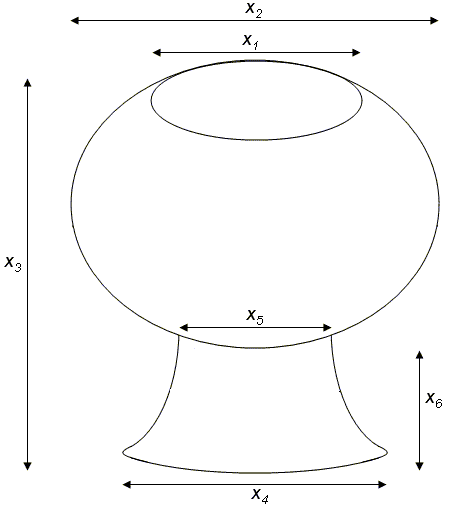
**4.3 – The Importance of Scaling/Standardizing**

Consider again all measurements recorded in the goblets dataset (Example 3.1) - the mouth width (), total width (), total height (), base width (), stem width (), and stem height ().



A quick review of image above (and the data) reveals that we’d certainly expect to have more variation than and and to have more variation than , etc. These differences in variation may adversely affect our principal component analysis.

To alleviate this issue, we usually (almost always) ***standardize*** the data before performing a principal component analysis.

> meanvec=apply(Goblets,2,mean)

MouthWidth TotalWidth TotalHeight BaseWidth StemWidth StemHeight

13.28 17.84 20.44 14.60 6.36 8.12

> sigma=var(Goblets)

MouthWidth TotalWidth TotalHeight BaseWidth StemWidth StemHeight

MouthWidth 9.043333 8.13000 6.330000 8.408333 4.478333 5.548333

TotalWidth 8.130000 18.80667 22.115000 14.891667 5.435000 10.853333

TotalHeight 6.330000 22.11500 36.923333 21.225000 3.293333 16.361667

BaseWidth 8.408333 14.89167 21.225000 17.166667 4.358333 11.841667

StemWidth 4.478333 5.43500 3.293333 4.358333 4.656667 1.955000

StemHeight 5.548333 10.85333 16.361667 11.841667 1.955000 9.860000

This is more easily achieved by using the scale command in R.

> Std.Goblets = scale(Goblets)

As expected the mean and variance are 0 and 1, respectively. The covariance terms are

> apply(Std.Goblets,2,mean)

Std.MouthWidth Std.TotalWidth Std.TotalHeight Std.BaseWidth Std.StemWidth Std.StemHeight

0 0 0 0 0 0

> var(Std.Goblets)

Std.MouthWidth Std.TotalWidth Std.TotalHeight Std.BaseWidth Std.StemWidth Std.StemHeight

Std.MouthWidth 1.0000 0.6234 0.3464 0.6748 0.6901 0.5876

Std.TotalWidth 0.6234 1.0000 0.8392 0.8288 0.5808 0.7970

Std.TotalHeight 0.3464 0.8392 1.0000 0.8431 0.2512 0.8575

Std.BaseWidth 0.6748 0.8288 0.8431 1.0000 0.4875 0.9102

Std.StemWidth 0.6901 0.5808 0.2512 0.4875 1.0000 0.2885

Std.StemHeight 0.5876 0.7970 0.8575 0.9102 0.2885 1.0000

The covariance structure does not change in the standardization process; thus, the covariance terms in the above matrix are agree with the correlations on the original data.

> cor(Goblets)

MouthWidth TotalWidth TotalHeight BaseWidth StemWidth StemHeight

MouthWidth 1.0000 0.6234 0.3464 0.6748 0.6901 0.5876

TotalWidth 0.6234 1.0000 0.8392 0.8288 0.5808 0.7970

TotalHeight 0.3464 0.8392 1.0000 0.8431 0.2512 0.8575

BaseWidth 0.6748 0.8288 0.8431 1.0000 0.4875 0.9102

StemWidth 0.6901 0.5808 0.2512 0.4875 1.0000 0.2885

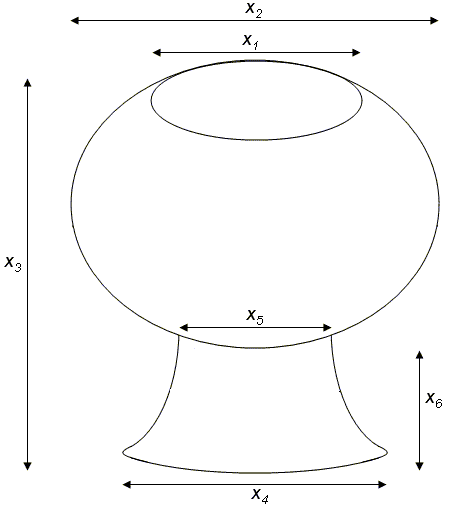
StemHeight 0.5876 0.7970 0.8575 0.9102 0.2885 1.0000

Paraphrasing the results above :

Doing a principal component analysis using the correlation matrix instead of the variance/covariance matrix is equivalent to first standardizing your data (to alleviate issues with differences in scale amongst the ) and running a PC analysis on the standardized variables. In JMP for example you can choose between performing PCA on the variance/covariance matrix (unstandardized) or PCA on the correlation matrix (standardized).

Example 4.1: Goblets (cont’d)

All measurements recorded -- the mouth width (), total width (), total height (), base width (), stem width (), and stem height () on each of the 25 goblets.



Doing the PC analysis in R, using the cor=TRUEoption.

> pca=princomp(Goblets,cor=TRUE)

Using the summary() function to determine the proportion of variance due to each PC

> summary(pca)

Importance of components:

Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6

Standard deviation 2.066828 1.0450729 0.62028038 0.37735439 0.25552624 0.208823116



Proportion of Variance 0.711963 0.1820296 0.06412462 0.02373272 0.01088228 0.007267849

Cumulative Proportion 0.711963 0.8939925 0.95811715 0.98184987 0.99273215 1.000000000

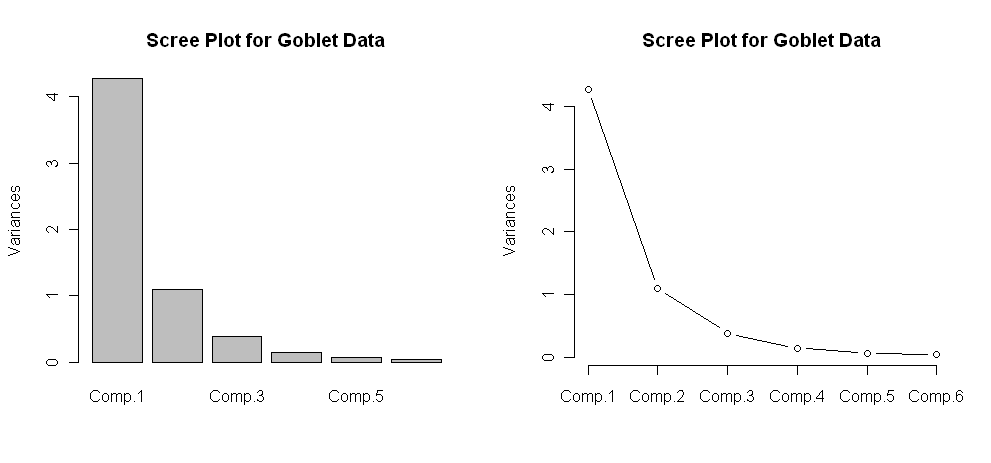
A **scree plot** is a plot that can be used to help determine how many principal components should be retained. In the above, we see that last three (or four) principal components add very little to our understanding of the total variation in the Goblet data.

Obtaining a Scree Plot in R...

> plot(pca,type="bar",main="Scree Plot for Goblet Data")

A line version of this plot can be obtained in R by changing the type to line…

> plot(pca,type="line",main="Scree Plot for Goblet Data")



How are the principal component scores computed, i.e. what are the defining linear combinations of the original ?

> pca$loadings

Loadings:

Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6

MouthWidth -0.366 0.486 0.618 0.324 -0.278 0.256

TotalWidth -0.452 -0.375 0.674 -0.439

TotalHeight -0.411 -0.441 -0.316 -0.383 0.624

BaseWidth -0.462 -0.115 0.159 -0.541 -0.382 -0.556

StemWidth -0.296 0.683 -0.491 -0.359 0.221 0.163

StemHeight -0.438 -0.298 0.332 -0.133 0.758 0.130

|  |
| --- |
| Custom R Function |
| loadplot <- function (x, variables = 1:min(5, dim(x)[2]), nbars = 6, ...)  {  p <- dim(x)[1]  q <- dim(x)[2]  on.exit(par(oldpar))  oldpar <- par(mfrow = c(length(variables), 1), cex = 0.75,  pty = "m")  par(cex = 0.75)  vnames <- dimnames(x)[[1]]  cnames <- dimnames(x)[[2]]  if (nbars > p)  nbars <- 1:p  else if (nbars < 1)  nbars <- 1  else nbars <- 1:nbars  for (i in variables) {  xi <- x[, i]  barplot(xi, names = vnames, main = cnames[i], density = 20,  col = "blue", ...)  abline(h=0)  }  invisible()  } |

> loadplot(pca$loadings, variables=1:3)

Timeline

Description automatically generated

**Questions:**

1. What is the make-up of the 1st principal component?
2. What is the make-up of the 2nd principal component?

Plotting the scores in R

> plot(pca$scores,main="Plot of PC1 vs PC2 for Goblet Data")

|  |  |
| --- | --- |
|  |  |

A better way to visualize the results from a PCA is to use a plot called \*biplot\*. This plot shows the first two PC's and includes the direction to which observations are being pulled based upon their principal component loadings on the variables.

