Galen Vincent

PHGN 311 - Mathematical Physics

Project 3

---

1 - Get the data

a) Import the selfie in color, use Mathematica to convert to grayscale.

```
In[1]:= SetDirectory[NotebookDirectory[]];
```

```
In[2]:= picraw = Import["selfie.jpg"];
```

```
In[3]:= pic = ColorConvert[picraw, "Grayscale"];
```

b) Get the data out of the picture with a simple Mathematica command, asking for values in range 0-255:

```
In[4]:= picData = N[ImageData[pic, "Byte"]];
```

Note that Mathematica takes in values between 0 and 1 in order to plot in grayscale, so I normalize my pixel values by dividing by 255 before I pass them in to be plotted. I do this consistently through the entire project.

Printed by Wolfram Mathematica Student Edition

In[5]:=
```
Image[picData / 255]
```

Out[5]=

In[6]:=
```
Dimensions[picData]
```

Out[6]=
```
{800, 600}
```

My picture is originally 800 by 600 pixels, which means that there are 800 x 600 cells in the data matrix,

Printed by Wolfram Mathematica Student Edition

each with a value between 0 and 255. Above, you can see the code in Mathematica where I converted my image into a matrix called “picData” of these grayscale values.

c) Cut a square out of the picture, by selecting a range of 500 x 500 entries in the data matrix (which corresponds to 500 x 500 pixels of data). I store this data in a new variable, for use later.

```
In[7]:= sqPicData = picData[[251 ;; 750, 101 ;; 600]];

In[8]:= Dimensions[sqPicData]

Out[8]= {500, 500}

In[9]:= Image[sqPicData / 255]
```

Out[9]=

Above, I take a square matrix of pixels out of my big picture by simply choosing a square range of matrix indices from my big matrix. This effectively takes a square cut out of my image.

2 - Analysis Part I: Deconstructing Your Self

a) Perform singular value decomposition (SVD) on your rectangular matrix. Plot the singular values as a function of index, from largest to smallest, on a log scale.

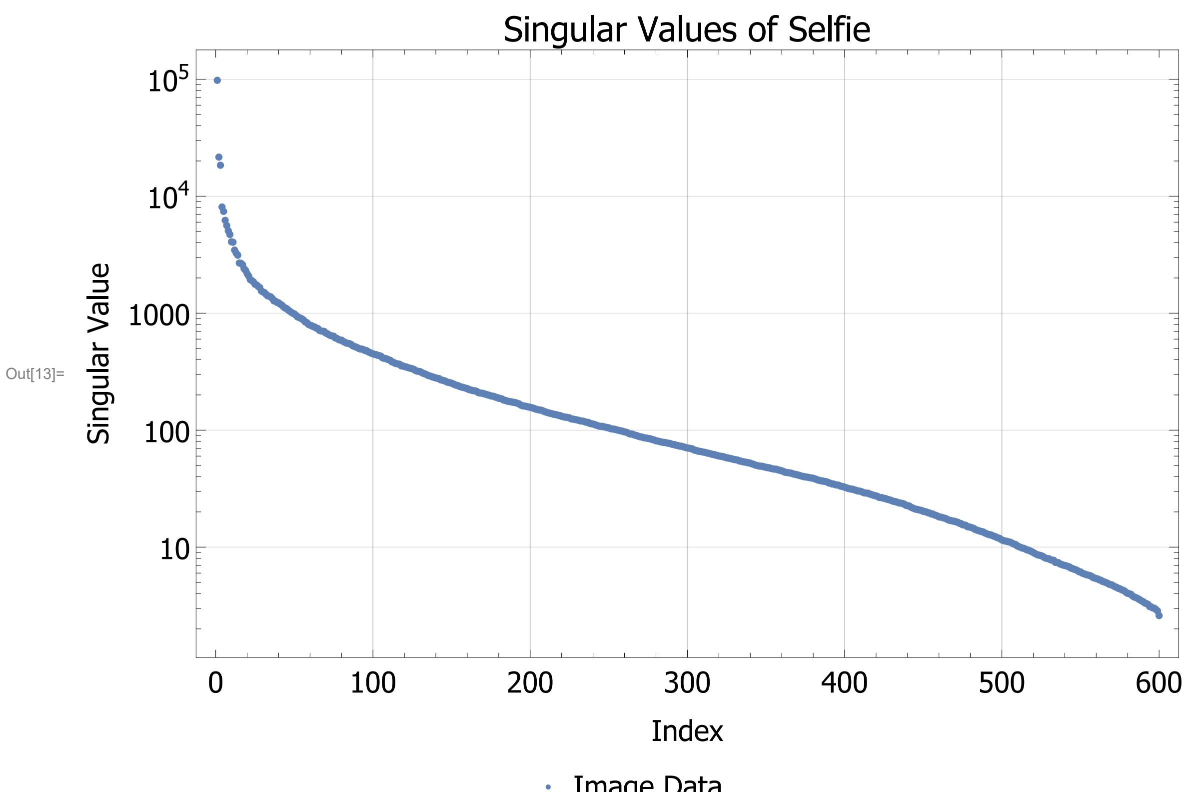
Take the SVD of my image data, broken down into: Data = U S V^T

```
In[10]:= {u, s, v} = SingularValueDecomposition[picData];

In[11]:= Dimensions[s]

Out[11]= {800, 600}
```

Plot the singular values as a function of index from largest to smallest on a log scale.

```
In[12]:= sValues = Diagonal[s];

In[13]:= p1 = ListLogPlot[sValues, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
           Frame → True, PlotLabel → "Singular Values of Selfie",
           FrameLabel → {"Index", "Singular Value"}, GridLines → Automatic,
           ImageSize → Large, PlotLegends → Placed[{"Image Data"}, Below]]
```

Out[13]=



b) Create a random matrix of the same size. Perform SVD and plot the singular values on the same plot as (a).

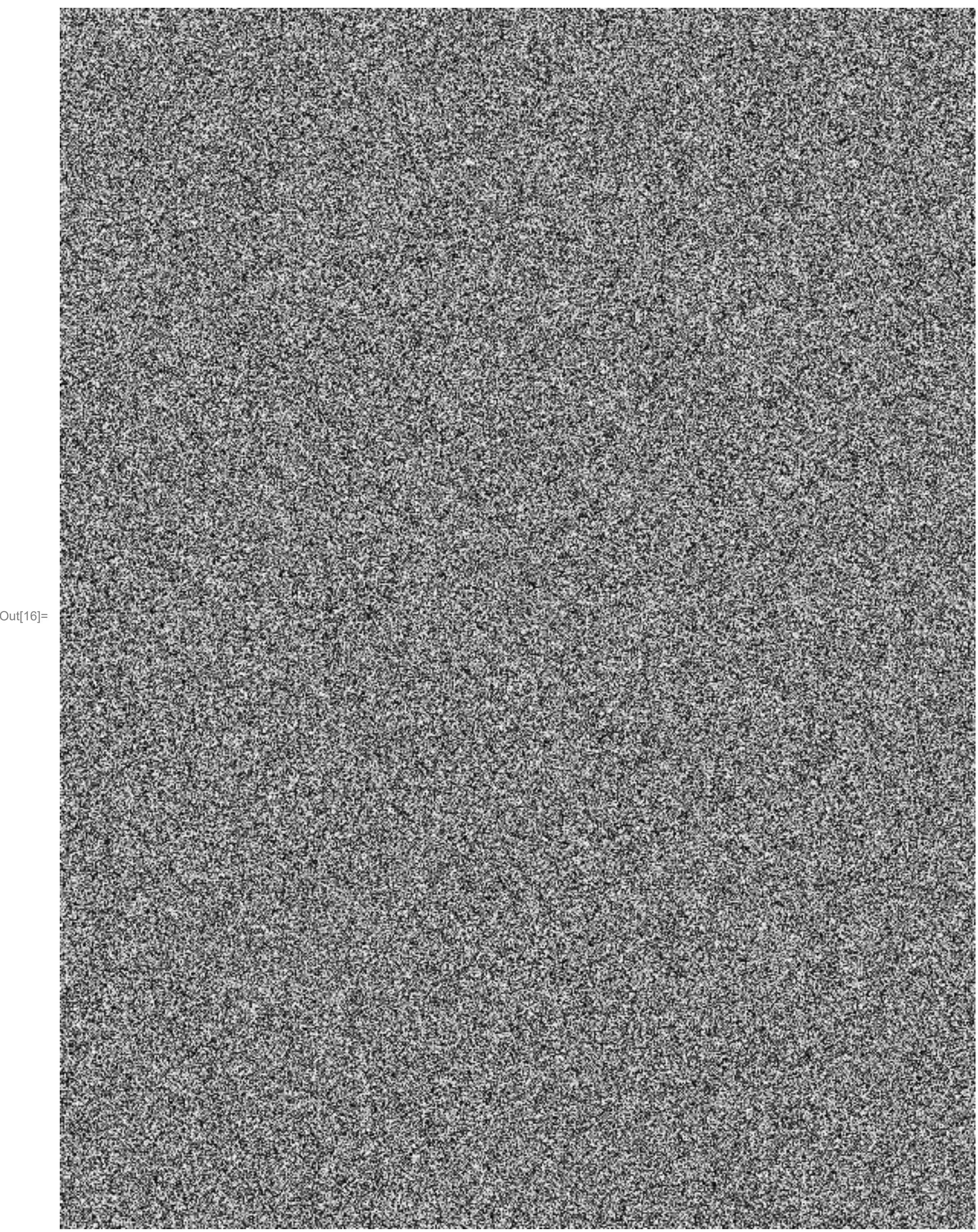
Create a random matrix of the same size as the image above:

```
In[14]:= f[i_, j_] := RandomReal[255];
         random = N[Array[f, {800, 600}]];
```

Here’s what it looks like:

In[16]:=
```
Image[random / 255]
```

Out[16]=



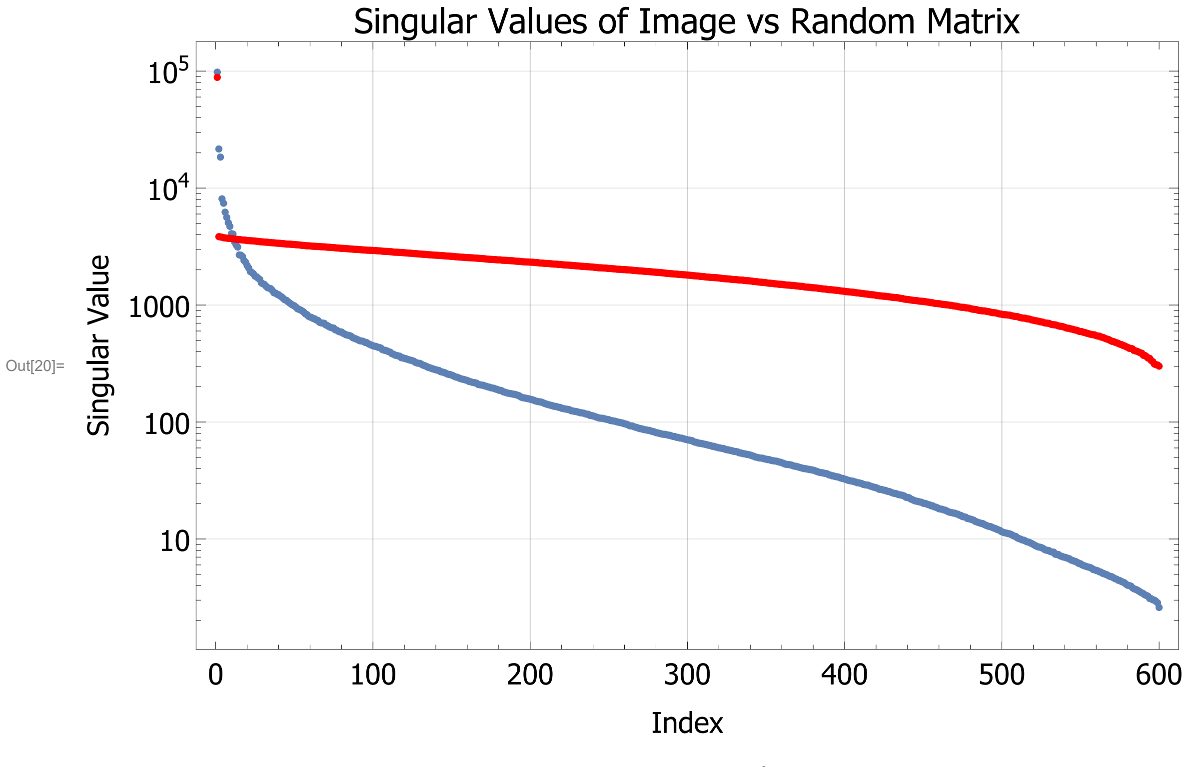
Take the SVD of this random matrix, broken down into: Random = U S V^T

In[17]:=
```
{ur, sr, vr} = SingularValueDecomposition[random];
```

Plot the singular values as a function of index from largest to smallest on the same plot as in part a).

```
In[18]:= srValues = Diagonal[sr];

In[19]:= p2 = ListLogPlot[srValues, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
    Frame → True, PlotLabel → "Singular Values of Random Matrix",
    FrameLabel → {"Index", "Singular Value"}, GridLines → Automatic,
    ImageSize → Large, PlotStyle → Red, PlotLegends → Placed[{"Random"}, Below]];

In[20]:= Show[p1, p2, PlotLabel → "Singular Values of Image vs Random Matrix"]
```

Out[20]=



c) Compare (a) vs. (b). Explain how they're different, based on your plot.

The singular values of the selfie, shown in blue above, are visually significantly different than those of the random matrix values, shown in red. The image singular values are lower than the random matrix values for the large majority of the indices, except for the first 25 or so. Also, the slope of the image values is steeper than that of the random ones. These observations tell us that there is a disproportionately large amount of information stored in the first 25 or so singular values of the image, and that the data contained in the image singular values drop off quickly as you increase the index. For the random image, there is a relatively equal amount of information stored in all of the singular values. What this means is that we can reconstruct the image with fewer singular values than it would take to reconstruct the random matrix.

d) Now create an animation in which you use the singular values to reconstruct your image. That is, take a truncated basis for your image in which you use only χ singular values to describe yourself. Show the resulting image for χ running from 1 to the max number. This can be a small movie or a slider bar in Mathematica. Either way explain how many singular values are required before you can make out your identity.

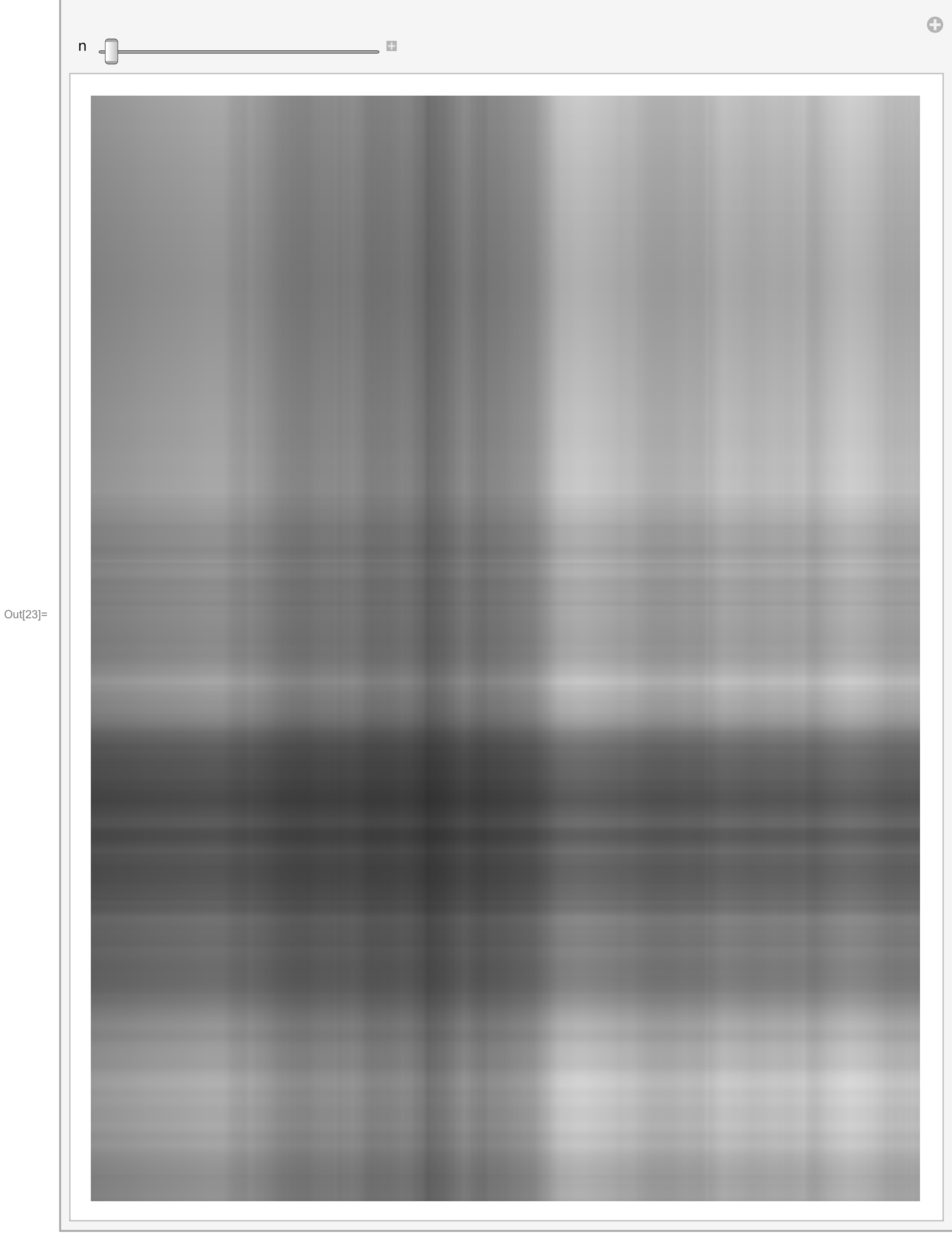
Create a matrix of the proper size which contains χ singular values along the diagonal, and zeros everywhere else.

In[21]:=
```
extraZeros = ConstantArray[0, {200, 600}];
svFunc[n_] := Join[DiagonalMatrix[sValues[[1 ;; n]], 0, 600], extraZeros];
```

Create the slide bar to see what the reconstruction looks like with n singular values.

In[23]:=
```
Manipulate[Image[u.svFunc[n].Transpose[v] / 255], {n, 1, 600, 2}]
```

Out[23]=



It takes about 25 singular values before I can tell who I am, and about 100 singular values before there is a very crisp image.

3 - Analysis Part 2: Self-Perspective

a) For the rest of this project, we'll work with the zoom, your square matrix showing your face. Make sure your matrix is diagonalizable. If it's not, you either work with singular values in all the below problems, or else take steps to make it diagonalizable by subtracting out the non-diagonal-izable part. Decide which is the best approach and make a coherent argument for it.

Here, I check if my matrix is diagonalizable. This is true if and only if there are n linearly independent eigenvectors for the matrix. I check this below.

```
In[24]:= eVecs = Chop[Eigenvectors[sqPicData]];

In[25]:= MatrixRank[eVecs]

Out[25]= 500
```

Because my matrix of eigenvectors is 500 x 500, and the rank of said matrix is also 500, I know that each of the eigenvectors is linearly independent of one another. This means that indeed, my square matrix is diagonalizable. There happen to be complex eigenvalues and eigenvectors, but that's totally acceptable. When looking at the size of the eigenvalues, I can just look at the magnitudes of them in complex space.

Now, I calculate the diagonalized matrix D, so that A = P D P^{-1}, where P are matrices that have the eigenvectors as columns. D will have the eigenvalues along the diagonal. Note that I have to take the transpose of the eigenvector matrix in Mathematica, as the eigenvectors are in the rows for that matrix, and we need them to be in the columns.

```
In[26]:= diag = Chop[Inverse[Transpose[eVecs]].sqPicData.Transpose[eVecs]];
```

Now I can check that the diagonalization was successful by re-creating my original matrix and plotting it. I take the real part of the re-creating because it produces a bunch of really small imaginary components (ignorable) that mess with Mathematica, so I just get rid of those:

In[27]:=
```
Image[Re[Transpose[eVecs].diag.Inverse[Transpose[eVecs]]] / 255]
```

Out[27]=

And indeed, we see that I can successfully diagonalize my image data.

b) Design an interesting unitary transformation. Perform the unitary transform on your face, and explain the result.

I designed a unitary transform that flips my image upside down. I do this by using the unitary matrix with ones on the “opposite diagonal”. Here is a scaled down example of what I mean:

In[28]:=
```
mat = {{0, 0, 1}, {0, 1, 0}, {1, 0, 0}};
MatrixForm[mat]
```

Out[29]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

I can check that these kinds of matrices are indeed unitary by checking that $U^{\dagger}U = UU^{\dagger}$ = Identity Matrix:

In[30]:=
```
mat.ConjugateTranspose[mat] // MatrixForm
ConjugateTranspose[mat].mat // MatrixForm
```

Out[30]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Out[31]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

And we see that indeed, the matrix is unitary. Now, I will create a 500 x 500 matrix of the same pattern, and apply it to my selfie.

In[32]:=
```
transform = Reverse[IdentityMatrix[500]];
```

Now, I apply it to my selfie:

In[33]:=
```
selfieTrans = transform.sqPicData;
```

In[34]:=
```
Image[selfieTrans / 255]
```

Out[34]=

Look at that! Its a nice transformation.

c) Design a non-unitary matrix that stretches, compresses, or otherwise distorts your face.

Explain your design, do the matrix multiplication, and show the result.

I designed a non-unitary matrix that stretches my image in the vertical direction. It does so by creating an extra row of pixels in between each of the existing rows. In these new rows, I designed my matrix to place the average of the pixel directly above it and directly below it, so that there is a smooth transition between pixels. Because I am creating an image that is larger than my original, I need to create more information, and taking the averages is how I do that. Here is a small example of what the matrix looks like, if it were going to be applied to a 5pixel x 5pixel image:

```
In[35]:= example  = Table[Which[ j == 2 i - 1 , 1,
     EvenQ[j] && (i == j/2 || i == j/2 + 1), .5, True, 0], {j, 1, 9}, {i, 1, 5}];
example // MatrixForm
```

Out[36]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

I can check that this matrix is indeed not unitary:

```
In[37]:= example.ConjugateTranspose[example] // MatrixForm
ConjugateTranspose[example].example // MatrixForm
```

Out[37]//MatrixForm=

$$\begin{pmatrix} 1. & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0.5 & 0.5 & 0.5 & 0.25 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0.5 & 1. & 0.5 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0.25 & 0.5 & 0.5 & 0.5 & 0.25 & 0. & 0. & 0. \\ 0. & 0. & 0. & 0.5 & 1. & 0.5 & 0. & 0. & 0. \\ 0. & 0. & 0. & 0.25 & 0.5 & 0.5 & 0.5 & 0.25 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0.5 & 1. & 0.5 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0.25 & 0.5 & 0.5 & 0.5 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.5 & 1. \end{pmatrix}$$

Out[38]//MatrixForm=

$$\begin{pmatrix} 1.25 & 0.25 & 0. & 0. & 0. \\ 0.25 & 1.5 & 0.25 & 0. & 0. \\ 0. & 0.25 & 1.5 & 0.25 & 0. \\ 0. & 0. & 0.25 & 1.5 & 0.25 \\ 0. & 0. & 0. & 0.25 & 1.25 \end{pmatrix}$$

We see that $U^{\dagger} U$ and $UU^{\dagger}$ are not equal to the identity matrix, and therefor our matrix is not unitary.

