1 Solving Linear Systems using QR Decomposition: Factorize and Solve

[Diese Aufgabe nicht mehr stellen, da qr-factor ja schon gemacht und qr-solve exakt solve-triangular ist daher ebenfalls erledigt. daher das besser in die prog aufgabe "curve-fitting mit qr" gesetzt!]

Let $A \in \mathbb{R}^{m\times n}$ be a matrix with $n \leq m$ and linearly independent columns (this implies R is invertible) and let $b \in \mathbb{R}^m$. Then, using a QR decomposition $A = QR$, we can compute the solution x of $Ax = b$ (basically in two steps) by solving

$$Rx = Q^T b.$$

Tasks:

1. Implement a function `factor_qr(A)` which computes a reduced QR decomposition of a matrix $A \in \mathbb{R}^{m\times n}$. Thus, it shall output an orthogonal matrix $Q \in \mathbb{R}^{m\times n}$ and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$, so that $A = QR$.
   You can copy the Gram-Schmidt algorithm implemented as a function `QR(A)` *from previous sheets or find an appropriate SciPy Routine.*

2. Implement a function `solve_qr((Q, R), b)` which takes as input the matrices $Q \in \mathbb{R}^{m\times n}$ and $R \in \mathbb{R}^{n\times n}$ computed by `factor_qr(A)` in form of a tuple, as well as a vector $b \in \mathbb{R}^m$. It shall then apply the solving procedure above and output the solution x of $Ax = b$.
   You can recycle the function `solve_tri(A, b, lower=False)` *from previous sheets or use an appropriate SciPy routine for triangular systems.*

3. Test your routine on multiple examples.

Solution:

```python
import numpy as np


def factor_qr(A, own=False):
    """
    computes reduced QR decomposition A=QR of a (mxn) matrix A with m >= n

    INPUT:
        A : numpy.ndarray of shape (m,n), m >= n
        own : switch to use either our or SciPy's routine
    OUTPUT:
        Q : orthogonal matrix Q as numpy.ndarray of shape (m,n)
        R : upper triangular matrix R as numpy.ndarray of shape (n,n)
    """
    m, n = A.shape
    if own:
        # import your own routine to compute reduced QR-decomposition
        # see previous sheets using Gram-Schmidt Algorithm
        pass
    else:
        import scipy.linalg as linalg
        Q, R = linalg.qr(A)
        # attention: SciPy computes full QR decomposition, i.e.,
        # Q is extended to an orthogonal (mxm) matrix
        # R is extended by zeroes to a (mxn) matrix
        # therefore we need to slice the output to obtain a redcued QR-decomp.
```

```
    return Q[0:m,0:n], R[0:n,0:n]

def solve_qr(QR, b, own=False):
    """
    solves a system Ax = b where A = QR
    Assumptions:
        A is expected to have shape (m, n) with m >= n
        the columns of A are indepenent, so that R is invertible
    Remark:
        if A is not square, then linear least square solution is computed

    INPUT:
        QR=(Q,R) : tuple containing
             Q orthogonal matrix Q as numpy.ndarray of shape (m,n)
             R upper triangular matrix R as numpy.ndarray of shape (n,n)
        b : right-hand side vector b as numpy.ndarray of shape (n,)
    OUTPUT:
        x : (least square) solution of Ax = b as numpy.ndarray of shape (n,)
    """
    Q, R = QR
    if own:
        # import your own routine to solve triangular systems here
        pass
    else:
        from scipy.linalg import solve_triangular as solve_tri
        x = solve_tri(R, Q.T @ b )
    return x

if __name__ == "__main__":
    # Test on random data:
    # a few vectors in high-dimensional space are independent with high prob.
    m, n = 1000, 50
    A = np.random.rand(n,n)
    b = np.random.rand(n)

    Q,R = factor_qr(A)
    x = solve_qr((Q,R),b)
    print("Ax = b is", np.allclose(A.dot(x), b, atol = 1e-8))
```