In R…

> biplot(pca)

Chart

Description automatically generated

**4.4 – Principal Components Using Singular Value Decomposition (SVD)**

As we have seen above the principal components can be derived by performing a spectral decomposition (i.e. eigenanalysis of the variance/covariance matrix or the correlation matrix). As an alternative we consider Singular Value Decomposition (SVD).

Singular value decomposition will decompose a data matrix into a product of three matrices.

where,

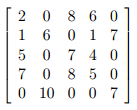
.

matrix containing the singular values in a submatrix.   
 The singular values are the square roots of the non-zero eigenvalues of .

.

Typically, the first few singular values are large relative to the rest. We can achieve an approximation to the original matrix by retaining the largest singular values and their corresponding eigenvectors contained in (. Below is an example of a SVD decomposition of a matrix and a lower dimensional approximation to the matrix using the method described above.

Consider the matrix below:



we will decompose this matrix using SVD in R using the command svd.

> X = matrix(scan(),nrow=5,ncol=5,byrow=T)

1: 2 0 8 6 0

6: 1 6 0 1 7

11: 5 0 7 4 0

16: 7 0 8 5 0

21: 0 10 0 0 7

26:

Read 25 items

> Xsvd = svd(X)

> Xsvd

$d

[1] 17.9183709 15.1713719 3.5640020 1.9842282 0.3495557

$u

[,1] [,2] [,3] [,4] [,5]

[1,] -0.54225536 0.06499573 0.82161708 0.10574661 -0.12448979

[2,] -0.10181247 -0.59346055 -0.11255162 0.78812338 0.06026999

[3,] -0.52495325 0.05938171 -0.21296861 -0.11574223 0.81372354

[4,] -0.64487038 0.07040626 -0.50874368 -0.05990271 -0.56282918

[5,] -0.06449519 -0.79692967 0.09000966 -0.59219473 -0.04412631

$v

[,1] [,2] [,3] [,4] [,5]

[1,] -0.46461713 0.02150651 -0.86850856 0.0007995538 -0.1713494

[2,] -0.07008599 -0.75998796 0.06307148 -0.6013456703 -0.2278412

[3,] -0.73509354 0.09879712 0.28400852 -0.2234845728 0.5650402

[4,] -0.48439167 0.02544740 0.39886566 0.3326838129 -0.7035231

[5,] -0.06496983 -0.64151954 -0.04427431 0.6912010359 0.3232839

> U = Xsvd$u

> D = diag(Xsvd$d)

> D

[,1] [,2] [,3] [,4] [,5]

[1,] 17.91837 0.00000 0.000000 0.000000 0.0000000

[2,] 0.00000 15.17137 0.000000 0.000000 0.0000000

[3,] 0.00000 0.00000 3.564002 0.000000 0.0000000

[4,] 0.00000 0.00000 0.000000 1.984228 0.0000000

[5,] 0.00000 0.00000 0.000000 0.000000 0.3495557

> V = Xsvd$v

> U%\*%D%\*%t(V)

[,1] [,2] [,3] [,4] [,5]

[1,] 2.000000e+00 -1.913400e-15 8.000000e+00 6.000000e+00 -8.552187e-16

[2,] 1.000000e+00 6.000000e+00 6.401130e-16 1.000000e+00 7.000000e+00

[3,] 5.000000e+00 1.762479e-15 7.000000e+00 4.000000e+00 7.216450e-16

[4,] 7.000000e+00 -1.110223e-16 8.000000e+00 5.000000e+00 -8.326673e-17

[5,] 5.306519e-15 1.000000e+01 -1.916869e-15 -5.349887e-15 7.000000e+00

Notice that the full singular value decomposition !

We can alternatively complete the SVD of by finding completing the eigenanalysis of & and then using the results to form This process is shown below.

> XXt = X%\*%t(X)  
> XXt

[,1] [,2] [,3] [,4] [,5]

[1,] 104 8 90 108 0

[2,] 8 87 9 12 109

[3,] 90 9 90 111 0

[4,] 108 12 111 138 0

[5,] 0 109 0 0 149

> XtX = t(X)%\*%X  
> XtX

[,1] [,2] [,3] [,4] [,5]

[1,] 79 6 107 68 7

[2,] 6 136 0 6 112

[3,] 107 0 177 116 0

[4,] 68 6 116 78 7

[5,] 7 112 0 7 98

> XXteigen = eigen(XXt)  
> XXteigen

$values

[1] 321.0680142 230.1705247 12.7021105 3.9371614 0.1221892

$vectors

[,1] [,2] [,3] [,4] [,5]

[1,] -0.54225536 -0.06499573 0.82161708 -0.10574661 -0.12448979

[2,] -0.10181247 0.59346055 -0.11255162 -0.78812338 0.06026999

[3,] -0.52495325 -0.05938171 -0.21296861 0.11574223 0.81372354

[4,] -0.64487038 -0.07040626 -0.50874368 0.05990271 -0.56282918

[5,] -0.06449519 0.79692967 0.09000966 0.59219473 -0.04412631

> U = XXteigen$vectors

> D = diag(sqrt(XXteigen$values))

> U

[,1] [,2] [,3] [,4] [,5]

[1,] -0.54225536 -0.06499573 0.82161708 -0.10574661 -0.12448979

[2,] -0.10181247 0.59346055 -0.11255162 -0.78812338 0.06026999

[3,] -0.52495325 -0.05938171 -0.21296861 0.11574223 0.81372354

[4,] -0.64487038 -0.07040626 -0.50874368 0.05990271 -0.56282918

[5,] -0.06449519 0.79692967 0.09000966 0.59219473 -0.04412631

Notice that the matrix obtained is the “same” with the exception that some of the columns of are negated from those obtained from the SVD. This is presents a problem when checking that . This happens because if is an eigenvector of a matrix then so is .  
> D

[,1] [,2] [,3] [,4] [,5]

[1,] 17.91837 0.00000 0.000000 0.000000 0.0000000

[2,] 0.00000 15.17137 0.000000 0.000000 0.0000000

[3,] 0.00000 0.00000 3.564002 0.000000 0.0000000

[4,] 0.00000 0.00000 0.000000 1.984228 0.0000000

[5,] 0.00000 0.00000 0.000000 0.000000 0.3495557

> XtXeigen = eigen(XtX)

> XtXeigen

$values

[1] 321.0680142 230.1705247 12.7021105 3.9371614 0.1221892

$vectors

[,1] [,2] [,3] [,4] [,5]

[1,] -0.46461713 -0.02150651 0.86850856 0.0007995538 0.1713494

[2,] -0.07008599 0.75998796 -0.06307148 -0.6013456703 0.2278412

[3,] -0.73509354 -0.09879712 -0.28400852 -0.2234845728 -0.5650402

[4,] -0.48439167 -0.02544740 -0.39886566 0.3326838129 0.7035231

[5,] -0.06496983 0.64151954 0.04427431 0.6912010359 -0.3232839

> V = XtXeigen$vectors

> V

[,1] [,2] [,3] [,4] [,5]

[1,] -0.46461713 -0.02150651 0.86850856 0.0007995538 0.1713494

[2,] -0.07008599 0.75998796 -0.06307148 -0.6013456703 0.2278412

[3,] -0.73509354 -0.09879712 -0.28400852 -0.2234845728 -0.5650402

[4,] -0.48439167 -0.02544740 -0.39886566 0.3326838129 0.7035231

[5,] -0.06496983 0.64151954 0.04427431 0.6912010359 -0.3232839

Again we notice that the matrix obtained is the “same” with the exception that some of the columns of are negated from those obtained from the SVD. As stated above this happens because if is an eigenvector of a matrix then so is .

Thus when conducting a SVD in R, we use the svd function in R rather than doing the eigenanalyses of and directly, which actually is far easier anyway.

**Approximating the Original Data Matrix (i.e. reducing dimension using PCA)**

As the first two or three singular values are much larger than the others we can use these singular values along with their associated eigenvectors in to approximate . Below I show the approximation to based upon 2 and 3 singular values and the associated eigenvectors.

