Homework 2 - Question 1 - Luke Arend

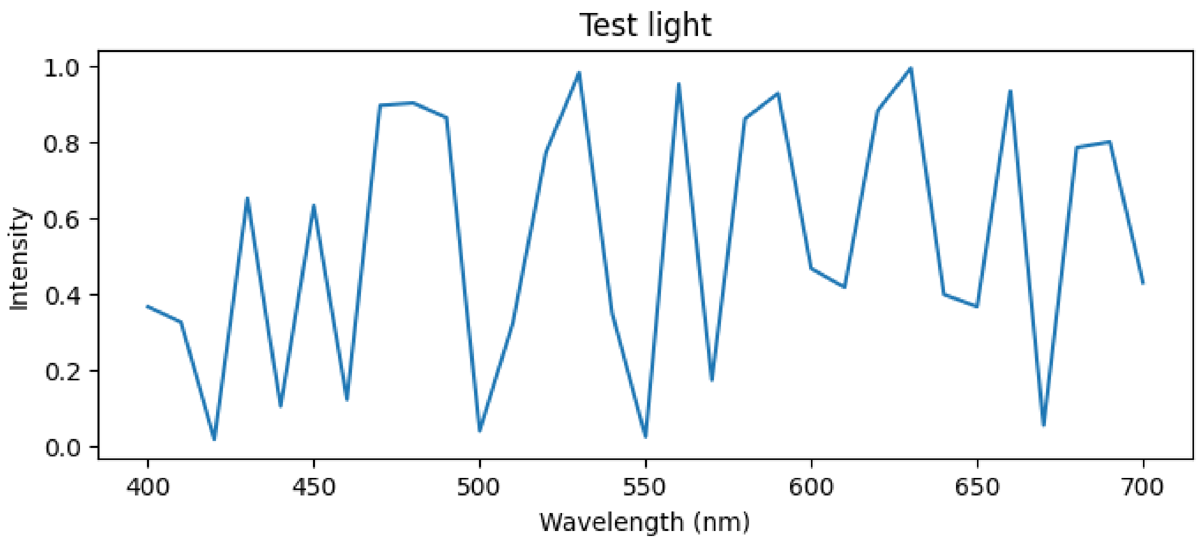
In [1]:
```
import numpy as np
import pandas as pd
import scipy
import seaborn as sns
import matplotlib.pyplot as plt
from mt2files.trichromacy import human_color_matcher as hmc
from mt2files.trichromacy import alt_human_color_matcher as hmc2

human_color_matcher = lambda x, P: np.squeeze(hmc(x, P))
alt_human_color_matcher = lambda x, P: np.squeeze(hmc2(x, P))

rng = np.random.default_rng()

obj = scipy.io.loadmat('mt2files/colMatch.mat')
primaries = obj['P']
wavelengths = np.linspace(400, 700, 31, dtype='int')
```

a)

Produce a random testlight and plot it.

In [2]:
```
testlight = rng.random(31)
```

In [3]:
```
plt.subplots(figsize=(8, 3))
sns.lineplot(pd.Series(testlight, index=wavelengths))
plt.title('Test light')
plt.xlabel('Wavelength (nm)')
plt.ylabel('Intensity')
plt.show()
```



Run an experiment with this testlight, having a human produce knob settings.

In [4]:
```
knob_settings = human_color_matcher(testlight, primaries)
knob_settings
```

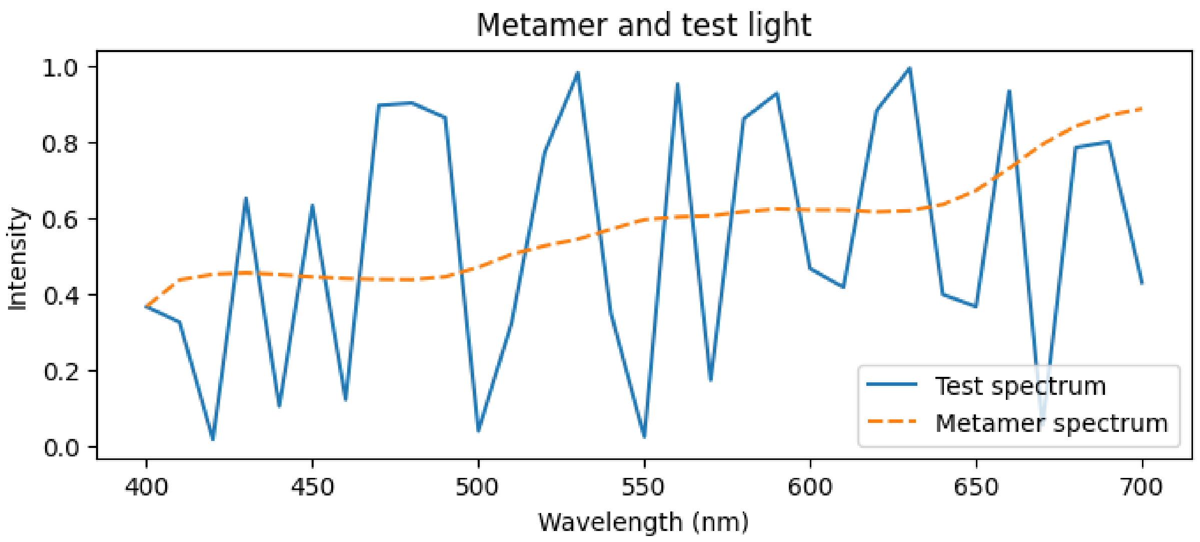
Out[4]:
```
array([0.81426568, 0.37252843, 0.41917979])
```

Combining the primaries using these knob setting, making a new light mixture. It is a metamer of the random testlight.

In [5]:
```
metamer = primaries @ knob_settings
```

Plot the spectra of the random testlight and metamer together.

In [6]:
```
plt.subplots(figsize=(8, 3))
sns.lineplot(pd.DataFrame(
    {
        'Test spectrum': testlight,
        'Metamer spectrum': metamer
    },
    index=np.linspace(400, 700, 31)
))
plt.title('Metamer and test light')
plt.xlabel('Wavelength (nm)')
plt.ylabel('Intensity')
plt.show()
```



Though two spectra are different, they are mapped to the same response by the human visual system. This is possible because the visual system has a null space: there exist certain vectors in stimulus space to which the system's response is zero. Any vector in the null space can be added to a stimulus to create a metamer for that stimulus. The two different stimuli above look the same because their difference vector lies in the null space of the visual system.

b)

If the system is linear, it has a transformation matrix `M` whose columns are the system's responses to the standard basis vectors:

```
In [7]: basis = np.identity(31)
        M = human_color_matcher(basis, primaries)
        M.shape
```

```
Out[7]: (3, 31)
```

Then multiplying any stimulus by `M` should predict the knob settings `human_color_mapper` will produce. Verify that `M` predicts the human response on 5 random test lights:

```
In [8]: testlights = np.random.rand(31, 5)
```

```
In [9]: predicted_responses = M @ testlights
        predicted_responses
```

```
Out[9]: array([[ 1.4479158 ,  0.94749064,  1.000194  ,  0.69911158,  1.21421439],
               [-0.3813556 ,  0.37734738, -0.28918319, -0.02830446,  0.20886837],
               [ 0.61507207,  0.33292647,  1.10058196,  0.61180029,  0.05377789]])
```

```
In [10]: actual_responses = human_color_matcher(testlights, primaries)
         actual_responses
```

```
Out[10]: array([[ 1.4479158 ,  0.94749064,  1.000194  ,  0.69911158,  1.21421439],
                [-0.3813556 ,  0.37734738, -0.28918319, -0.02830446,  0.20886837],
                [ 0.61507207,  0.33292647,  1.10058196,  0.61180029,  0.05377789]])
```

