Homework 1 - Question 1 - Luke Arend

a)

The component of $\vec{v}$ lying along $\hat{u}$ has a length equal to their dot product $\hat{u} \cdot \vec{v}$ and points in direction $\hat{u}$.

In [1]:
```
def projection(u_hat, v):
    return u_hat * np.dot(u_hat, v)
```

b)

Since $\vec{v}$ can be decomposed into a component along $\hat{u}$ and a component orthogonal to $\hat{u}$, subtracting the component of $\vec{v}$ along $\hat{u}$ from $\vec{v}$ gives the orthogonal component.

In [2]:
```
def ortho(u_hat, v):
    return v - projection(u_hat, v)
```

c)

The distance from $\vec{v}$ to the component that lies along direction $\hat{u}$ is the length of the component of $\vec{v}$ orthogonal to $\hat{u}$.

In [3]:
```
def norm(v):
    return np.sqrt(np.dot(v, v))

def distance(u_hat, v):
    return norm(ortho(u_hat, v))
```

Verify in 2 dimensions

Now we verify the code by testing it on random vectors. First we define a function which samples a random vector with dimensionality N.

In [4]:
```
import numpy as np
```

In [5]:
```
def randvec(N):
    return np.random.randn(N)
```

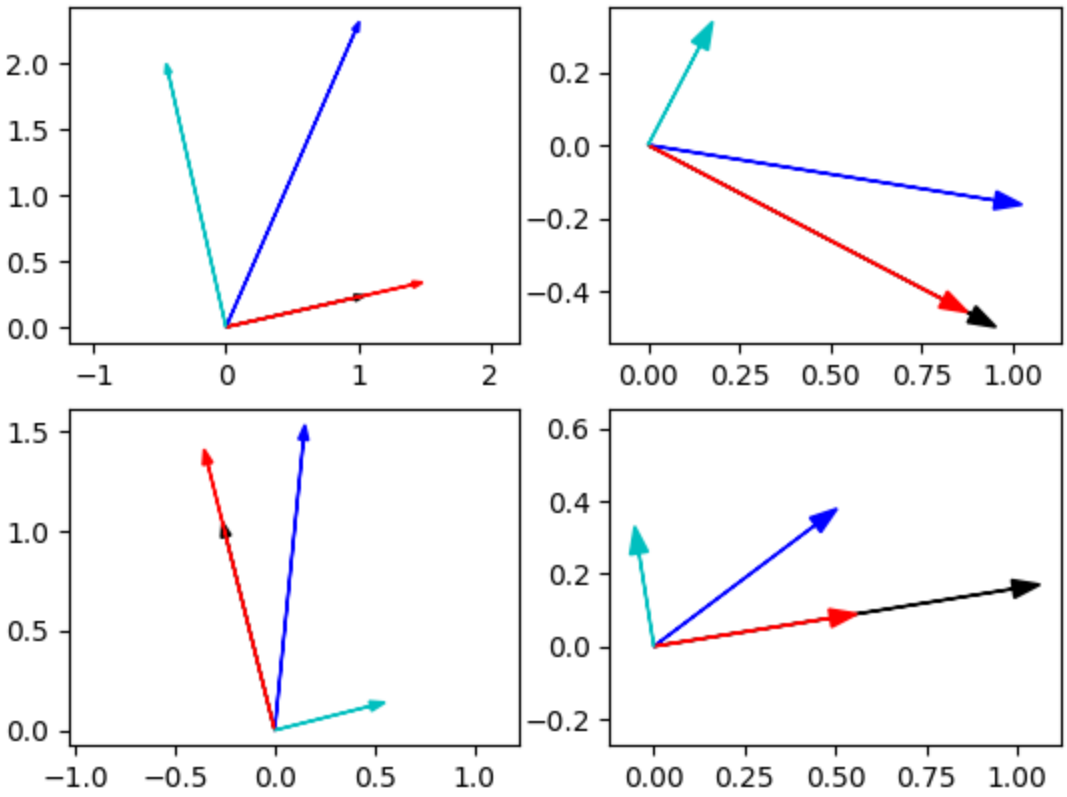
We also define a function which scales a vector to unit length.

In [6]:
```
def unit(v):
    return v / norm(v)
```

We verify visually with 2-dimensional vectors by plotting $\hat{u}$, $\vec{v}$ and the two components described in (a) and (b) for several random draws.

