ECSE 543 Assignment 3

Garrett Kinman – 260763260

Question 1: You are given a list of measured BH points for M19 steel (Table 1), with which to construct a continuous graph of B versus H.

a) Interpolate the first 6 points using full-domain Lagrange polynomials. Is the result plausible, i.e. do you think it lies close to the true B versus H graph over this range?

A program was written to find the Lagrange coefficients and Lagrange polynomial for a given x and y = f(x). It was created with a symbolic library, *Symbolics.jl*, and its output is shown below in Figure 1. The full code is in the appendix, under *interpolate.jl*.

```
julia> lagrange(B[1:6], H[1:6])
:(function (x,)
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:282 =#
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:283 =#
      (+)((+)((*)(88.65000000000077, x), (*)(-215.20833333333576, (^)(x,
2)), (*)(873.4375000000218, (^)(x, 3)), (*)(-963.5416666666933, (^)(x,
4))), (+)((*)(414.0625000000109, (^)(x, 5))))
  end)
```

Figure 1: Output Lagrange polynomial in prefix notation

In conventional infix notation, this would be approximately the following:

$$H(B) \approx 414.0625x^5 - 963.5417x^4 + 873.4375x^3 - 215.2083x^2 + 88.6500x$$

Below in Figure 2 are the plotted results of the above polynomial for B from 0.0 to 1.0. The code used to generate it is in the appendix in *question1.jl*. The results appear to be plausible over this range, as it produces a smooth-appearing curve that cleanly interpolates the known points.

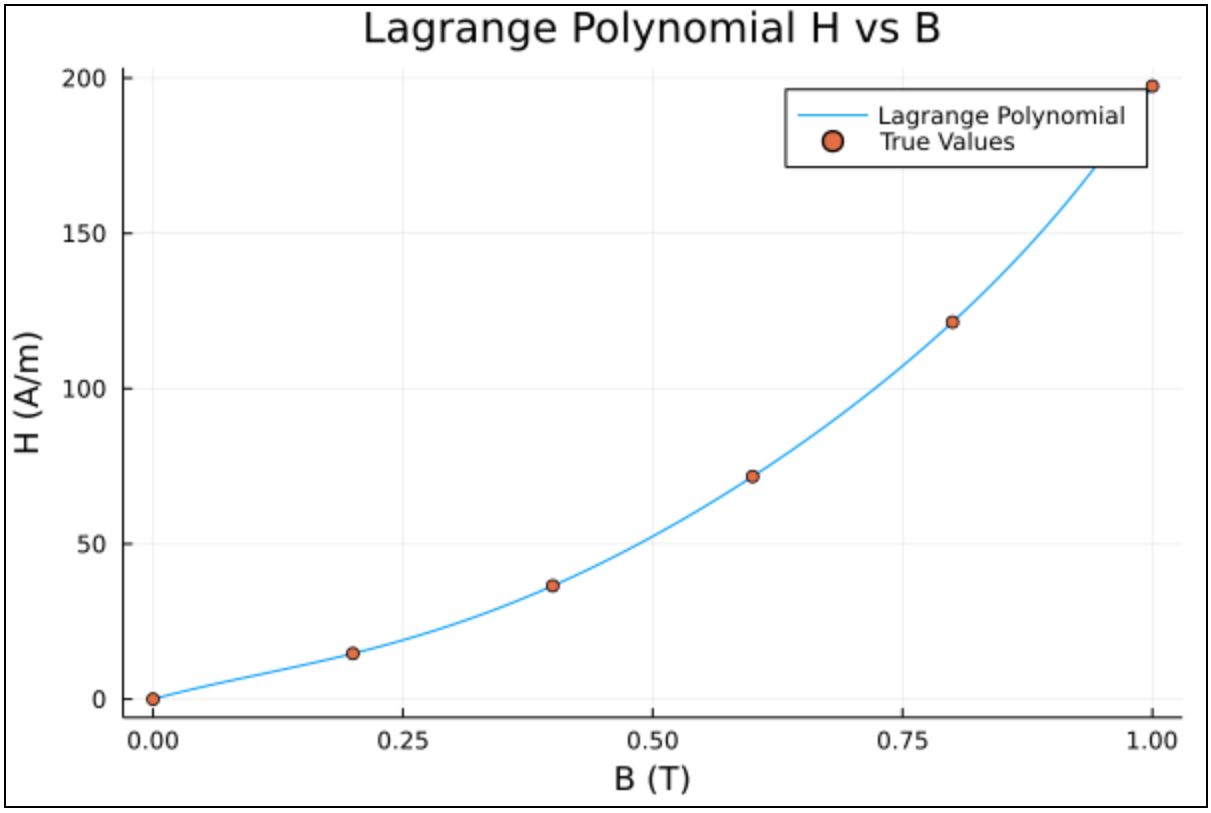

Figure 2: Lagrange polynomial for 0.0 to 1.0 and true B–H values

b) Now use the same type of interpolation for the 6 points at B = 0, 1.3, 1.4, 1.7, 1.8, 1.9. Is this result plausible?

In Figure 3 below is the output for interpolation over B = 0, 1.3–1.9.

```
julia> lagrange([B[1]; B[9:10]; B[13:15]], [H[1]; H[9:10]; H[13:15]])
:(function (x,)
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:282 =#
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:283 =#
      (+)((+)((*)(906781.8544220868, x), (*)(-2.337828829457741e6, (^)(x,
2)), (*)(2.2538202211505845e6, (^)(x, 3)), (*)(-966235.5722451136, (^)(x,
4))), (+)((*)(156393.28052408784, (^)(x, 5))))
  end)
```

Figure 3: Output Lagrange polynomial in prefix notation

In conventional infix notation, this would be approximately the following:

$$H(B) \approx 156393.28x^5 - 966235.57x^4 + 2.25x^3 - 2.34x^2 + 906781.85x$$

Below in Figure 4 are the plotted results of the above polynomial for the above stated values of B, along with the known B–H values. The code used to generate it is in the appendix in

question1.jl. The results appear to be implausible over this range, as it produces a curve that does not accurately interpolate between B = 0.0 and B = 1.0. This is likely because of the relatively large gap between B values, compared to the rest of the given B values.