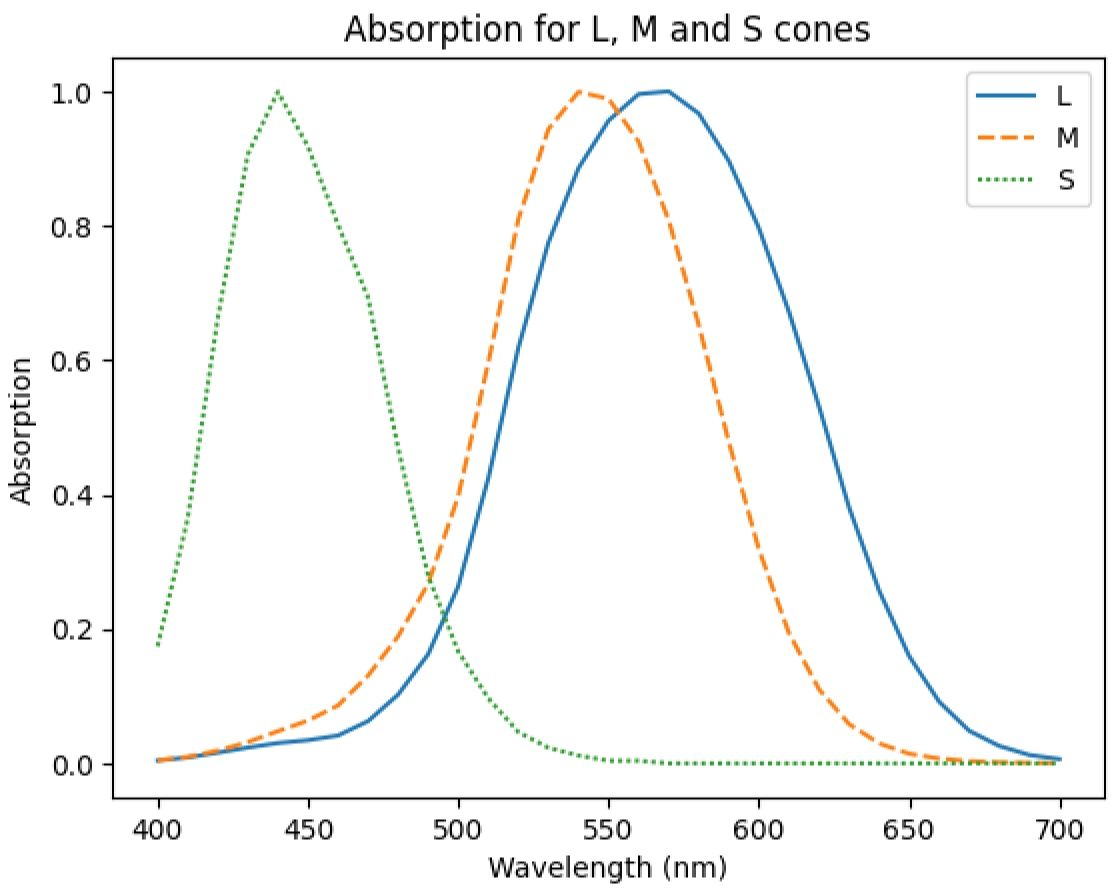
```
In [11]: np.allclose(predicted_responses, actual_responses)
```

```
Out[11]: True
```

c)

Here we have three different color photoreceptors. We can plot their spectral sensitivities:

```
In [12]: cones = obj['Cones']
         cones_df = pd.DataFrame({
             'L': cones[0, :],
             'M': cones[1, :],
             'S': cones[2, :]
         }, index=np.linspace(400, 700, 31))
         sns.lineplot(cones_df)
         plt.title('Absorption for L, M and S cones')
         plt.xlabel('Wavelength (nm)')
         plt.ylabel('Absorption')
         plt.show()
```



Verify that the cone responses are equal for two stimuli that are perceptually matched.

```
In [13]: stimulus1 = np.random.rand(31)
         stimulus2 = primaries @ human_color_matcher(stimulus1, primaries)
```

```
In [14]: response1 = cones @ stimulus1
         response1
```

```
Out[14]: array([4.80600497, 4.46938486, 3.37787137])
```

```
In [15]: response2 = cones @ stimulus2
         response2
```

```
Out[15]: array([4.80600497, 4.46938486, 3.37787137])
```

```
In [16]: np.allclose(response1, response2)
```

```
Out[16]: True
```

We can verify more rigorously that the cone responses are equal for any two metamers. Recall from **a)** that the difference between any two metamers is a stimulus in the null space of the human visual system. If the two metamer stimuli produce identical cone responses, then their difference vector is in the null space for the cones as well. Thus the cone response are equal for any two metamers if `M` and `cones` have the same null space.

We can inspect the singular value decomposition of each to see whether their null space is the same. Note that if X has the singular value decomposition $X = USV^T$, the null space of X is spanned by the columns of V which get scaled by 0 when multiplied by the S matrix.

```
In [17]: U, s, Vt = np.linalg.svd(M)
         U.shape, Vt.shape, s
```

```
Out[17]: ((3, 3), (31, 31), array([1.63144248, 0.86474169, 0.33562028]))
```

In the SVD of `M`, V has 31 columns and loses 28 of them when multiplied by the S matrix. The columns of V are orthogonal. So the null space of `M` is just $\mathbb{R}^{28}$.

```
In [18]: U, s, Vt = np.linalg.svd(cones)
         U.shape, Vt.shape, s
```

```
Out[18]: ((3, 3), (31, 31), array([3.78426349, 2.16448187, 0.7554711 ]))
```

The SVD of `cones` is similar: V has 31 columns and loses 28 when multiplied by S. So the null space of `cones` is also $\mathbb{R}^{28}$, the null space of `M`.

4)

```
In [19]: from mt2files.trichromacy import alt_human_color_matcher
```

We can compare normal knob settings with the knob settings for an abnormal human observer. Let's use 5 random test lights.

```
In [20]: testlights = np.random.rand(31, 5)
         normal_knobs = human_color_matcher(testlights, primaries)
         alt_knobs = alt_human_color_matcher(testlights, primaries)
```

```
In [21]: normal_knobs
```

```
Out[21]: array([[ 0.64249296,  0.93931338,  0.68282377,  1.28231734,  1.06882855],
                [ 0.37735342,  0.18864689,  0.83040969,  0.25140807,  0.51158365],
                [ 0.1868669 ,  0.56604701, -0.06313465, -0.12207872, -0.01354095]])
```

```
In [22]: alt_knobs
```

```
Out[22]: array([[ 0.3683977 ,  0.7008802 ,  0.42156781,  0.98842512,  0.83306135],
                [-2.87017559, -2.63635153, -2.26499712, -3.23067884, -2.28182787],
                [ 3.93190671,  3.82382594,  3.50647834,  3.8934525 ,  3.20781207]])
```

```
In [23]: np.mean(alt_knobs - normal_knobs, axis=1)
```

```
Out[23]: array([-0.26068876, -3.08868653,  3.56186319])
```