In [7]:
```
from matplotlib import pyplot as plt
```

In [8]:
```
def visual_test_2d(ax):
    # Draw random vectors
    u_hat = unit(randvec(2))
    v = randvec(2)

    # Compute projection
    v_proj = projection(u_hat, v)
    v_ortho = ortho(u_hat, v)

    # Plot vectors and projections
    origin = [0, 0]
    ax.arrow(*origin, *u_hat, head_width=0.05, color='k')   # black: u_hat
    ax.arrow(*origin, *v, head_width=0.05, color='b')       # blue: v
    ax.arrow(*origin, *v_proj, head_width=0.05, color='r')  # red: v_proj
    ax.arrow(*origin, *v_ortho, head_width=0.05, color='c') # cyan: v_ortho

    ax.axis('equal')

np.random.seed(0)

fig, axes = plt.subplots(2, 2)
for ax in axes.flat:
    visual_test_2d(ax)

plt.show()
```



Verify in N dimensions

If a vector $\vec{a}$ points in the same direction as $\hat{u}$, then its projection along $\hat{u}$ equals $\vec{a}$ itself.

In [9]:
```
def parallel_ok(u_hat, v):
    a = projection(u_hat, v)
    return np.allclose(a, projection(u_hat, a))
```

If a vector $\vec{b}$ is orthogonal to $\hat{u}$, then its projection along $\hat{u}$ has length 0.

In [10]:
```
def orthogonal_ok(u_hat, v):
    b = ortho(u_hat, v)
    return np.allclose(0, norm(projection(u_hat, b)))
```

If $\vec{a}$ and $\vec{b}$ are the components of $\vec{v}$ parallel and orthogonal to $\hat{u}$, then $\vec{a}$ and $\vec{b}$ sum to $\vec{v}$.

In [11]:
```
def sum_ok(u_hat, v):
    a = projection(u_hat, v)
    b = ortho(u_hat, v)
    return np.allclose(v, a + b)
```

If $\vec{a}$ and $\vec{b}$ are the components of $\vec{v}$ parallel and orthogonal to $\hat{u}$, then the sum of their squared lengths is equal to the squared length of $\vec{v}$.

In [12]:
```
def lengths_ok(u_hat, v):
    a = projection(u_hat, v)
```

```
    b = ortho(u_hat, v)
    return np.allclose(norm(v)**2, norm(a)**2 + norm(b)**2)
```

In [13]:
```
def all_ok(u_hat, v):
    cond1 = parallel_ok(u_hat, v)
    cond2 = orthogonal_ok(u_hat, v)
    cond3 = sum_ok(u_hat, v)
    cond4 = lengths_ok(u_hat, v)
    return np.all([cond1, cond2, cond3, cond4])
```

Verify for some random vectors with higher dimensionality.

In [14]:
```
for N in [0, 1, 2, 3, 4, 10, 100, 1000]:
    u_hat = unit(randvec(N))
    v = randvec(N)
    ok = all_ok(u_hat, v)
    print(f"N = {N}: {'ok' if ok else 'failed'}")
```

```
N = 0: ok
N = 1: ok
N = 2: ok
N = 3: ok
N = 4: ok
N = 10: ok
N = 100: ok
N = 1000: ok
```

In [ ]:
```
```

Homework 1 - Question 2 - Luke Arend

a)

0 → [1, -5]

This **cannot be a linear** system. If it is linear, there exists a 2x1 matrix M (or 2-vector $\vec{v}$) which when scaled by 0 gives the vector [1, -5]. Any vector scaled by 0 gives the zero vector so no such 2-vector exists.

b)

[3, 2] → 15

[-2, 2] → 6

This might be a linear system. If so, a 1x2 matrix M (or 2-vector $\vec{v}$) exists where $\vec{v} \cdot [3, 2] = 15$ and $\vec{v} \cdot [-2, 2] = 6$.

Let $\vec{v} = [x, y]$. Then $[x, y] \cdot [3, 2] = 15$ and $[x, y] \cdot [-2, 2] = 6$, which corresponds to the system of equations

$$3x + 2y = 15$$

$$-2x + 2y = 6.$$

Solving for y in the second equation, we get $y = (6 + 2x)/2 = x + 3$.

Substituting $x + 3$ for y in the first equation gives

$$3x + 2(x + 3) = 5x + 6 = 15.$$

Solving for x, we get $x = (15 - 6)/5 = 9/5$.

Substituting $9/5$ for x in the first equation gives

$$3(9/5) + 2y = 2y + 27/5 = 15.$$

Solving for y, we get
$y = (15 - 27/5)/2 = 15/2 - 27/10 = 75/10 - 27/10 = 48/10 = 24/5$.

So the system **could be linear** with matrix $M = [9/5, 24/5]^T$.

c)

[2, 6] → [3, 7]

[1, -2] → [-1, 3]

[5, 0] → [1, 4]

If this system is linear, then it has a 2x2 matrix $M = [[a, b], [c, d]]$ and these observations correspond to the system of equations

$2a + 6b = 3, 2c + 6d = 7$

$a - 2b = -1, c - 2d = 3$

$5a = 1, 5c = 4.$

The third observation says $a = 1/5.$

Substituting $1/5$ for a in the second equation gives

$1/5 - 2b = -1$, so

$b = (-4/5)/-2 = 2/5.$

Are these values for $a = 1/5$ and $b = 2/5$, derived from the second and third observation, consistent with the first observation $2a + 6b = 3$?

