In [1]:
```python
import numpy as np
```

# The QR Decomposition

From the SVD, we know that we can always decompose $A \in \mathbb{C}^{m \times n}$ as

$$A = U\Sigma V^* \tag{1}$$

where $U$ and $V$ have orthonomal columns, and $\Sigma$ is a diagonal matrix. By convention, the columns in $U$ and $V$ are organized to correspond to a list of singular values $\sigma_{ii}$ so that $\sigma_{11} \geq \sigma_{22} \geq \cdots \geq \sigma_{rr} > 0$. We also know that $\mathrm{Col}(U) = \mathrm{Col}(A)$. However, we have no guarantee that a the first $j$ columns of $A$ span the same space as the first $j$ columns of $U$.

In the following, assume that $A$ has full column rank $n$. Consider a set of orthonormal vectors $\{\mathbf{q}_1, \mathbf{q}_2, \ldots \mathbf{q}_n\}$ such that

$$\langle \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_j \rangle = \langle \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j \rangle, \qquad j = 1, 2, \ldots, n \tag{2}$$

where $\langle \ldots \rangle$ denotes the space spanned of the enclosed vectors. We can then write

$$\begin{aligned}
\mathbf{a}_1 &= r_{11}\mathbf{q}_1 && (3)\\
\mathbf{a}_2 &= r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 && (4)\\
\cdots & && (5)\\
\mathbf{a}_n &= r_{1n}\mathbf{q}_1 + r_{2n}\mathbf{q}_2 + \cdots + r_{nn}\mathbf{q}_n && (6)
\end{aligned}$$

#### Question

How can we write this in matrix form?

## The QR decomposition

The matrix form suggested by the above set of equations is the $QR$ decompostion, given by

$$A = QR \tag{7}$$

where $Q = [\mathbf{q}_1, \mathbf{q}_2, \ldots \mathbf{q}_n]$ is $m \times n$, and $R$ is an upper triangular $n \times n$ matrix with entries $r_{ij}$.

As with the SVD, we also have a *full* $QR$ decomposition in which $Q$ is a square $m \times m$ matrix, and $R$ is $m \times n$. For this purposes here, however, the $QR$ decomposition will refer to a *reduced* decomposition.

## Gram-Schmidt orthogonalization

How can we compute such vectors $\mathbf{q}_j$ and the entries of $R$? In our first approach, we will use a classical algorithm called the Gram-Schmidt algorithm.

### Step 1: $j = 1$

We know we need $\langle \mathbf{a}_1 \rangle = \langle \mathbf{q}_1 \rangle$. Using our above formulation, we have

$$\mathbf{a}_1 = r_{11}\mathbf{q}_1 \tag{8}$$

#### Question

What is $r_{11}$? What is $\mathbf{q}_1$?

$$r_{11} = \|\mathbf{a}_1\| \qquad \mathbf{q}_1 = \frac{\mathbf{a}_1}{r_{11}} \tag{9}$$

Note : $\mathbf{q}_1^*\mathbf{a}_1 = r_{11}$.

### Step 2 : $j = 2$

$$\mathbf{a}_2 = r_{12}\mathbf{q}_1 + r_{22}\mathbf{q}_2 \tag{10}$$

#### Question

How do we find $r_{12}$, $r_{22}$ and $\mathbf{q}_2$?

From the above, we can write

$$\mathbf{q}_1^*\mathbf{a}_2 = r_{12} \tag{11}$$

where $r_{12}$ is the projection of $\mathbf{a}_2$ onto $\mathbf{q}_1$. We then subtract out this component of $\mathbf{a}_2$ in the direction of $\mathbf{q}_1$ to define an intermediate vector

$$\mathbf{v} = \mathbf{a}_2 - r_{12}\mathbf{q}_1 \equiv r_{22}\mathbf{q}_2 \tag{12}$$

or

$$\mathbf{q}_2 = \frac{\mathbf{v}}{r_{22}} \tag{13}$$

where $r_{22} = \|\mathbf{a}_2 - r_{12}\mathbf{q}_1\|$.

### Step 2 : $j = k$

Define

$$\mathbf{v} = \mathbf{a}_k - \sum_{j=1}^{k-1} r_{jk}\mathbf{q}_j \tag{14}$$

Then

$$\mathbf{q}_k = \frac{\mathbf{v}}{r_{kk}} \tag{15}$$

where $r_{jk} = \mathbf{q}_j^*\mathbf{a}_k$ for $j < k$, and

$$r_{kk} = \|\mathbf{v}\| = \left\|\mathbf{a}_k - \sum_{j=1}^{k-1} r_{jk}\mathbf{q}_j\right\| \tag{16}$$

In [2]:
```python
def display_mat(msg,A):
    print(msg)
    display(A)
    print("")
```

## Gram-Schmidt algorithm

Here is an outline of the classical Gram-Schmidt algorithm :

- For $j = 1, 2, \ldots n$

  -- Set $\mathbf{v} = \mathbf{a}_j$, the $j^{th}$ column of $A$.

  -- Orthogonalize $\mathbf{v}$ against previous $\mathbf{q}_i, i = 0, 1, \ldots, j-1$.

  -- Set $\mathbf{q}_j$ equal to normalized vector $\mathbf{v}$.

The code for the Gram-Schmidt algorithm is below.

In [3]:
```python
# Classical Gram-Schmidt algorithm for orthogonalizing a set of column vectors.

from numpy.linalg import norm

def gram_schmidt_classic(A):
    m,n = A.shape
    assert n <= m, 'We must have n <= m'
    R = np.zeros((n,n))
    Q = np.zeros((m,n))
    tol = 1e-12
    for j in range(n):
        # Loop over columns of A;
        aj = A[:,j:j+1]
        v = aj
        # Orthogonalize against previous qi vectors, i = 0,1,2,3,...,j-1
        for i in range(j):
            qi = Q[:,i:i+1]
            R[i,j] = qi.T@aj    # m ops
            v = v - R[i,j]*qi
        R[j,j] = norm(v,2)
        assert R[j,j] > tol, "Columns are not linearly independent"
        Q[:,j:j+1] = v/R[j,j]

    return Q,R
```

Before continuing, we set up a few matrices that we can use for examples. We'll put them in a function so that we can easily access different choices.

In [35]:
```python
def matrix_example(id):

    # 3 x 1 example
    A1 = np.array(np.mat('1; 3; 5'),dtype=float)

    # A 3 x 2 example
    A2 = np.array(np.mat('1,2; 3,4; 5,-1'),dtype=float)

    # A 3x3 example.
    A3 = np.array(np.mat('1,2,-1; 3,4,4; 5,6,5'),dtype=float)

    A_mat = (A1,A2,A3)

    assert id > 0 and id < 4, "Assert id must be 1,2,3."

    return A_mat[id-1]
```

In [36]:
```python
A = matrix_example(3)

display_mat('A = ',A)

Q,R = gram_schmidt_classic(A)

display_mat('Q = ', Q)
display_mat('R = ',R)
display_mat('QR = ',Q@R)
display_mat('Q^*Q',Q.T@Q)
```

```
A =
array([[ 1.,  2., -1.],
       [ 3.,  4.,  4.],
       [ 5.,  6.,  5.]])
Q =
array([[ 0.16903085,  0.89708523, -0.40824829],
       [ 0.50709255,  0.27602622,  0.81649658],
       [ 0.84515425, -0.34503278, -0.40824829]])
R =
array([[ 5.91607978,  7.43735744,  6.08511063],
       [ 0.        ,  0.82807867, -1.51814423],
       [ 0.        ,  0.        ,  1.63299316]])
QR =
array([[ 1.,  2., -1.],
       [ 3.,  4.,  4.],
       [ 5.,  6.,  5.]])
Q^*Q
array([[ 1.00000000e+00,  2.77555756e-16,  0.00000000e+00],
       [ 2.77555756e-16,  1.00000000e+00, -9.15933995e-16],
       [ 0.00000000e+00, -9.15933995e-16,  1.00000000e+00]])
```

## Using QR to solve $A\mathbf{x} = \mathbf{b}$.

Suppose we have a $QR$ factorization of a non-singular matrix $A$. How can we use this factorization to solve $A\mathbf{x} = \mathbf{b}$?

#### Answer

1. $A = QR$
2. $QR\mathbf{x} = \mathbf{b}$
3. $R\mathbf{x} = Q^*\mathbf{b}$
4. Use back substitution to solve for $\mathbf{x}$.

## Modified Gram-Schmidt algorithm (more stable)

While the Gram-Schmidt algorithm used above is intuitive, it can be very sensitive to small changes in the input vectors, e.g. columns of $A$. For this reason, we used a "modified Gram-Schmidt" algorithm.

Recall the the classical Gram-Schmidt (CGS) orthogonalizes a set of vectors $[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n]$ by successivly *subtracting out* projections on the previously found orthonormal set $[\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]$.

$$\begin{aligned}
\mathbf{q}_1 &= \frac{\mathbf{a}_1}{r_{11}} && (17)\\
\mathbf{q}_2 &= \frac{\mathbf{a}_2 - (\mathbf{q}_1^*\mathbf{a}_2)\mathbf{q}_1}{r_{22}} && (18)\\
\mathbf{q}_3 &= \frac{\mathbf{a}_3 - (\mathbf{q}_1^*\mathbf{a}_3)\mathbf{q}_1 - (\mathbf{q}_2^*\mathbf{a}_3)\mathbf{q}_2}{r_{33}} && (19)\\
&\vdots && (20)
\end{aligned}$$

where the $r_{ii}$ are the normalizing factors needed to ensure that $\|\mathbf{q}_i\| = 1$.

Geometrically, we can interpret the first few steps for vectors in $\mathbb{R}^3$ as follows.

For $\mathbf{a}_1$, we have

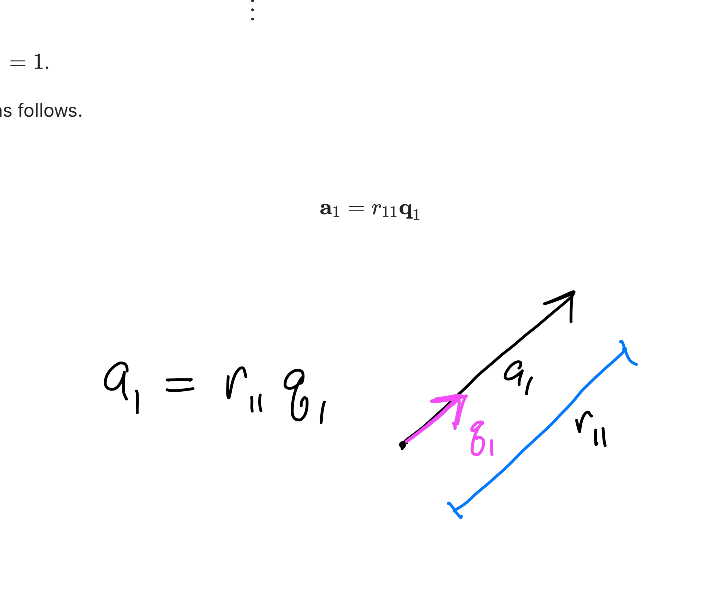
$$\mathbf{a}_1 = r_{11}\mathbf{q}_1 \tag{21}$$




For $\mathbf{a}_2$, we have

$$\mathbf{a}_2 = (\mathbf{q}_1^*\mathbf{a}_2)\mathbf{q}_1 + r_{22}\mathbf{q}_2 \tag{22}$$

from which it is clear that $\mathbf{a}_2 \in \langle \mathbf{q}_1, \mathbf{q}_2 \rangle$, or that $\mathbf{a}_2$, $\mathbf{q}_1$ and $\mathbf{q}_2$ are coplanar.




A key observation in the above is that $\mathbf{q}_2$ lies in a plane *orthogonal* to $\mathbf{a}_1$. From the expression

$$\mathbf{a}_2 = (\mathbf{q}_1^*\mathbf{a}_2)\mathbf{q}_1 + r_{22}\mathbf{q}_2 \tag{23}$$

we see that $\mathbf{q}_2$ contains all "components" of $\mathbf{a}_2$ not available in $\mathbf{q}_1$. Or, $\mathbf{q}_2$ is a *projection* onto the space *orthogonal* to $\mathbf{q}_1$.

This is the idea behind the "Modified Gram-Schmidt" procedure. Rather than subtract out the components we don't want, we project only onto the space of components that we *do* want.

Here is an outline of the Gram-Schmidt algorithm :

- Set $\mathbf{v}_i = \mathbf{a}_i, i = 1, 2, \ldots, n$
- For $i = 1, 2, \ldots n$

  -- Set $\mathbf{q}_i$ to the unit vector in direction $\mathbf{v}_i$.

  -- Orthogonalize remaining vectors $v_k, k = i+1, \ldots n$ against $\mathbf{q}_i$.

The code for the Modified Gram-Schmidt algorithm is below.

In [37]:
```python
def gram_schmidt(A):
    m,n = A.shape
    assert n <= m, 'We must have n \le m'
    R = np.zeros((n,n))
    Q = np.zeros((m,n))
    V = A.copy()
    # Loop over all columns of V.
    for i in range(n):
        # Assign qi to unit vector in direction vi
        v = V[:,i:i+1]
        R[i,i] = np.linalg.norm(v,2)
        assert R[i,i] > 0, "Columns are not linearly independent."
        qi = v/R[i,i]
        Q[:,i:i+1] = qi
        # Orthogalize remaining vectors vk, k = i+1,...,n against qi
        for j in range(i+1,n):
            vj = V[:,j:j+1]
            R[i,j] = vj.T@qi
            vj = vj - R[i,j]*qi
            V[:,j:j+1] = vj

            vjp1 = V[:,j:j+1]

    return Q,R
```

In [39]:
```python
A = matrix_example(3)

display_mat('A = ', A)

Q,R = gram_schmidt(A)

display_mat('Q = ', Q)
display_mat('R = ',R)
display_mat("Q^T*Q = ", Q.T@Q)
display_mat("QR = ",Q@R)
```

```
A =
array([[ 1.,  2., -1.],
       [ 3.,  4.,  4.],
       [ 5.,  6.,  5.]])
Q =
array([[ 0.16903085,  0.89708523, -0.40824829],
       [ 0.50709255,  0.27602622,  0.81649658],
       [ 0.84515425, -0.34503278, -0.40824829]])
R =
array([[ 5.91607978,  7.43735744,  6.08511063],
       [ 0.        ,  0.82807867, -1.51814423],
       [ 0.        ,  0.        ,  1.63299316]])
Q^T*Q =
array([[ 1.00000000e+00,  2.77555756e-16, -1.11022302e-16],
       [ 2.77555756e-16,  1.00000000e+00, -2.22044605e-16],
       [-1.11022302e-16, -2.22044605e-16,  1.00000000e+00]])
QR =
array([[ 1.,  2., -1.],
       [ 3.,  4.,  4.],
       [ 5.,  6.,  5.]])
```

Modified Gram-Schmidt follows typical self-help advice : Don't think about what you *don't* want in your present life; rather think about what you *do* want in your future life.

In [ ]: