Principal Components Analysis

YOUR NAME

In this lab, we perform PCA on the `USArrests` data set, which is part of the base `R` package. The rows of the data set contain the 50 states, in alphabetical order.

```
states <- row.names(USArrests)
states
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

The columns of the data set contain the four variables.

```
names(USArrests)
```

```
## [1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

We first briefly examine the data. We notice that the variables have vastly different means.

```
apply(USArrests, 2, mean)
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

Note that the `apply()` function allows us to apply a function—in this case, the `mean()` function—to each row or column of the data set. The second input here denotes whether we wish to compute the mean of the rows, 1, or the columns, 2. We see that there are on average three times as many rapes as murders, and more than eight times as many assaults as rapes.

We can also examine the variances of the four variables using the `apply()` function.

```
apply(USArrests, 2, var)
```

```
##     Murder    Assault   UrbanPop       Rape
##   18.97047 6945.16571  209.51878   87.72916
```

Not surprisingly, the variables also have vastly different variances: the `UrbanPop` variable measures the percentage of the population in each state living in an urban area, which is not a comparable number to the number of rapes in each state per 100,000 individuals. If we failed to scale the variables before performing PCA, then most of the principal components that we observed would be driven by the `Assault` variable, since it has by far the largest mean and variance. Thus, it is important to standardize the variables to have mean zero and standard deviation one before performing PCA.

We now perform principal components analysis using the `prcomp()` function, which is one of several functions in `R` that perform PCA.

```
pr.out <- prcomp(USArrests, scale = TRUE)
```

By default, the `prcomp()` function centers the variables to have mean zero. By using the option `scale = TRUE`, we scale the variables to have standard deviation one. The output from `prcomp()` contains a number of useful quantities.

```
names(pr.out)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

The `center` and `scale` components correspond to the means and standard deviations of the variables that were used for scaling prior to implementing PCA.

```
pr.out$center
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

```
pr.out$scale
```

```
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

The `rotation` matrix provides the principal component loadings; each column of `pr.out$rotation` contains the corresponding principal component loading vector.\footnote{This function names it the rotation matrix, because when we matrix-multiply the $\mathbf{X}$ matrix by `pr.out$rotation`, it gives us the coordinates of the data in the rotated coordinate system. These coordinates are the principal component scores.}

```
pr.out$rotation
```

```
##                 PC1        PC2        PC3         PC4
## Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
## Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
## UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
## Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

We see that there are four distinct principal components. This is to be expected because there are in general $\min(n-1, p)$ informative principal components in a data set with n observations and p variables.

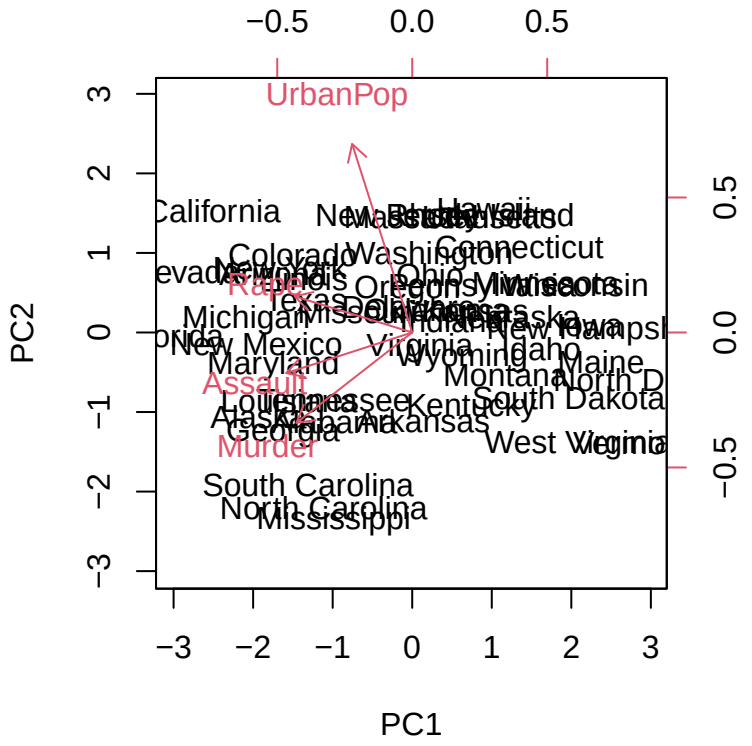
Using the `prcomp()` function, we do not need to explicitly multiply the data by the principal component loading vectors in order to obtain the principal component score vectors. Rather the 50×4 matrix `x` has as its columns the principal component score vectors. That is, the kth column is the kth principal component score vector.

