Numerical Methods

ECSE 543 - Assignment 3

Mido Assran - 260505216

December 12, 2016

Question 1

Part a

We interpolate the first six points of the BH data, denoted $x_1, x_2, \ldots, x_6$, using full-domain Lagrange Polynomials, each defined over the domain D, where

$$D = [x_1, x_6]$$

That is, we estimate the function being interpolated, $y(x)$, by six polynomials, each of degree five and defined over the entire domain D. The mathematical representation is

$$y(x) = \sum_{j=1}^{6} a_j L_j(x), \quad x \in D \tag{1}$$

where $L_j(x)$ is a fifth degree Lagrange Polynomial, and a_j is a model parameter - scalar. The class *LagrangeInterpolator* is used to perform the interpolation. The class is initialized with the domain points and corresponding range points - the six B and H data samples in this case. Subsequently, the initializer creates the Lagrange Polynomials corresponding to each of the six data points, and determines the associated model parameters as well.

The model parameters the set of a_j, are expected to simply be equal to the corresponding range points, the set of y_j. Nonetheless, we determine these model parameters by solving a least squares problem minimizing the error in our Lagrange Polynomial representation using a 2-norm metric relative to the target data points (the set of $\{y(x)\}$). The least squares problem is modified so that the matrix is positive definite, and then is subsequently solved using our previously created Choleski Decomposition. The implementation is provided in the listing. As expected, the result is simply that the the model parameters, the set of a_j, are simply equal to the corresponding range points, the set of y_j.

The Lagrange Polynomials are represented by the *LagrangePolynomial* class contained in the *polynomial_collective.py* file. The *LagrangePolynomial* class takes as input the set of domain points $\{x\}$, and the subscript index j. Upon initialization, the *LagrangePolynomial* class creates the following polynomial:

$$L_j(x) = \frac{F_j(x)}{F_j(x_j)}$$

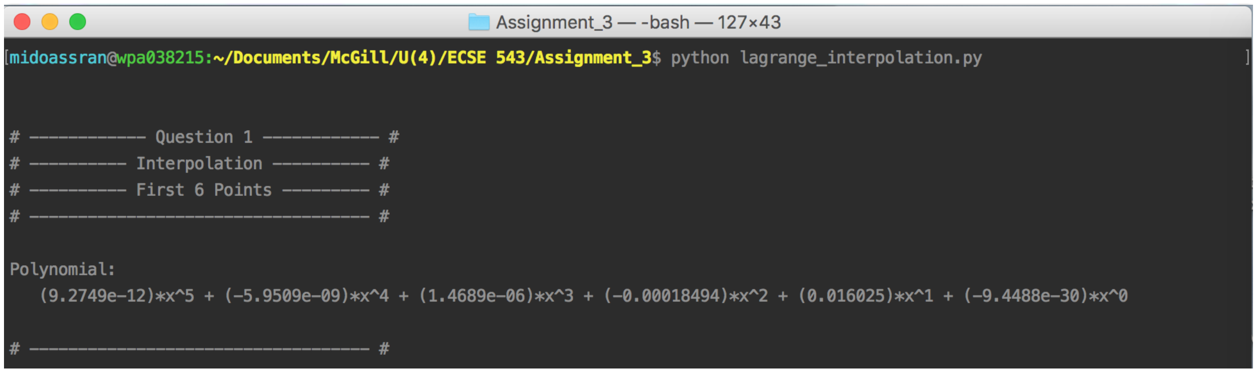
where

$$F_j(x) = \Pi_{r=1:6,\ r\neq j}\ (x - x_r)$$

This polynomial is created by performing a series of binomial multiplications and a scalar division. The binomial operations are performed using the *Polynomial* class represented in the *polynomial.py* file. Once the *LagrangePolynomial* is initialized, it can be evaluated by simply calling its instance method $evaluate(x)$, which takes as input a domain point at which to evaluate the Lagrange Polynomial, and returns the resulting scalar.

*Note that the implementation works with any number of domain points, but in our mathematical representations we show only six points since those are what are used in this subsection of the assignment.

Now back to the *LagrangeInterpolator* class. After it determines its model parameters, $\{a_j\}$, and the corresponding Lagrange Polynomials, $\{L_j\}$, it can be used to interpolate the data simply by calling its $interpolate()$ instance method. This method returns a Python lambda, i.e a functional method, which evaluates equation (1). Figures 1 and 2 show the corresponding interpolations of B vs H and H vs B respectively. The console outputs in the Figures also showcase the closed form expanded Lagrange Polynomial for each the B vs H and the H vs B interpolations. **This result is certainly plausible, and could lie close to the true B vs H curve over this range.**



```
Assignment_3 — -bash — 127×43
[midoassran@wpa038215:~/Documents/McGill/U(4)/ECSE 543/Assignment_3$ python lagrange_interpolation.py


# ------------ Question 1 ------------ #
# ---------- Interpolation ---------- #
# ---------- First 6 Points ---------- #
# ----------------------------------- #

Polynomial:
   (9.2749e-12)*x^5 + (-5.9509e-09)*x^4 + (1.4689e-06)*x^3 + (-0.00018494)*x^2 + (0.016025)*x^1 + (-9.4488e-30)*x^0

# ----------------------------------- #
```

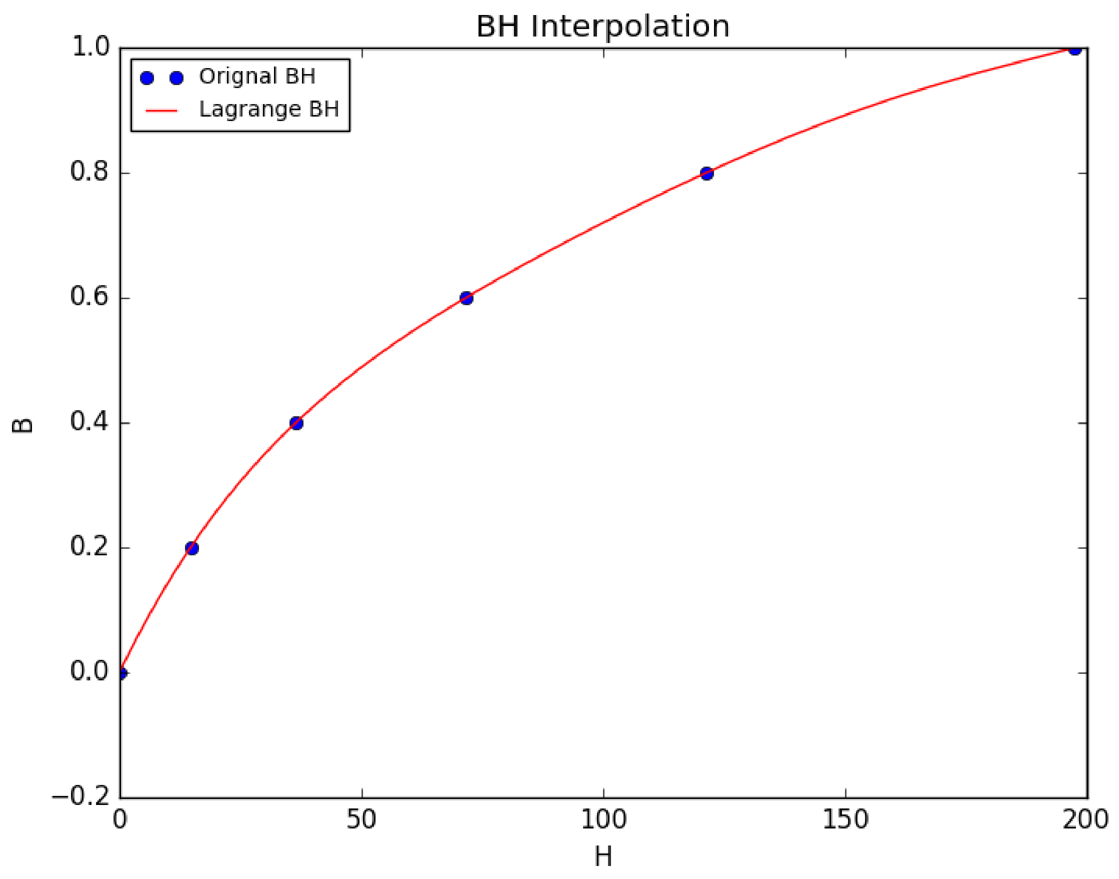
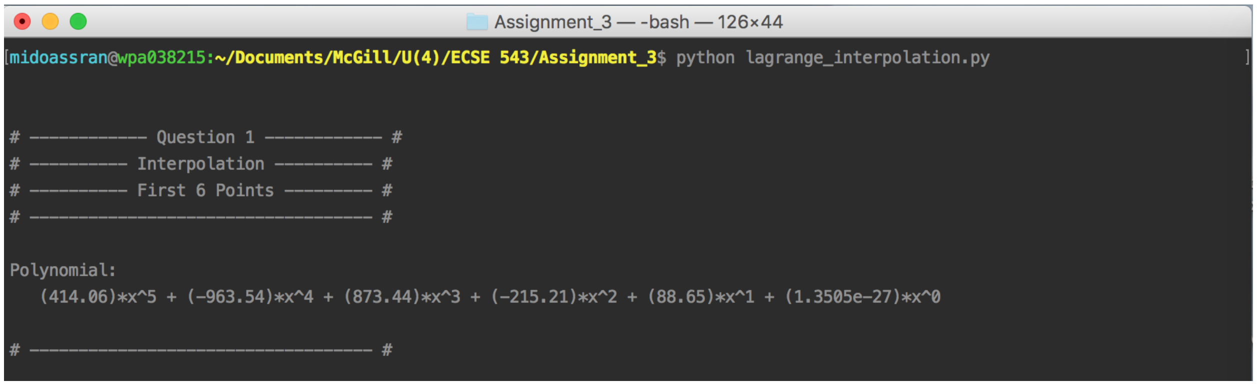


Figure 1: B vs H interpolation using first six points



```
[midoassran@wpa038215:~/Documents/McGill/U(4)/ECSE 543/Assignment_3$ python lagrange_interpolation.py


# ------------ Question 1 ------------ #
# ---------- Interpolation ---------- #
# ---------- First 6 Points --------- #
# ----------------------------------- #

Polynomial:
   (414.06)*x^5 + (-963.54)*x^4 + (873.44)*x^3 + (-215.21)*x^2 + (88.65)*x^1 + (1.3505e-27)*x^0

# ----------------------------------- #
```

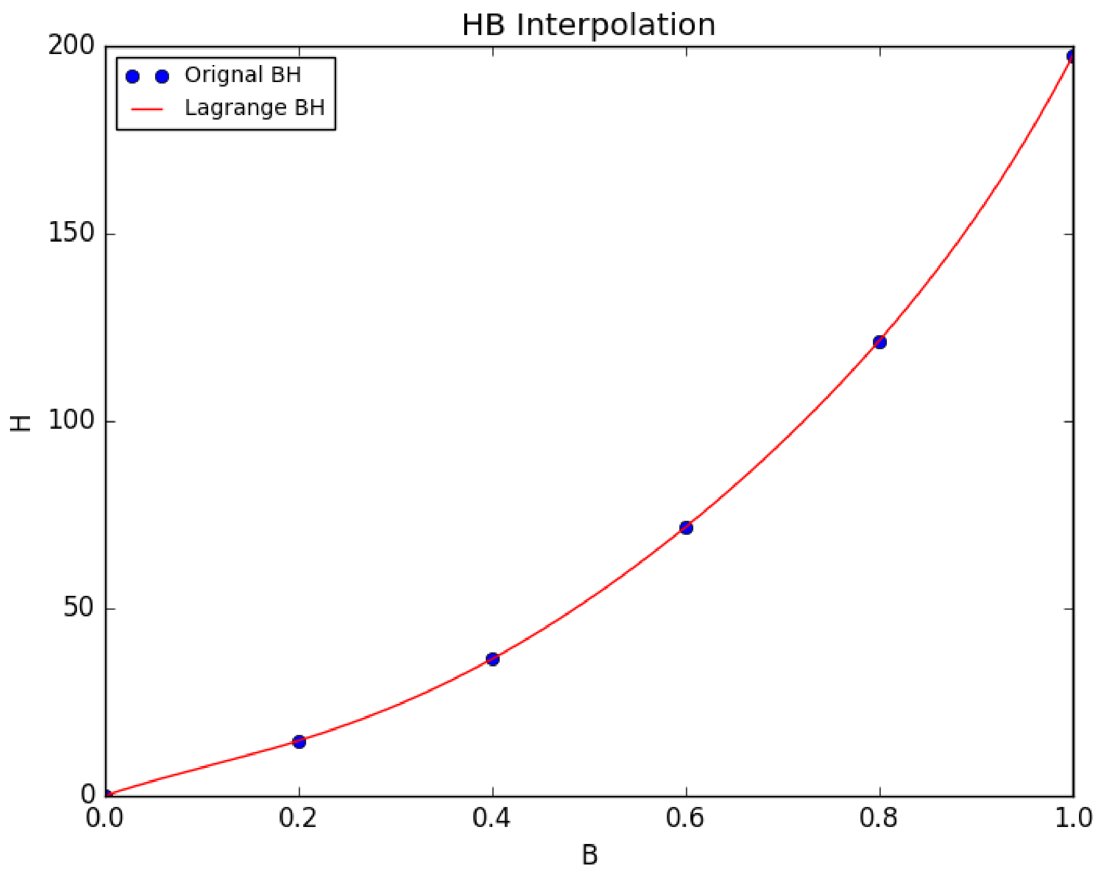
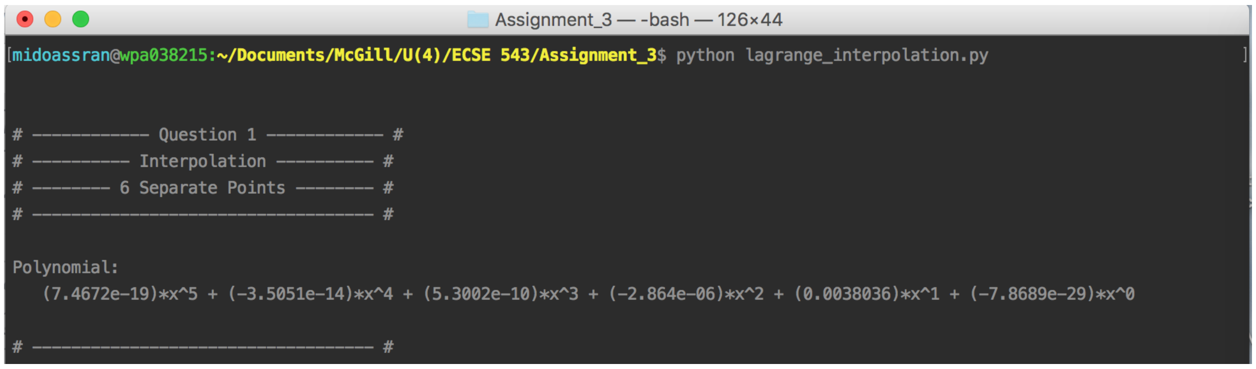


Figure 2: H vs B interpolation using first six points

Part b

Full-domain Lagrange Interpolation is once again performed: the same interpolation implementation code is used, except now the data used for the interpolation are the six points at $B = 0, 1.3, 1.4, 1.7, 1.8, 1.9$. Figures 3 and 4 show the corresponding interpolations of B vs H and H vs B respectively. The console outputs in the figures also showcases the closed form expanded Lagrange Polynomial for each the B vs H and the H vs B interpolations. **This result is not plausible.**



```
Assignment_3 — -bash — 126×44
[midoassran@wpa038215:~/Documents/McGill/U(4)/ECSE 543/Assignment_3$ python lagrange_interpolation.py


# ------------ Question 1 ------------ #
# ---------- Interpolation ---------- #
# -------- 6 Separate Points -------- #
# ----------------------------------- #

Polynomial:
   (7.4672e-19)*x^5 + (-3.5051e-14)*x^4 + (5.3002e-10)*x^3 + (-2.864e-06)*x^2 + (0.0038036)*x^1 + (-7.8689e-29)*x^0

# ----------------------------------- #
```

