ECSE 543 A3

Q1

a)

The implementation of Lagrange polynomial interpolation is attached in the Appendix. Figure 1 shows the interpolation with the first six points, and the result is not plausible. The points after the first six ones do not lie closely to the curve.

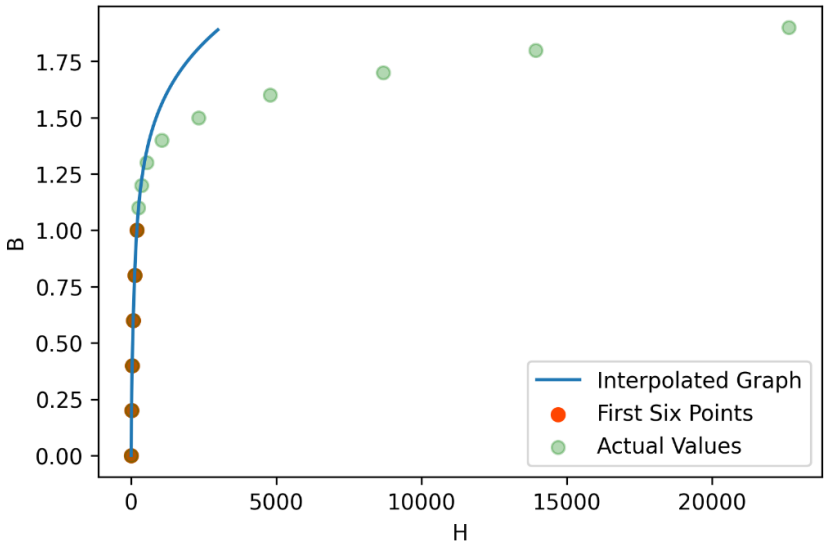


Figure 1 - Q1 a)

b)

Figure 2 shows the interpolation with the selected six points, and the result is still not plausible.

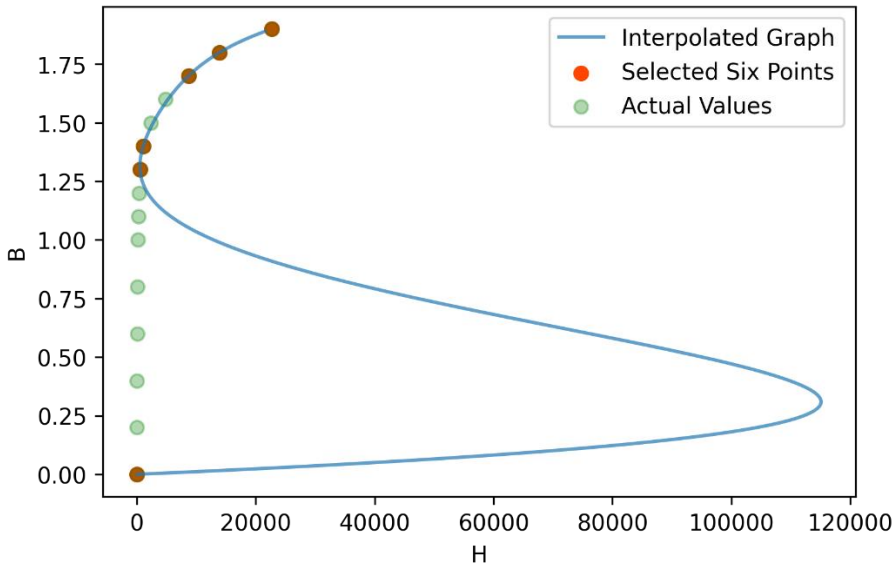


Figure 2 - Q1 b)

c)

The implementation of cubic Hermite polynomial interpolation is attached in the Appendix. To make the curve smoother between subdomains, the slopes at the subdomain connection points are forced to be continuous, which is shown in Figure 3. The result of the interpolation is shown in Figure 4, which lies closely to all the given data points, thus it is a plausible interpolation.

```
var b1 = 0.0
if sub_domain_index != 1 {
    b1 = (y_vector[sub_domain_index-1]-y_vector[sub_domain_index-2]) / (x_vector[sub_domain_index-1]-x_vector[sub_domain_index-2])
}
let b2 = (y_vector[sub_domain_index]-y_vector[sub_domain_index-1]) / (x_vector[sub_domain_index]-x_vector[sub_domain_index-1])
```