```
dim(pr.out$x)
```

```
## [1] 50  4
```

We can plot the first two principal components as follows:

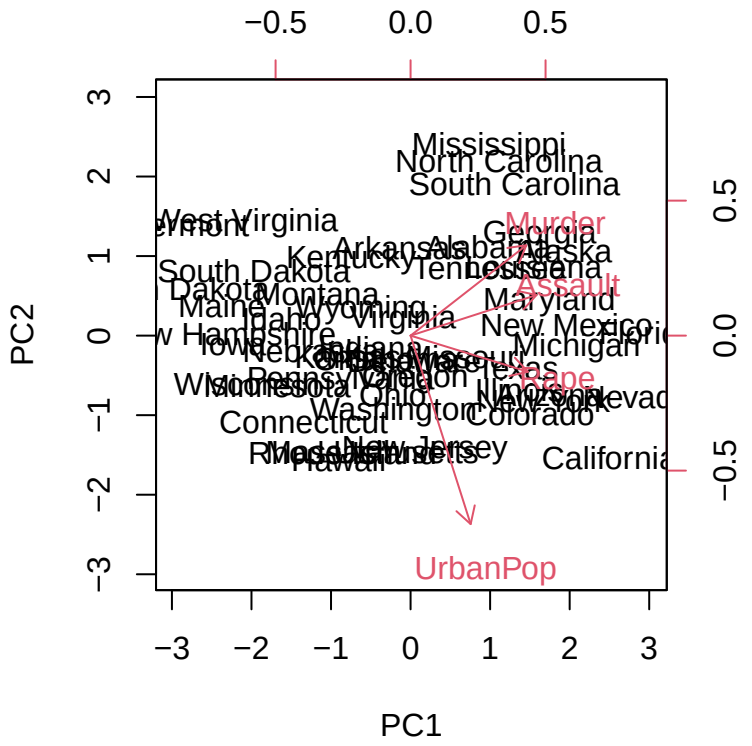
```
biplot(pr.out, scale = 0)
```



The `scale = 0` argument to `biplot()` ensures that the arrows are scaled to represent the loadings; other values for `scale` give slightly different biplots with different interpretations.

Notice that this figure is a mirror image of Figure 12.1. Recall that the principal components are only unique up to a sign change, so we can reproduce Figure 12.1 by making a few small changes:

```
pr.out$rotation = -pr.out$rotation
pr.out$x = -pr.out$x
biplot(pr.out, scale = 0)
```



The `prcomp()` function also outputs the standard deviation of each principal component. For instance, on the `USArrests` data set, we can access these standard deviations as follows:

```
pr.out$sdev
```

```
## [1] 1.5748783 0.9948694 0.5971291 0.4164494
```

The variance explained by each principal component is obtained by squaring these:

```
pr.var <- pr.out$sdev^2
pr.var
```

```
## [1] 2.4802416 0.9897652 0.3565632 0.1734301
```

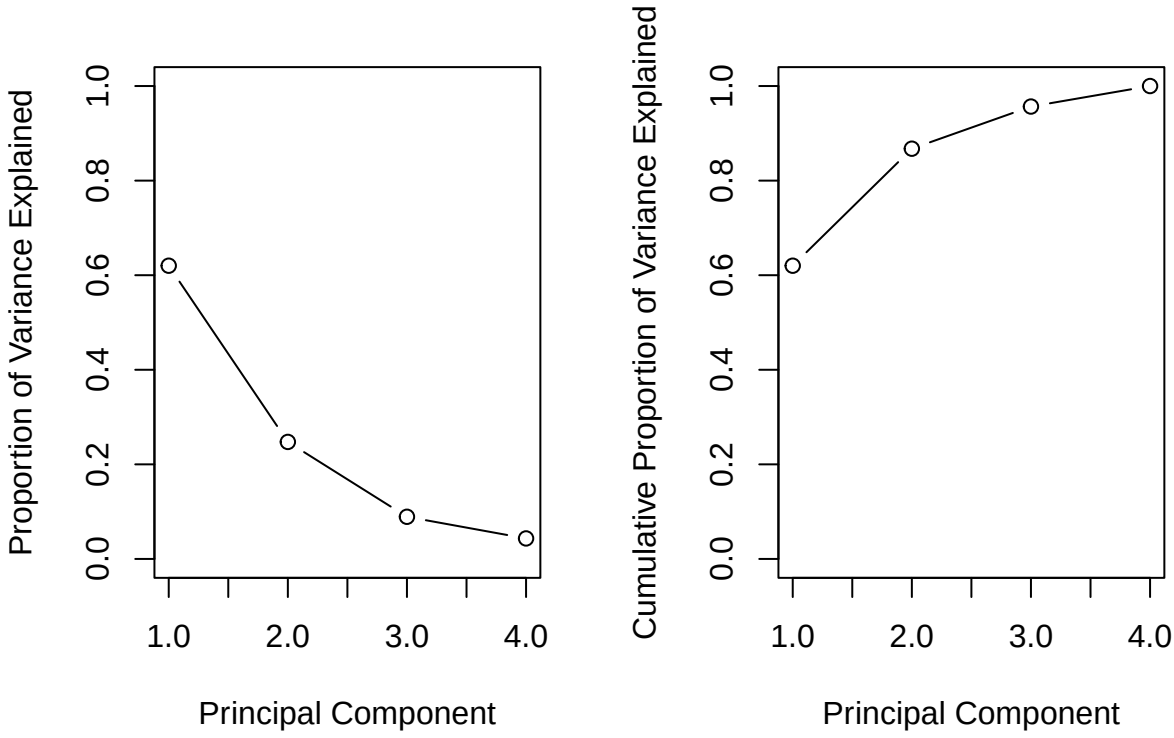
To compute the proportion of variance explained by each principal component, we simply divide the variance explained by each principal component by the total variance explained by all four principal components:

```
pve <- pr.var / sum(pr.var)
pve
```

```
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

We see that the first principal component explains 62.0% of the variance in the data, the next principal component explains 24.7% of the variance, and so forth. We can plot the PVE explained by each component, as well as the cumulative PVE, as follows:

```
par(mfrow = c(1, 2))
plot(pve, xlab = "Principal Component",
    ylab = "Proportion of Variance Explained", ylim = c(0, 1),
    type = "b")
plot(cumsum(pve), xlab = "Principal Component",
    ylab = "Cumulative Proportion of Variance Explained",
    ylim = c(0, 1), type = "b")
```



The result is shown in Figure 12.3. Note that the function `cumsum()` computes the cumulative sum of the elements of a numeric vector. For instance:

```
a <- c(1, 2, 8, -3)
cumsum(a)
```

```
## [1]  1  3 11  8
```

Matrix Completion

We now re-create the analysis carried out on the `USArrests` data in Section 12.3. We turn the data frame into a matrix, after centering and scaling each column to have mean zero and variance one.

```
X <- data.matrix(scale(USArrests))
pcob <- prcomp(X)
summary(pcob)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4
## Standard deviation     1.5749 0.9949 0.59713 0.41645
## Proportion of Variance 0.6201 0.2474 0.08914 0.04336
## Cumulative Proportion  0.6201 0.8675 0.95664 1.00000
```

We see that the first principal component explains 62% of the variance.

We saw in Section 12.2.2 that solving the optimization problem (12.6) on a centered data matrix $\mathbf{X}$ is equivalent to computing the first M principal components of the data. The *singular value decomposition* (SVD) is a general algorithm for solving (12.6).

```
sX <- svd(X)
names(sX)
```

```
## [1] "d" "u" "v"
```

```
round(sX$v, 3)
```

```
##        [,1]   [,2]   [,3]   [,4]
## [1,] -0.536 -0.418  0.341  0.649
## [2,] -0.583 -0.188  0.268 -0.743
## [3,] -0.278  0.873  0.378  0.134
## [4,] -0.543  0.167 -0.818  0.089
```

The `svd()` function returns three components, `u`, `d`, and `v`. The matrix `v` is equivalent to the loading matrix from principal components (up to an unimportant sign flip).

```
pcob$rotation
```

```
##                  PC1        PC2        PC3         PC4
## Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
## Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
## UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
## Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

The matrix `u` is equivalent to the matrix of *standardized* scores, and the standard deviations are in the vector `d`. We can recover the score vectors using the output of `svd()`. They are identical to the score vectors output by `prcomp()`.

```
t(sX$d * t(sX$u))
```

```
##              [,1]        [,2]        [,3]         [,4]
##  [1,] -0.97566045 -1.12200121  0.43980366  0.154696581
##  [2,] -1.93053788 -1.06242692 -2.01950027 -0.434175454
##  [3,] -1.74544285  0.73845954 -0.05423025 -0.826264240
##  [4,]  0.13999894 -1.10854226 -0.11342217 -0.180973554
##  [5,] -2.49861285  1.52742672 -0.59254100 -0.338559240
##  [6,] -1.49934074  0.97762966 -1.08400162  0.001450164
##  [7,]  1.34499236  1.07798362  0.63679250 -0.117278736
##  [8,] -0.04722981  0.32208890  0.71141032 -0.873113315
```

```
##  [9,] -2.98275967 -0.03883425  0.57103206 -0.095317042
## [10,] -1.62280742 -1.26608838  0.33901818  1.065974459
## [11,]  0.90348448  1.55467609 -0.05027151  0.893733198
## [12,]  1.62331903 -0.20885253 -0.25719021 -0.494087852
## [13,] -1.36505197  0.67498834  0.67068647 -0.120794916
## [14,]  0.50038122  0.15003926 -0.22576277  0.420397595
## [15,]  2.23099579  0.10300828 -0.16291036  0.017379470
## [16,]  0.78887206  0.26744941 -0.02529648  0.204421034
## [17,]  0.74331256 -0.94880748  0.02808429  0.663817237
## [18,] -1.54909076 -0.86230011  0.77560598  0.450157791
## [19,]  2.37274014 -0.37260865  0.06502225 -0.327138529
## [20,] -1.74564663 -0.42335704  0.15566968 -0.553450589
## [21,]  0.48128007  1.45967706  0.60337172 -0.177793902
## [22,] -2.08725025  0.15383500 -0.38100046  0.101343128
## [23,]  1.67566951  0.62590670 -0.15153200  0.066640316
## [24,] -0.98647919 -2.36973712  0.73336290  0.213342049
## [25,] -0.68978426  0.26070794 -0.37365033  0.223554811
## [26,]  1.17353751 -0.53147851 -0.24440796  0.122498555
## [27,]  1.25291625  0.19200440 -0.17380930  0.015733156
## [28,] -2.84550542  0.76780502 -1.15168793  0.311354436
## [29,]  2.35995585  0.01790055 -0.03648498 -0.032804291
## [30,] -0.17974128  1.43493745  0.75677041  0.240936580
## [31,] -1.96012351 -0.14141308 -0.18184598 -0.336121113
## [32,] -1.66566662  0.81491072  0.63661186 -0.013348844
## [33,] -1.11208808 -2.20561081  0.85489245 -0.944789648
## [34,]  2.96215223 -0.59309738 -0.29824930 -0.251434626
## [35,]  0.22369436  0.73477837  0.03082616  0.469152817
## [36,]  0.30864928  0.28496113  0.01515592  0.010228476
## [37,] -0.05852787  0.53596999 -0.93038718 -0.235390872
## [38,]  0.87948680  0.56536050  0.39660218  0.355452378
## [39,]  0.85509072  1.47698328  1.35617705 -0.607402746
## [40,] -1.30744986 -1.91397297  0.29751723 -0.130145378
## [41,]  1.96779669 -0.81506822 -0.38538073 -0.108470512
## [42,] -0.98969377 -0.85160534 -0.18619262  0.646302674
## [43,] -1.34151838  0.40833518  0.48712332  0.636731051
## [44,]  0.54503180  1.45671524 -0.29077592 -0.081486749
## [45,]  2.77325613 -1.38819435 -0.83280797 -0.143433697
## [46,]  0.09536670 -0.19772785 -0.01159482  0.209246429
## [47,]  0.21472339  0.96037394 -0.61859067 -0.218628161
## [48,]  2.08739306 -1.41052627 -0.10372163  0.130583080
## [49,]  2.05881199  0.60512507  0.13746933  0.182253407
## [50,]  0.62310061 -0.31778662  0.23824049 -0.164976866
```

```
pcob$x
```

```
##                       PC1         PC2         PC3          PC4
## Alabama        -0.97566045 -1.12200121  0.43980366  0.154696581
## Alaska         -1.93053788 -1.06242692 -2.01950027 -0.434175454
## Arizona        -1.74544285  0.73845954 -0.05423025 -0.826264240
## Arkansas        0.13999894 -1.10854226 -0.11342217 -0.180973554
## California     -2.49861285  1.52742672 -0.59254100 -0.338559240
## Colorado       -1.49934074  0.97762966 -1.08400162  0.001450164
## Connecticut     1.34499236  1.07798362  0.63679250 -0.117278736
## Delaware       -0.04722981  0.32208890  0.71141032 -0.873113315
```

```
## Florida        -2.98275967 -0.03883425  0.57103206 -0.095317042
## Georgia        -1.62280742 -1.26608838  0.33901818  1.065974459
## Hawaii          0.90348448  1.55467609 -0.05027151  0.893733198
## Idaho           1.62331903 -0.20885253 -0.25719021 -0.494087852
## Illinois       -1.36505197  0.67498834  0.67068647 -0.120794916
## Indiana         0.50038122  0.15003926 -0.22576277  0.420397595
## Iowa            2.23099579  0.10300828 -0.16291036  0.017379470
## Kansas          0.78887206  0.26744941 -0.02529648  0.204421034
## Kentucky        0.74331256 -0.94880748  0.02808429  0.663817237
## Louisiana      -1.54909076 -0.86230011  0.77560598  0.450157791
## Maine           2.37274014 -0.37260865  0.06502225 -0.327138529
## Maryland       -1.74564663 -0.42335704  0.15566968 -0.553450589
## Massachusetts   0.48128007  1.45967706  0.60337172 -0.177793902
## Michigan       -2.08725025  0.15383500 -0.38100046  0.101343128
## Minnesota       1.67566951  0.62590670 -0.15153200  0.066640316
## Mississippi    -0.98647919 -2.36973712  0.73336290  0.213342049
## Missouri       -0.68978426  0.26070794 -0.37365033  0.223554811
## Montana         1.17353751 -0.53147851 -0.24440796  0.122498555
## Nebraska        1.25291625  0.19200440 -0.17380930  0.015733156
## Nevada         -2.84550542  0.76780502 -1.15168793  0.311354436
## New Hampshire   2.35995585  0.01790055 -0.03648498 -0.032804291
## New Jersey     -0.17974128  1.43493745  0.75677041  0.240936580
## New Mexico     -1.96012351 -0.14141308 -0.18184598 -0.336121113
## New York       -1.66566662  0.81491072  0.63661186 -0.013348844
## North Carolina -1.11208808 -2.20561081  0.85489245 -0.944789648
## North Dakota    2.96215223 -0.59309738 -0.29824930 -0.251434626
## Ohio            0.22369436  0.73477837  0.03082616  0.469152817
## Oklahoma        0.30864928  0.28496113  0.01515592  0.010228476
## Oregon         -0.05852787  0.53596999 -0.93038718 -0.235390872
## Pennsylvania    0.87948680  0.56536050  0.39660218  0.355452378
## Rhode Island    0.85509072  1.47698328  1.35617705 -0.607402746
## South Carolina -1.30744986 -1.91397297  0.29751723 -0.130145378
## South Dakota    1.96779669 -0.81506822 -0.38538073 -0.108470512
## Tennessee      -0.98969377 -0.85160534 -0.18619262  0.646302674
## Texas          -1.34151838  0.40833518  0.48712332  0.636731051
## Utah            0.54503180  1.45671524 -0.29077592 -0.081486749
## Vermont         2.77325613 -1.38819435 -0.83280797 -0.143433697
## Virginia        0.09536670 -0.19772785 -0.01159482  0.209246429
## Washington      0.21472339  0.96037394 -0.61859067 -0.218628161
## West Virginia   2.08739306 -1.41052627 -0.10372163  0.130583080
## Wisconsin       2.05881199  0.60512507  0.13746933  0.182253407
## Wyoming         0.62310061 -0.31778662  0.23824049 -0.164976866
```

While it would be possible to carry out this lab using the `prcomp()` function, here we use the `svd()` function in order to illustrate its use.

We now omit 20 entries in the 50×4 data matrix at random. We do so by first selecting 20 rows (states) at random, and then selecting one of the four entries in each row at random. This ensures that every row has at least three observed values.

```
nomit <- 20
set.seed(15)
ina <- sample(seq(50), nomit)
inb <- sample(1:4, nomit, replace = TRUE)
```

```
Xna <- X
index.na <- cbind(ina, inb)
Xna[index.na] <- NA
```

Here, `ina` contains 20 integers from 1 to 50; this represents the states that are selected to contain missing values. And `inb` contains 20 integers from 1 to 4, representing the features that contain the missing values for each of the selected states. To perform the final indexing, we create `index.na`, a two-column matrix whose columns are `ina` and `inb`. We have indexed a matrix with a matrix of indices!

We now write some code to implement Algorithm 12.1. We first write a function that takes in a matrix, and returns an approximation to the matrix using the `svd()` function. This will be needed in Step 2 of Algorithm 12.1. As mentioned earlier, we could do this using the `prcomp()` function, but instead we use the `svd()` function for illustration.