> Xsvd

$d

[1] 17.9183709 15.1713719 3.5640020 1.9842282 0.3495557

$u

[,1] [,2] [,3] [,4] [,5]

[1,] -0.54225536 0.06499573 0.82161708 0.10574661 -0.12448979

[2,] -0.10181247 -0.59346055 -0.11255162 0.78812338 0.06026999

[3,] -0.52495325 0.05938171 -0.21296861 -0.11574223 0.81372354

[4,] -0.64487038 0.07040626 -0.50874368 -0.05990271 -0.56282918

[5,] -0.06449519 -0.79692967 0.09000966 -0.59219473 -0.04412631

$v

[,1] [,2] [,3] [,4] [,5]

[1,] -0.46461713 0.02150651 -0.86850856 0.0007995538 -0.1713494

[2,] -0.07008599 -0.75998796 0.06307148 -0.6013456703 -0.2278412

[3,] -0.73509354 0.09879712 0.28400852 -0.2234845728 0.5650402

[4,] -0.48439167 0.02544740 0.39886566 0.3326838129 -0.7035231

[5,] -0.06496983 -0.64151954 -0.04427431 0.6912010359 0.3232839

Two dimensional approximation to

> U = Xsvd$u

> V = Xsvd$v

> D = diag(Xsvd$d)

> U2 = U[,1:2]

> D2 = diag(Xsvd$d[1:2])

> V2 = V[,1:2]

> Xapprox2 = U2%\*%D2%\*%t(V2)

> Xapprox2

[,1] [,2] [,3] [,4] [,5]

[1,] 4.5355816 -0.068425900 7.2398346 4.7316035 -0.001317455

[2,] 0.6539711 6.970494560 0.4515103 0.6545638 5.894517555

[3,] 4.3897067 -0.025424351 7.0035221 4.5792624 0.033179993

[4,] 5.3916357 -0.001942943 8.5995564 5.6243404 0.065482963

[5,] 0.2769094 9.269641750 -0.3449983 0.2521144 7.831384884

> X

[,1] [,2] [,3] [,4] [,5]

[1,] 2 0 8 6 0

[2,] 1 6 0 1 7

[3,] 5 0 7 4 0

[4,] 7 0 8 5 0

[5,] 0 10 0 0 7

> X – Xapprox2

[,1] [,2] [,3] [,4] [,5]

[1,] -2.5355816 0.068425900 0.760165410 1.2683965 0.001317455

[2,] 0.3460289 -0.970494560 -0.451510307 0.3454362 1.105482445

[3,] 0.6102933 0.025424351 -0.003522082 -0.5792624 -0.033179993

[4,] 1.6083643 0.001942943 -0.599556442 -0.6243404 -0.065482963

[5,] -0.2769094 0.730358250 0.344998304 -0.2521144 -0.831384884

Three dimensional approximation to

> U3 = U[,1:3]

> D3 = diag(Xsvd$d[1:3])

> V3 = V[,1:3]

> Xapprox3 = U3%\*%D3%\*%t(V3)

> Xapprox3

[,1] [,2] [,3] [,4] [,5]

[1,] 1.992375772 0.1162628 8.0714811 5.8995799 -0.1309635

[2,] 1.002359586 6.9451944 0.3375848 0.4945652 5.9122775

[3,] 5.048922544 -0.0732969 6.7879538 4.2765151 0.0667851

[4,] 6.966383726 -0.1163018 8.0846026 4.9011318 0.1457595

[5,] -0.001703482 9.2898747 -0.2538899 0.3800684 7.8171819

> X

[,1] [,2] [,3] [,4] [,5]

[1,] 2 0 8 6 0

[2,] 1 6 0 1 7

[3,] 5 0 7 4 0

[4,] 7 0 8 5 0

[5,] 0 10 0 0 7

> X - Xapprox3

[,1] [,2] [,3] [,4] [,5]

[1,] 0.007624228 -0.1162628 -0.07148109 0.10042011 0.1309635

[2,] -0.002359586 -0.9451944 -0.33758478 0.50543484 1.0877225

[3,] -0.048922544 0.0732969 0.21204622 -0.27651513 -0.0667851

[4,] 0.033616274 0.1163018 -0.08460256 0.09886822 -0.1457595

[5,] 0.001703482 0.7101253 0.25388990 -0.38006837 -0.8171819

We can see the three dimensional approximation to is definitely better than the two dimensional approximation. SVD can be used to reduce the dimensionality of data set and examining the components (rows and columns) of the and matrices can help gain insight to underlying structure in the data matrix . We will not dig into the interpretation details at this point. SVD can also be used in the process of filling in or imputing the missing values in a data matrix.

Example 4.1: Prehistoric Goblets (cont’d)

In the SVD discussion above we saw an example where we considered two lower dimensional approximations of a matrix by using the first few singular values along with the associated rows and columns of the and matrices. Below we consider a SVD of the scaled data matrix for the goblets.

> Xs = scale(Goblets)

> Xs.svd = svd(Xs)

> Xs.svd

$d

[1] 10.125348 5.119791 3.038741 1.848651 1.251818 1.023020

$u

[,1] [,2] [,3] [,4] [,5] [,6]

[1,] 0.046658409 -0.004997796 0.21249724 -0.232359182 -0.01428663 0.0422986277

[2,] 0.035077873 -0.145985775 -0.28514098 0.496016995 0.55733342 0.1545544675

[3,] 0.253642407 -0.001163343 -0.40913245 -0.322994897 0.20119584 -0.2564515420

… … … … … … …

[24,] -0.485651937 -0.043883993 -0.15528698 -0.311403790 -0.04857464 0.0933668908

[25,] 0.112829803 -0.309502241 -0.04800909 0.195893281 -0.16898695 0.0473918209

$v

[,1] [,2] [,3] [,4] [,5] [,6]

[1,] 0.3660233 0.48592912 -0.6179335 -0.32436829 0.27835629 0.2556581

[2,] 0.4515367 -0.03412653 0.3752732 -0.67427405 -0.08391876 -0.4386709

[3,] 0.4111609 -0.44135161 0.3163501 0.02019451 0.38254463 0.6239630

[4,] 0.4618586 -0.11457532 -0.1588367 0.54119094 0.38182563 -0.5564635

[5,] 0.2963653 0.68277080 0.4914536 0.35921044 -0.22136144 0.1625790

[6,] 0.4381125 -0.29768029 -0.3324080 0.13346207 -0.75785442 0.1295892

> Goblets.pc = princomp(Xs)

> Goblets.pc$loadings

Do you see any similarities between the principal component analysis on the correlation matrix and the SVD of the scaled data matrix ***X***?

Loadings:

Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6

MouthWidth -0.366 0.486 -0.618 0.324 -0.278 0.256

TotalWidth -0.452 0.375 0.674 -0.439

TotalHeight -0.411 -0.441 0.316 -0.383 0.624

BaseWidth -0.462 -0.115 -0.159 -0.541 -0.382 -0.556

StemWidth -0.296 0.683 0.491 -0.359 0.221 0.163

StemHeight -0.438 -0.298 -0.332 -0.133 0.758 0.130

> Goblets.pc2 = prcomp(Xs)

> Goblets.pc2$rotation

PC1 PC2 PC3 PC4 PC5 PC6

MouthWidth 0.366023 0.4859291 -0.617933 -0.3243683 0.2783563 0.255658

TotalWidth 0.451537 -0.0341265 0.375273 -0.6742740 -0.0839188 -0.438671

TotalHeight 0.411161 -0.4413516 0.316350 0.0201945 0.3825446 0.623963

BaseWidth 0.461859 -0.1145753 -0.158837 0.5411909 0.3818256 -0.556463

StemWidth 0.296365 0.6827708 0.491454 0.3592104 -0.2213614 0.162579

StemHeight 0.438112 -0.2976803 -0.332408 0.1334621 -0.7578544 0.129589

We see that the columns of the matrix are the same as the PCA loadings, modulo the sign. Thus they are essentially equivalent. Using the SVD approach for conducting a PCA has advantages in certain contexts. The example below will illustrate a situation where the SVD approach is much preferred.

There are two basic functions that perform principal components analysis in R, princomp uses the eigenanalysis approach and prcomp uses the SVD approach, as the above examples shows.

One very important difference between the two is that the princomp function will give you an error when , whereas the prcomp function will not.

Example 4.2 – Image Compression and Processing

Consider the following pixel image.

