Lab 7

QR 1: Decomposition

Lab Objective: *The QR decomposition is a fundamentally important matrix factorization. It is straightforward to implement, is numerically stable, and provides the basis of several important algorithms. In this lab, we explore several ways to produce the QR decomposition and implement a few immediate applications.*

We restrict our discussion to real matrices. However, the results and algorithms presented here can be extended to complex matrices by replacing "transpose" with "hermitian conjugate" and "symmetric matrix" with "hermitian matrix."

The QR decomposition of a matrix A is a factorization $A = QR$, where Q is has orthonormal columns and R is upper triangular. Every $m \times n$ matrix A of rank $n \leq m$ has a QR decomposition, with two main forms.

- **Reduced QR**: Q is $m \times n$, R is $n \times n$, and the columns of Q $\{\mathbf{q}_j\}_{j=1}^{n}$ form an orthonormal basis for the column space of A.

- **Full QR**: Q is $m \times m$ and R is $m \times n$. In this case, the columns of Q $\{\mathbf{q}_j\}_{j=1}^{m}$ form an orthonormal basis for all of $\mathbb{F}^m$, and the last $m - n$ rows of R only contain zeros. If $m = n$, this is the same as the reduced factorization.

We distinguish between these two forms by writing $\widehat{Q}$ and $\widehat{R}$ for the reduced decomposition and Q and R for the full decomposition.

$$\underbrace{\begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n & \mathbf{q}_{n+1} & \cdots & \mathbf{q}_m \end{bmatrix}}_{Q\ (m \times m)} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix}}_{R\ (m \times n)} = A\ (m \times n)$$

QR via Gram-Schmidt

The *classical Gram-Schmidt algorithm* takes a linearly independent set of vectors and constructs an orthonormal set of vectors with the same span. Applying Gram-Schmidt to the columns of A, which are linearly independent since A has rank n, results in the columns of Q.

Let $\{\mathbf{x}_j\}_{j=1}^n$ be the columns of A. Define

$$\mathbf{q}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|}, \qquad \mathbf{q}_k = \frac{\mathbf{x}_k - \mathbf{p}_{k-1}}{\|\mathbf{x}_k - \mathbf{p}_{k-1}\|}, \quad k = 2,\ \ldots,\ n,$$

$$\mathbf{p}_0 = \mathbf{0},\ \text{and} \qquad \mathbf{p}_{k-1} = \sum_{j=1}^{k-1} \langle \mathbf{q}_j, \mathbf{x}_k \rangle \mathbf{q}_j, \quad k = 2,\ \ldots,\ n.$$

Each $\mathbf{p}_{k-1}$ is the projection of $\mathbf{x}_k$ onto the span of $\{\mathbf{q}_j\}_{j=1}^{k-1}$, so $\mathbf{q}_k' = \mathbf{x}_k - \mathbf{p}_{k-1}$ is the residual vector of the projection. Thus $\mathbf{q}_k'$ is orthogonal to each of the $\{\mathbf{q}_j\}_{j=1}^{k-1}$. Therefore, normalizing each $\mathbf{q}_k'$ produces an orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$.

To construct the reduced QR decomposition, let $\widehat{Q}$ be the matrix with columns $\{\mathbf{q}_j\}_{j=1}^n$, and let $\widehat{R}$ be the upper triangular matrix with the following entries:

$$r_{kk} = \|\mathbf{x}_k - \mathbf{p}_{k-1}\|, \qquad r_{jk} = \langle \mathbf{q}_j, \mathbf{x}_k \rangle = \mathbf{q}_j^\mathsf{T} \mathbf{x}_k,\ j < k.$$

This clever choice of entries for $\widehat{R}$ reverses the Gram-Schmidt process and ensures that $\widehat{Q}\widehat{R} = A$.

Modified Gram-Schmidt

If the columns of A are close to being linearly dependent, the classical Gram-Schmidt algorithm often produces a set of vectors $\{\mathbf{q}_j\}_{j=1}^n$ that are not even close to orthonormal due to rounding errors. The *modified Gram-Schmidt algorithm* is a slight variant of the classical algorithm which more consistently produces a set of vectors that are "very close" to orthonormal.

