MA 8463 - Homework 6

Jacob Kutch

(6.1) Verify that the overall truncation error for the FD scheme (6.14) is second-order in h_x

Define the term $K(x) = a(x)\frac{u_{xxx}(x)}{3!}\left(\frac{h_x}{2}\right)^2 + \dots$ so that the truncation error of $(au_x)_{i+\frac{1}{2}} - (au_x)_{i-\frac{1}{2}}$ is

$$K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}}) = h_x K'(x_i) + \dots$$

To find the truncation error of

$$-(au_x)_x(x_i) \approx \frac{1}{h_x^2}\left(-a_{i-\frac{1}{2}}u_{i-1} + (a_{i-\frac{1}{2}} + a_{i+\frac{1}{2}})u_i - a_{i+\frac{1}{2}}u_{i+1}\right),$$ let

$$(au_x)_x(x_i) \approx \frac{1}{h_x}\left(\underbrace{(au_x)_{i+\frac{1}{2}} - (au_x)_{i-\frac{1}{2}}}_{\text{using approximations}}\right) + O(h_x^2)$$

$$(au_x)_x(x_i) \approx \frac{1}{h_x}\left[a_{i+\frac{1}{2}}\left(\frac{u_{i+1}-u_i}{h_x}\right) - a_{i+\frac{1}{2}}\frac{u_{xxx}(x_{i+\frac{1}{2}})}{3!}\left(\frac{h_x}{2}\right)^2 + \dots\right.$$

$$\left. - a_{i-\frac{1}{2}}\left(\frac{u_i - u_{i-1}}{h_x}\right) + a_{i-\frac{1}{2}}\left(\frac{u_{xxx}(x_{i-\frac{1}{2}})}{3}\left(\frac{h_x}{2}\right)^2 + \dots\right)\right]$$

$$= \frac{1}{h_x}\left[a_{i+\frac{1}{2}}\left(\frac{u_{i+1}-u_i}{h_x}\right) - K(x_{i+\frac{1}{2}}) - a_{i-\frac{1}{2}}\left(\frac{u_i - u_{i-1}}{h_x}\right) + K(x_{i-\frac{1}{2}})\right]$$

$$= \frac{1}{h_x}\left[\frac{1}{h_x}\left(a_{i+\frac{1}{2}}(u_{i+1}-u_i) - a_{i-\frac{1}{2}}(u_i - u_{i-1})\right) - \left(K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right)\right]$$

$$-(au_x)_x(x_i) \approx \frac{1}{h_x^2}\left(a_{i-\frac{1}{2}}(u_i - u_{i-1}) - a_{i+\frac{1}{2}}(u_{i+1} - u_i)\right) + \frac{1}{h_x}\left(K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right)$$

$$-(au_x)_x(x_i) \approx \frac{1}{h_x^2}\left(a_{i-\frac{1}{2}}(u_i - u_{i-1}) - a_{i+\frac{1}{2}}(u_{i+1} - u_i)\right) + \frac{1}{h_x}\left(K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right)$$

$$\frac{1}{h_x^2}\left[-a_{i-\frac{1}{2}}u_{i-1} + a_{i-\frac{1}{2}}u_i + a_{i+\frac{1}{2}}u_i - a_{i+\frac{1}{2}}u_{i+1}\right] + \frac{1}{h_x}\left(K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right]$$

$$\frac{1}{h_x^2}\left[-a_{i-\frac{1}{2}}u_{i-1} + (a_{i-\frac{1}{2}} + a_{i+\frac{1}{2}})u_i - a_{i+\frac{1}{2}}u_{i+1}\right] + \underbrace{\frac{1}{h_x}\left[K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right]}$$

Now, examining truncate terms defined by $K(x)$,

$$\frac{1}{h_x}\left[K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})\right] = \frac{1}{h_x}\left[h_x K'(x_i) + \dots\right]$$

$$\frac{1}{h_x}\left[a_{i+\frac{1}{2}}\left(\frac{u_{xxx}(x_{i+\frac{1}{2}})}{3!}\right)\left(\frac{h_x}{2}\right)^2 + \dots - a_{i-\frac{1}{2}}\left(\frac{u_{xxx}(x_{i-\frac{1}{2}})}{3!}\right)\left(\frac{h_x}{2}\right)^2 - \dots\right]$$

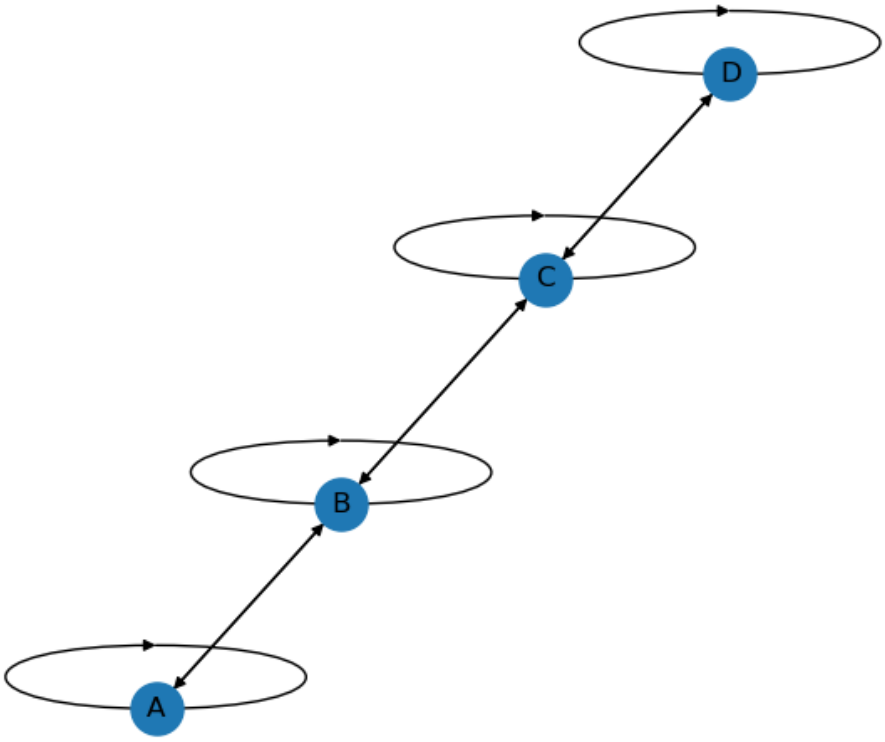
$$\frac{h_x}{24}\left(a_{i+\frac{1}{2}}u_{xxx}(x_{i+\frac{1}{2}}) + \dots - a_{i-\frac{1}{2}}u_{xxx}(x_{i-\frac{1}{2}}) - \dots\right)$$

$$\frac{h_x^2}{24}\left(\frac{a_{i+\frac{1}{2}}u_{xxx}(x_{i+\frac{1}{2}}) - a_{i-\frac{1}{2}}u_{xxx}(x_{i-\frac{1}{2}}) + \dots}{h_x}\right) = \frac{h_x^2}{24}\left(\frac{(au_{xxx})_{i+\frac{1}{2}} - (au_{xxx})_{i-\frac{1}{2}} + \dots}{h_x}\right)$$

$$\frac{h_x^2}{24}\left((au_{xxx})_x(x_i)\right) \rightarrow O(h_x^2) \qquad \square$$

Problem 2:

Output:



1 Directed Graph constructed from Matrix A

A is irreducibly diagonally-dominant.

10 iterations of Jacobi method:

```
[[ 0.        0.        0.        0.      ]
 [-1.       -1.       -1.        7.      ]
 [-2.       -2.        2.        6.5     ]
 [-3.       -1.        1.25      8.      ]
 [-2.       -1.875     2.5       7.625   ]
 [-2.875    -0.75      1.875     8.25    ]
 [-1.75     -1.5       2.75      7.9375  ]
 [-2.5      -0.5       2.21875   8.375   ]
 [-1.5      -1.140625  2.9375    8.109375]
 [-2.140625 -0.28125   2.484375  8.46875 ]
 [-1.28125  -0.828125  3.09375   8.2421875]]
```

10 iterations of Gauss-Seidel method:

```
[[ 0.        0.        0.        0.      ]
 [-1.       -1.5      -1.75      6.125   ]
 [-2.5      -3.125     0.5       7.25    ]
 [-4.125    -2.8125    1.21875   7.609375]
 [-3.8125   -2.296875  1.65625   7.828125]
```

[-3.296875 -1.8203125 2.00390625 8.00195312]
[-2.8203125 -1.40820312 2.296875 8.1484375]
[-2.40820312 -1.05566406 2.54638672 8.27319336]
[-2.05566406 -0.75463867 2.75927734 8.37963867]
[-1.75463867 -0.49768066 2.940979 8.4704895]
[-1.49768066 -0.27835083 3.09606934 8.54803467]]

optimal omega found by binary search: 1.4996141510130132
corresponding solution to optimal omega:
[-0.32380018 0.4647658 3.64461159 8.81552243]
infinity norm of this solution: 0.5352341959331475
random omega from uniform interval (0,2): 1.5282991903474765
corresponding solution to random omega:
[-0.10233015 0.48211134 3.67408629 8.82518825]
infinity norm of this solution: 0.5178886585608659

spectral radius of Jacobi iteration matrix: 2.923880
spectral radius of Gauss-Seidel iteration matrix: 0.500000
spectral radius of SOR iteration matrix: 0.799815

Code:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from copy import deepcopy

# a class is probably unnecessary but I'm fitting a lot into this one file
class IDD_test(object):
    def __init__(self, A):
        self.A = A
        self.A_shape = A.shape

    def make_dir_graph(self):
        rows, cols = np.where(self.A != 0)
        edges = zip(list(rows), list(cols))
        self.graph = nx.DiGraph()
        self.graph.add_edges_from(edges)

    def draw_graph(self):
        if not hasattr(self, 'graph'):
            self.make_dir_graph()
        node_labels = dict(enumerate([chr(65+i) for i in
range(self.A_shape[0])]))
```

```
        nx.draw(self.graph, node_size=500, labels = node_labels)
        plt.show()

    def is_reducible(self):
        self.make_dir_graph()
        return nx.is_strongly_connected(self.graph)

    def is_diag_dominant(self):
        m = self.A.shape[0]
        Lambda = np.zeros(m)
        dominance_test = np.zeros(m, dtype=bool)
        strictness_test = np.zeros(m, dtype=bool)
        for i in range(m):
            Lambda[i] = np.linalg.norm(self.A[i], ord=0) - np.abs(self.A[i,i])
            dominance_test[i] = (np.abs(self.A[i,i]) >= Lambda[i])
            strictness_test[i] = (np.abs(self.A[i,i]) > Lambda[i])
        return (sum(dominance_test) == m) and (sum(strictness_test) > 0)

    def is_IDD(self):
        return self.is_reducible() and (self.is_diag_dominant())

class RelaxationMethods(object):
    def __init__(self, A, b):
        self.A = A
        self.b = b
        self.A_shape = A.shape
        if (self.A_shape[0] != self.A_shape[1]):
            raise Exception(f'ERROR: A is non-square with dimensions
{self.A_shape[0]}x{self.A_shape[1]}! \nRelaxation methods require square
matrices.')
        if (len(self.A_shape) != 2):
            raise Exception(f'ERROR: A is of incorrect shape {self.A_shape}!
\nRelaxation methods require two-dimensional matrices.')
        self.D = np.diag(np.diag(A))
        self.E = -np.tril(A)
        self.F = -np.triu(A)
        self.relax_methods = ['Jacobi', 'GaussSeidel', 'SOR']

    def relaxation(self, init_x, method='Jacobi', omega=None, max_iter=100,
TOL=10e-8):
        if method not in self.relax_methods:
            raise Exception(f'ERROR: unspecified method \'{method}\' given in
call to relaxation! \nChoose from {self.relax_methods}')
        x = [init_x]
        k = 0
```

```
        stop_flag = False
        while (k < max_iter) and not stop_flag:
            if (k > 0 and np.linalg.norm(x[k]-x[k-1], ord=np.inf) < TOL):
                stop_flag = True
                break
            if omega is None:
                x.append(getattr(RelaxationMethods, method)(self, x[k]))
            else:
                x.append(getattr(self, method)(x[k], omega))
            k += 1
        return np.array(x)

    def get_spectral_radius(self, method='Jacobi', omega=None):
        if method not in self.relax_methods:
            raise Exception(f'ERROR: unspecified method \'{method}\' given in
call to relaxation! \nChoose from {self.relax_methods}')
        # spectral radius function
        def rho(T):
            eigvals, eigvec = np.linalg.eig(T)
            return(max(np.abs(eigvals)))

        if method == 'Jacobi':
            return rho(np.linalg.inv(self.D) @ (self.E + self.F))
        elif method == 'GaussSeidel':
            return rho(np.linalg.inv(self.D-self.E) @ self.F)
        elif method == 'SOR':
            return rho(np.linalg.inv(self.D-omega*self.E) @ ((1-omega)*self.D +
omega*self.F))

    def Jacobi(self, x_prev):
        xk = np.zeros(self.A_shape[1])
        for i in range(self.A_shape[1]):
            xk[i] = (1/self.A[i,i])*(self.b[i] - np.dot(self.A[i,:i], x_prev[:i])
- np.dot(self.A[i,(i+1):], x_prev[(i+1):]))
        return xk

    def GaussSeidel(self, x_prev):
        xk = np.zeros(self.A_shape[1])
        for i in range(self.A_shape[1]):
            xk[i] = (1/self.A[i,i])*(self.b[i] - np.dot(self.A[i,:i], xk[:i]) -
np.dot(self.A[i,(i+1):], x_prev[(i+1):]))
        return xk

    def SOR(self, x_prev, omega):
        if (omega < 0) or (omega > 2):
```

```
            raise Exception('ERROR: invalid value {omega} given for omega.
\nomega must be in the open interval (0,2)')
        x_GS = self.GaussSeidel(x_prev)
        return (1-omega)*x_prev + omega*x_GS

