1 GMRES

1. **Arnoldi step:** For given orthonormal vectors $q_1, \ldots, q_{r-1} \in \mathbb{R}^n$ considered as a matrix $Q_{r-1} = [q_1, \ldots, q_{r-1}] \in \mathbb{R}^{n \times (r-1)}$, and a vector $v \in \mathbb{R}^n$, implement a helper function

$$q_r, h_r := \texttt{Arnoldi\_step}(Q_{r-1}, v),$$

which, according to the Arnoldi iteration, appends these vectors (i.e., the matrix Q_{r-1}) by an orthonormal vector q_r through orthogonalizing v against $q_1, \ldots, q_{r-1}$ and also outputs the numbers $h_r := (h_{1,r-1}, \ldots, h_{r,r-1})^\top \in \mathbb{R}^r$, where $h_{\ell,r-1} := q_\ell^\top v$ for $\ell \leq r$. You can then call `Arnoldi_step(Q; v)` within `GMRES(...)`.

2. **GMRES:** Implement a function

$$x = \texttt{GMRES(A, b, x0, tol=1e-6, maxiter=None, N=None)},$$

which takes as arguments

- A : a function evaluating the matrix–vector product $v \mapsto A \cdot v$ for some matrix $A \in \mathbb{R}^{n \times n}$ (not as an array!)
- b : a vector $b \in \mathbb{R}^n$
- x0 : an arbitrary initial guess $x^0 \in \mathbb{R}^n$
- tol : error tolerance as float, which is set to 10^{-6} by default
- maxiter : optional maximum number of iterations, which is set to `None` by default
- N : optional preconditioner as a function (not as an array), for which $N(v) \approx A^{-1}v$

and then solves the system $Ax = b$ by applying the GMRES method as presented in the lecture (see pseudocode 1 below). It shall then return

- x : the approximation to the solution.

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.

3. **Test** your solver on a random invertible tridiagonal matrix

$$A = \begin{pmatrix} * & * & 0 & \cdots & 0 \\ * & * & * & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & * & * & * \\ 0 & \cdots & 0 & * & * \end{pmatrix} \in \mathbb{R}^{n \times n}$$

and some right-hand side b and initial guess x_0 of your choice. Choose different n and check how many iterations you need (potentially many!).
Hint: You can generate some random diagonals using `numpy.random.rand(n)` and then use

```
scipy.sparse.diags(diagonals, offsets=[-1,0,1]).tocsr()
```

to construct a sparse CSR matrix. You can add `np.ones(n)` to the main diagonal in order to "strengthen" the diagonal of A and thereby to get a better conditioned system. Then implement a function `A(x)` that outputs the matrix-vector product `A.dot(x)`.

4. **Preconditioner:** For the same system, run your GMRES routine with a preconditioner of your choice (for example Jacobi $N : v \mapsto D^{-1}v$). Do you observe any difference in the number of iterations needed? (Don't worry if not!)

5. **Compare** your `GMRES` solver to `scipy.linalg.solve(A_csr.toarray(),b)` for large dimension $n \geq 10^5$ and measure the time needed in each case. Also, find a SciPy implementation of GMRES and compare to yours.

INPUT: $A \in GL_n(\mathbb{R})$, $b \in \mathbb{R}^n$
OUTPUT: approximation $x_r \in K_r(A,b)$ to the exact solution $A^{-1}b$

GMRES(A, b, $x_0 = 0$, `tol = 1e-6`, `maxiter=None`, $N = I$):
$b := b - Ax_0$ //account for initial guess
$A := NA$, $b := Nb$ //account for preconditioner
//Initialization:
$q_1 := \frac{b}{\|b\|_2}$, $Q_1 := [q_1]$
$v := Aq_1$
$\tilde{q}_1 := \frac{v}{\|v\|_2}$, $\tilde{Q}_1 := [\tilde{q}_1]$, $\tilde{R}_1 = [\|v\|_2]$
for $r = 2, ..., \min(n, \texttt{maxiter})$ **do**
  //STEP 1: use Arnoldi to find column q_r by orthogonalizing v against $q_1, \ldots, q_{r-1}$
  $q_r, h_{r-1} :=$ `Arnoldi_step`$(Q_{r-1}; v)$ //we don't need h_{r-1} in this variant
  $Q_r := [Q_{r-1}, q_r]$
  $v := Aq_r$

  //STEP 2: use Arnoldi to find columns $\tilde{q}_r$, $\tilde{r}_r$ by orthogonalizing v against $\tilde{q}_1, \ldots, \tilde{q}_{r-1}$
  $\tilde{q}_r, \tilde{r}_r :=$ `Arnoldi_step`$(\tilde{Q}_{r-1}; v)$
  $\tilde{Q}_r := [\tilde{Q}_{r-1}, \tilde{q}_r]$, $\tilde{R}_r := [\tilde{R}_{r-1}, \tilde{r}_r]$

  //STEP 3: solve auxiliary least squares problems to obtain coordinates
  $c_r :=$ `solve_triangular`$(\tilde{R}_r, \tilde{Q}_r^\top b)$
  $x_r := Q_r c_r$
  //Attention: Evaluate the original residual here:
  if $\|N^{-1}(Ax_r - b)\|_2 <$ `tol` **then**
    break
  end
end
return $x_0 + x_r$

Algorithm 1: GMRES

Solution:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
import scipy.linalg
from numpy.linalg import norm
import scipy.linalg as linalg
import scipy.sparse as sparse
import scipy.sparse.linalg as spla
import scipy
#import matplotlib.pyplot as plt
from time import time


# ---------------------------------------------------------------- #
#    One Arnoldi (Lanczos) step
# ---------------------------------------------------------------- #
def Arnoldi_step(Q, v):
    """
    perform one step of Arnoldi iteration to orthogonalize v against the
    columns of Q
```

```
    Parameters:
    -----------
    Q : matrix containing the orthonormal vectors as columns
    v : vector

    Returns:
    --------
    qr : vector which is orthonormal to all columns in Q
    hr : vector containing the linear coefficients so that
         v = [Q,qr] @ hr
    """
    r = np.shape(Q)[1]
    hr = np.zeros(r+1)
    for j in range(r):
        hr[j] = np.dot(Q[:, j], v)
        v = v - hr[j] * Q[:, j]
    hr[-1] = norm(v)
    return v / hr[-1], hr


def Lanczos_step(Q, v):
    r = np.shape(Q)[1]
    hr = np.zeros(r+1)
    for j in range(max(0, r-2), r):
        hr[j] = np.dot(Q[:, j], v)
        v = v - hr[j] * Q[:, j]
    hr[-1] = norm(v)
    return v / hr[-1], hr


# ------------------------------------------------------------------ #
#    GMRES
# ------------------------------------------------------------------ #
def GMRES(A, b, x0, tol=1e-6, maxiter=None, N=None, sym=False):
    """
    solves a system Ax = b, where A is assumed to be invertible,
    using QR-based GMRES
    Parameters
    ----------
    A : python function
        for evaluating the matrix-vector product
    b : (n,) numpy.ndarray
         right-hand side
    x0: (n,) numpy.ndarray
         initial guess
    tol : float
          iteration stops if ||Axk - b|| < tol, tol = 1e-6
    maxiter : int (optional)
              maximum number of iterations
    N : python function (optional)
        for evaluation matrix-vector product of preconditioner
    sym : bool (optional)
          indicating whether A is symmetric or not
          if sym=True: Lanczos is used over Arnoldi

    Returns
    -------
    x : (n,) numpy.ndarray
        approximate solution to Ax=b, with ||Ax-b||<tol
    info : dict
    """
```

```
    # account for initial guess
    boriginal = b
    b = boriginal - A(x0)
    # account for preconditioner A(v) := A(N(v))
    if N:
        Aoriginal = A

        def A(x):
            return N(Aoriginal(x))
        b = N(b)

    # Initializing
    Q = b / norm(b)
    Q = Q[:, np.newaxis]
    v = A(Q[:, 0])
    tQ, tR = v / norm(v), norm(v)
    tQ = tQ[:, np.newaxis]
    n = len(b)
    xr = np.zeros(n)

    # if A is symmetric we (can) use Lanczos instead of Arnoldi
    if sym:
        def ortho(Q, v):
            return Lanczos_step(Q, v)
    else:
        def ortho(Q, v):
            return Arnoldi_step(Q, v)

    if maxiter:
        maxiter = min(maxiter, n+1)
    else:
        maxiter = n+1

    for r in range(1, maxiter):
        # STEP 1: Find next orthonormal basis vector of Krylov subspace
        qr, hr = ortho(Q, v)  # we do not use hr in our variant
        Q = np.hstack((Q, qr[:, np.newaxis]))
        v = A(Q[:, -1])

        # STEP 2: Find QR decomposition of AQ_r by appending previous one
        tqr, trr = ortho(tQ, v)
        tQ = np.hstack((tQ, tqr[:, np.newaxis]))
        tR = np.hstack((np.vstack((tR, np.zeros((1, r)))), trr[:, np.newaxis]))

        # STEP 3: Solve least squares problem involving
        #         AQ_k using its QR-decomposition
        cr = linalg.solve_triangular(tR, tQ.T@b)
        xr = Q @ cr

        # Evaluate current Error and break if its small enough
        lsq_err = norm(boriginal - Aoriginal(x0+xr))

        # collect some infos
        info = dict(iterationCount=r+1,
                    dimension=n,
                    residualNorm=lsq_err)
        if lsq_err < tol:
            break
    return x0 + xr, info


def main():
```

```
    # ------------- #
    #    SETTING
    # ------------- #
    # generate the random system matrix and rhs, as well as initial guess
    start_time = time()
    n = 10000  # 10000 # 100000
#     A_sparseMatrix =   (2 * sparse.eye(n, k=0) -
#                         1 * sparse.eye(n , k=1) -
#                         1 * sparse.eye(n , k=-1))
    # we regularize the matrix to prevent ill-conditioned systems
    diagonals = [np.random.rand(n) + 1.5 * np.ones(n),
                 np.random.rand(n-1),
                 np.random.rand(n-1)]
    A_sparseMatrix = scipy.sparse.diags(diagonals, [0, 1, -1]).tocsr()
    print("\n Dimension:", n)
    print(f" Time to generate the matrix: {time()-start_time:0.2f} [s]")

    def A_function(x):  # The matrix vector product as a function
        return A_sparseMatrix.dot(x)
    b = np.random.rand(n)  # np.ones(n) #
    x0 = np.random.rand(n)  # np.zeros(n) # b#*100 #

    # GMRES parameters
    tol = 1e-06
    maxiter = 1000
    sym = False

    # runtime parameter
    compare_LU_dense = 1
    compare_Scipy = 0
    for preconditionerChoice in ["none", "Jacobi", "GS"]:
        print("\n"+"*"*35+"\n\t\t\t"+preconditionerChoice+"\n"+"*"*35)

        # --------------------- #
        #    Preconditioner
        # --------------------- #
        # Jacobi
        precondJacobiArray = sparse.diags(1. / A_sparseMatrix.diagonal())
        def precondJacobi(b): return precondJacobiArray.dot(b)
        # none
        def precondNone(b): return b
        # Gauss-Seidel
        precondGSinvArray = scipy.sparse.tril(A_sparseMatrix).tocsr()
        def precondGS(b): return spla.spsolve(precondGSinvArray, b)
        # choice
        preconditionerFunctionDict = {"none": precondNone,
                                      "Jacobi": precondJacobi,
                                      "GS": precondGS}
        preconditionerFunction = \
                               preconditionerFunctionDict[preconditionerChoice]

        # ------------------------------ #
        #    Our GMRES
        # ------------------------------ #
        print("-" * 30 + "\n\t\t Our GMRES \n" + "-" * 30)
        start_time = time()
        xk, info = GMRES(A_function, b, x0=x0, tol=tol,
                         maxiter=maxiter, N=preconditionerFunction, sym=sym)
        print(f" Solving time = {time()-start_time:0.2f} [s]")
        print(" Number of iterations:\t", info["iterationCount"],
              f"\n Residual norm: {info['residualNorm']:0.2e}\n")
#         print(" 'Ax = b' is", np.allclose(A_function(xk), b))
```

```
        if compare_Scipy:
            # ------------------------------ #
            #    Scipy's GMRES
            # ------------------------------ #
            print("-" * 30 + "\n\t\t SciPy's GMRES \n" + "-" * 30)

            # callback for gmres
            class gmres_counter(object):
                def __init__(self):
                    self.niter = 0

                def __call__(self, rk=None):
                    self.niter += 1

            counter = gmres_counter()
            start_time = time()
            x, eC = \
            spla.gmres(A_sparseMatrix, b, x0=x0, tol=tol, maxiter=maxiter,
                       M=spla.LinearOperator((n, n), preconditionerFunction),
                       callback=counter)
            print(" Successful exit: ", eC == 0)
            print(f"\n Solving time = {time()-start_time:0.2f} [s]")
            print(" Number of iterations:\t", counter.niter)
            print(f" Residual norm: {norm(b - A_sparseMatrix.dot(x)):0.2e}")
#            print("\n 'Ax = b' is", np.allclose(A_sparseMatrix.dot(x), b))

    # ------------------------------- #
    #    Compare to Scipy's LU (dense)
    # ------------------------------- #
    if compare_LU_dense:
        print("\n" + "-" * 30 + "\n\t\t LU dense \n" + "-" * 30)
        start_time = time()
        xk = linalg.solve(A_sparseMatrix.toarray(), b)
        print(f" Solving time = {time()-start_time:0.2f} [s]")


if __name__ == "__main__":
    main()
```