Let $\mathbf{q}_1$ be the normalization of $\mathbf{x}_1$ as before. Instead of making just $\mathbf{x}_2$ orthogonal to $\mathbf{q}_1$, make *each* of the vectors $\{\mathbf{x}_j\}_{j=2}^n$ orthogonal to $\mathbf{q}_1$:

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_1, \mathbf{x}_k \rangle \mathbf{q}_1, \quad k = 2,\ \ldots,\ n.$$

Next, define $\mathbf{q}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|}$. Proceed by making each of $\{\mathbf{x}_j\}_{j=3}^n$ orthogonal to $\mathbf{q}_2$:

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_2, \mathbf{x}_k \rangle \mathbf{q}_2, \quad k = 3, \ldots, n.$$

Since each of these new vectors is a linear combination of vectors orthogonal to $\mathbf{q}_1$, they are orthogonal to $\mathbf{q}_1$ as well. Continuing this process results in the desired orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$. The entire modified Gram-Schmidt algorithm is described below in Algorithm 7.1.

Algorithm 7.1

1: **procedure** MODIFIED GRAM-SCHMIDT(A)
2: $m, n \leftarrow \text{shape}(A)$ ▷ Store the dimensions of A.
3: $Q \leftarrow \text{copy}(A)$ ▷ Make a copy of A with `np.copy()`.
4: $R \leftarrow \text{zeros}(n, n)$ ▷ An $n \times n$ array of all zeros.
5: **for** $i = 0 \ldots n-1$ **do**
6: $R_{i,i} \leftarrow \|Q_{:,i}\|$
7: $Q_{:,i} \leftarrow Q_{:,i}/R_{i,i}$ ▷ Normalize the ith column of Q.
8: **for** $j = i+1 \ldots n-1$ **do**
9: $R_{i,j} \leftarrow Q_{:,j}^{\mathsf{T}} Q_{:,i}$
10: $Q_{:,j} \leftarrow Q_{:,j} - R_{i,j} Q_{:,i}$ ▷ Orthogonalize the jth column of Q.
11: **return** Q, R

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n. Use Algorithm 7.1 to compute the reduced QR decomposition of A.

Consider the following tips for implementing the algorithm.

- In Python, the operation `a = a + b` can also be written as `a += b`.
- Use `scipy.linalg.norm()` to compute the norm of the vector in step 6.
- Note that steps 7 and 10 employ scalar multiplication or division, while step 9 uses vector multiplication.

To test your function, generate test cases with NumPy's `np.random` module. Verify that R is upper triangular, Q is orthonormal, and $QR = A$. You may also want to compare your results to SciPy's QR factorization algorithm.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate a random matrix and get its reduced QR decomposition via SciPy.
>>> A = np.random.random((6,4))
>>> Q,R = la.qr(A, mode="economic") # Use mode="economic" for reduced QR.
>>> print A.shape, Q.shape, R.shape
(6,4) (6,4) (4,4)

