Applications of Singular Value Decomposition

Math 157 Final Project

Luke Fahmy, A13849289

The *singular value decomposition (SVD) (https://en.wikipedia.org/wiki/Singular_value_decomposition)* is a factorization of any matrix M of the form $M = U\Sigma V^*$, where M is an $m \times n$ matrix, U is an $m \times m$ unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and V is an $n \times n$ unitary matrix.

The diagonal entries $\sigma_i = \Sigma_{ii}$ of Σ are called the "singular values" of M, and the number of non-zero singular values is equal to the *rank (https://en.wikipedia.org/wiki/Rank_(linear_algebra))* of M.

The SVD has many useful and practical applications. Among other things, it can be used to:

- Find the pseudoinverse of a matrix
- Solve homogenous linear equations
- Solve a total least squares minimization (as we saw in the homework)
- Easily represent the range, null space, and rank of a matrix
- Find low rank approximations to a matrix
- Find the nearest orthogonal matrix to a matrix

We will focus on the pseudoinverse, the low rank approximation, and the nearest orthogonal matrix. We will look at a practical application of each.

SVD and the Pseudoinverse for Systems with no solution

The *pseudoinverse (https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse)* A^+ of a matrix A is a general inverse for ANY matrix. For invertible matrices, $A^+ = A^{-1}$. For non-invertible matrices, the pseudoinverse has many properties, most importantly that $AA^+A = A$ and $A^+AA^+ = A^+$. In the first equation, AA^+ is acting as a left-identity. In the second equation, AA^+ is acting as a right-identity.

The SVD is very useful for computing the pseudoinverse. For $A = U\Sigma V^*$, $A^+ = V\Sigma^+ U^*$. We can get Σ^+ by taking the reciprocal of each non-zero element on the diagonal, leaving the zeroes in place, then transposing the matrix.

Let's have a look at the pseudoinverse with a matrix that is not square, so is definitely not invertible.

In [1]:
```
import matplotlib.pyplot as plt # Used to show the images
from skimage import data # Provides test images
from skimage.color import rgb2gray

import numpy.linalg
import numpy
```

In [2]:
```
image = data.chelsea() # Grabbing the image
image = rgb2gray(image) # Converting it to grayscale
print(image.shape)
print(numpy.linalg.matrix_rank(image))
```

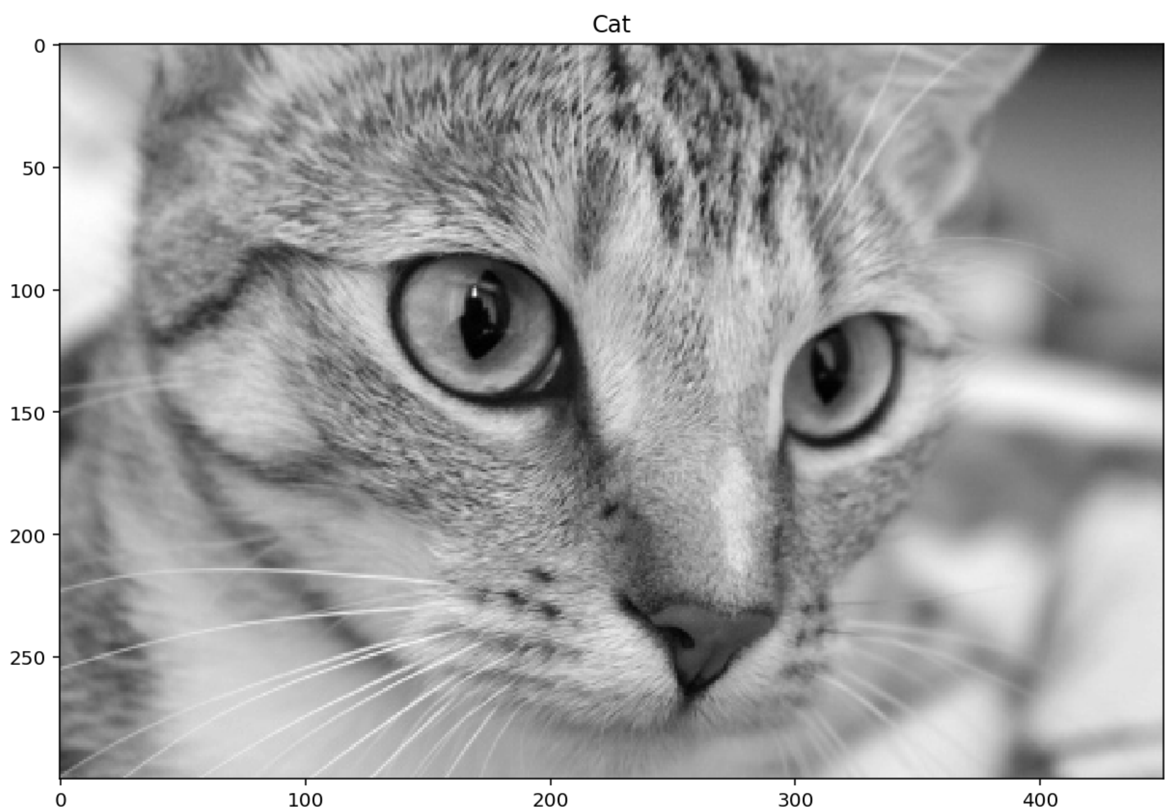
```
(300, 451)
300
```

We've loaded an image of a cat (apparently named Chelsea). We'll try to find some inverse of the image.

In [3]:
```
plt.figure()
plt.title("Cat")
plt.imshow(image, cmap="gray")
```

Out[3]: <matplotlib.image.AxesImage at 0x7f37a787c198>

Out[3]:



In [4]:
```
numpy.linalg.inv(image) # This gives us an error. Let's try SVD instead
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call
 last)
<ipython-input-4-9cb5d89e6d6f> in <module>
----> 1 numpy.linalg.inv(image) # This gives us an error. Let's try SV
D instead

<__array_function__ internals> in inv(*args, **kwargs)

/usr/local/lib/python3.6/dist-packages/numpy/linalg/linalg.py in inv
(a)
    540     a, wrap = _makearray(a)
    541     _assert_stacked_2d(a)
--> 542     _assert_stacked_square(a)
    543     t, result_t = _commonType(a)
    544 

/usr/local/lib/python3.6/dist-packages/numpy/linalg/linalg.py in _asse
rt_stacked_square(*arrays)
    211         m, n = a.shape[-2:]
    212         if m != n:
--> 213             raise LinAlgError('Last 2 dimensions of the array
 must be square')
    214 
    215 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

In [5]:
```
U, s, V = numpy.linalg.svd(image)
```

Remember, we want

$$A^{+} = V\Sigma^{+}U^{*}.$$

In [6]:
```
sigma_plus = 1/s # Taking the reciprocal of all non-zero singular values
sigma_plus = numpy.diag(sigma_plus) # Turning our diagonal into a matrix

# We have to add the extra zeros that were cut off by numpy to get the pr
oper dimension
extra_zeros = numpy.zeros((300, 151))
sigma_plus = numpy.hstack((sigma_plus, extra_zeros))
print(sigma_plus.shape)

sigma_plus = sigma_plus.transpose()
```

```
(300, 451)
```

In [7]:
```
image_plus = V.transpose()@sigma_plus@U.transpose() # We can just do tran
spose instead of conj transpose, since U and V are real
```

Now we have a pseudoinverse for our image. We used this image as an example because it is not square, but the pseudoinverse is not always very useful when it comes to images. However it IS useful for applications using linear systems, which are many. We'll see a more useful application of the pseudoinverse in the exercises. Also, the SVD provided us a convenient way to compute an inverse of ANY matrix. Since $A^+ = A^{-1}$ for an invertible matrix, we can use this process to get a good inverse for any matrix. This has the advantage of not having to check the dimensions or invertibility of a matrix before doing the computation.

SVD and the Low-rank matrix approximation for Image compression

Sometimes, we want to [approximate (https://en.wikipedia.org/wiki/Low-rank_approximation)](https://en.wikipedia.org/wiki/Low-rank_approximation) some matrix A with a matrix $\tilde{A}$ of some lower rank r. We can use the SVD $A = U\Sigma V^*$ to get $\tilde{A} = U\tilde{\Sigma}V^*$. $\tilde{\Sigma}$ is the same as Σ except that it only contains the r largest singular values, and the other singular values are replaced by zeroes.

In this case, our matrix will represent an image, where each entry is the value of a pixel. We can try to compute a lower rank approximation of this image to save space.

First we'll have a look at our image.

In [8]:
```
image = data.astronaut() # Grabbing the image
image = rgb2gray(image) # Converting it to grayscale
plt.figure()
plt.title("Astronaut")
plt.imshow(image, cmap='gray')
```

Out[8]: `<matplotlib.image.AxesImage at 0x7f37a734fc50>`

Out[8]:

Our image is a numpy array, so this makes it easy to compute the SVD.

In [9]:
```
print(type(image))
```

```
<class 'numpy.ndarray'>
```

In [10]:
```
U, s, V = numpy.linalg.svd(image)
print(len(s), len(s) == numpy.linalg.matrix_rank(image)) # The number of
 singular values is the rank of the image
```

```
512 True
```

Now, say we want an image about half the size, but no less than that. If we compute the sum of the singular values, we can find a subset of them that add up to half the total sum. Usually we go from largest singular value to smallest, until we hit our target sum.

In [11]:
```
total = sum(s)
goal = .5*total
goal
```

Out[11]:
```
571.4682311251961
```

In [12]:
```
sum_so_far = 0
for i in range(len(s)):
    sum_so_far += s[i]
    if sum_so_far >= goal:
        print(i, sum_so_far, s[i])
        break
```

```
11 581.3555158732051 15.53577904037669
```

So we can see that we want to use the first 12 singular values (the 0th to the 11th). Amazing! More than 50% of our image data can be accessed by 12 out of 512 of our singular values. Now we must get our $\tilde{\Sigma}$ by replacing every unused σ_i in Σ with 0.

In [13]:
```
print(s[0:15])
numpy.put(s, [x for x in range(12, len(s))], 0) # Puts a 0 in all the spe
cified indices
print(s[0:15])
```

```
[241.25751778  71.12185581  45.97712758  38.00791552  31.25038842
  29.98925186  28.45683894  21.94304069  21.85073754  18.91835586
  17.04670683  15.53577904  14.99775336  13.720993    12.69611465]
[241.25751778  71.12185581  45.97712758  38.00791552  31.25038842
  29.98925186  28.45683894  21.94304069  21.85073754  18.91835586
  17.04670683  15.53577904   0.           0.           0.        ]
```

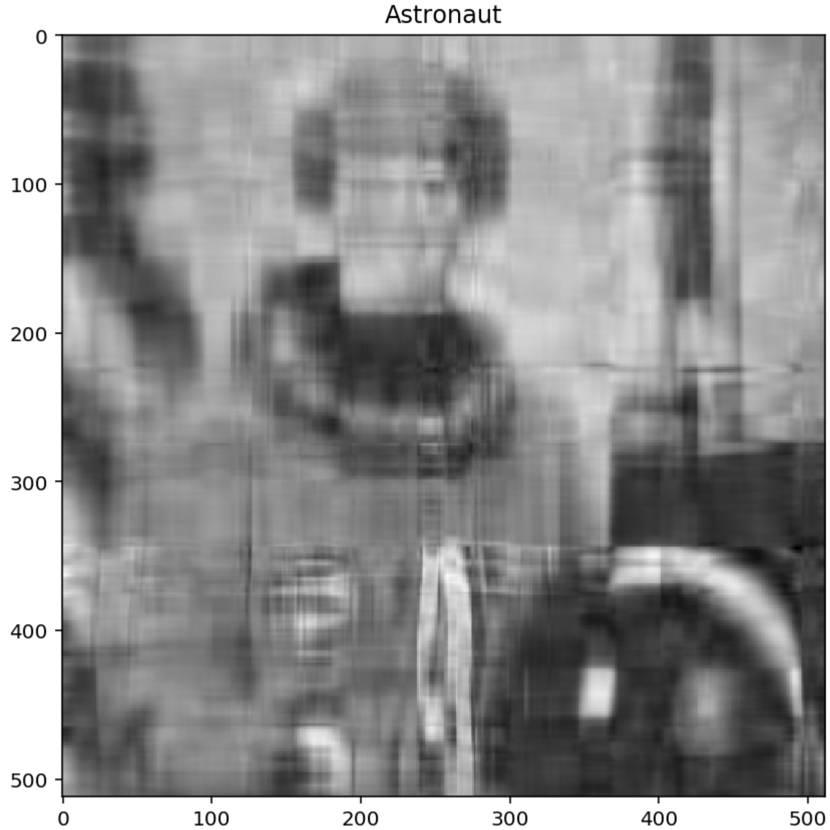
In [14]:
```
S = numpy.diag(s) # Turn s into a diagonal matrix for multiplication
```

Now we can construct our new image by multiplying all of the factors of its matrix representation together. Our new image matrix now has rank 12, down from 512, since we dropped all the parts of the original matrix corresponding to the zeroed singular values.

In [15]:
```
image = U@S@V
plt.figure()
plt.title("Astronaut")
plt.imshow(image, cmap='gray')
```

Out[15]: `<matplotlib.image.AxesImage at 0x7f37a72b9518>`

Out[15]:



SVD and the Nearest orthogonal matrix for Shape analysis

For a matrix A, we can use the SVD $A = U\Sigma V^*$ to compute the *orthogonal matrix (https://en.wikipedia.org/wiki/Orthogonal_matrix)* O nearest to A. In fact, $O = UV^*$. For this lecture, we will focus on the similar problem of finding an orthogonal matrix O which most closely maps A to B. This is called the *orthogonal Procustes problem (https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem)*, and is written:

$$O = \arg\min_{\Omega} \|A\Omega - B\|_F \quad \text{subject to} \quad \Omega^\mathsf{T}\Omega = I$$

In this case, we still have $O = UV^*$, but our matrix is $M = BA^T$, where we seek to map A to B.

A fun and interesting application of this is in the world of shape analysis, where we seek to get some measure of how similar and different a set of shapes are to each other.

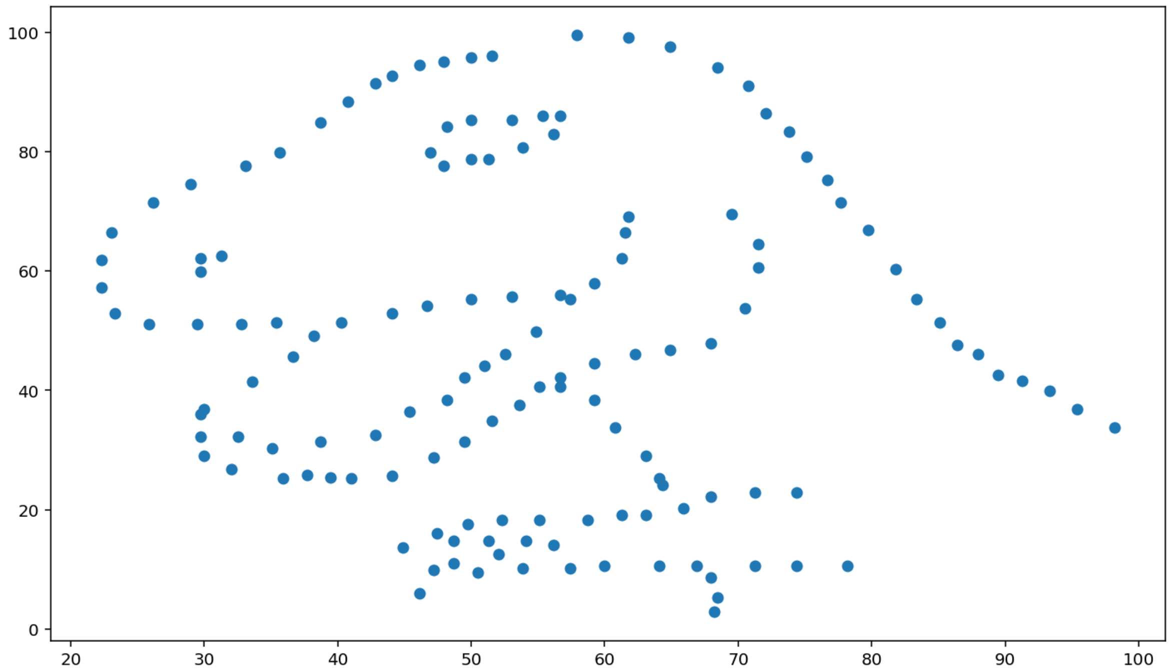
In [16]:
```
import pandas

good_dino = pandas.read_csv("Datasaurus_data.csv").to_numpy()
good_dino = good_dino.transpose()
print(good_dino.shape) # We want each point in its own column
```

```
(2, 141)
```

In [17]:
```
plt.scatter(good_dino[0], good_dino[1])
```

Out[17]: <matplotlib.collections.PathCollection at 0x7f379a500400>

Out[17]:



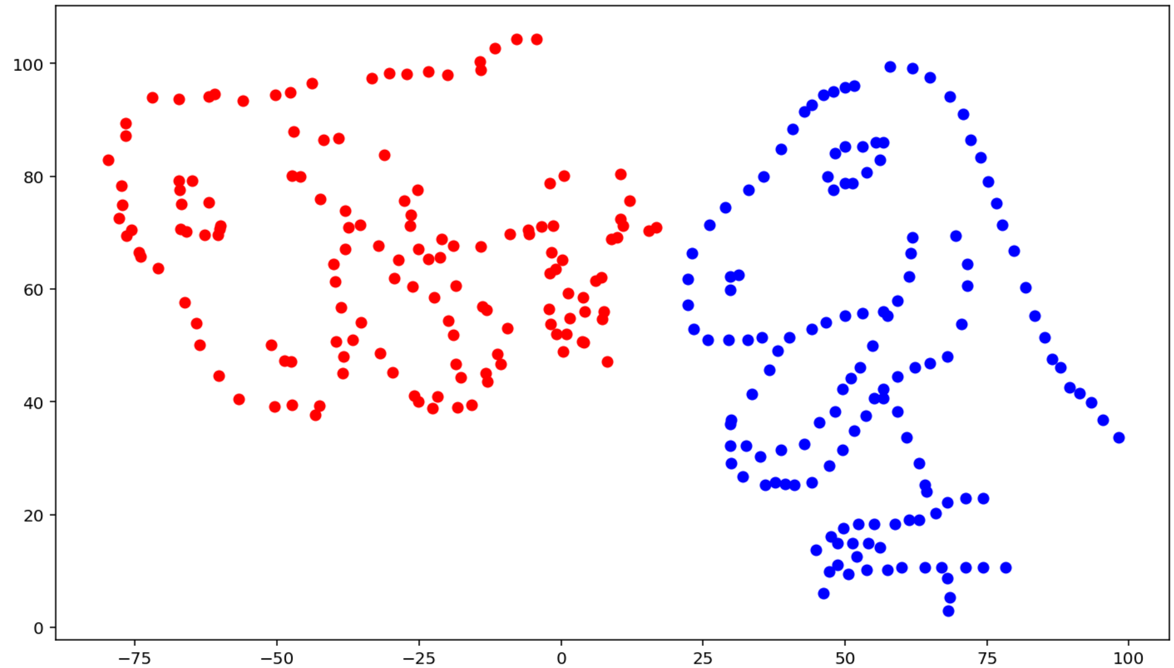
Now say we have a similar shape but transformed a bit, and we want to see just how similar it is to our dinosaur. In this case, because we are searching for an orthogonal matrix, the only major transformation we can use is rotation.

In [18]:
```
bad_dino = numpy.copy(good_dino) # Only want to copy the values

# Creating a rotation matrix that will rotate the points 75 degrees
theta = numpy.radians(75)
c, s = numpy.cos(theta), numpy.sin(theta)
R = numpy.array(((c, -s), (s, c)))

bad_dino = R@bad_dino
bad_dino = bad_dino + numpy.random.normal(loc=2, size=good_dino.shape) #
 Add some noise so that we don't have EXACTLY the same shape
```

In [19]:
```
plt.scatter(good_dino[0], good_dino[1], c='b')
plt.scatter(bad_dino[0], bad_dino[1], c='r')
```

Out[19]: <matplotlib.collections.PathCollection at 0x7f379a46dd30>

Out[19]:



We can see that this new dinosaur looks similar, but definitely not as clear as the other one. Now we'll do our shape analysis. We'll try to map bad_dino to good_dino using the process we've already outlined.

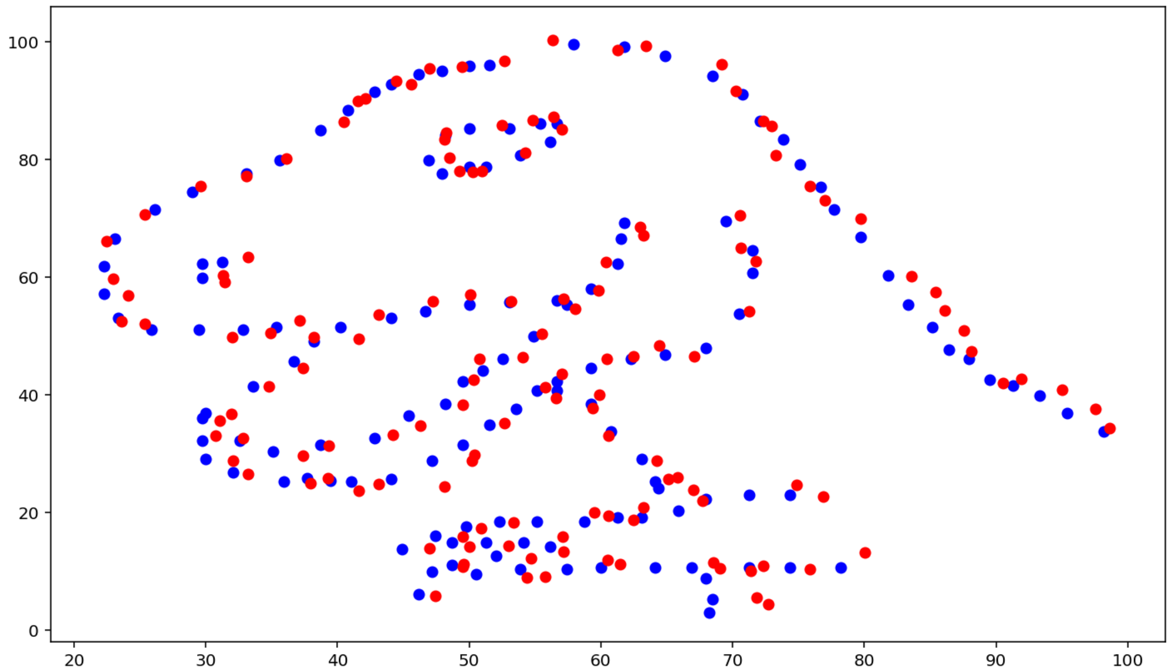
In [20]:
```
A = bad_dino
B = good_dino

M = B@A.transpose()
U, s, V = numpy.linalg.svd(M)
```

O is what maps bad_dino to good_dino (A to B), so we do $O * A$ to get our approximation of B. Once we have bad_dino superimposed on good_dino, we can do an eye test and a mathematical test for similarity.

In [21]:
```
O = U@V
A_to_B = O@A

plt.scatter(good_dino[0], good_dino[1], c='b')
plt.scatter(A_to_B[0], A_to_B[1], c='r')
```

Out[21]: <matplotlib.collections.PathCollection at 0x7f379a3d8eb8>

Out[21]:



We can use the square root of the sum of squared distances between corresponding points for our mathematical similarity test. If we represent the points of one shape by $\{(u_1, v_1), (u_2, v_2), \cdots, (u_n, v_n)\}$ and the other by $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$, then we have:

$$d = \sqrt{(u_1 - x_1)^2 + (v_1 - y_1)^2 + \cdots}$$

In [22]:
```
d = numpy.sum((A_to_B[0] - B[0])**2 + (A_to_B[1] - B[1])**2)
d = d**.5
print(d)
```

```
23.574196323359068
```

Exercises

Problem 1: Using Pseudoinverse and Low-rank approximation to recover an image

1a. You are given a corrupted image that has been transformed by the given matrix H. We have F = G*H, where F is the image you've been given, G is the original image, and H is the corrupting transformation. Use the SVD, H, and F to recover G the best you can.

In [47]:
```
### Don't touch
G = data.chelsea()
G = rgb2gray(cat)
H = numpy.zeros((451, 400))
H = H + numpy.random.normal(loc=0, size=H.shape)
F = G@H

### Your code starts here
```

1b. Now use the SVD in a different way to make the recovered image look even better. (Hint: different singular values represent different horizontal/vertical lines in an image)

In [0]:
```

```

Problem 2: Using Orthogonal Procrustes method to guess axis of symmetry of a square

1. Pretend you have never seen a square before, but you have a set of points representing a square. Write a program that, given the square, guesses an axis of symmetry.

In [102]:
```
### Don't touch
B = [(x, 100) for x in range(100, 201, 5)]
R = [(200, x) for x in range(100, 201, 5)]
T = [(x, 200) for x in range(100, 201, 5)]
L = [(100, x) for x in range(100, 201, 5)]
square = numpy.array(B)
square = numpy.vstack((B, R, T, L))
square = square.transpose()

### Your code starts here
```

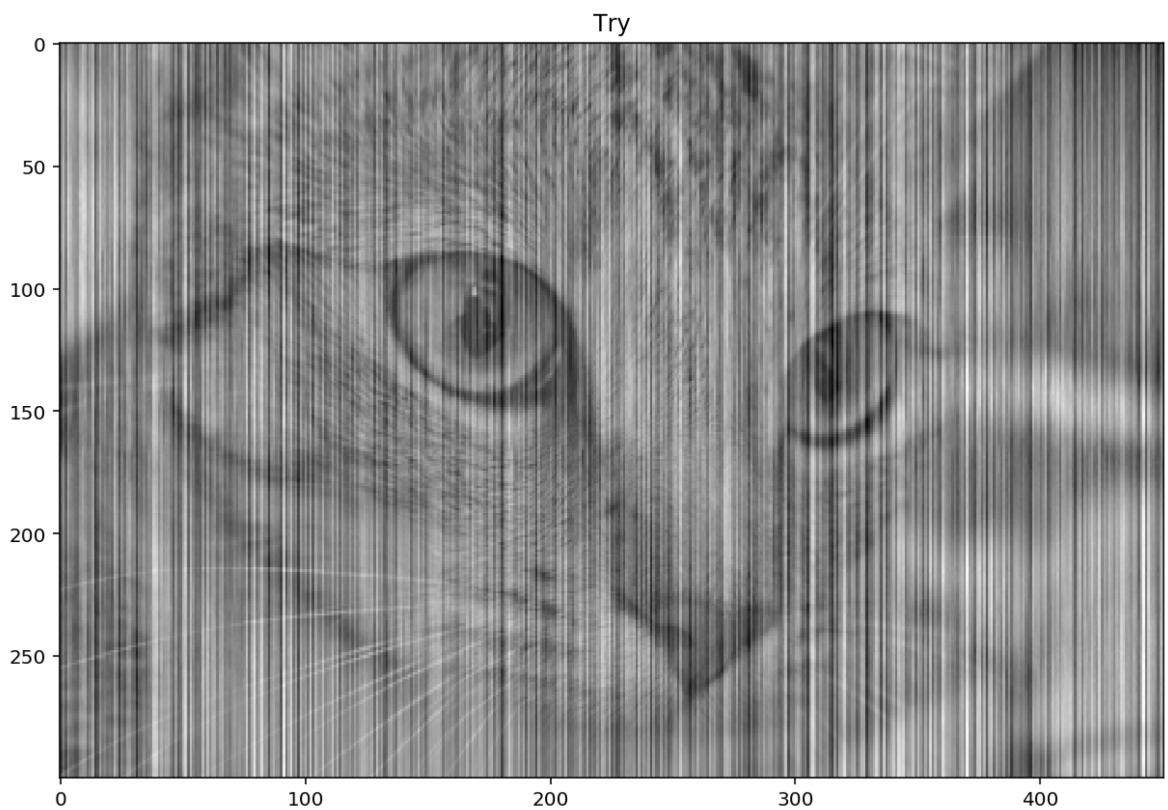
Solutions

1a. You are given a corrupted image that has been transformed by the given matrix H. We have F = G*H, where F is the image you've been given, G is the original image, and H is the corrupting transformation. Use the SVD, H, and F to recover G the best you can.

In [57]:
```
### Don't touch
G = data.chelsea()
G = rgb2gray(cat)
H = numpy.zeros((451, 400))
H = H + numpy.random.normal(loc=0, size=H.shape)
F = G@H

### Your code starts here
U, s, V = numpy.linalg.svd(H)
sigma_plus = 1/s # Taking the reciprocal of all non-zero singular values
sigma_plus = numpy.diag(sigma_plus) # Turning our diagonal into a matrix

# We have to add the extra zeros that were cut off by numpy to get the pr
oper dimension
extra_zeros = numpy.zeros((51, 400))
sigma_plus = numpy.vstack((sigma_plus, extra_zeros))

sigma_plus = sigma_plus.transpose()
H_inv = V.transpose()@sigma_plus@U.transpose()
G_try = F@H_inv

plt.figure()
plt.title("Try")
plt.imshow(G_try, cmap='gray')
```

Out[57]: <matplotlib.image.AxesImage at 0x7f3799cd8978>

Out[57]:

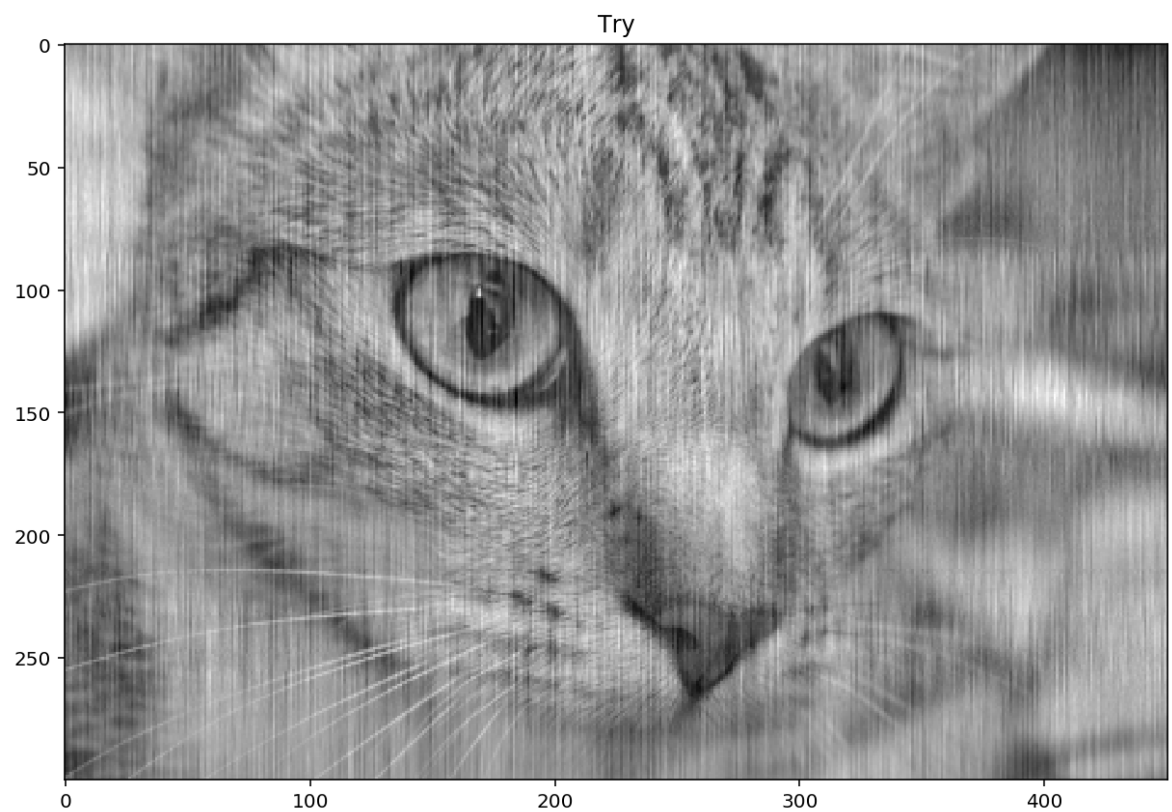


1b. Now use the SVD in a different way to make the recovered image look even better. (Hint: different singular values represent different horizontal/vertical lines in an image)

In [59]:
```
U, s, V = numpy.linalg.svd(G_try)

S = numpy.diag(s)
# We have to add the extra zeros that were cut off by numpy to get the proper dimension
extra_zeros = numpy.zeros((300, 151))
S = numpy.hstack((S, extra_zeros))
S[0][0] = 0

plt.figure()
plt.title("Try")
plt.imshow(U@S@V, cmap='gray')
```

Out[59]: `<matplotlib.image.AxesImage at 0x7f3799c856d8>`

Out[59]:



1. Pretend you have never seen a square before, but you have a set of points representing a square. Write a program that, given the square, guesses an axis of symmetry.

In [104]:
```
### Don't touch
B = [(x, 100) for x in range(100, 201, 5)]
R = [(200, x) for x in range(100, 201, 5)]
T = [(x, 200) for x in range(100, 201, 5)]
L = [(100, x) for x in range(100, 201, 5)]
square = numpy.vstack((B, R, T, L))
square = square.transpose()

### Your code starts here
axes = []
for i in range(0, 360):
    theta = numpy.radians(i)
    c, s = numpy.cos(theta), numpy.sin(theta)
    R = numpy.array(((c, -s), (s, c)))

    new_square = R@square
    new_square = new_square + numpy.random.normal(loc=5, size=square.shap
e)

    A = new_square
    B = square
    M = B@A.transpose()
    U, s, V = numpy.linalg.svd(M)

    O = U@V
    A_to_B = O@A

    d = numpy.sum((A_to_B[0] - B[0])**2 + (A_to_B[1] - B[1])**2)
    d = d**.5
    axes.append((i, d))

axes.sort(key  = lambda x: x[1], reverse = False)
print(axes[0][0], axes[1][0], axes[2][0], axes[3][0], axes[4][0])
```

```
269 93 92 90 96
```

Sources

1. https://en.wikipedia.org/wiki/Singular_value_decomposition (https://en.wikipedia.org/wiki/Singular_value_decomposition)
2. https://en.wikipedia.org/wiki/Low-rank_approximation (https://en.wikipedia.org/wiki/Low-rank_approximation)
3. https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem (https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem)
4. https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse (https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse)
5. https://en.wikipedia.org/wiki/Rank_(linear_algebra (https://en.wikipedia.org/wiki/Rank_(linear_algebra))
6. https://en.wikipedia.org/wiki/Orthogonal_matrix (https://en.wikipedia.org/wiki/Orthogonal_matrix)
7. Kiran Kedlaya's Math 157 lecture for Jan 31, 2020
8. https://scikit-image.org/docs/stable/auto_examples/data/plot_general.html#sphx-glr-auto-examples-data-plot-general-py (https://scikit-image.org/docs/stable/auto_examples/data/plot_general.html#sphx-glr-auto-examples-data-plot-general-py)
9. http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/ (http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/)
10. https://www.autodeskresearch.com/publications/samestats (https://www.autodeskresearch.com/publications/samestats)
11. https://simonensemble.github.io/2018-10-27-orthogonal-procrustes/ (https://simonensemble.github.io/2018-10-27-orthogonal-procrustes/)
12. https://en.wikipedia.org/wiki/Procrustes_analysis (https://en.wikipedia.org/wiki/Procrustes_analysis)
13. https://scipython.com/book/chapter-6-numpy/examples/creating-a-rotation-matrix-in-numpy/ (https://scipython.com/book/chapter-6-numpy/examples/creating-a-rotation-matrix-in-numpy/)

In [0]:

```
```