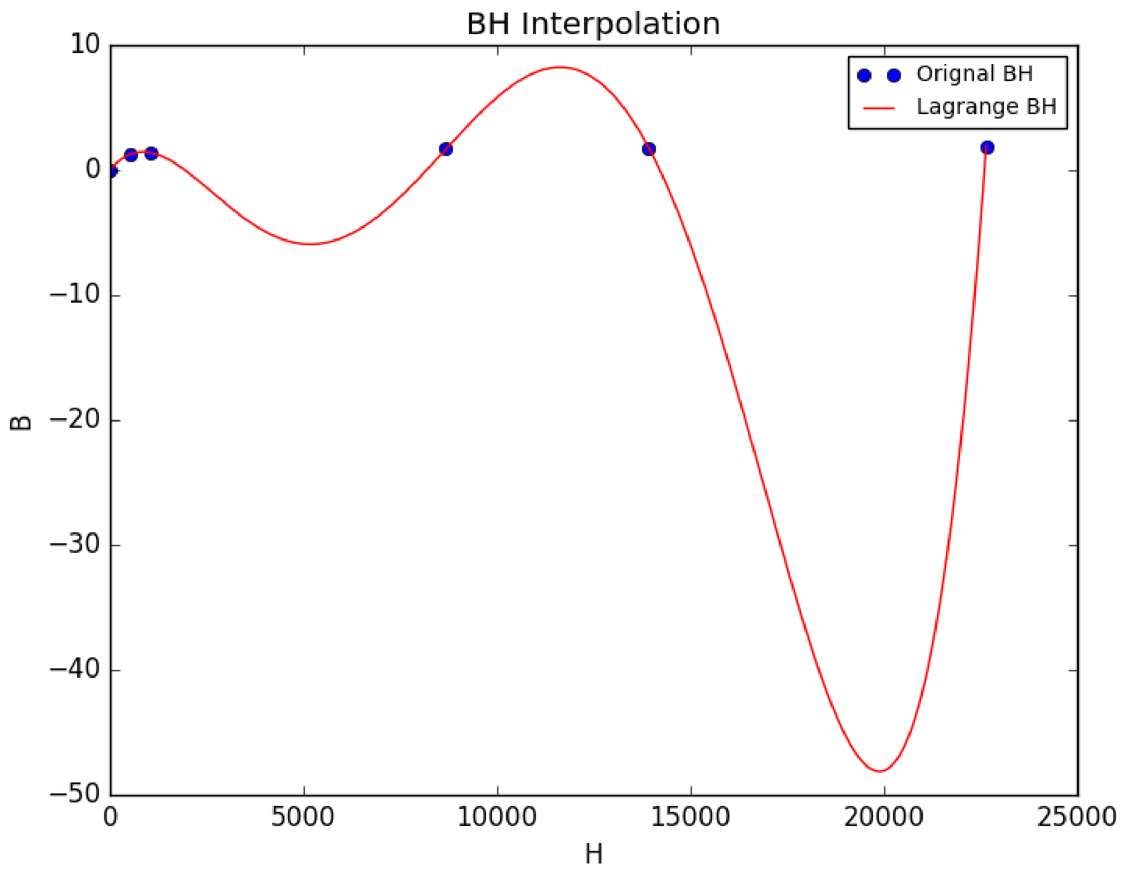
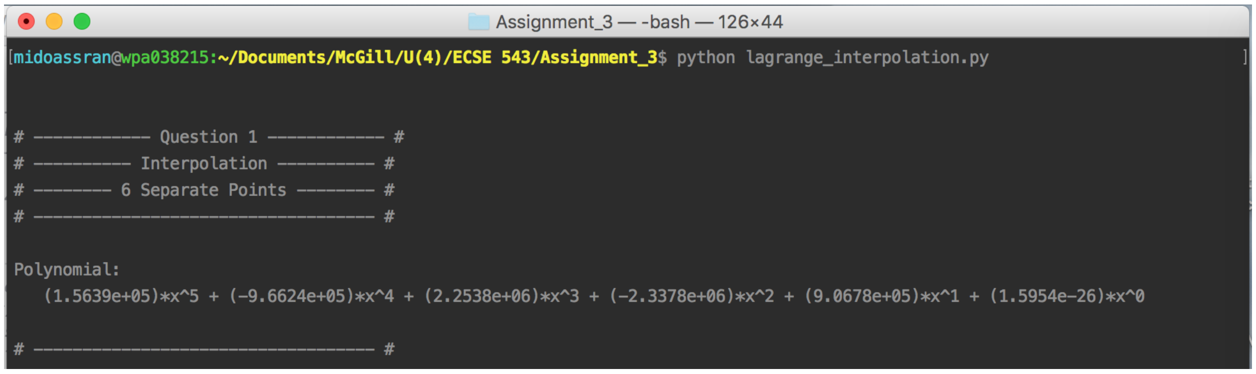


Figure 3: B vs H interpolation using separate six points



```
[midoassran@wpa038215:~/Documents/McGill/U(4)/ECSE 543/Assignment_3$ python lagrange_interpolation.py


# ------------ Question 1 ------------ #
# ---------- Interpolation ---------- #
# -------- 6 Separate Points -------- #
# ---------------------------------- #

Polynomial:
   (1.5639e+05)*x^5 + (-9.6624e+05)*x^4 + (2.2538e+06)*x^3 + (-2.3378e+06)*x^2 + (9.0678e+05)*x^1 + (1.5954e-26)*x^0

# ---------------------------------- #
```

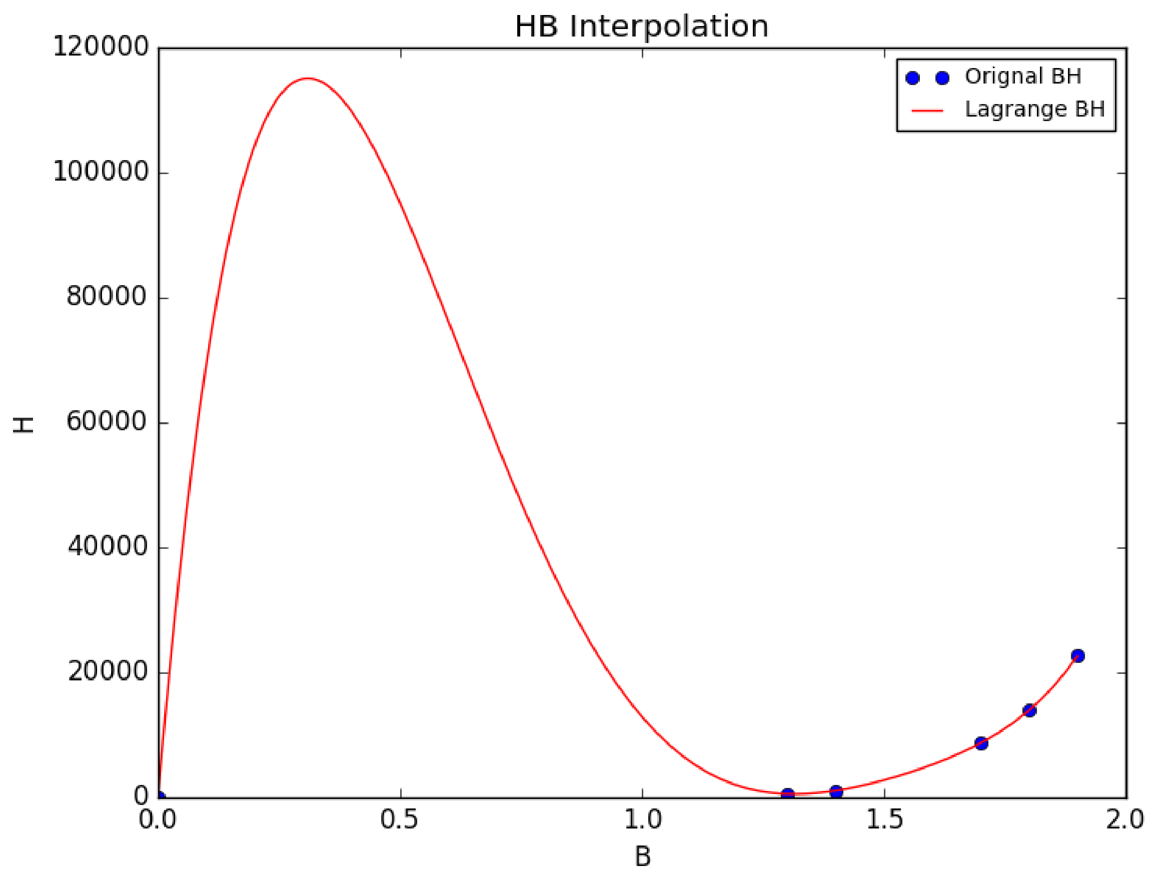


Figure 4: H vs B interpolation using separate six points

Part c

As was seen, full-domain Lagrange Polynomial representations are susceptible to "wiggles" when the domain and target points are not regularly interspersed. Hermite Polynomial representations are less susceptible to such issues since they incorporate first order information (i.e. the slopes). We take data from the six points corresponding to $B = 0, 1.3, 1.4, 1.7, 1.8, 1.9$, and construct sub-domain Hermite Polynomials. We choose to use the six points to construct five non-overlapping subdomains, denoted D_i, where each sub-domain is made up of two points from the dataset. Therefore, we estimate the function to be interpolated, $y(x)$, by creating a two-point Hermite Polynomial representation (cubic) for each subdomain. Our function being interpolated, $y(x)$, is estimated by the piecewise representation:

$$y(x) = \begin{cases} \sum_{j=1}^{2} a_j U_j(x) + b_j V_j(x), & \text{for} \quad x \in D_1 \\ \sum_{j=2}^{3} a_j U_j(x) + b_j V_j(x), & \text{for} \quad x \in D_2 \\ \sum_{j=3}^{4} a_j U_j(x) + b_j V_j(x), & \text{for} \quad x \in D_3 \\ \sum_{j=4}^{5} a_j U_j(x) + b_j V_j(x), & \text{for} \quad x \in D_4 \\ \sum_{j=5}^{6} a_j U_j(x) + b_j V_j(x), & \text{for} \quad x \in D_5 \end{cases} \tag{2}$$

where

$$\begin{aligned} D_1 &= [x_1, x_2] \\ D_2 &= (x_2, x_3] \\ D_3 &= (x_3, x_4] \\ D_4 &= (x_4, x_5] \\ D_5 &= (x_5, x_6] \end{aligned} \tag{3}$$

Even though each Hermite Polynomial sum is actually defined over a closed domain, the aggregate subdomain polynomials are only defined over the semi-open interval to avoid overlap (the choice to open up the lower subinterval was completely arbitrary).

The class *HermiteSubdomainInterpolator* is used to perform the interpolation. The class is initialized with the domain points and the corresponding range points - the six B and H data samples in this case. Subsequently the initializer creates two Hermite Polynomials for each of the five subdomains, and determines the associated model parameters as well.

The Hermite Polynomials are represented by the *HermitePolynomial* class contained in the *polynomial_collective.py* file. The *HertmiePolynomial* class has two public instance methods: *evaluate_U(x, j, dom)* and *evaluate_V(x, j, dom)* which take as input the the set of domain points over which the subdomain polynomial is to be defined (two points in this example), the subscript index j, and the point x in the subdomain at which to evaluate the U_j or V_j components of the Hermite

Polynomial. In a two-point subdomain, we define the polynomials over domain D_i as

$$\begin{aligned}
&U_j(x) = [1 - 2 \cdot L_j'(x_j) \cdot (x - x_j)]L_j^2(x), && \text{for} \quad x, x_j \in D_i \\
&V_j(x) = (x - x_j) \cdot L_j^2(x), && \text{for} \quad x, x_j \in D_i \\
&\text{where} && \\
&L_j(x) = \frac{x - x_k}{x_j - x_k}, && \text{for} \quad x, x_k, x_j \in D_i, \text{ and } k \neq j \\
&L_j'(x) = \frac{1}{x_j - x_k}, && \text{for} \quad x, x_k, x_j \in D_i, \text{ and } k \neq j
\end{aligned}$$

The model parameters are the set of a_j and b_j. The a_j are chosen to be equal to the corresponding range points (the set of y_j), and the b_j are chosen to be equal to the corresponding range derivatives $\frac{dy_j}{dx}$ - i.e the derivative of the function to be interpolated evaluated at the point x_j. To find the six slopes, we use a three point weighted average. Say we want to find the slope at the point (y_j, x_j). Then we define the posterior and the priori slopes:

$$s_{pos} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j}$$

$$s_{pri} = \frac{y_j - y_{j-1}}{x_j - x_{j-1}}$$

and take the slope at the point (y_j, x_j), denoted s_j, to be

$$\boxed{s_j = w \cdot s_{pos} + (1 - w) \cdot s_{pri}}$$

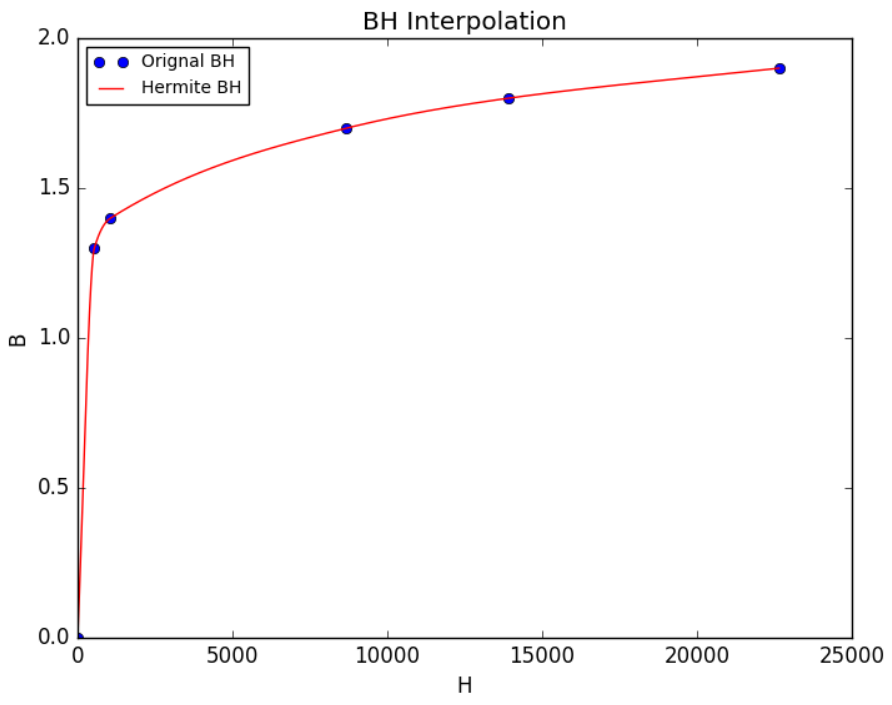
where the weight w is

$$w = \frac{s_{pri}}{s_{pos} + s_{pri}}$$

The result is that the prior and posterior slopes have an equilibrated representation in the slope estimate, without either one overpowering the other. Obviously for the first and last points in our domain we simply only use the posterior and the prior slopes respectively since there is no third point to be leveraged. If we were to simply define the slope at point (y_j, x_j) to be $\frac{y_{j+1} - y_{j-1}}{x_{j+1} - x_{j-1}}$, then the much larger slope would overpower the much smaller slope, and the interpolation would overshoot. Similarly, if we only use the posterior or the prior slopes, then we would be discarding valuable information.

Now back to the *HermiteSubdomainInterpolator* class. After it determines its model parameters, $\{a_j, b_j\}$, and the corresponding Hermite Polynomials, $\{U_j, V_j\}$, it can be used to interpolate the data simply by calling its *interpolate()* instance method. This method returns a Python lambda, i.e. a functional method, which evaluates equation (2). *Note that the implementation works with any number of domain points, but in our mathematical representations we show only six points since those are what are used in this subsection of the assignment.

