Math 365 / Comp 365: Homework 7

Lawson Busch

Worked with Raven Mcknight

Please drop a stapled hard copy of your answers in the folder outside OLRI 228

The following line sources functions from the class file `365Functions.r` . Feel free to use any of these functions throughout the semester.

```
source("https://drive.google.com/uc?export=download&id=10dNH3VbvxS8Z3OHjP4i9gRbtsf91VVBb") #365functions.R
require(Matrix)
require(expm)
require(igraph)
```

```
## Warning: package 'igraph' was built under R version 3.4.4
```

```
require(foreign)
```

Problem 1

Left Multiply by A inverse and then figure it out

Note: This is Part II of Activity 22

a. Show that if A is an $n \times n$ invertible matrix and λ is an eigenvalue of A with eigenvector v, then $\frac{1}{\lambda}$ is an eigenvalue of A^{-1} with the same eigenvector v.

$$Av = \lambda v$$

$$A^{-1}Av = A^{-1}\lambda v$$

$$v = \lambda A^{-1} v$$

$$\frac{1}{\lambda} v = A^{-1} v$$

b. What happens if you apply the power iteration to A^{-1} ?

If the power iteration is applied to A^{-1} , it will converge to the smallest eigenvalue.

c. Let A be an $n \times n$ matrix, and let $C = A - sI$, where I is the $n \times n$ identity matrix and s is a scalar.

Show that if λ is an eigenvalue of A with eigenvector v, then $\lambda - s$ is an eigenvalue of C with the same eigenvector v. Note: s is often called a **shift**.

$$(A - sI)v = Av - sIv$$

$$(A - sI)v = \lambda v - sv$$

$$(A - sI)v = (\lambda - s)v$$

d. Let's say you had a guess $\bar{\lambda}$ for an eigenvalue of A and wanted to find the associated eigenvector. Can you use the previous two results to come up with a good way to do that?

If we know that we have a λ near s, then we can shift A by λ to find the corresponding eigenvector.

Problem 2

Note: This is Exercise 2 of Activity 23

Congratulations, you just turned 21! First step: go on kayak.com to find a cheap flight. Second step: go on hotwire.com to find a cheap hotel. Third step: Vegas, baby!

Your plan is to bring $100 of gambling money, gamble $1 at a time, and stop gambling when you hit $0 (then learn how to take joy in your friends' successes) or when you hit $200 (at which point you'll have no trouble finding different ways to spend your winnings in Vegas). Let's assume at first that the probability of winning each game is 50% (probably not the best assumption).

a. Let's model your earnings with a Markov chain. What are the states? Draw a state diagram and label the edge with probabilities.

b. Form the transition matrix P. Hint: use my `TriDiag` function from `365Functions.r`. Check that P's rows sum to 1 using the command `rowSums`.

```
p1 = 1/2
vec1 = rep(0, 201)
vec1[1] = 1
vec1[201] = 1
upper = rep(p1, 200)
upper[1] = 0
lower = rep(1-p1, 200)
lower[200] = 0
P = TriDiag(vec1, lower, upper)
tP = t(P)
colSums(tP)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

c. Numerically compute the probability that you have $105 after placing 10 different $1 bets. How about $106?

```
x100 = rep(0, 201)
x100[101] = 1
Pk = as.matrix(tP)%^%10
xk100 = Pk %*% x100
xk100[106]
```

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

```
xk100[107]
```

```
## [1] 0.04394531
```

The probability that you have $105 after 10 iterations is 0%, while the probability that you have $106 after 10 iterations is 4.3%.

d. Use `eigen` to find the eigenvalues and eigenvectors of $P^\top$. What do you notice? Can you explain why the eigenvectors associated with the eigenvalue 1 are steady-state distributions?

