final project

姓名： 徐怡
学号： PB19111672

Q1

当$\alpha = 0$时，对(1)采用积分格式：

$$
\begin{aligned}
Z(u_1, \ldots, u_n) &= min_{u_1,\ldots,u_{n-1}} \sum_{i=1}^{n} \frac{1}{2}(\frac{u_i - u_{i-1}}{h})^2 h - \sum_{i=1}^{n-1} f_i u_i h \\
&= min_{u_1,\ldots,u_{n-1}} \sum_{i=1}^{n} \frac{1}{2h}(u_i^2 + u_{i-1}^2 - 2u_i u_{i-1}) - \sum_{i=1}^{n-1} f_i u_i h \\
&= min_{u_1,\ldots,u_{n-1}} \sum_{i=1}^{n-1} \frac{1}{h}(u_i^2 - u_i u_{i-1}) - \sum_{i=1}^{n-1} f_i u_i h \\
&= min_{u_1,\ldots,u_{n-1}} \sum_{i=1}^{n-1} \frac{1}{h}(u_i^2 - u_i u_{i-1} - f_i u_i h^2)
\end{aligned}
$$

上面的推导过程用到了题中所给的$u_0 = u_n = 0$这个条件。
对Z求偏导：

$$
\begin{aligned}
&\frac{\partial Z(u_1, \ldots, u_n)}{\partial u_i} = \frac{1}{h}(2u_i - u_{i-1} - u_{i+1} - f_i h^2) = 0 \\
&\frac{1}{h^2}(2u_i - u_{i-1} - u_{i+1}) = f_i
\end{aligned}
$$

所以线性方程组$A_h u_h = f_h$对应的系数矩阵A_h：

$$
A_h = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \ldots & 0 & 0 \\ -1 & 2 & -1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}
$$

Q2

题目

当$f(x) = \pi 2 sin(\pi x), n = 10, 20, 40, 80, 160$时，分别利用Jacobi和Gauss-Seidel迭代法求 解$A_h u_h = f_h$（迭代法的终止准则$\varepsilon = 10^{-10}$），并比较u_h与精确解$u_e(x) = sin(\pi x)$之间的 误差$e_h = \|u_h - u_e\|^2$，记录在一张表中。

Jacobi 迭代矩阵

$$
\begin{aligned}
& AX = (D + A - D)X = b \\
& D = diag\{a_{11}, \ldots, a_{nn}\} \\
& X^{(k+1)} = RX^{(k)} + g \\
& R = I - D^{-1}A,\ g = D^{-1}b
\end{aligned}
$$

Gauss-Seidel 迭代矩阵

$$
\begin{aligned}
& A = D + L + U \\
& X^{(k+1)} = SX^{(k)} + f \\
& S = -(D + L)^{-1}U,\ f = (D + L)^{-1}b
\end{aligned}
$$

迭代结果

n	Jacobi's error	Gauss-Seidel's error
10	0.018482034928874548	0.018482036208808273
20	0.006510206868035524	0.006510214971561621
40	0.0022995999225434473	0.002299646308961444
80	0.0008129789948976478	0.0008132435845553491
160	0.00028818334882278225	0.0002896823546997506

Q3

最小二乘法

是一种多项式拟合的思想。
先用$log()$函数对原问题进行转换：

$$
\begin{aligned}
& e_h = ah^{\beta} \\
& \log(e_h) = \log(a) + \beta \log(h)
\end{aligned}
$$

设$c_0 = \log(a), c_1 = \beta, x = \log(h), y\log(e_h)$，则$y = c_0 + c_1 x$。
由书 P50 的公式：

$$c_0 = \frac{(\sum_{i=1}^m x_i^2)(\sum_{i=1}^m y_i) - (\sum_{i=1}^m x_i)(\sum_{i=1}^m x_i y_i)}{m\sum_{i=1}^m x_i^2 - (\sum_{i=1}^m x_i)^2}$$
$$c_1 = \frac{m\sum_{i=1}^m x_i y_i - (\sum_{i=1}^m x_i)(\sum_{i=1}^m y_i)}{m\sum_{i=1}^m x_i^2 - (\sum_{i=1}^m x_i)^2}$$

计算结果

Jacobi's c_0	Gauss-Seidel's c_0	Jacobi's $c_1(\beta)$	Gauss-Se $c_1(\beta$
-0.5373672408101402	-0.5419548136517346	1.5007399046737264	1.49919619

Q4

迭代次数：

n	Jacobi iteration times	Gauss-Seidel iteration times
10	408	195
20	1539	729
40	5727	2697
80	21127	9891
160	77331	35970

由上表可知，Gauss-Seidel 迭代比 Jacobi 迭代所用的迭代次数少，算法效率更高。

Q5

当$\alpha = 1$时，对(1)采用积分格式：

$$
\begin{aligned}
Z(u_1,\dots,u_n) &= min_{u_1,\dots,u_{n-1}} \sum_{i=1}^{n} \frac{1}{2}(\frac{u_i-u_{i-1}}{h})^2 h + \sum_{i=1}^{n-1}(\frac{1}{4}u_i^4 - f_i u_i)h \\
&= min_{u_1,\dots,u_{n-1}} \sum_{i=1}^{n} \frac{1}{2h}(u_i^2+u_{i-1}^2-2u_iu_{i-1}) + \sum_{i=1}^{n-1}(\frac{1}{4}u_i^4 - f_i u_i)h \\
&= min_{u_1,\dots,u_{n-1}} \sum_{i=1}^{n-1} \frac{1}{h}(u_i^2-u_iu_{i-1}) + \sum_{i=1}^{n-1}(\frac{1}{4}u_i^4 - f_i u_i)h \\
&= min_{u_1,\dots,u_{n-1}} \sum_{i=1}^{n-1} \frac{1}{h}(u_i^2-u_iu_{i-1}+\frac{1}{4}u_i^4h^2 - f_iu_ih^2)
\end{aligned}
$$

上面的推导过程用到了题中所给的$u_0=u_n=0$这个条件。
对Z求偏导:

$$
\frac{\partial Z(u_1,\dots,u_n)}{\partial u_i} = \frac{1}{h}(2u_i-u_{i-1}-u_{i+1}+u_i^3h^2-f_ih^2)=0
$$
$$
\frac{2}{h^2}u_i - \frac{1}{h^2}u_{i-1} - \frac{1}{h^2}u_{i+1} + u_i^3 = f_i
$$

所以对应的非线性方程组为:

$$
\begin{cases}
\frac{2}{h^2}u_1 - \frac{1}{h^2}u_2 + u_1^3 = f_1 \\
\frac{2}{h^2}u_2 - \frac{1}{h^2}u_1 - \frac{1}{h^2}u_3 + u_2^3 = f_2 \\
\cdots\cdots \\
\frac{2}{h^2}u_{n-2} - \frac{1}{h^2}u_{n-3} - \frac{1}{h^2}u_{n-1} + u_{n-2}^3 = f_{n-2} \\
\frac{2}{h^2}u_{n-1} - \frac{1}{h^2}u_{n-2} + u_{n-1}^3 = f_{n-1}
\end{cases}
$$

Q6

Newton 迭代 - 非线性方程组

用 Newton 迭代法求解[Q5]中的非线性方程组。
记$X=(x_1,x_2,\dots,x_n)^T, G(x)=(g_1(x),g_2(x),\dots,g_n(x))^T$，则非线性方程组为:

$$
\begin{cases}
g_1(x_1,\dots,x_n) = 2u_1 - u_2 + u_1^3h^2 - f_1h^2 = 0 \\
g_2(x_1,\dots,x_n) = 2u_2 - u_1 - u_3 + u_2^3h^2 - f_2h^2 = 0 \\
\cdots\cdots \\
g_{n-2}(x_1,\dots,x_n) = 2u_{n-2} - u_{n-3} - u_{n-1} + u_{n-2}^3h^2 - f_{n-2}h^2 = 0 \\
g_{n-1}(x_1,\dots,x_n) = 2u_{n-1} - u_{n-2} + u_{n-1}^3h^2 - f_{n-1}h^2 = 0
\end{cases}
$$

Jacobi 矩阵为：

$$
J(X) = \begin{pmatrix}
2+3u_1^2h^2 & -1 & 0 & \dots & 0 & 0 \\
-1 & 2+3u_2^2h^2 & -1 & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -1 & 2+3u_{n-1}^2h^2
\end{pmatrix}
$$

迭代计算公式：

$$
J(X^{(k)})\Delta X^{(k)} = -G(X^{(k)}) \\
X^{(k+1)} = X^{(k)} + \Delta X^{(k)}
$$

计算结果

n	Newton iteration times	error
10	5	0.015018032824902102
20	5	0.005300889459115937
40	5	0.0018733686037394085
80	4	0.0006622669330658813
160	4	0.00023414062993914428

仿照[Q3]，用最小二乘法拟合迭代的收敛阶：

$$
e_h = ah^\beta \\
\log(e_h) = \log(a) + \beta\log(h)
$$

设$c_0 = \log(a), c_1 = \beta, x = \log(n) = -log(h), y\log(e_h)$，则$y = c_0 - c_1x$。
求解结果：
$c_0 = -0.7436269032632109, -c_1 = -1.5007103196269929$
所以$\beta = 1.5007103196269929$.

Appendix -- source code

```python
from math import pi, sin, log, pow
import numpy as np
# numpy has many methods on matrices,
# so that I don't need to write them by myself


list_jacob_error_ln = []
# stores the error (log form) in Jacob iter: 2-norm(uh-ue)


def Jacobi_Iter(A, b, ue, e=1e-10):
    # Jacobi iteration functions
    '''
    A = D + A - D, where D is diagonal of A
    x2 = R*x1 + g
    R = I - D^-1 * A
    g = D^-1 * b
    '''
    # initialization
    n = np.shape(A)[0]  # get the size of A: nxn
    I = np.matrix(np.identity(n))  # get an identity matrix
    D = np.matrix(np.zeros((n, n)))
    # get content of D
    for i in range(n):
        D[i, i] = A[i, i]  # D is the diagonal matrix
    # calculate iteration matrix
    R = I - (D.getI() * A)
    g = D.getI() * b
    x1 = np.matrix(np.zeros((n, 1)))
    x2 = np.matrix(np.ones((n, 1)))
    # x1 and x2 are the iteration variable
    iter_times = 0
    while abs(np.max(x1-x2)) > e:
        # np.max can give infinity norm of a vector (max element of it
        x1 = x2
        x2 = R * x1 + g
```

```
        iter_times = iter_times + 1
    # after iteration, converged
    error = np.linalg.norm(x2 - ue)
    list_jacob_error_ln.append(log(error))
    print(f"Jacobi iteration: {iter_times}, error: {error}")


list_gauss_error_ln = []
# stores the error (log form) in Gauss-Seidel iter: 2-norm(uh-ue)


def Gauss_Seidel_Iter(A, b, ue, e=1e-10):
    # Gauss-Seidel iteration function
    '''
    A = D + L + U, where D is diagonal of A,
    L is down triangle of A, U is up triangle of A
    x2 = S*x1 + f
    S = -(D + L)^-1 * U
    f = (D + L)^-1 * b
    '''
    # initialization
    n = np.shape(A)[0]  # get the size of A: nxn
    D = np.matrix(np.zeros((n, n)))
    L = np.matrix(np.zeros((n, n)))
    U = np.matrix(np.zeros((n, n)))
    # get content of D, L, U
    for i in range(n):
        for j in range(n):
            if (i == j):  # diagonal
                D[i, j] = A[i, j]
            elif (i < j):  # up triangle
                U[i, j] = A[i, j]
            else:  # i > j, down triangle
                L[i, j] = A[i, j]
    # calculate iteration matrix
    DL_inverse = (D + L).getI()
    S = - (DL_inverse * U)
    f = DL_inverse * b
    x1 = np.matrix(np.zeros((n, 1)))
```

```python
    x2 = np.matrix(np.ones((n, 1)))
    # x1 and x2 are the iteration variable
    iter_times = 0
    while abs(np.max(x1-x2)) > e:
        # np.max can give infinity norm of a vector (max element of it
        x1 = x2
        x2 = S * x1 + f
        iter_times = iter_times + 1
    # after iteration, converged
    error = np.linalg.norm(x2 - ue)
    list_gauss_error_ln.append(log(error))
    print(f"Gauss iteration: {iter_times}, error: {error}")
    return 0


list_h_ln = []
# stores every h (in log form)


def Q2():
    # Problem 2
    def f(x):
        return pi*pi*sin(pi*x)

    def u_precise(x):
        # precise solution of u(x)
        return sin(pi*x)
    for n in [10, 20, 40, 80, 160]:
        print("n = ", n)
        # initialization
        A = np.matrix(np.zeros((n-1, n-1)))
        # A is a matrix with every element to be 0, shape of (n-1)x(n-
        b = np.matrix(np.zeros((n-1, 1)))
        ue = np.matrix(np.zeros((n-1, 1)))
        h = 1/n
        list_h_ln.append(log(h))
        for i in range(n-1):
            if i == 0:  # the first row
                A[0, 1] = -1/h/h
```

```
            elif i == n-2:  # the last row
                A[i, i-1] = -1/h/h
            else:  # rows in the middle
                A[i, i-1] = -1/h/h
                A[i, i+1] = -1/h/h
            A[i, i] = 2/h/h
            b[i, 0] = f((i+1)*h)
            ue[i, 0] = u_precise((i+1)*h)
        # iteration
        Jacobi_Iter(A, b, ue, 1e-10)
        Gauss_Seidel_Iter(A, b, ue, 1e-10)


def Least_Squares_Method(x: list, y: list):
    '''
    use ln() to transform h and eh into x and y
    use the formulus on page 50
    '''
    # some preparation
    m = len(x)
    x_array = np.array(x)
    y_array = np.array(y)
    x_sum = np.sum(x_array)
    y_sum = np.sum(y_array)
    x_sqrt_sum = sum([x_*x_ for x_ in x])  # \sum x^2
    x_y_sum = sum([x[i]*y[i] for i in range(len(x))])  # \sum x*y
    x_sum_sqrt = pow(x_sum, 2)
    # calculate c0, c1
    denominator = m * x_sqrt_sum - x_sum_sqrt
    c0 = (x_sqrt_sum * y_sum - x_sum * x_y_sum) / denominator
    c1 = (m * x_y_sum - x_sum * y_sum) / denominator
    return c0, c1


def Q3():
    '''
    call least squares method to calculate for
    Jacobi and Gauss-Seidel iteration separately
    '''
```

```
    c0_J, c1_J = Least_Squares_Method(list_h_ln, list_jacob_error_ln)
    print(f"Jacobi: c0 = {c0_J}, c1 = {c1_J}")
    c0_G, c1_G = Least_Squares_Method(list_h_ln, list_gauss_error_ln)
    print(f"Gauss-Seidel: c0 = {c0_G}, c1 = {c1_G}")


list_newton_error_ln = []
# stores the error (log form) in Newton iter: 2-norm(uh-ue)



def Newton_Iter(e=1e-8):
    def f(x):
        return pi*pi*sin(pi*x) + pow(sin(pi*x), 3)

    def u_precise(x):
        # precise solution of u(x)
        return sin(pi*x)

    def Create_G(X, h):
        # given last time X, create G(X) array
        n = len(X)
        G = np.matrix(np.zeros((n, 1)))
        for i in range(n):
            G[i, 0] = 2 * X[i, 0] + pow(X[i, 0], 3) * pow(h, 2) \
                - f((i+1)*h) * pow(h, 2)
            if (i != 0):
                G[i, 0] -= X[i-1, 0]
            if (i != n-1):
                G[i, 0] -= X[i+1, 0]
        return G

    list_n_ln = []  # stores every n (in log form)
    for n in [10, 20, 40, 80, 160]:
        print("n = ", n)
        list_n_ln.append(log(n))
        # initialize Jacobi matrix and u_precise
        h = 1/n
        J = np.matrix(np.zeros((n-1, n-1)))
        ue = np.matrix(np.zeros((n-1, 1)))
```

```python
        for i in range(n-1):
            if (i != 0):
                J[i, i-1] = -1
            if (i != n-2):
                J[i, i+1] = -1
            J[i, i] = 2
            ue[i, 0] = u_precise((i+1)*h)
        # x1 and x2 are the iteration variable
        X = np.matrix(np.zeros((n-1, 1)))
        dx = np.matrix(np.ones((n-1, 1)))
        iter_times = 0
        while abs(np.max(dx)) > e:
            # update Jacobi matrix
            for i in range(n-1):
                J[i, i] = 2 + 3*pow(X[i, 0]*h, 2)
            G = Create_G(X, h)
            # solve function J*dx=-G(X)
            dx = -J.getI() * G
            X += dx
            iter_times += 1
        # after iteration, converged
        error = np.linalg.norm(X - ue)
        list_newton_error_ln.append(log(error))
        print(f"Newton iteration: {iter_times}, error: {error}")

    # least squares method -> convergence order
    c0_N, c1_N = Least_Squares_Method(list_n_ln, list_newton_error_ln)
    print(f"Newton: c0 = {c0_N}, c1 = {c1_N}")


def Q6():
    Newton_Iter(1e-8)


if __name__ == "__main__":
    # Q2()
    # Q3()
    Q6()
```