Figure 3 - Smooth Slope

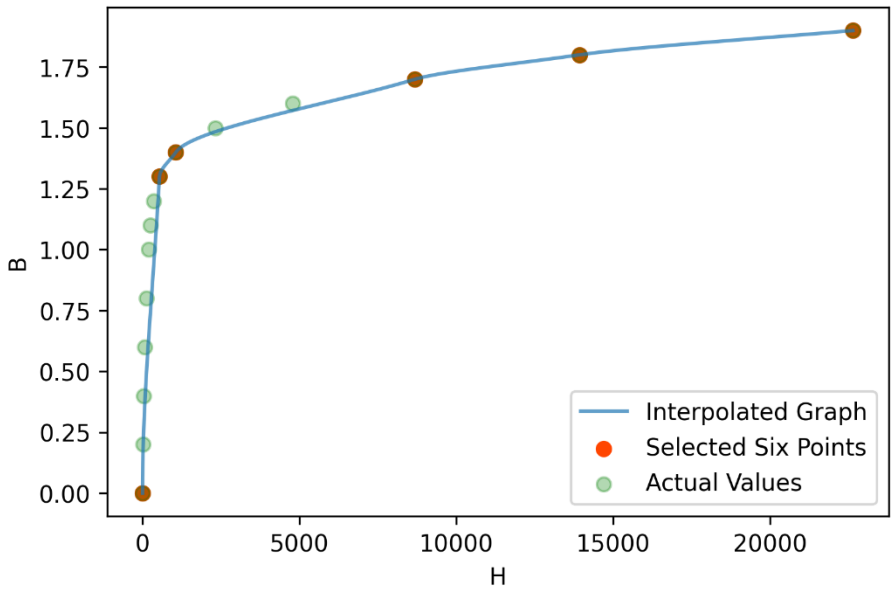
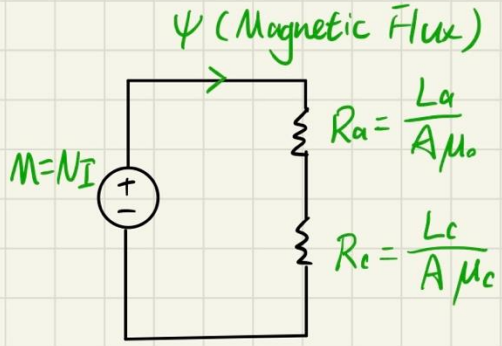


Figure 4 - Q1 c)

d)



$$\begin{cases} \mu_0 = 4\pi\times10^{-7} \\ A = 0.0001 \\ L_a = 0.005 \\ L_c = 0.3 \\ N = 800 \\ I = 10 \end{cases}$$

$$(R_a + R_c)\Psi = M$$

$$\left.\begin{cases} \Psi = B\cdot A \\ \mu_c = \frac{B}{H} \end{cases}\right\} \rightarrow \mu_c = \frac{\Psi}{HA}$$

$$\begin{cases} R_a = \frac{L_a}{A\mu_0} = \frac{0.005}{10^{-4}\times4\pi\times10^{-7}} = 3.97887\times10^7\ H^{-1} \\ R_c = \frac{L_c}{A\mu_c} = \frac{0.3}{A\times\frac{\Psi}{HA}} = \frac{0.3H}{\Psi} \end{cases}$$

$$3.97887\times10^7\Psi + 0.3H(\Psi) = 800\times10$$

$$f(\Psi) = 3.97887\times10^7\Psi + 0.3H(\Psi) - 8000 = 0$$

e)

Newton-Raphson method is implemented and is attached in the Appendix. Since only B is given, but the actual required values are flux, where flux = B*A, A = 1cm^2; thus, the B vector is multiplied by 1e-4, to convert it to flux, which is shown in Figure 5.