Below I create this matrix transformation for my 500 x 500 pixel selfie, and apply the matrix to my image.

```
In[39]:= transform2 = Table[Which[j == 2 i - 1 , 1, EvenQ[j] && (i == j/2 || i == j/2 + 1), .5, True, 0],
    {j, 1, 999}, {i, 1, 500}];
```

```
In[40]:= selfieTrans2 = transform2.sqPicData;
```

In[41]:=
```
Image[selfieTrans2 / 255]
```

Out[41]=

Wow, look at that! That was pretty dang fun to come up with and implement, I have to say. I'm glad that it worked so well.

4 - Analysis Part 3: Chaos and Randomness

a) Is your face Hermitian, unitary, or something else? Examine the properties of your face matrix and say as much as you can about it.

First, I can check if the matrix is Hermitian. A Hermitian matrix is defined as one that satisfies the property: $H = H^\dagger$, where $H^\dagger$ is the conjugate transpose of H.

```
In[42]:= sqPicData == Conjugate[Transpose[sqPicData]]

Out[42]= False
```

So we see that my face is not Hermitian. Next, we can check if it is unitary. This is done by checking to see if $UU^\dagger = U^\dagger U$ = identity matrix.

```
In[43]:= sqPicData.ConjugateTranspose[sqPicData] == ConjugateTranspose[sqPicData].sqPicData

Out[43]= False
```

Right away, we see that $UU^\dagger$ is not equal to $U^\dagger U$, so my face is therefore not unitary either.

One thing that I can say about my face is that it is diagonalizable, which means that there are 500 linearly independent eigenvectors associated with it (as it is a 500 x 500 matrix of data). We found this out in Part 3a.

b) Diagonalize your face. Are there any degeneracies or near-degeneracies in your image? Make a plot of your eigenvalues λ from largest to smallest. Explain the character of the eigenvectors associated with the largest eigenvalues. Why are they important to reconstructing your self-image?

I found the diagonal matrix, stored in a variable called "diag", in part 3a. The diagonal entries of this matrix are the eigenvalues. Below, I check for degeneracies (eigenvalues which are the same) by first sorting the eigenvalues from largest to smallest, and then looking at the difference between each of the values. Note that there are complex eigenvalues, so I will look at the absolute value of them in order to compare them. For complex eigenvalues, there is always an associated complex conjugate, which doesn't count as a degeneracy, so I will skip over looking at how those magnitudes compare with each other.

First, I get rid of all of the complex conjugates:

In[44]:=
```
complexDiags = Diagonal[diag];
complexDiagsNoRepeats = {};

For[n = 1, n ≤ Length[complexDiags], n++,
 If[Re[complexDiags[[n]]] == complexDiags[[n]],
  AppendTo[complexDiagsNoRepeats, complexDiags[[n]]],
  {AppendTo[complexDiagsNoRepeats, complexDiags[[n]]],
   n++}
 ]
]
```

Then, I sort the absolute values of the remaining eigenvalues by size, and look at the differences between the neighboring values for the smallest difference.

In[47]:=
```
eigenVals = Sort[Abs[complexDiagsNoRepeats], Greater];
```

In[48]:=
```
minDiff = eigenVals[[1]];
minDiffi = 0;
For[i = 1, i ≤ (Length[eigenVals] - 1), i++,
 If[(Abs[ eigenVals[[i]] - eigenVals[[i + 1]]] < minDiff),
  {minDiff = Abs[eigenVals[[i]] - eigenVals[[i + 1]]],
   minDiffi = i;}]
]
```

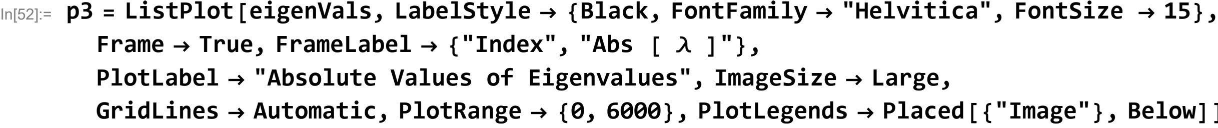
In[51]:=
```
minDiff
```

Out[51]=
```
0.00589064
```

With this information, I can say that there are no degeneracies in my eigenvalues to within 0.006.

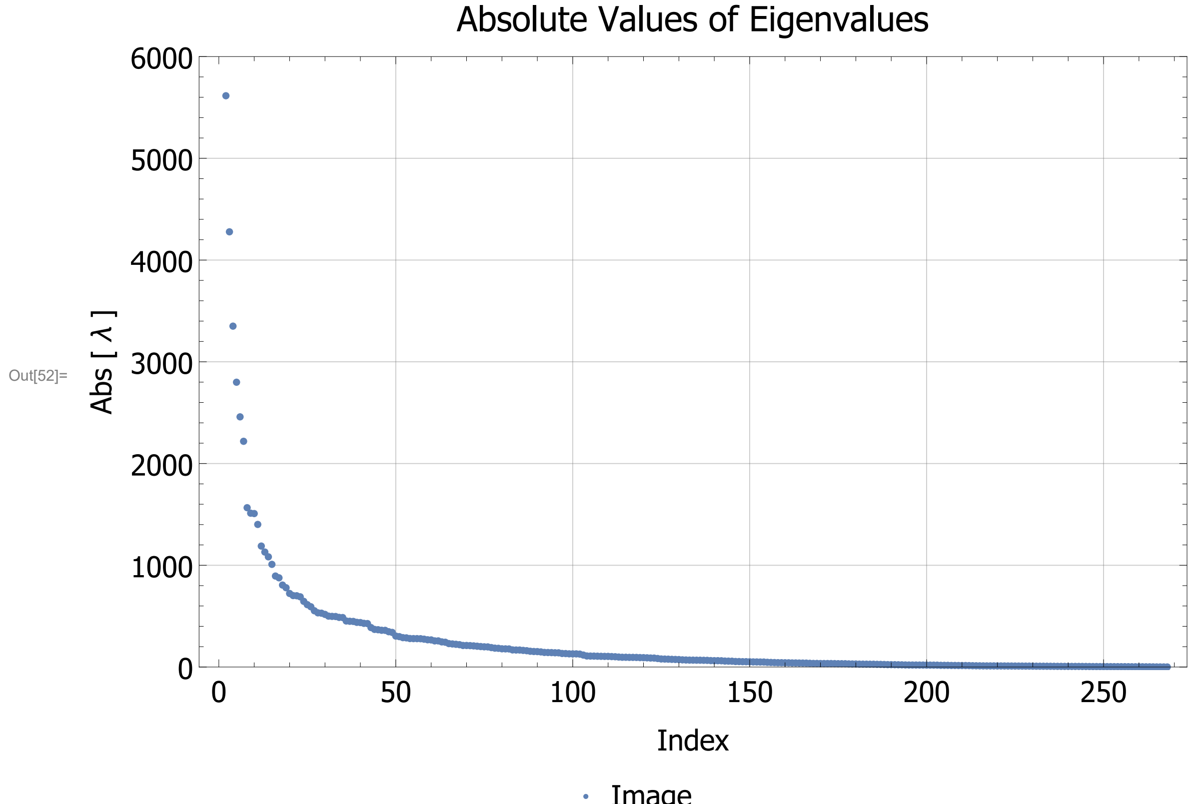
Next, I plot these absolute values of eigenvalues:

In[52]:=



```
p3 = ListPlot[eigenVals, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
  Frame → True, FrameLabel → {"Index", "Abs [ λ ]"},
  PlotLabel → "Absolute Values of Eigenvalues", ImageSize → Large,
  GridLines → Automatic, PlotRange → {0, 6000}, PlotLegends → Placed[{"Image"}, Below]]
```

Out[52]=



Note that there is one eigenvalue of value 60,000 which I do not show on the plot, just so it can have better resolution for all of the lower ones.

In[53]:=

```
eigenVals[[1]]
```

Out[53]=

```
60 085.7
```

The eigenvectors associated with the biggest eigenvalues contain the most information about the image, just like in the singular value decomposition. This is evident as the first few eigenvalues are so much larger than the rest. Because of this, they are the most important vectors in re-creating the image, as they contribute the most information back to the image matrix.

c) Create a random square matrix. Plot the eigenvalues from largest to smallest, and show on the same plot as (b). Compare this result to that obtained from your face. Is your face random? Random matrix theory is a well-known way to identify what we call Hamiltonian, or closed system chaos for many degrees of freedom.

Below, I create a random square matrix of size 500 x 500:

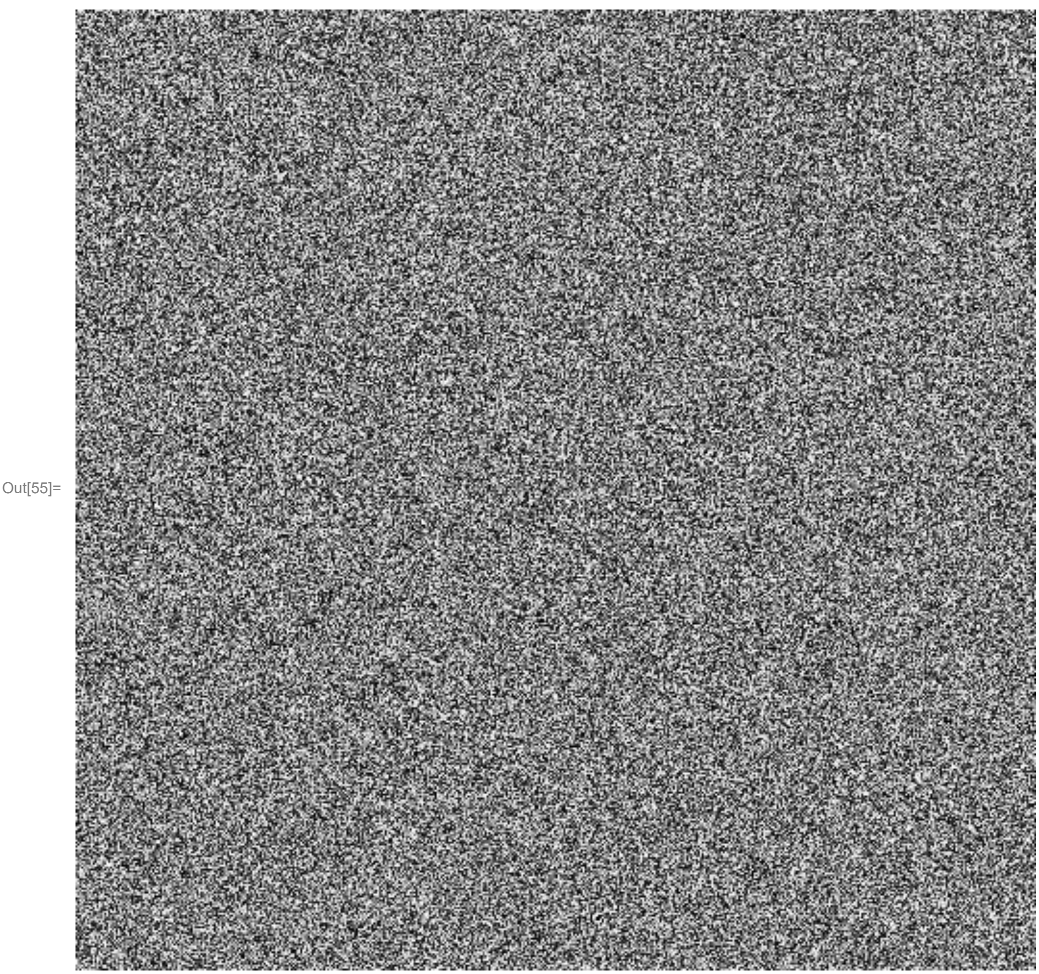
In[54]:=

```
randomSq = N[Array[f, {500, 500}]];
```

Here’s what it looks like:

In[55]:=
```
Image[randomSq / 255]
```

Out[55]=



Now I’ll find and plot the eigenvalues from this random matrix on top of the plot from part b), after getting rid of the complex conjugate repeats, in the same way that I did for part b).

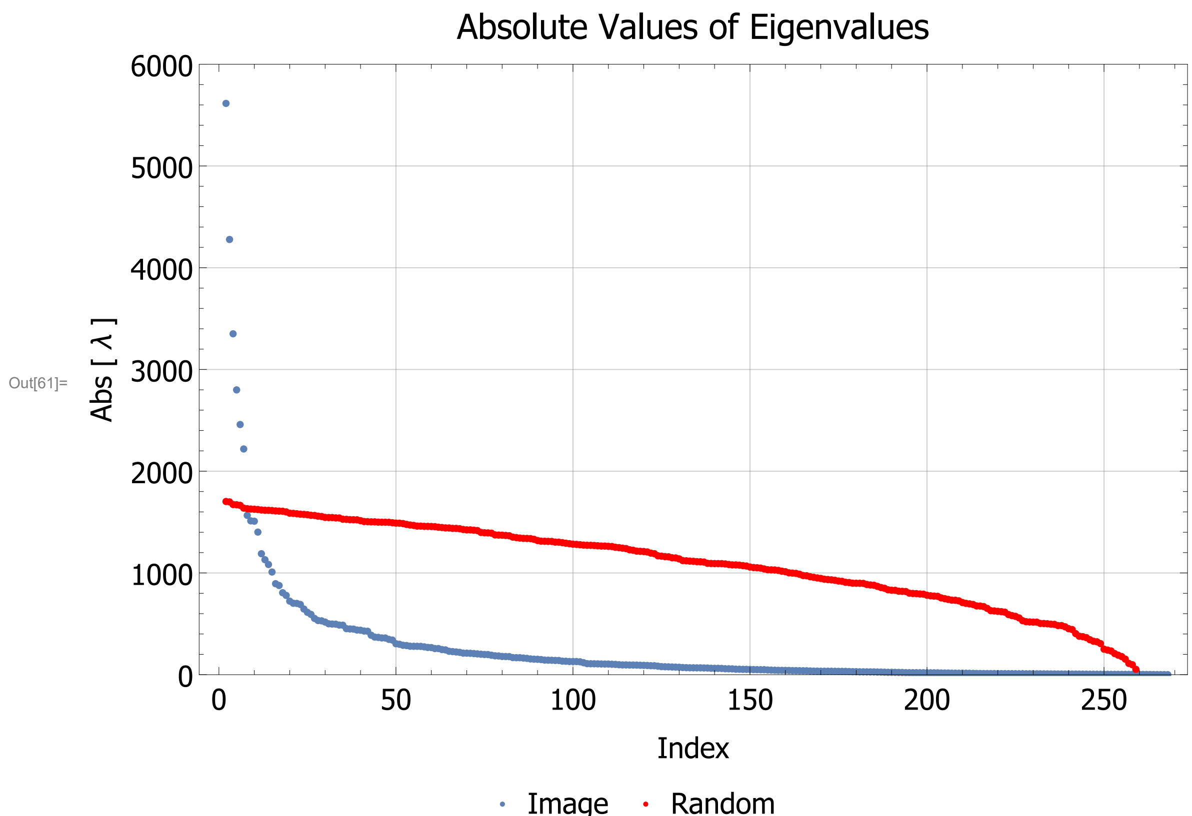
In[56]:=
```
randomEigenVals = Eigenvalues[randomSq];
```

In[57]:=
```
rcomplexDiagsNoRepeats = {};

For[n = 1, n ≤ Length[randomEigenVals], n++,
 If[Re[randomEigenVals[[n]]] == randomEigenVals[[n]],
  AppendTo[rcomplexDiagsNoRepeats, randomEigenVals[[n]]],
  {AppendTo[rcomplexDiagsNoRepeats, randomEigenVals[[n]]],
   n++}
 ]
]
```

In[59]:=
```
reigenVals = Sort[Abs[rcomplexDiagsNoRepeats], Greater];
```

In[60]:=
```
p4 = ListPlot[reigenVals, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
   Frame → True, FrameLabel → {"Index", "Abs [ λ ]"},
   PlotLabel → "Absolute Values of Eigenvalues", ImageSize → Large,
   GridLines → Automatic, PlotStyle → Red, PlotLegends → Placed[{"Random"}, Below]];
```

In[61]:=
```
Show[p3, p4]
```

Out[61]=


I cut the range off in this plot, once again, to not show the first eigenvalue, so that there is some resolution on the y axis. It is evident from this plot that my face is not random, as the eigenvalues of my face show a very different pattern from the random matrix. Just as with the singular values, the first few eigenvalues contain much more information about my face than they do for the random matrix.

d) Plot a histogram for the differences in successive eigenvalues, $\Delta\lambda = \lambda_{i+1} - \lambda_i$, for (b) and (c) and bin the result appropriately. Do your image and your random matrix display Poisson or Wigner-Dyson statistics? Make a fit and distinguish between the two. Below I provide the general expressions, which you may have to normalize or otherwise slightly alter via a rescaling of $\Delta\lambda$. Keep in mind these are probabilities, and the total probability is always 1. Note in the below expression rescaling by the mean value of $\Delta\lambda$.

First, I find the differences for the image data and the random matrix:

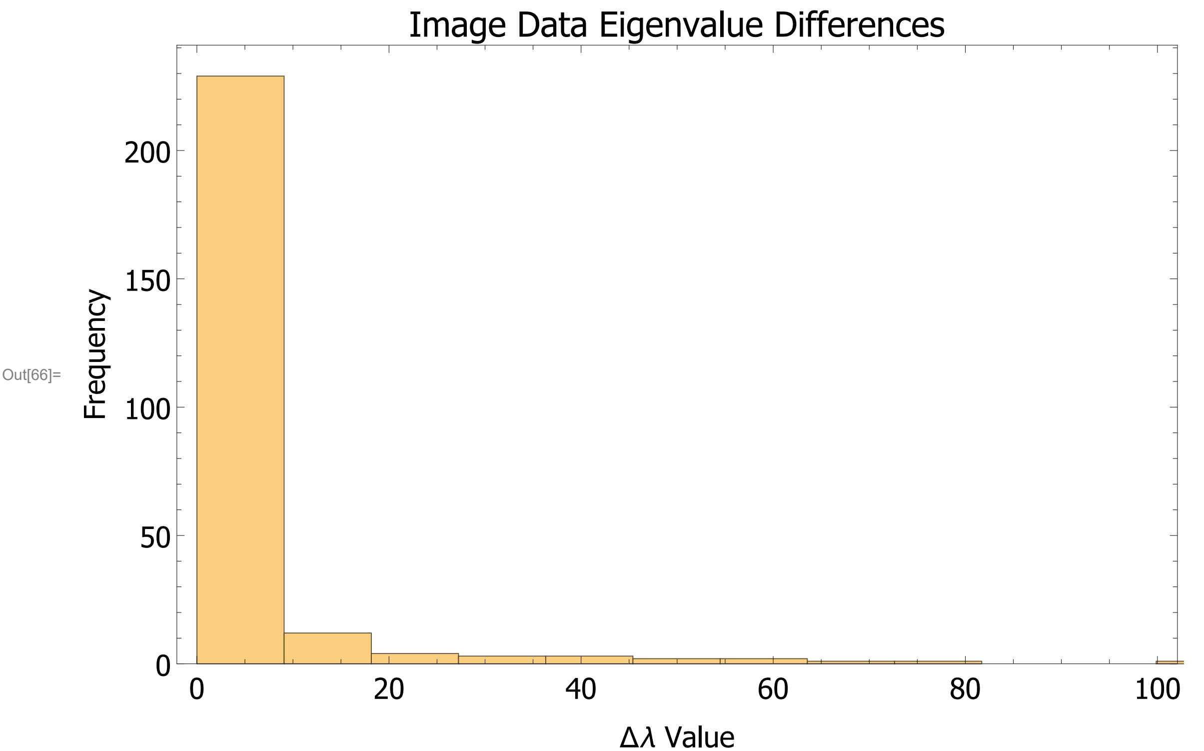
In[62]:=
```
λdiffImage = {};
For[n = 1, n ≤ Length[eigenVals] - 1, n++,
 AppendTo[λdiffImage, Abs[eigenVals[[n]] - eigenVals[[n + 1]]]]]
λdiffRand = {};
For[n = 1, n ≤ Length[reigenVals] - 1, n++,
 AppendTo[λdiffRand, Abs[reigenVals[[n]] - reigenVals[[n + 1]]]]]
```

Now, I plot histograms for each of these data sets. Note that I didn't plot all of the bins for each of the histograms, as the range on the x axis goes out so far that resolution of the bins would be a problem. Just keep in mind that there are a few bins in both images to the right of what is displayed, but not very

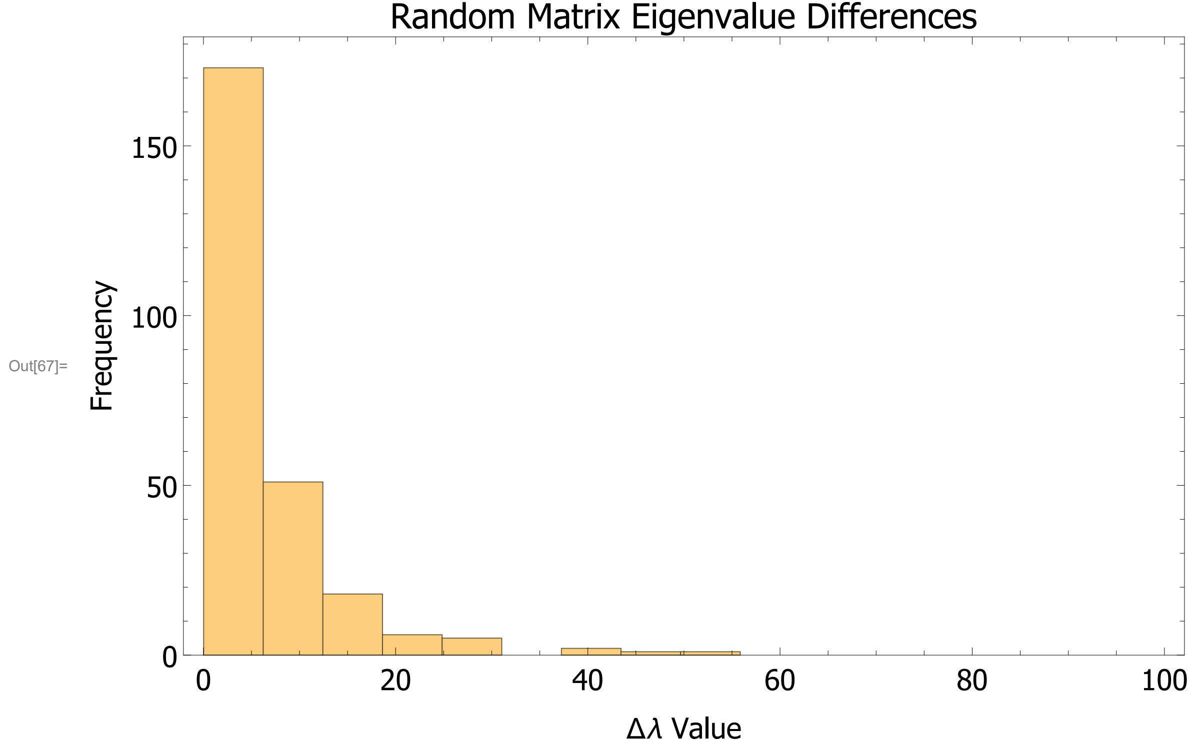
many.

In[66]:=
```
Histogram[λdiffImage, {Min[λdiffImage], Max[λdiffImage] + .00001,
   1/6000 (Max[λdiffImage] - Min[λdiffImage] + .00001)}, PlotRange → {{0, 100}, Automatic},
  Frame → True, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
  FrameLabel → {"Δλ Value", "Frequency"},
  PlotLabel → "Image Data Eigenvalue Differences", ImageSize → Large]
Histogram[λdiffRand , {Min[λdiffRand ], Max[λdiffRand ] + .00001,
   1/10000 (Max[λdiffRand ] - Min[λdiffRand ] + .00001)}, PlotRange → {{0, 100}, Automatic},
  Frame → True, LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
  FrameLabel → {"Δλ Value", "Frequency"},
  PlotLabel → "Random Matrix Eigenvalue Differences", ImageSize → Large]
```

Out[66]=



Out[67]=



First, I'll work with the image data.

First, I find the un-normalized Poisson distribution. Note that when I found the mean of $\Delta\lambda$, I ignored the first λ difference, as it is so large that it heavily skews the mean and makes the distribution really bad.

In[68]:=
```
μλImage = Mean[λdiffImage[[2 ;; Length[λdiffImage]]]];
```

In[69]:=
```
poisson[dλ_] = Exp[-dλ / μλImage]
```

Out[69]= $e^{-0.0473774\, d\lambda}$

Find the normalization constant for the distribution that makes the integral from 0 to ∞ = 1.

In[70]:=
```
norm = Integrate[poisson[x], {x, 0, ∞}]
```

Out[70]= 21.1071

In[71]:=
```
npoisson[dλ_] = poisson[dλ] / norm
```

Out[71]= $0.0473774\, e^{-0.0473774\, d\lambda}$

Next, I find the un-normalized Wigner-Dyson distribution, doing the same exclusion in the mean as mentioned above.

In[72]:=
$$wd[d\lambda_] = \frac{1}{2}\pi\,(d\lambda / \mu\lambda Image)\ Exp\left[\frac{-\pi\,(d\lambda / \mu\lambda Image)^2}{4}\right]$$

Out[72]= $0.0744202\, d\lambda\, e^{-0.00176292\, d\lambda^2}$

Find the normalization constant for the distribution that makes the integral from 0 to ∞ = 1.

In[73]:=
```
norm1 = Integrate[wd[x], {x, 0, ∞}]
```

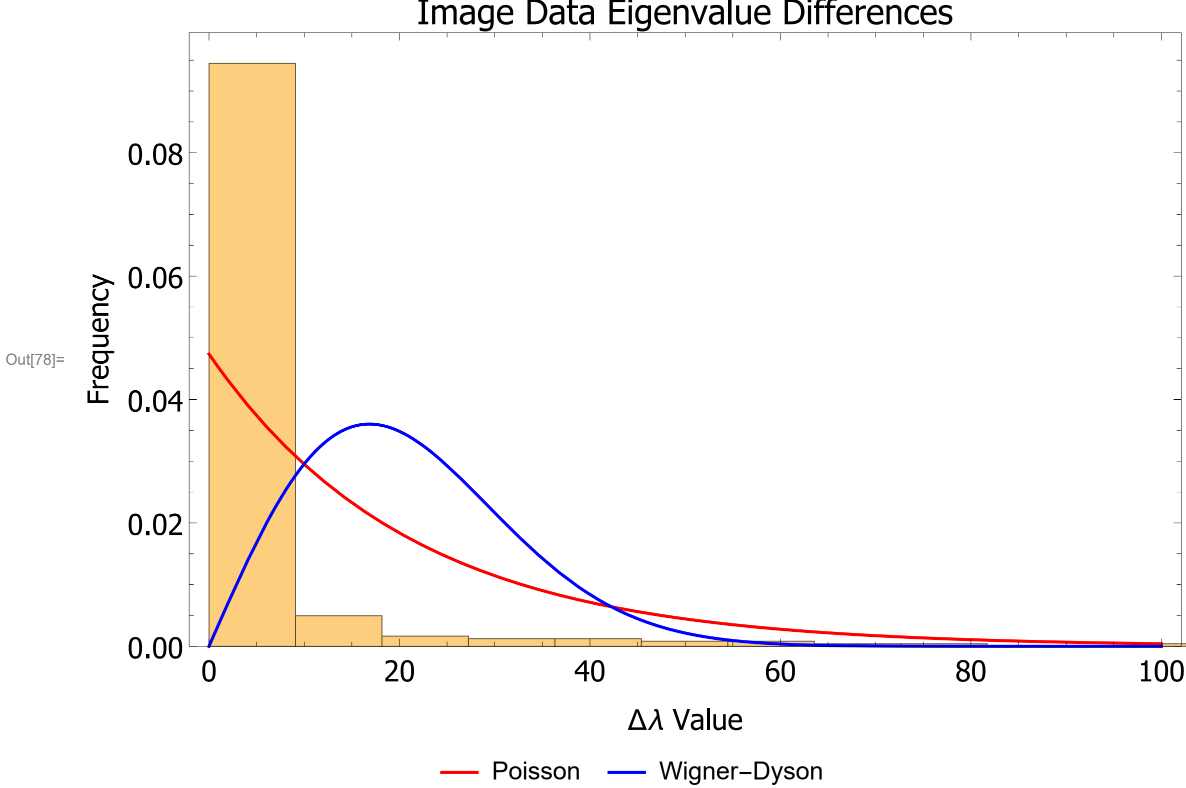
Out[73]=
```
21.1071
```

In[74]:=
```
nwd[dλ_] = wd[dλ] / norm1;
```

Plot the distributions on top of the normalized histogram of the image data.

In[75]:=
```
p5 = Histogram[λdiffImage, {Min[λdiffImage],
    Max[λdiffImage] + .00001, 1/6000 (Max[λdiffImage] - Min[λdiffImage] + .00001)},
   "PDF", PlotRange → {{0, 100}, Automatic}, Frame → True,
   LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
   FrameLabel → {"Δλ Value", "Frequency"},
   PlotLabel → "Image Data Eigenvalue Differences", ImageSize → Large];
p6 = Plot[npoisson[x], {x, 0, 100}, PlotStyle → Red,
   PlotLegends → Placed[{"Poisson"}, Below]];
p7 = Plot[nwd[x], {x, 0, 100}, PlotStyle → Blue,
   PlotLegends → Placed[{"Wigner-Dyson"}, Below]];
Show[
 p5,
 p6,
 p7]
```

Out[78]=



The fits here are really bad, as a result of the fact that there are many difference values that are very large, as a result of the first few eigenvalues, which are enormous, and drop off very quickly. This really messes with this distribution, as it pulls the mean way to the right, and forces the distribution to be much

more broad. Theoretically, however, I think that a structured image may display more Wigner-Dyson statistics. I say this because the first few eigenvalues contain more information for a structured image, which means that there will be more larger $\Delta\lambda$ than that for a random image, which has mostly all the same size drops. For the structured image, the greater number of bigger drops would create a distribution (theoretically) that would look like Wigner-Dyson. I'm not quite sure why I'm not seeing this for my image, though.

Next, I'll work with the random data.

First, I find the un-normalized Poisson distribution. Note that when I found the mean, I ignored the first λ difference, as it is so large that it heavily skews the mean.

```
In[79]:= μλrand = Mean[λdiffRand[[2 ;; Length[λdiffRand]]]];
```

```
In[80]:= poisson2[dλ_] = Exp[-dλ / μλrand]
```

Out[80]= $e^{-0.155792\, d\lambda}$

Find the normalization constant for the distribution that makes the integral from 0 to ∞ = 1.

```
In[81]:= norm2 = Integrate[poisson2[x], {x, 0, ∞}]
```

Out[81]= 6.41883

```
In[82]:= npoisson2[dλ_] = poisson2[dλ] / norm2
```

Out[82]= $0.155792\, e^{-0.155792\, d\lambda}$

Next, I find the un-normalized Wigner-Dyson distribution, doing the same exclusion in the mean as mentioned above.

In[83]:= $\text{wd2}[d\lambda_] = \frac{1}{2}\pi\,(d\lambda / \mu\lambda\text{rand})\ \text{Exp}\left[\frac{-\pi\,(d\lambda / \mu\lambda\text{rand})^2}{4}\right]$

Out[83]= $0.244717\, d\lambda\, e^{-0.0190624\, d\lambda^2}$

Find the normalization constant for the distribution that makes the integral from 0 to ∞ = 1.

```
In[84]:= norm3 = Integrate[wd2[x], {x, 0, ∞}]
```

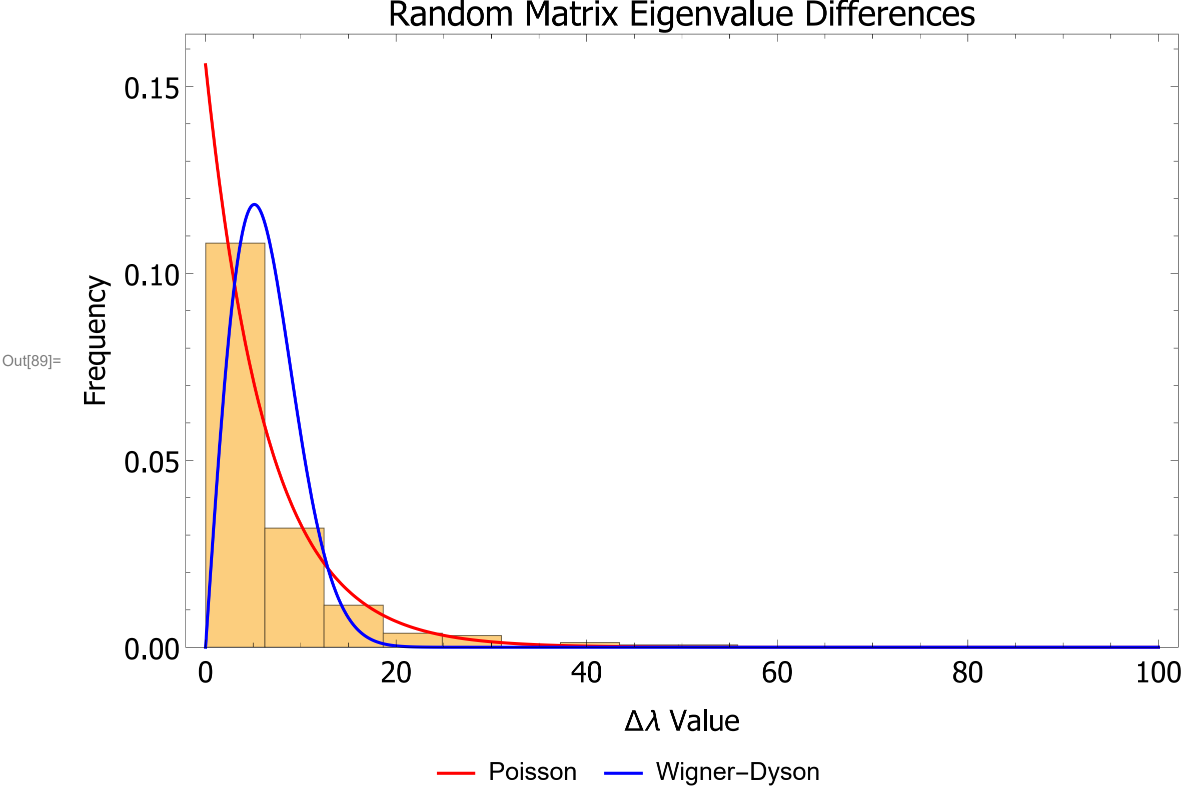
Out[84]= 6.41883

```
In[85]:= nwd2[dλ_] = wd2[dλ] / norm3;
```

Plot the distributions on top of the normalized histogram of the image data.

In[86]:=
```
p8 = Histogram[λdiffRand, {Min[λdiffRand],
    Max[λdiffRand] + .00001, 1/10000 (Max[λdiffRand] - Min[λdiffRand] + .00001)},
   "PDF", PlotRange → {{0, 100}, Automatic}, Frame → True,
   LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
   FrameLabel → {"Δλ Value", "Frequency"},
   PlotLabel → "Random Matrix Eigenvalue Differences", ImageSize → Large];
p9 = Plot[npoisson2[x], {x, 0, 100}, PlotStyle → Red,
   PlotLegends → Placed[{"Poisson"}, Below], PlotRange → All];
p10 = Plot[nwd2[x], {x, 0, 100}, PlotStyle → Blue,
   PlotLegends → Placed[{"Wigner-Dyson"}, Below], PlotRange → All];
Show[
 p8,
 p9,
 p10]
```

Out[89]=



Here, the distributions actually make more sense, as the random matrix eigenvalues don’t start off huge and drop off quickly, they begin at a medium value and slowly drop off, which makes the distributions much easier to fit. I would say that based on the binning shown, the differences display nearly perfect Poisson distribution, as I predicted in the previous analysis.

e) Use the Brody distribution to quantify exactly how close your face is to a random matrix. The Brody parameter q allows one to interpolate between Poisson (q = 0) and Wigner- Dyson (q = 1) limits.

In[90]:=
```
brodyImage[dλ_, q_] =
  (Gamma[(2 + q)/(1 + q)])^(q+1) (1 + q) (dλ/μλImage)^q Exp[-(Gamma[(2 + q)/(1 + q)])^(q+1) (dλ/μλImage)^(q+1)]
```

Out[90]= $0.0473774^{q}\, d\lambda^{q}\, e^{-0.0473774^{1+q}\, d\lambda^{1+q}\, \text{Gamma}\left[\frac{2+q}{1+q}\right]^{1+q}} (1+q)\, \text{Gamma}\left[\frac{2+q}{1+q}\right]^{1+q}$

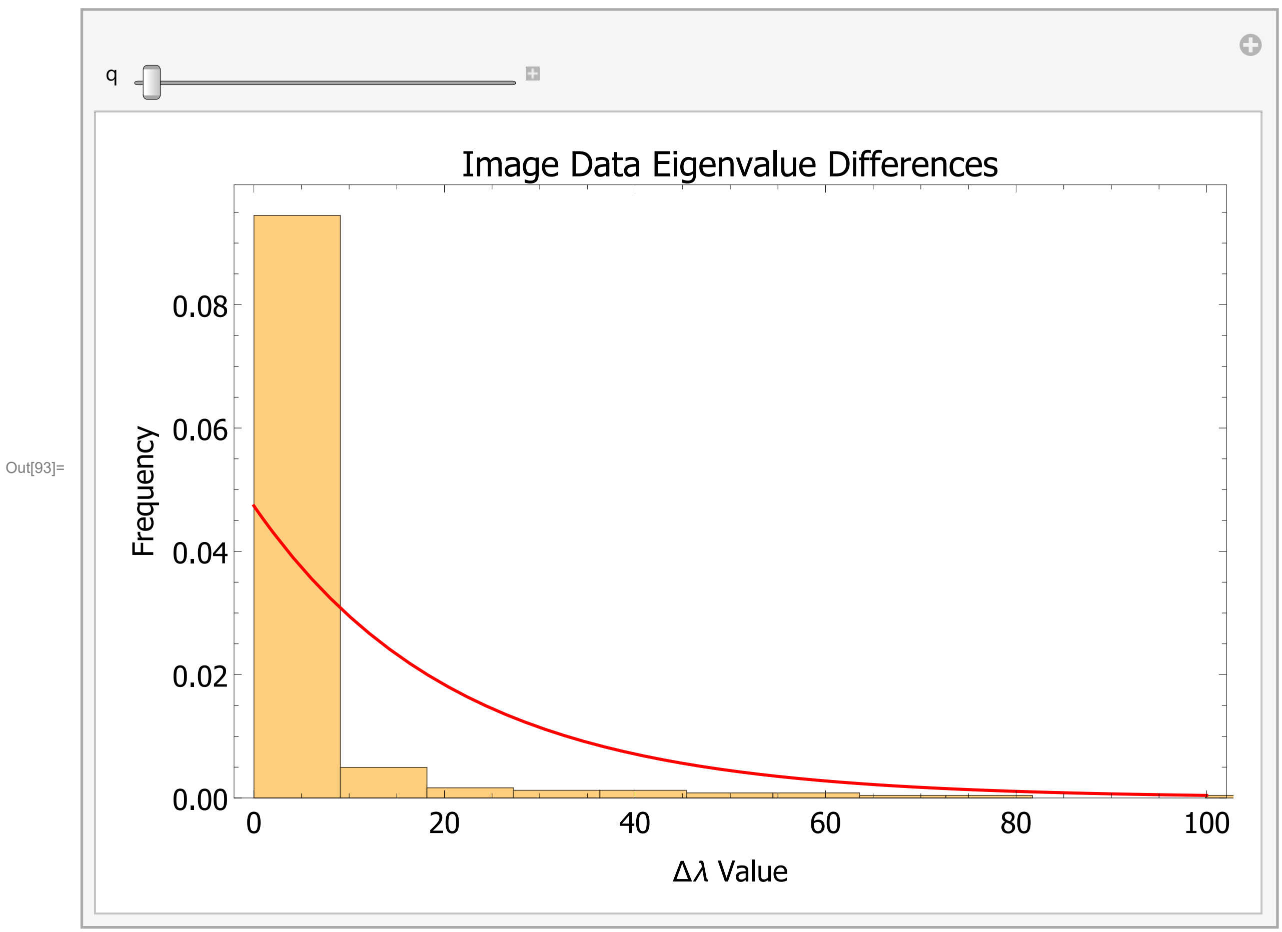
In[91]:=
```
norm4 = Integrate[brodyImage[x, 3], {x, 0, ∞}]
```

Out[91]= 21.1071

In[92]:=
```
nbrodyImage[dλ_, q_] = brodyImage[dλ, q] / norm4;
```

In[93]:=
```
Manipulate[Show[p5, Plot[nbrodyImage[x, q], {x, 0, 100}, PlotStyle → Red]], {q, 0, 1}]
```

Out[93]=



If I say that the random matrix is associated with completely Poisson statistics, or a q value of 0, then I can say with confidence that my image is not random. I say this becasue while playing around with the slide bar above, I see that the the best possible fit for my data occurs at q values that aren’t zero. No q value seems to fit the data very well, but especially not a q of zero. This analysis isn’t the easiest to perform, as neither of the two fits seem to work with my image data very well, but I can still come to the conclusion that my image isn’t random, as it doesn’t show a Poisson distribution.

5 - Analysis Part 4: Selfie Entropy

a) Explain why, if the trace is basis independent, you can use the SVD of M. Explain why the base of the log doesn't matter.

Because the SVD is just the representation of M in a different basis. So, if the trace is basis independent, then the trace of the SVD of M will be the same as the trace of the original matrix. This is advantageous, becasue the log of a full matrix is difficult to take, but the log of a diagonalized matrix is quite trivial (its just the logarithms of the diagonal entries).

The base of the log doesn't matter becasue if all we are interested in is comparing different S_{vn} values, any logarithm base will keep the relationship between different S_{vn} values the same. For example, if S_{vn}(A) = small and S_{vn}(B) = big, then this relationship will be seen no matter what the base of the logarithm in the calculation of S_{vn} is. The two matrices can then be analyzed against each other in the same way, as long as the same base is used for both calculations.

b) Now calculate $S_{vN}(\chi)$ and plot your result. If your matrix turns out to be nondiagonalizable, do your best to generalize SVD so that you can still perform the log operation.

χ is the value which determines how many of the singular values I should use. Because we are allowed to use the SVD of our matrix to calculate the entropy, as explained above, the calculation is quite easy. The logarithm of a diagonal matrix is done by taking the logarithm of each of the diagonal entries, and a diagonal matrix times another diagonal matrix produces a third diagonal matrix, where the values are the corresponding diagonals multiplied together. Because of these handy facts, we are able to simplify the von Neumann entropy equation down:

S_{vn} = -Tr(SVD log SVD) = - $\sum$ SVD_i log SVD_i

Where SVD_i is the ith diagonal entry of the SVD matrix.

We can calculate this value using χ singular values, and see how much information each singular value adds to the image. First, we need to take the SVD of my square matrix:

```
In[94]:= {us, ss, vs} = SingularValueDecomposition[sqPicData];
```

Then pull out the singular values:

```
In[95]:= squareSVs = Diagonal[ss];
```

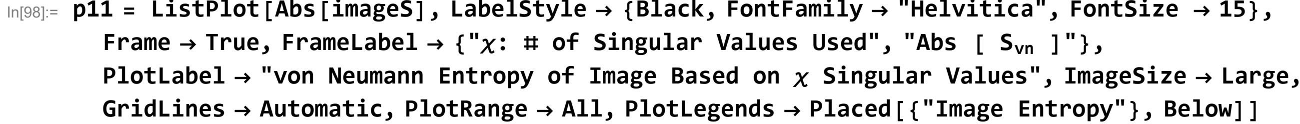
And now I can write a function that calculates $S_{vn}(\chi)$, for varying numbers of singular values:

```
In[96]:= S[χ_] := -Sum[squareSVs[[n]] Log[squareSVs[[n]]], {n, 1, χ}];
```

Now I'll make a table for all possible values of χ, and plot it, looking at the absolute value of the entropies to get a picture of the information being added to the image:

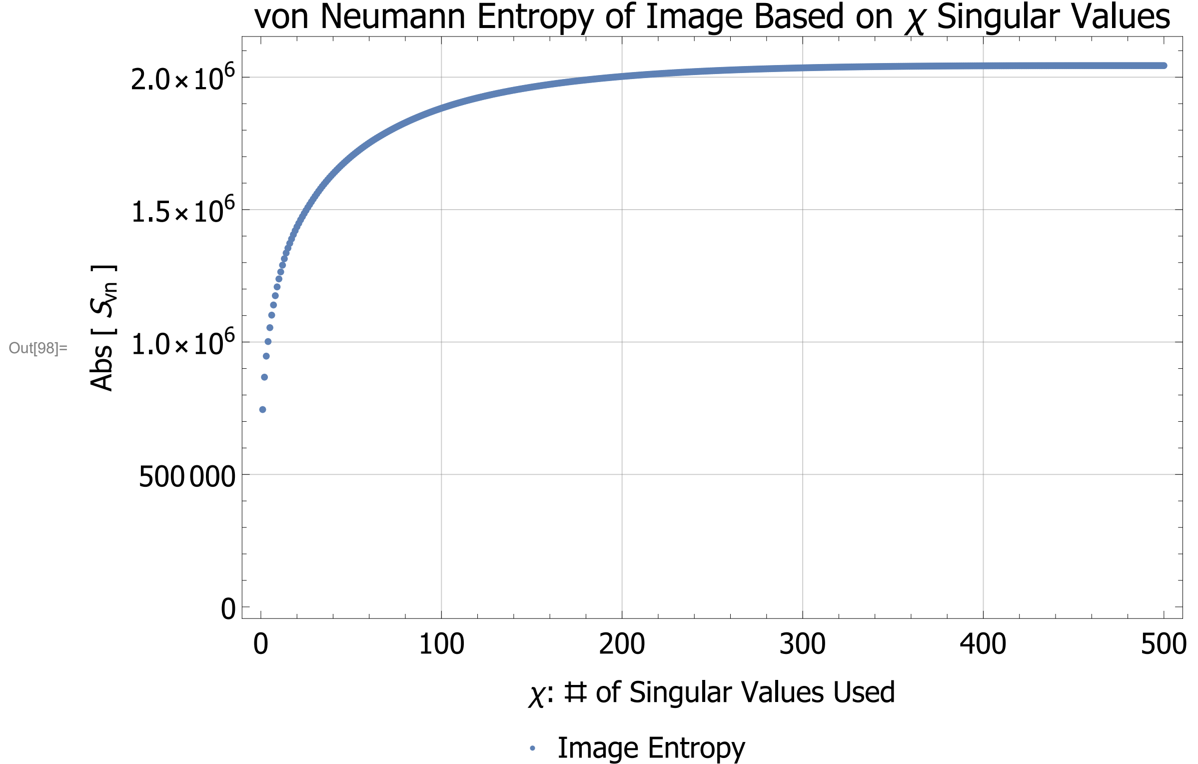
```
In[97]:= imageS = Table[S[n], {n, 1, 500}];
```

In[98]:=



```
p11 = ListPlot[Abs[imageS], LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
  Frame → True, FrameLabel → {"χ: # of Singular Values Used", "Abs [ Svn ]"},
  PlotLabel → "von Neumann Entropy of Image Based on χ Singular Values", ImageSize → Large,
  GridLines → Automatic, PlotRange → All, PlotLegends → Placed[{"Image Entropy"}, Below]]
```

Out[98]=



c) Design a model for (b) and fit your model. Calculate your goodness of fit criterion, and show both your data and your model on the same plot.

I played around for quite a while with a bunch of different models to fit this data, but eventually came to the conclusion that a logarithmic model will do the best job of modeling this relationship. It holds the same principle that it grows quickly when χ is small, and then flattens out as χ gets larger. As will be seen in the fit below, it doesn't do the best job of modeling what is happening, but it does do a fairly decent job, and I couldn't find any other relatively simple (linear) models that make sense, and have this same relationship with growth rates.

The model that I attempted to fit was: $y(\chi) = a + b \ln(\chi)$

It is quite simple, but becasue of logarithm properties, the value of b allows the model to be very versatile.

The goodness of fit criterion that I chose to use was χ^2, assuming that each of the data points have underlying Poisson statistics.

First, I find the linear fit:

In[99]:=
```
logFit = LinearModelFit[Abs[imageS], Log[x], x];
logFit // Normal
```

Out[100]= 926 330. + 193 508. Log[x]

Now, I calculate the χ^2 for the model to the data, and see what confidence level it gives:

In[101]:=
$$\chi 2 = \text{Sum}\left[\frac{(\text{Abs[imageS][[n]]} - \text{logFit[n]})^2}{\text{logFit[n]}}, \{n, 1, 500\}\right]$$

Out[101]= 906 156.

In[102]:=
```
1 - CDF[ChiSquareDistribution[498], χ2]
```

Out[102]= 0.

Ah, a 0% confidence in our fit. This is unfortunate, but I still uphold that the logarithm is the best linear model to fit to the data to describe its relationship. I could instead look at the RSquared value, or the correlation coefficient squared, for this fit. We learned to calculate this for a purely linear model (a + b x), but I can use Mathematica to do it for this model:

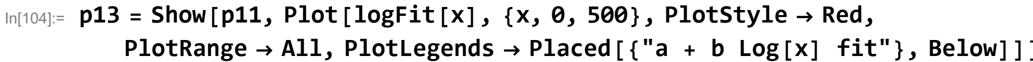
In[103]:=
```
logFit["RSquared"]
```

Out[103]= 0.920104

This R^2 value actually gives me a .92 out of 1 correlation between my data and the model. That's not too bad at all.

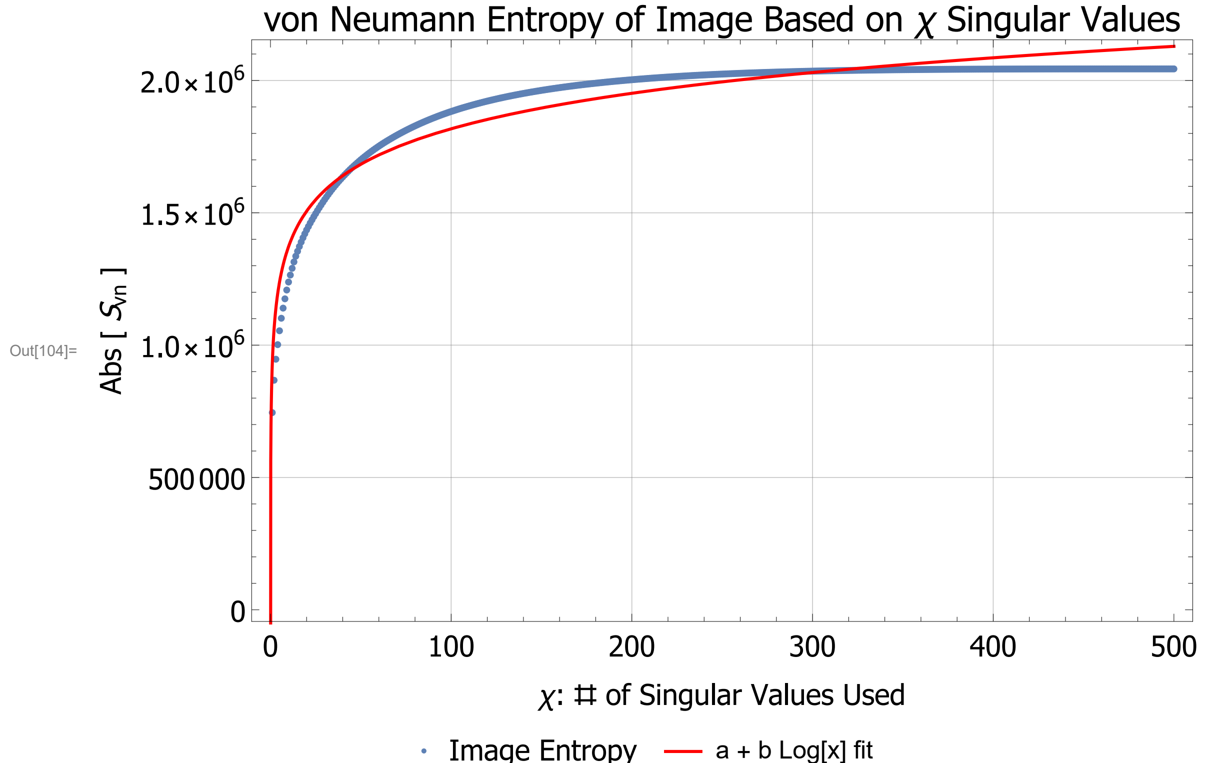
Here is what the fit looks like on the data:

In[104]:=



```
p13 = Show[p11, Plot[logFit[x], {x, 0, 500}, PlotStyle → Red,
   PlotRange → All, PlotLegends → Placed[{"a + b Log[x] fit"}, Below]]]
```

Out[104]=



d) Repeat (b)-(c) for your random matrix. Compare the information content of your Selfie to your random matrix. Make your comparison as quantitative as possible. For instance, out of the total information in your matrix, how many singular values are required to capture a given percent of it? Reflect qualitatively on how well or how badly this approach compares to whatever your eye and brain are doing when they resolve an image.

First, take the SVD of the random image produced in part 4c of the project, and extract the singular values:

In[105]:=
```
{usr, ssr, vsr} = SingularValueDecomposition[randomSq];
```

In[106]:=
```
squareRSVs = Diagonal[ssr];
```

And now I can write a function that calculates $S_{vn}(\chi)$, for varying numbers of singular values:

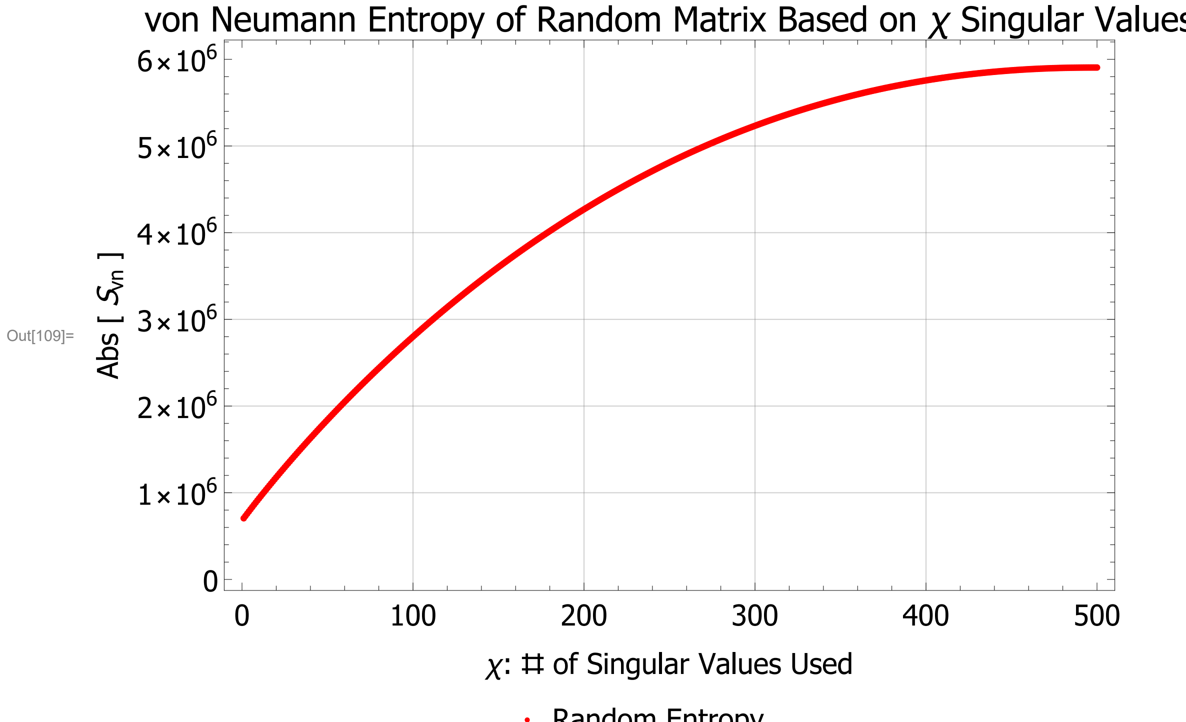
In[107]:=
```
SR[χ_] := -Sum[squareRSVs[[n]] Log[squareRSVs[[n]]], {n, 1, χ}];
```

And get a table of the entropy values for all of the values of χ, and plot it:

In[108]:=
```
randomS = Table[SR[n], {n, 1, 500}];
```

In[109]:=
```
p12 = ListPlot[Abs[randomS], LabelStyle → {Black, FontFamily → "Helvitica", FontSize → 15},
  Frame → True, FrameLabel → {"χ: # of Singular Values Used", "Abs [ S_vn ]"},
  PlotLabel → "von Neumann Entropy of Random Matrix Based on χ Singular Values",
  ImageSize → Large, GridLines → Automatic, PlotRange → All,
  PlotStyle → Red, PlotLegends → Placed[{"Random Entropy"}, Below]]
```

Out[109]=



For fitting this data, I almost immediately pictured this as an upside down parabola, and tried that right away. Turns out, its a good fit! The model I fit was the standard form for a parabola:

$y(\chi) = a + bx + cx^2$

I once again used a χ^2 analysis as my goodness of fit criterion, assuming Poisson statistics under my data.

First, I found the fit:

In[110]:=
```
fit = LinearModelFit[Abs[randomS], {x, x^2}, x];
fit // Normal
```

Out[111]= $797997. + 22069.4\,x - 24.0179\,x^2$

Now, I calculate the χ^2 for the model to the data, and see what confidence level it gives:

In[112]:=
```
χ22 = Sum[(Abs[randomS][[n]] - fit[n])^2 / fit[n], {n, 1, 500}]
```

Out[112]= 242156.

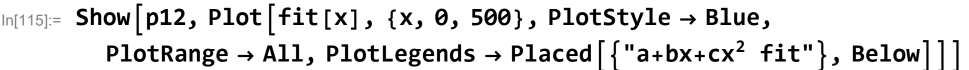
In[113]:=
```
1 - CDF[ChiSquareDistribution[498], χ22]
```
Out[113]=
```
0.
```

Ah, still a zero percent confidence model. I think this probably has something to do with the fact that I am assuming Poisson statistics for my data which has such large values. It would be much more meaningful to assume a Gaussian distribution under the data points, but we unfortunately do not have the variance of that distribution, which we need to do so. I chose Poisson in both cases becasue it is the only way to calculate χ^2 when you only know the value of the data points.

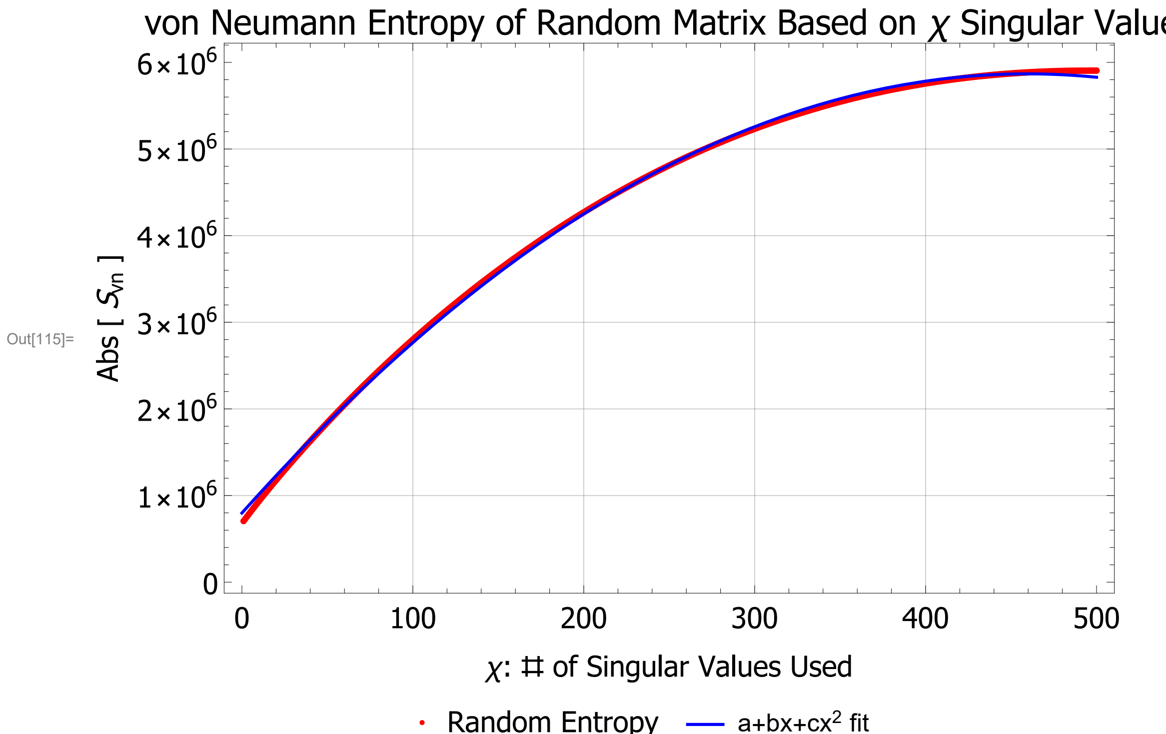
I will also calculate the R^2 value for this fit:

In[114]:=
```
fit["RSquared"]
```
Out[114]=
```
0.999549
```

That's more like it! We see that I get an R^2 value of .9995 out of 1 for my model on the data. this tells us that the data is most likely correlated along with the model. Here is what the model looks like on the data:

In[115]:=


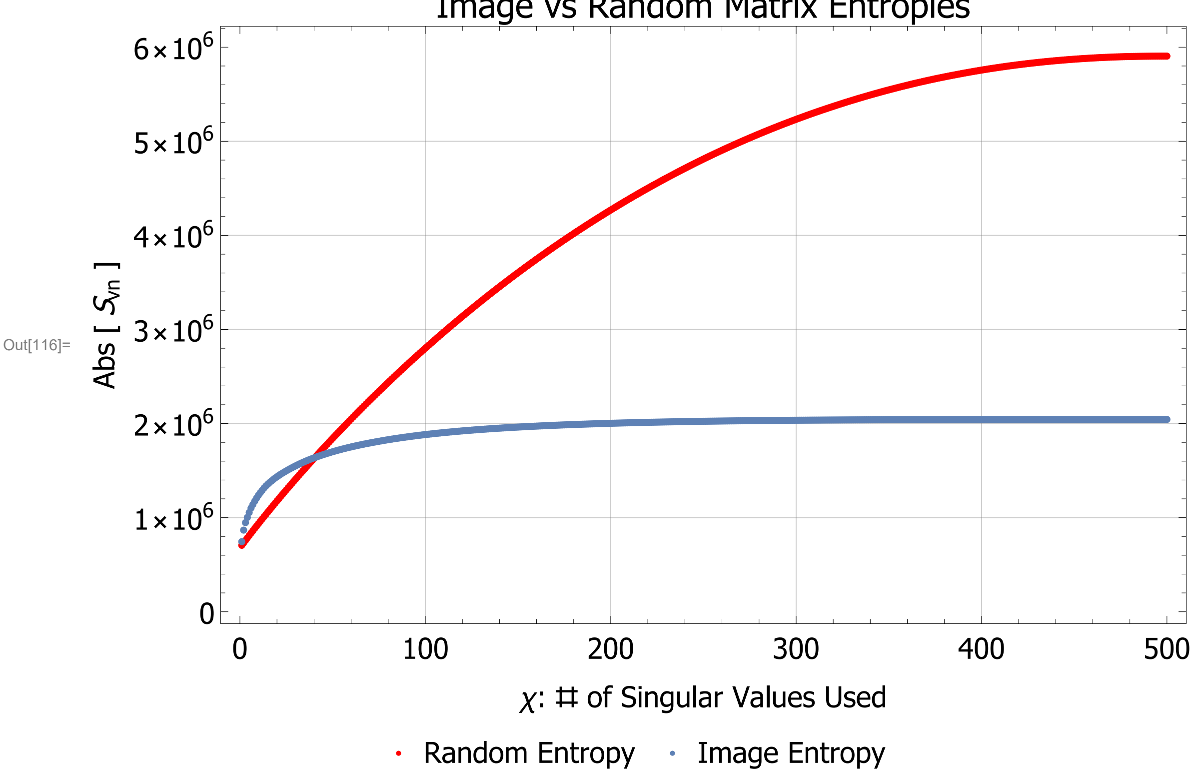
```
Show[p12, Plot[fit[x], {x, 0, 500}, PlotStyle → Blue,
  PlotRange → All, PlotLegends → Placed[{"a+bx+cx²  fit"}, Below]]]
```

Out[115]=


Now, I will compare the plots of the image versus the random matrix entropy. This can be seen on the plot below:

In[116]:=

```
Show[p12, p11, PlotLabel → "Image vs Random Matrix Entropies"]
```

Out[116]=


From this plot, it is easy to see that the random matrix contains much more entropy, or disorderedness, than the image does, which is to be expected, as a random matrix is about as disordered as you can get! It is also evident from the plot above that the first few singular values of the image contain most of the information of the image, and get it almost all the way up to it's full entropy value. This is not true for the random data, which is increasing for the addition of singular values all the way up to the 500th one. To show this quantitatively, I will calculate the number of singular values that it takes in order to obtain 75% of the maximum entropy for both the image and the random matrix:

In[117]:=
```
maxImage = Abs[Last[imageS]]
```

Out[117]= 2.04376×10^6

In[118]:=
```
maxRandom = Abs[Last[randomS]]
```

Out[118]= 5.90498×10^6

In[119]:=
```
image75p = 0;
rand75p = 0;
For[n = 1, n ≤ 500, n++,
  {If[(Abs[imageS[[n]]] > 0.75 * maxImage) && image75p == 0, image75p = n],
   If[(Abs[randomS[[n]]] > 0.75 * maxRandom) && rand75p == 0, rand75p = n]}]
```

In[122]:=
```
image75p
rand75p
```

Out[122]= 29

Out[123]= 214

And we see that it only takes 29 singular values to get to 75% of the image maximum, but 214 singular values to get to the random matrix maximum. That’s a huge difference, and exemplifies how so much more information is stored in the first few image singular values than the random ones.

How does this compare to what our eyes and brain are doing when resolving an image? Well, when we glance at an image, we might only need a very small percentage of the things we see from it to come to a conclusion as to what it is. This makes it so that our brain can process things much faster! For an example, as shown in a study (https://www.mrc-cbu.cam.ac.uk/people/matt.davis/cmabridge/), it doesn’t matter what order the letters in the middle of a word are, as long as the first and last letters have the correct position, our brain can figure out what the word is. This is our brain taking the most valuable information it can: the first and the last letters, and figuring out the rest without all the information. This makes reading much quicker! This is a fun comparison to draw to how less singular values are needed to resolve an image. Our brain probably uses less information when looking at images as well, to speed up how fast we process things.

e) Last but not least, consider the typical entropy measures for grayscale images, as you can find described at https://www.mathworks.com/help/images/ref/entropy.html or http://www.astro.-cornell.edu/research/projects/compression/entropy.html . Using a Gibbs/Shannon type entropy as described, compare quantitatively to the von Neumann entropy. When is one measure or the other more useful?

I will look at the entropy calculations for the entropy described in the Mathworks page. I will call the entropy from these calculations E as opposed to S. The entropy is defined by the following formula:

$$E = - \sum p_i \log_2(p_i)$$

where p_i is the probability of measuring the given pixel value of index i. For our 256 possible pixel values, this is done by counting how many of each pixel value there are, and dividing by 500^2 (the total number of pixels). This will give p_i, the probability of obtaining a pixel of that value. Below, I find these probabilities for both my image, and my random matrix:

In[124]:=
```
imagePis = Table[0, 256];
randomPis = Table[0, 256];
For[n = 1, n ≤ 500, n++,
 For[m = 1, m ≤ 500, m++,
  imagePis[[Floor[sqPicData[[n, m]]] + 1]] = imagePis[[Floor[sqPicData[[n, m]]] + 1]] + 1;
  randomPis[[Floor[randomSq[[n, m]]] + 1]] =
   randomPis[[Floor[randomSq[[n, m]]] + 1]] + 1;]]

imagePis = N[imagePis / (500^2)];
randomPis = N[randomPis / (500^2)];
```

Now, I get rid of all the zero values in these lists, as they will mess with logarithm in the calculation of

entropy. It is okay to get rid of them, as they should contribute nothing to the entropy of the picture.

In[129]:=
```
Ipi = DeleteCases[imagePis, x_ /; x == 0];
Rpi = DeleteCases[randomPis, x_ /; x == 0];
```

Now I can calculate the entropies of each:

In[131]:=
```
imageE = -Sum[Ipi[[n]] Log[2, Ipi[[n]]], {n, 1, Length[Ipi]}]
randomE = -Sum[Rpi[[n]] Log[2, Rpi[[n]]], {n, 1, Length[Rpi]}]
```

Out[131]= 7.85204

Out[132]= 7.99365

I double checked these values by calculating them with Matlab, and these calculated values are indeed correct to Mathworks' definition. How does this entropy compare to the von Neumann entropy? For one, they are positive, which comes as a result of the fact that all probabilities (p_i) are less than 1, and taking the Log of them in the sum turns each term in the sum negative. As the overall sum has a negative factor in front, the entropy is therefore positive. This entropy is also much smaller in value.

Maybe the most interesting difference is how close these two images are together in their entropy values. For the von Neumann entropy, the random matrix had a value about three times as large as the image. Here, the two values only differ by a very small amount. The difference is still there, as it is still a fact that the image has smaller entropy than the random matrix, but it is a much smaller difference. This leads me to believe that this type of entropy is much better at distinguishing small differences between very non-random images. I believe that this entropy should be very sensitive for the smaller values (0-6), and would be very good at differentiating between randomness there.

So, I would say that it is more useful to use the von Neumann entropy when you are trying to distinguish generally if an image is random or structured. It has a lot of resolution between completely random and completely structured. It would then be more useful to use the Gibbs/Shannon entropy when trying to distinguish small levels of randomness differences between very structured images. This could be very helpful in identifying the existence of small imperfections on an otherwise perfect surface, etc. They each hold the same quality that random images have higher entropy than structured ones, but their outputs are more useful in comparing different types of images.