1 Splitting Methods: relax. Richardson, relax. Jacobi, Gauß-Seidel and SOR

1. Implement a function

   ```
   x, error, numiter = iter_solve(A, b, x0, method="Jacobi", theta=.1, tol=1e-08, maxiter=50)
   ```

   which takes as arguments
   - A : a matrix $A \in \mathbb{R}^{n \times n}$
   - b : a vector $b \in \mathbb{R}^n$
   - x0 : an initial guess $x^0 \in \mathbb{R}^n$
   - `method` : optional parameter to choose between relax. Richardson, weighted Jacobi, Gauß-Seidel and SOR and which is set to `"Jacobi"` by default
   - theta : relaxation parameter θ which is set to 0.1 by default
     (note: Gauß–Seidel is SOR with `theta=1.0`)
   - tol : error tolerance as float, which is set to 10^{-8} by default
   - maxiter : maximum number of iterations, which is set to 50 by default

   and then solves the system $Ax = b$ by applying the specified iterative scheme. It shall then return
   - `x` : list of all iterates x^k
   - `error` : list containing all residuals $\|Ax^k - b\|_2$
   - `numiter` : number of iterations that have been performed

   The iteration shall break if the residual is tolerably small, i.e.,

   $$\|Ax^k - b\|_2 < \texttt{tol}$$

   or the maximum number of iterations `maxiter` has been reached.

   Hint: Implement the element-based formulas for the Jacobi, Gauß-Seidel and SOR method (see previous exercise).

2. **2d:** Test <u>all</u> methods on the following two-dimensional setting:

   $$A = 4\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

   What is the exact solution x^*? Play around with the parameters `x0`, `theta`, `tol` and `maxiter`. Also create the following two plots for one fixed setting:
   - Plot the error $\|Ax^k - b\|_2$ for each iterate $x^k \in \mathbb{R}^2$, $k = 1, \dots, m$, for <u>all</u> methods into one plot (use different colors).
   - Plot the iterates $x^k \in \mathbb{R}^2$, $k = 1, \dots, m$, themselves for all methods into one plot (use different colors).

3. **nd:** Next, test all methods on the higher–dimensional analogue

   $$A = n^2 \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n},$$

   for different dimensions $n \in \mathbb{N}$ and data $b, x^0 \in \mathbb{R}^n$ of your choice. Play around with the parameters.

Hint: Of course, it can happen that the iterations do not converge. Use small values for θ when you use the Richardson iteration. This will assure that $\rho(I - NA) < 1$.

Solution:

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")


# ---------------------------------------------------------------#
#                     ITERATIVE SOLVER
# ---------------------------------------------------------------#
def Richardson_step(A, b, theta, x):
    return x - theta * (A @ x - b)


def Jacobi_step(A, b, theta, x):
    n = len(x)
    xnew = np.zeros(n)
    for i in range(n):
        s1 = np.dot(A[i, :i], x[:i])
        s2 = np.dot(A[i, i + 1:], x[i + 1:])
        xnew[i] = (1. - theta) * x[i] + theta / A[i, i] * (b[i] - s1 - s2)
    return xnew


def SOR_step(A, b, theta, x):
    n = len(x)
    xnew = np.zeros(n)
    for i in range(n):
        s1 = np.dot(A[i, :i], xnew[:i])  # <-- here we use already the new info
        s2 = np.dot(A[i, i + 1:], x[i + 1:])
        xnew[i] = (1. - theta) * x[i] + theta / A[i, i] * (b[i] - s1 - s2)
    return xnew


def steepestDescent_step(A, b, theta, x):
    r = A@x - b
    theta = np.dot(r, r) / np.dot(A@r, r)
    return x - theta * (A @ x - b)


def conjugateGradient(A, b, x0, maxiter=50, tol=1e-8):
    X = [x0]
    n = len(x0)
    error = []
    r = b - A @ X[0]
    p = r
    alpha_alt = np.dot(r, r)
    for numiter in range(max(maxiter, n)):
        error += [np.linalg.norm(A.dot(X[-1]) - b)]
        if error[-1] < tol:
            return X, error, numiter
        v = A @ p
        lambd = alpha_alt / np.dot(v, p)
        X += [X[-1] + lambd * p]
        r = r - lambd * v
        alpha_neu = np.dot(r, r)
        p = r + alpha_neu/alpha_alt * p
        alpha_alt = alpha_neu
    return X, error, numiter
```

```python
def iter_solve(A, b, x0, method="Jacobi", theta=.1, maxiter=50, tol=1e-8):
    """
    solves a system Ax = b, where A is assumed to be invertible,
    with relaxed splitting methods: Jacobi, Richardson

    Parameters
    ----------
    A : (n, n) numpy.ndarray
         system matrix
    b : (n,) numpy.ndarray
         right-hand side
    x0: (n,) numpy.ndarray
         initial guess
    method : string
           indicates method: "Jacobi" (=default), "Richardson", "GS", "SOR"
    theta : number (int or float)
          relaxation parameter (step length) default theta = 0.1
    tol : number (float)
          error tolerance, iteration stops if ||Ax-b|| < tol
    maxiter : int
        number of iterations that are performed , default m=50

    Returns
    -------
    X : list of length N (=m or less), containing iterates
        columns represent iterates from x_0 to x_(N-1)
    error : list of length numiter containing norm of all residuals
    numiter : integer indicating how many iterations have been performed
    """
    X = [x0]
    error = []
    if method in ["GS", "SOR", "Jacobi", "steepestDescent"] and \
       np.prod(A.diagonal()) == 0:
        print(f"WARNING: Method was chosen to be {method} \
              but A has zero diagonal entries!")
        return None
    elif method == "GS":
        theta = 1.0
        method = "SOR"
    elif method == "CG":
        return conjugateGradient(A, b, x0, maxiter=maxiter, tol=tol)
    # choose the function to compute the iteration instruction
    # according to method
    stepInstructionDictionary = {"Jacobi": Jacobi_step,
                                 "Richardson": Richardson_step,
                                 "SOR": SOR_step,
                                 "steepestDescent": steepestDescent_step}
    stepInstruction = stepInstructionDictionary[method]

    # ITERATION
    for numiter in range(maxiter):
        error += [np.linalg.norm(A.dot(X[-1]) - b)]
        if error[-1] < tol:
            return X, error, numiter
        X += [stepInstruction(A, b, theta, X[-1])]

    return X, error, numiter


def main(A, b, x0, maxiter, methThet, plot=False, verbose=False):

    X = np.zeros((maxiter, 2, len(methThet)))
```

```python
    colors = ['r', 'g', 'b', "y", "c", "m"]
    # ---------------------------------------------------------------#
    #                     PLOT error and iterates
    # ---------------------------------------------------------------#
    if plot:
        plt.figure()

    for i, method in enumerate(methThet):
        X, error, numiter = iter_solve(A, b, x0, method=method,
                                       theta=methThet[method],
                                       maxiter=maxiter)
        if verbose:
            print(method, "\n \t\t  >", f"NumIter = {numiter}",
                  f"\t residual = {error[-1]:0.2e}")
        if plot:
            plt.subplot(1, 2, 1)
            plt.plot(error, colors[i]+"-x")
            plt.title("Residual $||Ax_k - b||_2$")
            plt.legend(method)
            plt.subplot(1, 2, 2)
            X = np.array(X)
            plt.plot(X[:, 0], X[:, 1], colors[i] + "o-")
            plt.legend(list(methThet.keys()))
            plt.title("Iterates $x_k$")
            plt.axis("equal")
    if plot:
        plt.show()
        plt.axis("equal")
    return X, error, numiter


if __name__ == "__main__":
    # ---------------------------------------------------------------#
    #       2d EXAMPLE
    # ---------------------------------------------------------------#
    A = 4. * np.array([[2, -1],
                       [-1, 2]])
    b = np.zeros(2)
    x0 = np.array([5, 8])
    maxiter = 100
    methThet = {"Richardson": 0.1, "Jacobi": 1, "GS": 1, "SOR": 1.2,
                "steepestDescent": -1, "CG": -1}
    print("-"*40+"\n 2d EXAMPLE (numiter, error) \n"+"-"*40)
    X, error, numiter = main(A, b, x0, maxiter,
                             methThet, plot=True, verbose=True)

    # ---------------------------------------------------------------#
    #                HIGHER DIMENSIONAL EXAMPLE
    # ---------------------------------------------------------------#
    n = 100  # 10000 # 100000
    A = n ** 2 * (2 * np.eye(n, k=0) - np.eye(n, k=1) - np.eye(n, k=-1))
    b = np.random.rand(n)
    x0 = np.random.rand(n)  # b#*100#
    firstMmethods = 4
    maxiter = 50
    methThet = {"Richardson": 0.00001, "Jacobi": 0.5, "GS": 1, "SOR": 1.9,
                "steepestDescent": 1}
    print("-"*40 + "\n nd EXAMPLE with n = {}\n".format(n) + "-"*40)
    X, error, numiter = main(A, b, x0, maxiter,
                             methThet, plot=0, verbose=True)
```