The flux result (1.6127e-4 Wb) and the number of iterations is shown in Figure 6.

```
// ============== A3 Q1 def testing ==============
let B = [0.0e-4,0.2e-4,0.4e-4,0.6e-4,0.8e-4,1.0e-4,1.1e-4,1.2e-4,1.3e-4,1.4e-4,1.5e-4,1.6e-4,1.7e-4,1.8e-4,1.9e-4]
let 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]
var (answer,iteration) = Newton_Raphson(guess: 0.0, threshold: 1e-6, X: B, Y: H)
print("Newton Raphson: ")
print(answer)
print("Iteration: ")
print(iteration)
```

Figure 5 - Update Flux

```
alfred@DESKTOP-PR9OIDO:/mnt/e/Github Projects/ECSE-543/A3$ swift A3P1.swift
Newton Raphson:
0.00016126943763218496
Iteration:
44
```

Figure 6 - Newton Raphson Results

f)

The Successive Substitution is implemented, as shown in Figure 7. This method initially did not converge. After reducing the step size of value updating, as shown in Figure 7, this method finally converges. The result is shown in Figure 8, which is close to the result of the Newton-Raphson method.

```
func Successive_Sub(guess: Double, threshold: Double, X: [Double], Y: [Double]) -> (Double,Int){
    let f_0 = Evaluate_Nonlinear_Function(input: 0.0, X: X, Y: Y)
    var output = guess
    var iteration = 0
    while(abs(Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/f_0) > threshold){
        output = output - Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/(3.97887*1e7) * 1e-3 // smaller step size
        iteration += 1
    }
    return (output,iteration)
}
```

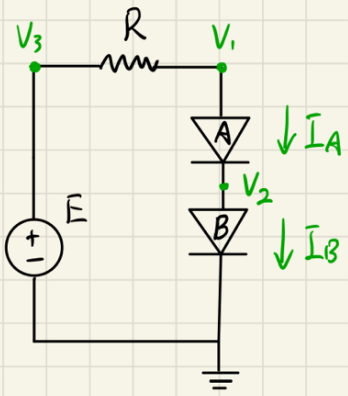
Figure 7 - Successive Substitution with Smaller Step

```
alfred@DESKTOP-PR9OIDO:/mnt/e/Github Projects/ECSE-543/A3$ swift A3P1.swift
Successive Substitution:
0.00016126935538851 31
```

Figure 8 - Successive Substitution Result

Q2

a)



$$\begin{cases} I_{SA} = 0.6\mu \\ I_{SB} = 1.2\mu \\ V_t = \frac{kT}{q} = 25m \\ E = 220m \\ R = 500 \end{cases} \quad \because V_3 = E$$

Diode:

$$I_D = I_S\left(\exp((V_p - V_n)/V_t) - 1\right)$$

$$\begin{cases} I_A = I_{SA}\left(\exp((V_1 - V_2)/V_t) - 1\right) \\ I_B = I_{SB}\left(\exp((V_2 - 0)/V_t) - 1\right) \end{cases}$$

At Node 2:

$$\frac{V_1 - V_3}{R} + I_A = 0$$

At Node 3:

$$-I_A + I_B = 0$$

$$\begin{cases} \frac{1}{R}(V_1 - E) + I_{SA}\left(\exp((V_1 - V_2)/V_t) - 1\right) = 0 \\ I_{SB}\left(\exp(V_2/V_t) - 1\right) - I_{SA}\left(\exp((V_1 - V_2)/V_t) - 1\right) = 0 \end{cases}$$

$$f(\bar{V}_n) = \begin{bmatrix} \frac{1}{R}(V_1 - E) + I_{SA}\left(\exp((V_1 - V_2)/V_t) - 1\right) \\ I_{SB}\left(\exp(V_2/V_t) - 1\right) - I_{SA}\left(\exp((V_1 - V_2)/V_t) - 1\right) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

b)

The Newton-Raphson method for Q2 is implemented and is attached in the Appendix. The final node voltages and the values of **f** and node voltages at each iteration is shown in Figure 9. To show the rate of convergence, delta_x is also recorded and plotted, as shown in Figure 10. Although there is not a clear trend of convergence being quadratic (took only 5 iterations to converge), the rate of convergence can be concluded to be fast (only 5 iterations).

```
alfred@DESKTOP-PR9OIDO:/mnt/e/Github Projects/ECSE-543/A3$ swift A3P1.swift
Final Converged Node Voltages: [V1, V2]
[0.1821396235002092, 0.08719379125430769]
Delta X:
[0.21496650016275412, 0.014010859308527347, 0.003322249594506977, 0.0001879705062550195
5, 5.380847820727837e-07]
f(Vn) Values: ([f1(Vn), f2(Vn)])
[[-0.000390625, 0.0], [7.417230577020583e-05, -4.8001808035091816e-05], [1.155913612879900
31e-05, -1.1538035340957723e-05], [6.916518909560286e-07, -4.869660430844707e-07], [1.837
1177924996884e-09, -1.627918842005767e-09]]
Vn Values: ([V1, V2])
[[0.19812073101961686, 0.08341925516615446], [0.1841399544148662, 0.08433689566842323], [
0.18230748657784016, 0.08710806942596115], [0.18214000800517066, 0.08719341483543555], [0
.1821396235002092, 0.08719379125430769]]
```

Figure 9 - Newton Raphson Results for Q2

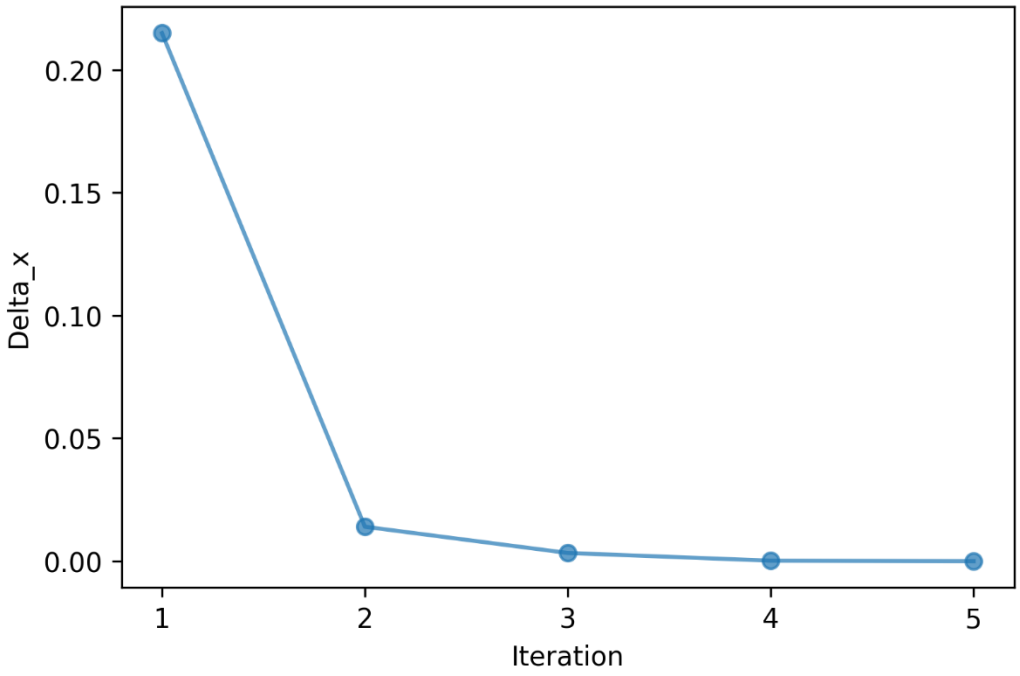


Figure 10 - Rate of Convergence

Appendix

Lagrange Polynomial Interpolation

```
// ================= A3 Q1 ab ============================

func Lagrange_L(current_order: Int, x_vector: [Double], x_value: Double) -> Double{
    var numerator_L = 1.0;
    var denominator_L = 1.0;
    for i in 0...x_vector.count-1{
        if i != current_order {
            numerator_L *= x_value - x_vector[i]
            denominator_L *= x_vector[current_order] - x_vector[i]
        }
    }
    return numerator_L/denominator_L;
}

func Lagrange_interpolate(x_vector: [Double], y_vector: [Double], x_value: Double) -> Double{
    var y_value = 0.0
    for i in 0...x_vector.count-1{
        y_value += y_vector[i] * Lagrange_L(current_order: i, x_vector: x_vector, x_value: x_value)
    }
    return y_value
}
```

Cubic Hermite Interpolation

```
// =============== A3 Q1 c ============================
func Cubic_Hermite_interpolate(x_vector: [Double], y_vector: [Double], x_value: Double) -> Double{
    // find subdomain
    var sub_domain_index = 0;
    while(true){
        if x_value == x_vector[sub_domain_index]{
            return y_vector[sub_domain_index]
        }
        if x_value < x_vector[sub_domain_index] {
            break;
        }else{
            sub_domain_index += 1
            if sub_domain_index > 5{
                sub_domain_index = 5
                break
            }
        }
    }
    let L1 = (x_value-x_vector[sub_domain_index])/(x_vector[sub_domain_index-1] - x_vector[sub_domain_index])
    let L2 = (x_value-x_vector[sub_domain_index-1])/(x_vector[sub_domain_index] - x_vector[sub_domain_index-1])
    let L1_d = 1/(x_vector[sub_domain_index-1] - x_vector[sub_domain_index])
    let L2_d = 1/(x_vector[sub_domain_index] - x_vector[sub_domain_index-1])

    let U1 = (1-2*L1_d*(x_value-x_vector[sub_domain_index-1])) * (L1 * L1)
    let U2 = (1-2*L2_d*(x_value-x_vector[sub_domain_index])) * (L2 * L2)
    let V1 = (x_value-x_vector[sub_domain_index-1]) * (L1 * L1)
    let V2 = (x_value-x_vector[sub_domain_index]) * (L2 * L2)

    var b1 = 0.0
    if sub_domain_index != 1 {
        b1 = (y_vector[sub_domain_index-1]-y_vector[sub_domain_index-2]) / (x_vector[sub_domain_index-1]-x_vector[sub_domain_index-2])
    }
    let b2 = (y_vector[sub_domain_index]-y_vector[sub_domain_index-1]) / (x_vector[sub_domain_index]-x_vector[sub_domain_index-1])

    return y_vector[sub_domain_index-1] * U1 + b1 * V1 + y_vector[sub_domain_index] * U2 + b2 * V2
}
```

Newton Raphson (with Piecewise Interpolation)

```
// =============== A3 Q1 d e f ====================

func Newton_Raphson(guess: Double, threshold: Double, X: [Double], Y: [Double]) -> (Double,Int){
    var output = guess
    let f_0 = Evaluate_Nonlinear_Function(input: 0.0, X: X, Y: Y)
    var iteration = 0
    while (abs(Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/f_0) > threshold){
        output -= Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/Nonlinear_Derivative(input: output, X: X, Y: Y)
        iteration += 1
    }
    return (output,iteration)
}

func Evaluate_Nonlinear_Function(input: Double, X: [Double], Y: [Double]) -> Double{
    return 3.97887 * 1e7 * input + 0.3 * Piecewise(input: input, X: X, Y: Y) - 8000
}

func Evaluate_Nonlinear_Function_Derivative(input: Double, X: [Double], Y: [Double]) -> Double{
    return 3.97887 * 1e7 + 0.3 * Nonlinear_Derivative(input: input, X: X, Y: Y)
}

func Nonlinear_Derivative(input: Double, X: [Double], Y: [Double]) -> Double{
    var index = X.count-1
    for i in 1...X.count-1 {
        if input < X[i]{
            index = i
            break
        }
    }
    return (Y[index]-Y[index-1])/(X[index]-X[index-1])
}

func Piecewise(input: Double, X: [Double], Y: [Double]) -> Double{
    var index = X.count-1
    for i in 1...X.count-1 {
        if input < X[i]{
            index = i-1
            break
        }
    }
    let slope = Nonlinear_Derivative(input: input, X: X, Y: Y)
    return Y[index] + (input-X[index]) * slope
}
```

Successive Substitution

```
func Piecewise(input: Double, X: [Double], Y: [Double]) -> Double{
    var index = X.count-1
    for i in 1...X.count-1 {
        if input < X[i]{
            index = i-1
            break
        }
    }
    let slope = Nonlinear_Derivative(input: input, X: X, Y: Y)
    return Y[index] + (input-X[index]) * slope
}

func Successive_Sub(guess: Double, threshold: Double, X: [Double], Y: [Double]) -> (Double,Int){
    let f_0 = Evaluate_Nonlinear_Function(input: 0.0, X: X, Y: Y)
    var output = guess
    var iteration = 0
    while(abs(Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/f_0) > threshold){
        output = output - Evaluate_Nonlinear_Function(input: output, X: X, Y: Y)/(3.97887*1e7) * 1e-3 // smaller step size
        iteration += 1
    }
    return (output,iteration)
}
```

Newton Raphson for Q2 (with Jacobian)

```
func Q2_Jaccobian(X: [Double]) -> [[Double]]{
    var output = Array(repeating: Array(repeating: 0.0, count: 2), count: 2)
    output[0][0] = 1/512 + (1/(25e-3))*0.8e-6 * exp((X[0]-X[1])/(25e-3))
    output[0][1] = (-1)*(1/(25e-3))*0.8e-6 * exp((X[0]-X[1])/(25e-3))
    output[1][0] = (-1) * (1/(25e-3))*0.8e-6 * exp((X[0]-X[1])/(25e-3))
    output[1][1] = (1/(25e-3))*1.1e-6 * exp((X[1])/(25e-3)) + (1/(25e-3))*0.8e-6 * exp((X[0]-X[1])/(25e-3))
    return output
}

func Q2_F_Eva(X: [Double]) -> [Double]{
    var output = [0.0, 0.0]
    output[0] = (X[0]-0.2)/512 + 0.8e-6 * (exp((X[0]-X[1])/(25e-3))-1)
    output[1] = 1.1e-6 * (exp((X[1])/(25e-3))-1) - 0.8e-6 * (exp((X[0]-X[1])/(25e-3))-1)

    return output
}

func Q2_NR(guess: [Double], threshold: Double) -> ([Double],[[Double]],[[Double]]){
    var X = guess
    var f_value = [0.0,0.0]
    var f_d = [[0.0,0.0],[0.0,0.0]]
    var delta_x = [0.0,0.0]
    var f_set = [[Double]]()
    var x_set = [[Double]]()
    while(true){
        f_value = Q2_F_Eva(X: X)
        f_set.append(f_value)
        f_d = Q2_Jaccobian(X: X)
        delta_x = choleskySolver(A: f_d,b: f_value)!
        X = subtract(A: X, B: delta_x)
        x_set.append(X)
        if two_norm(A: delta_x) < threshold{
            return (X, f_set, x_set)
        }
    }
}
```

Testing

```
// ============= A3 Q1 abc testing =================

var first_six_x = [0.0,0.2,0.4,0.6,0.8,1.0];
var first_six_y = [0.0,14.7,36.5,71.7,121.4,197.4]

var six_x = [0.0, 1.3, 1.4, 1.7, 1.8, 1.9]
var six_y = [0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2]

var continuous_x = [0.00]
for i in 1...190 {
    continuous_x.append(0.01 * Double(i))
}

var y_Lagrange_first_six = [Double]()
var y_Lagrange_six = [Double]()
var y_Cubic_Hermite_six = [Double]()
var counter = 0
for element in continuous_x{
    y_Lagrange_first_six.append(Lagrange_interpolate(x_vector: first_six_x, y_vector: first_six_y, x_value: element))
    y_Lagrange_six.append(Lagrange_interpolate(x_vector: six_x, y_vector: six_y, x_value: element))
    y_Cubic_Hermite_six.append(Cubic_Hermite_interpolate(x_vector: six_x, y_vector: six_y, x_value: element))
}

print(y_Lagrange_first_six)
print(continuous_x)
print(y_Lagrange_six)
print(y_Cubic_Hermite_six)

// ============== A3 Q1 def testing ==============
let B = [0.0e-4,0.2e-4,0.4e-4,0.6e-4,0.8e-4,1.0e-4,1.1e-4,1.2e-4,1.3e-4,1.4e-4,1.5e-4,1.6e-4,1.7e-4,1.8e-4,1.9e-4]
let 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]
var (answer,iteration) = Newton_Raphson(guess: 0.0, threshold: 1e-6, X: B, Y: H)
print("Newton Raphson: ")
print(answer)
print("Iteration: ")
print(iteration)
print("Successive Substitution: ")
var (answer_2,iteration_2) = Successive_Sub(guess: 0.0, threshold: 1e-6, X: B, Y: H)
print(answer_2)



// ============== A3 Q2 testing ==================
var (answer,delta,f_values, x_values) = Q2_NR(guess: [0.0,0.0],threshold: 1e-6)
print("Final Converged Node Voltages: [V1, V2]")
print(answer)
print("Delta X: ")
print(delta)
print("f(Vn) Values: ([f1(Vn), f2(Vn)])")
print(f_values)
print("Vn Values: ([V1, V2])")
print(x_values)
```