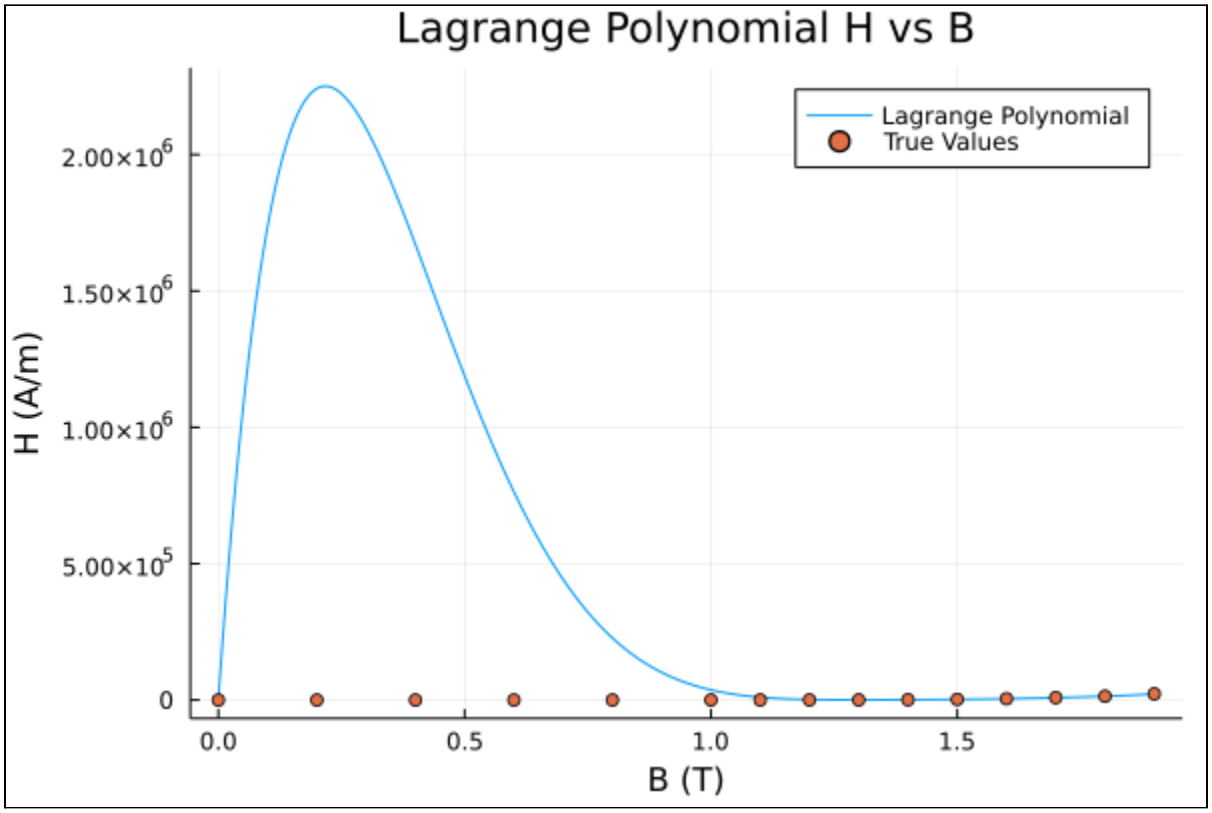

Figure 4: Lagrange polynomial for B = 0, 1.3, 1.4, 1.7, 1.8, 1.9, alongside known B–H values

c) An alternative to full-domain Lagrange polynomials is to interpolate using cubic Hermite polynomials in each of the 5 subdomains between the 6 points given in (b). With this approach, there remain 6 degrees of freedom - the slopes at the 6 points. Suggest ways of fixing the 6 slopes to get a good interpolation of the points.

One method of approximating the 6 slopes is to use the average slope between the j-th and (j + 1)-th points, and for the final point (for which there is no (j + 1)-th point) use y/x. A program was written within *interpolate.jl* and *question1.jl* to calculate the Hermite polynomial and interpolate for this setup. The calculated Hermite polynomial is shown below in Figure 5.

```
julia> hermite(B_sub, H_sub, H_sub')
:(function (x,)
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:282 =#
      #=
C:\Users\Wombat\.julia\packages\SymbolicUtils\2UXNG\src\code.jl:283 =#
```

```
     (+)((+)((*)(415.846153846154, x), (*)(-1.1699512947536526e11, (^)(x,
2)), (*)(6.671464727195928e11, (^)(x, 3)), (*)(-1.6858627196046233e12,
(^)(x, 4))), (+)((*)(2.4778275841595938e12, (^)(x, 5)),
(*)(-2.334363871440526e12, (^)(x, 6)), (*)(1.4618875958639148e12, (^)(x,
7)), (*)(-6.085754438975063e11, (^)(x, 8))), (+)((*)(1.6239943311112059e11,
(^)(x, 9)), (*)(-2.5207926897783447e10, (^)(x, 10)),
(*)(1.734143651253302e9, (^)(x, 11))))
  end)
```

Figure 5: Output Hermite polynomial in prefix notation

In conventional infix notation, this would be approximately the following:

$$H(B) \approx 1.73e9x^{11} - 2.52e10x^{10} + 1.62e11x^{9} - 6.09e11x^{8} + 1.46e12x^{7} - 2.33e12x^{6} + \ldots$$
$$\ldots + 2.48e12x^{5} - 1.69e12x^{4} + 6.67e11x^{3} - 1.17e11x^{2} + 416x$$

Below in Figure 6 are the plotted results of the above polynomial for the above stated values of B, along with the known B–H values. Observe that it still fails to generalize well to the large gap between B = 0.0 and B = 1.3.

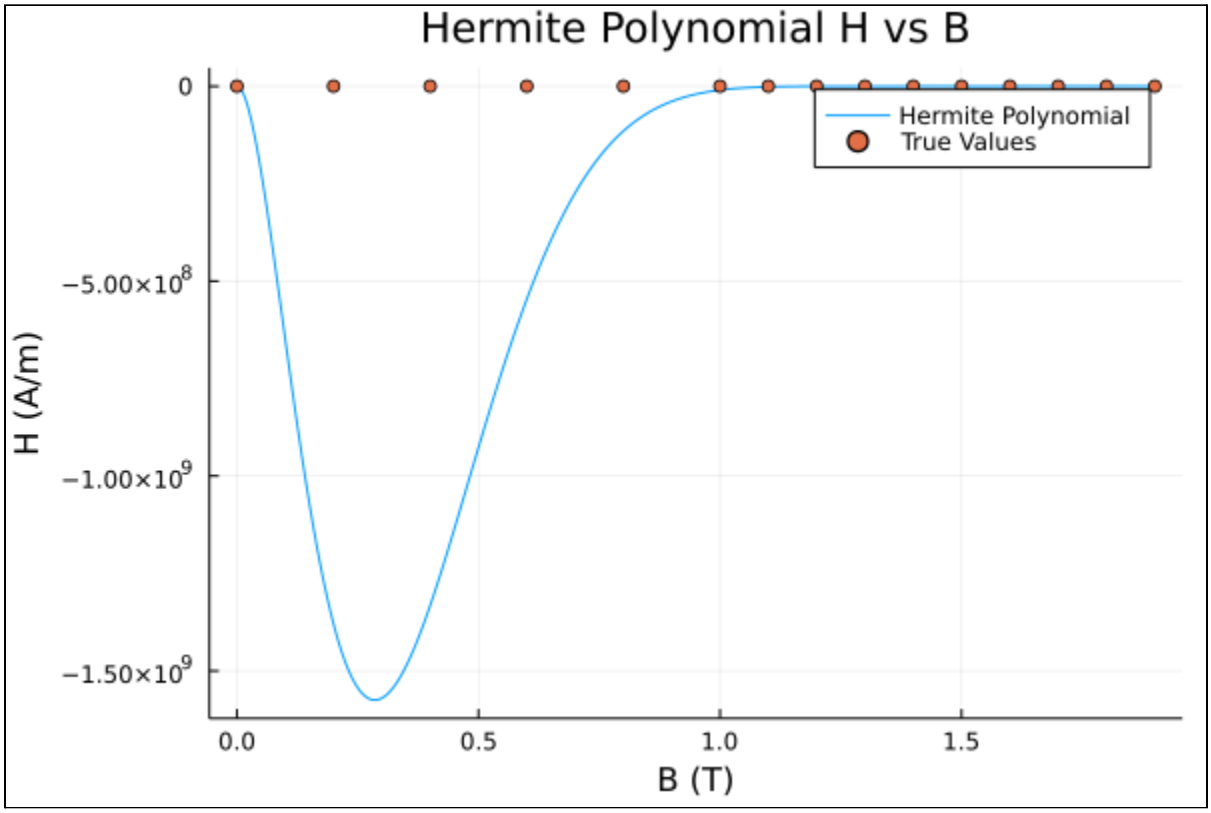

Figure 5: Hermite polynomial for B = 0, 1.3, 1.4, 1.7, 1.8, 1.9, alongside known B–H values

Question 2: The magnetic circuit of Figure 1 has a core made of MI9 steel, with a cross-sectional area 1 cm^2. L_c = 30 cm and L_a = 0.5 cm. The coil has N = 1000 turns and carries a current I = 8 A.

a) Derive a (nonlinear) equation for the flux in the core, of the form f(ψ) = 0.

$$R_G = \frac{l_G}{\mu_0 A_G} = \frac{0.5cm}{4\pi \times 10^{-7} \times 1cm^2} = 39.789 \times 10^6 \Omega$$

$$\mu = \frac{B}{H} = \frac{\varphi}{HA}$$

$$\varphi = A \times B$$

$$R_c = \frac{l_c}{\mu A_c} = \frac{l_c H}{A_c B} = \frac{l_c H}{\varphi}$$

$$NI = \varphi \left(R_G + R_c\right)$$

$$NI = \varphi \left(R_G + \frac{l_c H}{\varphi}\right)$$

$$f\left(\varphi\right) = R_G \varphi + l_c H - NI = 0$$

$$f\left(\varphi\right) = 39.788735 \times 10^{-6} \varphi + 0.3H - 8000 = 0$$

b) Solve the nonlinear equation using Newton-Raphson. Use a piecewise-linear interpolation of the data in Table 1. Start with zero flux and finish when |f(ψ) / f(0)| < 10^{-6}. Record the final flux, and the number of steps taken.

Before programming this problem, a substitution can be made for the above equation: replace H with the piecewise linear interpolation H(B).

$$f\left(\varphi\right) = 39.788735 \times 10^{-6} \varphi + 0.3 \times piecewiseLinear(B) - 8000 = 0$$

$$f\left(\varphi\right) = 39.788735 \times 10^{-6} \varphi + 0.3 \times piecewiseLinear\left(\frac{\varphi}{A}\right) - 8000 = 0$$

$$f\left(\varphi\right) = 39.788735 \times 10^{-6} \varphi + 0.3 \times piecewiseLinear\left(\frac{\varphi}{1cm^2}\right) - 8000 = 0$$

A program was written to create the piecewise linear interpolation, with the assumption that any values beyond B = 1.9 would use the same linear segment as from B = 1.8 to B = 1.9. This proved necessary, as at iteration 1, the estimated flux was 1.9995e-4 Wb, which translates to B = 1.9995, which is outside the range of B in Table 1.

Two other functions, *f(ψ)*, and *newton_raphson(error)* were written using the *Zygote.jl* package for auto-differentiation. Respectively, they represent the above non-linear function for flux, and for performing Newton-Raphson. The printed output is shown below in Figure 6.

```
Iteration: 0, Flux: 0
Iteration: 1, Flux: 0.0001999538356581924
Iteration: 2, Flux: 0.00016892691737673714
Iteration: 3, Flux: 0.00016126937023830623
```

Figure 6: Output of Newton-Raphson approximation of flux

Observe that it took 3 iterations to obtain an estimate, 1.613e-4 Wb, of the flux. All the code can be found in *question2.jl* in the appendix.

c) Try solving the same problem with successive substitution. If the method does not converge, suggest and test a modification of the method that does converge.

A function, *substitution(error)*, was written to perform the substitution method, and its output is shown in Figure 7 below. Notice how it diverges because of the too-large step size.

```
Iteration: 0, Flux: 0
Iteration: 1, Flux: 8000.0
Iteration: 2, Flux: -2.412525821057433e12
Iteration: 3, Flux: 9.652331010572897e19
Iteration: 4, Flux: -2.9108122860143813e28
Iteration: 5, Flux: 1.1645936988122431e36
Iteration: 6, Flux: -3.512015535940894e44
Iteration: 7, Flux: 1.4051305138909462e52
Iteration: 8, Flux: -4.237392147873209e60
Iteration: 9, Flux: 1.6953481399402182e68
Iteration: 10, Flux: -5.112589062066073e76
Iteration: 11, Flux: 2.0455077213004204e84
Iteration: 12, Flux: -6.168550373766294e92
Iteration: 13, Flux: 2.467989753448031e100
Iteration: 14, Flux: -7.442611415030355e108
Iteration: 15, Flux: 2.9777318167502063e116
Iteration: 16, Flux: -8.979818809734305e124
Iteration: 17, Flux: 3.5927567203626126e132
Iteration: 18, Flux: -1.0834523174595855e141
Iteration: 19, Flux: 4.334809729708285e148
Iteration: 20, Flux: -1.3072300778898221e157
Iteration: 21, Flux: 5.230127407813213e164
Iteration: 22, Flux: -1.5772272106507108e173
Iteration: 23, Flux: 6.310365254209157e180
```

```
Iteration: 24, Flux: -1.9029899296936834e189
Iteration: 25, Flux: 7.613716939675813e196
Iteration: 26, Flux: -2.296036137381573e205
Iteration: 27, Flux: 9.186264709295552e212
Iteration: 28, Flux: -2.7702626597770136e221
Iteration: 29, Flux: 1.1083608699648092e229
Iteration: 30, Flux: -3.342436592878166e237
Iteration: 31, Flux: 1.3372832777462527e245
Iteration: 32, Flux: -4.032788132194751e253
Iteration: 33, Flux: 1.6134876405340273e261
Iteration: 34, Flux: -4.865725846175727e269
Iteration: 35, Flux: 1.946739639594927e277
Iteration: 36, Flux: -5.870699683213403e285
Iteration: 37, Flux: 2.3488219736941096e293
Iteration: 38, Flux: -7.083242225323938e301
Iteration: 39, Flux: Inf
Iteration: 40, Flux: NaN
```