# Verify that R is upper triangular, Q is orthonormal, and QR = A.
>>> np.allclose(np.triu(R), R)
True
>>> np.allclose(np.dot(Q.T, Q), np.identity(4))
True
>>> np.allclose(np.dot(Q, R), A)
True
```

Consequences of the QR Decomposition

The special structures of Q and R immediately provide some simple applications.

Determinants

Let A be $n \times n$. Then Q and R are both $n \times n$ as well.[1] Since Q is orthonormal and R is upper-triangular,

$$\det(Q) = \pm 1 \qquad\qquad \det(R) = \prod_{i=1}^{n} r_{i,i}$$

Then since $\det(AB) = \det(A)\det(B)$, we have the following:

$$|\det(A)| = |\det(QR)| = |\det(Q)\det(R)| = \left|\prod_{k=1}^{n} r_{kk}\right|$$

Problem 2. Write a function that accepts an invertible $n \times n$ matrix A. Use the QR decomposition of A to calculate $|\det(A)|$.

You may use your QR decomposition algorithm from Problem 1 or SciPy's QR routine. Can you implement this function in a single line?

Linear Systems

The LU decomposition is usually the matrix factorization of choice to solve the linear system $A\mathbf{x} = \mathbf{b}$ because the triangular structures of L and U facilitate forward and backward substitution. However, the QR decomposition avoids the potential numerical issues that come with Gaussian elimination.

Since Q is orthonormal, $Q^{-1} = Q^\mathsf{T}$. Therefore, solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving the system $R\mathbf{x} = Q^\mathsf{T}\mathbf{b}$. Since R is upper-triangular, $R\mathbf{x} = Q^\mathsf{T}\mathbf{b}$ can be solved quickly with back substitution.[2]

Problem 3. Write a function that accepts an invertible $n \times n$ matrix A and a vector $\mathbf{b}$ of length n. Use the QR decomposition to solve $A\mathbf{x} = \mathbf{b}$ in the following steps:

1. Compute Q and R.
2. Calculate $\mathbf{y} = Q^\mathsf{T}\mathbf{b}$.
3. Use back substitution to solve $R\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

[1] An $n \times n$ orthonormal matrix is sometimes called *unitary* in other texts.

[2] See Problem 3 of the Linear Systems lab for a refresher on back substitution.

QR via Householder

The Gram-Schmidt algorithm orthonormalizes A using a series of transformations that are stored in an upper triangular matrix. Another way to compute the QR decomposition is to take the opposite approach: triangularize A through a series of orthonormal transformations. Orthonormal transformations are numerically stable, meaning that they are less susceptible to rounding errors. In fact, this approach is usually faster and more accurate than Gram-Schmidt methods.

The idea is for the kth orthonormal transformation Q_k to map the kth column of A to the span of $\{\mathbf{e}_j\}_{j=1}^{k}$, where the $\mathbf{e}_j$ are the standard basis vectors in $\mathbb{R}^m$. In addition, to preserve the work of the previous transformations, Q_k should not modify any entries of A that are above or to the left of the kth diagonal term of A. For a 4×3 matrix A, the process can be visualized as follows.

$$Q_3Q_2Q_1 \begin{bmatrix} \boxed{\begin{matrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{matrix}} \end{bmatrix} = Q_3Q_2 \begin{bmatrix} * & * & * \\ 0 & \boxed{\begin{matrix} * & * \\ * & * \\ * & * \end{matrix}} \\ 0 & \\ 0 & \end{bmatrix} = Q_3 \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & \boxed{\begin{matrix} * \\ * \end{matrix}} \\ 0 & 0 & \end{bmatrix} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{bmatrix}$$

Thus $Q_3Q_2Q_1A = R$, so that $A = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}R$ since each Q_k is orthonormal. Furthermore, the product of square orthonormal matrices is orthonormal, so setting $Q = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}$ yields the full QR decomposition.

How to correctly construct each Q_k isn't immediately obvious. The ingenious solution lies in one of the basic types of linear transformations: reflections.

Householder Transformations

The *orthogonal complement* of a nonzero vector $\mathbf{v} \in \mathbb{R}^n$ is the set of all vectors $\mathbf{x} \in \mathbb{R}^n$ that are orthogonal to $\mathbf{v}$, denoted $\mathbf{v}^\perp = \{\mathbf{x} \in \mathbb{R}^n \mid \langle \mathbf{x}, \mathbf{v} \rangle = 0\}$. A *Householder transformation* is a linear transformation that reflects a vector $\mathbf{x}$ across the orthogonal complement $\mathbf{v}^\perp$ for some specified $\mathbf{v}$.

