605-HW04-SVD

Michael Y.

September 22, 2019

Contents

Setup

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

##
## Attaching package: 'kableExtra'

## The following object is masked from 'package:dplyr':
##
##     group_rows
```

```
#### Function to write Matrix, courtesy of Vinayak Patel :
writeMatrix <- function(x) {
  begin <- "\\begin{bmatrix}"
  end   <- "\\end{bmatrix}"
  X     <-    apply(x, 1, function(x) {
      paste(
          paste(x, collapse = "&"),
          "\\\\"
      )
    }
  )
  paste(c(begin, X, end),
        collapse = "")
  }
```

```
#### Function to write numerical Matrix, controlling decimals, adapted from above :
wnumMatrix <- function(x) {
  begin <- "\\begin{bmatrix}"
  end   <- "\\end{bmatrix}"
  X     <-    apply(x, 1, function(x) {
      paste(
          paste(format(x, digits = options()$digits), collapse = "&"),
          "\\\\"
      )
    }
  )
  paste(c(begin, X, end),
        collapse = "")
  }
```

Function to kill tiny values (drop tiny decimal places from the right)

```r
killtiny <- function (INPUTMATRIX, INPUTDIGITS=getOption("digits")) {
  apply(formatC(x = INPUTMATRIX,
                digits = INPUTDIGITS,
                format = "f"),
        MARGIN = c(1,2),
        FUN=as.numeric)}
```

Part 1 - Verify Relationship of SVD and Eigenvalues

matrix A:

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   -1    0    4
```

In this problem, we'll verify using R that SVD and Eigenvalues are related as worked out in the weekly module.

Given a 3x2 matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ -1 & 0 & 4 \end{bmatrix}$

write code in R to compute $X = AA^T$ and $Y = A^T A$.

Then, compute the eigenvalues and eigenvectors of X and Y using the built-in commans [sic] in R.

Then, compute the left-singular, singular values, and right-singular vectors of A using the svd command.

Examine the two sets of singular vectors and show that they are indeed eigenvectors of X and Y.

In addition, the two non-zero eigenvalues (the 3rd value will be very close to zero, if not zero) of both X and Y are the same and are squares of the non-zero singular values of A.

Your code should compute all these vectors and scalars and store them in variables.

Please add enough comments in your code to show me how to interpret your steps.

Write code in R to compute $X = AA^T$ and $Y = A^TA$.

```
# Compute X
X = A %*% t(A)    # A*AT is 2x3 * 3x2 --> 2x2
X
```

```
##      [,1] [,2]
## [1,]   14   11
## [2,]   11   17
```

```
# Compute Y
Y = t(A) %*% A    # AT*A is 3x2 * 2x3 --> 3x3
Y
```

```
##      [,1] [,2] [,3]
## [1,]    2    2   -1
## [2,]    2    4    6
## [3,]   -1    6   25
```

$$X = \begin{bmatrix} 14 & 11 \\ 11 & 17 \end{bmatrix}; Y = \begin{bmatrix} 2 & 2 & -1 \\ 2 & 4 & 6 \\ -1 & 6 & 25 \end{bmatrix}$$

Then, compute the eigenvalues and eigenvectors of X and Y using the built-in commans [sic] in R.

```r
# Get the eigenvalues and eigenvectors of X
eigX = eigen(X)
eigX
```

```
## eigen() decomposition
## $values
## [1] 26.60180165559  4.39819834441
##
## $vectors
##                 [,1]            [,2]
## [1,] 0.657604286508 -0.753363526039
## [2,] 0.753363526039  0.657604286508
```

```r
X_eigenval1 = as.matrix(eigX$values)
X_eigenval1
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
```

```r
# kill the tiny part of X eigenvalues
X_eigenval2 = killtiny(X_eigenval1)
X_eigenval2
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
```

```r
# Get the eigenvalues and eigenvectors of Y
eigY = eigen(Y)
eigY
```

```
## eigen() decomposition
## $values
## [1] 26.6018016555872094386359094640  4.3981983444127434879078464290  0.0000000000000001058981956930
##
## $vectors
##                  [,1]            [,2]            [,3]
## [1,] -0.0185662914215 -0.672790268817  0.739600261634
## [2,]  0.2549993688964 -0.718451044302 -0.647150228929
## [3,]  0.9667629568231  0.176582420208  0.184900065408
```

```r
Y_eigenval1 = as.matrix(eigY$values)
Y_eigenval1
```

```
##                                 [,1]
## [1,] 26.6018016555872094386359094640
## [2,]  4.3981983444127434879078464290
## [3,]  0.0000000000000001058981956930
```

```r
# kill the tiny part of Y eigenvalues
Y_eigenval2 = killtiny(Y_eigenval1)
Y_eigenval2
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
## [3,]  0.00000000000
```

The two non-zero eigenvalues (the 3rd value will be very close to zero, if not zero) of both X and Y are the same

```
# examine difference between eigenvalues for X and Y
# add a zero to pretend there is a third eigenvalue for X, so the matrices can be compared
X_eigenval3 = rbind(X_eigenval2,0)
X_eigenval3
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
## [3,]  0.00000000000
```

```
# difference of X eigenvalues (with appended zero) vs. Y eigenvalues:
X_eigenval3 - Y_eigenval2
```

```
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
```

```
# Are the eigenvalues equal?
all.equal(X_eigenval3, Y_eigenval2)
```

```
## [1] TRUE
```

Thus, the two eigenvalues for X, $eig(X) = \begin{bmatrix} 26.601801655587 \\ 4.398198344413 \end{bmatrix}$,

are equal to the first two eigenvalues for Y, $eig(Y) = \begin{bmatrix} 26.601801655587 \\ 4.398198344413 \\ 0 \end{bmatrix}$.

What are these eigenvalues (in analytic terms) ?

Get the characteristic polynomials:

```
Xpoly = charpoly(X)
Xpoly
```

```
## [1]   1 -31 117
```

```
Ypoly = charpoly(Y)
Ypoly
```

```
## [1]   1 -31 117   0
```

So, the eigenvalues for X solve $t^2 - 31t^2 + 117 = 0$,
and the eigenvalues for Y solve $t^3 - 31t^2 + 117t = 0$.

By the quadratic formula, the solution for eigenvalues of X is

$$eigvals_X = \frac{31 \pm \sqrt{(-31)^2 - 4 \cdot 1 \cdot 117}}{2} = \frac{31 \pm \sqrt{493}}{2}$$

(The square root cannot be factored further as $493 = 29 * 17$, both of which are prime.)

The eigenvalues for Y are going to be the above pair of values as well as 0, as Y is singular.

Check that these match the eigenvalues

```
# check first eigenval
eigval1 = (31 + sqrt(493))/2
eigval1
```

```
## [1] 26.6018016556
```

```
all.equal(X_eigenval2[1],eigval1)
```

```
## [1] TRUE
```

```
# check second eigenval
eigval2 = (31 - sqrt(493))/2
eigval2
```

```
## [1] 4.39819834441
```

```
all.equal(X_eigenval2[2],eigval2)
```

```
## [1] TRUE
```

Get the roots of the polynomials numerically

```
# Two roots of the characteristic polynomial for X:
Xroots = t(t(as.numeric(rev(polyroot(rev(Xpoly))))))
Xroots
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
```

```
# Three roots of the characteristic polynomial for Y:
Yroots = t(t(as.numeric(rev(polyroot(rev(Ypoly))))))
Yroots
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
## [3,]  0.00000000000
```

Compute the left-singular, singular values, and right-singular vectors of A using the svd command:

```
A_svd = svd(x=A, nu=2, nv=3)    # we need to specify nv=3 so function will return the entire V
# left side
U = A_svd$u                  # U is 2x2
print('U: ')
```

```
## [1] "U: "
```

```
U
```

```
##                  [,1]            [,2]
## [1,] -0.657604286508 -0.753363526039
## [2,] -0.753363526039  0.657604286508
```

```
# diagonal entries
D = A_svd$d                  # D has 2 entries
print('D: ')
```

```
## [1] "D: "
```

```
D
```

```
## [1] 5.15769344335 2.09718819957
```

```
# sqrt of diagonal entries
sqrt_D = as.matrix(D^0.5)
print('sqrt(D): ')
```

```
## [1] "sqrt(D): "
```

```
sqrt_D
```

```
##               [,1]
## [1,] 2.27105557910
## [2,] 1.44816718633
```

```
# diagonal matrix
DI = D *  eye(length(D))    # Note this is "*" rather than "%*%" because
                            # we just want to put the diagonal entries into I
print('Diag: ')
```

```
## [1] "Diag: "
```

```
DI
```

```
##               [,1]          [,2]
## [1,] 5.15769344335 0.00000000000
## [2,] 0.00000000000 2.09718819957
```

```
# DDD needs to be 2x3 because V is 3x3
DDD = cbind(DI,0)
print('D as 2x3: ')
```

```
## [1] "D as 2x3: "
```

```
DDD
```

```
##                [,1]          [,2] [,3]
## [1,] 5.15769344335 0.00000000000    0
## [2,] 0.00000000000 2.09718819957    0
```

```
# right side
V = A_svd$v                 # V is 3x3
print('V: ')
```

```
## [1] "V: "
```

```
V
```

```
##                   [,1]            [,2]            [,3]
## [1,]  0.0185662914215 -0.672790268817 -0.739600261634
## [2,] -0.2549993688964 -0.718451044302  0.647150228929
## [3,] -0.9667629568231  0.176582420208 -0.184900065408
```

$$U = \begin{bmatrix} -0.657604286507732 & -0.753363526039492 \\ -0.753363526039492 & 0.657604286507732 \end{bmatrix}$$

$$D = \begin{bmatrix} 5.15769344335113 & 0 \\ 0 & 2.09718819956931 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.0185662914214504 & -0.672790268816595 & -0.739600261633639 \\ -0.254999368896366 & -0.718451044302279 & 0.647150228929434 \\ -0.966762956823082 & 0.176582420208403 & -0.18490006540841 \end{bmatrix}$$

To get the dimensions correct for multiplication, we need to add a blank column to D:

$$DDD = \begin{bmatrix} 5.15769344335113 & 0 & 0 \\ 0 & 2.09718819956931 & 0 \end{bmatrix}$$

The two non-zero eigenvalues of both X and Y are squares of the non-zero singular values of A:

```r
D_squared = as.matrix(D^2,2,1)
D_squared
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
```

```r
# Compare against non-zero eigenvalues
X_eigenval2
```

```
##                 [,1]
## [1,] 26.60180165559
## [2,]  4.39819834441
```

```r
# Are they almost same?
X_eigenval2 - D_squared
```

```
##                                [,1]
## [1,] -0.0000000000000255795384874
## [2,]  0.0000000000000256683563293
```

```r
all.equal(X_eigenval2,D_squared)
```

```
## [1] TRUE
```

```r
killtiny(X_eigenval2)==killtiny(D_squared)
```

```
##      [,1]
## [1,] TRUE
## [2,] TRUE
```

Examine the two sets of singular vectors and show that they are indeed eigenvectors of X and Y.

Remember that eigenvectors can be negated in sign – it just flips the direction – so we may have to flip signs on columns to check

Check U against eigenvectors of X:

```
# display U
U
```

```
##                 [,1]            [,2]
## [1,] -0.657604286508 -0.753363526039
## [2,] -0.753363526039  0.657604286508
```

```
# display eigenvectors of X:
X_eigvecs = eigX$vectors
X_eigvecs
```

```
##                [,1]            [,2]
## [1,] 0.657604286508 -0.753363526039
## [2,] 0.753363526039  0.657604286508
```

```
# we need to flip the signs on the first column of the eigenvectors to get a match
# Matrix to flip the signs in column 1:
FlipSigns_col1 = matrix(c(-1,-1,1,1),2,2,F)
FlipSigns_col1
```

```
##      [,1] [,2]
## [1,]   -1    1
## [2,]   -1    1
```

```
X_eigvecs_flipsign1 = X_eigvecs * FlipSigns_col1   # Note that this uses elementwise "*"
                                                   # rather than "%*%"
X_eigvecs_flipsign1
```

```
##                 [,1]            [,2]
## [1,] -0.657604286508 -0.753363526039
## [2,] -0.753363526039  0.657604286508
```

```
# check if this is almost equal to U
all.equal(X_eigvecs_flipsign1, U)
```

```
## [1] TRUE
```

```
X_eigvecs_flipsign1 - U
```

```
##                                [,1] [,2]
## [1,] 0.000000000000000222044604925    0
## [2,] 0.000000000000000000000000000    0
```

```r
# If we kill the decimals far to the right, do they match?
killtiny(X_eigvecs_flipsign1) == killtiny(U)
```

```
##      [,1] [,2]
## [1,] TRUE TRUE
## [2,] TRUE TRUE
```

Check that the eigenvectors (U) and the eigenvalues for X actually work: $Xu_i = \lambda_{X_i} u_i$

```r
# check the first eigenvalue and eigenvector
X_lhs1=X %*% U[,1]
X_lhs1
```

```
##                  [,1]
## [1,] -17.4934587975
## [2,] -20.0408270943
```

```r
X_rhs1=(as.matrix(X_eigenval2[1] * U[,1],2,1))
X_rhs1
```

```
##                  [,1]
## [1,] -17.4934587975
## [2,] -20.0408270943
```

```r
all.equal(X_lhs1,X_rhs1)
```

```
## [1] TRUE
```

```r
# check the second eigenvalue and eigenvector
X_lhs2=X %*% U[,2]
X_lhs2
```

```
##                  [,1]
## [1,] -3.31344221297
## [2,]  2.89227408420
```

```r
X_rhs2=(as.matrix(X_eigenval2[2] * U[,2],2,1))
X_rhs2
```

```
##                  [,1]
## [1,] -3.31344221297
## [2,]  2.89227408420
```

```r
all.equal(X_lhs2,X_rhs2)
```

```
## [1] TRUE
```

Check V against eigenvectors of Y:

```r
# display V:
V
```

```
##                  [,1]            [,2]            [,3]
## [1,]  0.0185662914215 -0.672790268817 -0.739600261634
## [2,] -0.2549993688964 -0.718451044302  0.647150228929
## [3,] -0.9667629568231  0.176582420208 -0.184900065408
```

```r
# display eigenvectors of Y:
Y_eigvecs = eigY$vectors
Y_eigvecs
```

```
##                  [,1]            [,2]            [,3]
## [1,] -0.0185662914215 -0.672790268817  0.739600261634
## [2,]  0.2549993688964 -0.718451044302 -0.647150228929
## [3,]  0.9667629568231  0.176582420208  0.184900065408
```

```r
# we need to flip the signs on the first and third columns of the eigenvectors to get a match
# Matrix to flip the signs in column 1
FlipSigns_col13 = matrix(c(-1,-1,-1,1,1,1,-1,-1,-1),3,3,F)
FlipSigns_col13
```

```
##      [,1] [,2] [,3]
## [1,]   -1    1   -1
## [2,]   -1    1   -1
## [3,]   -1    1   -1
```

```r
Y_eigvecs_flipsign13 = Y_eigvecs * FlipSigns_col13   # Note that this uses elementwise "*"
                                                      # rather than "%*%"
Y_eigvecs_flipsign13
```

```
##                  [,1]            [,2]            [,3]
## [1,]  0.0185662914215 -0.672790268817 -0.739600261634
## [2,] -0.2549993688964 -0.718451044302  0.647150228929
## [3,] -0.9667629568231  0.176582420208 -0.184900065408
```

```r
# check if this is almost equal to V
all.equal(Y_eigvecs_flipsign13, V)
```

```
## [1] TRUE
```

```r
Y_eigvecs_flipsign13 - V
```

```
##                                   [,1]                              [,2]                              [,3]
## [1,] -0.0000000000000000589805981832  0.000000000000000222044604925 -0.000000000000000333066907388
## [2,]  0.0000000000000002220446049250 -0.000000000000000444089209850 -0.000000000000000333066907388
## [3,] -0.0000000000000003330669073875 -0.000000000000000166533453694  0.000000000000000111022302463
```

```
# If we kill the decimals far to the right, do they match?
killtiny(Y_eigvecs_flipsign13) == killtiny(V)
```

```
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE
```

Check that the eigenvectors and eigenvalues for Y actually work: $Yv_i = \lambda_{Y_i} v_i$

```r
# check the first eigenvalue and eigenvector
Y_lhs1=Y %*% V[,1]
Y_lhs1
```

```
##                   [,1]
## [1,]   0.493896801873
## [2,]  -6.783442633681
## [3,] -25.717636425377
```

```r
Y_rhs1=(as.matrix(Y_eigenval2[1] * V[,1],3,1))
Y_rhs1
```

```
##                   [,1]
## [1,]   0.493896801873
## [2,]  -6.783442633681
## [3,] -25.717636425376
```

```r
all.equal(Y_lhs1,Y_rhs1)
```

```
## [1] TRUE
```

```r
# check the second eigenvalue and eigenvector
Y_lhs2=Y %*% V[,2]
Y_lhs2
```

```
##                  [,1]
## [1,] -2.959065046446
## [2,] -3.159890193592
## [3,]  0.776644508213
```

```r
Y_rhs2=(as.matrix(Y_eigenval2[2] * V[,2],3,1))
Y_rhs2
```

```
##                  [,1]
## [1,] -2.959065046446
## [2,] -3.159890193592
## [3,]  0.776644508213
```

```r
all.equal(Y_lhs2,Y_rhs2)
```

```
## [1] TRUE
```

```r
# check the third eigenvalue (here, zero) and eigenvector
Y_lhs3=Y %*% V[,3]
Y_lhs3
```

```
##                                 [,1]
## [1,]  0.000000000000000555111512313
## [2,]  0.000000000000000444089209850
## [3,] -0.000000000000000888178419700
```

```
Y_rhs3=(as.matrix(Y_eigenval2[3] * V[,3],3,1))
Y_rhs3
```

```
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
```

```
all.equal(Y_lhs3,Y_rhs3)
```

```
## [1] TRUE
```

```
# are they exactly equal if we kill the tiny bits
killtiny(Y_lhs3)==killtiny(Y_rhs3)
```

```
##      [,1]
## [1,] TRUE
## [2,] TRUE
## [3,] TRUE
```

Check that the SVD works: $A = UDV^T$

$$A = U \cdot D \cdot V^T = \begin{bmatrix} -0.6576 & -0.7534 \\ -0.7534 & 0.6576 \end{bmatrix} \begin{bmatrix} 5.158 & 0.000 & 0.000 \\ 0.000 & 2.097 & 0.000 \end{bmatrix} \begin{bmatrix} 0.01857 & -0.25500 & -0.96676 \\ -0.6728 & -0.7185 & 0.1766 \\ -0.7396 & 0.6472 & -0.1849 \end{bmatrix}$$

```
# confirm that A = U * DDD * t(V)
result = U %*% DDD %*% t(V)
result
```

```
##      [,1]                          [,2] [,3]
## [1,]    1 2.000000000000000000000000000    3
## [2,]   -1 0.000000000000000111022302463    4
```

```
# is it approximately equal to A ?
all.equal(result,A)
```

```
## [1] TRUE
```

```
# If we get rid of the tiny bits, is it exactly equal to A?
result2 = killtiny(result)
result2
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   -1    0    4
```

```
# do they exactly match?
A == result2
```

```
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
```

Part 2 - Inverse using co-factors

Using the procedure outlined in section 1 of the weekly handout, write a function to compute the inverse of a well-conditioned full-rank square matrix using co-factors.

In order to compute the co-factors, you may use built-in commands to compute the determinant.

Your function should have the following signature:

$B = myinverse(A)$

where A is a matrix and B is its inverse and $A \cdot B = I$.

The off-diagonal elements of I should be close to zero, if not zero.

Likewise, the diagonal elements should be close to 1, if not 1.

Small numerical precision errors are acceptable but the function myinverse should be correct and must use co-factors and determinant of A to compute the inverse.

Function myinverse

```r
myinverse <- function(A){
### Function to compute the inverse of a well-conditioned full-rank square matrix using co-factors

  # confirm that A is of type "matrix"
  if (class(A) != "matrix") stop ("Parameter must be of type MATRIX.")

  # confirm that A is square
  Arows = nrow(A)
  Acols = ncol(A)
  if (Arows != Acols) stop ("Parameter must be a SQUARE matrix.")

  # check that A is non-singular
  Adet = det(A)
  epsilon = 1e-10
  if (abs(Adet) < epsilon) stop ("Parameter must be NON-singular.")

  # check that A is well-conditioned
  # condition number is obtained from the svd -
  # it is the ratio of the largest and smallest singular values
  Acond = cond(A)
  if (Acond > 1/epsilon) stop ("Parameter must be WELL-conditioned.")

  # Create an identity matrix of the same size as A, in which to populate the cofactors:
  detcofactors = zeros(Arows)

  for (i in 1:Arows){

    for (j in 1:Acols){

      # derive the cofactor by dropping the current row and column, using A[-i,-j]
      # in the case of a 2x2 matrix, this would yield back a scalar,
      # which would cause problems for the det() function.
      # so, ensure that the cofactor is a matrix, even if it is a 1x1 matrix
      cofactor = matrix(A[-i,-j],(Arows-1),(Acols-1),T)

      # compute the determinant of the cofactors,
      # multiplied by a negative sign in the case of odd (row+column)
      detcofactors[i,j] =  det(cofactor) * (-1)^(i+j)

    } # for j

  } # for i

  # The inverse of A is calculated as the transpose of detcofactors, divided by the determinant of A
  Ainverse <- t(detcofactors)/Adet

  return(Ainverse)
}
```

Create some matrices for testing:

```
# Create a matrix containing the single element, 0
A0 = matrix(0)
A0
```

```
##      [,1]
## [1,]    0
```

```
# Create a matrix containing the single element, 7
A1 = matrix(7)
A1
```

```
##      [,1]
## [1,]    7
```

```
# Create a 2x2 non-singular matrix
A22 = c(
  1,2,
  2,1
)
A22 = matrix(A22,2,2,T)
A22
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    2    1
```

```
# Create a 2x3 non-square matrix
A23 = c(
  1,2,3,
  6,5,4
)
A23 = matrix(A23,2,3,T)
A23
```

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

```
# Create a 3x3 non-singular matrix
A33 = c(
1, 2, 3,
0, 4, 5,
0, 0, 6
)
A33 = matrix(A33,nrow=3,byrow=T)
A33
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0    4    5
## [3,]    0    0    6
```

```r
# Create a 3x3 singular matrix
A33sing = c(
  2,    2,   -1,
  2,    4,    6,
 -1,    6,   25
)

A33sing = matrix(A33sing,nrow=3,byrow=T)
A33sing
```

```
##      [,1] [,2] [,3]
## [1,]    2    2   -1
## [2,]    2    4    6
## [3,]   -1    6   25
```

```r
# zero determinant
det(A33sing)
```

```
## [1] 0
```

```r
# Create a 3x3 ill-conditioned matrix
A33cond = c(
  2,    2,   -1,
  2,    4,    6,
 -1,    6,   25.00000001
)
A33cond = matrix(A33cond,nrow=3,byrow=T)
A33cond
```

```
##      [,1] [,2]        [,3]
## [1,]    2    2 -1.00000000
## [2,]    2    4  6.00000000
## [3,]   -1    6 25.00000001
```

```r
# Tiny determinant
det(A33cond)
```

```
## [1] 0.0000000400000015333
```

```r
# huge condition number
cond(A33cond)
```

```
## [1] 77810501921.5
```

Test a 1x1 matrix containing just the element "0":

$$A_0 = \begin{bmatrix} 0 \end{bmatrix}$$

```r
# compute myinverse
myA0inv = try(myinverse(A0))
```

```
## Error in myinverse(A0) : Parameter must be NON-singular.
```

Test a 2x3 matrix:

$$A_{2x3} = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \end{bmatrix}$$

```r
# compute myinverse
myA23inv = try(myinverse(A23))
```

```
## Error in myinverse(A23) : Parameter must be a SQUARE matrix.
```

Test a 3x3 *singular* matrix:

$$A_{3x3-sing} = \begin{bmatrix} 2 & 2 & -1 \\ 2 & 4 & 6 \\ -1 & 6 & 25 \end{bmatrix}$$

```r
# compute myinverse
myA33nsinv = try(myinverse(A33ns))
```

```
## Error in myinverse(A33ns) : object 'A33ns' not found
```

Test a 3x3 *ill-conditioned* matrix:

$$A_{3x3-ill-cond} = \begin{bmatrix} 2 & 2 & -1 \\ 2 & 4 & 6 \\ -1 & 6 & 25.00000001 \end{bmatrix}$$

```r
# compute myinverse
myA33condinv = try(myinverse(A33cond))
```

```
## Error in myinverse(A33cond) : Parameter must be WELL-conditioned.
```

Test a 1x1 matrix containing just the element "7":

$$A_{1x1} = \begin{bmatrix} 7 \end{bmatrix}$$

```
# compute myinverse
myA1inv = myinverse(A1)
myA1inv
```

```
##                  [,1]
## [1,] 0.142857142857
```

```
# check to see if it gives back the identity
checkA1 = A1 %*% myA1inv
checkA1
```

```
##      [,1]
## [1,]    1
```

```
all.equal(checkA1,eye(nrow(A1)))
```

```
## [1] TRUE
```

```
# does myinverse come quite close to the official inverse?
myA1inv - inv(A1)
```

```
##                                  [,1]
## [1,] 0.0000000000000000277555756156
```

```
all.equal(myA1inv,inv(A1))
```

```
## [1] TRUE
```

```
killtiny(myA1inv)
```

```
##                  [,1]
## [1,] 0.142857142857
```

```
inv(A1)
```

```
##                  [,1]
## [1,] 0.142857142857
```

```
# does the inverse of the inverse of A1 give you back A1 ?
doubleinverseA1 = myinverse(myinverse(A1))
doubleinverseA1
```

```
##      [,1]
## [1,]    7
```

```
all.equal(doubleinverseA1,A1)
```

```
## [1] TRUE
```

Test a 2x2 non-singular matrix:

$$A_{2x2} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

```
# compute myinverse
myA22inv = myinverse(A22)
myA22inv
```

```
##                  [,1]            [,2]
## [1,] -0.333333333333  0.666666666667
## [2,]  0.666666666667 -0.333333333333
```

```
# check to see if it gives back the identity
checkA22 = A22 %*% myA22inv
checkA22
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
all.equal(checkA22,eye(nrow(A22)))
```

```
## [1] TRUE
```

```
# does myinverse come quite close to the official inverse?
inv(A22)
```

```
##                  [,1]            [,2]
## [1,] -0.333333333333  0.666666666667
## [2,]  0.666666666667 -0.333333333333
```

```
myA22inv - inv(A22)
```

```
##                                  [,1]                               [,2]
## [1,] -0.0000000000000000555111512313  0.000000000000000110223024625
## [2,]  0.000000000000000110223024625 -0.0000000000000000555111512313
```

```
all.equal(myA22inv,inv(A22))
```

```
## [1] TRUE
```

```
killtiny(myA22inv)
```

```
##                  [,1]            [,2]
## [1,] -0.333333333333  0.666666666667
## [2,]  0.666666666667 -0.333333333333
```

```r
inv(A22)
```

```
##                  [,1]            [,2]
## [1,] -0.333333333333  0.666666666667
## [2,]  0.666666666667 -0.333333333333
```

```r
# does the inverse of the inverse of A22 give you back A22 ?
doubleinverseA22 = myinverse(myinverse(A22))
doubleinverseA22
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    2    1
```

```r
all.equal(doubleinverseA22,A22)
```

```
## [1] TRUE
```

Test a 3x3 non-singular matrix:

$$A_{3x3} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}$$

```
# compute myinverse
myA33inv = myinverse(A33)
myA33inv
```

```
##      [,1]  [,2]             [,3]
## [1,]    1 -0.50 -0.0833333333333
## [2,]    0  0.25 -0.2083333333333
## [3,]    0  0.00  0.1666666666667
```

```
# check to see if it gives back the identity
checkA33 = A33 %*% myA33inv
checkA33
```

```
##      [,1] [,2]                          [,3]
## [1,]    1    0 -0.0000000000000000111022302463
## [2,]    0    1 -0.0000000000000000111022302463
## [3,]    0    0  1.0000000000000000222044604925
```

```
all.equal(checkA33,eye(nrow(A33)))
```

```
## [1] TRUE
```

```
# does myinverse come quite close to the official inverse?
inv(A33)
```

```
##      [,1]  [,2]             [,3]
## [1,]    1 -0.50 -0.0833333333333
## [2,]    0  0.25 -0.2083333333333
## [3,]    0  0.00  0.1666666666667
```

```
myA33inv - inv(A33)
```

```
##                              [,1]                            [,2]                            [,3]
## [1,] 0.0000000000000000444089209850 -0.0000000000000001110223024625 -0.0000000000000000277555756156
## [2,] 0.0000000000000000000000000000  0.0000000000000000555111512313 -0.0000000000000001110223024625
## [3,] 0.0000000000000000000000000000  0.0000000000000000000000000000  0.0000000000000000555111512313
```

```
all.equal(myA33inv,inv(A33))
```

```
## [1] TRUE
```

```
killtiny(myA33inv)
```

```
##      [,1]  [,2]            [,3]
## [1,]    1 -0.50 -0.083333333333
## [2,]    0  0.25 -0.208333333333
## [3,]    0  0.00  0.166666666667
```

```
inv(A33)
```

```
##      [,1]  [,2]             [,3]
## [1,]    1 -0.50 -0.0833333333333
## [2,]    0  0.25 -0.2083333333333
## [3,]    0  0.00  0.1666666666667
```

```
# does the inverse of the inverse of A33 give you back A33 ?
doubleinverseA33 = myinverse(myinverse(A33))
doubleinverseA33
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0    4    5
## [3,]    0    0    6
```

```
all.equal(doubleinverseA33,A33)
```

```
## [1] TRUE
```