Principal Components and Images

Simon Jackman

March 9, 2010

1 Principal Components

A brief review of some theory before delving into the example; the following discussion is a short paraphrase of section 3.5 of I.T. Jolliffe (2002), *Principal Component Analysis* (2nd edition), Springer, New York. The idea of using image reconstruction to teach principal components comes from Jeff Lewis (UCLA, Political Science).

The method of principal components takes a n-by-p matrix $\mathbf{X}$ and yields a succession of linear combinations of the p column vectors of $\mathbf{X}$, with maximum variance subject to the constraint that each linear combination is uncorrelated with the previously recovered linear combinations. The recovered orthogonal linear combinations of $\mathbf{X}$ are known as the *principal components* of $\mathbf{X}$.

The k-th PC can be written $\mathbf{X}\alpha_k$ where α_k is a vector of length p containing the coefficients or loadings of $\mathbf{X}$ on the k-th PC; here $k = 1, \ldots, p$.

It is well known that we can compute PCs using the SVD (the singular value decomposition)

$$\mathbf{X} = \mathbf{ULA}'$$

where $\mathbf{U}$ is a n-by-r matrix with orthonormal columns (i.e., $\mathbf{U}'\mathbf{U} = \mathbf{I}_r$), $\mathbf{A}$ is a p-by-r matrix also with orthonormal columns (i.e., $\mathbf{A}'\mathbf{A} = \mathbf{I}_r$) and $\mathbf{L}$ is a r-by-r diagonal matrix, where $r \leq p$.

The SVD is closely related to an eigen-decomposition of $\mathbf{X}'\mathbf{X}$. In fact, r is the rank of $\mathbf{X}$ (and thus the rank of $\mathbf{X}'\mathbf{X}$), meaning that p - r of the p eigenvalues of $\mathbf{X}'\mathbf{X}$ are zero. For full rank $\mathbf{X}$, $p = r$. Moreover, the (r non-zero) eigenvalues of $\mathbf{X}'\mathbf{X}$ are the diagonal elements of $\mathbf{L}$. The corresponding r eigenvectors of $\mathbf{X}'\mathbf{X}$ are the columns of $\mathbf{A}$. This means that the k-th PC is simply $\mathbf{X}\mathbf{a}_k$, $k = 1, \ldots, r$. Incidentally, the eigenvectors of $\mathbf{XX}'$ are the columns of $\mathbf{U}$, useful if we were to transpose the role of variables and observations in the analysis.

The SVD decomposition can be re-expressed in scalar form as

$$x_{ij} = \sum_{k=1}^{r} u_{ik} l_k^{1/2} a_{jk}$$

where u_{ik}, a_{jk} are the (i,k) and (j,k) elements of $\mathbf{U}$ and $\mathbf{A}$, respectively, and $l_k^{1/2}$ is the k-th diagonal element of $\mathbf{L}$. If we only use the 1st m PCs ($m < r$) then we obtain

$$\tilde{x}_{ij}^{(m)} = \sum_{k=1}^{m} u_{ik} l_k^{1/2} a_{jk}$$

as an approximation to x_{ij}. It has been known since at least 1938 that $\tilde{x}_{ij}^{(m)}$ is the optimal rank m approximation to x_{ij}, in the sense of minimizing a sum of squared error criterion. That is, the n-by-p

matrix $\tilde{\mathbf{X}}^{(m)}$ is the rank m matrix that is closest to the original matrix $\mathbf{X}$, in an element-by-element least squares sense.

Note an interesting parallel between the SVD of $\mathbf{X}$ and an eigen-decomposition of $\mathbf{X}'\mathbf{X}$. The eigen-decomposition gives a series of successively better approximations of rank $1, 2, \ldots, r$ to $\mathbf{X}'\mathbf{X}$; the SVD does the same thing with respect to the matrix $\mathbf{X}$ itself.

2 Image Compression

We put the SVD to work in recovering principal components of an image. The image will be represented as a matrix $\mathbf{X}$, with each element of the matrix x_{ij} a pixel value. In its raw form, the image has n rows and p columns for a total of np pixel values. An approximation based on the 1st m principal components of $\mathbf{X}$ will generate a n by p matrix, but generated as follows:

$$\tilde{\mathbf{X}}^{(m)} = \mathbf{U}_m \mathbf{L}_m \mathbf{A}_m'$$

where $\mathbf{U}_m$ is a n-by-m matrix containing the 1st m columns of $\mathbf{U}$, $\mathbf{L}_m$ is a diagonal matrix containing the m largest eigenvalues of $\mathbf{X}'\mathbf{X}$, and $\mathbf{A}_m$ is a p-by-m matrix, containing the corresponding m eigenvectors of $\mathbf{X}'\mathbf{X}$. For $m << p$, the matrices $\mathbf{U}_m$, $\mathbf{L}_m$ and $\mathbf{A}_m$ will collectively contain substantially fewer elements than the original $\mathbf{X}$ matrix, dramatically cutting down on the memory needed to store the image and the bandwidth needed to transmit the image. In this sense we are using SVD and principal components to perform a crude form of image compression.

We begin with an image that is well known to students of American politics and American popular culture, shown in Figure 1; this image also appears on a t-shirt worn regularly by Alex Tahk.

The `R` package `biOps` has support for image processing, including reading images in JPEG or TIFF format. Figure 1 is in JPEG format, and we read it as follows:

R Code

```
> library(biOps)
> x <- readJpeg("ElvisNixon.jpg")
```

The `biOps` package represents the image as follows:

R Code

```
> class(x)
```

```
[1] "imagedata" "array"
```

R Code

```
> dim(x)
```

```
[1] 2400 3070    3
```

That is, the image is an object of class `imagedata` (a class defined by the `biOps` package, for which there are numerous helpful methods, beyond our scope here) and also as an object of class `array`, with dimensions 2400 by 3070 by 3. An image like that in Figure 1 is simply an array of pixels, with each pixel taking on an integer in the range 0 to 255 (inclusive) for each of three color channels in the RGB color space (red, green and blue, respectively). The 3rd dimension indexes the three color channels.

We convert the image to a grayscale image (or "black-and-white" image), with the `imgGreenBand` command. The resulting object `y` is now just a two dimensional array, what we would call a matrix:

Figure 1: The Original Image

R Code

```
> y <- imgGreenBand(x)
> dim(y)
```

```
[1] 2400 3070
```

The first 100 entries of the first row of `y` are as follows:

R Code

```
> y[1,1:100]
```

```
  [1] 216 214 215 213 213 213 210 209 207 205 205 207 205 204 204 202 200 197
 [19] 193 187 182 180 173 164 153 141 131 117 105  95  90  84  81  83  85  86
 [37]  90  92  98 107 116 128 142 151 162 173 180 190 198 206 213 219 222 228
 [55] 232 235 236 238 239 239 240 241 241 241 242 241 240 239 238 238 237 237
 [73] 237 235 234 233 231 230 230 229 227 225 225 225 225 227 229 230 232 234
 [91] 234 235 234 233 232 229 229 227 224 221
```

We will also re-scale the image, since right now it contains 7368000 pixels:

R Code

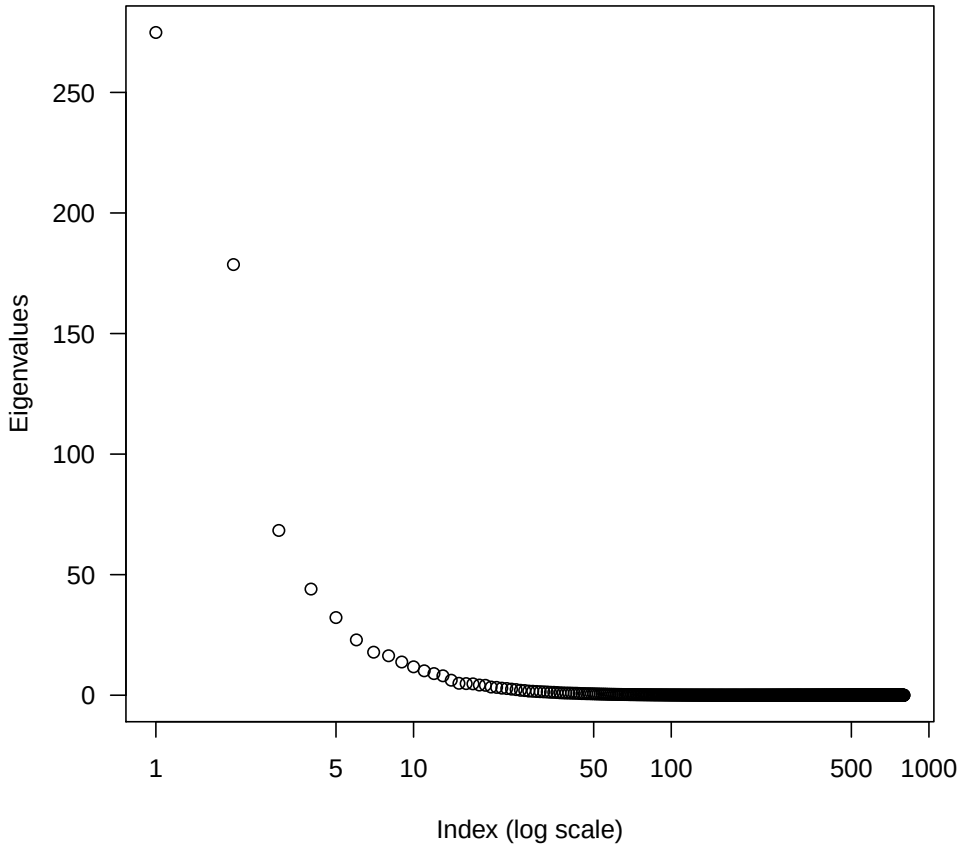
```
> y <- imgMedianShrink(y,1/3,1/3)
> dim(y)
```

```
[1]  800 1023
```

We now compute a correlation matrix over $\mathbf{y}'$ (we take the transpose because $\mathbf{y}$ has $n < k$), and look at its eigenvalues, with the following code chunk generating a "scree plot":

R Code

```
> r <- cor(t(y))
> lambda <- eigen(r)$values
> plot(y=lambda,
+      x=1:length(lambda),
+      log="x",
+      las=1,ylab="Eigenvalues",xlab="Index (log scale)")
```



There are 800 eigenvalues here; the number of columns in $\mathbf{y}'$ (or the number of rows in $\mathbf{y}$). But only relatively few are "large", say, greater than one. In fact, we have just 37 eigenvalues of r greater than one:

R Code

```
> table(lambda>1)
```

```
FALSE  TRUE
  763    37
```

R Code

```
> lambda[lambda>1]
```

```
 [1] 274.867407 178.615533  68.348555  44.034584  32.212559  22.951311
 [7]  17.849712  16.379764  13.785370  11.780917  10.127799   8.982727
[13]   8.057273   6.234990   4.952323   4.828187   4.726457   4.231635
[19]   4.115349   3.396935   3.252199   2.935491   2.787100   2.533196
[25]   2.343375   2.048518   1.949331   1.725263   1.644889   1.556270
[31]   1.475916   1.401756   1.331100   1.228653   1.203701   1.124041
[37]   1.018235
```

We now turn to the principal components analysis. First, I define a utility function of sorts that will make sure that the output contains valid pixel values (integers in the range 0 to 255):

R Code

```
> std <- function(x){
+   y <- matrix(c(0,255),2,1)
```

```
+   A <- cbind(range(x),1)
+   coefs <- solve(A,y)
+   xmat <- cbind(x,1)
+   z <- xmat%*%coefs
+   z <- round(z)
+   z[z<0] <- 0
+   z[z>255] <- 255
+   z
+ }
```

Next, I define a function that takes two arguments, `s` (the output of a singular value decomposition) and `k`, the number of principal components to use when reconstructing the image.

R Code

```
> reconstruct <- function(s,k=1){
+   U <- s$u[,1:k]
+   D <- matrix(0,k,k)
+   diag(D) <- s$d[1:k]
+   V <- s$v[,1:k]
+   xhat <- U%*%D%*%t(V)
+   z <- apply(xhat,1,std)
+   return(t(z))
+ }
```

I also define a function that returns the RMSE (root mean square error) of the approximation $\tilde{\mathbf{X}}^{(m)}$; this will provide a quantitative assessment of model fit:

R Code

```
> rmse <- function(y,z){
+   e <- y - z
+   return(sqrt(mean(e^2)))
+ }
```

Now we put these functions to work. The function `svd` performs a singular value decomposition of the image; we save the output in `s`. We then call the `reconstruct` function with an increasing sequence of principal components, as given in `ksteps`. We save the reconstructed images in `JPEG` files using the `writeJpeg` function in the `biOps` package. I then call the `ImageMagick` utilities `convert` and `montage` to annoate and stack the output `JPEG` files, respectively.

R Code

```
> s <- svd(y)   ## singular value decomposition
> ksteps <- c(1:20,25,30,35,40,50,60,75,100,200,800)
> M <- length(ksteps)
> e <- rep(NA,M)
> for(i in 1:M){
+   if(i<10)
+     iNum <- paste("0",i,sep="")
+   else
+     iNum <- as.character(i)
+   jpegFileName <- paste("reconstruction",iNum,".jpg",sep="")
+
+   z <- reconstruct(s,k=ksteps[i])
+   e[i] <- rmse(y,z)
+   writeJpeg(filename="tmp.jpg",
+             imgdata=imagedata(z))
+
+   ## add a label with ImageMagick convert function
+   annOpts <- paste("-quality 100",
+                    "-fill darkblue",
+                    "-font Helvetica",
+                    "-pointsize 64")
+   drawCommands <- paste("-draw \"text 910,780 \'",
```

```
+                         rep(" ",2-floor(log(ksteps[i],base=10))),
+                         ksteps[i],
+                         "\'\"",
+                         sep="")
+
+   system(paste("/usr/local/bin/convert",
+                annOpts,
+                drawCommands,
+                "tmp.jpg",
+                jpegFileName))
+ }
> ## make montage
> allFiles <- paste(dir(pattern=glob2rx("reconstruction*.jpg")),collapse=" ")
> montageCmd <- paste("/usr/local/bin/montage",
+                     "-quality 100",
+                     allFiles,
+                     "-tile 5x6",
+                     "-geometry +5+5",
+                     "montage.jpg")
> system(montageCmd)
```

Figures 2 and 3 show the reconstructed images; the number in the top left corner shows the number of principal components used in generating the particular image.

3 Compression

Note the compression being obtained here. The uncompressed image consumes a substantial amount of memory in R:

R Code

```
> imageSize <- object.size(y)
> imageSize
```

```
3274280 bytes
```

while the representation based on say, 20 principal components is put together using quantities that are much smaller:

R Code

```
> s20 <- object.size(s$u[,1:20]) + object.size(s$d[1:20]) + object.size(s$v[,1:20])
> s20
```

```
292280 bytes
```

R Code

```
> s20/imageSize
```

```
0.0892654262921925 bytes
```

For 75 principal components we obtain

R Code

```
> s75 <- object.size(s$u[,1:75]) + object.size(s$d[1:75]) + object.size(s$v[,1:75])
> s75
```

```
1094840 bytes
```

R Code

```
> s75/imageSize
```

```
0.33437580170297 bytes
```

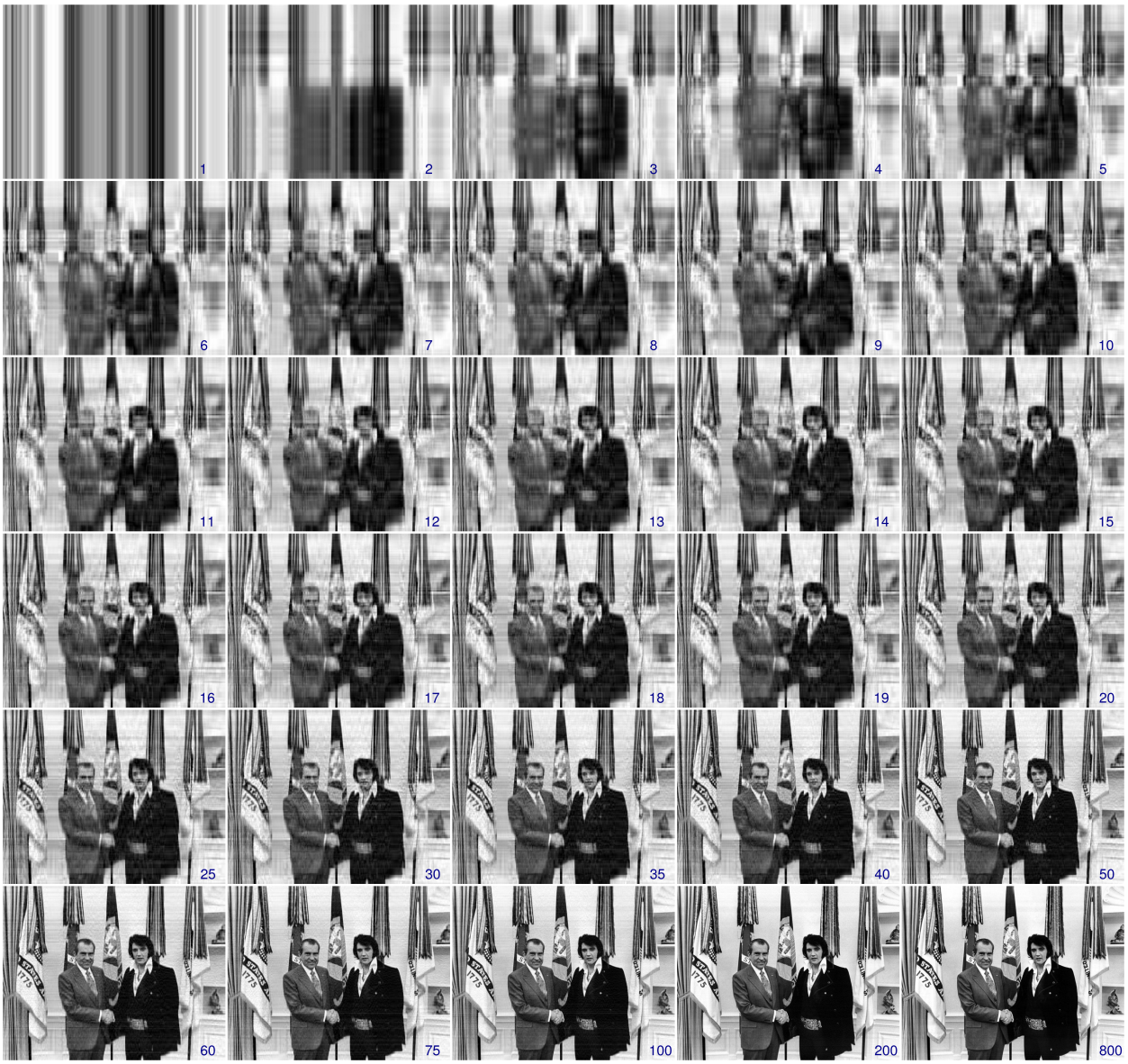


Figure 2: Reconstructions of Figure 1 via Principal Components.

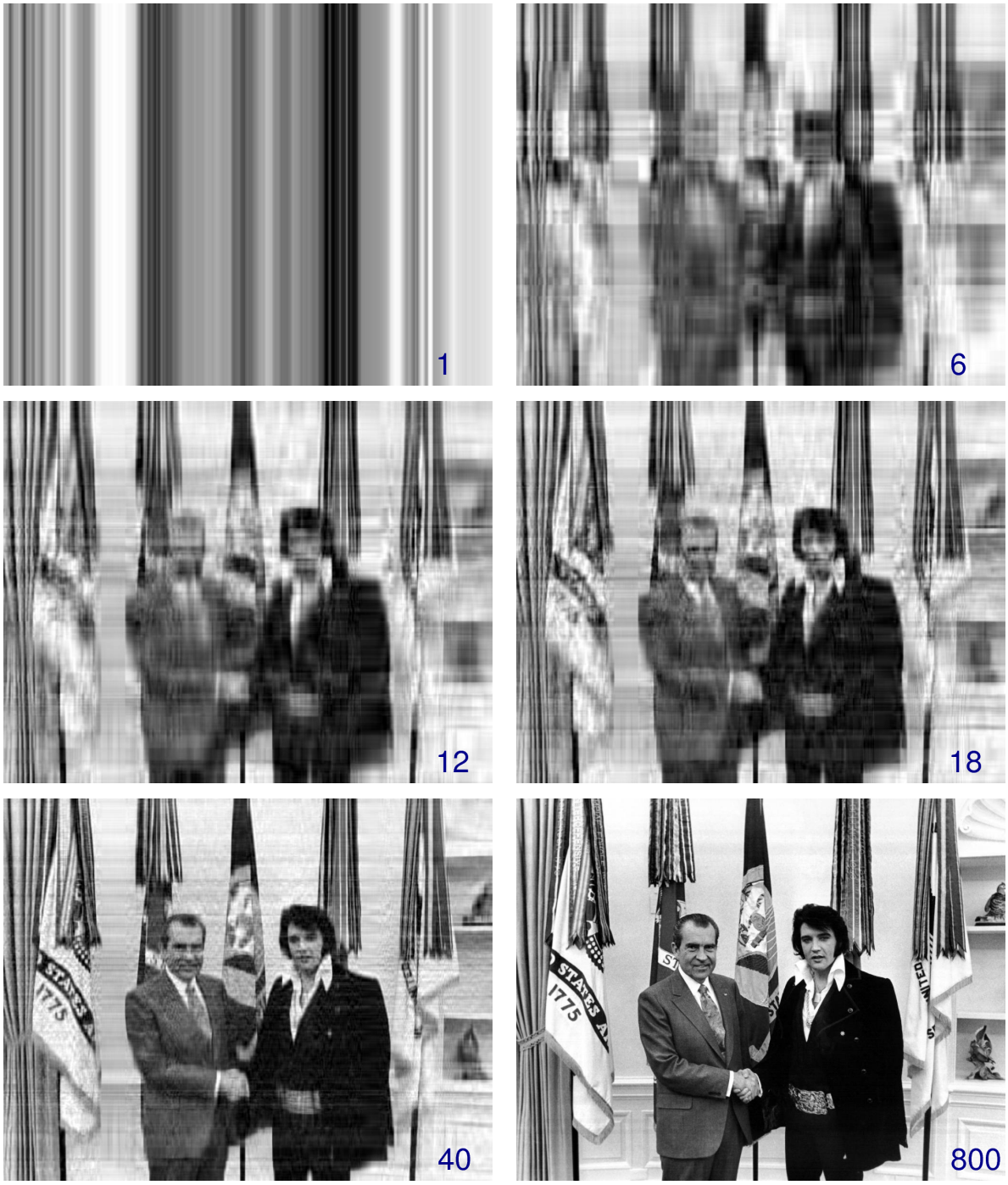


Figure 3: Reconstructions of Figure 1 via Principal Components.

4 RMSE

Finally, also note the RMSE associated with the series of approximations we employ here:

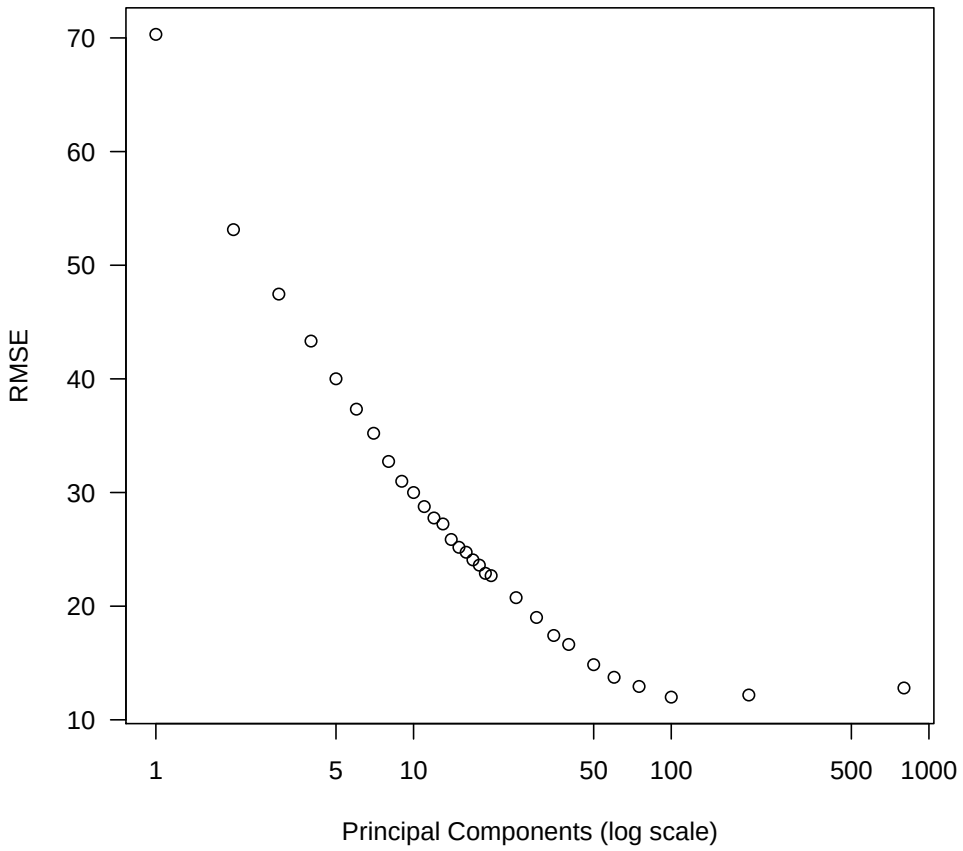
R Code

```
> e
```

```
 [1] 70.31476 53.13173 47.46061 43.32714 40.00887 37.33567 35.21456 32.73504
 [9] 30.99049 30.00023 28.76435 27.76268 27.23549 25.86774 25.18017 24.74490
[17] 24.07475 23.60934 22.88823 22.68588 20.75123 19.00905 17.42042 16.63200
[25] 14.85263 13.74332 12.93699 11.99818 12.18418 12.80405
```

R Code

```
> plot(e~ksteps,
+      xlab="Principal Components (log scale)",
+      log="x",
+      las=1,
+      ylab="RMSE")
```



Interestingly, the series appears to reach a minimum around 100 principal components, at which point rounding error seems to kick in. Notice too that my function `std` normalizes the pixel values to span 0 to 255, while the original image did not have that much variability, imposing a certain level of "minimum error" here.

Sweave time for this document:
1.55 seconds.