The result of the interpolation is shown in Figure 5, where both the B vs H and the H vs B interpolations are shown. Clearly our weighted average works extremely well in both cases when the posterior is much larger than the prior (HB interpolation), and when the prior is much larger than the posterior (BH interpolation). **This result is certainly plausible and could lie close to the true BH curve.**



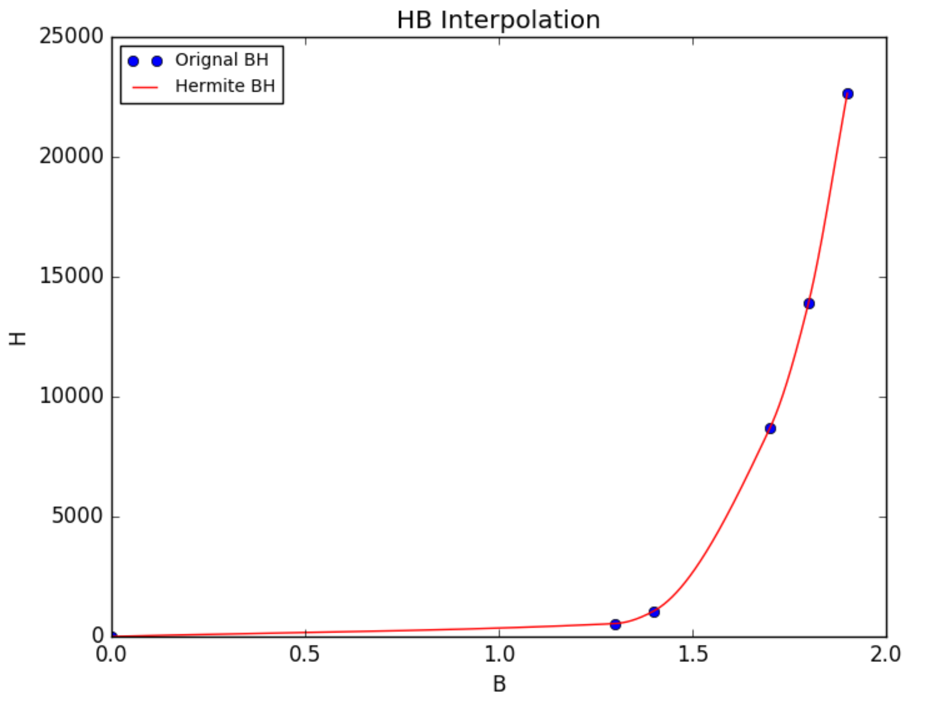


Figure 5: BH interpolation using separate six points

Part d

Here we derive a non-linear equation for the magnetic flux in the iron core in Figure 6. We construct the magnetic circuit shown in Figure 7, where R_c is the reluctance of the core, R_g is the reluctance of the air gap, ψ is the magnetic flux, and M is the magnetomotive force. The constituent equation is

$$(R_g + R_c) \cdot \psi = M \tag{4}$$

where

$$M = N \cdot I$$
$$R_g = \tfrac{L_g}{A\mu_0}$$
$$R_c = \tfrac{L_c}{A\mu}$$

N is the number of turns of the coil, I is the current through the coil, A is the cross-sectional area of the core, μ_0 is the permeability of free space, and μ is the permeability of the core, which is dependent on the magnetic flux. Therefore the reluctance of the core can be expressed as a function of the magnetic flux

$$R_c = \frac{L_c}{A \cdot \frac{B}{H(B)}}$$

or even more directly

$$R_c = \frac{L_c}{A \cdot \frac{\frac{\psi}{A}}{H(\psi)}}$$

which simplifies to

$$R_c = \frac{L_c \cdot H(\psi)}{\psi}$$

where $H(\psi)$ is the magnetic field intensity which is a function of the magnetic flux density, however since the cross-section of the core is a constant, we simply write the flux density dependence $H(B)$ as a magnetic flux dependence instead $H(\psi)$. Substituting into equation (4) and reordering we have:

$$\boxed{f(\psi) = R_g \cdot \psi + L_c \cdot H(\psi) - M = 0}$$

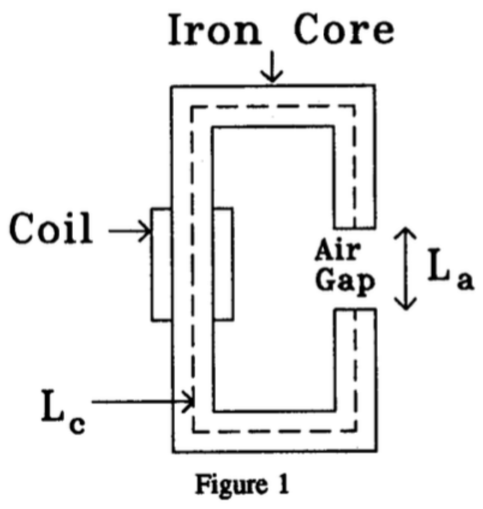


Figure 6: Iron core

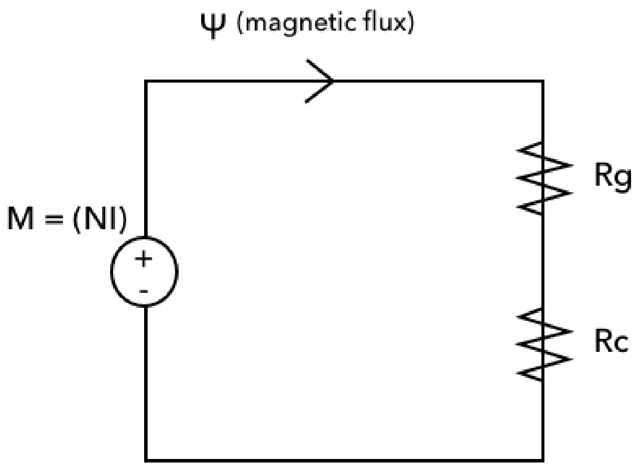


Figure 7: Magnetic circuit

Part e

Here we solve the non-linear equation derived in Part d using the Newton-Raphson method. The first step is to find a piecewise-linear interpolation of the BH data to represent the $H(\psi)$ term. To construct a piecewise-linear interpolation of the data we choose to use a two-point subdomain Lagrange Polynomial interpolation, since such an interpolation defined over a subdomain would create an order one polynomial. We construct $n-1$ non-overlapping subdomains from the n BH points, denoted D_i, where each sub-domain is made up of two points from the dataset. Therefore, we estimate the function to be interpolated, $H(B)$, by creating a two-point Lagrange Polynomial representation (linear) for each subdomain. Our function being interpolated, $H(B)$, is estimated by the piecewise representation:

$$H(B) = \begin{cases} \sum_{j=1}^{2} a_j L_j(B), & \text{for} \quad B \in D_1 \\ \sum_{j=2}^{3} a_j L_j(B), & \text{for} \quad B \in D_2 \\ \vdots \\ \sum_{j=n-1}^{n} a_j L_j(B), & \text{for} \quad B \in D_{n-1} \end{cases} \tag{5}$$

where

$$\begin{aligned} D_1 &= [B_1, B_2] \\ D_2 &= (B_2, B_3] \\ &\vdots \\ D_{n-1} &= (B_{n-1}, B_n] \end{aligned} \tag{6}$$

The class *LagrangeSubdomainInterpolator* is used to perform the interpolation. The class is initialized with the domain points and the corresponding range points - the n B and H data samples in this case. Subsequently the initializer creates two Lagrange Polynomials for each of the $n-1$ subdomains, and determines the associated model parameters as well.

The Lagrange Polynomials are represented by the *LagrangePolynomial* class contained in the *polynomial_collective.py* file. The *LagrangePolynomial* class takes as input the set of domain points $\{B\}$, and the subscript index j. Upon initialization, the *LagrangePolynomial* class creates the following polynomial:

$$L_j(B) = \frac{F_j(B)}{F_j(B_j)}$$

where

$$F_j(B) = (B - B_r), \qquad \text{for} \quad B, B_j \in D_i, \text{ and } r \neq j$$

This polynomial is created by performing a series of binomial multiplications and a scalar division. The binomial operations are performed using the *Polynomial* class represented in the *polynomial.py* file. Once the *LagrangePolynomial* is initialized, it can be evaluated by simply calling its instance method *evaluate*(B), which takes as input a domain point at which to evaluate the Lagrange Polynomial, and returns the resulting scalar. The model parameters, the set of a_j, are simply chosen to be equal to the corresponding range points, the set of B_j.

Now the next step is to solve equation (4) using the $H(B)$ representation interpolated in equation (5). The *NewtonRaphsonSolver* class is used to solve the non-linear equation. The solution can be obtained by calling the class' *solve(starting_guess, f, g, stopping_ratio)* method, which takes as input the starting guess of the optimal argument, the non-linear function in the form of a Python lambda, denoted f, the derivative of the non-linear function in the form of a Python lambda, denoted g, and the ratio of the non-linear function at the final and starting argument values used as a stopping condition for the iterative solver. The Newton Raphson iterations are obtained by solving the equation

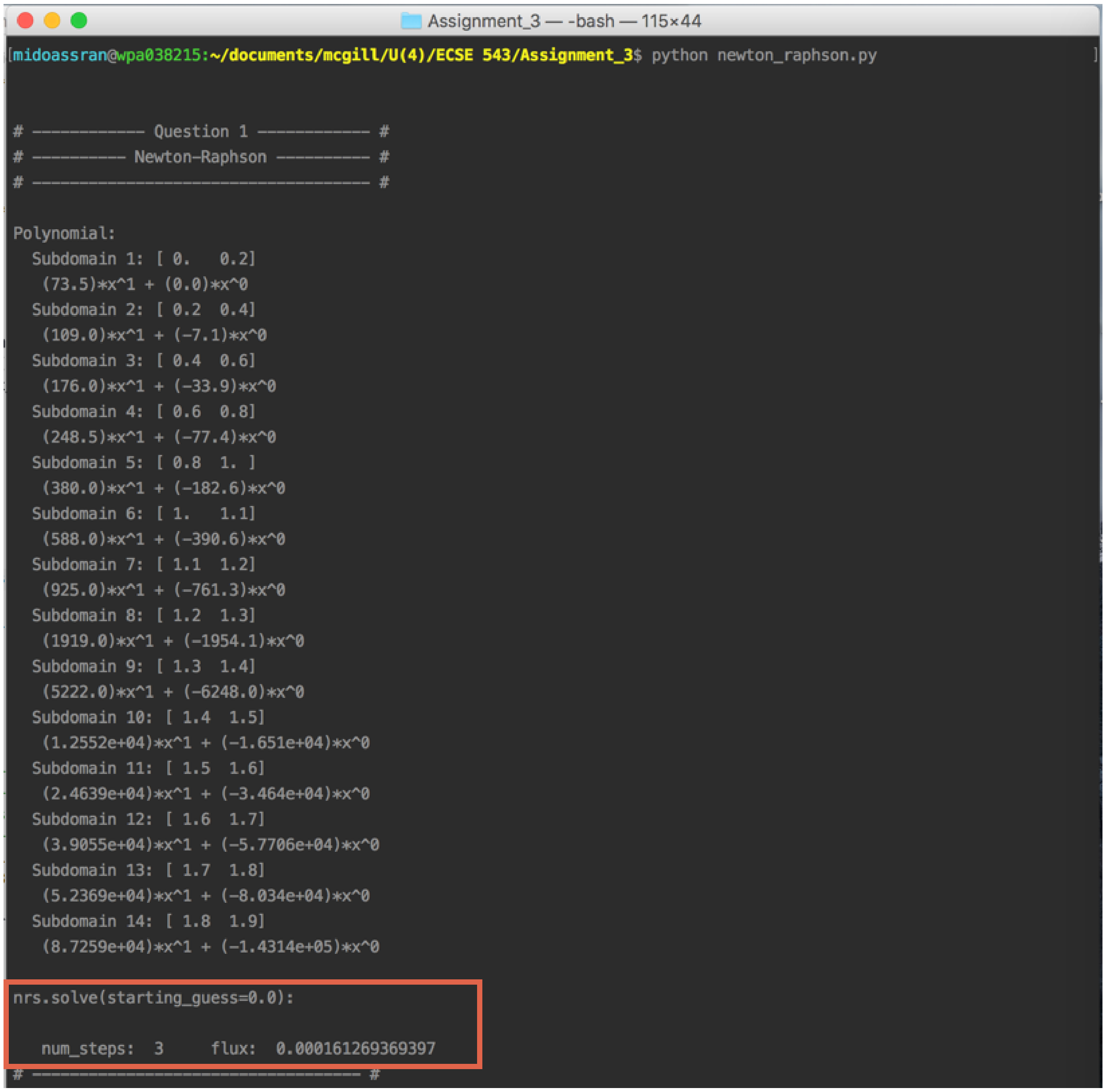
$$g^{(k)}(\psi^{(k+1)} - \psi^{(k)}) + f^{(k)} = 0$$

which gives the $\psi^{(k+1)}$ update

$$\boxed{\psi^{(k+1)} = \psi^{(k)} - \frac{f^{(k)}}{g^{(k)}}}$$

where $\psi^{(k)}$ is the estimate of the flux at the k^{th} optimization iteration, $g^{(k)}$ is the derivative evaluated at $\psi^{(k)}$, and $f^{(k)}$ is the non-linear equation evaluated at $\psi^{(k)}$.

The console output is provided in Figure 8, which shows the piecewise linear representation of $H(B)$ using the Subdomain Lagrange Polynomials, and the number of steps and the final estimate of the flux highlighted in red. **It takes 3 iteration steps from a starting guess of $\psi = 0$, and the final flux is $\psi = 0.000161269369397$.**



Assignment_3 — -bash — 115×44

```
[midoassran@wpa038215:~/documents/mcgill/U(4)/ECSE 543/Assignment_3$ python newton_raphson.py


# ------------ Question 1 ------------ #
# ---------- Newton-Raphson ---------- #
# ------------------------------------ #

Polynomial:
  Subdomain 1: [ 0.   0.2]
   (73.5)*x^1 + (0.0)*x^0
  Subdomain 2: [ 0.2  0.4]
   (109.0)*x^1 + (-7.1)*x^0
  Subdomain 3: [ 0.4  0.6]
   (176.0)*x^1 + (-33.9)*x^0
  Subdomain 4: [ 0.6  0.8]
   (248.5)*x^1 + (-77.4)*x^0
  Subdomain 5: [ 0.8  1. ]
   (380.0)*x^1 + (-182.6)*x^0
  Subdomain 6: [ 1.   1.1]
   (588.0)*x^1 + (-390.6)*x^0
  Subdomain 7: [ 1.1  1.2]
   (925.0)*x^1 + (-761.3)*x^0
  Subdomain 8: [ 1.2  1.3]
   (1919.0)*x^1 + (-1954.1)*x^0
  Subdomain 9: [ 1.3  1.4]
   (5222.0)*x^1 + (-6248.0)*x^0
  Subdomain 10: [ 1.4  1.5]
   (1.2552e+04)*x^1 + (-1.651e+04)*x^0
  Subdomain 11: [ 1.5  1.6]
   (2.4639e+04)*x^1 + (-3.464e+04)*x^0
  Subdomain 12: [ 1.6  1.7]
   (3.9055e+04)*x^1 + (-5.7706e+04)*x^0
  Subdomain 13: [ 1.7  1.8]
   (5.2369e+04)*x^1 + (-8.034e+04)*x^0
  Subdomain 14: [ 1.8  1.9]
   (8.7259e+04)*x^1 + (-1.4314e+05)*x^0

nrs.solve(starting_guess=0.0):

   num_steps:  3     flux:  0.000161269369397
# ----------------------------------- #
```

Figure 8: Console output of Newton Raphson solver for Part e. The piecewise linear representation of $H(B)$ using the Subdomain Lagrange Polynomials is shown. The number of steps and the final estimate of the flux are also recorded.

Part f

Here we solve the non-linear equation derived in Part d using the Successive Substitution method. We use the same implementation code as that used for the *Newton Raphson* method, except in that we constantly use a derivative of 1, rather than a variable metric. The Successive Substitution update is

$$\boxed{\psi^{(k+1)} = \psi^{(k)} - f^{(k)}}$$

The non-linear equation to be solved for

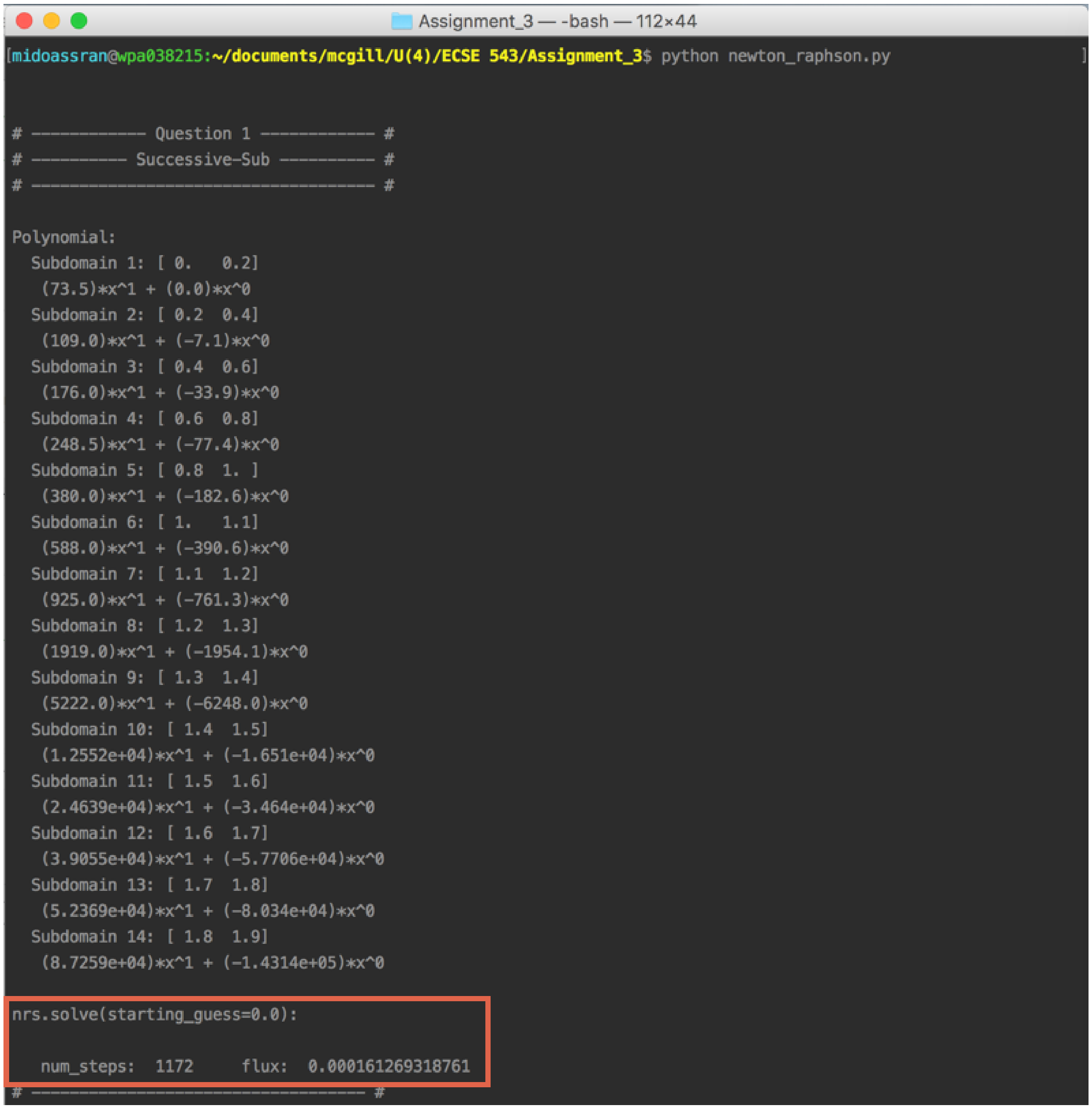
$$f(\psi) = R_g \cdot \psi + L_c \cdot H(\psi) - M = 0$$

does not converge. To get around this issue we modify non-linear equation to the form

$$\boxed{f(\psi) = (R_g \cdot \psi + L_c \cdot H(\psi) - M) \cdot 10^{-10} = 0}$$

This form of the non-linear equation does indeed converge using the Successive Substitution method. The reasoning for scaling down the function is to flatten it out. Since we are always using a constant slope of one in each update, the lower we hit the function on the range, the shorter the distance we will travel along the ψ axis. It is key that we ensure that the value of ψ remain in the domain over which our piecewise linear interpolation of $H(\psi)$ is valid, otherwise we could diverge.

The console output is provided in Figure 9, which shows the piecewise linear representation of $H(B)$ using the Subdomain Lagrange Polynomials, and the number of steps and the final estimate of the flux highlighted in red. **It takes 1172 iteration steps from a starting guess of $\psi = 0$, and the final flux is $\psi = 0.000161269318761$.**


Assignment_3 — -bash — 112×44

```
[midoassran@wpa038215:~/documents/mcgill/U(4)/ECSE 543/Assignment_3$ python newton_raphson.py          ]


# ------------ Question 1 ------------ #
# ---------- Successive-Sub ---------- #
# ---------------------------------- #

Polynomial:
  Subdomain 1: [ 0.   0.2]
   (73.5)*x^1 + (0.0)*x^0
  Subdomain 2: [ 0.2  0.4]
   (109.0)*x^1 + (-7.1)*x^0
  Subdomain 3: [ 0.4  0.6]
   (176.0)*x^1 + (-33.9)*x^0
  Subdomain 4: [ 0.6  0.8]
   (248.5)*x^1 + (-77.4)*x^0
  Subdomain 5: [ 0.8  1. ]
   (380.0)*x^1 + (-182.6)*x^0
  Subdomain 6: [ 1.   1.1]
   (588.0)*x^1 + (-390.6)*x^0
  Subdomain 7: [ 1.1  1.2]
   (925.0)*x^1 + (-761.3)*x^0
  Subdomain 8: [ 1.2  1.3]
   (1919.0)*x^1 + (-1954.1)*x^0
  Subdomain 9: [ 1.3  1.4]
   (5222.0)*x^1 + (-6248.0)*x^0
  Subdomain 10: [ 1.4  1.5]
   (1.2552e+04)*x^1 + (-1.651e+04)*x^0
  Subdomain 11: [ 1.5  1.6]
   (2.4639e+04)*x^1 + (-3.464e+04)*x^0
  Subdomain 12: [ 1.6  1.7]
   (3.9055e+04)*x^1 + (-5.7706e+04)*x^0
  Subdomain 13: [ 1.7  1.8]
   (5.2369e+04)*x^1 + (-8.034e+04)*x^0
  Subdomain 14: [ 1.8  1.9]
   (8.7259e+04)*x^1 + (-1.4314e+05)*x^0

nrs.solve(starting_guess=0.0):

   num_steps:  1172     flux:  0.000161269318761
# --------------------------------- #
```

Figure 9: Console output of Successive Substitution solver for Part f. The piecewise linear representation of $H(B)$ using the Subdomain Lagrange Polynomials is shown. The number of steps and the final estimate of the flux are also recorded.

Question 2

Part a

Here we derive nonlinear equations for a vector of nodal voltages, v_n, in the form $f(v_n) = 0$ for the electric circuit shown in Figure 10. The first equation, $f_1(v_1, v_2)$ is derived by carrying out a Kirchhoff Voltage Loop

$$\boxed{f_1(v_1, v_2) = v_1 - E + R \cdot I_{SA}(e^{\frac{v_1 - v_2}{kT/q}} - 1) = 0}$$

where v_1 and v_2 are the nodal voltages, E is the source voltage, R is the resistor's resistance, the term $I_{SA}(e^{\frac{v_1 - v_2}{kT/q}} - 1)$ is the current through diode A, $kT/q \approx 25mV$ and I_{SA} is diode A's saturation current. The second equation, $f_2(v_1, v_2)$ is derived using a nodal analysis at node v_2

$$\boxed{f_2(v_1, v_2) = I_{SA}(e^{\frac{v_1 - v_2}{kT/q}} - 1) - I_{SB}(e^{\frac{v_2 - 0}{kT/q}} - 1) = 0}$$

where v_1 and v_2 are the nodal voltages, E is the source voltage, R is the resistor's resistance, the term $I_{SA}(e^{\frac{v_1 - v_2}{kT/q}} - 1)$ is the current through diode A, the term $I_{SB}(e^{\frac{v_2}{kT/q}} - 1)$ is the current through diode B, $kT/q \approx 25mV$, I_{SB} is diode B's saturation current and I_{SA} is diode A's saturation current. In vector form we have

$$\mathbf{f}(\mathbf{v}) = [f_1(\mathbf{v}), f_2(\mathbf{v})]^T = 0$$

where

$$\mathbf{v} = [v_1, v_2]^T$$

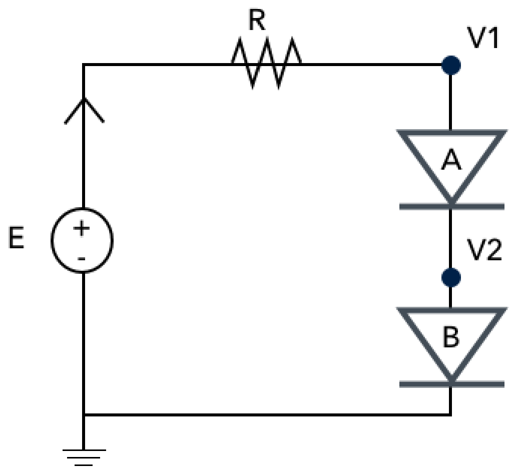


Figure 10

Part b

The *NewtonRaphsonSolver* class is used to solve the non-linear equation $\mathbf{f} = \mathbf{0}$. The solution can be obtained by calling the class' *solve_2D(starting_guess, f, J, stopping_ratio)* method, which takes as input the 2-D array input of starting guesses (the starting node voltages), the non-linear 2D function ($[f_1, f_2]$) in the form of a list of Python lambdas, the Jacobian of the non-linear functions $\mathbf{f}$, denoted by J, in the form of a 2×2 matrix of Python lambdas, and the minimum error used as the stopping condition for the iterative solver. The Newton Raphson iterations are obtained by solving the equation

$$\mathbf{J^{(k)}} \cdot (\mathbf{v}^{(k+1)} - \mathbf{v}^{(k)}) + \mathbf{f^{(k)}} = 0$$

which gives the $\mathbf{v^{(k+1)}}$ update

$$\boxed{\mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} - (\mathbf{J^{(k)}})^{-1} \cdot \mathbf{f^{(k)}}}$$

where $\mathbf{v}^{(k)}$ is the estimate of the node voltages at the k^{th} optimization iteration, $\mathbf{J}^{(k)}$ is the Jacobian evaluated at $\mathbf{v}^{(k)}$, and $\mathbf{f}^{(k)}$ is the vector of non-linear equation evaluated at $\mathbf{v}^{(k)}$. In actuality, our implementation determines $temp = -(\mathbf{J^{(k)}})^{-1} \cdot \mathbf{f^{(k)}}$ by solving

$$(\mathbf{J^{(k)}})^T \cdot (\mathbf{J^{(k)}}) \cdot temp = -(\mathbf{J^{(k)}})^T \mathbf{f^{(k)}}$$

using our previously created Choleski Decomposition implementation. The error metric, denoted $e^{(k)}$, is simply chosen to be

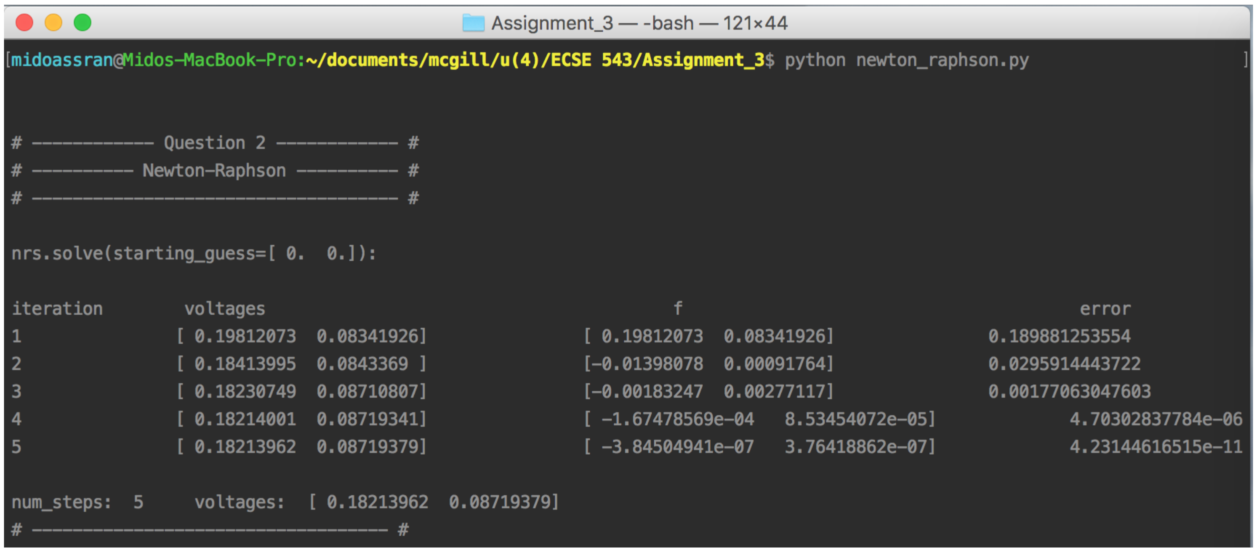
$$e^{(k)} = \frac{||\mathbf{f^{(k)}}||_2}{E}$$

where we have normalized the norm of the vector of functions by the voltage-source voltage.

The console output is provided in Figure 11, and showcases the vector values of the voltage $\mathbf{v}^{(k)}$, the function $\mathbf{f}^{(k)}$, and the error $e^{(k)}$ at each iteration k. Figure 12 plots the error metric on a logarithmic y axis, and fits it to an ideal quadratic convergence curve. The convergence behaviour observed is known as superlinear, or equivalently, quadratic. The mathematical representation of a quadratically convergent series is

$$\frac{1}{2^{2^{i \cdot \kappa}}}$$

where κ is an arbitrary constant. This ideal curve is shown by the dotted blue line in Figure 12, and clearly fits the observed error's convergence curve quite well. **The convergence is quadratic.**



```
[midoassran@Midos-MacBook-Pro:~/documents/mcgill/u(4)/ECSE 543/Assignment_3$ python newton_raphson.py


# ------------ Question 2 ------------ #
# ---------- Newton-Raphson ---------- #
# ---------------------------------- #

nrs.solve(starting_guess=[ 0.  0.]):

iteration        voltages                                  f                                      error
1               [ 0.19812073  0.08341926]                [ 0.19812073  0.08341926]                0.189881253554
2               [ 0.18413995  0.0843369 ]                [-0.01398078  0.00091764]                0.0295914443722
3               [ 0.18230749  0.08710807]                [-0.00183247  0.00277117]                0.00177063047603
4               [ 0.18214001  0.08719341]                [ -1.67478569e-04   8.53454072e-05]              4.70302837784e-06
5               [ 0.18213962  0.08719379]                [ -3.84504941e-07   3.76418862e-07]              4.23144616515e-11

num_steps:  5     voltages:  [ 0.18213962  0.08719379]
# ---------------------------------- #
```

Figure 11: Console output of Newton Raphson showing the voltage estimates, the function f, and the error at each iteration.

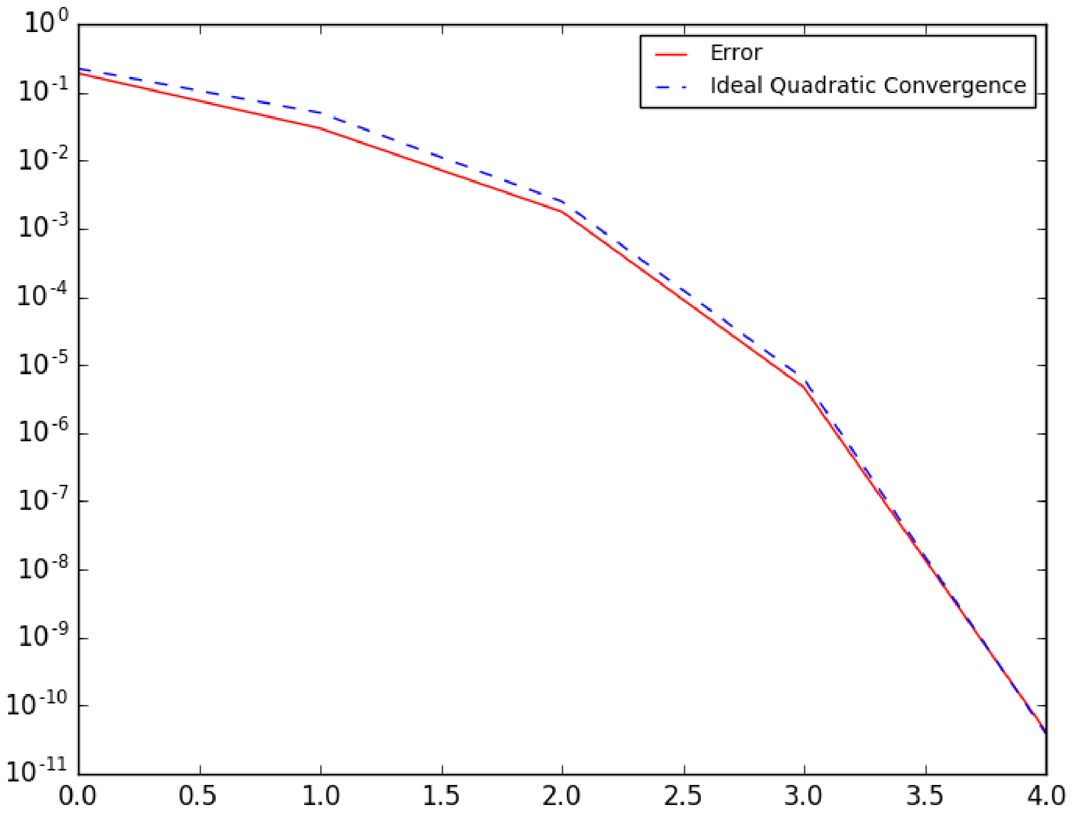


Figure 12: The Newton Raphson error metric plotted on a logarithmic y axis, and fitted to an ideal quadratic convergence curve.

Question 3

Program implementation

We write a program that integrates a function $f(x)$ on the interval $[x_0, x_N]$ by dividing it into subintervals and performing one-point Gauss-Legendre integration for each subinterval a Riemann Sum fashion. The basic Gauss Legendre approximation is

$$\int_{\zeta=-1}^{1} f(\zeta)d\zeta = \sum_{i=0}^{n} w_i f(\zeta_i)$$

where we have $n+1$ unknown weights, and $n+1$ abscissa points, therefore we have $2n+2$ degrees of freedom. Therefore, the following functions, $\zeta^0, \zeta^1, \ldots, \zeta^{2n-1}$ are integrated exactly. For one point Gauss Legendre we have only one abscissa point, ζ_0, and one weight, w_0, to be determined; consequently we have two degrees of freedom with one point Gauss Legendre, and we must solve a linear system of two equations:

$$\int_{\zeta=-1}^{1} d\zeta = w_0$$

$$\int_{\zeta=-1}^{1} \zeta d\zeta = w_0\zeta_0$$

However in general we do not wish to use the Gauss Legendre interval $[-1, 1]$, we would like to generalize this to a an arbitrary interval $[a, b]$ and solve

$$\int_{x=a}^{b} f(x)dx = \sum_{i=0}^{n} w_i f(x_i)$$

To do this we simply carry out a coordinate mapping. The result of the mapping is the new system of equations to be solved

$$\boxed{\begin{aligned} \frac{b-a}{2}\int_{\zeta=-1}^{1} d\zeta &= w_0 \qquad (7) \\ \frac{b-a}{2}\int_{\zeta=-1}^{1} \left(\frac{b-a}{2}\zeta + \frac{b+a}{2}\right)d\zeta &= w_0\zeta_0 \qquad (8) \end{aligned}}$$

which when solved give us

$$\boxed{\begin{aligned} w_0 &= b-a \\ \zeta_0 &= \frac{1}{2}(b+a) \end{aligned}}$$

where w_0 is the weight in the one point Gauss Legendre subinterval, and ζ_0 is the abscissa in the one point Gauss Legendre subinterval. We have that the weight is equal to the interval width, and the abscissa is the midpoint of the subinterval. As a side note, this result is a common Riemann Sum technique. The error in the integral approximation is determined from the dominant term in the Taylor Series expansion of the function to be integrated, $f(x)$. For one point Gauss Legendre, the constant and linear terms will be integrated exactly, but the quadratic and higher degree terms will not. Therefore, the dominant error in one point Gauss Legendre is the quadratic term in the Taylor Series expansion. The error is simply obtained by finding the difference between the "true" solution and the program output.

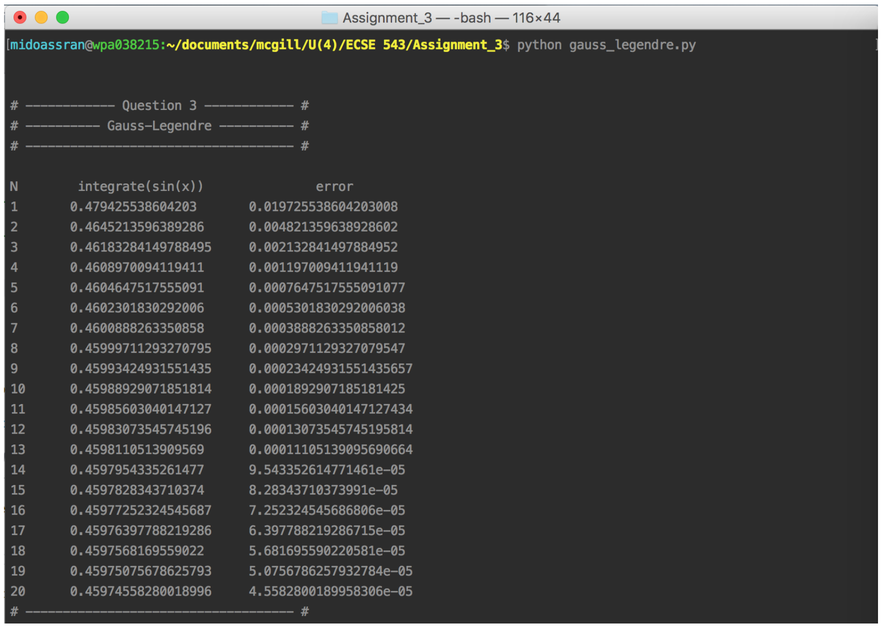
The integration is carried out using the *OnePointGaussLegendre* class, by calling its *integrate(function, interval, num_segments)* function which takes as input the function to be integrated in the form of a Python lambda, the interval over which to perform the integration, and the number of segments the interval is to be split up into. This method then performs the aforementioned logic, and returns the result of the integration (scalar).

Part a

We use the program to integrate the function $f(x) = sin(x)$ on the interval $[0, 1]$ for N number of segments $1, 2, \ldots, 20$. Figure 13 shows the base 10 logarithm of the error in the integration plotted versus the base 10 logarithm of the number of subintervals used. The console output providing the exact integration and error estimates for each number of subintervals is also shown in Figure 13. The error minimization curve observed is known as a "sublinearly convergent" error in the literature, and corresponds to minimization of the mathematical form

$$error \propto \frac{1}{N+1}$$

Intuitively what this means is that when the number of segments used is quite small, it is very beneficial to increase the number of segments used. This also means that further incremental increases in the number of segments used results in less incremental benefits. That is, our error will improve very quickly initially, and then the improvements will slow down as we increase the number of segments used.



```
Assignment_3 — -bash — 116×44
[midoassran@wpa038215:~/documents/mcgill/U(4)/ECSE 543/Assignment_3$ python gauss_legendre.py


# ------------ Question 3 ------------ #
# ---------- Gauss-Legendre ---------- #
# ---------------------------------- #

N        integrate(sin(x))              error
1       0.479425538604203       0.019725538604203008
2       0.4645213596389286      0.004821359638928602
3       0.46183284149788495     0.002132841497884952
4       0.4608970094119411      0.001197009411941119
5       0.4604647517555091      0.0007647517555091077
6       0.4602301830292006      0.0005301830292006038
7       0.4600888263350858      0.0003888263350858012
8       0.45999711293270795     0.0002971129327079547
9       0.45993424931551435     0.00023424931551435657
10      0.45988929071851814     0.0001892907185181425
11      0.45985603040147127     0.00015603040147127434
12      0.45983073545745196     0.00013073545745195814
13      0.4598110513909569      0.00011105139095690664
14      0.4597954335261477      9.543352614771461e-05
15      0.4597828343710374      8.28343710373991e-05
16      0.45977252324545687     7.252324545686806e-05
17      0.45976397788219286     6.397788219286715e-05
18      0.4597568169559022      5.681695590220581e-05
19      0.45975075678625793     5.0756786257932784e-05
20      0.45974558280018996     4.5582800189958306e-05
# ---------------------------------- #
```

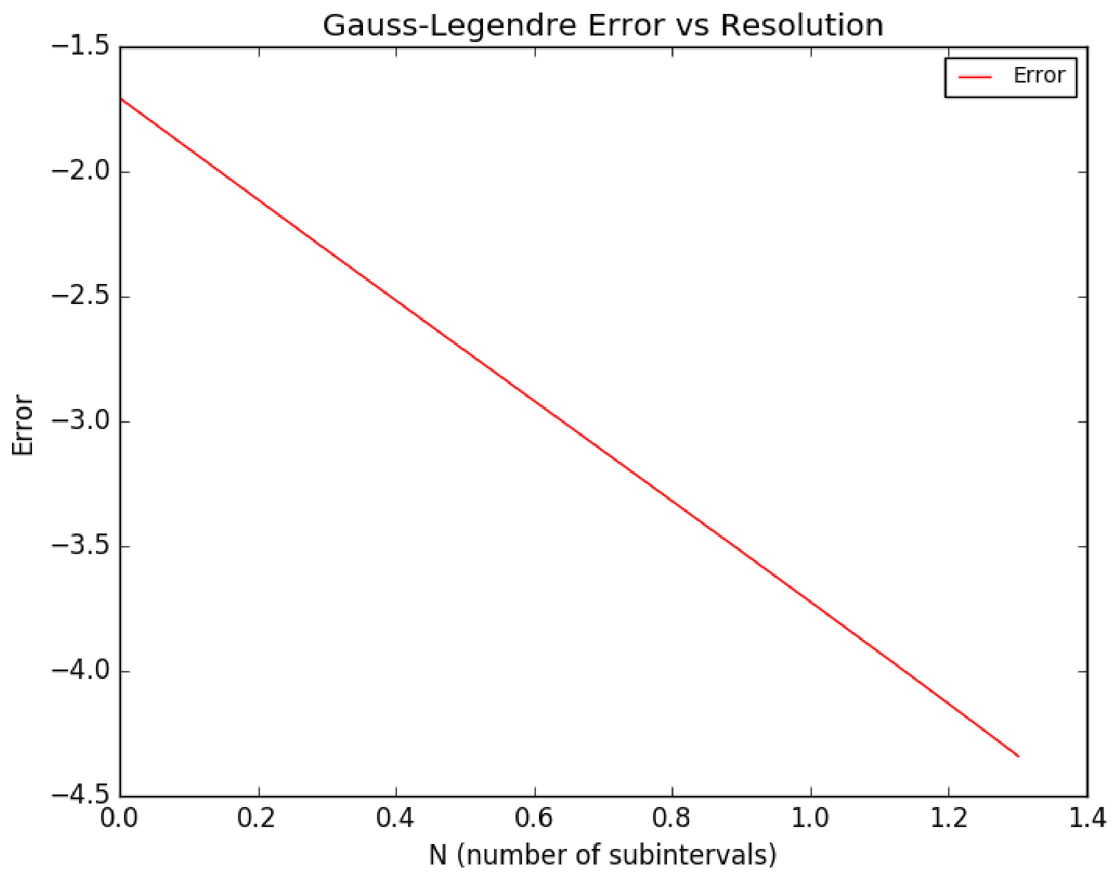


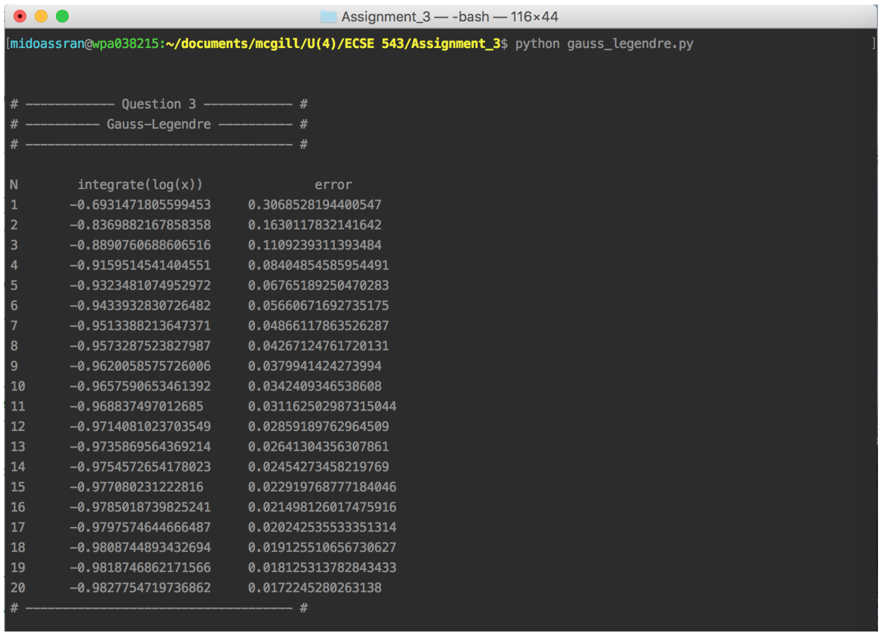
Figure 13: The base 10 logarithm in the error in the Gauss Legendre integration of $sin(x)$ over the interval $[0, 1]$ plotted vs the base 10 logarithm of the number of subintervals the range is broken into.

Part b

We use the program to integrate the function $f(x) = ln(x)$ on the interval $[0, 1]$ for N number of segments $1, 2, \ldots, 20$. Figure 14 shows the base 10 logarithm of the error in the integration plotted versus the base 10 logarithm of the number of subintervals used. The console output providing the exact integration and error estimates for each number of subintervals is also shown in Figure 14. The error minimization curve observed is known as a "sublinearly convergent" error in the literature, and corresponds to minimization of the mathematical form

$$error \propto \frac{1}{N+1}$$

Intuitively what this means is that when the number of segments used is quite small, it is very beneficial to increase the number of segments used. This also means that further incremental increases in the number of segments used results in less incremental benefits. That is, our error will improve very quickly initially, and then the improvements will slow down as we increase the number of segments used. This result is very similar to that of Part a.



```
[midoassran@wpa038215:~/documents/mcgill/U(4)/ECSE 543/Assignment_3$ python gauss_legendre.py


# ------------ Question 3 ------------ #
# ---------- Gauss-Legendre ---------- #
# ---------------------------------- #

N        integrate(log(x))              error
1       -0.6931471805599453     0.3068528194400547
2       -0.8369882167858358     0.1630117832141642
3       -0.8890760688606516     0.1109239311393484
4       -0.9159514541404551     0.08404854585954491
5       -0.9323481074952972     0.06765189250470283
6       -0.9433932830726482     0.05660671692735175
7       -0.9513388213647371     0.04866117863526287
8       -0.9573287523827987     0.04267124761720131
9       -0.9620058575726006     0.0379941424273994
10      -0.9657590653461392     0.0342409346538608
11      -0.968837497012685      0.031162502987315044
12      -0.9714081023703549     0.02859189762964509
13      -0.9735869564369214     0.02641304356307861
14      -0.9754572654178023     0.02454273458219769
15      -0.977080231222816      0.022919768777184046
16      -0.9785018739825241     0.021498126017475916
17      -0.9797574644666487     0.020242535533351314
18      -0.9808744893432694     0.019125510656730627
19      -0.9818746862171566     0.018125313782843433
20      -0.9827754719736862     0.0172245280263138
# ---------------------------------- #
```

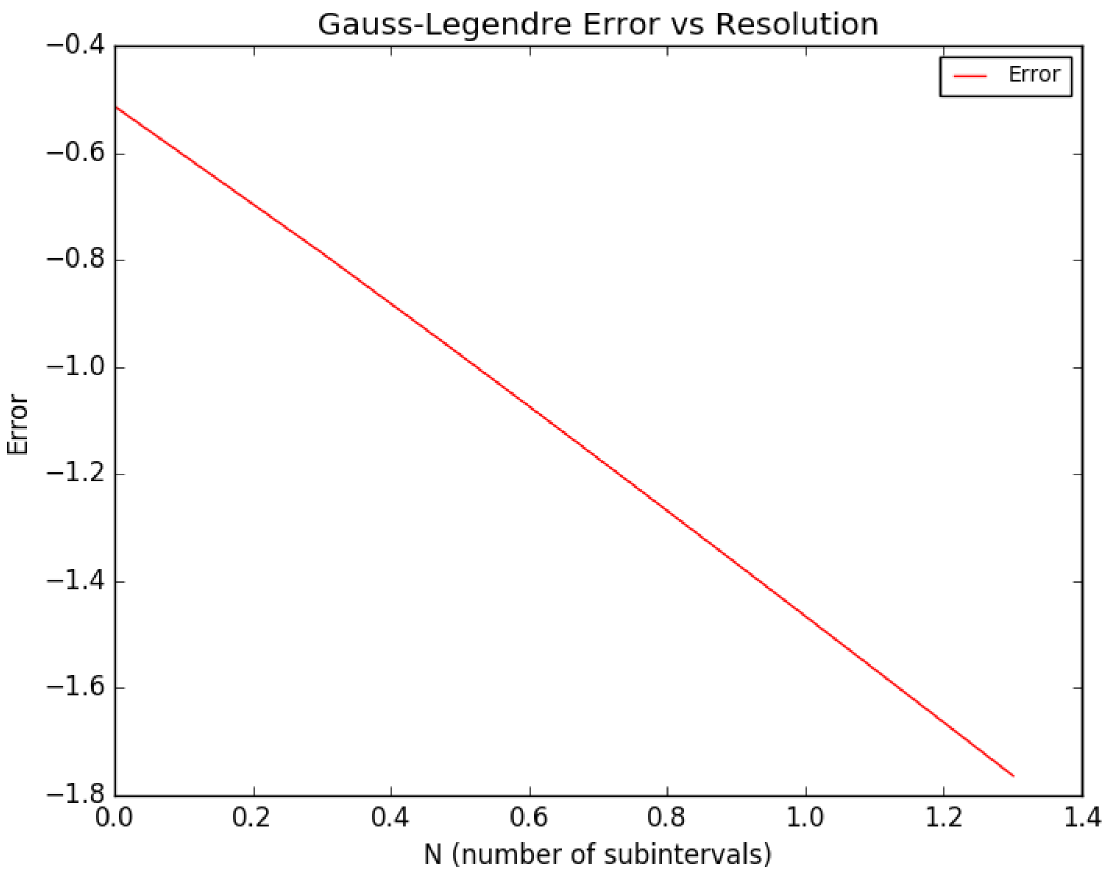


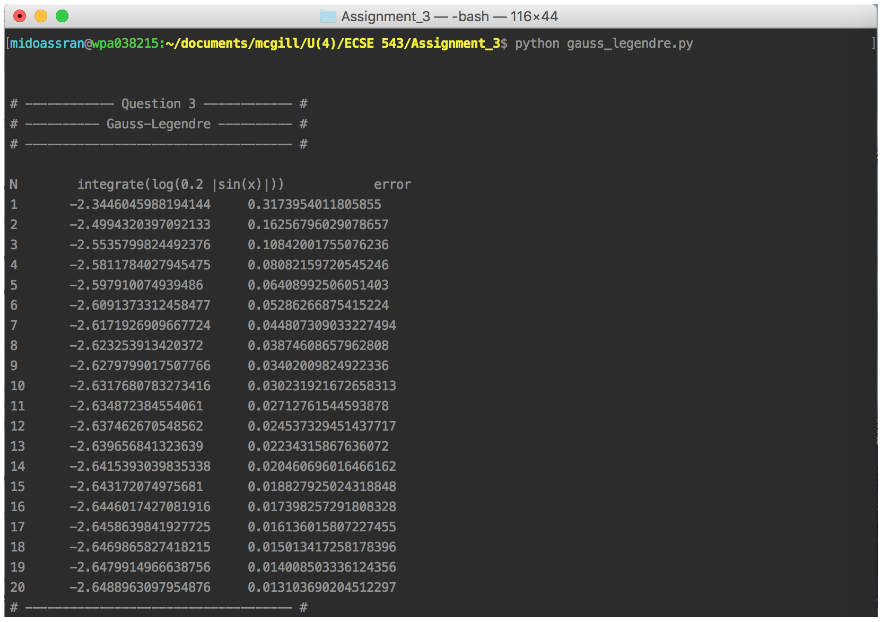
Figure 14: The base 10 logarithm in the error in the Gauss Legendre integration of $ln(x)$ over the interval $[0, 1]$ plotted vs the base 10 logarithm of the number of subintervals the range is broken into.