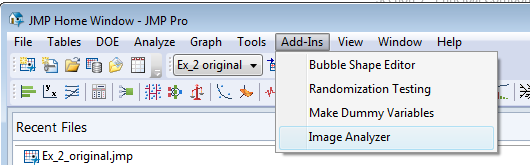
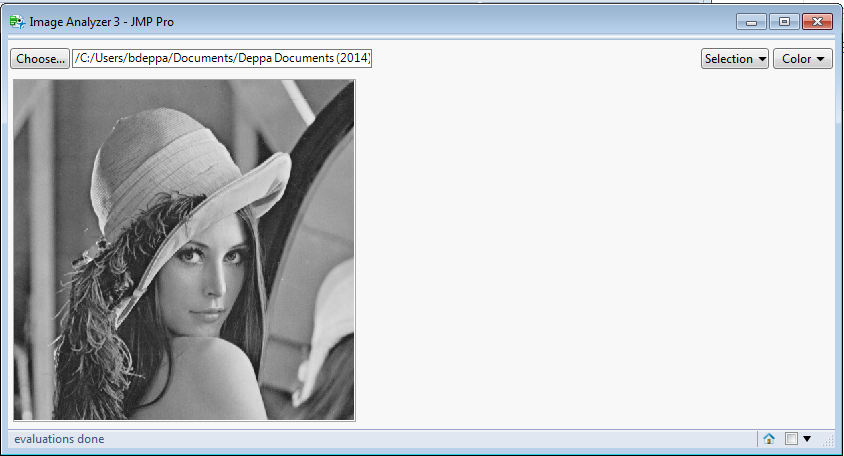


The information contained in this image can be stored in a data matrix where the elements of the matrix contain the gray scale intensity of the pixel. The intensity is value between 0 (black) and 1 (white), thus intensities closer 1 indicate lighter pixels and those closer to 0 are dark pixels. We can perform a singular value decomposition of the scaled intensity matrix to produce a lower dimensional (less detailed) version of the image. This is a form of *digital image compression*.

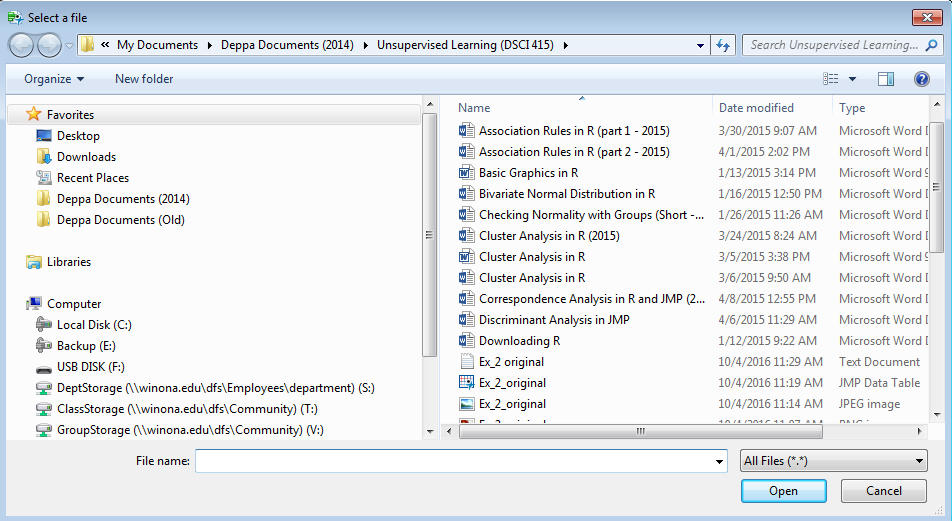
Although there are R packages that allow us to read in image files directly we will use the *JMP Image Analyzer* add-in script to read in the image file. You can down it here:

<https://community.jmp.com/docs/DOC-7181> Use **File > Open** to navigate to the add-in file once you have downloaded it.

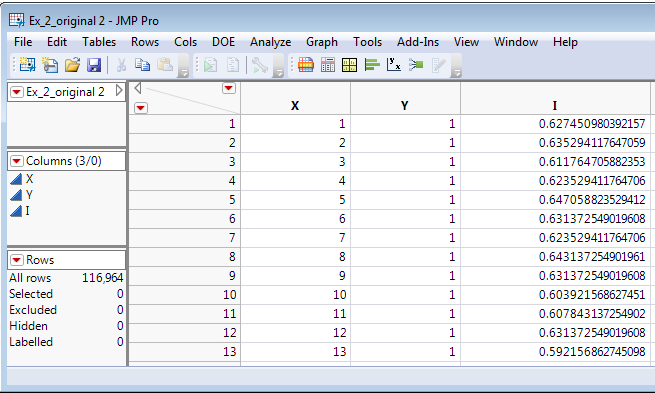
Once you have an image file in .JPEG format you can then select Image Analyzer from the Add-In drop-down menu to open any .JPEG or .jpg image file.

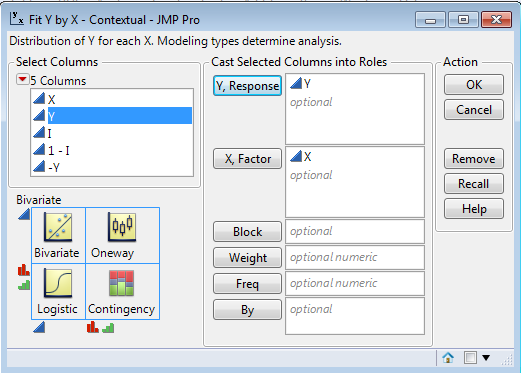
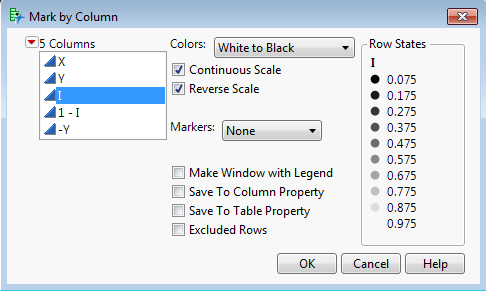
Choose the .JPEG image file.



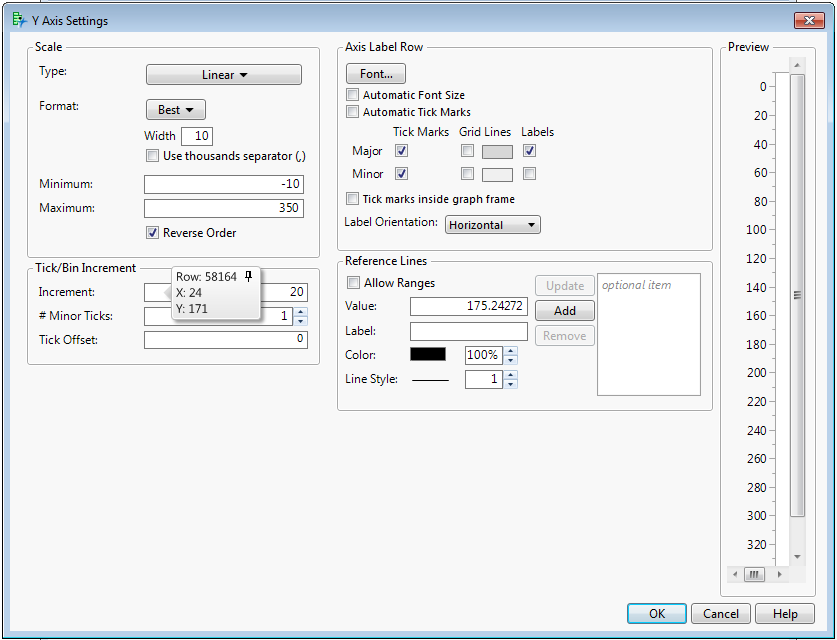
A portion of the resulting file is shown below:

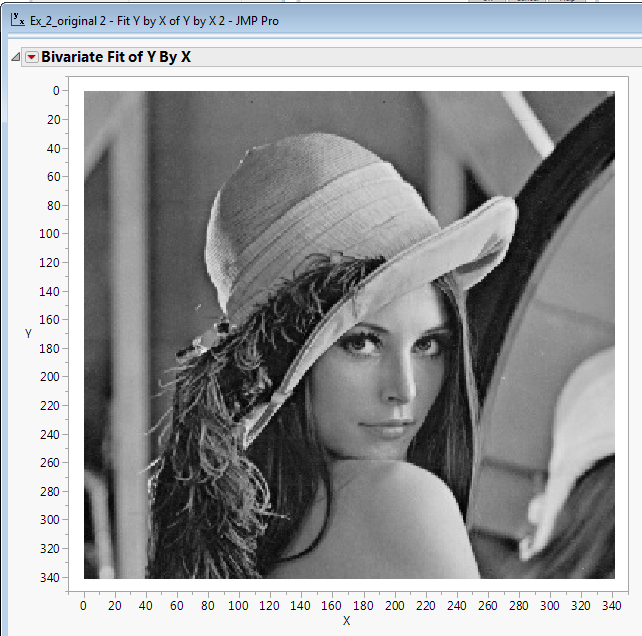


We can use **Analyze > Fit Y by X** to view the image by color coding the pixels using **Rows > Color or Marker by Column**.

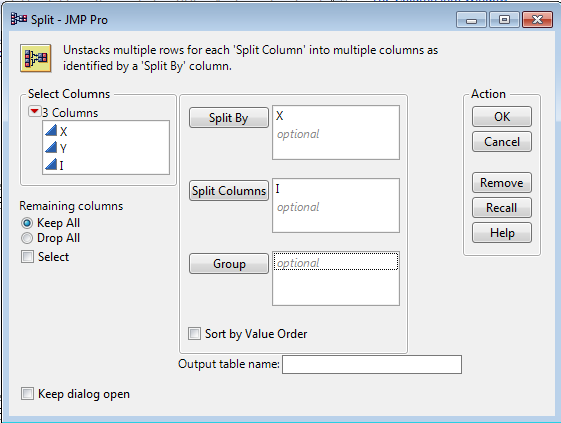
We can reverse the Y-axis by right-clicking on it and selecting **Reverse Order**.

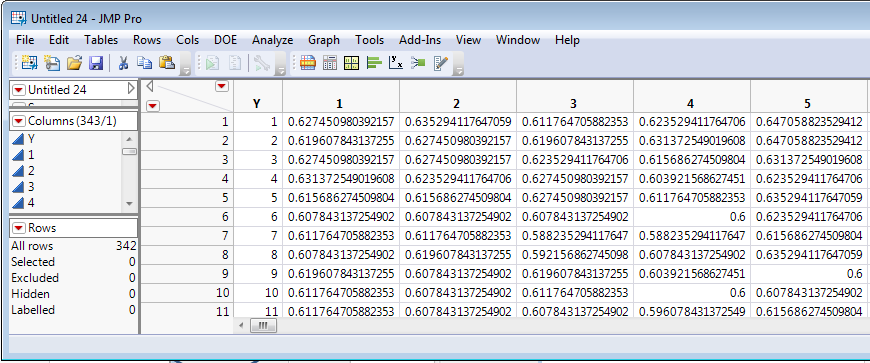


This was a lot of effort, but it is worthwhile in the sense we now understand how a digital image is really just a data matrix with numeric variables representing the details for each pixel. If we had a color image we can measure the Red, Green, and Blue (RGB) intensities of the cells along with Intensity, Hue, Lightness and Saturation. Thus for each pixel in the image we obtain seven numeric values. Thus image information would need to be stored in an array (i.e. multiple matrices) consisting of seven separate matrices, one for each pixel attribute.

We can then use the **Tables > Split** command to turn the image into a data matrix.

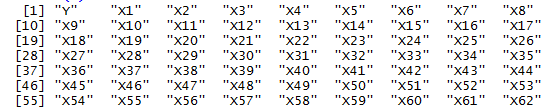


The resulting image matrix can then be read into R after saving it as .csv or .txt file using comma delimiters.



> Ximage = read.table(file.choose(),header=T,sep=”,”)

> names(Ximage)

  
> X = as.matrix(Ximage[,-1])



> x = seq(1:342)

> y = seq(1:342)

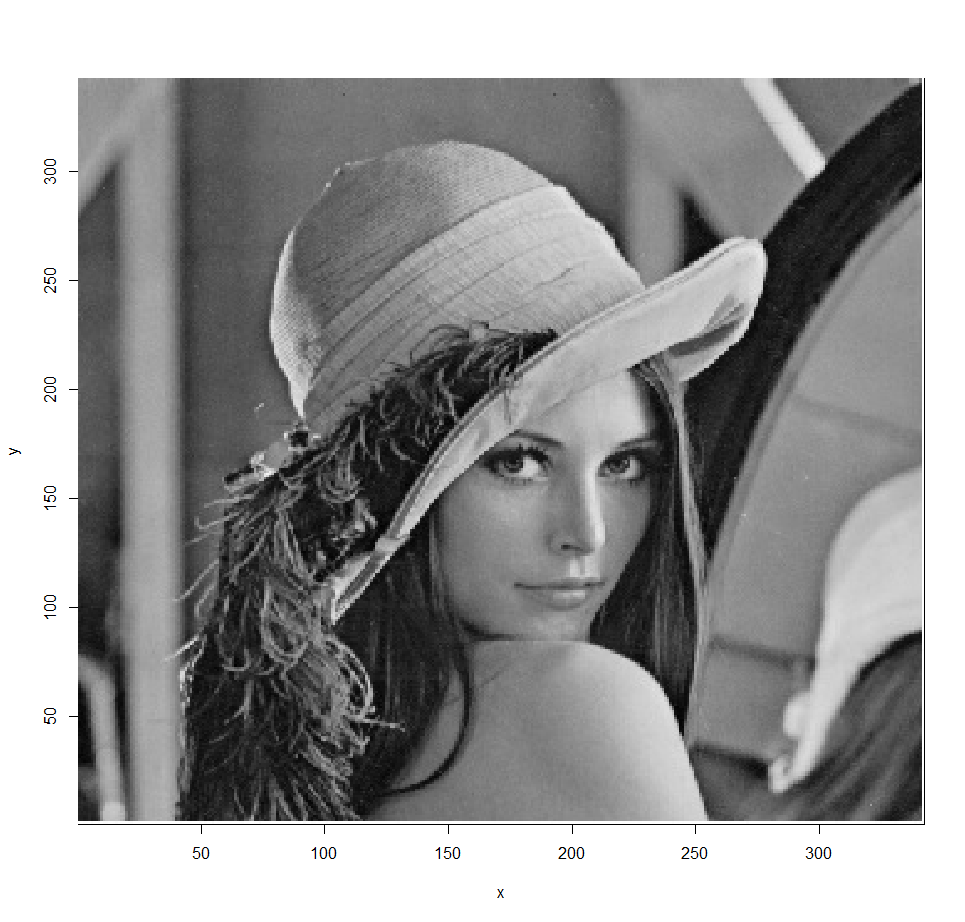
> image(x,y,X,col=gray((0:341)/341))

SIDEWAYS!! Ugh….

This can be fixed using the commands below, details of which are not all that important.

> X = apply(X,2,rev)

> image(x,y,t(X),col=gray((0:341)/341))



**Reading the Image directly into R (without using JMP)**

The **png** and **Rcurl** packages can be used to read in an image file into R and convert the image into a n x n matrix of pixel intensities. (Note: A \*.png image file is being read in here, in JMP we used a \*.jpg file type).

library(png)

library(RCurl)

myurl = "http://www.statsclass.org/dsci415/Datasets/Girl\_Image.png"

girl\_image = readPNG(getURLContent(myurl))

girl\_mat=girl\_image[,,1] # will hold the grayscale values

> dim(girl\_mat)

[1] 342 342

girl\_mat =apply(girl\_mat, 2, rev) # otherwise the image will be rotated

image(t(girl\_mat), col = gray((0:341)/341)) # plot in grayscale

A person wearing a hat

Description automatically generated with medium confidence

We will now consider using SVD to compress the detail of the image using the same procedure we did with the toy SVD example above.

> Xsvd = svd(X)

> Xsvd$d

[1] 1.699e+02 2.771e+01 2.149e+01 1.692e+01 1.540e+01 1.451e+01

[7] 1.201e+01 1.074e+01 8.851e+00 8.296e+00 6.990e+00 6.396e+00

[13] 6.245e+00 6.099e+00 5.740e+00 5.602e+00 5.485e+00 4.854e+00

[19] 4.580e+00 4.448e+00 4.328e+00 4.133e+00 3.804e+00 3.526e+00

[25] 3.486e+00 3.411e+00 3.214e+00 3.118e+00 3.040e+00 2.989e+00

[31] 2.793e+00 2.757e+00 2.717e+00 2.707e+00 2.636e+00 2.589e+00

First 100 singular values of the SVD decomposition of the image matrix. The first singular value is 169.9 and the second is 27.7, etc.

[37] 2.534e+00 2.505e+00 2.429e+00 2.402e+00 2.334e+00 2.232e+00

[43] 2.221e+00 2.175e+00 2.120e+00 2.075e+00 2.004e+00 1.990e+00

[49] 1.950e+00 1.921e+00 1.889e+00 1.842e+00 1.798e+00 1.777e+00

[55] 1.737e+00 1.690e+00 1.676e+00 1.665e+00 1.642e+00 1.610e+00

[61] 1.580e+00 1.567e+00 1.541e+00 1.511e+00 1.485e+00 1.435e+00

[67] 1.408e+00 1.374e+00 1.367e+00 1.359e+00 1.316e+00 1.275e+00

[73] 1.256e+00 1.240e+00 1.227e+00 1.202e+00 1.174e+00 1.167e+00

[79] 1.141e+00 1.132e+00 1.120e+00 1.103e+00 1.093e+00 1.075e+00

[85] 1.058e+00 1.033e+00 1.024e+00 1.013e+00 1.002e+00 9.871e-01

[91] 9.848e-01 9.606e-01 9.526e-01 9.402e-01 9.111e-01 9.053e-01

[97] 8.872e-01 8.810e-01 8.646e-01 8.508e-01

> dim(Xsvd$u)

[1] 342 342

> dim(Xsvd$v)

[1] 342 342

By extracting the singular values () and their corresponding vectors in and we can construct a lower dimensional representation of the image, i.e. a compressed image.

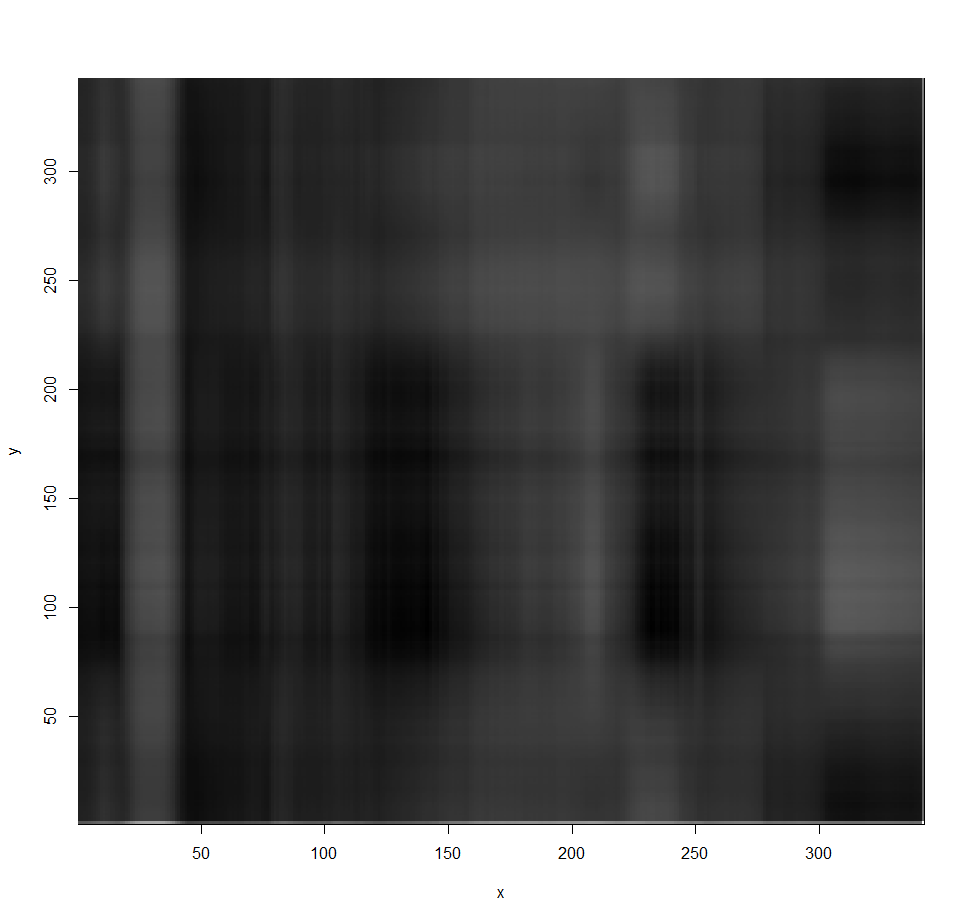
k = 2

> D2 = diag(Xsvd$d[1:2])

> U2 = Xsvd$u[,1:2]

> V2 = Xsvd$v[,1:2]

> Xapprox2 = U2%\*%D2%\*%t(V2)

> image(x,y,t(Xapprox2),col=gray((0:341)/341))  


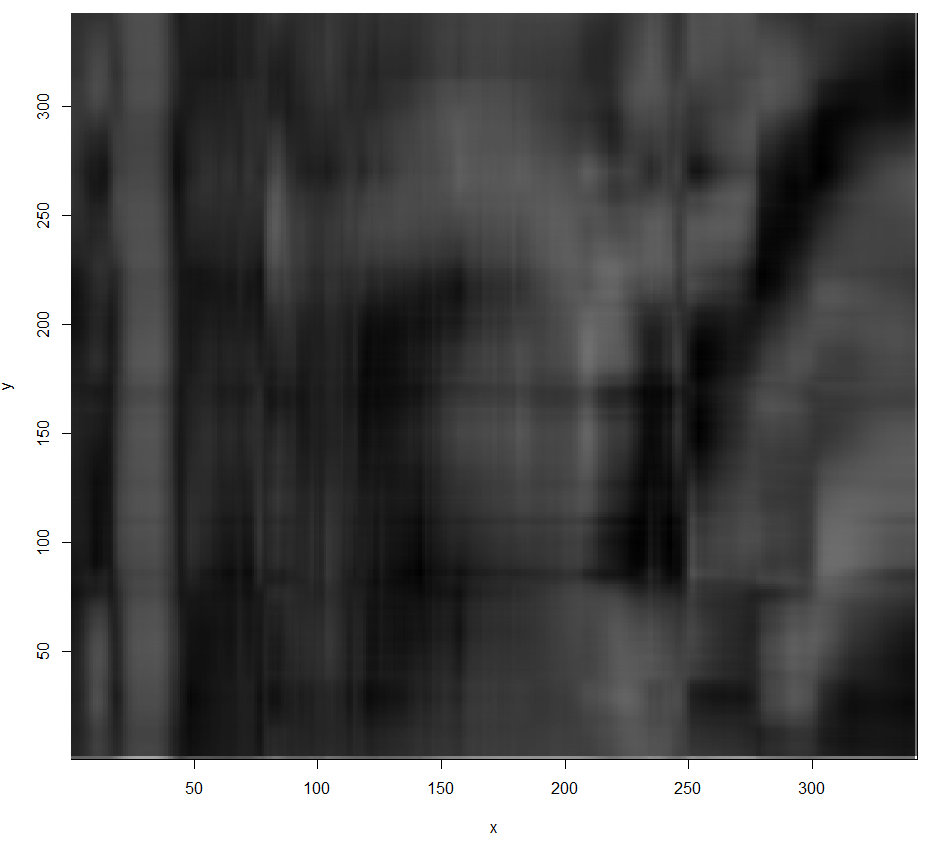
k = 5

> D5 = diag(Xsvd$d[1:5])

> U5 = Xsvd$u[,1:5]

> V5 = Xsvd$v[,1:5]

> Xapprox5 = U5%\*%D5%\*%t(V5)

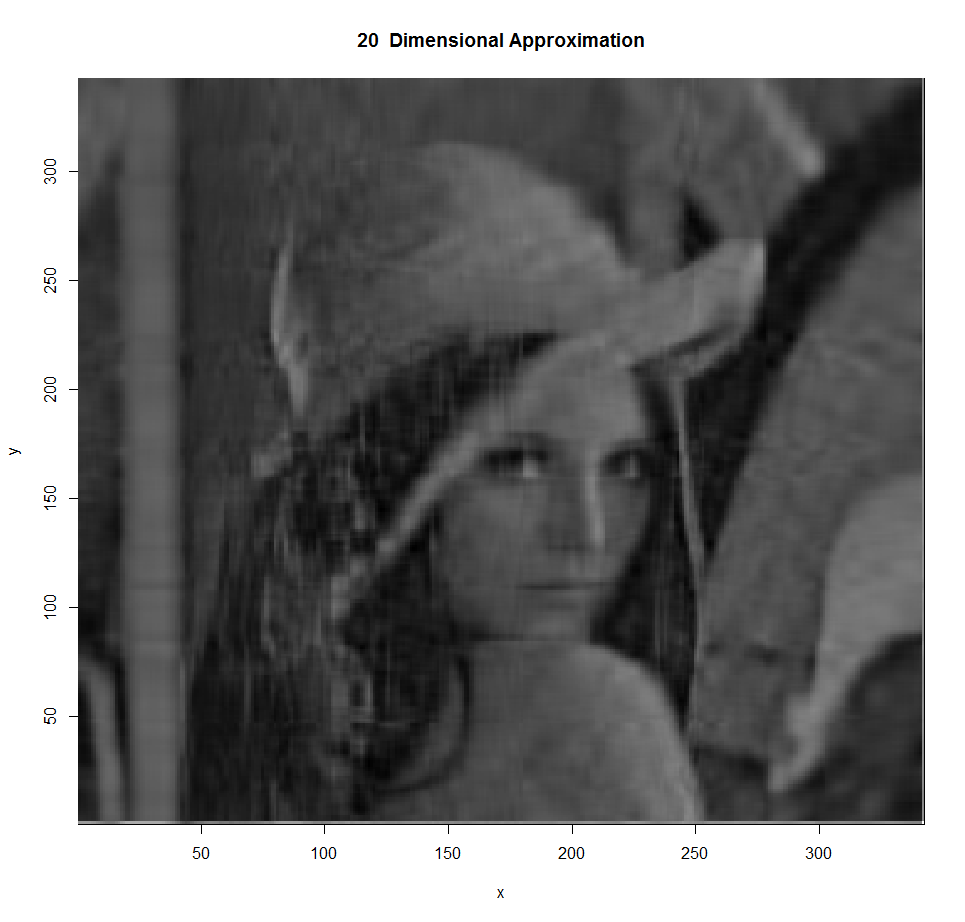
> image(x,y,t(Xapprox5),col=gray((0:341)/341))  


This is rather tedious as the commands for the and dimensional representations were essentially identical except for the number of dimensions to extract. We can easily write function that does all of this for a given .

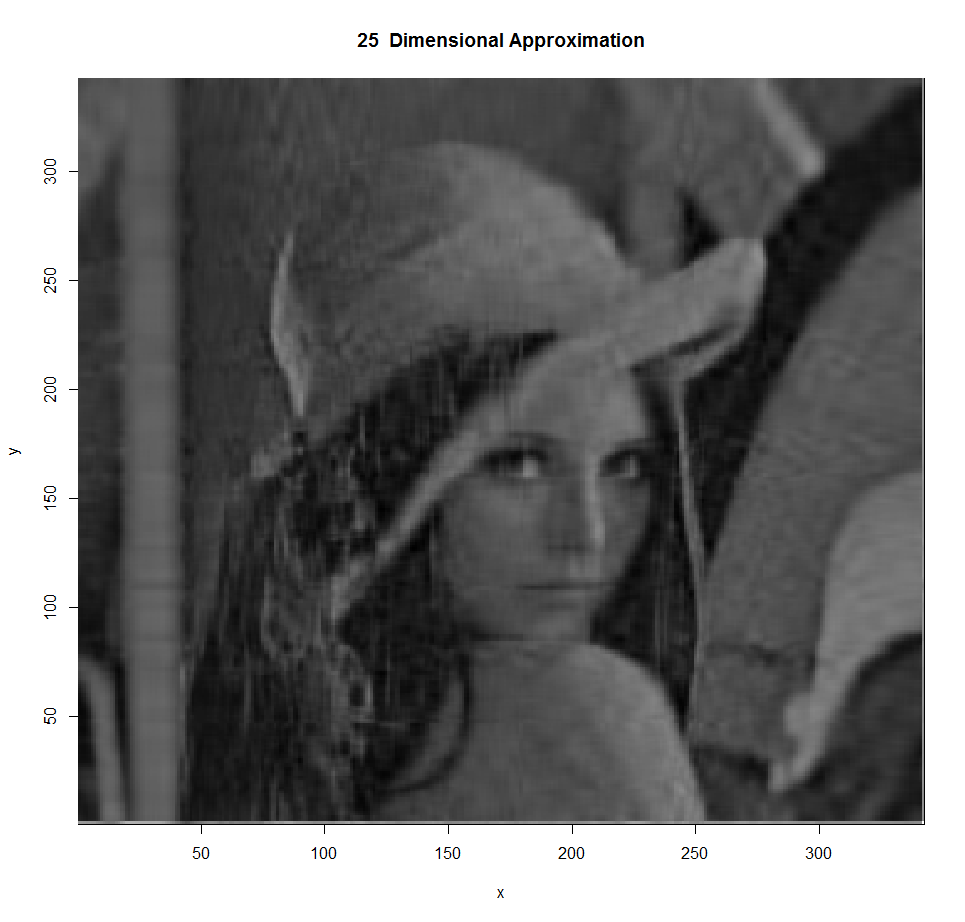
|  |
| --- |
| Custom R Function |
| image.svd = function(Xsvd,k=10){  Dk = diag(Xsvd$d[1:k])  Uk = Xsvd$u[,1:k]  Vk = Xsvd$v[,1:k]  Xk = Uk%\*%Dk%\*%t(Vk)  p = dim(Xsvd$u)[1]  x = seq(1:p)  y = x  image(x,y,t(Xk),col=gray((0:(p-1))/(p-1)))  title(paste(k," Dimensional Approximation"))  } |

k = 20

> image.svd(Xsvd,k=20)

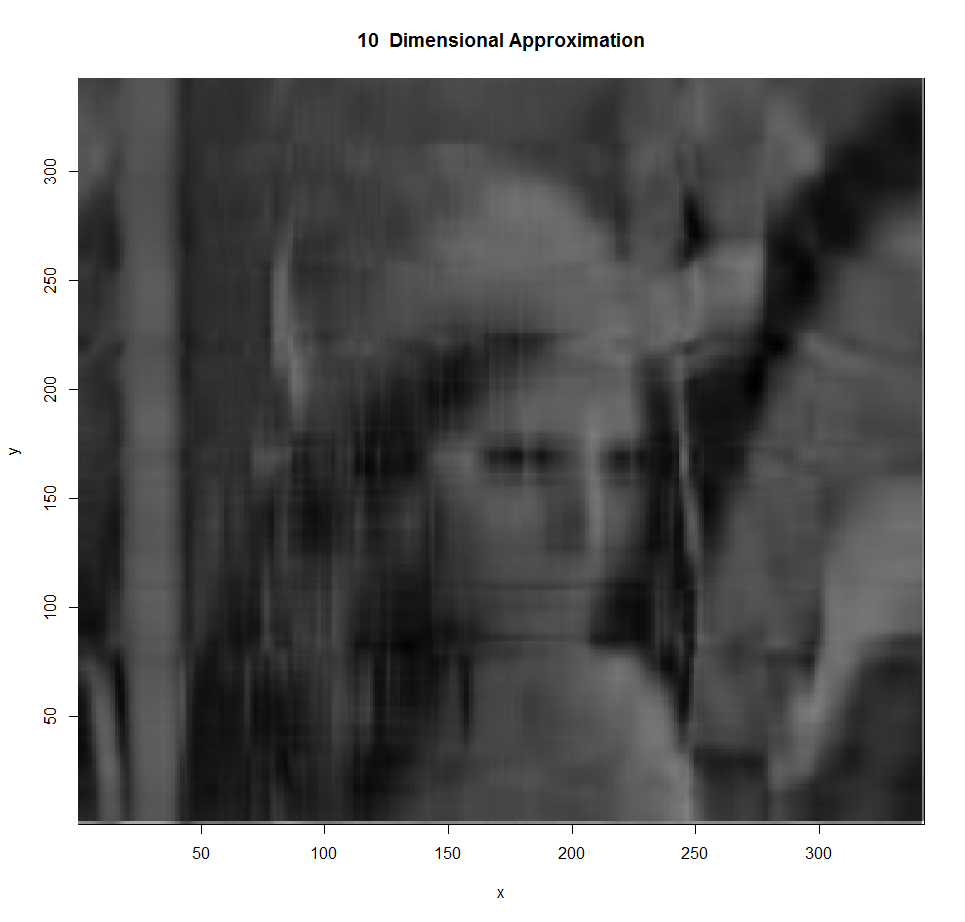


k = 25  
> image.svd(Xsvd,k=25)



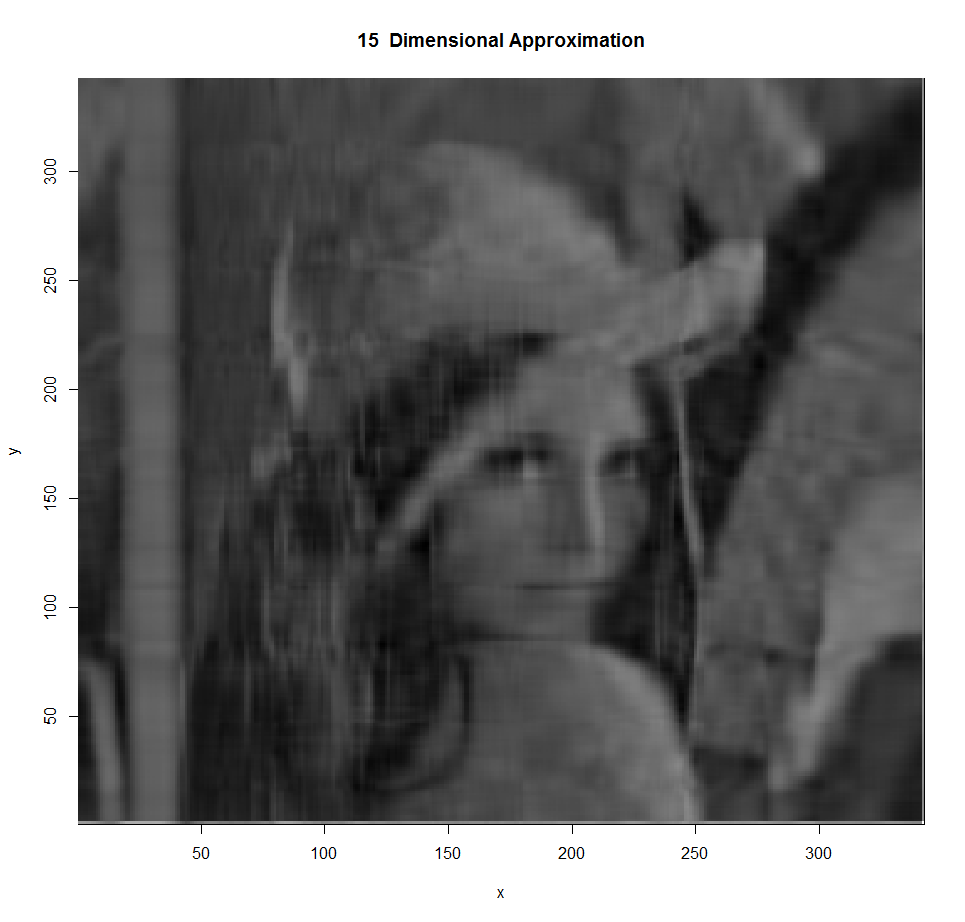
k = 10

> image.svd(Xsvd,k=10)



