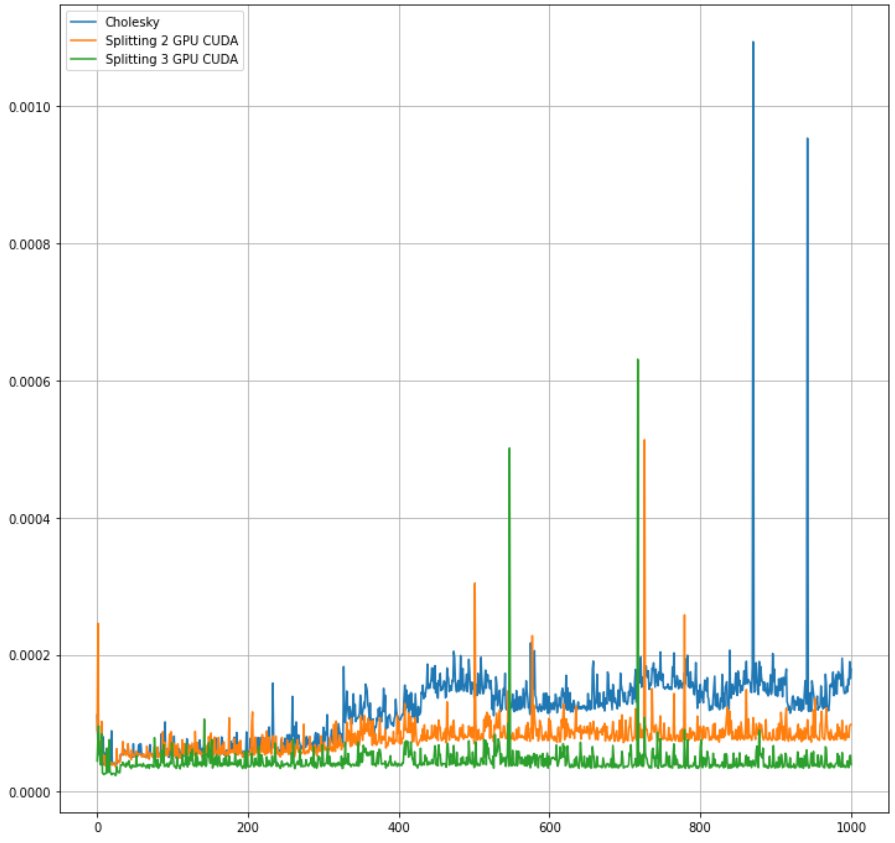
- **Splitting 1 GPU CUDA is partial CPU/GPU**



IV) Cholesky and Numpy GPU and CUDA NVIDIA GPU time comparison



V) Cholesky and Splitting on GPU CUDA(NVIDIA)



Conclusion:

CPU operation takes more time than GPU. So it is efficient to use Cholesky on CUDA GPU than Splitting matrices on CPU even with Numpy GPU.

Now the splitting matrix function can be converted to GPU operations and due to the very high amount of cores in CUDA, it takes almost constant time to finish the task (200 microsecs for 1000 x 1000 SPD) with too much speed.

Code:
https://colab.research.google.com/drive/1phymcBF7dKPibB-98ZBtPuyiqj4omi_Z?usp=sharing

Problem 5

Applications of various numerical integration methods(Random Interval Riemann sum / Riemann integration, Simpsons method, Trapezoidal method) and comparing their accuracies by using these methods on some crazy integrals.

Part 1: Obtaining the value of omega constant Ω.

Ω omega constant is a constant which satisfies the integrals mentioned below:

$$\int_{-\infty}^{\infty} \frac{dx}{(e^x - x)^2 + \pi^2} = \frac{1}{1+\Omega}$$

or

$$\Omega = \frac{1}{\pi} Re \int_0^{\pi} log\left(\frac{e^{e^{it}} + e^{it}}{e^{e^{it}} - e^{it}}\right) dx$$

Where **Ω** omega constant,

Omega constant is a constant satisfying the below condition:

The **omega constant**

$$\Omega e^{\Omega} = 1.$$

The value of **Ω** is:

$\Omega = 0.56714\ 32904\ 09783\ 87299\ 99686\ 62210$ ... (sequence A030178 in the OEIS).

$1/\Omega = 1.76322\ 28343\ 51896\ 71022\ 52017\ 76951$ ... (sequence A030797 in the OEIS).

Ref:

1. https://en.wikipedia.org/wiki/Omega_constant
2. https://math.stackexchange.com/questions/3027621/request-for-crazy-integrals

Now using the second integral, we will find the value of **Ω** using several numerical methods.

Below is the script for finding the estimated value of **Ω** using three different methods;

1. Random Interval Riemann Sum / Riemann Integration
2. Simpson Integration
3. Trapezoidal Integration

Value of Ω using Riemann sum/integration: Ω = 0.567143
Value of Ω using Simpson's integration: Ω = 0.000000
Value of Ω using Trapezoidal integration: Ω = 0.567143

Observations:

Simpsons integration could not provide the answers and the rest could.

The reason can be that the function includes complex numbers for the integrations and there we can not use Simpson's integration.

Prob 2: Using the value of π, get the value of e, and compare these algorithms' error and efficiency.

Let's check these algorithms' accuracy by comparing the estimated values with the actual value.

For that let's take a crazy integration mentioned below:

$$\int_0^1 \frac{\sin(\pi x)}{x^x(1-x)^{1-x}}dx = \frac{\pi}{e}$$

Here, using the value of π, we can obtain the value of e by the equation below:

$$e = \frac{\pi}{\int_0^1 \frac{sin(\pi x)}{x^x(1-x)^{1-x}} dx}$$

Below is the script of the above problem and comparison of different methods by their accuracies.

Method: Riemann sum/integration
n = 3000000
estimated value of e = 2.718282
Error: 1.790359E-10

Method: Riemann sum/integration
n = 10000000
estimated value of e = 2.718282
Error: 1.560529E-11

Method: Simpsons integration
n = 3000000
estimated value of e = 2.718282
Error: 4.440829E-16

Method: Simpsons integration
n = 10000000
estimated value of e = 2.718282
Error: 0.000000E+00

Method: Trapezoidal integration
n = 3000000
estimated value of e = 2.718282
Error: 2.815526E-13

Method: Trapezoidal integration

n = 10000000
estimated value of e = 2.718282
Error: 1.434408E-13

Observations:

Simpson's method is fast and accurate than any other method but it is much more sensitive as we have seen in the first problem, it could not converge to the actual answer because it involved complex numbers in integrations.

The most stable method for integration is the Random Interval Riemann Sum or Riemann Integration.

The Trapezoidal integration method is the intermediate of the Simpsons method and Riemann integration with random intervals in terms of stability, accuracy.

Future scope:

While finding some unknown constant from integration then using this method we find unknown constant. sometimes integration not done easily like in probability x is distribute with u mean and variance σ then we use this method to count probability.