# CS5016: Computational Methods and Applications

## Lab Report - 7

111901030
Mayank Singla

**Q1.**

I am creating the function `solve1DHeatEquation` which takes the required parameters to solve the 1D heat equation using the finite difference method. I am generating random points on the rod and using the initial conditions, I am creating the values of `u` and then using for loop and matrix representation formula, evaluating the values of `u` at different times and storing them and returning them. Then using the values returned from the function, I am simply creating the animation and plotting it using `matplotlib.pyplot.imshow` function.

I am taking the `μ` value here to be as `5 * 1e-5` as for `μ = 1`, the values were too high to be plotted. The animation values are written for a small time only as it was taking a long time otherwise.

**Q2.**

I am creating the function `solve2DHeatEquation` which takes the required parameters to solve the 2D heat equation using the finite difference method. Again, as in the previous question, here I am generating random 2D points in the given region uniformly and this time using the formula from the derivation of the 2D Heat equation in the theory lecture, I am evaluating the values of `u` at these points and returning from the function. Then again as in the previous question, I am creating the animation to plot the curve.

I am taking the `μ` value here to be as `5 * 1e-5` as for `μ = 1`, the values were too high to be plotted. The animation values are written for a small time only as it was taking a long time otherwise.

**Q3.**

I am creating the function `computeNthRoot` which is the required function in the question. I am creating a nested function for the equation $x^n - a = 0$ and then finding the root of this equation using the bisection method in the interval `[0, a]` and computing the answer.

Here is an example output I obtained.

```
m = 19
num = 6**m
eps = 0.00001
print(f"The {m}th root of {num} is
{computeNthRoot(m, num, eps)}")
```

```
The 19th root of 609359740010496 is 6.000003184560722
```