Figure 7: Output of the substitution method approximation of the flux

The difference between the substitution method and Newton-Raphson is the latter uses a step size of $f(\psi_n) / f'(\psi_n)$, whereas the former uses $f(\psi_n)$. Dividing by the derivative in effect scales down the step size. Thus, the function for substitution was given an extra argument, *step_divisor*, which just scales down the step size at each iteration. Having tested starting from 1e1 and increasing by one order of magnitude each time (i.e., 1e1, 1e2, 1e3, …), 1e8 was the smallest divisor that converges. Its output is below in Figure 8. Note how it converges in 6 iterations, reaching an estimated flux of 1.162e-4 Wb.

```
Iteration: 0, Flux: 0
Iteration: 1, Flux: 8.0e-5
Iteration: 2, Flux: 0.000127804812
Iteration: 3, Flux: 0.00015545747100923182
Iteration: 4, Flux: 0.00016261491098577287
Iteration: 5, Flux: 0.00016050303452667703
Iteration: 6, Flux: 0.0001617058270487179
```

Figure 8: Output of the substitution method approximation of the flux, but with step size divisor

Question 3: In the circuit shown below, the DC voltage E is 220 mV, the resistance R is 500, the diode A reverse saturation current I_{sA} is 0.6 A, the diode B reverse saturation current I_{sB} is 1.2 A, and assume kT/q to be 25 mV.

a) Derive nonlinear equations for a vector of nodal voltages, v_n, in the form $f(v_n) = 0$. Give f explicitly in terms of the variables I_{sA} , I_{sB} , E, R and v_n.

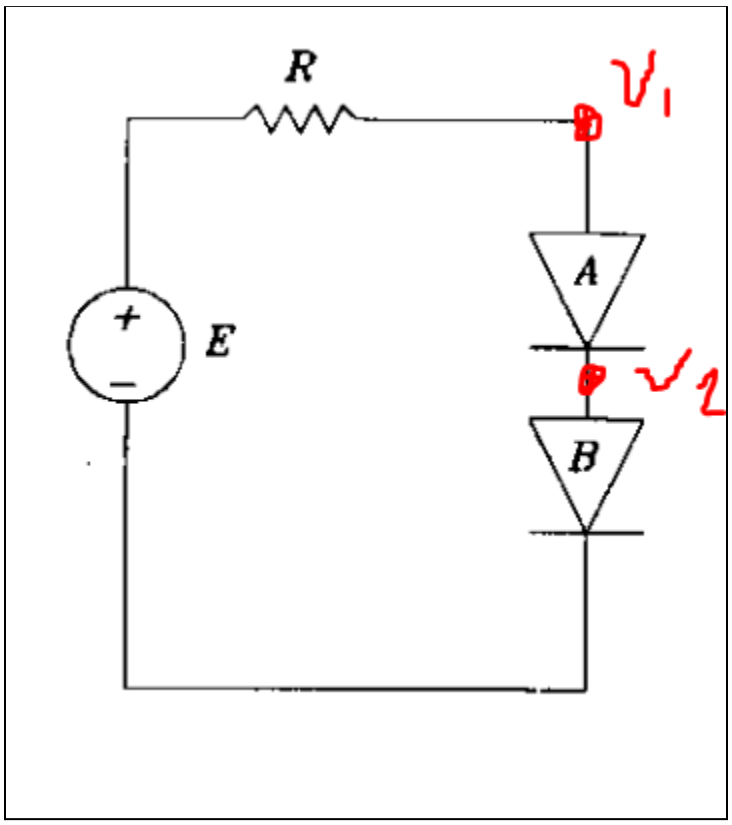

Figure 9: Circuit with node voltages labelled

First, we can define three equations for the current in the mesh:

$$I = \frac{E - v_1}{R} \quad (1)$$

$$I = I_{sA} \times \left(e^{\frac{v_1 - v_2}{v_T}} - 1\right) \quad (2)$$

$$I = I_{sB} \times \left(e^{\frac{v_2}{v_T}} - 1\right) \quad (3)$$

Then:

$$v_T = \frac{kT}{q} = 25mV$$

$$f_1 = (1) - (2) = \frac{E - v_1}{R} - I_{sA} \times \left(e^{\frac{v_1 - v_2}{v_T}} - 1 \right) = 0$$

$$f_2 = (1) - (3) = \frac{E - v_1}{R} - I_{sB} \times \left(e^{\frac{v_2}{v_T}} - 1 \right) = 0$$

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 44.06 \times 10^{-5} - 0.002v_1 - 0.6 \times 10^{-6} e^{40(v_1 - v_2)} \\ 44.12 \times 10^{-5} - 0.002v_1 - 1.2 \times 10^{-6} e^{40v_2} \end{bmatrix} = \mathbf{0}$$

b) Solve the equation f = 0 by the Newton-Raphson method. At each step, record f and the voltage across each diode. Is the convergence quadratic? [Hint: define a suitable error measure k].

In order to solve by the Newton-Raphson method for this vector equation, a matrix of partial derivatives of f_1 and f_2 is needed.

$$\mathbf{f}' = \begin{bmatrix} \frac{\delta f_1}{\delta v_1} & \frac{\delta f_1}{\delta v_2} \\ \frac{\delta f_2}{\delta v_1} & \frac{\delta f_2}{\delta v_2} \end{bmatrix}$$

$$\mathbf{f}' \left(\mathbf{v_{n+1}} - \mathbf{v_n} \right) + \mathbf{v} = \mathbf{0}$$

$$\mathbf{v_{n+1}} = \mathbf{v_n} - \mathbf{f}'^{-1}\mathbf{f}$$

Then, a program was written, as can be found in the appendix under *question3.jl*, to perform Newton-Raphson on this system. A break condition of if the max(abs($\mathbf{v_n}$-$\mathbf{v_{n+1}}$)) < 1e-5 is used. That is, if the maximum absolute value of the element-wise subtraction of the old and new iterated estimates is smaller than 1e-5. Below is the output of the program. Note how it converges in 5 iterations to values of v_1 = 0.198 V and v_2 = 0.0906 V.

```
iteration,v₁,v₂,error
0,0,0,0.21825396825396826
1,0.21825396825396826,0.07275132275132364,0.012558878144285524
2,0.20569509010968273,0.08158102631046119,0.007668710170528517
3,0.20010958229029016,0.0892497364809897,0.0018985204074767614
4,0.1982110618828134,0.09051583274769154,7.692769104608588e-5
5,0.1981341341917673,0.09057062760850919,1.2596266935060996e-7
```

Figure 10: Output of Newton-Raphson approximation of node voltages

Below in Figure 11 is the plot of the error and the nodal voltage estimates converging over iterations. The convergence is indeed quadratic.

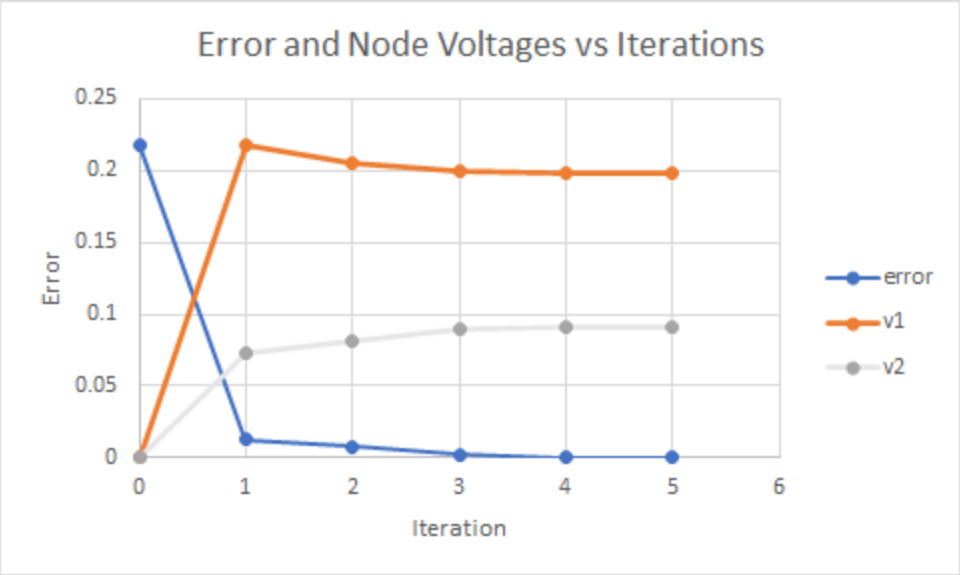

Figure 11: Plot of the convergence of nodal voltages and error

Question 4

a) Integrate the function cos(x) on the interval x=0 to x=1, by dividing the interval into N equal segments and using one-point Gauss-Legendre integration for each segment. Plot $\log_{10}(E)$ versus $\log_{10}(N)$ for N=1, 2,..., 20, where E is the absolute error in the computed integral. Comment on the result.

A program—*question4.jl* in the appendix—was written to complete this. Essentially, it takes in the interval, a function, and N, then it calculates the **w** and **x** vectors from which to perform the Gauss-Legendre integration. It uses the midpoint of each segment as the x_i. Below in Figure 12 is the CSV-formatted raw output for each iteration, including the ground truth, which is calculated using the analytical definite integral of cos(x) for x = 0 to 1.

```
Ground truth ₀∫¹cos(x)dx = 0.8414709848078965
N,Integral,Error
1,0.8775825618903728,0.036111577082476254
2,0.8503006452922328,0.00882966048433631
3,0.8453793458454515,0.003908361037554986
4,0.8436663167025465,0.002195331894650043 3
5,0.8428750743698316,0.001404089561935051 3
6,0.8424456991964263,0.000974714388529807 1
7,0.8421869475034672,0.000715962695570726 6
8,0.8420190672464982,0.000548082438601715 8
9,0.8419039961670827,0.000433011359186163 96
```

```
10,0.8418217000072957,0.0003507151993992208
11,0.8417608174053209,0.0002898325974244331
12,0.8417145153208724,0.0002435305129758758
13,0.8416784838788398,0.0002074990709433066 6
14,0.8416498955690674,0.0001789107611709361 8
15,0.8416268329703337,0.0001558481624371888
16,0.8416079585815617,0.0001369737736651721 6
17,0.8415923163990292,0.0001213315911327006 4
18,0.8415792084113782,0.0001082236034817274 4
19,0.841568115345253,9.713053735649346e-5
20,0.8415586444272832,8.765961938672628e-5
```

Figure 12: Raw output for approximation of the integral using Gauss-Legendre

Below in Figure 13 is the plot of the absolute error and N as a log–log plot. Note that there exists a linear relationship between $\log_{10}(E)$ and $\log_{10}(N)$, meaning there are diminishing returns (in terms of reduced error) to high values of N, and that the relationship between E and N is monomial, i.e., of the form $y = kx^b$. Because b is equivalent to the slope of the plot, we can determine thusly:

$$b = slope = \frac{\log_{10}(E_2) - \log_{10}(E_1)}{\log_{10}(N_2) - \log_{10}(N_1)} = \frac{-4.057 + 1.442}{1.301 - 0} = -2.010$$

$$\Rightarrow E = kN^{-2.010}$$

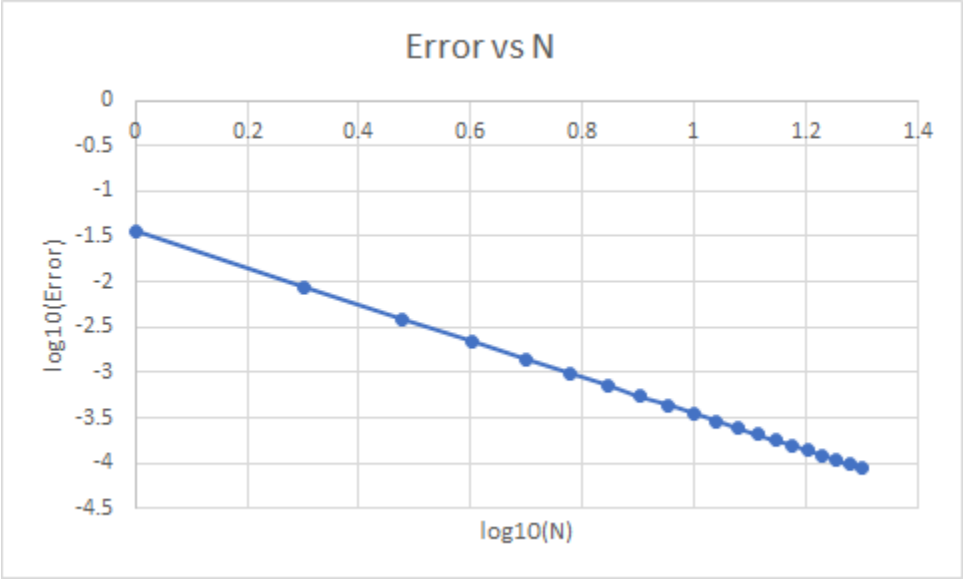

Figure 13: Log–log plot of E (error) vs N

b) Repeat part (a) for the function ln(x), only this time plot for N=10, 20,...200. Comment on the result.