```
#eigen(tP)
```

The eigenvectors associated with the eigenvalue 1 are steady-state distributions because they are associated with the 0 value and the 200 value, which is the end point for the Markov Chain and each of these points has no transitions to another state.

e. This Markov chain is not irreducible! Once you reach $0 or $200, you cannot reach any other state. We say that those two states are ***absorbing*** states. We are interested in the absorption probabilities; i.e., what is the probability that you reach $0 before $200, and vice versa?

To answer this, we can first reorder the state labels so that the absorption states $0 and $200 are the first two listed, and then everything else. That is, we can rearrange the transition matrix P into the following form:

$$P = \begin{bmatrix} I & 0 \\ B & Q \end{bmatrix},$$

where

- I is a 2x2 identity matrix
- 0 is a $2 \times (N-2)$ matrix of zeros
- B is a $(N-2) \times 2$ matrix giving the transition probabilities from the non-absorbing states (called the ***transient*** states) to the absorbing states, and
- Q is the matrix of transition probabilities from the transient states to other transient states.

To compute B and Q in `R`, you can either directly access the appropriate submatrices of the original transition matrix P or you can rearrange P into the form above. Here is an example of how to rearrange a matrix in R:

```
(Z=matrix(1:16,nrow=4,ncol=4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```
new.order=c(1,4,2,3)
(Z.rearranged=Z[new.order,new.order])
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   13    5    9
## [2,]    4   16    8   12
## [3,]    2   14    6   10
## [4,]    3   15    7   11
```

Once you have B, Q, and I, to find the absorption probabilities, compute the ***fundamental matrix*** $S = (I - Q)^{-1}$, and the probabiity of absorbing into state j (say \$0 in this case) starting from transient state i (say \$100 in this case) is $(SB)_{ij}$. If you start with \$100, what is the probability of reaching \$200 before going broke? How does it change if you start with \$120 and only aim to make \$80 profit?

Aside: we won't go into details about why SB is the solution to the absorption probabilities. For that, you'll need to take some probability!

```
I = diag(2)
N = rbind(rep(0, 199), rep(0, 199))
B = cbind(P[2:200, 1], P[2:200, 201])
Q = P[2:200, 2:200]
weirdP = rbind(cbind(I, N), cbind(B, Q))
```

```
S = solve(diag(nrow(Q))-Q)
SB = S%*%B

SB[100, 1]
```

```
## [1] 0.5
```

```
SB[100, 2]
```

```
## [1] 0.5
```

```
SB[120, 2]
```

```
## [1] 0.6
```

The probability of reaching $200 before going broke if you start at $100 is 50%. If you if you start with $120 and only aim to make $80 profit the probability increases to 60%.

f. Does your probability of reaching $200 before going broke change if you bet $10 at a time or $100 at a time?

```
p2 = 1/2
vec2 = rep(0, 21)
vec2[1] = 1
vec2[21] = 1
upper = rep(p2, 20)
upper[1] = 0
lower = rep(1-p2, 20)
lower[20] = 0
P2 = TriDiag(vec2, lower, upper)
tP2 = t(P2)
colSums(tP2)
```

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

```
I2 = diag(2)
N2 = rbind(rep(0, 19), rep(0, 19))
B2 = cbind(P2[2:20, 1], P2[2:20, 21])
Q2 = P2[2:20, 2:20]
weirdP2 = rbind(cbind(I2, N2), cbind(B2, Q2))
```

```
S2 = solve(diag(nrow(Q2))-Q2)
SB2 = S2%*%B2

SB2[10, 2]
```

```
## [1] 0.5
```

The probability of reaching $200 before going broke remains the same at 50% if you bet $10 at a time.

```
p3 = 1/2
vec3 = rep(0, 3)
vec3[1] = 1
vec3[3] = 1
upper = rep(p3, 2)
upper[1] = 0
lower = rep(1-p3, 2)
lower[2] = 0
P3 = TriDiag(vec3, lower, upper)
tP3 = t(P2)
colSums(tP2)
```

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

```
I3 = diag(2)
N3 = rbind(rep(0, 1), rep(0, 1))
B3 = cbind(P[2, 1], P3[2, 3])
Q3 = as.matrix(P3[2, 2])
weirdP3 = rbind(cbind(I3, N3), cbind(B3, Q3))
```

```
S3 = solve(diag(nrow(Q3))-Q3)
SB3 = S3%*%B3
```

```
SB3[1, 2]
```

```
## [1] 0.5
```

The probability also does not change if you start with a bet of $100, as it stays at 50% that you will reach $200 before you go broke.

g. The actual odds of winning a game in Vegas are not equal to 50%! Let's say you are betting on red at the roulette wheel. Assuming it is a wheel with a double zero, your chances of winning each game are $18/38 \approx 47.4\%$. Now does your probability of reaching $200 before going broke change if you bet $10 at a time or $100 at a time? What is the best strategy?

Note 1 : the model in this problem does not take into account any utility you might derive from the free beverages provided by the casino for the duration of your gambling activities.

Note 2: Here (http://www.onlineroulette.ca/guides/american-vs-european-roulette.php) is some more information about single zero versus double zero roulette wheels.

