Gram-Schmidt algorithm (QR factorization)

The Gram-Schmidt algorithm is an algorithm that can be used to compute a reduced (sometimes also called "thin" or "economic") QR-decomposition of a matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and $\mathrm{rank}(A) = n$, i.e., of a matrix whose columns are linearly independent.

The basic idea is to successively built up an orthonormal system from a given set of linearly independent vectors; in our case the columns of $A = [a_1, \ldots, a_n] \in \mathbb{R}^{m\times n}$. We choose the first column as starting point for the algorithm and set $\widetilde{q}_1 := a_1$. Of course, in order to generate an orthogonal matrix Q we have to rescale the vector and set $q_1 := \frac{\widetilde{q}_1}{\|\widetilde{q}_1\|}$. The successive vectors $\widetilde{q}_k$ are generated by subtracting all the "shares" $a_k^\top q_\ell \cdot q_\ell$ (=$\mathrm{proj}_{q_\ell}(a_k)$) of the previous vectors q_ℓ from the column a_k, i.e.,

$$\widetilde{q}_k := a_k - \sum_{\ell=1}^{k-1} a_k^\top q_\ell \, q_\ell.$$

The following algorithm computes a *reduced* QR-decomposition of some matrix $A \in \mathbb{R}^{m\times n}$.

$r_{11} \leftarrow \|a_1\|$
$q_1 \leftarrow \frac{a_1}{r_{11}}$

for $k = 2, \ldots, n$
  for $\ell = 1, \ldots, k-1$
    $r_{\ell k} \leftarrow a_k^\top q_\ell$

  $\widetilde{q}_k \leftarrow a_k - \sum_{\ell=1}^{k-1} r_{\ell k}\, q_\ell$
  $r_{kk} \leftarrow \|\widetilde{q}_k\|$
  $q_k \leftarrow \frac{\widetilde{q}_k}{r_{kk}}$

Task:

1. Implement the Gram-Schmidt algorithm as a function `qr_factor(A)`, which takes a matrix A as input and returns the matrices $\widehat{Q}$ and $\widehat{R}$.

2. Run your algorithm on an example matrix (e.g., `numpy.random.rand(m,n)`) and test your result by computing $\widehat{Q}^\top \widehat{Q}$ and $\widehat{Q}\widehat{R} - A$, where the first should yield the identity and the latter a zero-matrix.

3. Find a SciPy routine to compute the QR decomposition (for the reduced QR you may need to set the parameters accordingly).

Hint: You can use `numpy.allclose()` to check whether two `numpy.ndarray`'s are equal up to a certain tolerance.

Solution:

```python
import numpy as np
import scipy.linalg as la

def qr_factor(A):
    """
    Computes a (reduced) QR-decomposition of a (mxn)-matrix with m>=n
    via Gram-Schmidt Algorithm.

    Parameters
    ----------
    A : (mxn) matrix with m>=n

    Returns
    -------
    Q : (mxn) with orthonormal columns
```

```
    R : (nxn) upper triangular matrix
    """
    m, n = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((m, n))

    R[0, 0] = np.linalg.norm(A[:, 0])
    Q[:, 0] = A[:, 0] / R[0, 0]

    for k in range(1, n):
        for l in range(0, k):
            R[l, k] = A[:, k] @ Q[:, l]
        q = A[:, k] - Q @ R[:, k]
        R[k, k] = np.linalg.norm(q)
        Q[:, k] = q / R[k, k]

    return Q, R


if __name__ == "__main__":
    # Example
    m, n = 4, 2
    A = np.random.rand(m, n)
    print("A = \n", A)
    # Another Example
    A = np.array([[3, 1], [1, 2]])
    print("A = \n", A)
    Q, R = qr_factor(A)
    Q2, R2 = la.qr(A) #, mode='economic')  # Compare to SciPy
    print("Ours:\n-------\nQ = \n", np.round(Q, 2), "\nR = \n", np.round(R, 2))
    print("Sc:\n-------\nQ = \n", np.round(Q2, 2), "\nR = \n", np.round(R2, 2))
    print("\nTest 1: Q^TQ = I is", np.allclose(Q.transpose()@Q, np.eye(n)))
    print("\nTest 2: QR = A is", np.allclose(Q@R, A))
```