Part c

We use the program to integrate the function $f(x) = ln(0.2|sin(x)|)$ on the interval $[0, 1]$ for N number of segments $1, 2, \dots, 20$. Figure 15 shows the base 10 logarithm of the error in the integration plotted versus the base 10 logarithm of the number of subintervals used. The console output providing the exact integration and error estimates for each number of subintervals is also shown in Figure 15. Once again, the error minimization curve observed appears to be "sublinearly convergent", and corresponds to minimization of the mathematical form

$$error \propto \frac{1}{N+1}$$

However it also appears to minimize the error faster than Part a and Part b, i.e less sublinear than Part a and Part b, but this is very difficult to judge qualitatively.



```
[midoassran@wpa038215:~/documents/mcgill/U(4)/ECSE 543/Assignment_3$ python gauss_legendre.py


# ------------ Question 3 ------------ #
# ---------- Gauss-Legendre ---------- #
# ---------------------------------- #

N        integrate(log(0.2 |sin(x)|))           error
1       -2.3446045988194144     0.3173954011805855
2       -2.4994320397092133     0.16256796029078657
3       -2.5535799824492376     0.10842001755076236
4       -2.5811784027945475     0.08082159720545246
5       -2.597910074939486      0.06408992506051403
6       -2.6091373312458477     0.05286266875415224
7       -2.6171926909667724     0.044807309033227494
8       -2.623253913420372      0.03874608657962808
9       -2.6279799017507766     0.03402009824922336
10      -2.6317680783273416     0.030231921672658313
11      -2.634872384554061      0.02712761544593878
12      -2.637462670548562      0.024537329451437717
13      -2.639656841323639      0.02234315867636072
14      -2.6415393039835338     0.020460696016466162
15      -2.643172074975681      0.018827925024318848
16      -2.6446017427081916     0.017398257291808328
17      -2.6458639841927725     0.016136015807227455
18      -2.6469865827418215     0.015013417258178396
19      -2.647991496638756      0.014008503336124356
20      -2.6488963097954876     0.013103690204512297
# ---------------------------------- #
```

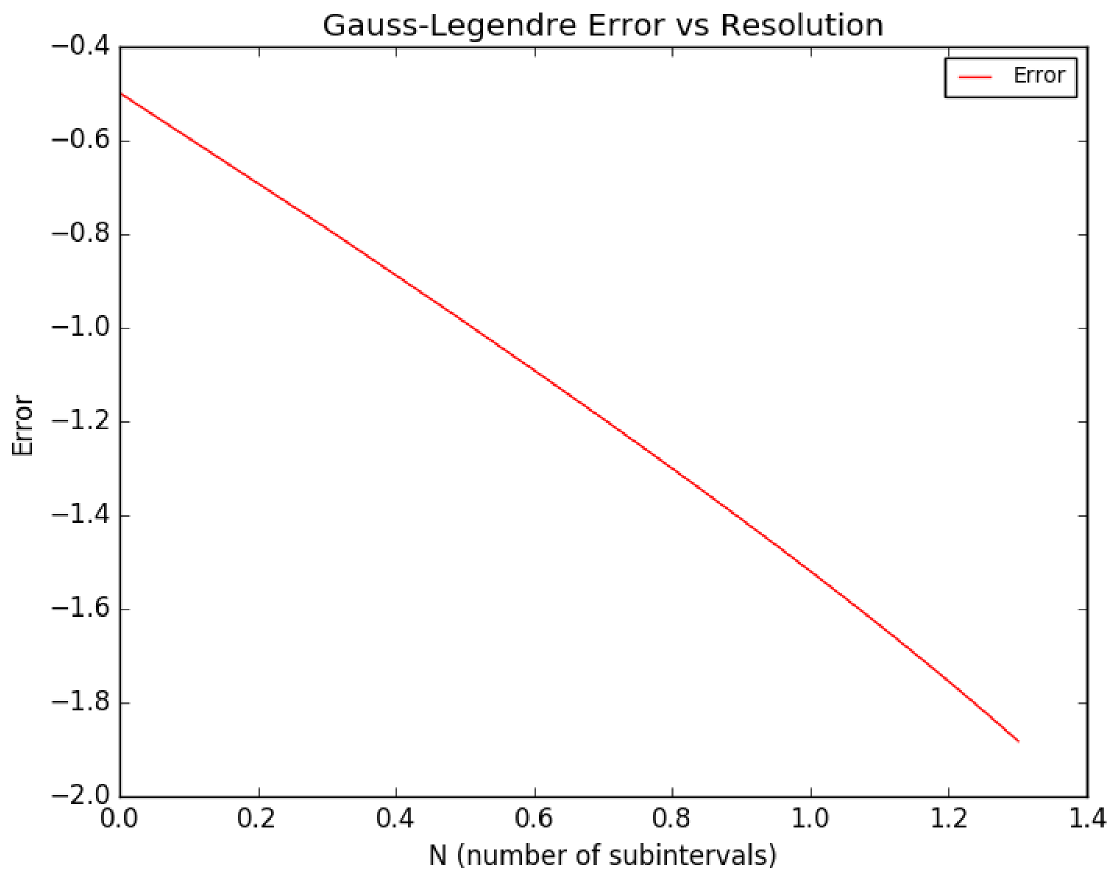
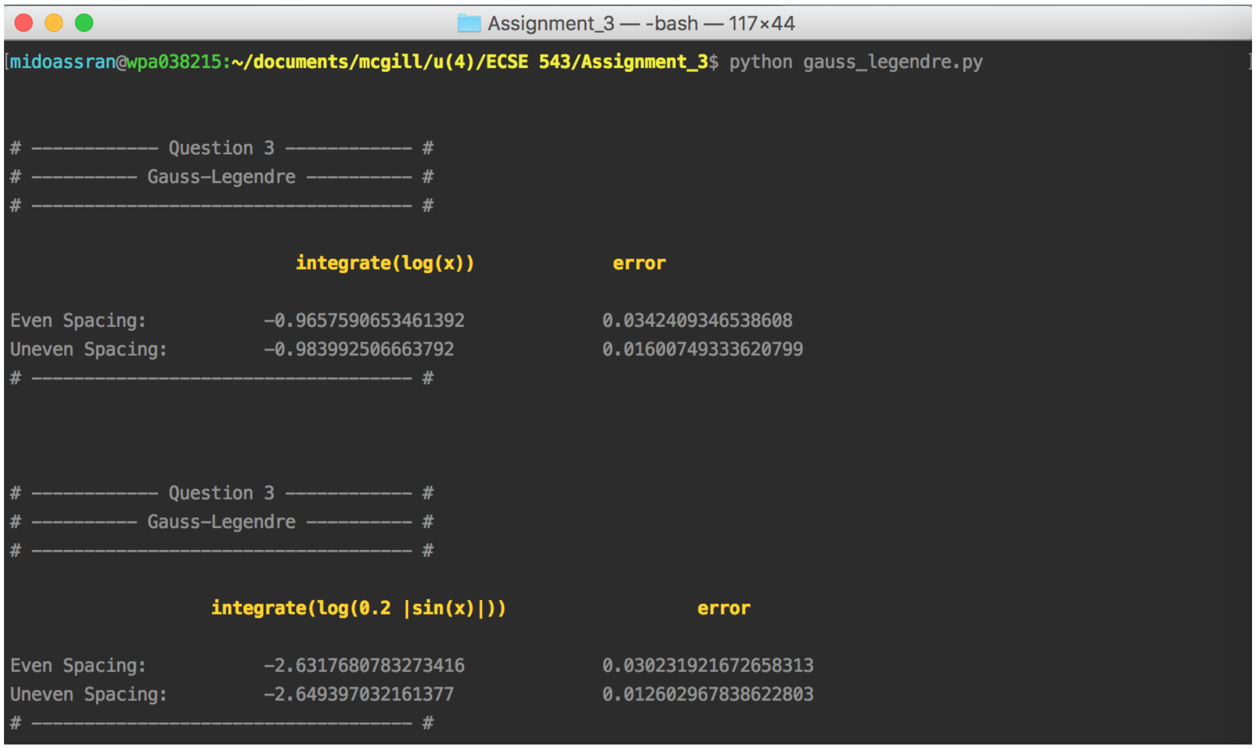


Figure 15: The base 10 logarithm in the error in the Gauss Legendre integration of $ln(0.2|sin(x)|)$ over the interval $[0, 1]$ plotted vs the base 10 logarithm of the number of subintervals the range is broken into.

Part d

Here we integrate $f(x)$ in Part b and Part c using ten unequally spaced segments. We choose to make the segment widths much smaller closer to $x = 0$, and make the segment widths larger closer to $x = 1$. The integration is carried out using the *OnePointGaussLegendre* class, by calling its *integrate_unequal(function, interval, num_segments)* function which takes as input the function to be integrated in the form of a Python lambda, the interval over which to perform the integration, and the number of segments the interval is to be split up into. This method then performs the aforementioned logic, and returns the result of the integration (scalar). The result is shown in Figure 16, which shows the error in the Gauss Legendre integration of $ln(x)$ and $ln(0.2|sin(x)|)$ over the interval $[0, 1]$ for evenly spaced subintervals and unevenly spaced subintervals. **The error is lower when unevenly spaced subintervals are used. For $ln(x)$ the evenly spaced error is 0.0342409346538608, whereas this error is 0.016007493336220799 for unevenly spaced subintervals. For $ln(0.2|sin(x)|)$ the evenly spaced error is 0.030231921672658313, whereas the error is 0.012602967838622803 for unevenly spaced subintervals.**



```
Assignment_3 — -bash — 117×44
[midoassran@wpa038215:~/documents/mcgill/u(4)/ECSE 543/Assignment_3$ python gauss_legendre.py


# ------------ Question 3 ------------ #
# ---------- Gauss-Legendre ---------- #
# ------------------------------------ #

                          integrate(log(x))             error

Even Spacing:           -0.9657590653461392           0.0342409346538608
Uneven Spacing:         -0.983992506663792            0.01600749333620799
# ------------------------------------ #



# ------------ Question 3 ------------ #
# ---------- Gauss-Legendre ---------- #
# ------------------------------------ #

                  integrate(log(0.2 |sin(x)|))                error

Even Spacing:           -2.6317680783273416           0.030231921672658313
Uneven Spacing:         -2.649397032161377            0.012602967838622803
# ------------------------------------ #
```

Figure 16: The error in the Gauss Legendre integration of $ln(x)$ and $ln(0.2|sin(x)|)$ over the interval $[0, 1]$ for evenly spaced subintervals and unevenly spaced subintervals. The error is lower when unevenly spaced subintervals are used.

```python
# ----------------------------------------------------------- #
# Lagrange Interpolation
# ----------------------------------------------------------- #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: LagrangeInterpolator is a class that uses Lagrange
# polynomials to interpolate a data set by minimizing the least
# squares error with respect to the (domain, target) points.

import numpy as np
from utils import *
from choleski import CholeskiDecomposition
from polynomial_collective import LagrangePolynomial

class LagrangeInterpolator:

    def __init__(self, dom, target):
        """
        :type dom: ndarray([float])
        :type target: ndarray([float])
        """
        self.dom, self.target = dom, target

        self.polynomials = []
        for j in range(len(dom)):
            self.polynomials.append(LagrangePolynomial(j, dom))

        self.degree = self.polynomials[0].degree

         # Least squares curve fitting
        self.params = self.determine_model_parameters()

        self.coefficients = [
           sum([self.params[j] * lp.coefficients[k]
                for j, lp in enumerate(self.polynomials)])
            for k in range(self.degree + 1)]

        print(self)

    def interpolate(self):
        """
        :rtype: lambda(float x)
        """
        a, polys, dom, target = self.params, self.polynomials, self.dom, self.target
        n = len(dom)    # Number of domain coordinates
        y = lambda x: sum([a[j] * polys[j].evaluate(x) for j in range(n)])
        return y

    def determine_model_parameters(self):
        """
        < Least Squares curve fitting >
        :rtype: ndarray([float])
        """
        polys, dom, target = self.polynomials, self.dom, self.target

        # Create Matrix
        G = np.empty([dom.shape[0], dom.shape[0]])
        G[:] = -1
        for i, row in enumerate(G):
            for j, _ in enumerate(row):
                if G[i,j] == -1:
                    G[i,j] = sum([polys[i].evaluate(x_k) * polys[j].evaluate(x_k) for x_k in dom])
                    G[j,i] = G[i,j]

        # Create target
        b = np.empty([dom.shape[0]])
        b[:] = -1
        for i, _ in enumerate(b):
            b[i] = sum([target[k] * polys[i].evaluate(x_k) for k, x_k in enumerate(dom)])


        # Solve LSE using Choleski Decomposition: solve Ga = b
        A = matrix_dot_matrix(matrix_transpose(G), G)   # Make positive definite
        y = matrix_dot_vector(matrix_transpose(G), b)
        chol_d = CholeskiDecomposition()

        return chol_d.solve(A=A, b=y)

    def __str__(self):
        l = ["({:.5})*x^{}".format(c, self.degree-i) for i, c in enumerate(self.coefficients)]
        return "Polynomial:\n    " + " + ".join(l) + "\n"


if __name__ == "__main__":
```

```python
import matplotlib.pyplot as plt

print("\n", end="\n")
print("# ------------ Question 1 ------------ #", end="\n")
print("# ---------- Interpolation ---------- #", end="\n")
print("# ---------- First 6 Points ---------- #", end="\n")
print("# ------------------------------------ #", end="\n\n")
B = np.array([0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
H = np.array([0.0, 14.7, 36.5, 71.7, 121.4, 197.4])
dom, target = B, H
li = LagrangeInterpolator(dom=dom, target=target)
y = li.interpolate()
x_range = np.linspace(0.0, dom[-1], num=10000)
interpolation = [y(x) for x in x_range]
# Perform postprocessing
fig, ax = plt.subplots()
ax.plot(dom, target, 'bo', label="Orignal BH")
ax.plot(x_range, interpolation, 'r', label="Lagrange BH")
legend = ax.legend(loc='best', fontsize='small')
plt.title('HB Interpolation')
plt.ylabel('H')
plt.xlabel('B')
plt.show()
print("# ------------------------------------ #", end="\n\n")


print("\n", end="\n")
print("# ------------ Question 1 ------------ #", end="\n")
print("# ---------- Interpolation ---------- #", end="\n")
print("# -------- 6 Separate Points -------- #", end="\n")
print("# ------------------------------------ #", end="\n\n")
B = np.array([0.0, 1.3, 1.4, 1.7, 1.8, 1.9])
H = np.array([0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2])
dom, target = B, H
li = LagrangeInterpolator(dom=dom, target=target)
y = li.interpolate()
x_range = np.linspace(0.0, dom[-1], num=10000)
interpolation = [y(x) for x in x_range]
# Perform postprocessing
fig, ax = plt.subplots()
ax.plot(dom, target, 'bo', label="Orignal BH")
ax.plot(x_range, interpolation, 'r', label="Lagrange BH")
```

```
    legend = ax.legend(loc='best', fontsize='small')
    plt.title('HB Interpolation')
    plt.ylabel('H')
    plt.xlabel('B')
    plt.show()
    print("# ------------------------------------------------ #", end="\n\n")
```