$2(1/5) + 6(2/5) = 14/5 \neq 3.$

So the three observations are not consistent assuming any linear system. Therefore this system **cannot be linear**.

d)

[3, 1.5] → [-3, -6]

[-8, -4] → [8, 16]

This might be a linear system. If so, it has matrix $M = [[a, b], [c, d]]$ and these observations correspond to the system of equations

$3a + 1.5b = -3, 3c + 1.5d = -6$

$-8a - 4b = 8, -8c - 4d = 16.$

This is two systems of equations, one in a, b and one in $c, d.$

Solving for b in terms of a and d in terms of c gives

$b = -2a - 2$ and

$d = -2c - 4$.

There are many choices of a, b and c, d which satisfy the equations above. Let $a = 1$ and $c = 1$. Then $b = -4$ and $d = -6$.

So the system **could be linear**, where $M = [[1, -4], [1, -6]]$ is one matrix explaining the observations. But there are many such matrices (we can find one starting from any a and c by choosing b and d appropriately).

In [ ]:

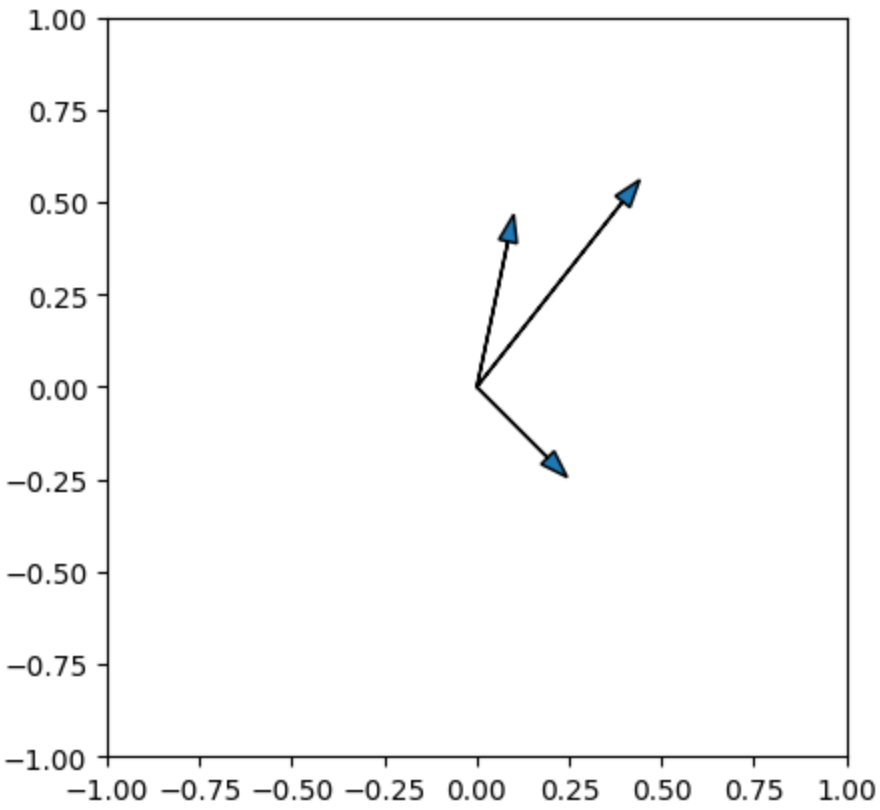
Homework 1 - Question 3 - Luke Arend

a)

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
def plotVec2(M):
    assert M.shape[0] == 2 # matrix must have two rows
    for v in M.T:
        plt.arrow(0, 0, *v, head_width=0.05, length_includes_head=True)
    plt.axis('square')
    plt.xlim([-1, 1])
    plt.ylim([-1, 1])
```

In [3]:
```
np.random.seed(0)
M = 0.25 * np.random.randn(2, 3)
plotVec2(M)
plt.show()
```



b)

`vecLenAngle` takes two vectors as arguments and returns the magnitude of each vector, as well as the angle (in radians) between them. It returns 0 for the angle if either vector has zero length.

In [4]:
```
def vecLenAngle(v1, v2):
    L1 = np.sqrt(np.sum(v1 ** 2))
    L2 = np.sqrt(np.sum(v2 ** 2))
    if L1 * L2 == 0:
        angle = 0
    else:
        angle = np.arccos(np.sum(v1 * v2) / (L1 * L2))
    return L1, L2, angle
```

In [5]:
```
v1, v2 = np.random.randn(2, 2)
print(f"v1: {v1}\nv2: {v2}")
print(f"vecLenAngle(v1, v2): {vecLenAngle(v1, v2)}")
```

```
v1: [ 0.95008842 -0.15135721]
v2: [-0.10321885  0.4105985 ]
vecLenAngle(v1, v2): (0.9620691272564312, 0.42337366611495336, 1.9750603468048809)
```

c)

Generate a random matrix M.

In [6]:
```
M = np.random.randn(2, 2)
M
```

Out[6]:
```
array([[0.14404357, 1.45427351],
       [0.76103773, 0.12167502]])
```

Decompose it using the singular value decomposition.

In [7]:
```
U, S, Vt = np.linalg.svd(M)
np.allclose(np.dot(U * S, Vt), M)
```

Out[7]:
```
True
```

Generate standard basis vectors. Compute their lengths and the angle between them.

In [8]:
```
I = np.identity(2)
```

In [9]:
```
vecLenAngle(I[:, 0], I[:, 1])
```

Out[9]:
```
(1.0, 1.0, 1.5707963267948966)
```

Their lengths are 1 and the angle between them is $\pi/2$ radians or 90 degrees.

Rotate the standard basis by V^T, and compute the lengths/angle between those.

In [10]:
```
X1 = np.dot(Vt, I)
vecLenAngle(X1[:, 0], X1[:, 1])
```

Out[10]:
```
(1.0000000000000002, 1.0000000000000002, 1.5707963267948966)
```

After applying V^T, their lengths are still 1 and the angle between them is still $\pi/2$ radians.

Now scale the vectors by factors of S_{diag} along the directions of the standard basis and compute the lengths/angle.

In [11]:
```
X2 = np.dot(np.diag(S), X1)
vecLenAngle(X2[:, 0], X2[:, 1])
```

Out[11]:
```
(0.7745495268152256, 1.4593547350431704, 1.3002637488305455)
```

Following S, the vector lengths have changed. Also the angle between them is no longer $\pi/2$ radians. Call their new lengths L_1 and L_2 and the new angle θ.

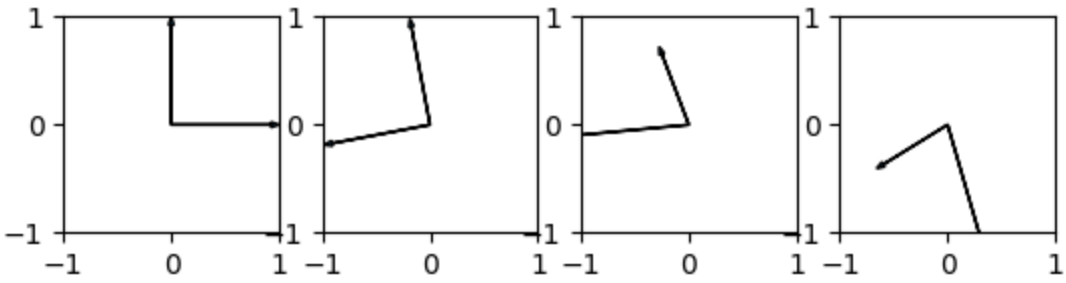
Finally rotate the vectors by U, and compute the lengths/angle again.

In [12]:
```
X3 = np.dot(Vt, X2)
vecLenAngle(X3[:, 0], X3[:, 1])
```

Out[12]:
```
(0.7745495268152257, 1.4593547350431706, 1.3002637488305453)
```

Following U, the lengths L_1 and L_2 and the angle θ are preserved.

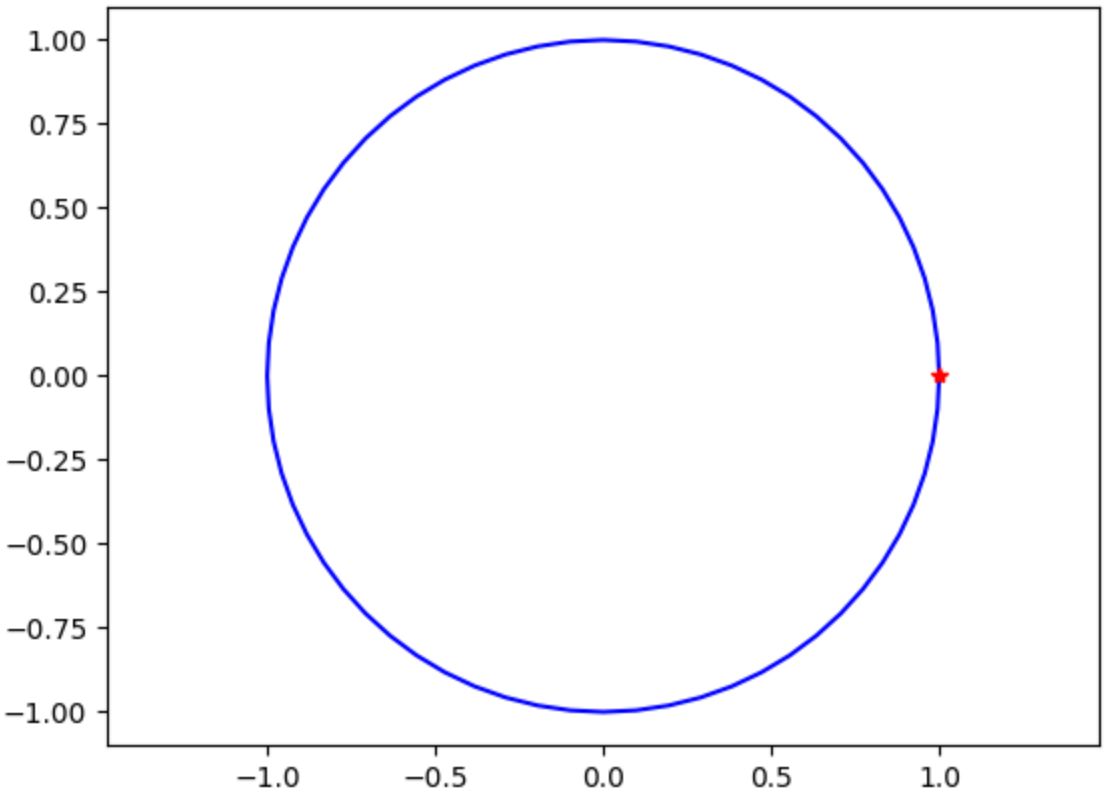
In [13]:
```
fig, axes = plt.subplots(1, 4)
for i, X in enumerate([I, X1, X2, X3]):
    plt.sca(axes[i])
    plotVec2(X)
plt.show()
```



d)

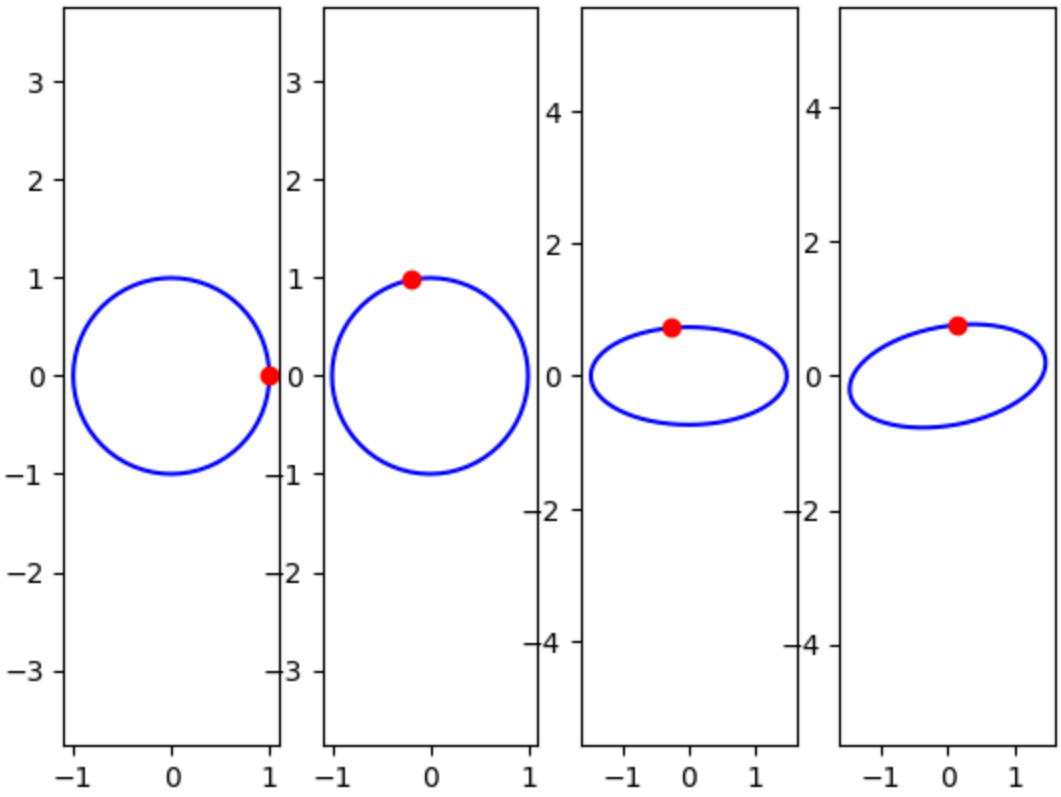
Generate a 2x65 data matrix P of unit vectors tracing out the unit circle in 64 steps.

In [14]:
```
theta = 2 * np.pi * np.arange(65) / 64
P = np.array([np.cos(theta), np.sin(theta)])
plt.plot(*P, 'b')
plt.plot(*P[:, 0], 'r*')
plt.axis('equal')
plt.show()
```



The action of the matrix M in the previous problem was decomposed into three separate actions: rotating by V_T, scaling by the singular values S_{diag} and rotating by U. Rotation by V_T just rotates the unit circle. The scaling operation by S stretches or compresses the circle along the axes of the standard basis, giving an axis-aligned ellipse. The rotation by U rotates the ellipse in the plane. This is shown below.

In [15]:
```
X1 = np.dot(Vt, P)
X2 = np.dot(np.diag(S), X1)
X3 = np.dot(U, X2)