    # binary search for solution
    def get_optimal_omega(self, x0, exact_sol, iterations=10, min_int_size=10e-
4):
        interval = [0,2]
        solutions = [] # solutions found for each omega
        # while the size of the most recent subinterval is less than some TOL
        while(np.abs(interval[1]-interval[0]) > min_int_size):
            # pivot point
            mid_point = (interval[1]+interval[0])/2
            # subintervals separated by pivot point
            subintervals = [[interval[0], mid_point], [mid_point, interval[1]]]
            # random values chosen uniformly on each subinterval and
corresponding solutions
            test_omega = [np.random.uniform(*subintervals[0]),
np.random.uniform(*subintervals[1])]
            test_sol = [self.relaxation(x0, method='SOR', omega=test_omega[0],
max_iter=iterations)[-1],
                        self.relaxation(x0, method='SOR', omega=test_omega[1],
max_iter=iterations)[-1]]
            # compare error in each test solution then get index of smallest
error
            errors = [np.linalg.norm(exact_sol-sol) for sol in test_sol]
            min_idx = np.argmin(errors)
            # save omega and corresponding solution that gave smallest error
            solutions.append((test_omega[min_idx], test_sol[min_idx]))
            # update interval to be the subinterval that produced the smallest
error
            interval = subintervals[min_idx]
        # return most recent solution
        return solutions[-1]


# using Example 6.17 in the lecture note as a ground truth
def test_Jacobi():
    A = np.array([[2,-1,0],[-1,2,-1],[0,-1,2]])
    b = np.array([1,0,5])
    x0 = np.array([1,1,1])
    test_obj = RelaxationMethods(A, b)
    # assert np.allclose(test_obj.relaxation(x0, max_iter=3)[-1], [3/2, 2, 7/2])
```

```
if __name__ == '__main__':
    A = np.array([[2,-2,0,0],
                  [-1,2,-1,0],
                  [0,-1,2,-1],
                  [0,0,-1,2]])
    b = np.array([-2,-2,-2,14])
    exact_sol = np.array([0,1,4,9])
    x0 = np.array([0,0,0,0])
    ##########################################################################
####################
    # part (a) - determine if A is irreducibly diagonally-dominant
    truth_list = ['is not', 'is']
    IID_obj = IDD_test(A)
    IID_obj.draw_graph()
    print(f'A {truth_list[int(IID_obj.is_IDD())]} irreducibly diagonally-
dominant.\n\n')
    ##########################################################################
####################
    # part (b) - perform 10 iterations of Jacobi and Gauss-Seidel Methods at x0
    relax_obj = RelaxationMethods(A, b)
    Jacobi_sol = relax_obj.relaxation(x0, method='Jacobi', max_iter=10)
    GS_sol = relax_obj.relaxation(x0, method='GaussSeidel', max_iter=10)
    print(f'10 iterations of Jacobi method:\n {Jacobi_sol}')
    print(f'10 iterations of Gauss-Seidel method:\n {GS_sol}\n\n')
    ##########################################################################
####################
    # part (c) - find best omega and compare with random omega
    best_omega, best_sol = relax_obj.get_optimal_omega(x0, exact_sol,
iterations=10)
    print(f'optimal omega found by binary search: {best_omega}')
    print(f'corresponding solution to optimal omega: \n{best_sol}')
    print(f'infinity norm of this solution: {np.linalg.norm(best_sol-exact_sol,
ord=np.inf)}')
    random_omega = np.random.uniform(0,2)
    rand_SOR_sol = relax_obj.relaxation(x0, method='SOR', omega = random_omega,
max_iter=10)
    print(f'random omega from uniform interval (0,2): {random_omega}')
    print(f'corresponding solution to random omega: \n{rand_SOR_sol[-1]}')
    print(f'infinity norm of this solution: {np.linalg.norm(rand_SOR_sol[-1]-
exact_sol, ord=np.inf)}\n\n')
    ##########################################################################
####################
    # part (d) - display spectral radii of iteration matrices for each method
    print('spectral radius of Jacobi iteration matrix: %f' %
relax_obj.get_spectral_radius(method='Jacobi'))
```

```
    print('spectral radius of Gauss-Seidel iteration matrix: %f' %
relax_obj.get_spectral_radius(method='GaussSeidel'))
    print('spectral radius of SOR iteration matrix: %f' %
relax_obj.get_spectral_radius(method='SOR', omega=best_omega))
```

Problem 3:
Will be submitted late