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.

- Amir Sahrani (12661651)
- Jonas Schäfer (14039982)

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 [ ]:
```python
# Import some basic packages
import numpy as np
import matplotlib.pyplot as plt
import warnings
from scipy.sparse import diags, SparseEfficiencyWarning, csr_array
from scipy.sparse.linalg import spsolve
from scipy.sparse.linalg import inv as spinv
import typeguard
warnings.simplefilter('ignore', SparseEfficiencyWarning)  # Suppress conf
from pandas import DataFrame
```

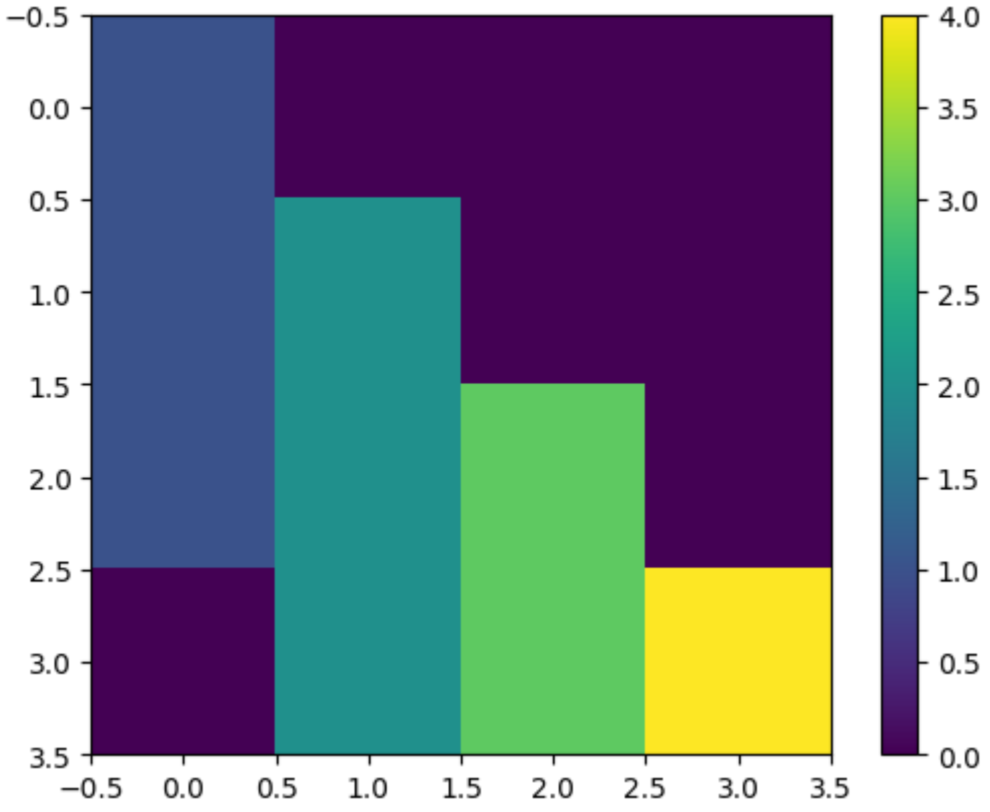
In [ ]:
```python
# This is how to create a sparse matrix from a given list of (row, column
row  = [0,   3,   1,   0]
col  = [0,   3,   1,   2]
data = [4.0, 5.0, 7.0, 9.0]
M = csr_array((data, (row, col)), shape=(4, 4))

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.t
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 [ ]:
```python
# 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]) # type: ignore
print("This matrix has values on its diagonal and on offdiagonals 1 and 2
print(M.toarray())

M = diags(diagonals, [0, -1, -2]) # type: ignore
print("This matrix has values on its diagonal and on offdiagonals 1 and 2
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 AB
OVE 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 BE
LOW 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        0.333333
33 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

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

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 $y''(x_k)$, $k = 2, \dots, n$ given in (2). To get full points for this part of the exercise you must create output in the form of a sparse matrix.

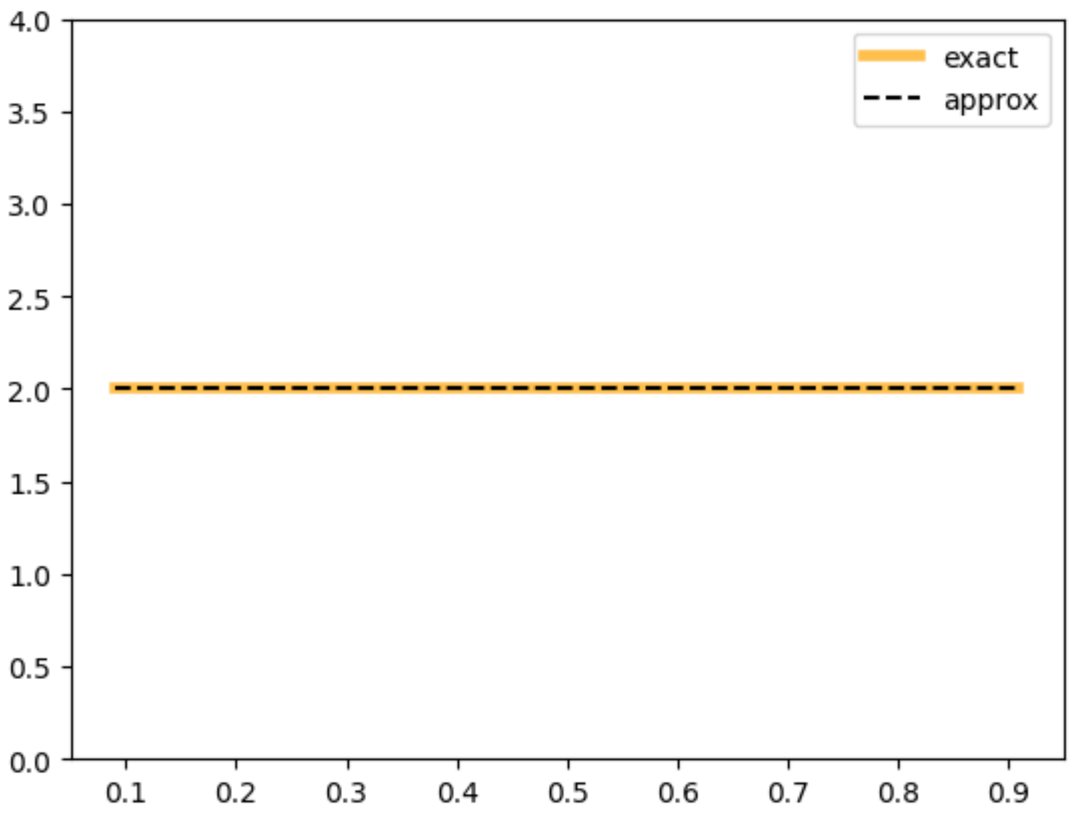
In [ ]:
```python
def SecondDerMatrix(n):
    '''Returns sparse matrix to map y one to second order derivative of y
    A = diags([np.ones(n-1), np.full(n, -2), np.ones(n-1)], [-1, 0, 1]) #
    h = 1/(n+1)

    return A/h**2
```

(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 $y''(x)$.

$(2x-1)^2 - 1$

```
In [ ]: def quadratic(x):
            return x**2 -x

        def second_derivative(x):
            return 2

        n = 10
        steps = np.linspace(0,1,n+2, endpoint=True)[1:-1]
        vec_quad = np.vectorize(quadratic)
        vec_der = np.vectorize(second_derivative)

        exact_y = vec_quad(steps)
        exact_der_y = vec_der(steps)
        A = SecondDerMatrix(n)
        approx_der_y = A @ exact_y

        plt.plot(steps, exact_der_y, label='exact', linewidth=4, alpha=0.7, color
        plt.plot(steps, approx_der_y, label='approx', linestyle='--', color='blac
        plt.ylim((0,4))
        plt.legend()
        plt.show()
```



(c) (2 pts)

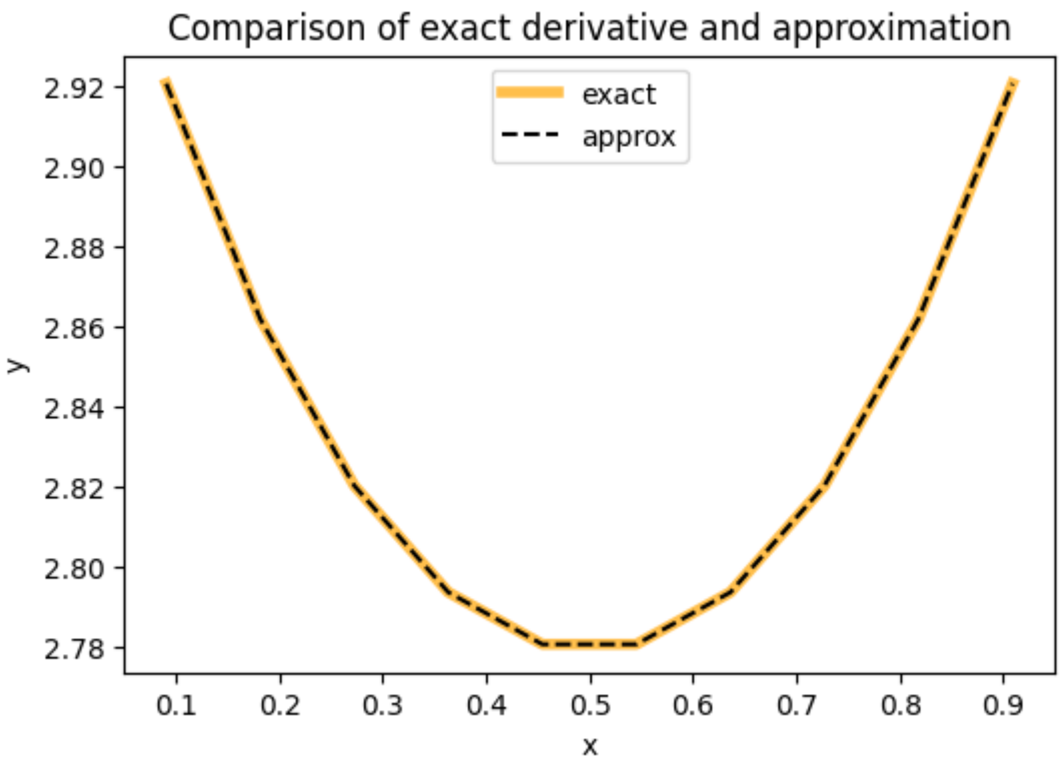
Defining $\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$ and $E(\vec{y}) = \begin{bmatrix} \exp(y_1) \\ \vdots \\ \exp(y_n) \end{bmatrix}$, the equations (3) can be written in the form

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

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):
    n = len(y)
    M = SecondDerMatrix(n)
    return (M @ y) + np.exp(y)

def JacobianF(y):
    '''Returns Jacobian of F given a vector y'''
    return SecondDerMatrix(len(y)) + diags(np.exp(y))

approx_data = F(exact_y)
exact_data = exact_der_y + np.exp(exact_y)
min_y_approx = min(approx_data)
min_y_exact = min(exact_data)
print(f'exact min = {min_y_exact} = approx min = {min_y_approx}')
plt.figure(figsize=(6,4))
plt.plot(steps, exact_der_y + np.exp(exact_y), label='exact', linewidth=4
plt.plot(steps, approx_data, label='approx', linestyle='--', color='black
plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparison of exact derivative and approximation')
plt.legend(loc='upper center')
plt.show()
```

```
exact min = 2.7804115390294966 = approx min = 2.7804115390294966
```



(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 $\vec{s}$.

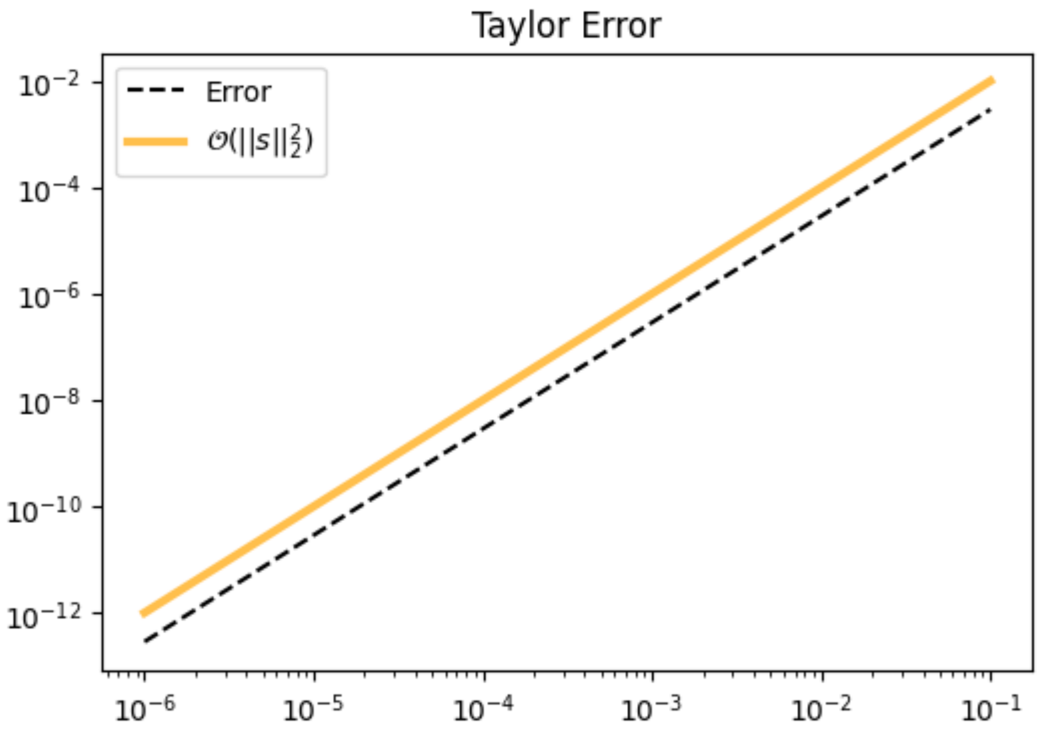
3. Verify numerically that the error $||F(\vec{y}+\vec{s}) - T_F(\vec{y},\vec{s})||_2$ is $\mathcal{O}(||\vec{s}||_2^2)$. Hint: take vectors $\vec{s}$ with $\|\vec{s}\|_2 = \mathcal{O}(h)$ for multiple values for h, e.g. $h = 10^{-k}$ for a range of k.

Subquestion 1.

$$T_F(\mathbf{y},\mathbf{s}) = F(\mathbf{y}) + (M + E(\mathbf{y}))\cdot \mathbf{s}$$
$$= F(\mathbf{y}) + \mathbf{J_f}(\mathbf{y})\cdot \mathbf{s}$$

In [ ]:
```
# Subquestions 2 and 3.
def TaylorF(y, h):
    '''Returns Taylor approximation of F at y with step size h'''
    return F(y) + JacobianF(y) @ h


# s = np.full(n, 10**-2)
# taylor_approx = TaylorF(exact_y, s)
# approx_data = F(exact_y + s)

# plt.figure(figsize=(6,4))
# plt.title('Comparison of Taylor approximation $T_{F}(\\mathbf{y}, \\mat
# plt.plot(steps, taylor_approx, label='Taylor $T_{F}(\\mathbf{y}, \\math
# plt.plot(steps, approx_data, label='$F(\\mathbf{y} + \\mathbf{s})$', li
# plt.legend()
# plt.show()


def explore_taylor_error(h_space, n=10):
    errors = []
    O_errors = []
    exact_y = np.linspace(0,1,n)
    for h in h_space:
        s = np.ones(n)
        s = s / np.linalg.norm(s) * h
        approx_data = F(exact_y + s)
        taylor_approx = TaylorF(exact_y, s)
        error = np.linalg.norm(approx_data - taylor_approx)
        O_error = h**2
        errors.append(error)
        O_errors.append(O_error)
        print(f'h = {h:.0e}, error = {error:.0e}, O_error = {O_error:.0e}
    return errors, O_errors

h_space = np.logspace(-6, -1, 6)
errors, O_errors = explore_taylor_error(h_space,10)

plt.figure(figsize=(6,4))
plt.title('Taylor Error')
plt.plot(h_space, errors, label='Error', color='black', linestyle='--')
plt.plot(h_space, O_errors, label='$\\mathcal{O}(||s||_2^2)$', color='ora
plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.show()
```

```
h = 1e-06, error = 3e-13, O_error = 1e-12
h = 1e-05, error = 3e-11, O_error = 1e-10
h = 1e-04, error = 3e-09, O_error = 1e-08
h = 1e-03, error = 3e-07, O_error = 1e-06
h = 1e-02, error = 3e-05, O_error = 1e-04
h = 1e-01, error = 3e-03, O_error = 1e-02
```



(e) (2 pts)

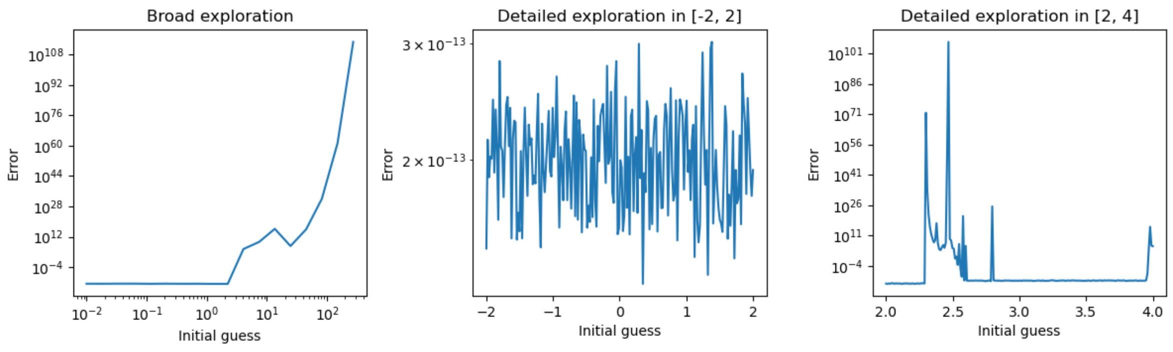
1. Finish the function `NewtonSolve` below to solve the system of equations.
2. Take $n = 40$, and experiment with your function. Try to find a choice of `y0` such that the method doesn't converge, as well as a choice of `y0` such that the method converges. In your answer, list the types of convergence 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 [ ]:
```python
# Subquestion 1.
def NewtonSolve(y0, K):
    """ Use Newton's method to solve F(y) = 0 with initial guess y0 and K
    y = y0
    delta = 1
    while np.linalg.norm(delta) > 1e-6 and K > 0:
        jac = JacobianF(y)
        new = y - spinv(jac) @ F(y)
        delta = y - new
        y = new
        K -= 1
    return y
```

In [ ]:
```python
# Subquestion 2.
def explore_newtonsolver_error(space, K=10, dim=40):
    errors = []
    for n in space:
        guess = np.full(dim, n)
        approx_sol = NewtonSolve(guess, K)
        error = np.linalg.norm(F(approx_sol))
        errors.append(error)
    return errors

plt.figure(figsize=(12,4), layout='tight')
space = np.logspace(-2, 4, 24)
errors = explore_newtonsolver_error(space)

plt.suptitle('Impact of selected initial guesses on NewtonSolve convergen
plt.subplot(131)
plt.plot(space, errors)
plt.title('Broad exploration')
plt.xlabel('Initial guess')
plt.ylabel('Error')
plt.xscale('log')
plt.yscale('log')

space = np.linspace(-2, 2, 202)
errors = explore_newtonsolver_error(space)
plt.subplot(132)
plt.plot(space, errors)
plt.title('Detailed exploration in [-2, 2]')
plt.xlabel('Initial guess')
plt.ylabel('Error')
plt.yscale('log')

space = np.linspace(2, 4, 202)
errors = explore_newtonsolver_error(space)
plt.subplot(133)
plt.plot(space, errors)
plt.title('Detailed exploration in [2, 4]')
plt.xlabel('Initial guess')
plt.ylabel('Error')
plt.yscale('log')
plt.show()
```

```
/var/folders/tg/bfb__1d16td__2vd879zjkrh0000gn/T/ipykernel_5022/284072906
3.py:8: RuntimeWarning: overflow encountered in exp
  return SecondDerMatrix(len(y)) + diags(np.exp(y))
/var/folders/tg/bfb__1d16td__2vd879zjkrh0000gn/T/ipykernel_5022/284072906
3.py:4: RuntimeWarning: overflow encountered in exp
  return (M @ y) + np.exp(y)
```

Impact of selected initial guesses on NewtonSolve conversion by analysis of its error given $K = 10$



In [ ]:
```
converging_guess = np.full(40, 2)
approx_sol = NewtonSolve(converging_guess, 10)
F_approx_sol = F(approx_sol)
non_converging_guess = np.full(40, 5)
```

```
approx_sol_2 = NewtonSolve(non_converging_guess, 10)
F_approx_sol_2 = F(approx_sol_2)

DataFrame({'Solution with convergence c_sol': approx_sol, 'F(c_sol)': F_a
```

Out[ ]:

	Solution with conversion c_sol	F(c_sol)	Solution without conversion no_c_sol	F(no_c_sol)
0	0.013101	3.774758e-15	0.775937	0.090457
1	0.025599	-9.325873e-15	1.550636	0.940188
2	0.037487	1.310063e-14	2.323090	5.495863
3	0.048757	-3.996803e-15	3.092741	25.672838
4	0.059402	2.220446e-15	3.864554	116.549564
5	0.069417	-2.975398e-14	4.677336	688.778744
6	0.078793	-2.353673e-14	5.835921	2420.698037
7	0.087526	1.598721e-14	8.230864	341.929419
8	0.095610	-1.310063e-14	8.595380	2741.322708
9	0.103039	3.130829e-14	7.374351	13595.943495
10	0.109809	-5.151435e-14	13.292754	579659.594802
11	0.115914	8.659740e-15	11.344099	84173.648664
12	0.121352	5.884182e-14	9.221785	16016.609462
13	0.126118	-5.595524e-14	10.610182	36398.748783
14	0.130210	1.243450e-14	9.531687	15988.558428
15	0.133623	-7.327472e-15	9.761177	17172.571135
16	0.136357	-2.220446e-16	9.886878	19422.831981
17	0.138409	3.308465e-14	9.865212	19288.345986
18	0.139778	-4.307665e-14	9.866982	19280.073861
19	0.140462	2.642331e-14	9.866978	19282.976550
20	0.140462	-1.998401e-15	9.866978	19282.976549
21	0.139778	1.376677e-14	9.866982	19280.073859
22	0.138409	-2.375877e-14	9.865212	19288.345983
23	0.136357	2.819966e-14	9.886878	19422.831977
24	0.133623	-6.439294e-14	9.761177	17172.571133
25	0.130210	6.927792e-14	9.531687	15988.558428
26	0.126118	8.881784e-16	10.610182	36398.748782
27	0.121352	-5.484502e-14	9.221785	16016.609462
28	0.115914	3.708145e-14	11.344099	84173.648667
29	0.109809	-2.309264e-14	13.292754	579659.594844
30	0.103039	3.130829e-14	7.374351	13595.943493
31	0.095610	-4.152234e-14	8.595380	2741.322713
32	0.087526	4.418688e-14	8.230864	341.929423

	Solution with conversion c_sol	F(c_sol)	Solution without conversion no_c_sol	F(no_c_sol)
33	0.078793	-2.353673e-14	5.835921	2420.698037
34	0.069417	1.287859e-14	4.677336	688.778746
35	0.059402	2.220446e-15	3.864554	116.549564
36	0.048757	-1.110223e-14	3.092741	25.672838
37	0.037487	-1.110223e-15	2.323090	5.495863
38	0.025599	1.110223e-15	1.550636	0.940188
39	0.013101	3.774758e-15	0.775937	0.090457

For $n = 40$ and $K = 10$ the method converges for a vector $y0 \in \R^n$ e.g. if all its values $y_i = 2$ but does not converge for $y_i = 4$. As can be seen from the figure above, numerous guesses cause a very large error (i.e. no convergence). We find stable convergence for $-2 \leq y_i \leq 2$ (center figure), from there on we can observe a highly instable interval around [2.25, 2.6] (right figure). Thereforth, we observe increasing convergence with exceptions until convergence disappates further as y_i increases (left figure).