The matrix representation of the Householder transformation corresponding to $\mathbf{v}$ is given by $H_\mathbf{v} = I - 2\frac{\mathbf{v}\mathbf{v}^\mathsf{T}}{\mathbf{v}^\mathsf{T}\mathbf{v}}$. Since $H_\mathbf{v}^\mathsf{T}H_\mathbf{v} = I$, Householder transformations are orthonormal.

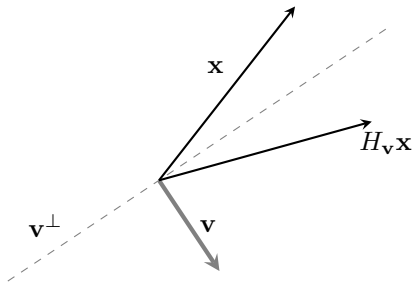

Figure 7.1: The vector $\mathbf{v}$ defines the orthogonal complement $\mathbf{v}^\perp$. Applying the Householder transformation $H_\mathbf{v}$ to $\mathbf{x}$ reflects $\mathbf{x}$ across $\mathbf{v}^\perp$.

Householder Triangularization

The *Householder algorithm* uses Householder transformations for the orthonormal transformations in the QR decomposition process described on the previous page. The goal in choosing Q_k is to send $\mathbf{x}_k$, the kth column of A, to the span of $\{\mathbf{e}_j\}_{j=1}^{k}$. In other words, if $Q_k\mathbf{x}_k = \mathbf{y}_k$, the last $m-k$ entries of $\mathbf{y}_k$ should be 0.

$$Q_k\mathbf{x}_k = Q_k \begin{bmatrix} z_1 \\ \vdots \\ z_k \\ z_{k+1} \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{y}_k$$

To begin, decompose $\mathbf{x}_k$ into $\mathbf{x}_k = \mathbf{x}_k' + \mathbf{x}_k''$, where $\mathbf{x}_k'$ and $\mathbf{x}_k''$ are of the form

$$\mathbf{x}_k' = [z_1 \quad \cdots \quad z_{k-1} \quad 0 \quad \cdots \quad 0]^\mathsf{T} \qquad \text{and} \qquad \mathbf{x}_k'' = [0 \quad \cdots \quad 0 \quad z_k \quad \cdots \quad z_m]^\mathsf{T}.$$

Because $\mathbf{x}_k'$ represents elements of A that lie above the diagonal, only $\mathbf{x}_k''$ needs to be altered by the reflection.

The two vectors $\mathbf{x}_k'' \pm \|\mathbf{x}_k''\|\mathbf{e}_k$ both yield Householder transformations that send $\mathbf{x}_k''$ to the span of $\mathbf{e}_k$ (see Figure 7.2). Between the two, the one that reflects $\mathbf{x}_k''$ further is more numerically stable. This reflection corresponds to

$$\mathbf{v}_k = \mathbf{x}_k'' + \operatorname{sign}(z_k)\|\mathbf{x}_k''\|\mathbf{e}_k,$$

where z_k is the first nonzero component of $\mathbf{x}_k''$ (the kth component of $\mathbf{x}_k$).

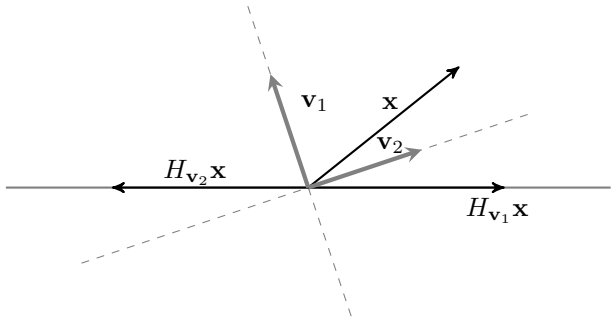

Figure 7.2: There are two possible reflections that map $\mathbf{x}$ into the span of $\mathbf{e}_1$, defined by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. In this illustration, $H_{\mathbf{v}_2}$ is the more stable transformation since it reflects $\mathbf{x}$ further than $H_{\mathbf{v}_1}$.

After choosing $\mathbf{v}_k$, set $\mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}$. Then $H_{\mathbf{v}_k} = I - 2\frac{\mathbf{v}_k\mathbf{v}_k^\mathsf{T}}{\|\mathbf{v}_k\|^2} = I - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T}$, and hence Q_k is given by the following block matrix.

$$Q_k = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & I_{m-k+1} - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T} \end{bmatrix}$$

Here I_p denotes the $p \times p$ identity matrix, and thus each Q_k is $m \times m$.

It is apparent from its form that Q_k does not affect the first $k-1$ rows and columns of any matrix that it acts on. Then by starting with $R = A$ and $Q = I$, at each step of the algorithm we need only multiply the entries in the lower right $(m-k+1) \times (m-k+1)$ submatrices of R and Q by $I - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T}$. This completes the Householder algorithm, detailed below.

Algorithm 7.2

1: **procedure** HOUSEHOLDER(A)
2: $m, n \leftarrow \text{shape}(A)$
3: $R \leftarrow \text{copy}(A)$
4: $Q \leftarrow I_m$ ▷ The $m \times m$ identity matrix.
5: **for** $k = 0 \dots n-1$ **do**
6: $\mathbf{u} \leftarrow \text{copy}(R_{k:,k})$
7: $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$ ▷ u_0 is the first entry of $\mathbf{u}$.
8: $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$ ▷ Normalize $\mathbf{u}$.
9: $R_{k:,k:} \leftarrow R_{k:,k:} - 2\mathbf{u}\left(\mathbf{u}^\mathsf{T} R_{k:,k:}\right)$ ▷ Apply the reflection to R.
10: $Q_{k:,:} \leftarrow Q_{k:,:} - 2\mathbf{u}\left(\mathbf{u}^\mathsf{T} Q_{k:,:}\right)$ ▷ Apply the reflection to Q.
11: **return** Q^T, R

Problem 4. Write a function that accepts as input a $m \times n$ matrix A of rank n. Use Algorithm 7.2 to compute the full QR decomposition of A.

Consider the following implementation details.

- NumPy's `np.sign()` is an easy way to implement the sign() operation in step 7. However, `np.sign(0)` returns 0, which will cause a problem in the rare case that $u_0 = 0$ (which is possible if the top left entry of A is 0 to begin with). The following code defines a function that returns the sign of a single number, counting 0 as positive.

```
sign = lambda x: 1 if x >= 0 else -1
```

- In steps 9 and 10, the multiplication of $\mathbf{u}$ and $(\mathbf{u}^\mathsf{T}X)$ is an *outer product* ($\mathbf{x}\mathbf{y}^\mathsf{T}$ instead of the usual $\mathbf{x}^\mathsf{T}\mathbf{y}$). Use `np.outer()` instead of `np.dot()` to handle this correctly.

As with Problem 1, use NumPy and SciPy to generate test cases and validate your function.

```
>>> A = np.random.random((5, 3))
>>> Q,R = la.qr(A)                  # Get the full QR decomposition.
>>> print A.shape, Q.shape, R.shape
(5,3) (5,5) (5,3)
>>> np.allclose(Q.dot(R), A)
True
```

Upper Hessenberg Form

An *upper Hessenberg matrix* is a square matrix that is nearly upper triangular, with zeros below the first subdiagonal. Every $n \times n$ matrix A can be written $A = QHQ^\mathsf{T}$ where Q is orthonormal and H, called the *Hessenberg form* of A, is an upper Hessenberg matrix. Putting a matrix in upper Hessenberg form is an important first step to computing its eigenvalues numerically.

This algorithm also uses Householder transformations. To find orthogonal Q and upper Hessenberg H such that $A = QHQ^\mathsf{T}$, it suffices to find such matrices that satisfy $Q^\mathsf{T}AQ = H$. Thus, the strategy is to multiply A on the left and right by a series of orthonormal matrices until it is in Hessenberg form.

Using the same Q_k as in the kth step of the Householder algorithm introduces $n-k$ zeros in the kth column of A, but multiplying Q_kA on the right by Q_k^T destroys all of those zeros. Instead, choose a Q_1 that fixes $\mathbf{e}_1$ and reflects the first column of A into the span of $\mathbf{e}_1$ and $\mathbf{e}_2$. The product Q_1A then leaves the first row of A alone, and the product $(Q_1A)Q_1^\mathsf{T}$ leaves the first column of (Q_1A) alone.

$$\underset{A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix}} \xrightarrow{Q_1^\mathsf{T}} \underset{(Q_1A)Q_1^\mathsf{T}}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix}}$$

Continuing the process results in the upper Hessenberg form of A.

$$Q_3Q_2Q_1AQ_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

This implies that $A = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}HQ_3Q_2Q_1$, so setting $Q = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}$ results in the desired factorization $A = QHQ^\mathsf{T}$.

Constructing the Reflections

Constructing the Q_k uses the same approach as in the Householder algorithm, but shifted down one element. Let $\mathbf{x}_k = \mathbf{y}_k' + \mathbf{y}_k''$ where $\mathbf{y}_k'$ and $\mathbf{y}_k''$ are of the form

$$\mathbf{y}_k' = [z_1 \quad \cdots \quad z_k \quad 0 \quad \cdots \quad 0]^\mathsf{T} \qquad \text{and} \qquad \mathbf{y}_k'' = [0 \quad \cdots \quad 0 \quad z_{k+1} \quad \cdots \quad z_m]^\mathsf{T}.$$

Because $\mathbf{y}_k'$ represents elements of A that lie above the first subdiagonal, only $\mathbf{y}_k''$ needs to be altered. This suggests using the following reflection.

$$\mathbf{v}_k = \mathbf{y}_k'' + \operatorname{sign}(z_k)\|\mathbf{y}_k''\|\mathbf{e}_k \qquad\qquad \mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}$$

$$Q_k = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & I_{m-k} - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T} \end{bmatrix}$$

The complete algorithm is given below. Note how similar it is to Algorithm 7.2.

Algorithm 7.3

procedure HESSENBERG(A)
  $m, n \leftarrow \text{shape}(A)$
  $H \leftarrow \text{copy}(A)$
  $Q \leftarrow I_m$
  for $k = 0 \ldots n-3$ **do**
    $\mathbf{u} \leftarrow \text{copy}(H_{k+1:,k})$
    $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$
    $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$
    $H_{k+1:,k:} \leftarrow H_{k+1:,k:} - 2\mathbf{u}(\mathbf{u}^\mathsf{T} H_{k+1:,k:})$ ▷ Apply Q_k to H.
    $H_{:,k+1:} \leftarrow H_{:,k+1:} - 2(H_{:,k+1:}\mathbf{u})\mathbf{u}^\mathsf{T}$ ▷ Apply Q_k^T to H.
    $Q_{k+1:,:} \leftarrow Q_{k+1:,:} - 2\mathbf{u}(\mathbf{u}^\mathsf{T} Q_{k+1:,:})$ ▷ Apply Q_k to Q.
  return H, Q^T

Problem 5. Write a function that accepts a nonsingular $n \times n$ matrix A. Use Algorithm 7.3 to compute its upper Hessenberg form, upper Hessenberg H and orthogonal Q satisfying $A = QHQ^\mathsf{T}$.

Test your function and compare your results to `scipy.linalg.hessenberg()`.

```
# Generate a random matrix and get its upper Hessenberg form via SciPy.
>>> A = np.random.random((8,8))
>>> H, Q = la.hessenberg(A, calc_q=True)

# Verify that H has all zeros below the first subdiagonal and QHQ^T = A.
>>> np.allclose(np.triu(H, -1), H)
True
>>> np.allclose(np.dot(np.dot(Q, H), Q.T), A)
True
```

NOTE

When A is symmetric, its upper Hessenberg form is a tridiagonal matrix. This is because the Q_k's zero out everything below the first subdiagonal of A and the Q_k^T's zero out everything to the right of the first superdiagonal. Tridiagonal matrices make computations fast, so the computing the Hessenberg form of a symmetric matrix is very useful.

Additional Material

Complex QR Decomposition

The QR decomposition also exists for matrices with complex entries. The standard inner product in $\mathbb{R}^m$ is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\mathsf{T}\mathbf{y}$, but the (more general) standard inner product in $\mathbb{C}^m$ is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\mathsf{H}\mathbf{y}$. The $^\mathsf{H}$ stands for the *Hermitian conjugate*, the conjugate of the transpose. Making a few small adjustments in the implementations of Algorithms 7.1 and 7.2 accounts for using the complex inner product.

1. Replace any transpose operations with the conjugate of the transpose.

```
>>> A = np.reshape(np.arange(4) + 1j*np.arange(4), (2,2))
>>> print(A)
[[ 0.+0.j  1.+1.j]
 [ 2.+2.j  3.+3.j]]

>>> print(A.T)                      # Regular transpose.
[[ 0.+0.j  2.+2.j]
 [ 1.+1.j  3.+3.j]]

>>> print(A.conj().T)               # Hermitian conjugate.
[[ 0.-0.j  2.-2.j]
 [ 1.-1.j  3.-3.j]]
```

2. Conjugate the first entry of vector or matrix multiplication before multiplying with `np.dot()`.

```
>>> x = np.arange(2) + 1j*np.arange(2)
>>> print(x)
[ 0.+0.j  1.+1.j]

>>> np.dot(x, x)                    # Standard real inner product.
2j

>>> np.dot(x.conj(), y)             # Standard complex inner product.
(2 + 0j)
```

3. In the complex plane, there are infinitely many reflections that map a vector $\mathbf{x}$ into the span of $\mathbf{e}_k$, not just the two displayed in Figure 7.2. Using $\text{sign}(z_k)$ to choose one is still a valid method, but it requires updating the `sign()` function so that it can handle complex numbers.

```
sign = lambda x: 1 if np.real(x) >= 0 else -1
```

QR with Pivoting

The LU decomposition can be improved by employing Gaussian elimination with partial pivoting, where the rows of A are strategically permuted at each iteration. The QR factorization can be similarly improved by permuting the columns of A at each iteration. The result is the factorization $AP = QR$, where P is a permutation matrix that encodes the column swaps. SciPy's `scipy.linalg.qr()` can compute the pivoted QR decomposition.

```
# Get the decomposition AP = QR for a random matrix A.
>>> A = np.random.random((8,10))
>>> Q,R,P = la.qr(A, pivoting=True)

# P is returned as a 1-D array that encodes column ordering,
# so A can be reconstructed with fancy indexing.
>>> np.allclose(Q.dot(R), A[:,P])
True
```

QR via Givens

The Householder algorithm uses reflections to triangularize A. However, A can also be made upper triangular using rotations. To illustrate the idea, recall that the following matrix represents a counterclockwise rotation of θ radians.

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

This transformation is orthonormal. Given $\mathbf{x} = [a, b]^\mathsf{T}$, if θ is the angle between $\mathbf{x}$ and $\mathbf{e}_1$, then $R_{-\theta}$ maps $\mathbf{x}$ to the span of $\mathbf{e}_1$.

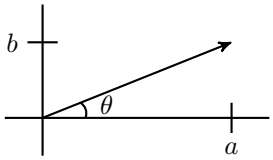

Figure 7.3: Rotating clockwise by θ sends the vector $[a, b]^\mathsf{T}$ to the span of $\mathbf{e}_1$.

In terms of a and b, $\cos\theta = \frac{a}{\sqrt{a^2+b^2}}$ and $\sin\theta = \frac{b}{\sqrt{a^2+b^2}}$. Therefore,

$$R_{-\theta}\mathbf{x} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \frac{a}{\sqrt{a^2+b^2}} & \frac{b}{\sqrt{a^2+b^2}} \\ -\frac{b}{\sqrt{a^2+b^2}} & \frac{a}{\sqrt{a^2+b^2}} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sqrt{a^2+b^2} \\ 0 \end{bmatrix}.$$

The matrix R_θ above is an example of a 2×2 *Givens rotation matrix*. In general, the Givens matrix $G(i, j, \theta)$ represents the orthonormal transformation that rotates the 2-dimensional span of $\mathbf{e}_i$ and $\mathbf{e}_j$ by θ radians. The matrix representation of this transformation is a generalization of R_θ.

$$G(i,j,\theta) = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}$$

Here I represents the identity matrix, $c = \cos\theta$, and $s = \sin\theta$. The c's appear on the i^{th} and j^{th} diagonal entries.

Givens Triangularization

As demonstrated, θ can be chosen such that $G(i, j, \theta)$ rotates a vector so that its jth-component is 0. Such a transformation will only affect the i^{th} and j^{th} entries of any vector it acts on (and thus the i^{th} and j^{th} rows of any matrix it acts on).

To compute the QR decomposition of A, iterate through the subdiagonal entries of A in the order depicted by Figure 7.4. Zero out the ij^{th} entry with a rotation in the plane spanned by $\mathbf{e}_{i-1}$ and $\mathbf{e}_i$, represented by the Givens matrix $G(i-1, i, \theta)$.

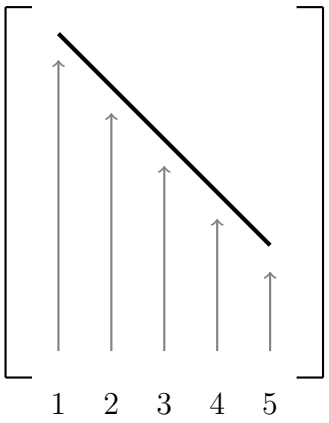

Figure 7.4: The order in which to zero out subdiagonal entries in the Givens triangularization algorithm. The heavy black line is the main diagonal of the matrix. Entries should be zeroed out from bottom to top in each column, beginning with the leftmost column.

On a 2×3 matrix, the process can be visualized as follows.

$$\begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix} \xrightarrow{G(2,3,\theta_1)} \begin{bmatrix} * & * \\ \boxed{*} & \boxed{*} \\ \boxed{0} & * \end{bmatrix} \xrightarrow{G(1,2,\theta_2)} \begin{bmatrix} \boxed{*} & \boxed{*} \\ \boxed{0} & * \\ 0 & * \end{bmatrix} \xrightarrow{G(2,3,\theta_3)} \begin{bmatrix} * & * \\ 0 & \boxed{*} \\ 0 & \boxed{0} \end{bmatrix}$$

At each stage, the boxed entries are those modified by the previous transformation. The final transformation $G(2, 3, \theta_3)$ operates on the bottom two rows, but since the first two entries are zero, they are unaffected.

Assuming that at the ij^{th} stage of the algorithm a_{ij} is nonzero, Algorithm 7.4 computes the Givens triangularization of a matrix. Notice that the algorithm does not actually form the entire matrices $G(i, j, \theta)$; instead, it modifies only those entries of the matrix that are affected by the transformation.

Algorithm 7.4

procedure GIVENS TRIANGULARIZATION(A)
  $m, n \leftarrow \text{shape}(A)$
  $R \leftarrow \text{copy}(A)$
  $Q \leftarrow I_m$
  for $j = 0 \dots n-1$ **do**
    for $i = m-1 \dots j+1$ **do**
      $a, b \leftarrow R_{i-1,j}, R_{i,j}$
      $G \leftarrow [[a, b], [-b, a]]/\sqrt{a^2 + b^2}$
      $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$
      $Q_{i-1:i+1,:} \leftarrow GQ_{i-1:i+1,:}$
  return Q^T, R

QR of a Hessenberg Matrix via Givens

The Givens algorithm is particularly efficient for computing the QR decomposition of a matrix that is already in upper Hessenberg form, since only the first subdiagonal needs to be zeroed out. Algorithm 7.5 details this process.

Algorithm 7.5

procedure GIVENS TRIANGULARIZATION OF HESSENBERG(H)
  $m, n \leftarrow \text{shape}(H)$
  $R \leftarrow \text{copy}(H)$
  $Q \leftarrow I_m$
  for $j = 0 \dots \min\{n-1, m-1\}$ **do**
    $i = j + 1$
    $a, b \leftarrow R_{i-1,j}, R_{i,j}$
    $G \leftarrow [[a, b], [-b, a]]/\sqrt{a^2 + b^2}$
    $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$
    $Q_{i-1:i+1,:i+1} \leftarrow GQ_{i-1:i+1,:i+1}$
  return Q^T, R