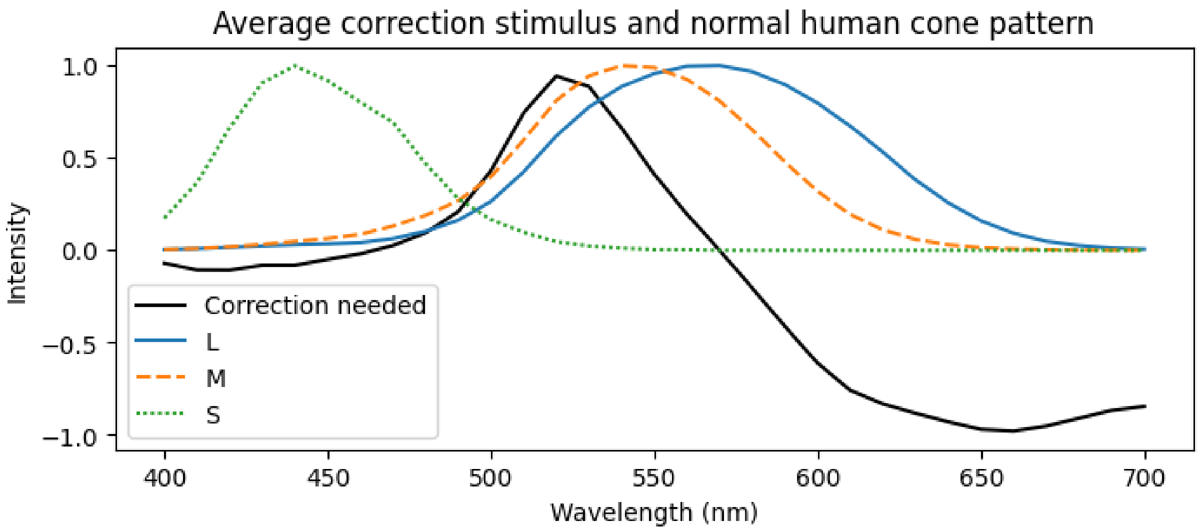
On average, the abnormal observer chooses less of the first primary than a normal human, but uses very large amounts of the second and third primary (negative and positive, respectively) to compensate. We can measure the difference settings over hundreds of trials.

```
In [24]: testlights = np.random.rand(31, 50)
         normal_knobs = human_color_matcher(testlights, primaries)
         alt_knobs = alt_human_color_matcher(testlights, primaries)
         difference_settings = np.mean(alt_knobs - normal_knobs, axis=1)
         difference_settings
```

```
Out[24]: array([-0.23189496, -2.74753248,  3.16844545])
```

What is the underlying cause of color deficiency in the alternate observer? We can diagnose this by asking what is so special about the difference settings above. This knob setting, applied to the primaries, produces the average stimulus which must be added to a random stimulus to correct that stimulus for the alternate observer. We can plot the average "correction" stimulus, and compare it to the normal human cone absorption pattern:

```
In [25]: correction_stimulus = primaries @ difference_settings
         _, ax = plt.subplots(figsize=(8, 3))
         sns.lineplot(pd.Series(correction_stimulus, index=wavelengths), ax=ax, label
         sns.lineplot(cones_df)
         plt.title('Average correction stimulus and normal human cone pattern')
         plt.xlabel('Wavelength (nm)')
         plt.ylabel('Intensity')
         plt.show()
```



It is apparent that the alternate observer needs a huge boost of intensity around the M cone wavelength in order to get the same amount of M signal as a normal human observer. This suggests a deficiency in the M cone. The alternate observer must also attenuate long wavelengths. This reduces activation of the L cone, bringing its amount of activation relative to the M cone closer to that of the normal human observer. Lastly, because the M and L frequencies have both been "turned down" relative to the normal human observer, the alternate observer must reduce S frequencies slightly to compensate. A human **lacking M cones** explains the color deficiency seen above.

Homework 2 - Question 2 - Luke Arend

In [1]:
```
import scipy
obj = scipy.io.loadmat('mt2files/regress2.mat')
D = obj['D']
```

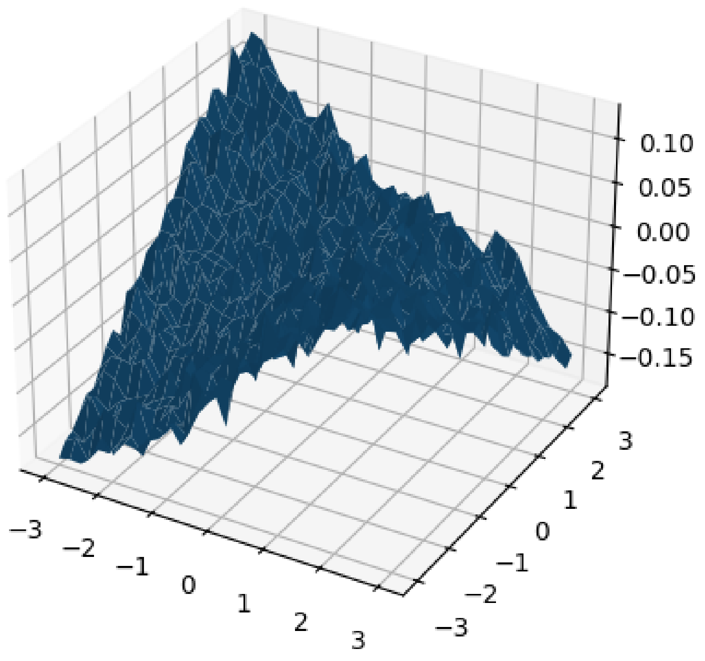
a)

In [2]:
```
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [3]:
```
x, y, z = D.T
X = np.reshape(x, (30, 30))
Y = np.reshape(y, (30, 30))
Z = np.reshape(z, (30, 30))
```

In [4]:
```
%matplotlib widget
plt.clf()
ax = plt.subplot(projection='3d')
ax.plot_surface(X, Y, Z)
plt.show()
```

Figure



b)

In [5]:
```
def solve(X):
    U, S, Vt = np.linalg.svd(X)

    n = X.shape[1]
    inds = np.arange(n)
    S_inv = np.zeros((n, 900))
    S_inv[inds, inds] = 1 / S

    X_inv = np.dot(Vt.T, np.dot(S_inv, U.T))
    beta = np.squeeze(np.dot(X_inv, z))
    return beta
```

In [6]:
```
x, y, z = D.T
n = len(x)
```

0th order: regressor is just a constant $\vec{\beta}_0$. Its error is minimized at the mean of Z.

In [7]:
```
X0 = np.ones((n, 1))
beta_0 = solve(X0)
beta_0
```

Out[7]:
```
array(-0.02218021)
```

1st order regressor has the form $\vec{\beta}_0 + \vec{\beta}_1 x + \vec{\beta}_2 y$.

In [8]:
```
X1 = np.array([np.ones(n), x, y]).T
beta_1 = solve(X1)
beta_1
```

Out[8]:
```
array([-0.02218021,  0.00621269,  0.00624989])
```

2nd order regressor: $\vec{\beta}_0 + \vec{\beta}_1 x + \vec{\beta}_2 y + \vec{\beta}_3 x^2 + \vec{\beta}_3 xy + \vec{\beta}_3 y^2$.

In [9]:
```
X2 = np.array([np.ones(n), x, y, x ** 2, x * y, y ** 2]).T
beta_2 = solve(X2)
beta_2
```

Out[9]:
```
array([-1.97707332e-02,  6.21268937e-03,  6.24989215e-03, -9.74152803e-05,
       -1.54639853e-02, -6.53927722e-04])
```

3rd order regressor:
$\vec{\beta}_0 + \vec{\beta}_1 x + \vec{\beta}_2 y + \vec{\beta}_3 x^2 + \vec{\beta}_3 xy + \vec{\beta}_3 y^2 + \vec{\beta}_2 x^3 + \vec{\beta}_3 x^2 y + \vec{\beta}_3 xy^2 + \vec{\beta}_3 y^3$.

In [10]:
```
X3 = np.array([
    np.ones(n), x, y,
    x ** 2, x * y, y ** 2,
    x ** 3, x ** 2 * y, x * y ** 2, y ** 3
]).T
```

```
beta_3 = solve(X3)
beta_3
```

```
Out[10]: array([-1.97707332e-02,  2.22461394e-02,  5.72997159e-03, -9.74152803e-05,
               -1.54639853e-02, -6.53927722e-04, -2.87107447e-03,  1.34375340e-04,
                1.60591357e-04,  1.54398032e-05])
```

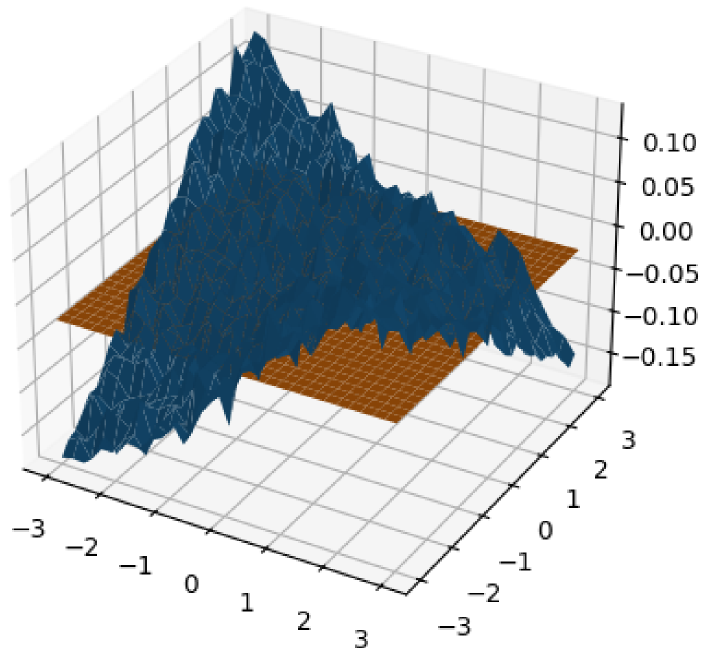
c)

```
In [11]: z0 = np.dot(X0, beta_0)
         z1 = np.dot(X1, beta_1)
         z2 = np.dot(X2, beta_2)
         z3 = np.dot(X3, beta_3)
         Z0 = np.reshape(z0, (30, 30))
         Z1 = np.reshape(z1, (30, 30))
         Z2 = np.reshape(z2, (30, 30))
         Z3 = np.reshape(z3, (30, 30))
```

The 0th order solution is just a constant level surface at the mean of our data. Not a great fit.

```
In [12]: %matplotlib widget
         plt.clf()
         ax0 = plt.subplot(projection='3d')
         ax0.plot_surface(X, Y, Z)
         ax0.plot_surface(X, Y, Z0)
         plt.show()
```

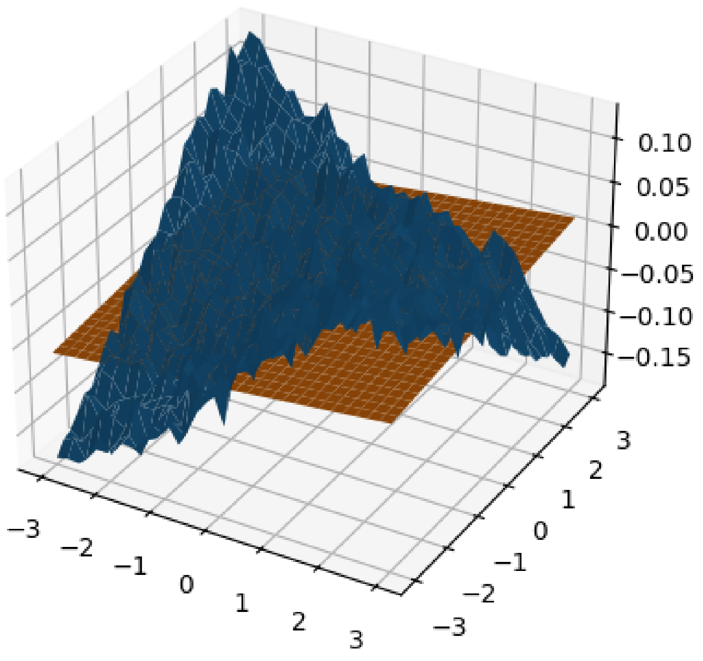
Figure



The 1st order solution is a plane, oriented as best it can to minimize the error. Still doesn't capture the underlying distribution.

```
In [13]: %matplotlib widget
plt.clf()
ax1 = plt.subplot(projection='3d')
ax1.plot_surface(X, Y, Z)
ax1.plot_surface(X, Y, Z1)
plt.show()
```

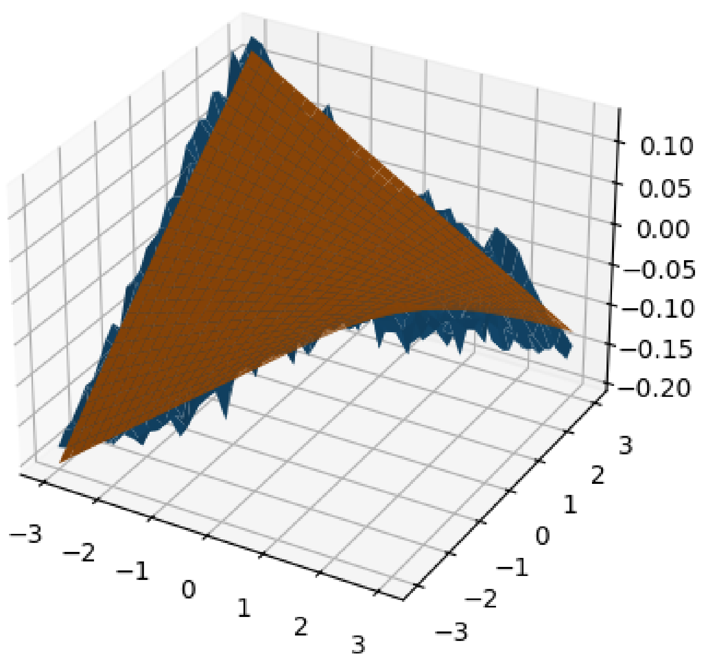
Figure



The 2th order solution has quadratic curvature and looks like a saddle in two dimensions. It fits the distribution well.

In [14]:
```
%matplotlib widget
plt.clf()
ax = plt.subplot(projection='3d')
ax.plot_surface(X, Y, Z)
ax.plot_surface(X, Y, Z2)
plt.show()
```

Figure

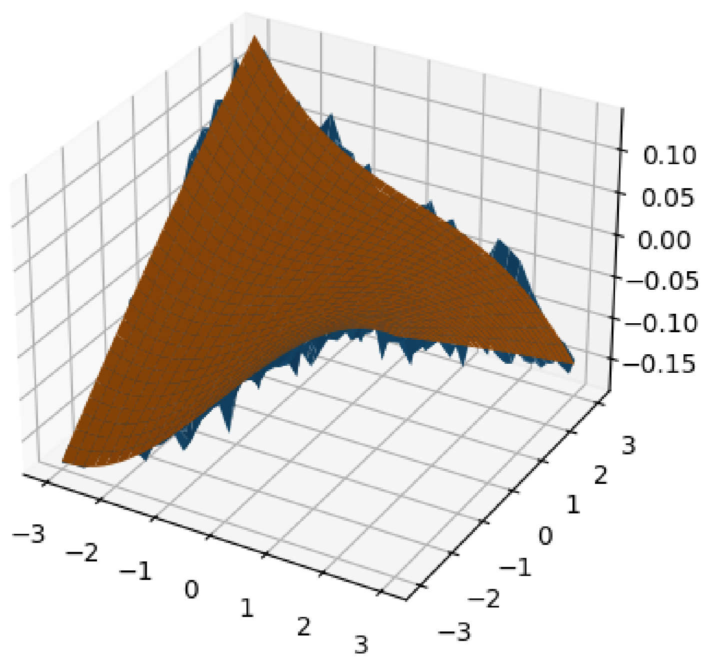


The 3th order solution is similar to the 2nd, but has slightly more curvature tracking idiosyncrasies of the distribution. It fits the distribution well, but likely would not generalize to points outside the distribution as well as the 2nd order solution.

I think the second order solution is the best - the 0th and 1st order solutions underfit, while the 3rd order solution may overfit.

In [15]:
```
%matplotlib widget
plt.clf()
ax = plt.subplot(projection='3d')
ax.plot_surface(X, Y, Z)
ax.plot_surface(X, Y, Z3)
plt.show()
```

Figure



```
In [ ]:
```

Homework 2 - Question 3 - Luke Arend

In [1]:
```
import numpy as np
import scipy
from matplotlib import pyplot as plt
import seaborn as sns

obj = scipy.io.loadmat('mt2files/constrainedLS.mat')
D = obj['data']
w = np.squeeze(obj['w'])
D.shape, w
```

Out[1]:
```
((300, 2), array([0.24, 0.54]))
```

a)

Given vectors $\{\vec{d}_n\}$ and $\vec{w}$, we want to minimize $\sum_n(\vec{\beta}^T\vec{d}_n)^2$ subject to the constraint $\vec{\beta}^T\vec{w}=1$.

In [2]:
```
U, s, Vt = np.linalg.svd(D)
```

Assume the data matrix D has the singular value decomposition USV^T. Then we have the matrix problem:

Find $\vec{\beta}$ that minimizes $||USV^T\vec{\beta}||^2$ such that $\vec{\beta}^T\vec{w}=1$.

We apply a change of variables to $\vec{\beta}$ to eliminate V^T. In particular we define a new variable $\vec{\beta}^*$ which is $\vec{\beta}$ rotated by V^T. The new constraint vector becomes $\vec{w}^*=V^T\vec{w}$. Then our problem is:

Find $\vec{\beta}^*$ that minimizes $||US\vec{\beta}^*||^2$ such that $\vec{\beta}^{*T}\vec{w}^*=1$.

To eliminate S we scale $\vec{\beta}^*$ by S. This gives us a new variable $\tilde{\beta}=S\vec{\beta}^*$ and new constraint vector $\tilde{w}=S^{-1}\vec{w}^*$. Now our problem is:

Find $\tilde{\beta}$ that minimizes $||U\tilde{\beta}||^2$ such that $\tilde{\beta}^T\tilde{w}=1$.

b)

U rotates $\tilde{\beta}$ by an orthonormal matrix, leaving its magnitude unchanged. So the quantity to be minimized is just $||\tilde{\beta}||^2$.

The shortest $\tilde{\beta}$ which satisfies $\tilde{\beta}^T\tilde{w}=1$ must point in the same direction as $\tilde{w}$ and have length $\frac{1}{||\tilde{w}||}$, so that its dot product with $\tilde{w}$ is 1. Therefore

$$\tilde{\beta} = \frac{\tilde{w}}{\|\tilde{w}\|^2}.$$

And we can write $\tilde{w}$ in terms of the initial constraint $\vec{w}$:

$$\tilde{w} = S^{-1}\vec{w}^* = S^{-1}V^T\vec{w}.$$

In [3]:
```
w1 = np.diag(1 / s) @ Vt @ w
b1 = w1 / (w1 @ w1)
w1, b1
```

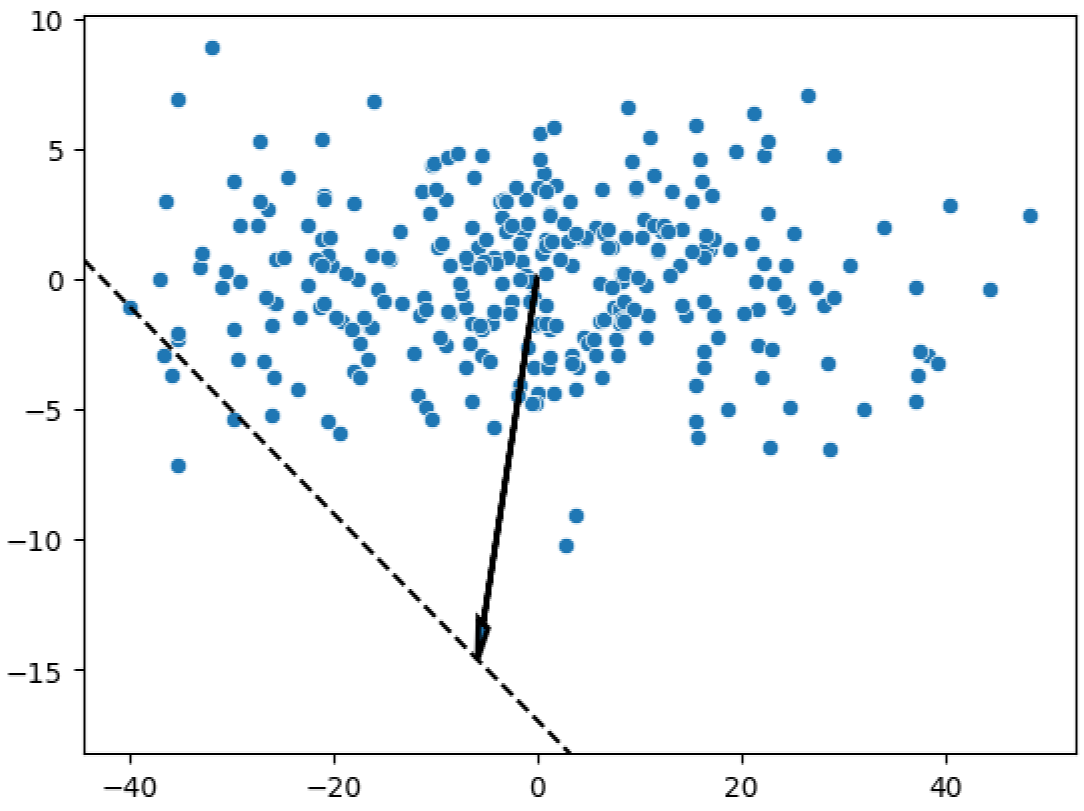
Out[3]:
```
(array([-0.0234546 , -0.05898736]), array([ -5.82054025, -14.63841709]))
```

We plot the solution vector, constraint line and transformed data points below.

The constraint line has the equation $\tilde{\beta}^T\vec{w} = 1$ or $\tilde{\beta}_0 w_0 + \tilde{\beta}_1 w_1 = 1$. So it is the line where (β_0, β_1) satisfies $\beta_1 = \frac{1}{w_1} - \frac{w_0}{w_1}\beta_0$.

The transformed data points come from D applying the variable changes to D^T and transposing back.

In [4]:
```
D1 = (np.diag(s) @ Vt @ D.T).T
y0 = 1 / w1[1]
m = -w1[0]/w1[1]
sns.scatterplot(x=D1[:, 0], y=D1[:, 1])
plt.axline((0, y0), slope=m, color="black", linestyle='--')
plt.arrow(0, 0, b1[0], b1[1], head_width=1, linewidth=2,
          length_includes_head=True)
plt.show()
```



c)

To transform the solution back to the original space, shrink $\tilde{\beta}$ by S and rotate by V:

$$\vec{\beta} = VS^{\#}\tilde{\beta}$$

```
In [5]: b = Vt.T @ np.diag(1 / s) @ b1
        b
```

```
Out[5]: array([-0.3782429 ,  2.01995981])
```

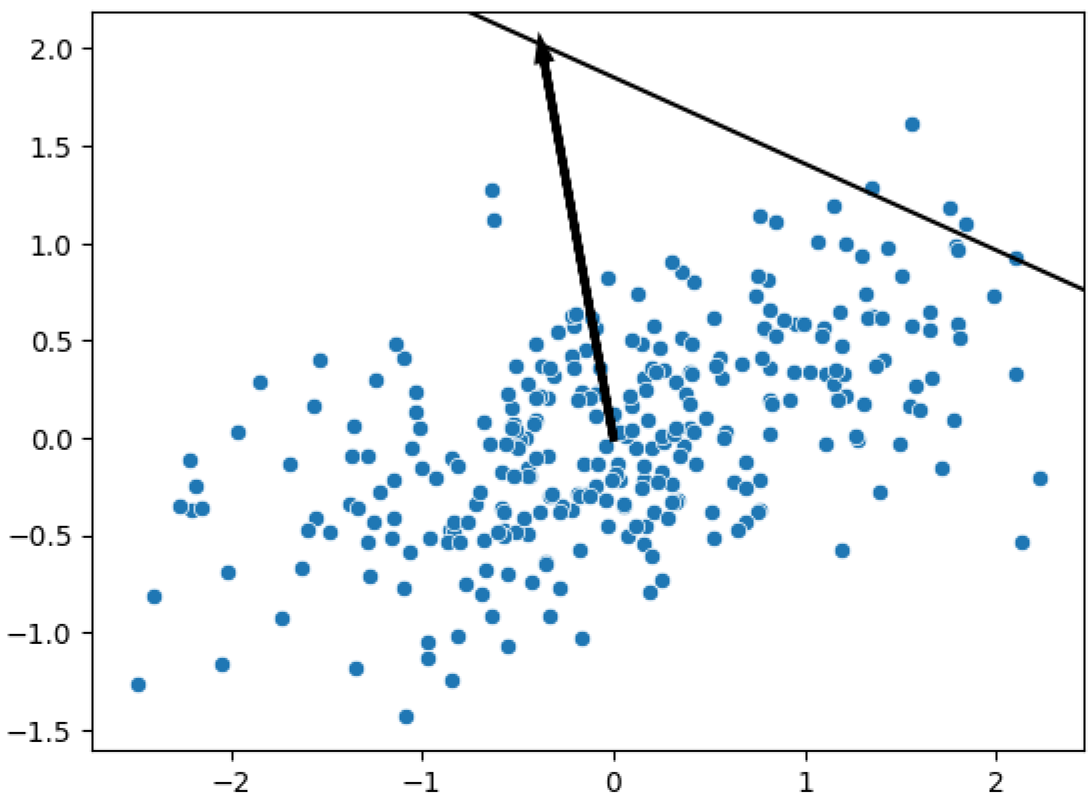
This $\vec{\beta}$ solves the initial constraint.

```
In [6]: b @ w
```

```
Out[6]: 1.0
```

We can plot the $\vec{\beta}$, the original constraint line, and the original data.

```
In [7]: sns.scatterplot(x=D[:, 0], y=D[:, 1])
        plt.axline((0, 1 / w[1]), slope=-w[0]/w[1], color="black")
        plt.arrow(0, 0, b[0], b[1], head_width=0.05, linewidth=3, color='k',
                  length_includes_head=True)
        plt.show()
```



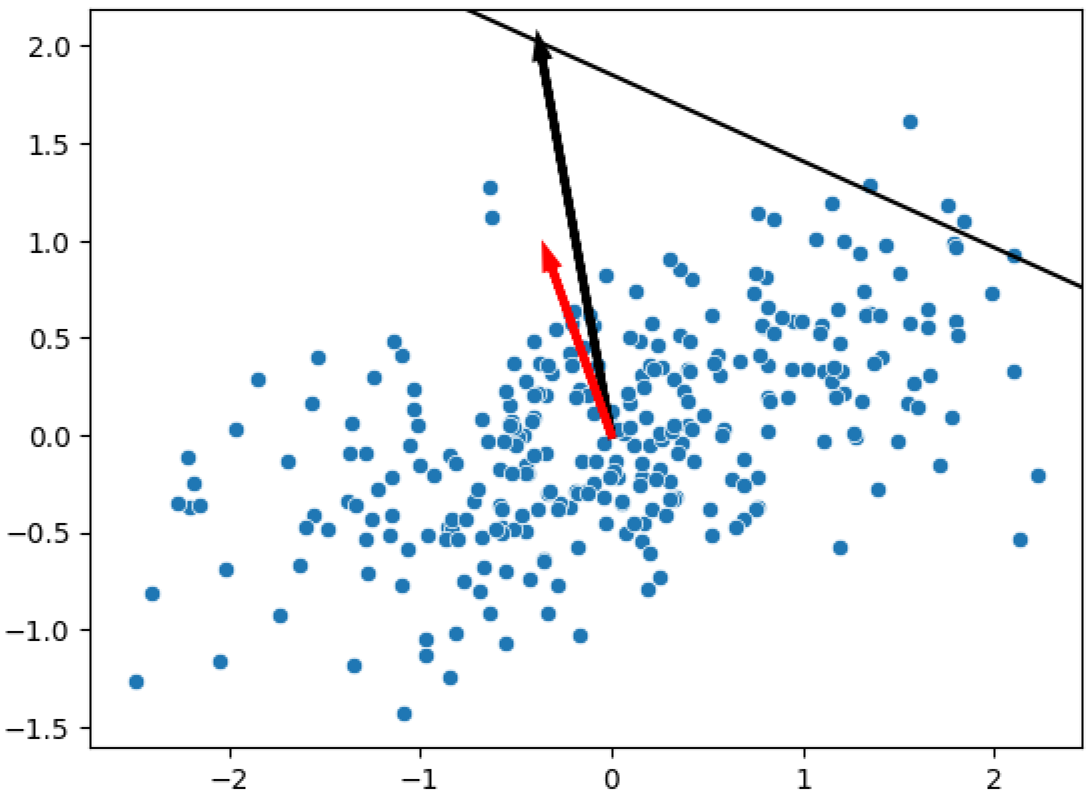
The optimal vector $\vec{\beta}$ is not perpendicular to the constraint line. If $\vec{\beta}$ were perpendicular to (and lying on) the constraint line it would satisfy the constraint, but would then have greater dot products with the points in D, no longer minimizing $\sum_n (\vec{\beta}^T \vec{d}_n)^2$.

What is the total least squares solution (i.e. the vector that minimizes the $||D\hat{\beta}||^2$ where $\hat{\beta}$ is a unit vector)? Geometrically, $\hat{\beta}$ should point in the direction of the *minimum* spread of the data points in D. This will be the second principle component of D, or the second eigenvector of $D^T D$.

In [8]:
```
C = D.T @ D
U, s, Vt = np.linalg.svd(C)
b_hat = Vt.T[:, 1]
```

Now we plot again showing the least squares solution vector in red. The solutions are not the same.

In [9]:
```
sns.scatterplot(x=D[:, 0], y=D[:, 1])
plt.axline((0, 1 / w[1]), slope=-w[0]/w[1], color="black")
plt.arrow(0, 0, b[0], b[1], head_width=0.05, linewidth=3, color='k',
          length_includes_head=True)
plt.arrow(0, 0, b_hat[0], b_hat[1], head_width=0.05, linewidth=3, color='r',
          length_includes_head=True)
plt.show()
```



```
In [ ]:
```

Homework 2 - Question 4 - Luke Arend

In [1]:
```
import numpy as np
import scipy
from matplotlib import pyplot as plt
import seaborn as sns
```

In [2]:
```
obj = scipy.io.loadmat('mt2files/PCA.mat')
M = obj['M']
```

a)

Compute the principle components of the population response via the SVD of `M`. First, mean-center the data:

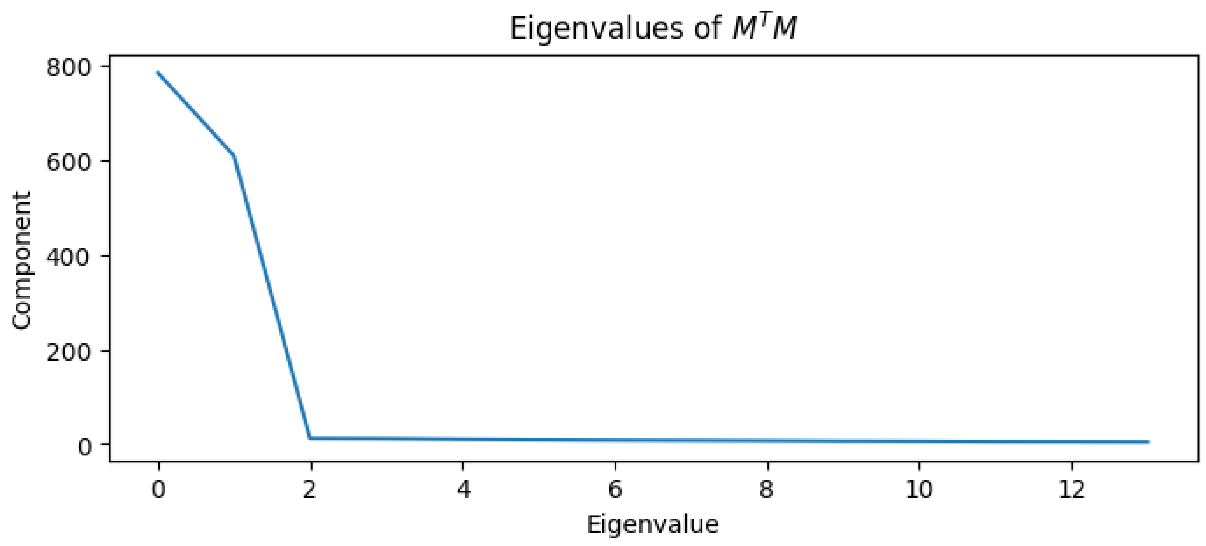
In [3]:
```
M -= np.mean(M, axis=0)
```

Next, compute eigenvectors of the covariance matrix $C = \tilde{M}^T \tilde{M}$. Let $M = USV^T$. Then the eigenvalues of C are the squared singular values of $\tilde{M}$ and its eigenvectors are the columns of V:

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

In [5]:
```
eigenvalues = s ** 2
eigenvectors = Vt
```

In [6]:
```
plt.subplots(figsize=(8, 3))
plt.plot(eigenvalues)
plt.title('Eigenvalues of $M^T M$')
plt.xlabel('Eigenvalue')
plt.ylabel('Component')
plt.show()
```

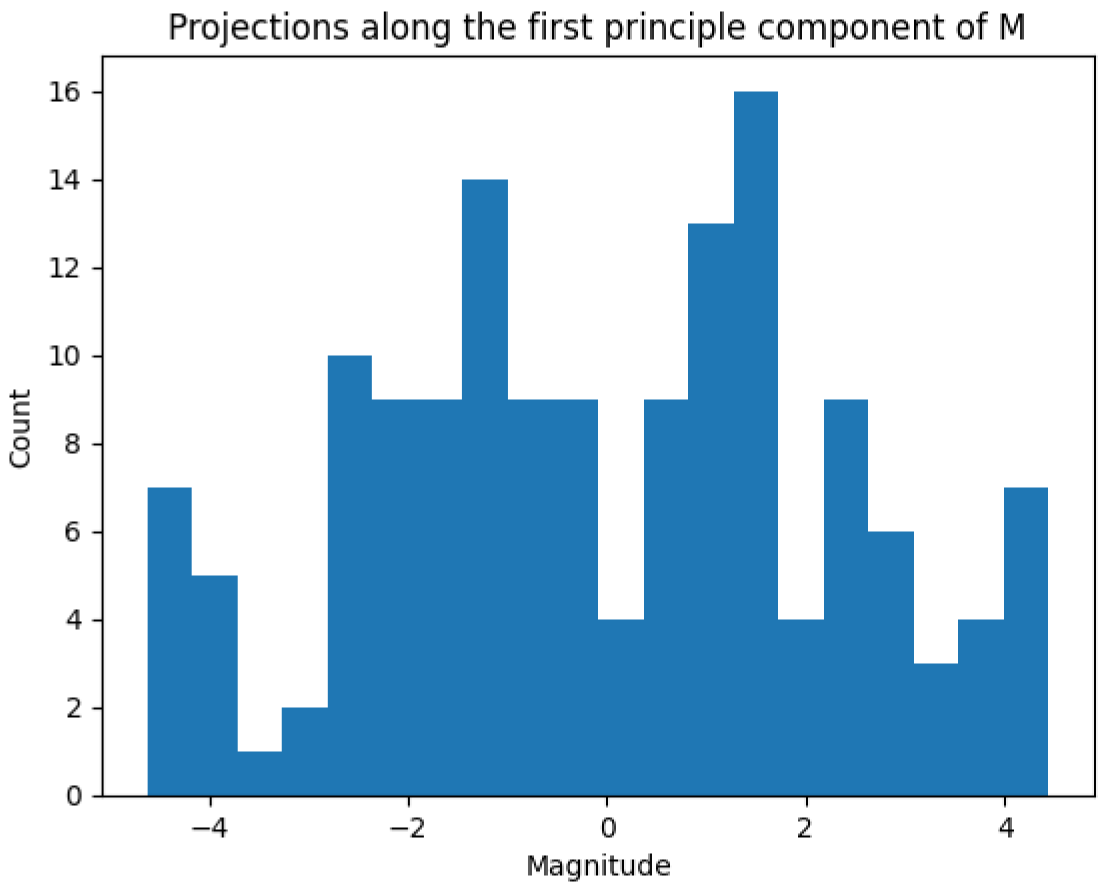


M has only two components with large eigenvalues. This suggests that the underlying dimensionality of the data is 2.

b)

Now we project $\tilde{M}$ onto the first principle component and plot a histogram of the values.

```
In [7]: v1 = eigenvectors[0]
vals = M @ v1
plt.hist(vals, bins=20)
plt.title('Projections along the first principle component of M')
plt.xlabel('Magnitude')
plt.ylabel('Count')
plt.show()
```



The values are quite spread out. We can verify that the sum of their squared magnitude is the first eigenvalue of C.

```
In [8]: eigenvalues[0]
```

```
Out[8]: 784.4378048854699
```

```
In [9]: np.sum(vals ** 2)
```

```
Out[9]: 784.4378048854713
```

```
In [10]: explained_variance = np.sum(vals ** 2)
         total_variance = np.sum(M ** 2)
         explained_variance / total_variance
```

```
Out[10]: 0.5297683493388249
```

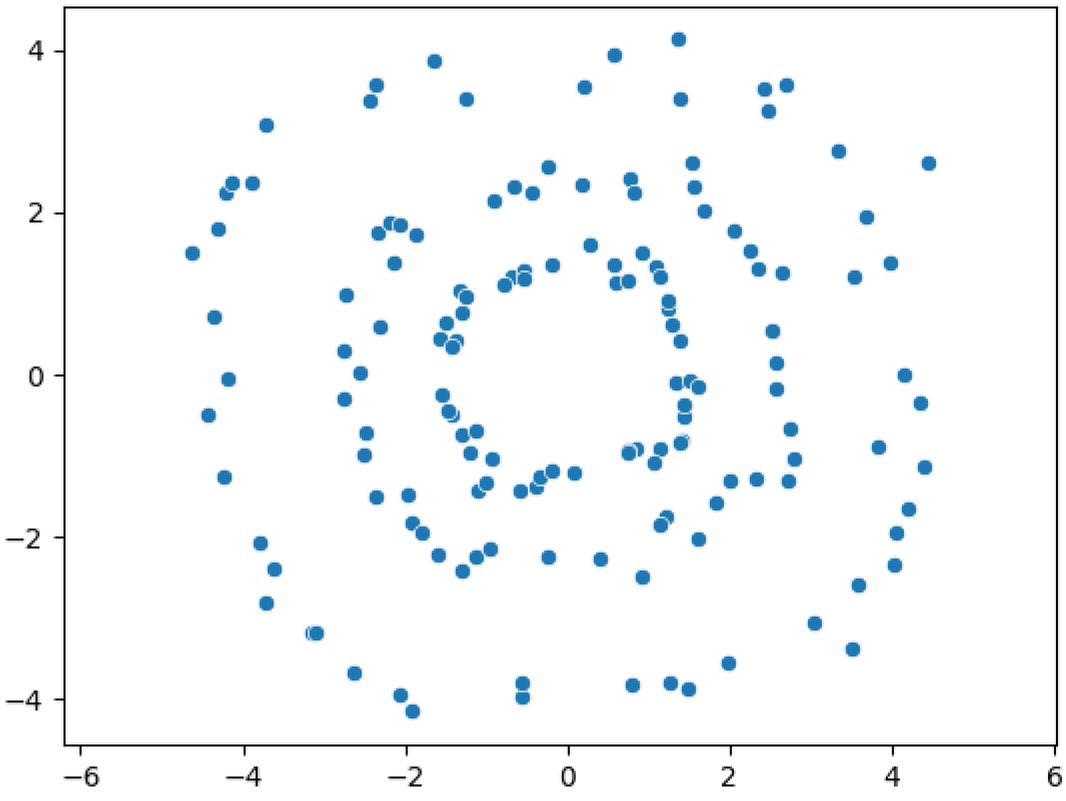
The first eigenvector accounts for 52.9% of the total variance in M.

c)

Below we can plot the distribution projected onto the first two principle components.

```
In [11]: sns.scatterplot(x=M @ eigenvectors[0], y=M @ eigenvectors[1])
         plt.axis('equal')
```

```
Out[11]: (-5.081171038629532, 4.901658189780306, -4.576938316111025, 4.5400232817188
         675)
```



The squared lengths of these projected vectors is the sum of λ_1 and λ_2.

```
In [12]: eigenvalues[0] + eigenvalues[1]
```

```
Out[12]: 1393.8714772402632
```

```
In [13]: Y = np.array([M @ eigenvectors[0], M @ eigenvectors[1]])
         var = np.sum([np.sum(row ** 2) for row in Y])
         var
```

```
Out[13]: 1393.8714772402645
```

We can measure the total variance to see how much variance is explained by λ_1 and λ_2.

```
In [14]: totalvar = np.sum([np.sum(row ** 2) for row in M @ eigenvectors])
         totalvar
```

```
Out[14]: 1480.718517564301
```

```
In [15]: var / totalvar
```

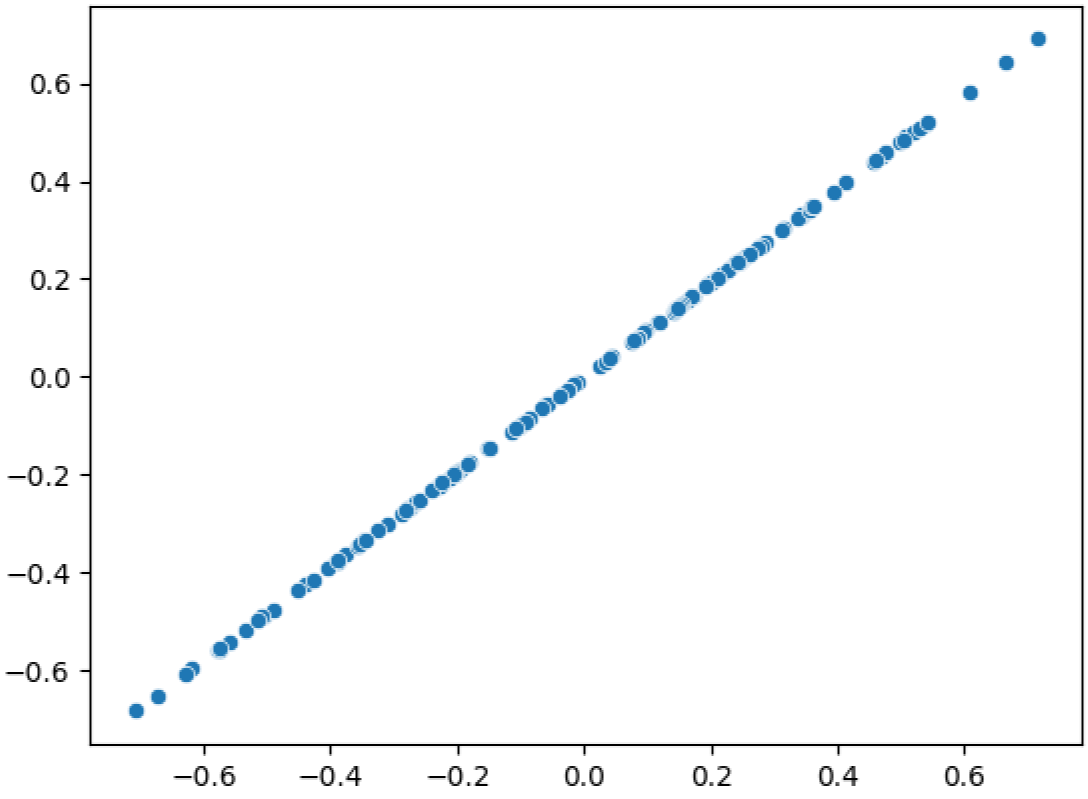
```
Out[15]: 0.9413480419851202
```

So the first two principle components explain 94.1% of the total variance.

d)

We can project the data onto the first two principle components only and plot it. This reconstructs the data from a "compressed" form that explains 94.1% of the variance using two instead of 14 dimensions.

In [16]:
```
pcs = eigenvectors.copy()
pcs[2:, :] = 0
Y = M @ pcs
sns.scatterplot(x=Y[:, 0], y=Y[:, 1])
plt.show()
```



In [ ]: