Homework set 4

Before you turn this problem in, make sure everything runs as expected (in the menubar, select Kernel → Restart Kernel and Run All Cells...).

Please **submit this Jupyter notebook through Canvas** no later than **Mon Nov. 27, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

Exercise 0

Write down the names + student ID of the people in your group.

Zijian Zhang, 14851598

Lina Xiang, 14764369

About imports

Please import the needed packages by yourself.

Sparse matrices

A *sparse matrix* or *sparse array* is a matrix in which most of the elements are zero. There is no strict definition how many elements need to be zero for a matrix to be considered sparse. In many examples, the number of nonzeros per row or column is a small fraction, a few percent or less, of the total number of elements of the row or column. By contrast, if most of the elements are nonzero, then the matrix is considered *dense*.

In the context of software for scientific computing, a sparse matrix typically refers to a storage format, in which elements which are known to be zero are not stored. In Python, the library `scipy.sparse` defines several sparse matrix classes, such as `scipy.sparse.csr_array`. To construct such an object, one passes for each nonzero element the value, and the row and column coordinates. In some cases, one can also just pass the nonzero (off-)diagonals, see `scipy.sparse.diags`.

Functions for dense matrices do not always work with sparse matrices. For example for the product of a sparse matrix with a (dense) vector, there is the member function `scipy.sparse.csr_array.dot`, and for solving linear equations involving a sparse matrix, there is the function `scipy.sparse.linalg.spsolve`.

In [1]:

```
# Import some basic packages
import numpy as np
import matplotlib.pyplot as plt
import math
```

In [2]:

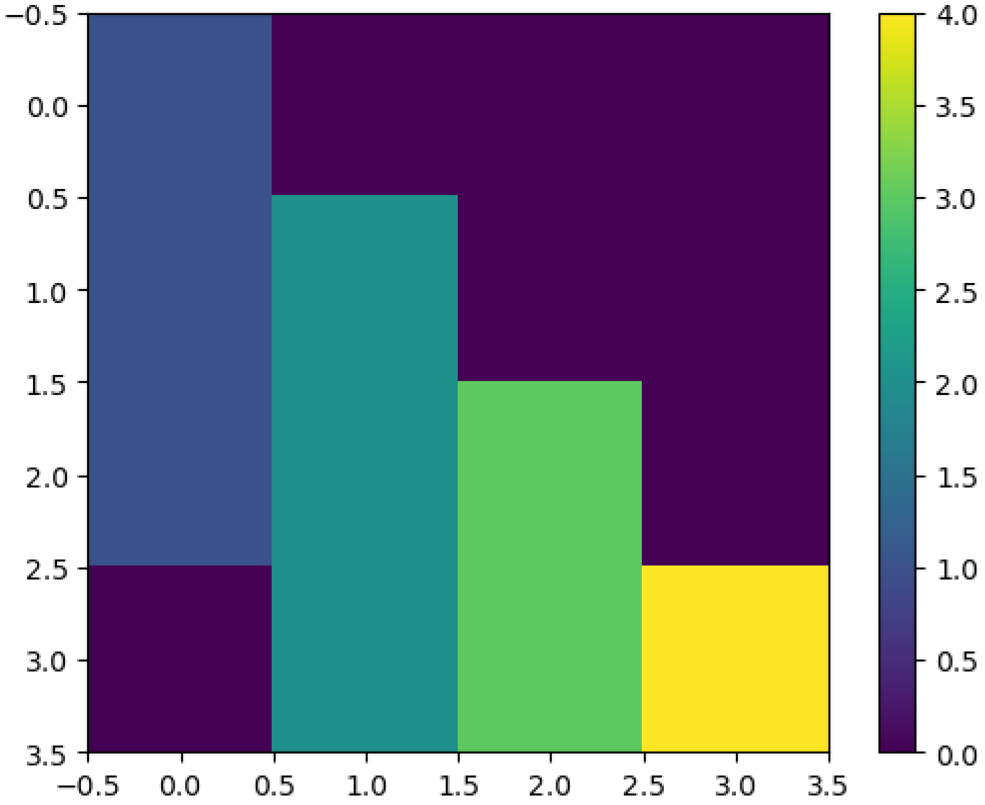
```
from scipy.sparse import csr_array

# This is how to create a sparse matrix from a given list of (row, column, 
row  = [0,   3,   1,   0]
col  = [0,   3,   1,   2]
data = [4.0, 5.0, 7.0, 9.0]
M = csr_array((data, (row, col)), shape=(4, 4))

print("When printing a sparse matrix, it shows its nonzero entries:")
print(M)

print("If you want to see its `dense` matrix form, you have to use `mat.toar
print(M.toarray())

# This is how to perform matrix-vector products.
x = np.array([1, 2, 3, 4])
print("For x={}, Mx = {}".format(x, M.dot(x)))
```

```
When printing a sparse matrix, it shows its nonzero entries:
  (0, 0)	4.0
  (0, 2)	9.0
  (1, 1)	7.0
  (3, 3)	5.0
If you want to see its `dense` matrix form, you have to use `mat.toarray()
`:
[[4. 0. 9. 0.]
 [0. 7. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 5.]]
For x=[1 2 3 4], Mx = [31. 14.  0. 20.]
```

In [3]:

```
from scipy.sparse import diags, SparseEfficiencyWarning
from scipy.sparse.linalg import spsolve
import warnings
warnings.simplefilter('ignore', SparseEfficiencyWarning)  # Suppress confusi

# This is how to create a sparse matrix from a given list of subdiagonals.
diagonals = [[1, 2, 3, 4], [1, 2, 3], [1, 2]]
M = diags(diagonals, [0, 1, 2])
print("This matrix has values on its diagonal and on offdiagonals 1 and 2 ro
print(M.toarray())

M = diags(diagonals, [0, -1, -2])
print("This matrix has values on its diagonal and on offdiagonals 1 and 2 ro
print(M.toarray())

print("If you want to visualize the matrix for yourself, use `plt.imshow`:")
plt.imshow(M.toarray())
plt.colorbar()
plt.show()

# This is how to solve sparse systems.
b = np.array([1, 2, 3, 4])
x = spsolve(M, b)
print("For b={}, the solution x to Mx=b is {}".format(b, x))
print("And indeed, Mx - b = {}".format(M.dot(x) - b))
```

```
This matrix has values on its diagonal and on offdiagonals 1 and 2 rows ABO
VE it.
[[1. 1. 1. 0.]
 [0. 2. 2. 2.]
 [0. 0. 3. 3.]
 [0. 0. 0. 4.]]
This matrix has values on its diagonal and on offdiagonals 1 and 2 rows BEL
OW it.
[[1. 0. 0. 0.]
 [1. 2. 0. 0.]
 [1. 2. 3. 0.]
 [0. 2. 3. 4.]]
If you want to visualize the matrix for yourself, use `plt.imshow`:
```



```
For b=[1 2 3 4], the solution x to Mx=b is [1.         0.5        0.3333333
3 0.5       ]
And indeed, Mx - b = [0. 0. 0. 0.]
```

Exercise 1

Consider the following boundary value problem involving a nonlinear ordinary differential equation:

$$y''(x) + \exp(y(x)) = 0, \quad 0 < x < 1, \quad y(0) = y(1) = 0. \tag{1}$$

The purpose of this exercise is to approximate the solution to this boundary value problem, by discretizing the problem and then solving the resulting system of nonlinear equations.

Problem (1) will be discretized using finite differences. Suppose we use $n+2$ discretization points for x, denoted $x_k = kh$ for $k \in \{0, \dots, n+1\}$ and $h = 1/(n+1)$. The approximate solution is denoted $y_k = y(x_k)$.

We will use a *second-order central finite difference* approximation for the second derivative:

$$y''(x_k) \approx \frac{y_{k-1} - 2y_k + y_{k+1}}{h^2}. \tag{2}$$

The term $\exp(y(x_k))$ can simply be approximated by $\exp(y_k)$. Thus for $x = x_k$, equation (1) becomes

$$\frac{y_{k-1} - 2y_k + y_{k+1}}{h^2} + \exp y_k = 0, \quad k = 1, \ldots, n. \tag{3}$$

The boundary conditions (the conditions $y(0) = y(1) = 0$), lead to the requirement that $y_0 = y_{n+1} = 0$. To find the remaining values y_k, $k = 1, \ldots, n$, equation (3) will be used for $k = 1, \ldots, n$. In this way, one obtains n equations for n unknowns, to which, in principle, a rootfinding method can be applied.

We will write $\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$ for the vector of values to be determined.

(a) (2 pts)

As a first step, finish the function `SecondDerMatrix` that returns a matrix $\mathbf{M}$ that maps the vector $\vec{y}$ to the vector of the approximate values $y''(x_k)$, $k = 1, \ldots, n$ given in (2). To get full points for this part of the exercise you must create output in the form of a sparse matrix.

In [4]:

```
def SecondDerMatrix(n):
    h = 1 / (n + 1)
    diagonals = [[1] * (n - 1), [-2] * n, [1] * (n - 1)]
    M = diags(diagonals, [-1, 0, 1])
    M /= h**2
    return M
```

(b) (1 pt)

Second-order central finite differences are exact for quadratic functions. In order to test your implementation, choose $n = 10$ and apply the second derivative matrix from part (a) to a quadratic function $y(x)$ with $y(0) = y(1) = 0$ for which you know the second derivative $y''(x)$.

In [5]:

```
n = 10
M = SecondDerMatrix(n)
x = np.linspace(0, 1, n + 2)
y = x * x - x  # y''(x) = 2
y = y[1:-1]  # y: [y0, y1, ..., y(n+1)] -> [y1, y2, ..., yn]
print(M.dot(y))  # output should be [2, 2, ..., 2]
```

```
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

(c) (2 pts)

Defining $\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$ and $E(\vec{y}) = \begin{bmatrix} \exp(y_1) \\ \vdots \\ \exp(y_n) \end{bmatrix}$, the equations (3) can be written in the form

$$F(\vec{y}) := \mathbf{M} \cdot \vec{y} + E(\vec{y}) = \vec{0}.$$

Finish the function `F` that defines $F(\vec{y}) = \mathbf{M} \cdot \vec{y} + E(\vec{y})$. Finish the function `JacobianF` that computes the Jacobian $\mathbf{J}_F(\vec{y})$ of $F(\vec{y})$. To get full points for this part of the exercise, the Jacobian must be computed in the form of a sparse matrix.

In [6]:

```
def F(y):
    M = SecondDerMatrix(len(y))
    return M.dot(y) + np.exp(y)


def JacobianF(y):
    M = SecondDerMatrix(len(y))
    diagonals = np.exp(y)
    Jexp = diags(diagonals)
    return M + Jexp
```

(d) (3 pts)

1. Write down the first order Taylor expansion $T_F(\vec{y}, \vec{s})$ for $F(\vec{y} + \vec{s})$.
2. In order to check your implementation of the Jacobian matrix, compute and print both $F(\vec{y} + \vec{s})$ and its first order Taylor approximation $T_F(\vec{y}, \vec{s})$ for a choice $\vec{y}$ and $\vec{s}$.
3. Verify numerically that the error $||F(\vec{y} + \vec{s}) - T_F(\vec{y}, \vec{s})||_2$ is $\mathcal{O}(||\vec{s}||_2^2)$. Hint: take vectors $\vec{s}$ with $||\vec{s}||_2 = \mathcal{O}(h)$ for multiple values for h, e.g. $h = 10^{-k}$ for a range of k.

Subquestion 1.

$T_F(\vec{y}, \vec{s}) \approx F(\vec{y}) + \mathbf{J}_F(\vec{y})\vec{s}$

In [7]:

```
# Subquestion 2.
n = 10
x = np.linspace(0, 1, n + 2)[1:-1]
y = x**2 - x
s = 0.01 * np.ones_like(y)
T_F = F(y) + JacobianF(y).dot(s)
print("Subquestion 2.")
print(f"F(y+s) = {F(y+s)}")
print(f"T_F(y,s) = {T_F}")
print()

# Subquestion 3.
print("Subquestion 3.")
flag = 1
threshold = 1
```

```
for k in range(1, 10):
    h = 2**-k
    n = 2**k - 1
    x = np.linspace(0, 1, n + 2)[1:-1]
    y = x**2 - x
    s = h * np.ones_like(y) / np.sqrt(n)
    T_F = F(y) + JacobianF(y).dot(s)
    error = np.linalg.norm(F(y + s) - T_F, 2)
    s2 = s.T.dot(s)
    error_s2 = error/s2
    print(f"h = {h}, error/s^2 = {error_s2}")
    if error_s2 > threshold:
        flag = 0
        break
if flag:
    print("Verification succeeded!")
else:
    print("Verification failed.")
```

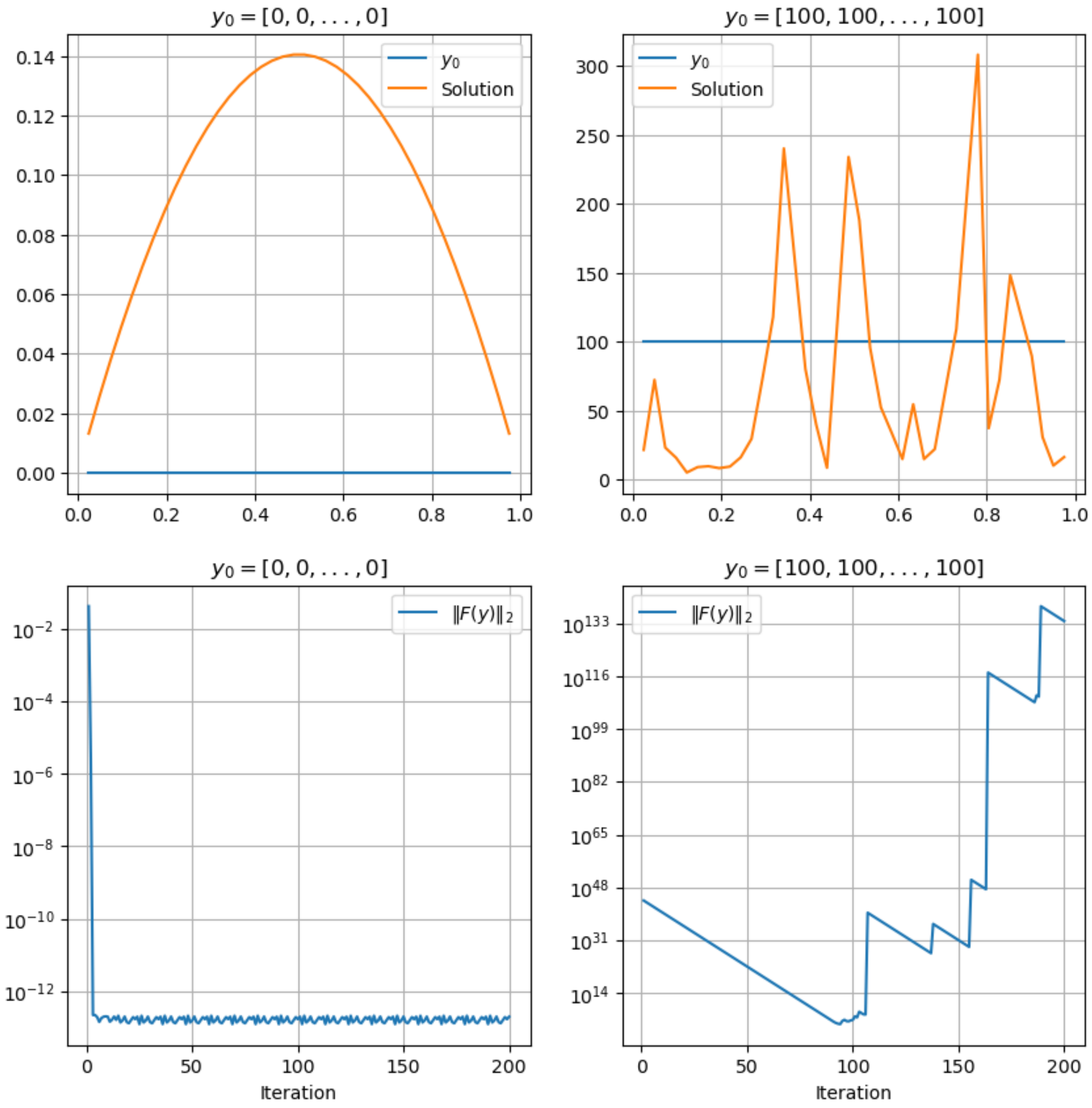
```
Subquestion 2.
F(y+s) = [1.71993124 2.87043662 2.82832714 2.80139208 2.78825481 2.78825481
 2.80139208 2.82832714 2.87043662 1.71993124]
T_F(y,s) = [1.71988506 2.87039339 2.828286   2.80135228 2.78821565 2.788215
65
 2.80135228 2.828286   2.87039339 1.71988506]

Subquestion 3.
h = 0.5, error/s^2 = 0.46329696832253653
h = 0.25, error/s^2 = 0.2462923298545746
h = 0.125, error/s^2 = 0.15964255471735284
h = 0.0625, error/s^2 = 0.10920044082913484
h = 0.03125, error/s^2 = 0.07614303995044426
h = 0.015625, error/s^2 = 0.05350563178525392
h = 0.0078125, error/s^2 = 0.03772481009262513
h = 0.00390625, error/s^2 = 0.026638876302021157
h = 0.001953125, error/s^2 = 0.018824073880718238
Verification succeeded!
```

(e) (2 pts)

1. Finish the function `NewtonSolve` below to solve the system of equations.
2. Take $n = 40$, and experiment with your function. Try to find a choice of `y0` such that the method doesn't converge, as well as a choice of `y0` such that the method converges. In your answer, list the types of convergence behavior you found. Show a convergent example (if you found any) and a nonconvergent example (if you found any). Show the solutions you found for each example.

In [8]:
```
# Subquestion 1.
def NewtonSolve(y0, K):
    """ Use Newton's method to solve F(y) = 0 with initial guess y0 and K it
    y = y0
    for _ in range(K):
        J = JacobianF(y)
        s = spsolve(J, -F(y))
        y += s
    return y
```

In [9]:
```
# Subquestion 2, code part
def Residuals(y0, K):
    y = y0
    Fy = []
    for _ in range(K):
        J = JacobianF(y)
        s = spsolve(J, -F(y))
        y += s
        Fy.append(F(y))
    Fy = np.array(Fy)
    return np.linalg.norm(Fy, axis=1)


n = 40

# Convergent example
y0_1 = np.zeros(n)
y200_1 = NewtonSolve(y0_1.copy(), 200)
Fy_1 = Residuals(y0_1.copy(), 200)

# Nonconvergent example
y0_2 = 100 * np.ones(n)
y200_2 = NewtonSolve(y0_2.copy(), 200)
Fy_2 = Residuals(y0_2.copy(), 200)

# Plot
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
x = np.linspace(0, 1, n + 2)[1:-1]
ks = np.arange(1, 201)

ax = axes[0][0]
ax.set_title(r"$y_0=[0, 0, ..., 0]$")
ax.plot(x, y0_1, label=r"$y_0$")
ax.plot(x, y200_1, label='Solution')
ax.grid()
ax.legend()

ax = axes[0][1]
ax.set_title(r"$y_0=[100, 100, ..., 100]$")
ax.plot(x, y0_2, label=r"$y_0$")
ax.plot(x, y200_2, label='Solution')
ax.grid()
ax.legend()

ax = axes[1][0]
ax.set_title(r"$y_0=[0, 0, ..., 0]$")
ax.set_xlabel("Iteration")
ax.semilogy(ks, Fy_1, label=r"$\|F(y)\|_2$")
ax.grid()
ax.legend()

ax = axes[1][1]
ax.set_title(r"$y_0=[100, 100, ..., 100]$")
ax.set_xlabel("Iteration")
ax.semilogy(ks, Fy_2, label=r"$\|F(y)\|_2$")
ax.grid()
ax.legend()

plt.show()
```



We select the initial vector $\vec{y}_0 = [0, 0, \ldots, 0]$ to facilitate convergence of the method. In contrast, we choose $\vec{y}_0 = [100, 100, \ldots, 100]$ as an example where the method fails to converge. In the case of $\vec{y}_0 = [0, 0, \ldots, 0]$, the norm $\|F(y)\|_2$ rapidly diminishes to a value below 10^{-12}, subsequently entering a phase of oscillation with amplitudes smaller than 10^{-12}. Conversely, when $\vec{y}_0 = [100, 100, \ldots, 100]$, $\|F(y)\|_2$ initially decreases but then exhibits a non-regular increase. Hence, it can be inferred that the method converges for $\vec{y}_0 = [0, 0, \ldots, 0]$ and diverges when $\vec{y}_0 = [100, 100, \ldots, 100]$. The solutions for each scenario are detailed above.