Listing 1: lagrange_interpolation.py

```
# ------------------------------------------- #
# Lagrange Subdomain Interpolation
# ------------------------------------------- #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: LagrangeSubdomainInterpolator is a class that uses Lagrange
# polynomials to interpolate a data set by minimizing the least
# squares error with respect to the (domain, target) points.

import numpy as np
import matplotlib.pyplot as plt
from utils import *
from choleski import CholeskiDecomposition
from polynomial_collective import LagrangePolynomial

class LagrangeSubdomainInterpolator:

    def __init__(self, dom, target):
        """
        :type dom: ndarray([float])
        :type target: ndarray([float])
        """
        self.dom, self.target = dom, target

        self.polynomials = []
        self.sub_doms = []
        for i, v in enumerate(dom[:-1]):
            sub_dom = dom[i:i+2]
            self.sub_doms.append(sub_dom)
            self.polynomials.append(
                (LagrangePolynomial(0, sub_dom), LagrangePolynomial(1, sub_dom))
            )

        self.degree = self.polynomials[0][0].degree

        self.params = self.determine_model_parameters()

        polys = self.polynomials
        self.coefficients = [
                [self.params[j] * polys[j][0].coefficients[k]
                + self.params[j+1] * polys[j][1].coefficients[k]
                for k in range(self.degree + 1)]
            for j in range(len(self.sub_doms))]

        # print(self.coefficients)
        print(self)

    def interpolate(self):
        """
        :rtype: tuple(ndarray([float]), ndarray([float]))
        """
        a, polys, sub_doms = self.params, self.polynomials, self.sub_doms
        y = []
        for i, sub_dom in enumerate(sub_doms):
            y.append(lambda x, i=i: a[i] * polys[i][0].evaluate(x) + a[i+1] * polys[i][1].evaluate(x))
        return y, sub_doms

    def determine_model_parameters(self):
        """
        :rtype: ndarray([float])
        """
        return self.target

    def determine_sub_domain_index(self, x, sub_doms):
        """
        :type x: float
        :type sub_doms: ndarray([float])
        :rtype: int
        """
        for i, rng in enumerate(sub_doms):
            x_min, x_max = rng[0], rng[1]
            if (x >= x_min) and (x <= x_max):
                return i
        return -1


    def __str__(self):

        r_str = "Polynomial:\n"
        for i, poly_coeff in enumerate(self.coefficients):
            r_str += "  Subdomain " + str(i+1) + ": " + str(self.sub_doms[i]) + "\n   "
            l = ["({:.5})*x^{}".format(c, self.degree-i) for i, c in enumerate(poly_coeff)]
            r_str += " + ".join(l) + "\n"

        return r_str

if __name__ == "__main__":
    print("\n", end="\n")
    print("# ------------ Question 1 ------------ #", end="\n")
    print("# ---------- Interpolation ---------- #", end="\n")
    print("# -------- 6 Separate Points -------- #", end="\n")
    print("# ----------------------------------- #", end="\n\n")
    B = np.array([0.0, 1.3, 1.4, 1.7, 1.8, 1.9])
    H = np.array([0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2])
    dom, target = H, B
    lsi = LagrangeSubdomainInterpolator(dom=dom, target=target)
    y, sub_doms = lsi.interpolate()
    x_range = np.linspace(0.0, dom[-1], num=40000)
    interpolation = []
    for x in x_range:
        indx = lsi.determine_sub_domain_index(x, sub_doms)
        interpolation.append(y[indx](x))
    # Perform postprocessing
    fig, ax = plt.subplots()
    ax.plot(dom, target, 'bo', label="Orignal BH")
    ax.plot(x_range, interpolation, 'r', label="Hermite BH")
    legend = ax.legend(loc='best', fontsize='small')
    plt.title('BH Interpolation')
    plt.ylabel('B')
    plt.xlabel('H')
    plt.show()
    print("# ----------------------------------- #", end="\n\n")
```

Listing 2: lagrange_subdomain_interpolation.py

```
# —————————————————————————————————————————— #
# Hermite Subdomain Interpolation
# —————————————————————————————————————————— #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: HermiteSubdomainInterpolator is a class that uses Hermite
# polynomials to interpolate a data set by minimizing the least
# squares error with respect to the (domain, target) points.

import numpy as np
import matplotlib.pyplot as plt
from utils import *
from choleski import CholeskiDecomposition
from polynomial_collective import HermiteSubdomainPolynomial

class HermiteSubdomainInterpolator:

    def __init__(self, dom, target):
        """
        :type dom: ndarray([float])
        :type target: ndarray([float])
        """
        self.dom, self.target = dom, target
        self.polynomial = HermiteSubdomainPolynomial()

    def interpolate(self):
        """
        :rtype: tuple(list(lambda(float x)), list([float, float]))
        """
        n = 2    # Number of points in subdomain
        sub_doms = []
        poly, dom, target = self.polynomial, self.dom, self.target
        a, b = self.determine_model_parameters()
        y = []
        for i, v in enumerate(dom[:-1]):
            sub_doms.append(dom[i:i+2])
            y.append(lambda x, i=i: \
                    sum([a[i+j] * poly.evaluate_U(x=x, j=j, dom=dom[i:i+2]) \
                    + b[i+j] * poly.evaluate_V(x=x, j=j, dom=dom[i:i+2]) \
                    for j in range(n)]))

```

```python
        return y, sub_doms

    def determine_model_parameters(self):
        """
        :type dom: ndarray([float])
        :type target: ndarray([float])
        :rtype: tuple(ndarray([float]), ndarray([float]))
        """
        dom, target = self.dom, self.target
        a = target
        b = []
        for i, _ in enumerate(target):
            if i == 0:
                b.append((target[i+1] - target[i]) / (dom[i+1] - dom[i]))
            elif i == len(target) - 1:
                b.append((target[i] - target[i-1]) / (dom[i] - dom[i-1]))
            else:
                s1 = (target[i+1] - target[i]) / (dom[i+1] - dom[i])
                s2 = (target[i] - target[i-1]) / (dom[i] - dom[i-1])
                w = s2 / (s1 + s2)
                b.append(w * s1 + (1.0 - w) * s2)

        return a, b

    def determine_sub_domain_index(self, x, sub_doms):
        """
        :type x: float
        :type sub_doms: ndarray([float])
        :rtype: int
        """
        for i, rng in enumerate(sub_doms):
            x_min, x_max = rng[0], rng[1]
            if (x >= x_min) and (x <= x_max):
                return i
        return -1


if __name__ == "__main__":
    print("\n", end="\n")
    print("# ------------ Question 1 ------------ #", end="\n")
    print("# ---------- Interpolation ---------- #", end="\n")
    print("# -------- 6 Separate Points -------- #", end="\n")
    print("# -------------------------------------------------- #", end="\n\n")
    B = np.array([0.0, 1.3, 1.4, 1.7, 1.8, 1.9])
    H = np.array([0.0, 540.6, 1062.8, 8687.4, 13924.3, 22650.2])
    dom, target = B, H
    hsi = HermiteSubdomainInterpolator(dom=dom, target=target)
    y, sub_doms = hsi.interpolate()
    x_range = np.linspace(0.0, dom[-1], num=40000)
    interpolation = []
    for x in x_range:
        indx = hsi.determine_sub_domain_index(x, sub_doms)
        interpolation.append(y[indx](x))
    # Perform postprocessing
    fig, ax = plt.subplots()
    ax.plot(dom, target, 'bo', label="Orignal BH")
    ax.plot(x_range, interpolation, 'r', label="Hermite BH")
    legend = ax.legend(loc='best', fontsize='small')
    plt.title('HB Interpolation')
    plt.ylabel('H')
    plt.xlabel('B')
    plt.show()
    print("# -------------------------------------------------- #", end="\n\n")
```

Listing 3: hermite_subdomain_interpolation.py

```
# ──────────────────────────────────────── #
# Polynomial Collective
# ──────────────────────────────────────── #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: Collection of polynomial classes / data structures.
from polynomial import Polynomial

class LagrangePolynomial:

    def __init__(self, j, dom):
        polynomial = Polynomial()
        for r, x_r in enumerate(dom):
            if r != j:
                polynomial.multiply_binomial(-1.0 * x_r)
        polynomial.divide_scalar(polynomial.evaluate(dom[j]))
        self.polynomial = polynomial
        self.coefficients = polynomial.coefficients
        self.degree = polynomial.degree

    def evaluate(self, x):
        return self.polynomial.evaluate(x)

    def __str__(self):
        l = ["{:.5}x^{}".format(c, self.degree-i) for i, c in enumerate(self.coefficients)]
        return " + ".join(l)


class HermiteSubdomainPolynomial:

    def _lagrange(self, x, j, dom):
        if j == 0:
            return (x - dom[j+1]) / (dom[j] - dom[j+1])
        elif j == 1:
            return (x - dom[j-1]) / (dom[j] - dom[j-1])

    def _lagrange_p(self, j, dom):
        if j== 0:
            return 1.0 / float(dom[j] - dom[j+1])
        elif j == 1:
            return 1.0 / float(dom[j] - dom[j-1])
    def evaluate_U(self, x, j, dom):
        return (1 - 2 * self._lagrange_p(j, dom) * (x - dom[j])) * (self._lagrange(x, j, dom)**2)

    def evaluate_V(self, x, j, dom):
        return (x - dom[j]) * (self._lagrange(x, j, dom) ** 2)
```
Listing 4: polynomial_collective.py

```python
# ---------------------------------------------- #
# Polynomials
# ---------------------------------------------- #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: Polynomial data structure and utils class.

class Polynomial:

    def __init__(self):
        self.degree = 0
        self.coefficients = []

    def multiply_binomial(self, binomial_arg):
        # Lazy instantiation
        if self.degree == 0:
            self.degree += 1
            self.coefficients.append(1.0)
            self.coefficients.append(binomial_arg)

        else:
            self.degree += 1
            self.coefficients.append(0.0)

            temp = []
            for i, v in enumerate(self.coefficients):
                if i == 0:
                    temp.append(self.coefficients[i])
                else:
                    temp.append(v +  binomial_arg * self.coefficients[i-1])
            self.coefficients = temp


    def divide_scalar(self, scalar):
        self.coefficients = [v / scalar for v in self.coefficients]

    def evaluate(self, x):
        val = 0
        for i, c in enumerate(self.coefficients):
            d = self.degree - i
            val += c * (x ** d)
        return val
    def __str__(self):
        l = ["{:.5}x^{}".format(c, self.degree-i) for i, c in enumerate(self.coefficients)]
        return " + ".join(l)
```

Listing 5: polynomial.py

```
# ------------------------------------------ #
# Newton Raphson Solver
# ------------------------------------------ #
# Author: Mido Assran
# Date: Dec. 1, 2016
# Description: NewtonRaphsonSolver solves non-linear equations by using
# the Newton-Raphson method.

import random
import numpy as np
import matplotlib.pyplot as plt
from utils import matrix_dot_matrix, matrix_transpose, matrix_dot_vector
from choleski import CholeskiDecomposition

class NewtonRaphsonSolver:

    def solve(self, starting_guess, f, g, stopping_ratio=1e-6):
        """
        :type starting_guess: float
        :type stopping_ratio: float
        :type f: lambda(float)
        :type g: lambda(float)
        :rtype: dict('num_steps': int, 'arg_history': list(float))
        """
        x = starting_guess
        arg_history = [x]

        progress_ratio = abs(f(x) / f(starting_guess))
        itr = 0
        while progress_ratio > stopping_ratio:
            itr += 1
            x += -1.0 * f(x) / g(x)
            arg_history.append(x)
            progress_ratio = abs(-1.0 * f(x) / f(starting_guess))

        return {'num_steps': itr, 'arg_history': arg_history}


    def solve_2D(self, starting_guess, f, J, stopping_ratio=1e-6):
        """
        :type starting_guess: ndarray([float, float])
        :type stopping_ratio: float
        :type f: list(lambda(float, float))
        :type J: list(lambda(float, float))
        :rtype: dict('num_steps': int,
                     'arg_history': list([float, float]),
                     'error_history': list(float))
        """
        x = starting_guess
        arg_history = [x]
        chol = CholeskiDecomposition()
        error = np.linalg.norm([f[0](x[0], x[1]), f[1](x[0], x[1])])/200e-3
        error_history = []

        print('iteration \t voltages \t\t\t\t\t f \t\t\t\t\t error')

        itr = 0
        while error > stopping_ratio:
            itr += 1
            A = np.empty([2,2])
            A[0,0] = Jacobian[0][0](x[0], x[1])
            A[0,1] = Jacobian[0][1](x[0], x[1])
            A[1,0] = Jacobian[1][0](x[0], x[1])
            A[1,1] = Jacobian[1][1](x[0], x[1])
            b = np.empty([2])
            b[0] = -f[0](x[0], x[1])
            b[1] = -f[1](x[0], x[1])
            b = matrix_dot_vector(matrix_transpose(A), b)
            A = matrix_dot_matrix(matrix_transpose(A), A)
            temp = chol.solve(A=A, b=b)
            x += temp
            arg_history.append(x)
            error = np.linalg.norm([f[0](x[0], x[1]), f[1](x[0], x[1])])/200e-3
            error_history.append(error)
            print(itr, x, b, error, sep='\t\t')


        return {'num_steps': itr, 'arg_history': arg_history, 'error_history': error_history}

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    from lagrange_subdomain_interpolation import LagrangeSubdomainInterpolator

    print("\n", end="\n")
    print("# ------------ Question 1 ------------ #", end="\n")
    print("# ---------- Newton-Raphson ---------- #", end="\n")
    print("# ------------------------------------ #", end="\n\n")
    # Piece-wise Linear Interpolation of data
    B = np.array([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 = np.array([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])
    dom, target = B, H
    lsi = LagrangeSubdomainInterpolator(dom=dom, target=target)
    y, sub_doms = lsi.interpolate()
    x_range = np.linspace(0.0, dom[-1], num=20)
    A = 1e-2 * 1e-2
    h_phi = lambda phi, lsi=lsi, sub_doms=sub_doms, A=A: y[lsi.determine_sub_domain_index(phi/A, sub_doms)](phi/A)
    g_phi = lambda phi, lsi=lsi, sub_doms=sub_doms, A=A: lsi.coefficients[lsi.determine_sub_domain_index(phi/A, sub_doms)][0]/A
    R_g = 0.5e-2 / (1e-4 * 1.25663706e-6)
    L_c = 30e-2
    M = 800.0 * 10.0
    objective = lambda phi, h_phi=h_phi: R_g * phi + L_c * h_phi(phi) - M
    derivative = lambda phi, g_phi=g_phi: R_g + L_c * g_phi(phi)
    xtst = - (objective(0.0))
    nrs = NewtonRaphsonSolver()
    starting_guess = 0.0
    print("nrs.solve(starting_guess=" + str(starting_guess), "):", sep="")
    result = nrs.solve(starting_guess=starting_guess, f=objective, g=derivative)
    print("\n    num_steps: ", result['num_steps'], "     flux: ", result['arg_history'][-1])
    print("# ----------------------------------- #", end="\n\n")


    print("\n", end="\n")
    print("# ------------ Question 1 ------------ #", end="\n")
    print("# ---------- Successive-Sub ---------- #", end="\n")
    print("# ------------------------------------ #", end="\n\n")
    # Piece-wise Linear Interpolation of data
    B = np.array([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 = np.array([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])
    dom, target = B, H
    lsi = LagrangeSubdomainInterpolator(dom=dom, target=target)
    y, sub_doms = lsi.interpolate()
    x_range = np.linspace(0.0, dom[-1], num=20)
    A = 1e-2 * 1e-2
    h_phi = lambda phi, lsi=lsi, sub_doms=sub_doms, A=A: y[lsi.determine_sub_domain_index(phi/A, sub_doms)](phi/A)
    R_g = 0.5e-2 / (1e-4 * 1.25663706e-6)
    L_a = 30e-2
    M = 800.0 * 10.0
    objective = lambda phi: 1e-10 * (R_g * phi + (L_a * h_phi(phi) - M))
    derivative = lambda phi: 1.0
    nrs = NewtonRaphsonSolver()
    starting_guess = 0.0
    print("nrs.solve(starting_guess=" + str(starting_guess), "):", sep="")
    result = nrs.solve(starting_guess=starting_guess, f=objective, g=derivative)
    print("\n   num_steps: ", result['num_steps'], "    flux: ", result['arg_history'][-1])
    print("# ------------------------------------ #", end="\n\n")


    print("\n", end="\n")
    print("# ------------ Question 2 ------------ #", end="\n")
    print("# ---------- Newton-Raphson ---------- #", end="\n")
    print("# ------------------------------------ #", end="\n\n")
    E = 200e-3; kT_q = 25e-3; R = 512; I_sa = 0.8e-6; I_sb = 1.1e-6

    Jacobian = [[0, 0],[0, 0]]
    Jacobian[0][0] = lambda v1, v2: 1.0 + (R * I_sa / kT_q) * np.exp((v1-v2) / kT_q)
    Jacobian[1][0] = lambda v1, v2: (I_sa / kT_q) * np.exp((v1-v2) / kT_q)
    Jacobian[0][1] = lambda v1, v2: - (R * I_sa / kT_q) * np.exp((v1-v2) / kT_q)
    Jacobian[1][1] = lambda v1, v2: - (I_sa / kT_q) * np.exp((v1-v2) / kT_q) - (I_sb / kT_q) * np.exp(v2 / kT_q)

    objective = [0, 0]
    objective[0] = lambda v1, v2: v1 - E + (R * I_sa) * (np.exp((v1-v2) / kT_q) - 1.0)
    objective[1] = lambda v1, v2: I_sa * (np.exp((v1-v2) / kT_q) - 1.0) - I_sb * (np.exp(v2/ kT_q) - 1.0)

    nrs = NewtonRaphsonSolver()
    starting_guess = np.array([0.0, 0.0])
    print("nrs.solve(starting_guess=" + str(starting_guess), "):", sep="", end="\n\n")
    result = nrs.solve_2D(starting_guess=starting_guess, f=objective, J=Jacobian)
    print("\nnum_steps: ", result['num_steps'], "    voltages: ", result['arg_history'][-1])

    n1_error = [1.0 / (2 ** (2 ** (i+1 / 0.9))) for i, e in enumerate(result['error_history'])]
    n_error = [e for i, e in enumerate(result['error_history'])]
    # Perform postprocessing
    fig, ax = plt.subplots()
    ax.semilogy(n_error, 'r', label="Error")
    ax.semilogy(n1_error, '--b', label="Ideal Quadratic Convergence")
    legend = ax.legend(loc='best', fontsize='small')
    plt.show()
    print("# ---------------------------------------- #", end="\n\n")
```

Listing 6: newton_raphson.py

```
# ----------------------------------------------------------- #
# One Point Gauss-Legendre
# ----------------------------------------------------------- #
# Author: Mido Assran
# Date: Dec. 8, 2016
# Description: One Point Gauss-Legendre is a class that uses the one point
# Gauss-Legendre method to perform integration of a function over an
# arbitrary interval by splitting up into sub-intervals, and
# performing a coordinate mapping for compatability
# with the Gauss-Legendre interval.

import matplotlib.pyplot as plt


class OnePointGaussLegendre:

    def integrate(self, function, interval, num_segments):
        """
        :type function: lambda
        :type interval: tuple(float, float)
        :type num_segments: int
        :rtype: float
        """

        sub_interval_width = (interval[1] - interval[0]) / num_segments
        sub_intervals = [(interval[0] + i * sub_interval_width,
                          interval[0] + (i+1) * sub_interval_width)
                          for i in range(num_segments)]

        weights = [sub_interval_width for i in range(num_segments)]
        abscissas = [(x[0] + x[1]) * 0.5 for x in sub_intervals]

        integral = sum([weights[i] * function(abscissas[i])
                        for i in range(num_segments)])

        return integral

    def integrate_uneven(self, function, interval, num_segments):
        """
        :type function: lambda
        :type interval: tuple(float, float)
        :type num_segments: int
        :rtype: float
        """

        # Create uneqal spacings
        scalings = [i for i in range(1, num_segments+1)]
        scalings = [p / sum(scalings) for p in scalings]
        sub_interval_width = (interval[1] - interval[0]) / num_segments
        widths = [sub_interval_width / scalings[i] for i in range(num_segments)][::-1]
        widths = [widths[i] / sum(widths) for i in range(num_segments)]

        # Create sub_intervals
        sub_intervals = []
        running_lower = interval[0]
        for i in range(num_segments):
            l = running_lower
            u = l + widths[i]
            running_lower += widths[i]
            sub_intervals.append((l,u))

        weights = [widths[i] for i in range(num_segments)]
        abscissas = [(x[0] + x[1]) * 0.5 for x in sub_intervals]

        integral = sum([weights[i] * function(abscissas[i])
                        for i in range(num_segments)])

        return integral



if __name__ == "__main__":
    import math

    # print("\n", end="\n")
    # print("# ------------ Question 3 ------------ #", end="\n")
    # print("# ---------- Gauss-Legendre ---------- #", end="\n")
    # print("# ------------------------------------ #", end="\n\n")
    # opgl = OnePointGaussLegendre()
    # f = lambda x: math.sin(x)
    # print("N \t integrate(sin(x)) \t\t error")
    # rng = (0.0, 1.0)
    # truth = 0.45970
    # errors = []
    # n_vector = [i for i in range(1,21)]
    # for N in n_vector:
    #     result = opgl.integrate(function=f, interval=rng, num_segments=N)
    #     error = abs(truth-result)
    #     print(N, result, error, sep='\t')
    #     errors.append(error)
    # n_vector = [math.log(N, 10) for N in n_vector]
    # errors = [math.log(e, 10) for e in errors]
    # fig, ax = plt.subplots()
    # ax.plot(n_vector, errors, 'r', label="Error")
    # # ax.set_yscale('log')
    # # ax.set_xscale('log')
    # # ax.set_xlim([n_vector[0], n_vector[-1]])
    # # ax.set_ylim([errors[-1], errors[0]])
    # legend = ax.legend(loc='best', fontsize='small')
    # plt.title('Gauss-Legendre Error vs Resolution')
    # plt.ylabel('Error')
    # plt.xlabel('N (number of subintervals)')
    # plt.show()
    # print("# ---------------------------------------- #", end="\n\n")
    #
    #
    # print("\n", end="\n")
    # print("# ------------ Question 3 ------------ #", end="\n")
    # print("# ---------- Gauss-Legendre ---------- #", end="\n")
    # print("# ---------------------------------------- #", end="\n\n")
    # opgl = OnePointGaussLegendre()
    # f = lambda x: math.log(x)
    # print("N \t integrate(log(x)) \t\t error")
    # rng = (0.0, 1.0)
    # truth = -1.0
    # errors = []
    # n_vector = [i for i in range(1,21)]
    # for N in n_vector:
    #     result = opgl.integrate(function=f, interval=rng, num_segments=N)
    #     error = abs(truth-result)
    #     print(N, result, error, sep='\t')
    #     errors.append(error)
    # n_vector = [math.log(N, 10) for N in n_vector]
    # errors = [math.log(e, 10) for e in errors]
    # fig, ax = plt.subplots()
    # ax.plot(n_vector, errors, 'r', label="Error")
    # # ax.set_yscale('log')
    # # ax.set_xscale('log')
    # # ax.set_xlim([n_vector[0], n_vector[-1]])
    # # ax.set_ylim([errors[-1], errors[0]])
    # legend = ax.legend(loc='best', fontsize='small')
    # plt.title('Gauss-Legendre Error vs Resolution')
    # plt.ylabel('Error')
    # plt.xlabel('N (number of subintervals)')
    # plt.show()
    # print("# ———————————————————————————— #", end="\n\n")


    print("\n", end="\n")
    print("# —————— Question 3 —————— #", end="\n")
    print("# ————— Gauss-Legendre ————— #", end="\n")
    print("# ——————————————————————————— #", end="\n\n")
    opgl = OnePointGaussLegendre()
    f = lambda x: math.log(0.2 * abs(math.sin(x)))
    print("N \t integrate(log(0.2 |sin(x)|)) \t\t error")
    rng = (0.0, 1.0)
    truth = -2.662
    errors = []
    n_vector = [i for i in range(1,21)]
    for N in n_vector:
        result = opgl.integrate(function=f, interval=rng, num_segments=N)
        error = abs(truth-result)
        print(N, result, error, sep='\t')
        errors.append(error)
    n_vector = [math.log(N, 10) for N in n_vector]
    errors = [math.log(e, 10) for e in errors]
    test = [3*math.log(0.948 / ((N + 1)), 10) for N in n_vector]
    fig, ax = plt.subplots()
    ax.plot(n_vector, errors, 'r', label="Error")
    ax.plot(n_vector, test, 'b', label="Ideal")
    # ax.set_yscale('log')
    # ax.set_xscale('log')
    # ax.set_xlim([n_vector[0], n_vector[-1]])
    # ax.set_ylim([errors[-1], errors[0]])
    legend = ax.legend(loc='best', fontsize='small')
    plt.title('Gauss-Legendre Error vs Resolution')
    plt.ylabel('Error')
    plt.xlabel('N (number of subintervals)')
```

```
plt.show()
print("# ---------------------------------------------- #", end="\n\n")


# print("\n", end="\n")
# print("# --------------- Question 3 --------------- #", end="\n")
# print("# ------------ Gauss-Legendre ------------ #", end="\n")
# print("# ---------------------------------------------- #", end="\n\n")
# opgl = OnePointGaussLegendre()
# f = lambda x: math.log(x)
# print("\t\t\t \033[1m  integrate(log(x)) \t\t error")
# print("\033[0m")
# rng = (0.0, 1.0)
# truth = -1.0
# N = 10
# result_even = opgl.integrate(function=f, interval=rng, num_segments=N)
# error_even = abs(truth-result_even)
# print("Even Spacing:", result_even, error_even, sep="\t\t")
# result_uneven = opgl.integrate_uneven(function=f, interval=rng, num_segments=N)
# error_uneven = abs(truth-result_uneven)
# print("Uneven Spacing:", result_uneven, error_uneven, sep="\t\t")
# print("# ---------------------------------------------- #", end="\n\n")
#
#
#
# print("\n", end="\n")
# print("# --------------- Question 3 --------------- #", end="\n")
# print("# ------------ Gauss-Legendre ------------ #", end="\n")
# print("# ---------------------------------------------- #", end="\n\n")
# opgl = OnePointGaussLegendre()
# f = lambda x: math.log(0.2 * abs(math.sin(x)))
# print("\t\t \033[1m  integrate(log(0.2 |sin(x)|)) \t\t error")
# print("\033[0m")
# rng = (0.0, 1.0)
# truth = -2.662
# N = 10
# result_even = opgl.integrate(function=f, interval=rng, num_segments=N)
# error_even = abs(truth-result_even)
# print("Even Spacing:", result_even, error_even, sep="\t\t")
# result_uneven = opgl.integrate_uneven(function=f, interval=rng, num_segments=N)
# error_uneven = abs(truth-result_uneven)
# print("Uneven Spacing:", result_uneven, error_uneven, sep="\t\t")
```

```
    # print("# ———————————————————————————— #", end="\n\n")
```

Listing 7: gauss_legendre.py

```
# ---------------------------------------------------------- #
# Utils
# ---------------------------------------------------------- #
# Author: Mido Assran
# Date: 5, October, 2016
# Description: Utils provides a cornucopia of useful matrix
# and vector helper functions.

import random
import numpy as np

def matrix_transpose(A):
    """
    :type A: np.array([float])
    :rtype: np.array([floats])
    """

    # Initialize A_T(ranspose)
    A_T = np.empty([A.shape[1], A.shape[0]])

    # Set the rows of A to be the columns of A_T
    for i, row in enumerate(A):
        A_T[:, i] = row

    return A_T


def matrix_dot_matrix(A, B):
    """
    :type A: np.array([float])
    :type B: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != B.shape[0]):
        return None

    A_dot_B = np.empty([A.shape[0], B.shape[1]])
    A_dot_B[:] = 0  # Initialize entries of the new matrix to zero

    B_T = matrix_transpose(B)

    for i, row_A in enumerate(A):
        for j, column_B in enumerate(B_T):
            for k, v in enumerate(row_A):
                A_dot_B[i, j] += v * column_B[k]

    return A_dot_B


def matrix_dot_vector(A, b):
    """
    :type A: np.array([float])
    :type b: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != b.shape[0]):
        return None

    A_dot_b = np.empty([A.shape[0]])
    A_dot_b[:] = 0  # Initialize entries of the new vector to zero

    for i, row_A in enumerate(A):
        for j, val_b in enumerate(b):
            A_dot_b[i]  += row_A[j] * val_b

    return A_dot_b


def vector_to_diag(b):
    """
    :type b: np.array([float])
    :rtype: np.array([float])
    """

    diag_b = np.empty([b.shape[0], b.shape[0]])
    diag_b[:] = 0     # Initialize the entries to zero

    for i, val in enumerate(b):
        diag_b[i, i] = val
```

```python
    return diag_b

def generate_positive_semidef(order, seed=0):
    """
    :type order: int
    :type seed: int
    :rtype: np.array([float])
    """

    np.random.seed(seed)
    A = np.random.randn(order, order)
    A = matrix_dot_matrix(A, matrix_transpose(A))

    # TODO: Replace matrix_rank with a custom function
    from numpy.linalg import matrix_rank
    if matrix_rank(A) != order:
        print("WARNING: Matrix is singular!", end="\n\n")

    return A
```

Listing 8: utils.py

```python
# ------------------------------------------ #
# Choleski Decomposition
# ------------------------------------------ #
# Author: Mido Assran
# Date: 30, September, 2016
# Description: CholeskiDecomposition solves the linear system of equations:
# Ax = b by decomposing matrix A using Choleski factorization and using
# forward and backward substitution to determine x. Matrix A must
# be symmetric, real, and positive definite.

import random
import timeit
import numpy as np
from utils import matrix_transpose

DEBUG = False

class CholeskiDecomposition(object):

    def __init__(self):
        if DEBUG:
            np.core.arrayprint._line_width = 200

    def solve(self, A, b, band=None):
        """
        :type A: np.array([float])
        :type b: np.array([float])
        :type band: int
        :rtype: np.array([float])
        """

        start_time = timeit.default_timer()

        # If the matrix, A, is banded, leverage that!
        if band is not None:
            self._band = band

        # If the matrix, A, is not square, exit
        if A.shape[0] != A.shape[1]:
            return "Matrix 'A' is not square!"

        n = A.shape[1]

        # ------------------------------------------------------------------ #
        # Simultaneous Choleski factorization of A and chol-elimination
        # ------------------------------------------------------------------ #
        # Choleski factorization & forward substitution
        for j in range(n):

            # If the matrix A is not positive definite, exit
            if A[j,j] <= 0:
                return "Matrix 'A' is not positive definite!"

            A[j,j] = A[j,j] ** 0.5    # Compute the j,j entry of chol(A)
            b[j] /= A[j,j]            # Compute the j entry of forward-sub

            for i in range(j+1, n-1):

                # Banded matrix optimization
                if (band is not None) and (i == self._band):
                    self._band += 1
                    break

                A[i,j] /= A[j,j]      # Compute the i,j entry of chol(A)
                b[i] -= A[i,j] * b[j] # Look ahead modification of b

                if A[i,j] == 0:       # Optimization for matrix sparsity
                    continue

                # Look ahead moidification of A
                for k in range(j+1, i+1):
                    A[i,k] -= A[i,j] * A[k,j]

            # Perform computation for the test source
            if (j != n-1):
                A[n-1,j] /= A[j,j]        # Compute source entry of chol(A)
                b[n-1] -= A[n-1,j] * b[j] # Look ahead modification of b
                # Look ahead moidification of A
                for k in range(j+1, n):
                    A[n-1,k] -= A[n-1,j] * A[k,j]
        # ------------------------------------------------------------------ #

        # ------------------------------------------------------------------ #
        # Now solve the upper traingular system
        # ------------------------------------------------------------------ #
        # Transpose(A) is the upper-tiangular matrix of chol(A)
        A[:] = matrix_transpose(A)

        # Backward substitution
        for j in range(n - 1, -1, -1):
            b[j] /= A[j,j]

            for i in range(j):
                b[i] -= A[i,j] * b[j]
        # ------------------------------------------------------------------ #

        elapsed_time = timeit.default_timer() - start_time

        if DEBUG:
            print("Execution time:\n", elapsed_time, end="\n\n")

        # The solution was overwritten in the vector b
        return b

if __name__ == "__main__":
    from utils import generate_positive_semidef, matrix_dot_vector

    order = 10
    seed = 5

    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# ------ Choleski Decomposition ------ #", end="\n")
    print("# -------------------------------------- #", end="\n\n")
    chol_d = CholeskiDecomposition()
    # Create a symmetric, real, positive definite matrix.
    A = generate_positive_semidef(order=order, seed=seed)
    x = np.random.randn(order)
    b = matrix_dot_vector(A=A, b=x)
    print("A:\n", A, end="\n\n")
    print("x:\n", x, end="\n\n")
    print("b (=Ax):\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
    print("result = solve(A, b):\n", v, end="\n\n")
    print("2-norm error:\n", np.linalg.norm(v - x), end="\n\n")
    print("# ---------------------------------------------------------------- #", end="\n\n")
```

Listing 9: choleski.py