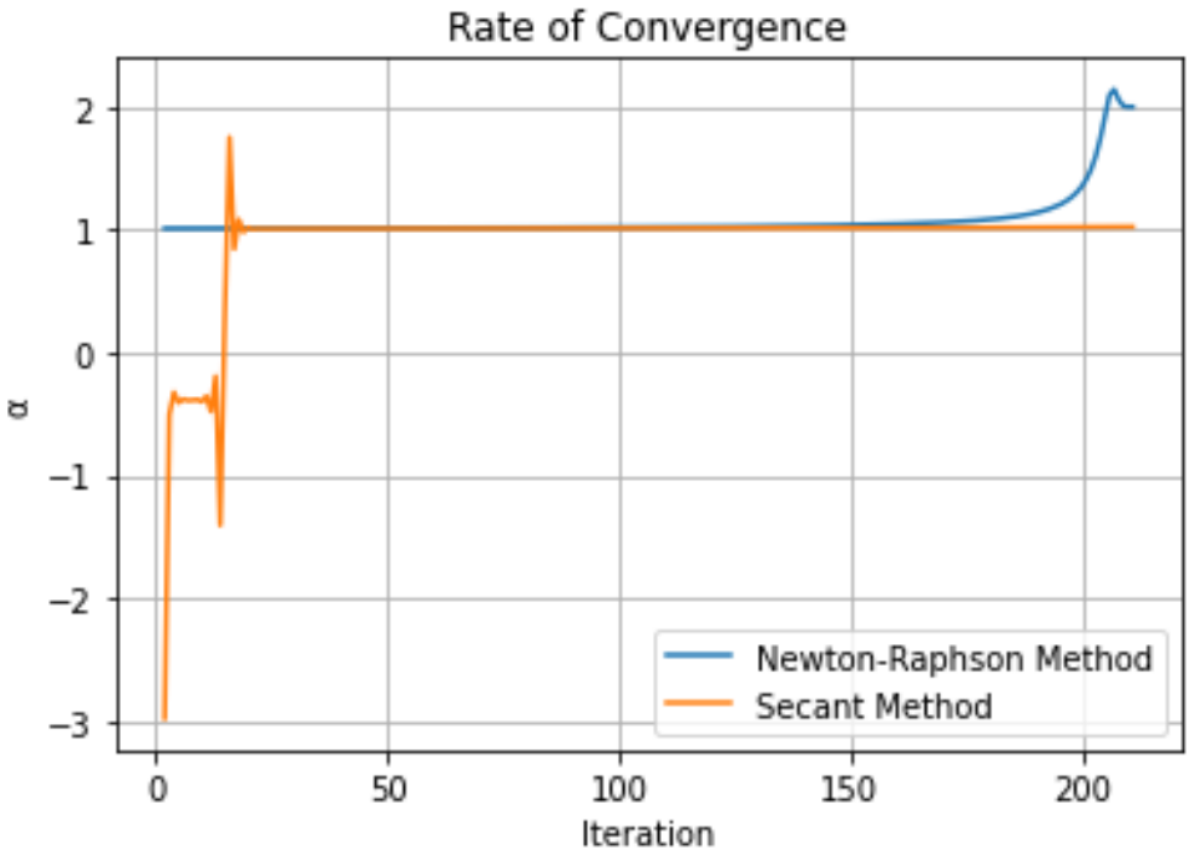
**Q4.**

Here, I am creating two functions `newton_raphson_method` and `secant_method` to approximate the roots of the input function using the corresponding methods using simple for loops and their standard formulas and returning the values obtained. Then I am creating the function `get_rate_of_convergence` which computes the rate of convergence of the sequence of points using the formula *(found on the internet)*

$$\alpha = (\text{Log } |(x_{+1} - x) / (x - x_{-1})|) / (\text{Log } |(x - x_{-1}) / (x_{-1} - x_{-2})|)$$

Using this function, I am calculating the rate of convergence of both methods for each iteration and then finally plotting the curves for both.

Here is the curve I obtained for the function $f(x) = x * e^x$




From the plot, we can see that rate of convergence of the secant method is less than that of the newton method.

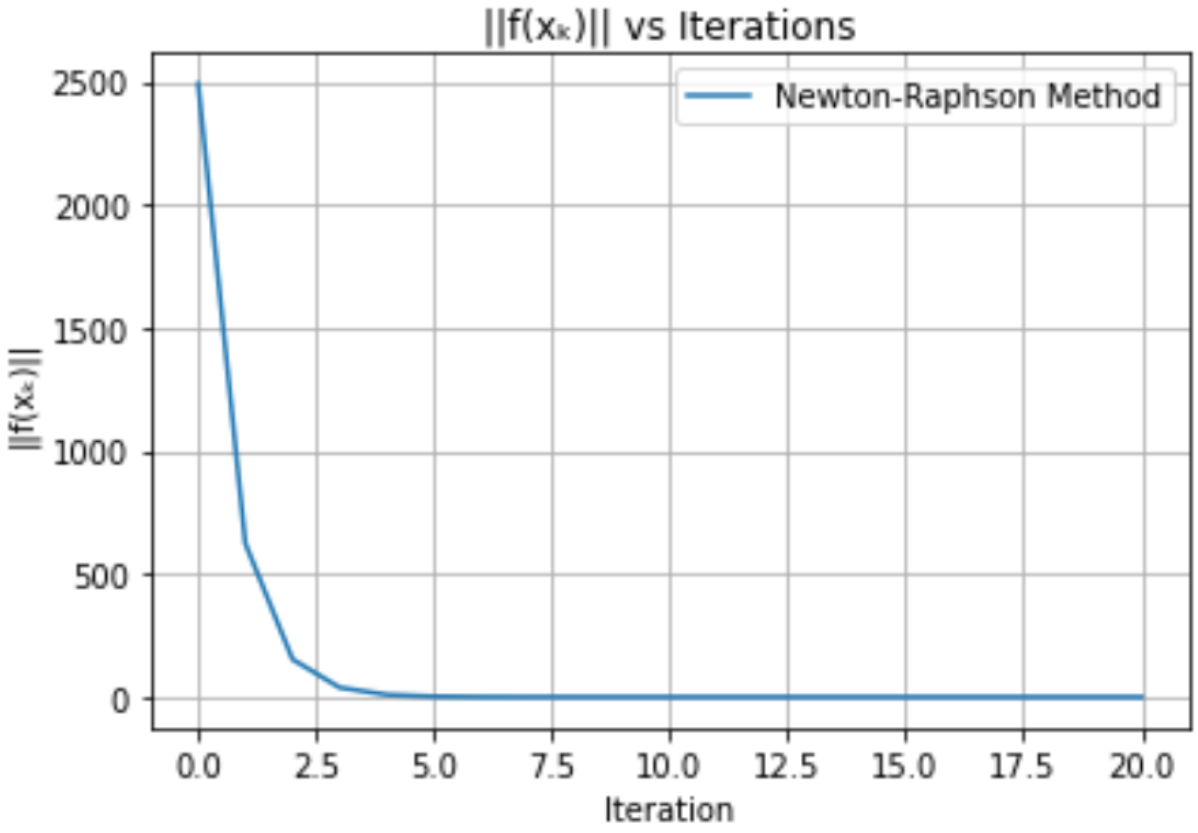
**Q5.**

I am again creating the function `newton_raphson_method` to approximate the function which is in the multivariate vector-valued form to find the roots of the equation. I am using the formula we saw in the theory lecture to compute the roots. I am creating the functions `inp_fun` and `inp_fun_jacobi` to find the values of the function and the values of the **Jacobi** matrix of the function and passing them as the arguments to my main function. Then finally I am printing the results and plotted the required plot.
Here are the results I obtained.

```
numIter = 20
x0Init = [1, 2, 3]

# Applying the Newton-Raphson Method
xNR = newton_raphson_method(inp_fun, inp_fun_jacobi, x0Init, numIter)
```

```
The root of the function is [ 0.83328161  0.03533462 -0.49854928]
```




**Q6.**

I am creating the function `computeRootsAberth` which is the required function mentioned in the question. I am creating the polynomial `g(x)` as mentioned in the question using the overloaded methods from the `Polynomial` class and then I am creating the function `printRoots` in the `Polynomial` class which evaluates all the roots of the polynomial and then finally I am printing the results.

In the `printRoots` function, I am first of all calculating the Lagrange upper and lower bound of the roots of the polynomial as mentioned in the provided link and then initializing the roots of the polynomial as complex numbers with their real parts between lower and upper bounds.

Then I am creating a nested function `hasConverged` to check if the roots obtained have converged or not.For this I am using the logic that `df / dx = f'(x)`, then `dx = df / f'(x) = f(x) / f'(x)`

So, we need `dx < epsilon` and we can compute this quantity from the roots to check for convergence. This way I will make sure that the error is within the limits of $10^{-3}$.

Then using a simple loop and the formula for the Aberth method, I am evaluating the roots till the convergence is reached and then return the values from the function.
Here is the output I obtained.

```
computeRootsAberth([1, 3, 5, 7, 9], 1e-3)
```

```
The roots of the polynomial g(x) = -945 + 1689x - 950x² + 230x³ - 25x⁴ + x⁵ are:
[(6.999998822931529+3.2354032343528896e-07j),
(0.9999999999999998+7.485256624665854e-19j),
(9.000000000000048-2.7301443167994863e-30j),
(2.9999999981128513-1.0023095210255108e-09j),
(5.000003933439729-3.2049125083949415e-06j)]
```

We can observe that the real parts of the roots obtained are close to the actual roots and the imaginary parts are almost 0.

**Q7.**
I am creating the function `computeZerosFunction` to approximate the zeros of the function in a given interval. I am approximating the input function using the `bestFitFunction` we created for the `Polynomial` class a few labs before to get a polynomial approximation for the function and then using the `printRoots` function created in the previous question, I am evaluating the roots of the function. I have modified the `printRoots` function to take an optional argument so that I can obtain only real root from that function because in the case of complex numbers we can't talk about intervals. Since I am using the same function as in the previous question, the error limit constraints are also satisfied.
Here is an example output I obtained.

```
inp_fun = lambda x: np.sin(x)
inp_fun_str = "sin(x)"
computeZerosFunction(inp_fun,
inp_fun_str, 0, 10, 1e-3)
```

```
The roots of sin(x) in [0, 10] are:
6.28317487487208
2.635105810456196
4.827128018670653
9.424792369343916
2.1562768386267307e-05
8.890632690307365
8.211697370167169
3.1415858599551947
4.804385452434257
```