

# 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.

Koen Weverink (14711982) & Jasper Timmer (12854328)

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

```
In [ ]: from scipy.sparse import csr_array

# This is how to create a sparse matrix from a given list of (row, column, value)
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.toarray`")
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 [ ]: from scipy.sparse import diags, SparseEfficiencyWarning
from scipy.sparse.linalg import spsolve
import warnings
warnings.simplefilter('ignore', SparseEfficiencyWarning) # Suppress confusing w

# 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 rows A")
print(M.toarray())

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

```

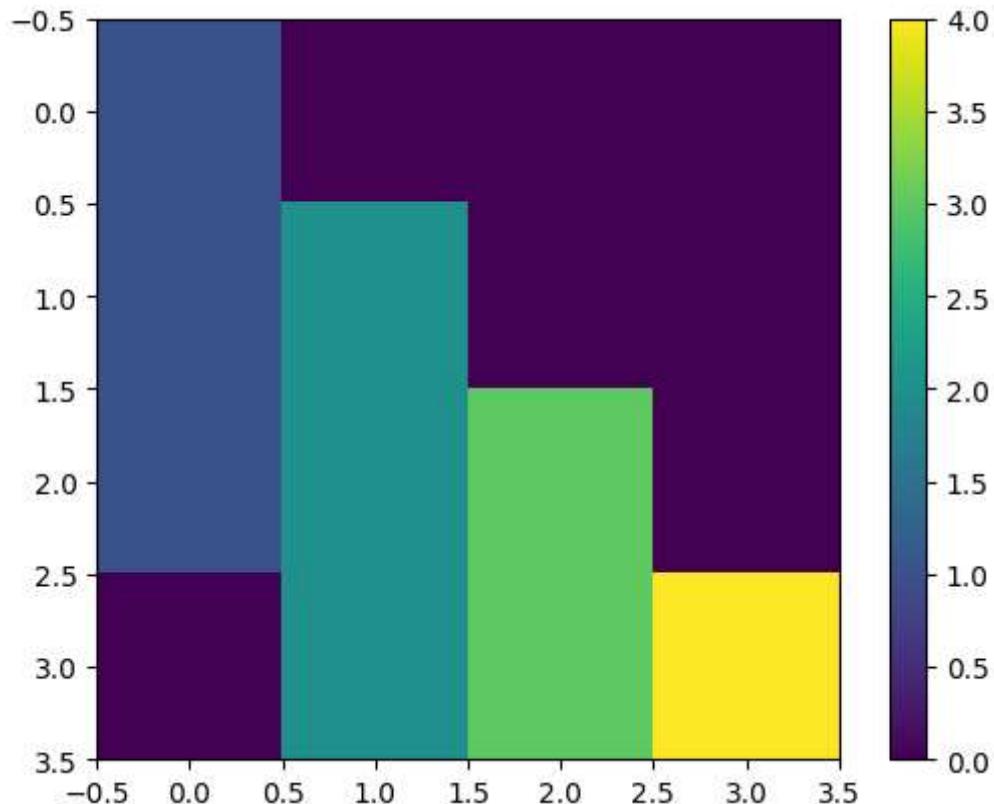
This matrix has values on its diagonal and on offdiagonals 1 and 2 rows ABOVE it.

```
[[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 BELOW 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.

5 ]

0.5

0.33333333 0.

And indeed,  $Mx - b = [0. \ 0. \ 0. \ 0.]$

## Exercise 1

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

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

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:

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

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

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

The boundary conditions (the conditions  $y(0) = y(1) = 1$ ), lead to the requirement that  $y_0 = y_{n+1} = 0$ . To find the remaining values  $y_k$ ,  $k = 1, \dots, n$ , equation (3) will be used for  $k = 1, \dots, n$ . 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  $y''(x_k)$ ,  $k = 1, \dots, n$  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 [ ]: from scipy.sparse import csr_array

def SecondDerMatrix(n):
    """
    Generate a sparse matrix representing the second derivative operator for a d
    Parameters:
    - n (int): Number of discrete points in the domain.

    Returns:
    - scipy.sparse.csr_matrix: Sparse matrix representing the second derivative
    """
    h = 1.0 / (n + 1)

    main_diag = -2 * np.ones(n)
    sub_diag = np.ones(n - 1)
    sup_diag = np.ones(n - 1)
```

```

M = csr_array(np.diag(main_diag) + np.diag(sub_diag, k = -1) + np.diag(sup_d)

M *= 1 / h**2

return M

SecondDerMatrix(8).toarray()

```

```

Out[ ]: array([[-162.,   81.,    0.,    0.,    0.,    0.,    0.,    0.],
               [  81., -162.,   81.,    0.,    0.,    0.,    0.,    0.],
               [  0.,   81., -162.,   81.,    0.,    0.,    0.,    0.],
               [  0.,    0.,   81., -162.,   81.,    0.,    0.,    0.],
               [  0.,    0.,    0.,   81., -162.,   81.,    0.,    0.],
               [  0.,    0.,    0.,    0.,   81., -162.,   81.,    0.],
               [  0.,    0.,    0.,    0.,    0.,   81., -162.,   81.],
               [  0.,    0.,    0.,    0.,    0.,    0.,   81., -162.]])

```

## (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 [ ]: n = 10
M = SecondDerMatrix(n)

x = np.linspace(0, 1, n + 2)
y = x ** 2 - x

second_derivative = M.dot(y[1:-1])

actual_value = 2

print("Analytical solution:", actual_value)
print("Numerical approximation:", second_derivative)

```

Analytical solution: 2

Numerical approximation: [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 [ ]: def F(y):
    """
        Evaluate the nonlinear function F(y) defined as M * y + exp(y), where M is t
    Parameters:
    - y (numpy.ndarray): Input vector.

    Returns:
    - numpy.ndarray: Result of evaluating the nonlinear function.

    """
    M = SecondDerMatrix(len(y))
    return M.dot(y) + np.exp(y)

def JacobianF(y):
    """
        Compute the Jacobian matrix of the nonlinear function F(y) at the given point y.
    Parameters:
    - y (numpy.ndarray): Input vector.

    Returns:
    - scipy.sparse.csr_matrix: Jacobian matrix of F(y).
    """
    n = len(y)
    h = 1 / (n + 1)

    main_diag = -2 * 1 / h**2 + np.exp(y)
    off_diag = np.ones(n - 1) / h**2

    J = csr_array(np.diag(main_diag) + np.diag(off_diag, k = -1) + np.diag(off_d
    return J

y = [1, 2, 1, 2, 1]
print(F(y))
print(JacobianF(y).toarray())

```

```
[ 2.71828183 -64.6109439   74.71828183 -64.6109439   2.71828183]
[[[-69.28171817  36.          0.          0.          0.        ]
 [ 36.          -64.6109439   36.          0.          0.        ]
 [ 0.          36.          -69.28171817  36.          0.        ]
 [ 0.          0.          36.          -64.6109439   36.        ]
 [ 0.          0.          0.          36.          -69.28171817]]]
```

## (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.

The first order Taylor expansion for  $F(\vec{y} + \vec{s})$  is  $F(\vec{y} + \vec{s}) \approx F(\vec{y}) + J_F(\vec{y}) \cdot \vec{s}$ . Which is, in this case,  $\mathbf{M} \cdot \vec{y} + E(\vec{y}) + J_F(\vec{y}) \cdot \vec{s}$ . Below is given an elaboration of the Matrix translation of this formula:

$$\begin{bmatrix} \frac{1}{h^2}(-2y_1 + y_2) \\ \frac{1}{h^2}(y_1 - 2y_2 + y_3) \\ \vdots \\ \frac{1}{h^2}(y_{n-2} - 2y_{n-1} + y_n) \\ \frac{1}{h^2}(y_{n-1} - 2y_n) \end{bmatrix} + \begin{bmatrix} e^{y_1} \\ e^{y_2} \\ \vdots \\ e^{y_{n-1}} \\ e^{y_n} \end{bmatrix} + \begin{bmatrix} \frac{2}{h^2} + e^{y_1} & \frac{1}{h^2} & 0 & \dots & 0 \\ \frac{1}{h^2} & \frac{2}{h^2} + e^{y_2} & \frac{1}{h^2} & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \frac{1}{h^2} & \frac{2}{h^2} + e^{y_{n-1}} & \frac{1}{h} \\ 0 & \dots & 0 & \frac{1}{h^2} & \frac{2}{h^2} + e^{y_n} \end{bmatrix}$$

Working this out will eventually result in: \$\$

$\{\mathbf{M}\} \cdot \vec{y} + E(\vec{y}) + \{\mathbf{M}\} \cdot \vec{s} + E(\vec{y}) \cdot \vec{s}^T$

```
In [ ]: def T_F(y, s):
    """
        Evaluate the directional derivative of the nonlinear system function F(y) at

        Parameters:
        - y (numpy.ndarray): Point at which to evaluate the directional derivative.
        - s (numpy.ndarray): Direction vector.

        Returns:
        - numpy.ndarray: Result of evaluating the directional derivative.
    """

    return F(y) + JacobianF(y).dot(s)

y = np.array([1, 2, 1, 2, 1])
s = np.array([0.001, 0.001, 0.001, 0.001, 0.001])

print(f"T_F(y, s) = {T_F(y, s)}")
print(f"F(y + s) = {F(y + s)}")

n = 10
y = np.arange(n)
s = np.ones(n)
s_norm = np.linalg.norm(s)

k = np.arange(n + 1, dtype=float)
hs = np.power(10, -k)

new_s_matrix = (hs[:, np.newaxis] / s_norm) * s
norm_s_array = np.linalg.norm(new_s_matrix, axis=1) ** 2

error_list = [np.linalg.norm(F(y + new_s) - T_F(y, new_s)) for new_s in new_s_ma
print(norm_s_array)
print(error_list)

plt.plot(hs, error_list, label=r'|| F(\vec{y} + \vec{s}) - T_F(\vec{y}, \vec{s}) ||_
```

```

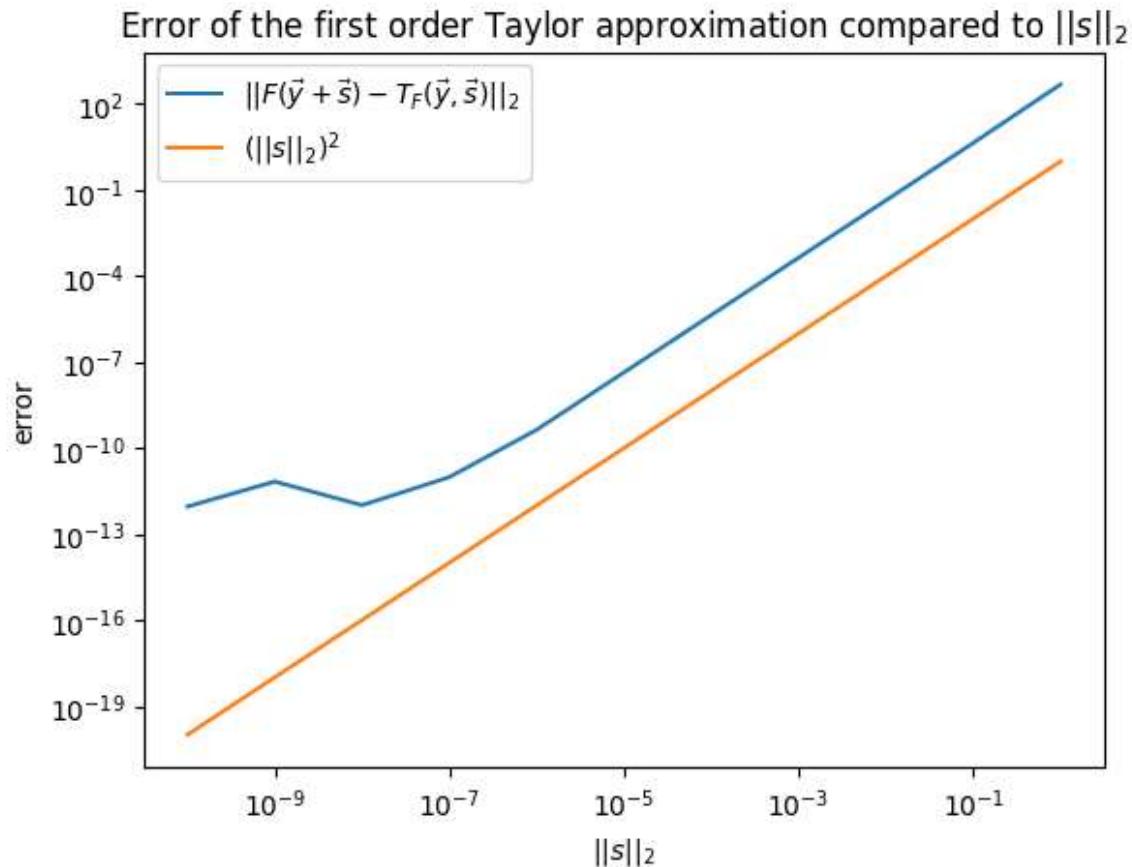
plt.plot(hs, norm_s_array, label = r'$\|s\|_2^2$')
plt.xlabel('$\|s\|_2$')
plt.ylabel('error')
plt.xscale('log')
plt.yscale('log')
plt.title('Error of the first order Taylor approximation compared to $\|s\|_2$')
plt.legend()
plt.show()

```

```

T_F(y, s) = [ 2.68500011 -64.60355484  74.72100011 -64.60355484   2.68500011]
F(y + s) = [ 2.68500147 -64.60355115  74.72100147 -64.60355115   2.68500147]
[1.e+00 1.e-02 1.e-04 1.e-06 1.e-08 1.e-10 1.e-12 1.e-14 1.e-16 1.e-18
 1.e-20]
[485.50991502569156, 4.403382566259102, 0.043616858204482176, 0.000435754876916
2599, 4.357135051950163e-06, 4.356304446730756e-08, 4.3330624114044883e-10, 9.6
61124251207325e-12, 1.0173427625490205e-12, 6.775956472493152e-12, 9.1052844316
47262e-13]

```



## (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 [ ]: `def NewtonSolve(y0, K):`  
 `"""`

`Apply Newton's method to solve a nonlinear system of equations.`

```

Parameters:
- y0 (numpy.ndarray): Initial guess for the solution.
- K (int): Number of iterations to perform.

Returns:
- tuple: A tuple containing:
    - y (numpy.ndarray): The estimated solution to the nonlinear system.
    - newton_iterations (list): List of norms of the solution at each iteration
"""

y = y0
newton_iterations = []
for _ in range(K):
    sk = spsolve(JacobianF(y), -F(y))
    y += sk
    newton_iterations.append(np.linalg.norm(y))
return y, newton_iterations

```

In [ ]:

```

n = 40

y0_0 = np.zeros(n)
y0_1 = np.ones(n)
y0_2 = y0_1 / 2 + 1
y0_3 = np.arange(n, dtype=np.float64)
y0_4 = y0_3 / 2 + 1

K = 1000

y0, newton_iterations = NewtonSolve(y0_0, K)
y1, newton_iterations_1 = NewtonSolve(y0_1, K)
y2, newton_iterations_2 = NewtonSolve(y0_2, K)
y3, newton_iterations_3 = NewtonSolve(y0_3, K)
y4, newton_iterations_4 = NewtonSolve(y0_4, K)

plt.plot(range(K), newton_iterations, label = 'y1')
plt.plot(range(K), newton_iterations_1, label = 'y2')
plt.plot(range(K), newton_iterations_2, label = 'y3')
plt.plot(range(K), newton_iterations_3, label = 'y4')
plt.plot(range(K), newton_iterations_4, label = 'y5')
plt.xscale('log')
plt.xlabel('iterations')
plt.ylabel('Newton solve norm')
plt.yscale('log')
plt.legend(fontsize=9)
plt.ylim(10**(-3), 10**3)

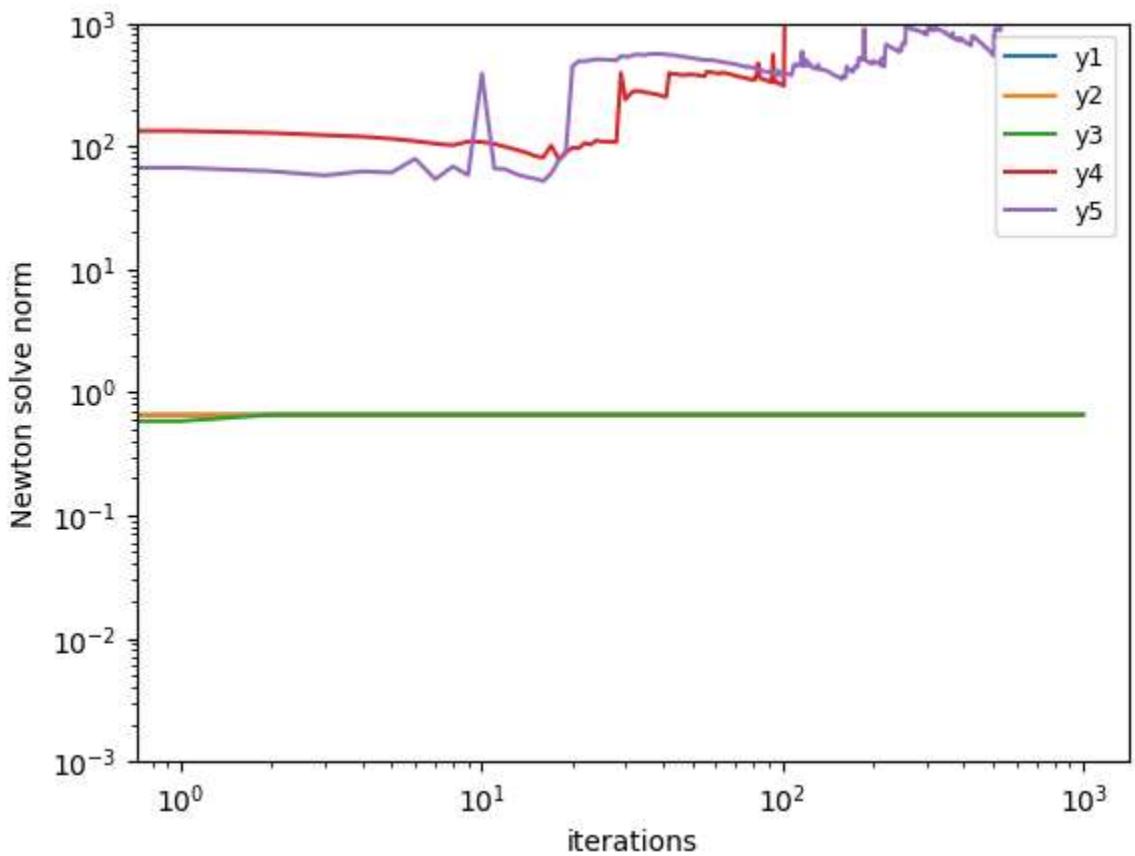
```

```

C:\Users\koenw\AppData\Local\Temp\ipykernel_15852\2373521970.py:31: RuntimeWarning: overflow encountered in exp
    main_diag = -2 * 1 / h**2 + np.exp(y)
C:\Users\koenw\AppData\Local\Temp\ipykernel_15852\2373521970.py:14: RuntimeWarning: overflow encountered in exp
    return M.dot(y) + np.exp(y)
c:\Users\koenw\AppData\Local\Programs\Python\Python310\lib\site-packages\scipy\sparse\linalg\_dsolve\linsolve.py:276: MatrixRankWarning: Matrix is exactly singular
    warn("Matrix is exactly singular", MatrixRankWarning)

```

Out[ ]: (0.001, 1000)



We investigated the following initial values:

$\mathbf{y}_{0,1}$ :

A vector of zeros of length  $n$ :

$$\mathbf{y}_{0,1} = [0, 0, \dots, 0]$$

$\mathbf{y}_{0,2}$ :

A vector of ones of length  $n$ :

$$\mathbf{y}_{0,2} = [1, 1, \dots, 1]$$

$\mathbf{y}_{0,3}$ :

A vector obtained by halving each element of  $\mathbf{y}_{0,2}$  and adding 1:

$$\mathbf{y}_{0,3} = \frac{1}{2}\mathbf{y}_{0,2} + 1$$

$\mathbf{y}_{0,4}$ :

A vector containing sequential numbers starting from 0 up to  $n - 1$ :

$$\mathbf{y}_{0,4} = [0, 1, 2, \dots, n - 1]$$

$\mathbf{y}_{0,5}$ :

A vector obtained by halving each element of  $\mathbf{y}_{0,4}$  and adding 1:

$$\mathbf{y}_{0,5} = \frac{1}{2}\mathbf{y}_{0,4} + 1$$

As visible on the graph,  $\mathbf{y}_{0,4}$  and  $\mathbf{y}_{0,5}$  diverge, while the other sets of initial values converge.