```
fit.svd <- function(X, M = 1) {
   svdob <- svd(X)
   with(svdob,
       u[, 1:M, drop = FALSE] %*%
       (d[1:M] * t(v[, 1:M, drop = FALSE]))
     )
}
```

Here, we did not bother to explicitly call the `return()` function to return a value from `fit.svd()`; however, the computed quantity is automatically returned by `R`. We use the `with()` function to make it a little easier to index the elements of `svdob`. As an alternative to using `with()`, we could have written

inside the `fit.svd()` function.

To conduct Step 1 of the algorithm, we initialize `Xhat` — this is $\tilde{\mathbf{X}}$ in Algorithm 12.1 — by replacing the missing values with the column means of the non-missing entries.

```
Xhat <- Xna
xbar <- colMeans(Xna, na.rm = TRUE)
Xhat[index.na] <- xbar[inb]
```

Before we begin Step 2, we set ourselves up to measure the progress of our iterations:

```
thresh <- 1e-7
rel_err <- 1
iter <- 0
ismiss <- is.na(Xna)
mssold <- mean((scale(Xna, xbar, FALSE)[!ismiss])^2)
mss0 <- mean(Xna[!ismiss]^2)
```

Here `ismiss` is a new logical matrix with the same dimensions as `Xna`; a given element equals `TRUE` if the corresponding matrix element is missing. This is useful because it allows us to access both the missing and non-missing entries. We store the mean of the squared non-missing elements in `mss0`. We store the mean squared error of the non-missing elements of the old version of `Xhat` in `mssold`. We plan to store the mean squared error of the non-missing elements of the current version of `Xhat` in `mss`, and will then iterate Step 2 of Algorithm 12.1 until the *relative error*, defined as `(mssold - mss) / mss0`, falls below `thresh = 1e-7`. (Algorithm 12.1 tells us to iterate Step 2 until (12.14) is no longer decreasing. Determining whether (12.14) is decreasing requires us only to keep track of `mssold - mss`. However, in practice, we keep track of `(mssold - mss) / mss0` instead: this makes it so that the number of iterations required for Algorithm 12.1 to converge does not depend on whether we multiplied the raw data $\mathbf{X}$ by a constant factor.)

In Step 2(a) of Algorithm 12.1, we approximate `Xhat` using `fit.svd()`; we call this `Xapp`. In Step 2(b), we use `Xapp` to update the estimates for elements in `Xhat` that are missing in `Xna`. Finally, in Step 2(c), we compute the relative error. These three steps are contained in this `while()` loop:

```
while(rel_err > thresh) {
    iter <- iter + 1
    # Step 2(a)
    Xapp <- fit.svd(Xhat, M = 1)
    # Step 2(b)
    Xhat[ismiss] <- Xapp[ismiss]
    # Step 2(c)
    mss <- mean(((Xna - Xapp)[!ismiss])^2)
    rel_err <- (mssold - mss) / mss0
    mssold <- mss
    cat("Iter:", iter, "MSS:", mss,
      "Rel. Err:", rel_err, "\n")
    }
```

```
## Iter: 1 MSS: 0.3821695 Rel. Err: 0.6194004
## Iter: 2 MSS: 0.3705046 Rel. Err: 0.01161265
## Iter: 3 MSS: 0.3692779 Rel. Err: 0.001221144
## Iter: 4 MSS: 0.3691229 Rel. Err: 0.0001543015
## Iter: 5 MSS: 0.3691008 Rel. Err: 2.199233e-05
## Iter: 6 MSS: 0.3690974 Rel. Err: 3.376005e-06
## Iter: 7 MSS: 0.3690969 Rel. Err: 5.465067e-07
## Iter: 8 MSS: 0.3690968 Rel. Err: 9.253082e-08
```

We see that after eight iterations, the relative error has fallen below `thresh = 1e-7`, and so the algorithm terminates. When this happens, the mean squared error of the non-missing elements equals 0.369.

Finally, we compute the correlation between the 20 imputed values and the actual values:

```
cor(Xapp[ismiss], X[ismiss])
```

```
## [1] 0.6535043
```

In this lab, we implemented Algorithm 12.1 ourselves for didactic purposes. However, a reader who wishes to apply matrix completion to their data should use the `softImpute` package on `CRAN`, which provides a very efficient implementation of a generalization of this algorithm.