k = 15

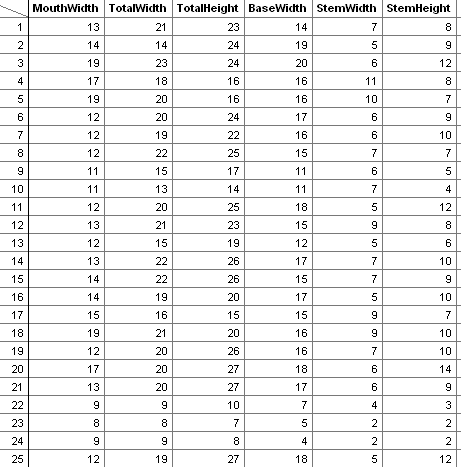
> image.svd(Xsvd,k=15)



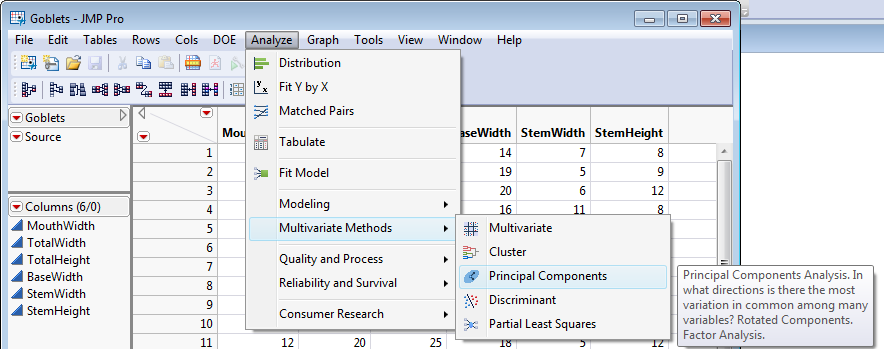
Etc…

**4.5 – Performing PCA in JMP**

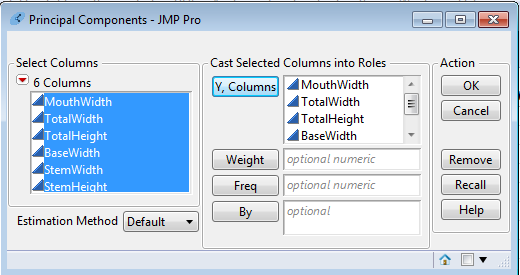
Example 3.1: Prehistoric Goblets  
Again using the goblet data in JMP (**Goblets.csv**).



Select **Analyze > Multivariate Methods > Principal Components.**

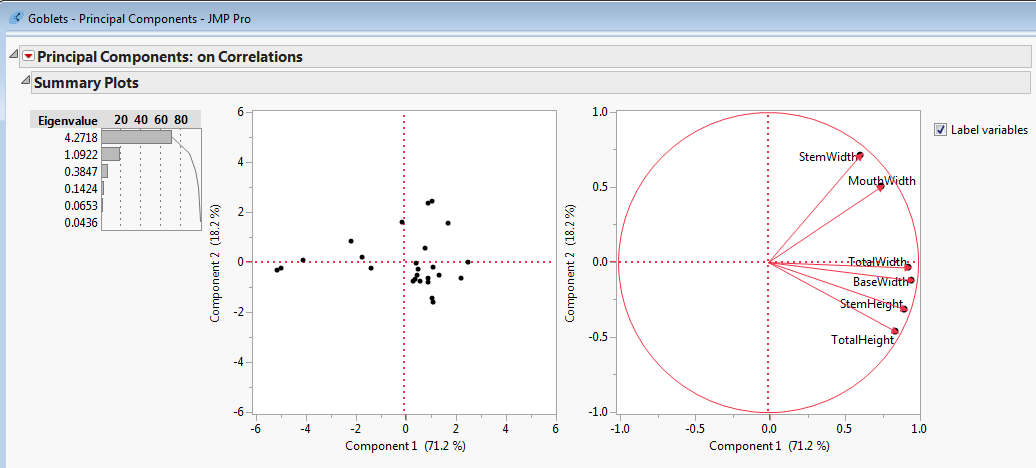


Place all variables to be included in your principal component analysis into the , Columns box. Click OK.

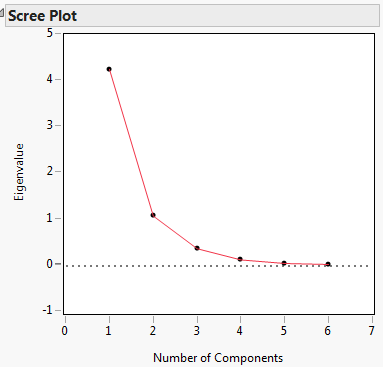


The default output is the eigenvalues, the percentage of total variation due to each principal component and the cumulative percentages.

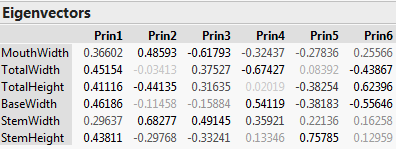
**Comment**: The default PC analysis in JMP uses the correlation matrix (i.e. the standardized variables).



The scree plot can be obtained from the drop-down menu.

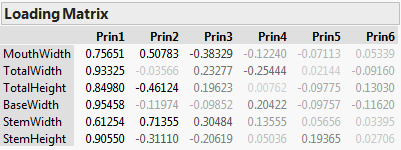


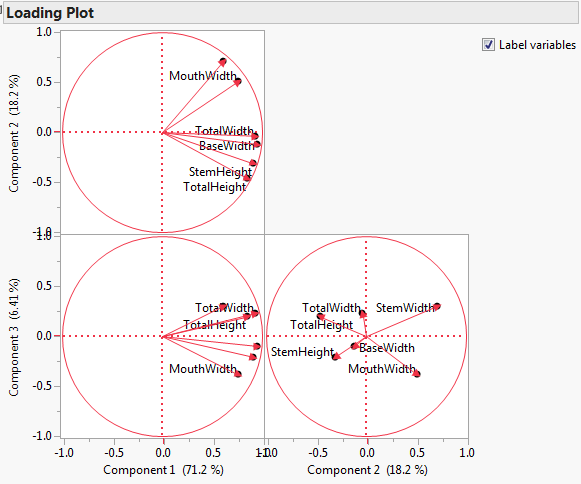
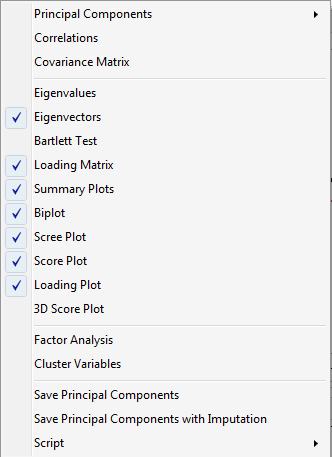
Likewise, the eigenvectors can be obtained from the red drop-down menu.

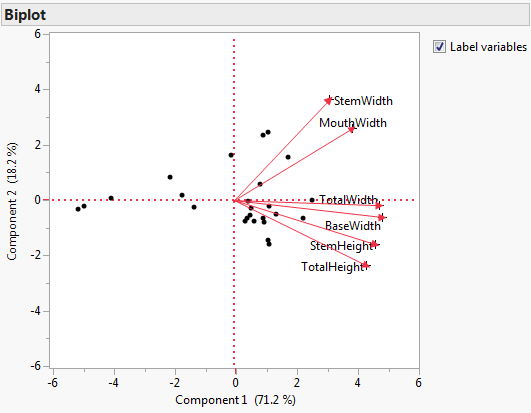


**Comment:**  The eigenvectors presented here are of the opposite sign of those computed by R. This has little effect on our interpretation of the principal components. For example, the second principal component is again contrasting the goblet’s width (stem and mouth) with its total height.

If you select Loading Matrix from the Principal Components drop-down menu you obtain the following:



These values are different than the eigenvectors that define the principal components but certainly share some similarities. These the pairwise correlations of each of the original with the principal component scores for each PC. These correlations give an alternative way to interpret what features of these data the different principal components are measuring.  
 



The Loading Plot shows the loading matrix values for two principal components at a time. The biplots combine the loading plots with a plot of the principal component scores () on the same graph.