Image Compression via PCA.

Numerical Linear Algebra Final Project
MATH 4510
Fred Hohman

Prompt.

The goal of this notebook is to compress arbitrary grayscale images using numerical linear algebra techniques to obtain the most visually appealing compressed image. We will be comparing the effectiveness of our program by comparing an image to its accompanying 50%, 90%, and 99% compressed version. The two methods to compress an image discussed in class were Principal Component Analysis (PCA) and Linear Gradient method. While both have their appropriate uses, this notebook will be implementing PCA as I find it more scientific and less “guess-and-check-like.” Upon reading notes on both methods, PCA takes a realistic approach to compression matrix generation by gathering real images to use as data and attempts to find an underlying structure that is common amongst all pictures. The following write-up will provide code and commentary into the workings of my implementation of PCA.

Notes:
1. Knowing in advanced that our compression algorithms would be used on pictures of animals, the training set I have compiled contains mostly animals.
2. This notebook will only evaluate if the Import images function is pointed at the correct file path of the training set. By default, the notebook will have a file path specific to my machine. To run this code on your own machine, simply change the path below to the downloaded (or another) training set.

We will use this picture of the Mac OS X Lion below as an example towards the end of the notebook.

```
LionImage =
```

Principal Component Analysis (PCA).

PCA is a statistical method that uses orthogonal transformations to turn a potentially correlated set of data into a linearly uncorrelated set of data which contain principal components. The number of principal components will be less than or equal to the total number of variables in the original dataset. Furthermore, the principal components are sorted in a way so that the first component contains the largest possible variance in the data, and each succeeding component has the next highest variance.

We can use the ideas presented in PCA to compress an image. Since animal pictures are not noise and randomly colored pixels, it is not unreasonable to think that there should be some underlying structure. In other words, if given a pure white pixel in an image, the chances that the surrounding pixels will be white or some variant of white is *probably* high. Most images do not contain sharp white to black transitions (except edges!).

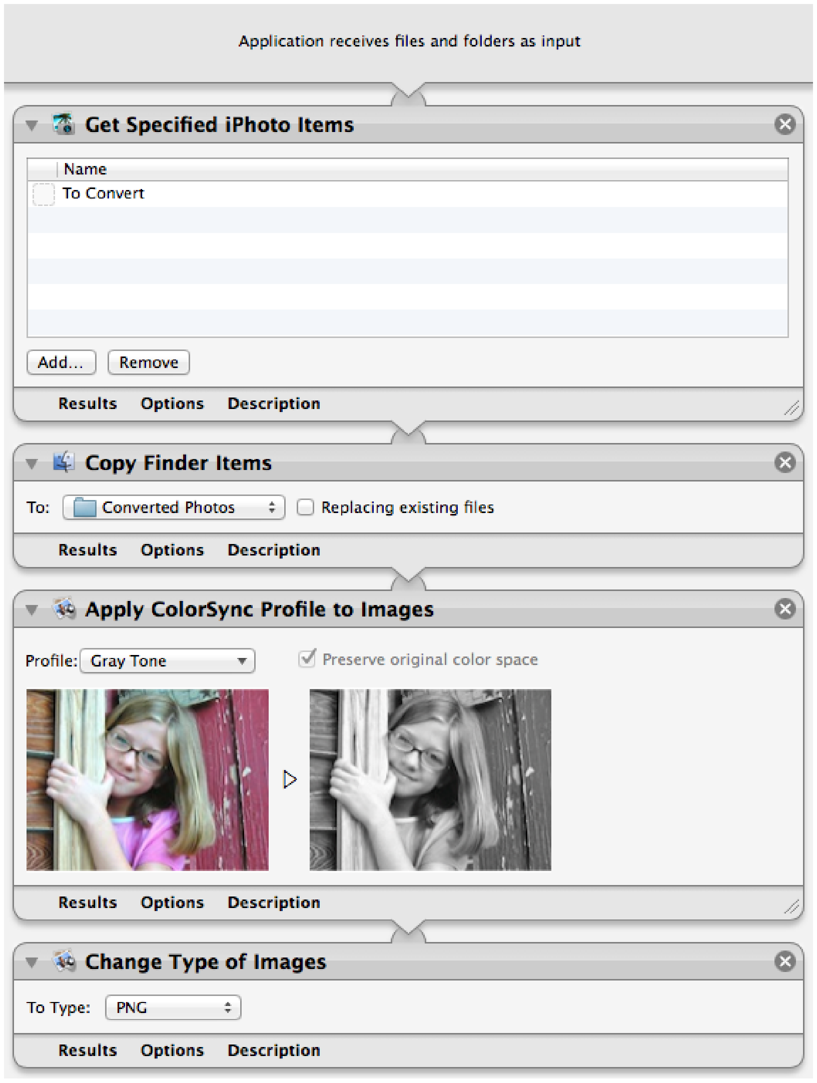
To compress an image, we want to remove superfluous pixels and replace them with other colors that are already being used by the image. However, we want to do this in a way that can eliminate pixels while maintaining image quality. So we are finding a subspace of our original image that is lower dimensional but still accurately represents the data. To compress any image, we will need to find a “basis” that contains every combination of pixels of every animal picture so that we can accurately reconstruct arbitrary images. Of course, we cannot find a basis that can represent every picture, so we will want to gather a large enough dataset that can represent our unobtainable dataset. Our approach to PCA will consider an initial dataset of 144 pictures. Each picture will be sliced in 32x32 blocks of pixels, where each block denotes a 1024-vector in $\mathbb{R}^{1024}$.

We will be compressing pictures by 50%, 90%, and 99%. Each percentage will have an accompanying matrix that represents the basis for each compression.

In summary, we want to find a basis for animal pictures to better compress arbitrary images. We will do this by gathering sample images that attempt to represent every picture of an animal so that our compression algorithm can recreate the given image. So the natural question is: *if we add more pictures to our training set does that imply a better compression?*

Automating the Training Set.

To implement PCA, I have created a training set of animal images that we will use as our original data. To convert these images to black and white, I followed an OS X Automator tutorial and made a program that batch converts images from an iPhoto photo album to an appropriate training set directory. This makes adding more images to the training set quick and easy. The work flow can be seen below.


Application receives files and folders as input

Get Specified iPhoto Items
Name
To Convert

Copy Finder Items
To: Converted Photos
Replacing existing files

Apply ColorSync Profile to Images
Profile: Gray Tone
Preserve original color space ✓

Change Type of Images
To Type: PNG

Image Manipulation Code.

The following functions will be used throughout the notebook to convert our images into *Mathematica* vectors.

Image to Data.
These functions take in an image, slices it into 32x32 blocks, Zero Means the vectors, and adjusts the contrast so that the entry values range from 0 to 1. Note the addition that I included into the ContrastAdjust function. When applying ImageToData to my training set I was getting divide by 0 errors over pure black blocks. The included “If” statement accounts for this by throwing out division values that are less that 0.001.

```
ImageToRawVectors[Img_] := Flatten[Map[#[[1]] &,
    Map[ImageData, ImagePartition[Img, 32], {2}], {4}], {{1}, {2}, {3, 4}}];
ZeroMean[v_] := {Mean[v], v - Mean[v]};
ContrastAdjust[{μ_, v_}] :=
  {μ, Max[Abs[v]], (1 / If[Max[Abs[v]] < 0.001, 1, Max[Abs[v]]]) v};
Preprocess[v_] := ContrastAdjust[ZeroMean[v]];

ImageToData[Img_] := {#[[ ;; ,  ;; , 1 ;; 2]], #[[ ;; ,  ;; , 3]]} &[
   Map[Preprocess, ImageToRawVectors[Img], {2}]];
```

Data to Image.
These functions undo the previous functions by ReMeaning the data, decontrasting, and converting the new vectors into an image.

```
ReMean[{μ_, v_}] := (# + μ) & /@ v;
DeContrast[{μ_, λ_, v_}] := {μ, λ v};
Postprocess[{AuxData_, Vectors_}] := Map[ReMean, Map[DeContrast,
    Map[{#[[1]], #[[2]], #[[3 ;;]]} &, Join[AuxData, Vectors, 3], {2}], {2}], {2}];

DataToImage[{AuxData_, Vectors_}] := ImageAssemble[
  Map[Image, Map[Partition[#, 32] &, Postprocess[{AuxData, Vectors}], {2}], {2}]]
```

Compression.
This function will take an arbitrary picture’s vectors, and apply one our compression matrices to obtain new compressed vectors.

```
CompressImage[A_, Vectors_] := Module[{APlus},
   APlus = PseudoInverse[A];
   Map[Flatten[APlus.Transpose[{#}]] &, Vectors, {2}]
  ];
```

Decompression.
This function will take our compressed vectors and reconstruct the image.

```
DecompressImage[A_, CompressedVectors_] :=
  Map[Flatten[A.Transpose[{#}]] &, CompressedVectors, {2}];
```

Implementation.

To start, let’s first import our training set.
Note: if you wish to run this code you will need to change the file path to the Training Set directory titled “Converted Photos.”
To see how many images we have, we can check the length of ImageSet.

```
Images = FileNames["*.png", "/Users/fredhohman/Dropbox/Spring
      2014/MATH 4510/Final Project/Converted Photos"];
ImageSet = Import[#] & /@ Images;
Length[ImageSet]
```

```
144
```

We now have a list of images titled ImageSet that we can call at any time.
We now want to convert every image into it’s auxiliary data and vectors. To do this, we will Map Image-ToData onto every image.
Warning: this cell will take a non-trivial amount of time to finish evaluating (no more than a couple of minutes).

```
ImageSetData = Map[ImageToData, ImageSet];
```

Let’s do some checks. We know we can call one picture’s data by taking the k’th part of our ImageSet-Data where k is an Integer with $1 \leq k \leq$ Length[ImageSetData]. So let’s take the first part of ImageSet-Data, and then take the second part so we are ignoring the auxiliary data and only considering the raw vectors of the image.

```
Dimensions[ImageSetData[[1, 2]]]
```

```
{10, 15, 1024}
```

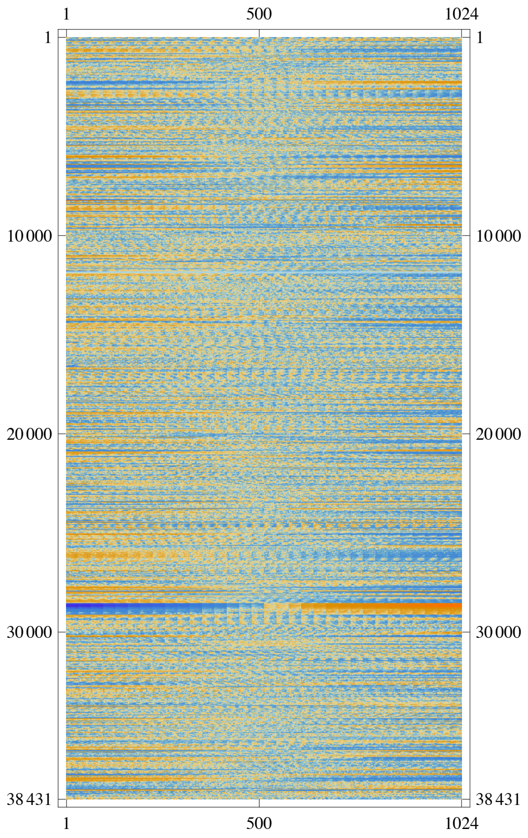
Notice that this specific picture has been broken up in 10*15=150 32x32 blocks of pixels, i.e., it’s a 10x15 matrix of 1024-vectors.
Now let’s flatten this matrix and all the other pictures and stack all the vectors into a new matrix. This will be a matrix that consists of all the 1024-vectors of our entire training set stacked on top of each other. Call this matrix M. We already know that this matrix will have 1024 columns, as each vector has the same length. The number of rows will be determined by how many pictures (and resolution) we include in our training set.

```
M = Flatten[ImageSetData[[All, 2]], 2];
Dimensions[M]
```

```
{38 431, 1024}
```

As you can see, the size of our matrix is quite large. To get a feel for the size of the matrix as well as it’s vales, let’s have *Mathematica* take a MatrixPlot.

```
MatrixPlot[M]
```



Now we can create our cross-correlation matrix for our data by dotting the columns of the total dataset together. Let’s also check the size of this new matrix.

```
CorrelationMatrix = Transpose[M].M;
Dimensions[%]
```

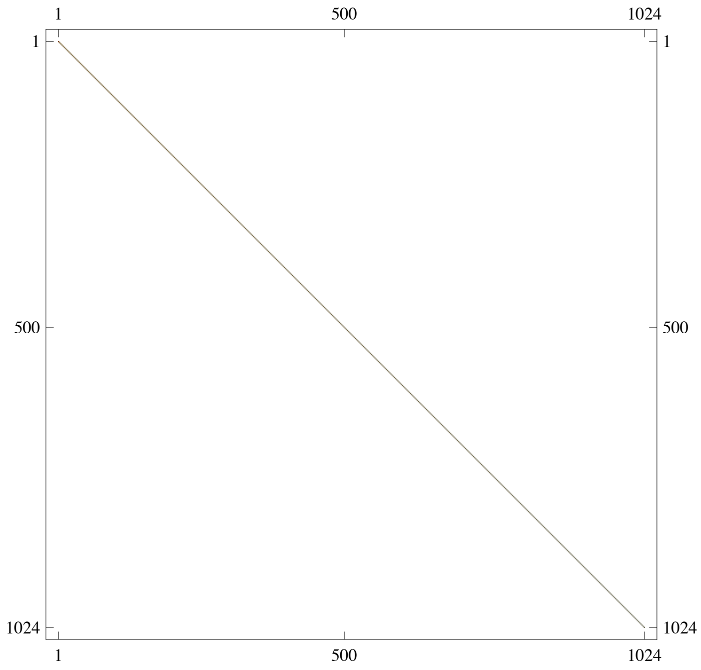
```
{1024, 1024}
```

Now we can take the singular value decomposition of our matrix to find our desired basis.
Bonus: since our matrix is square, it’s singular values will also be its eigenvalues!

```
{U, Σ, V} = SingularValueDecomposition[CorrelationMatrix];
```

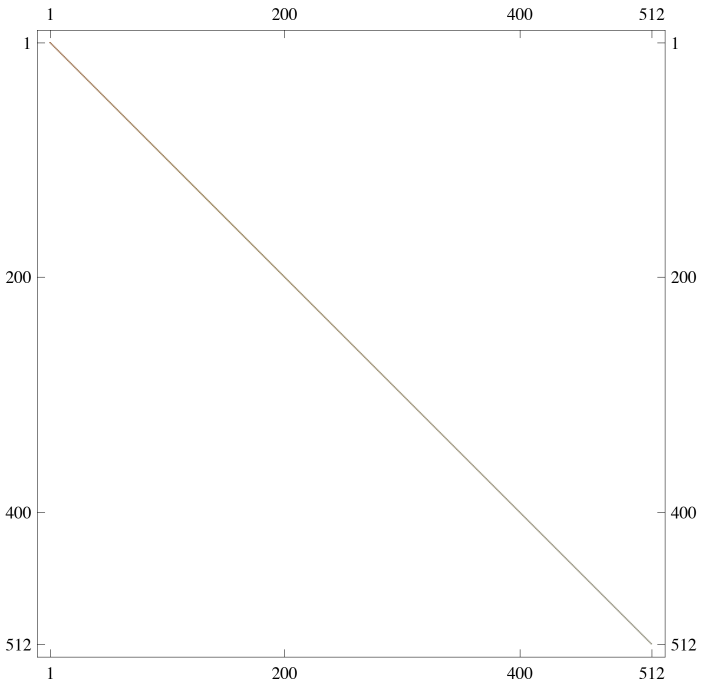
As another check, we can inspect Σ to make sure it is diagonal.

```
MatrixPlot[Σ]
```



If we were to perform a whitening transformation on our basis vectors in U we would be rotating our data into a new space of principal components by dividing each vector in U by it’s corresponding singular value. We could drop a desired amount of singular values (512, 1024-102, or 1024-10) by the following code (only considering the 512 example):

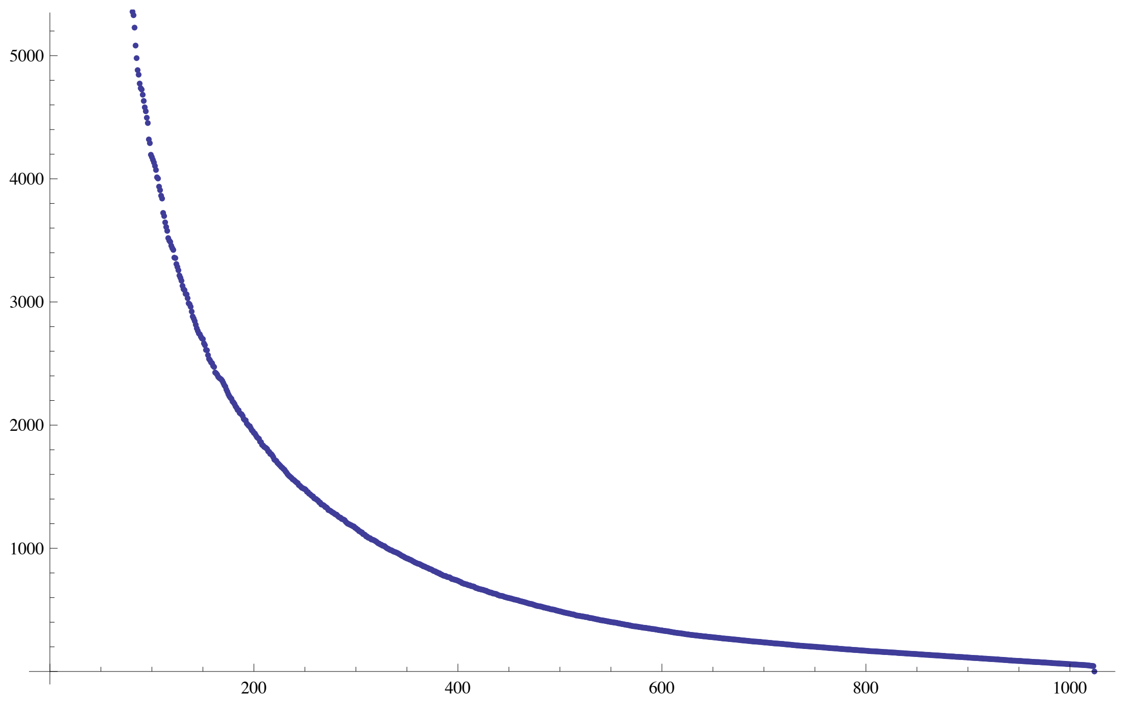
```
DiagonalΣ = Drop[Diagonal[Σ], 512];
MatrixPlot[DiagonalMatrix[DiagonalΣ]]
```



This whitening transformation is typically used as a preprocess step before applying more advanced image compression algorithms such as Independent Component Analysis (ICA); however, this notebook will only consider techniques used in PCA.

Now that we know Σ is diagonal, we can ListPlot the sorted singular values.

```
ListPlot[Diagonal[Σ], ImageSize → Large]
```



Just as we would expect, the singular values of Σ are sorted from largest to smallest, as seen by the decaying trend of our ListPlot. Ideally, we want this plot to be steeper, that is, we want as many of our singular values as possible to be towards the left side of our graph so that when we use higher compression percentages we are not losing significant singular values.

Now it is time to create our three compression matrices. We will make them by pulling the first i'th column vectors of U where i= 512, 102, or 10 (depending on compression percentage). This will give us a new basis for every compression matrix.

50% compression (512 1024-vectors).

```
A50Working = Transpose[U];
A50Working = Drop[A50Working, - (1024 - 512)];
Hohman50 = Transpose[A50Working];
Dimensions[Hohman50]
```

```
{1024, 512}
```

90% compression (102 1024-vectors).

```
A90Working = Transpose[U];
A90Working = Drop[A90Working, - (1024 - 102)];
Hohman90 = Transpose[A90Working];
Dimensions[Hohman90]
```

```
{1024, 102}
```

99% compression (10 1024-vectors).

```
A99Working = Transpose[U];
A99Working = Drop[A99Working, -(1024 - 10)];
Hohman99 = Transpose[A99Working];
Dimensions[Hohman99]
```

```
{1024, 10}
```

Now we can compress our image by solving the various least squares problems by using the functions defined above. The following code will compare all three compression percentages against the original picture.
Note: this works for images that are *only* in the training set.

CompressedTrainingImageVectors takes one of the pictures from the initial training set and one of the specified compression matrices and applies the compression to the raw vectors.

```
CompressedTrainingImageVectors[TrainingImageNumber_, CompressionMatrix_] :=
  CompressImage[CompressionMatrix, ImageSetData[[TrainingImageNumber, 2]]];
```

DecompressedTrainingImageVectors follows similarly by decompressing the raw vectors.

```
DecompressedTrainingImageVectors[TrainingImageNumber_, CompressionMatrix_] :=
  DecompressImage[CompressionMatrix,
   CompressedTrainingImageVectors[TrainingImageNumber, CompressionMatrix]];
```

CompressedImage takes the new vectors and constructs an image from them by using DataToImage.

```
CompressedImage[TrainingImageNumber_, CompressionMatrix_] :=
 DataToImage[{ImageSetData[[TrainingImageNumber, 1]],
   DecompressedTrainingImageVectors[TrainingImageNumber, CompressionMatrix]}]
```

The following function formats the output appropriately to view all 4 images in a 2x2 grid.

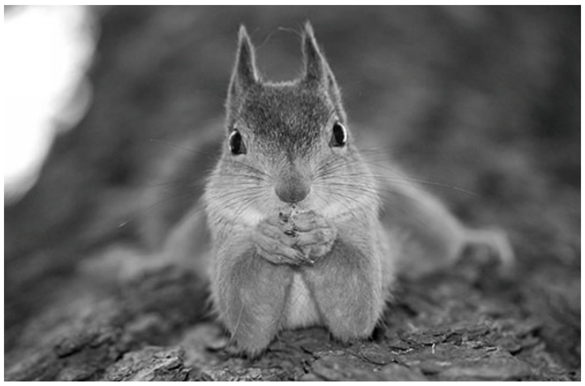
```
TrainingImageComparisonAll[TrainingImageNumber_] :=
 Grid[{
   {Grid[{{"Original"}, {Show[ImageSet[[TrainingImageNumber]], ImageSize → 300]}}],
    Grid[{{"50% Compression"},
       {Show[CompressedImage[TrainingImageNumber, Hohman50], ImageSize → 300]}}]},

   {Grid[{{"90% Compression"},
       {Show[CompressedImage[TrainingImageNumber, Hohman90], ImageSize → 300]}}],
    Grid[{{"99% Compression"}, {Show[CompressedImage[
          TrainingImageNumber, Hohman99], ImageSize → 300]}}]}
  }, Alignment →
   Top]
```

So we can now run TrainingImageComparisonAll. It’s input is one number that corresponds to the placement of an image in our training set.

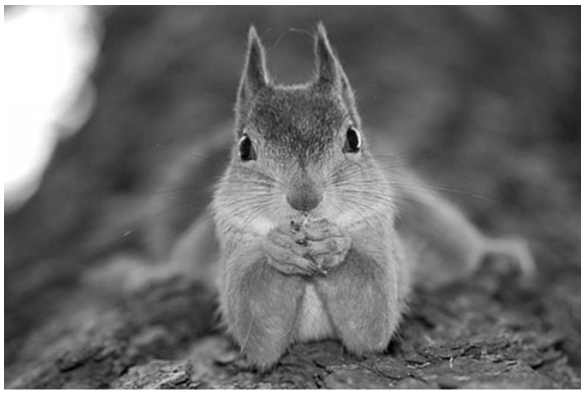
If you decide to download my training set (or use your own) this line of code below is a great way to compare the effectiveness of the various compressions.

```
TrainingImageComparisonAll[1]
```

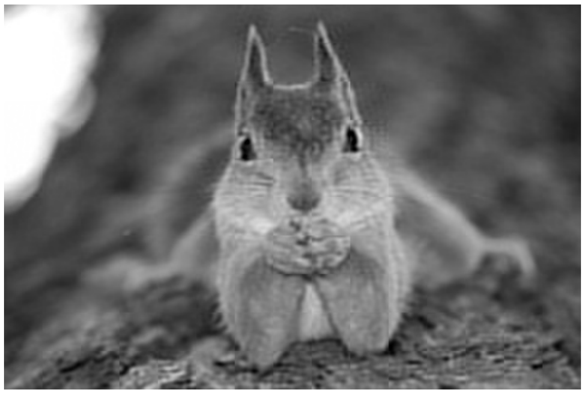
Original



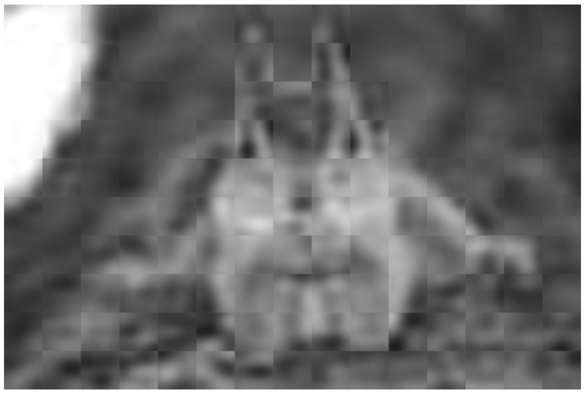
50% Compression



90% Compression



99% Compression

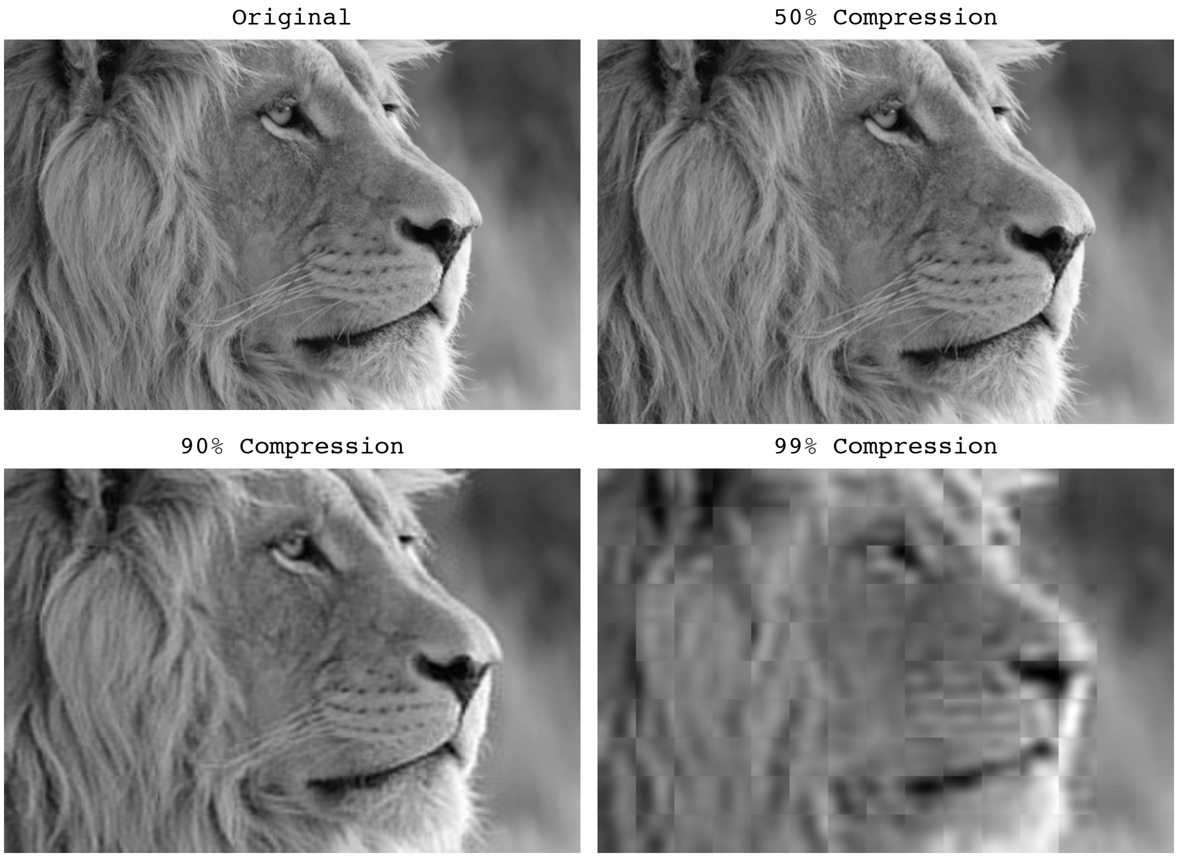


As a final test, we can apply our compression to an image that is outside of our training set—the lion picture defined above. The function below uses the same process of compressing pictures and displaying them in a 2x2 grid.

```
ImageComparison[Image_] :=
 Module[{ImageAuxData, ImageVectors, CompressedImageVectors50,
   CompressedImageVectors90, CompressedImageVectors99, DecompressedImageVectors50,
   DecompressedImageVectors90, DecompressedImageVectors99},
  {ImageAuxData, ImageVectors} = ImageToData[Image];
  CompressedImageVectors50 = CompressImage[Hohman50, ImageVectors];
  CompressedImageVectors90 = CompressImage[Hohman90, ImageVectors];
  CompressedImageVectors99 = CompressImage[Hohman99, ImageVectors];
  DecompressedImageVectors50 =
   DecompressImage[Hohman50, CompressedImageVectors50];
  DecompressedImageVectors90 = DecompressImage[
    Hohman90, CompressedImageVectors90];
  DecompressedImageVectors99 = DecompressImage[Hohman99,
    CompressedImageVectors99];

  Grid[{
    {Grid[{{"Original"}, {Show[Image, ImageSize → 300]}}],
     Grid[{{"50% Compression"}, {Show[DataToImage[
          {ImageAuxData, DecompressedImageVectors50}], ImageSize → 300]}}]},

    {Grid[{{"90% Compression"}, {Show[DataToImage[
          {ImageAuxData, DecompressedImageVectors90}], ImageSize → 300]}}],
     Grid[{{"99% Compression"}, {Show[DataToImage[{ImageAuxData,
           DecompressedImageVectors99}], ImageSize → 300]}}]}
   }, Alignment → Top]
 ]
```

To use this function, simply pass it an image and evaluate.

```
ImageComparison[LionImage]
```



Results and Final Thoughts.

As we can see from the examples above, the 50% compression matrix does a great job at removing unneeded pixels and reconstructing an approximate image. The 90% matrix has a basis that has significant corresponding singular values left out, so digital noise starts to creep into the image; however, the picture's subject is easily discernible. Finally, the 99% compressed matrix distorts the image since it lacks many significant singular values. Furthermore, the 32x32 block structure becomes apparent in this compression percentage. In the lion example we can still discern what the image's subject is, but it's unclear if every picture will have a visible structure such as the one above.

So recall our question from above: *if we add more pictures to our training set does that imply a better compression?*
When I began the project, I started using a training set of a dozen pictures. Once my code was running I began to search the Internet for pre-generated animal training sets. I found relevant pictures from two sources: Institute of Science and Technology of Austria and USC Image Database. After sifting through these image sets I picked a range of images that included high depth of field, rounded objects, sharp edges, and busy backgrounds. As a final addition to the training set, I surveyed some friends and asked them the somewhat disturbing question "*What animal represents an average of all other animals? As in, what animal contains physical characteristics as most other animals.*" I was met with some interesting

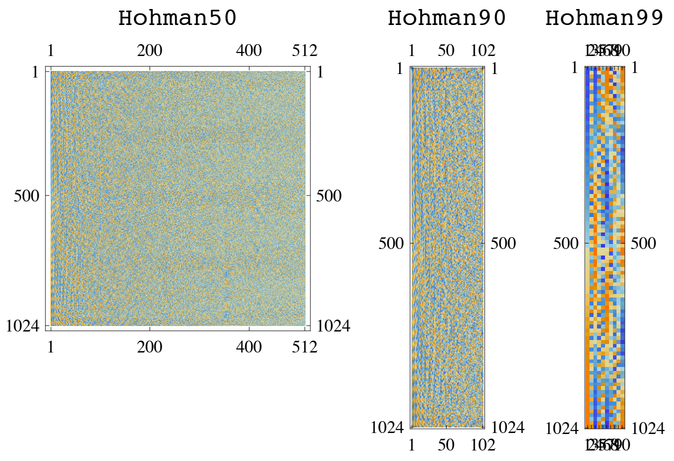
results. If you dig through my training set, I included pictures of raccoons, deer, and coatis (an interesting animal). If I could add pictures of the most common animal, maybe I could converge to a better basis faster than adding hundreds of more pictures.

After importing all the images and running my code again, my compressed images became much better. In fact, my compressed images became better after every new addition. The TrainingImageComparison function made it easy to explore the various images used inside the training set. So yes, adding more pictures in our training set does give better compression results; however, it is unclear which characteristic of the data is more beneficial: *quality* or *quantity*. I will leave this question open as a final thought to the reader!

Matrix Submissions.

The most significant component of this project is the formation of the compressing matrices that solve the least squares problem. Before we export them, let's run each of the three matrices through a *Mathematica* MatrixPlot to see their overall structure as a visual aid.

```
Grid[{
  {Grid[{{"Hohman50"}, {Show[MatrixPlot[Hohman50]]}}],
   Grid[{{"Hohman90"}, {Show[MatrixPlot[Hohman90]]}}],
   Grid[{{"Hohman99"}, {Show[MatrixPlot[Hohman99]]}}]}
 }, Alignment → Top]
```



We can now export each matrix as a .dat file.

```
Export[NotebookDirectory[] <> "Hohman50.dat", Hohman50]
Export[NotebookDirectory[] <> "Hohman90.dat", Hohman90]
Export[NotebookDirectory[] <> "Hohman99.dat", Hohman99]
```

```
/Users/fredhohman/Dropbox/Spring 2014/MATH 4510/Final Project/Hohman50.dat

/Users/fredhohman/Dropbox/Spring 2014/MATH 4510/Final Project/Hohman90.dat

/Users/fredhohman/Dropbox/Spring 2014/MATH 4510/Final Project/Hohman99.dat
```

To ensure that the export was successful we can import the matrices, inspect their dimensions, and compare their 9 top-left most elements as a quick visual/numerical check since each matrix should have the same elements in this 3x3 block.

Hohman50.

```
ImportedHohman50 = Import[NotebookDirectory[] <> "Hohman50.dat"];
Dimensions[ImportedHohman50]
ImportedHohman50[[1 ;; 3, 1 ;; 3]] // MatrixForm
Hohman50[[1 ;; 3, 1 ;; 3]] // MatrixForm
```

```
{1024, 512}
```

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

Hohman90.

```
ImportedHohman90 = Import[NotebookDirectory[] <> "Hohman90.dat"];
Dimensions[ImportedHohman90]
ImportedHohman90[[1 ;; 3, 1 ;; 3]] // MatrixForm
Hohman90[[1 ;; 3, 1 ;; 3]] // MatrixForm
```

```
{1024, 102}
```

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

Hohman99.

```
ImportedHohman99 = Import[NotebookDirectory[] <> "Hohman99.dat"];
Dimensions[ImportedHohman99]
ImportedHohman99[[1 ;; 3, 1 ;; 3]] // MatrixForm
Hohman99[[1 ;; 3, 1 ;; 3]] // MatrixForm
```

```
{1024, 10}
```

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

$$\begin{pmatrix} -0.0389501 & -0.0353258 & -0.0452956 \\ -0.0396714 & -0.0349181 & -0.0458001 \\ -0.0405226 & -0.0337179 & -0.0461929 \end{pmatrix}$$

References.

I would like to thank fellow student Eric Lybrand for our brief discussion on the theoretical workings of PCA, and Dr. Cantarella for the provided code for image and raw vector conversions.

Other notable references include:

Image Manipulation Code.
“Image Compression Project”
http://www.jasoncantarella.com/downloads/4510_imagecompression.nb

Automator Batch Image Grayscale Converter Tutorial.
“Batch convert photos with Automator”
http://www.macworld.com/article/1153650/automate_image_conversion.html

Initial Animal Training Set.
“Animals With Attributes”
http://attributes.kyb.tuebingen.mpg.de

USC Viterbi Standard Test Images.
“The USC-SIPI Image Database”
http://sipi.usc.edu/database/database.php?volume=misc

Wikipedia.
“Principal Component Analysis”
http://en.wikipedia.org/wiki/Principal_component_analysis