fig, axes = plt.subplots(1, 4)
for i, X in enumerate([P, X1, X2, X3]):
    plt.sca(axes[i])
    plt.plot(*X, 'b')
    plt.plot(*X[:, 0], 'ro')
    plt.axis('equal')
plt.show()
```



In [ ]:

Homework 1 - Question 4 - Luke Arend

a)

The system is linear. Its scalar response is a weighted sum over the entries of the input vector. Its action can be represented as a dot product with the weight vector $\vec{w} = [6, 8.2, 1, 3, 1]$ or $r = M\vec{v}$ where M is the 5x1 matrix $\vec{w}^T$.

b)

For input vectors of a given length, the vector with entries proportional to $\vec{w}$ will produce the largest response. Geometrically, such a vector is perfectly aligned with the weight vector $\vec{w}$. So the unit-length stimulus vector which elicits the largest response is just $\frac{\vec{w}}{\|\vec{w}\|}$:

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

In [2]:
```
w = np.array([6, 8.2, 1, 3, 1])
v = w / np.sqrt(np.dot(w, w))
v
```

Out[2]:
```
array([0.56136089, 0.76719322, 0.09356015, 0.28068045, 0.09356015])
```

c)

A physically realizable unit vector $\vec{v}$ with the smallest possible response will contain nonnegative entries and minimize the dot product $\vec{w} \cdot \vec{v}$. The dot product is minimized when $\vec{v}$ has only one nonzero entry and that entry aligns with the smallest entry of $\vec{w}$. There are two such unit vectors: $[0, 0, 1, 0, 0]$ and $[0, 0, 0, 0, 1]$.

In [ ]:
```
```

Homework 1 - Question 5 - Luke Arend

First we implement Gram-Schmidt.

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

In [2]:
```
def add_column(X):
    if X.shape[0] == X.shape[1]:
        return X

    v = np.random.randn(X.shape[0])
    for u in X.T:
        v -= np.dot(u, v) * u
    v /= np.sqrt(np.dot(v, v))

    X = np.append(X.T, [v], axis=0).T
    return add_column(X)

def gramschmidt(n):
    X = np.zeros((n, 0))
    return add_column(X)
```

Next we generate a 3x3 orthonormal matrix U using Gram-Schmidt.

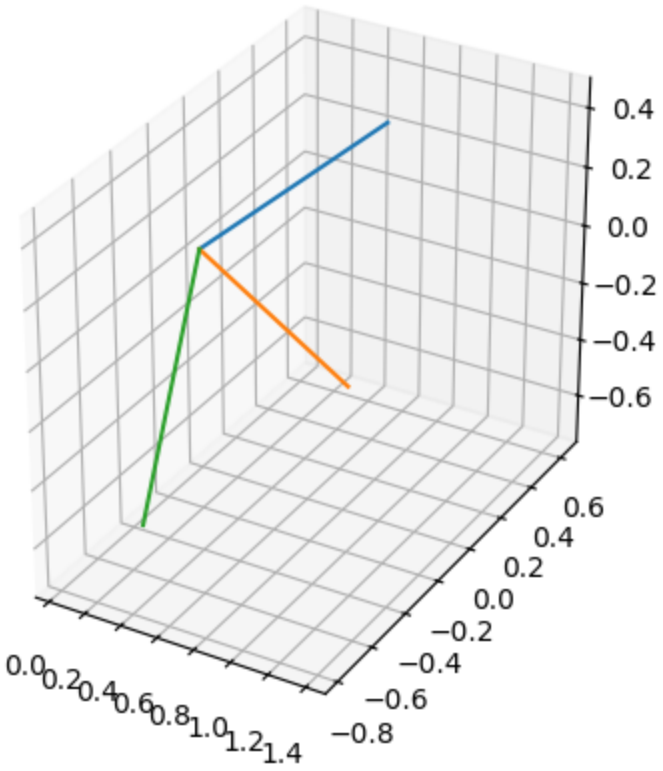
In [3]:
```
np.random.seed(0)
U = gramschmidt(3)
U
```

Out[3]:
```
array([[ 0.85771824,  0.27400307,  0.43501924],
       [ 0.1945646 ,  0.61021592, -0.7679721 ],
       [ 0.47588238, -0.74334302, -0.47008203]])
```

We plot the columns of U to see whether they are orthogonal and unit length.

In [4]:
```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [5]:
```
#%matplotlib notebook
ax = plt.subplot(projection='3d')
for i in range(3):
    plt.plot(xs=[0, U[0, i]], ys=[0, U[1, i]], zs=[0, U[2, i]])
plt.axis('square')
plt.show()
```



U appears to be orthonormal based on the figure above.

Let's show that U is orthonormal for dimensionality N = 1000.

In [6]:
```
%%time
U = gramschmidt(1000)
```

```
CPU times: user 1.84 s, sys: 616 ms, total: 2.45 s
Wall time: 2.21 s
```

In [7]:
```
np.allclose(np.dot(U, U.T), np.identity(1000))
```

Out[7]: True

U is orthonormal since the inverse of U is its transpose.

Also note that each row and column of U has magnitude 1.

In [8]:
```
np.allclose(1, np.sqrt(np.sum(U * U, axis=0)))
```

Out[8]: True

In [9]:
```
np.allclose(1, np.sqrt(np.sum(U * U, axis=1)))
```

Out[9]: True

In [ ]:
```

```

Homework 1 - Question 6 - Luke Arend

Consider a linear system which transforms $\vec{v}$ to $\vec{y}$ such that $\vec{y} = M\vec{v}$.

a)

The null space of the matrix M is the set of vectors which, when left-multiplied by M, result in the zero vector. It is the subspace of the input space which is completely discarded by the transform M.

The range space of the matrix M is the set of vectors reachable by applying M to any input vector. It is the subspace of the output space actually occupied by the transform M applied to all possible inputs.

b)

Consider a creature which produces a neural response vector to a vector input of pressure measurements. If the system has a non-zero null space, then some pressure vectors are mapped to the zero vector. This means that there exist stimuli which, when presented to the animal, produce no neural response. Call these "null" stimuli.

Since the system is linear, any null stimulus could be added to any ordinary stimulus to produce a metamer: a second, different stimulus which, when presented to the animal, produces the same neural response as the original.

c)

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

In [2]:
```
data = scipy.io.loadmat('Hw1Q6_MtxExamples.mat')
```

M1

In [3]:
```
M1 = data['mtx1']
M1
```

Out[3]:
```
array([[1.76371689, 0.18079736, 1.96484125],
       [0.68217653, 0.18724003, 0.64619217]])
```

In [4]:
```
U, S, Vt = np.linalg.svd(M1)
S
```

Out[4]:
```
array([2.81131413, 0.13596668])
```

1.

Since there are 3 input dimensions and only 2 singular values, the third input dimension is discarded by the transform M_1. It is spanned by the third column of V.

Sample a random vector along the third column of V. Applying M_1 gives the zero vector.

In [5]:
```
v = Vt.T[:, 2] * np.random.randn()
np.dot(M1, v)
```

Out[5]:
```
array([-2.06095361e-16, -4.60333619e-18])
```

2.

The range space of M_1 is spanned by the columns of U. Sample a random vector $\vec{y}$ from the range space.

In [6]:
```
y = np.dot(U.T, np.random.randn(2))
y
```

Out[6]:
```
array([-0.70201609, -0.53103311])
```

If $\vec{y} = M\vec{x}$, then $\vec{x} = M^{-1}\vec{y}$. And since $M = USV^T$, $M^{-1} = VS^{-1}U^T$. So $\vec{x} = VS^{-1}U^T\vec{y}$.

In [7]:
```
S_inv = np.diag(1 / S)
S_inv = np.concatenate([S_inv, [[0, 0]]], axis=0)
M_inv = np.dot(np.dot(Vt.T, S_inv), U.T)
x = np.dot(M_inv, y)
x
```

Out[7]:
```
array([-0.86015132, -1.66152058,  0.56770279])
```

Verify that $\vec{y} = M\vec{x}$ for `x` and `y`.

In [8]:
```
np.allclose(y, np.dot(M1, x))
```

Out[8]:
```
True
```

M2

In [9]:
```
M2 = data['mtx2']
M2
```

Out[9]:
```
array([[-0.19744334,  0.01215894, -0.58881013],
       [ 0.43287258, -0.02665713,  1.29090077],
       [ 1.07207807, -0.06602064,  3.19712192]])
```

In [10]:
```
U, S, Vt = np.linalg.svd(M2)
S
```

Out[10]:
```
array([3.68993791e+00, 3.25816135e-16, 1.25679497e-18])
```

1.

M_2 has just one singular value. So the null space is spanned by the last two columns of V.

If we sample a vector from the null space and apply M_2, we get the zero vector.

In [11]:
```
v = np.dot(Vt.T[:, -2:], np.random.randn(2))
np.dot(M2, v)
```

Out[11]:
```
array([ 1.41060538e-16, -2.97591557e-16, -6.86318549e-16])
```

2.

Since M_2 has one singular value, the first column of U spans the range space. Sample a random vector $\vec{y}$ from the range space.

In [12]:
```
y = U[:, 0] * np.random.randn()
y
```

Out[12]:
```
array([-0.1673523 ,  0.36690132,  0.90868972])
```

Find an $\vec{x}$ such that $\vec{y} = M_2\vec{x}$.

In [13]:
```
S_inv = np.diag([1 / S[0], 0, 0])
M_inv = np.dot(np.dot(Vt.T, S_inv), U.T)
x = np.dot(M_inv, y)
x
```

Out[13]:
```
array([ 0.0856405 , -0.00527391,  0.25539477])
```

In [14]:
```
np.allclose(y, np.dot(M2, x))
```

Out[14]:
```
True
```

M3

In [15]:
```
M3 = data['mtx3']
M3
```

Out[15]:
```
array([[-0.52084555,  3.27506098],
       [ 0.75993107, -0.69773866],
       [-0.60979668, -4.31920227]])
```

In [16]:
```
U, S, Vt = np.linalg.svd(M3)
S
```

Out[16]:
```
array([5.46570167, 1.1023178 ])
```

1.

Since there are 2 input dimensions and 2 singular values, the transform does not have a null space.

2.

Since there are 3 output dimensions and only 2 singular values, the range space is spanned by the first two columns of U.

We can sample a random vector $\vec{y}$ from the range space and solve for $\vec{x}$ such that $\vec{y} = M_3\vec{x}$.

In [17]:
```
y = np.dot(U[:, :2], np.random.randn(2))
y
```

Out[17]:
```
array([-0.53561024, -0.45220568,  1.83792154])
```

In [18]:
```
S_inv = np.diag(1 / S)
S_inv = np.concatenate([S_inv, [[0], [0]]], axis=1)
M_inv = np.dot(np.dot(Vt.T, S_inv), U.T)
x = np.dot(M_inv, y)
x
```

Out[18]:
```
array([-0.87264112, -0.30232154])
```

In [19]:
```
np.allclose(y, np.dot(M3, x))
```

Out[19]:
```
True
```

M4

In [20]:
```
M4 = data['mtx4']
M4
```

Out[20]:
```
array([[-0.095065  ,  0.7939639 ],
       [ 0.85959259,  0.27567482]])
```

In [21]:
```
U, S, Vt = np.linalg.svd(M4)
S
```

Out[21]:
```
array([0.94338616, 0.75122208])
```

1.

Since there are 2 input dimensions and 2 singular values, the transform does not have a null space.

2.

Since there are 2 output dimensions and 2 singular values, the range space is the column space of U.

Sample a random vector $\vec{y}$ from the range space and solve for $\vec{x}$ such that $\vec{y} = M_3\vec{x}$.

In [22]:
```
y = np.dot(U, np.random.randn(2))
y
```

Out[22]:
```
array([ 0.6986013, -1.283732 ])
```

In [23]:
```
S_inv = np.diag(1 / S)
M_inv = np.dot(np.dot(Vt.T, S_inv), U.T)
x = np.dot(M_inv, y)
x
```

Out[23]:
```
array([-1.7099428 ,  0.67515109])
```

In [24]:
```
np.allclose(y, np.dot(M4, x))
```

Out[24]:
```
True
```

M5

In [25]:
```
M5 = data['mtx5']
M5
```

Out[25]:
```
array([[-3.44738961, -5.50282864],
       [-3.19713133, -5.10335872]])
```

In [26]:
```
U, S, Vt = np.linalg.svd(M5)
S
```

Out[26]:
```
array([8.85615815e+00, 2.40750870e-16])
```

1.

The input space has 2 dimensions and there is just one singular value, so the second column of V spans the null space.

If we sample a random vector from the null space and apply M_5, we get the zero vector.

In [27]:
```
v = Vt.T[:, 1] * np.random.randn()
np.dot(M5, v)
```

Out[27]:
```
array([1.49850853e-15, 1.05727513e-15])
```

2.

Since the output space has 2 dimensions and there is one singular value, the first column of U spans the range space.

Sample a vector $\vec{y}$ from the range space and solve for $\vec{x}$ such that $\vec{y} = M_3\vec{x}$.

In [28]:
```
y = np.dot(U[:, 0], np.random.randn())
y
```

Out[28]:
```
array([0.80923837, 0.75049288])
```

In [29]:
```
S_inv = np.diag([1 / S[0], 0])
M_inv = np.dot(np.dot(Vt.T, S_inv), U.T)
x = np.dot(M_inv, y)
x
```

Out[29]:
```
array([-0.06616196, -0.10560974])
```

In [30]:
```
np.allclose(y, np.dot(M5, x))
```

Out[30]:
```
True
```

In [ ]:
```

```