Below in Figure 14 is the raw output for estimating the integral of ln(x) for N=10, 20, …, 200.

```
Ground truth ₀∫¹ln(x)dx = -1.0
N,Integral,Error
10,-0.9657590653461392,0.0342409346538608
20,-0.9827754719736862,0.0172245280263138
30,-0.9884938402873319,0.011506159712668107
40,-0.9913617009604191,0.00863829903958091
50,-0.9930851944722275,0.006914805527772461
60,-0.9942353473818812,0.00576465261811876
70,-0.9950574520104222,0.004942547989577828
80,-0.9956743404788303,0.0043256595211697
90,-0.9961543263261001,0.0038456736738998742
100,-0.9965384307395622,0.0034615692604378356
110,-0.9968527745070249,0.00314722549297508 1
120,-0.9971147802544643,0.002885219745535683
130,-0.9973365147802635,0.0026634852197364722
140,-0.9975266001991562,0.002473399800843823
150,-0.9976913612451843,0.002308638754815684
160,-0.9978355426612082,0.002164457338791781
170,-0.9979627735721441,0.002037226427855865
180,-0.9980758771710263,0.0019241228289736956
190,-0.9981770826716382,0.0018229173283618172
200,-0.9982681737137478,0.001731826286252236
```

Figure 14: Raw output of the integral approximation for ln(x)

Below in Figure 15 is the plot of the absolute error and N as a log–log plot. Note that there exists a linear relationship between $\log_{10}(E)$ and $\log_{10}(N)$, meaning this also has a monomial relationship between E and N. Using the same formula as for Question 4a, we can say the relationship is of the following form:

$$E = kN^{-0.9962}$$

Note that the b value here, approximately -1, is different than for cos(x), for which b was approximately -2. Intuitively, this means doubling N results in halving E for ln(x), and for cos(x), it means doubling N results in quartering E. This indicates there is something about ln(x) over x=(0,1] that requires more segments to achieve a certain level of accuracy. More on this in the next question.

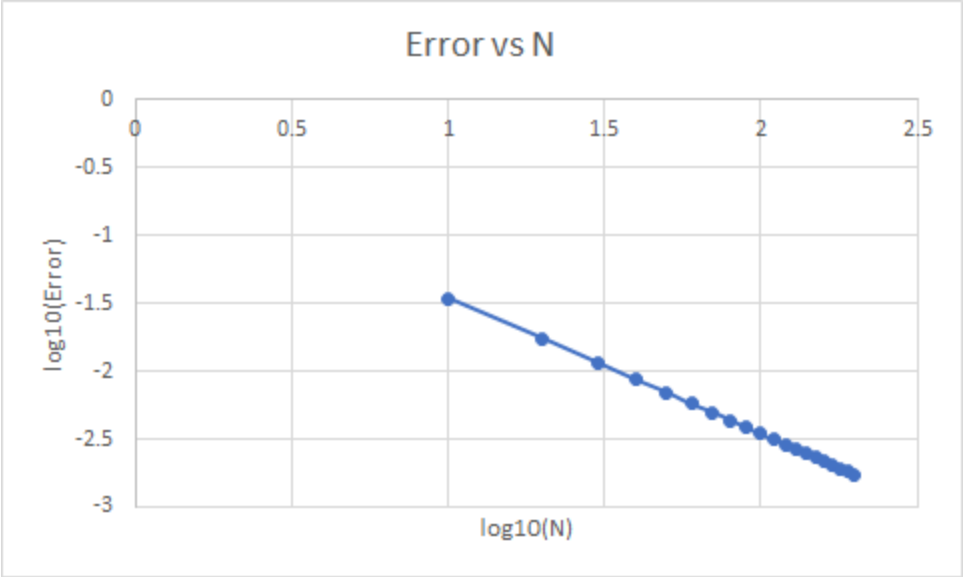

Figure 15: Log–log plot of E (error) vs N

c) An alternative to dividing the interval into equal segments is to use smaller segments in more difficult parts of the interval. Experiment with a scheme of this kind, and see how accurately you can integrate $\log_e(x)$ using only 10 segments.

In order to try to minimize the error using only 10 segments, one strategy is to try to see which portions of the curve are flattest, and give them the widest segments. In Figure 16 below, we can see how, compared to ln(x), cos(x) is relatively flat over the x = 0 to x = 1 interval. This likely explains the resulting b values from above.

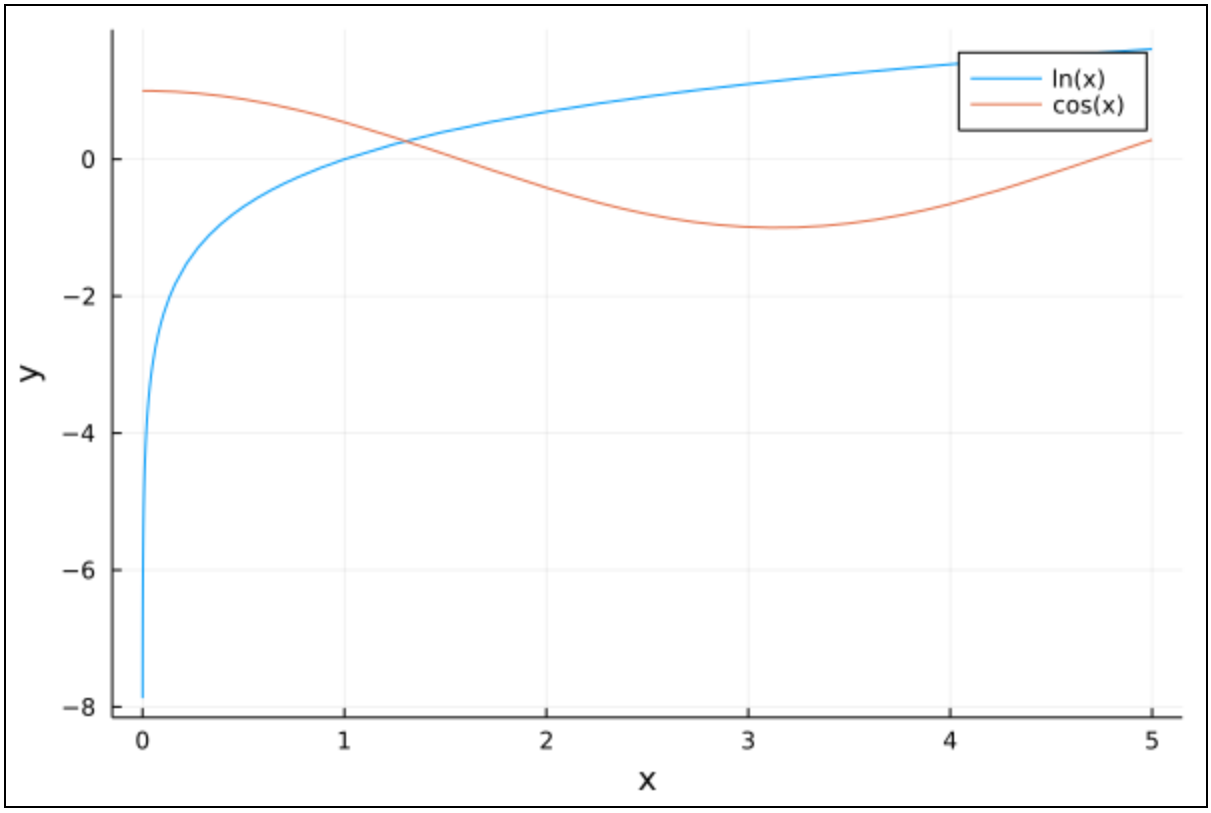

Figure 16: Plot of ln(x) and cos(x)

Regarding ln(x), however, we see the left side, nearest zero, is asymptotic, while the right side, nearest 1, is much more flat. Thus, the strategy will be to try to use more and smaller segments towards the left, and fewer and bigger ones on the right. Specifically, each segment will be scaled up by a certain amount from the preceding one. This will have the net effect of starting with the shortest segments near zero and getting exponentially longer as they go towards x = 1. Naively, we could use 2 as this ratio, but in the spirit of optimization, we will try a range of values from 1 (equivalent to even spacing) to 2. The results are shown below in Figure 17.

```
Ground truth ₀∫¹ln(x)dx = -1.0
Ratio,Integral,Error
1.0,-0.9657590653461391,0.03424093465386091
1.05,-0.9719754442606275,0.02802455573937246
1.1,-0.9769912542509798,0.023008745749020232
1.15,-0.9809310750065623,0.019068924993437708
1.2,-0.9839342905695286,0.016065709430471364
1.25,-0.9861407119946162,0.01385928800538383
1.3,-0.9876810320433765,0.012318967956623461
1.35,-0.9886716478987047,0.011328352101295347
1.4,-0.9892127563552103,0.010787243644789668
1.45,-0.9893885632877434,0.010611436712256617
1.5,-0.9892686631035534,0.010731336896446564
```

```
1.55,-0.9889099284370579,0.011090071562942061
1.6,-0.988358501891527,0.011641498108473036
1.65,-0.9876516668781392,0.012348333121860788
1.7,-0.9868194958070357,0.013180504192964348
1.75,-0.9858862458392097,0.014113754160790304
1.8,-0.9848715109595414,0.015128489040458604
1.85,-0.9837911569585915,0.01620884304140846
1.9,-0.9826580718449365,0.017341928155063524
1.95,-0.9814827638839343,0.018517236116065705
2.0,-0.9802738362302624,0.01972616376973757
```

Figure 17: Raw output of the integral approximation for ln(x) with varying segment width ratios

Note that the best approximation is achieved for a ratio of 1.45, achieving an absolute error of only 0.01061.

Appendix

interpolate.jl

```
using Symbolics

function lagrange(X, Y)
    n = length(X)
    @variables x # x is symbolic

    L = []

    for j ∈ 1:n
        F₁ = 1
        F₂ = 1
        for r ∈ 1:n
            # multiply in all the terms into the numerator and denominator
            if r != j
                F₁ *= (x - X[r])
                F₂ *= (X[j] - X[r])
            end
        end

        # set the j-th Lagrange polynomial term := F (calculated above)
        push!(L, F₁ / F₂)
    end
    # scale each coefficient by the corresponding y value,
    # then sum up the polynomials for the total polynomial expression
```

```
    y = sum(Y .* L)

    return build_function(simplify(y, expand=true), x)
end

function hermite(X, Y, Y′)
    n = length(X)
    @variables x # x is symbolic

    U = []
    V = []

    for j ∈ 1:n
        F₁ = 1
        F₂ = 1
        for r ∈ 1:n
            # multiply in all the terms into the numerator and denominator
            if r != j
                F₁ *= (x - X[r])
                F₂ *= (X[j] - X[r])
            end
        end
        # calculate j-th Lagrange polynomial term, and its derivative
        L = F₁ / F₂
        L′ = Symbolics.derivative(L, x)

        # create a callable version of L′
        L ′ = substitute(L′, Dict(x => X[j]))

        push!(U, (1 - 2 * L ′ * (x - X[j])) * (L^2))
        push!(V, (x - X[j]) * (L^2))

    end

    # scale each coefficient by the corresponding y or y′ value and add
    # then sum up the polynomials for the total polynomial expression
    y = sum((Y .* U) + (Y′ .* V))

    return build_function(simplify(y, expand=true), x)
end
```

question1.jl

```julia
using Plots

include("interpolate.jl")

B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
1.9]
H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8,
2318.0, 4781.9, 8687.4, 13924.3, 22650.2]

## question 1a
@time lagrange_polynomial = lagrange(B[1:6], H[1:6]) |> eval

plot(0.0:0.001:1.0, lagrange_polynomial.(0.0:0.001:1.0), label="Lagrange
Polynomial")
xlabel!("B (T)")
ylabel!("H (A/m)")
title!("Lagrange Polynomial H vs B")
scatter!(B[1:6], H[1:6], label="True Values")
savefig("Assignment 3/question1a.png")

## question 1b
@time lagrange_polynomial2 = lagrange([B[1]; B[9:10]; B[13:15]], [H[1];
H[9:10]; H[13:15]]) |> eval

plot(0.0:0.001:1.9, lagrange_polynomial2.(0.0:0.001:1.9), label="Lagrange
Polynomial")
xlabel!("B (T)")
ylabel!("H (A/m)")
title!("Lagrange Polynomial H vs B")
scatter!(B, H, label="True Values")
savefig("Assignment 3/question1b.png")

## question 1c
B_sub = [B[1]; B[9:10]; B[13:15]]
H_sub = [H[1]; H[9:10]; H[13:15]]

# construct an approximate
H_sub′ = zeros(Float64, length(H_sub))
for j ∈ 1:length(H_sub′)
    if j != length(H_sub′)
        H_sub′[j] = (H_sub[j + 1] - H_sub[j]) / (B_sub[j + 1] - B_sub[j])
    else
        H_sub′[j] = H_sub[j] / B_sub[j]
```

```
    end
end

@time hermite_polynomial = hermite(B_sub, H_sub, H_sub') |> eval

plot(0.0:0.001:1.9, hermite_polynomial.(0.0:0.001:1.9), label="Hermite
Polynomial")
xlabel!("B (T)")
ylabel!("H (A/m)")
title!("Hermite Polynomial H vs B")
scatter!(B, H, label="True Values")
savefig("Assignment 3/question1c.png")

##
```

question2.jl

```
using Zygote

B = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8,
1.9]
H = [0.0, 14.7, 36.5, 71.7, 121.4, 197.4, 256.2, 348.7, 540.6, 1062.8,
2318.0, 4781.9, 8687.4, 13924.3, 22650.2]

## function declarations

function piecewise_linear(x::Real, X::AbstractVector{<:Real},
Y::AbstractVector{<:Real})::Real
    n = length(X)

    for i ∈ 2:n # start at second element
        if x ≤ X[i]
            slope = (Y[i] - Y[i - 1]) / (X[i] - X[i - 1]) # slop of
interpolated segment
            y = slope * (x - X[i]) + Y[i] # slope-intercept form

            return y
        elseif x > X[end] # extend out last segment
            slope = (Y[end] - Y[end - 1]) / (X[end] - X[end - 1]) # slop
of last interpolated segment
            y = slope * (x - X[end]) + Y[end] # slope-intercept form

            return y
```

```
        end
    end
end

f(ϕ::Real)::Real = (39.788735e6 * ϕ) + (0.3 * piecewise_linear(ϕ / 1e-4, B,
H)) - 8_000

function newton_raphson(error::Real)
    # initialize estimate of flux, ϕ, as 0
    ϕ  = 0
    i = 0

    while true
        println("Iteration: $i, Flux: $ϕ ")

        # create new flux estimate
        ϕ ₊₁ = ϕ  - (f(ϕ ) / f'(ϕ ))

        # if error small enough, we're done
        # else, update the flux estimate
        if (abs(ϕ  - ϕ ₊₁) < error)
            break
        else
            i += 1
            ϕ  = ϕ ₊₁
        end
    end
end

function substitution(step_divisor::Real, error::Real)
    # initialize estimate of flux, ϕ, as 0
    ϕ  = 0
    i = 0

    while true
        println("Iteration: $i, Flux: $ϕ ")

        # create new flux estimate
        ϕ ₊₁ = ϕ  - (f(ϕ ) / step_divisor)

        # if error small enough, we're done
        # else, update the flux estimate
        if (abs(ϕ  - ϕ ₊₁) < error)
```

```
            break
        else
            i += 1
            ϕ  = ϕ ₊₁
        end
    end
end

## run newton-raphson method

newton_raphson(1e-6)

## run substitution method

substitution(1e8, 1e-6)
```

question3.jl

```
using Zygote

f₁(v₁::Real, v₂::Real) = 44.06e-5 - (0.002 * v₁) - (6e-7 * exp(40 * (v₁ -
v₂)))
f₂(v₁::Real, v₂::Real) = 44.12e-5 - (0.002 * v₁) - (12e-7 * exp(40 * v₂))

function newton_raphson(error::Real)
    # initialize estimate of node voltages, v, as 0
    v  = [0, 0]
    i = 0

    println("iteration,v₁,v₂,error")

    while true
        # create new node voltage estimates
        # v ₊₁ = v  - (f′⁻¹·f)
        f′ = [
            gradient(x -> f₁(x, v [2]), v [1])[1] gradient(x -> f₁(v [1],
x), v [2])[1]
            gradient(x -> f₂(x, v [2]), v [1])[1] gradient(x -> f₂(v [1],
x), v [2])[1]
        ]

        v ₊₁ = v  - (inv(f′) * [f₁(v ...), f₂(v ...)])
```

```
        ϵ  = max(abs.(v  - v ₊₁)...)

        println("$i,$(v [1]),$(v [2]),$ϵ ")

        # if error small enough, we're done
        # else, update the node voltage estimates
        if (ϵ  < error)
            break
        else
            i += 1
            v  = v ₊₁
        end
    end
end

newton_raphson(1e-5)
```

question4.jl

```
using Plots

function gauss_legendre(func::Function, interval::Tuple{Real, Real},
N::Integer; even_spacing=true, width_ratio=1)
    integral = 0

    if even_spacing
        segment_width = (interval[2] - interval[1]) / N

        # each element, xᵢ, of x will hold the midpoint of the i-th segment
        # e.g., a segment from x = 0.1 to 0.2 will be stored as 0.1
        x = zeros(N)
        for i ∈ 1:N
            x[i] = (segment_width * i) - (segment_width / 2)
        end

        # each element, wᵢ, of w will hold the width of the i-th segment
        w = ones(N) * segment_width

        integral = sum(w .* func.(x))
    else
        w_relative = [width_ratio^i for i ∈ 1:N]
        w = ((interval[2] - interval[1]) / sum(w_relative)) * w_relative
```

```
        x = zeros(N)
        for i ∈ 1:N
            x[i] = i == 1 ? w[i] / 2 : x[i - 1] + (w[i - 1] / 2) + w[i] /
2
        end

        integral = sum(w .* func.(x))
    end

    return integral
end

## question 4a

ground_truth = sin(1) - sin(0)
println("Ground truth ₀∫¹cos(x)dx = $ground_truth")

println("N,Integral,Error")
for N ∈ 1:20
    integral = gauss_legendre(cos, (0, 1), N)
    error = integral - ground_truth
    println("$N,$integral,$error")
end

## question 4b

ground_truth = (1*log(1) - 1)
println("Ground truth ₀∫¹ln(x)dx = $ground_truth")

println("N,Integral,Error")
for N ∈ 10:10:200
    integral = gauss_legendre(log, (0, 1), N)
    error = integral - ground_truth
    println("$N,$integral,$error")
end

## question 4c

plot(log, label="ln(x)")
plot!(cos, label="cos(x)")
xlabel!("x")
ylabel!("y")
savefig("Assignment 3/question4c.png")
```

```
ground_truth = (1*log(1) - 1)
println("Ground truth ₀∫¹ln(x)dx = $ground_truth")

println("Ratio,Integral,Error")
for r ∈ 1:0.05:2
    integral = gauss_legendre(log, (0, 1), 10, even_spacing=false,
width_ratio=r)
    error = integral - ground_truth
    println("$r,$integral,$error")
end
```