```
p2 = 18/32
vec2 = rep(0, 21)
vec2[1] = 1
vec2[21] = 1
upper = rep(p2, 20)
upper[1] = 0
lower = rep(1-p2, 20)
lower[20] = 0
P2 = TriDiag(vec2, lower, upper)
tP2 = t(P2)
colSums(tP2)
```

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

```
I2 = diag(2)
N2 = rbind(rep(0, 19), rep(0, 19))
B2 = cbind(P2[2:20, 1], P2[2:20, 21])
Q2 = P2[2:20, 2:20]
weirdP2 = rbind(cbind(I2, N2), cbind(B2, Q2))
```

```
S2 = solve(diag(nrow(Q2))-Q2)
SB2 = S2%*%B2

SB2[10, 2]
```

```
## [1] 0.9250582
```

```
p3 = 18/32
vec3 = rep(0, 3)
vec3[1] = 1
vec3[3] = 1
upper = rep(p3, 2)
upper[1] = 0
lower = rep(1-p3, 2)
lower[2] = 0
P3 = TriDiag(vec3, lower, upper)
tP3 = t(P2)
colSums(tP2)
```

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

```
I3 = diag(2)
N3 = rbind(rep(0, 1), rep(0, 1))
B3 = cbind(P[2, 1], P3[2, 3])
Q3 = as.matrix(P3[2, 2])
weirdP3 = rbind(cbind(I3, N3), cbind(B3, Q3))
```

```
S3 = solve(diag(nrow(Q3))-Q3)
SB3 = S3%*%B3

SB3[1, 2]
```

```
## [1] 0.5625
```

Here we see that the probability of going broke before reaching $200 is 92% if you bet $10 at a time but only 56% if you bet $100 at a time. This implies that you should bet big if you want to win big, as it is far more likely to work out.

Problem 3

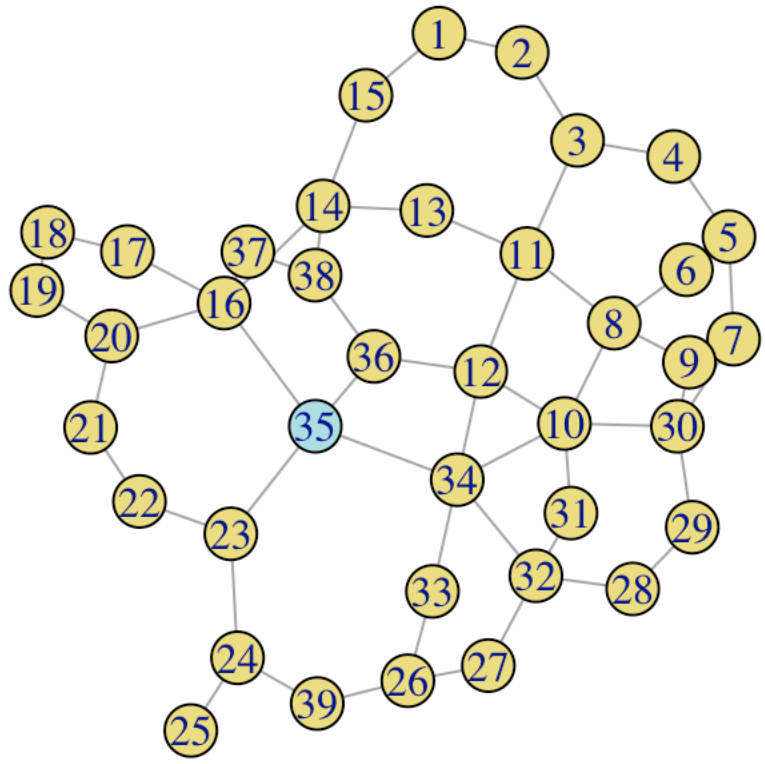
Note: This is Exercise 3 of Activity 23

Russian historians often attribute the dominance and rise to power of Moscow to its strategic position on medieval trade routes (see below). Others argue that sociological and political factors aided Moscow's rise to power, and thus Moscow did not rise to power strictly because of its strategic location on the trade routes. You are to use eigenvectors to analyze this question.

The following code loads the name of the Russian cities (in `RusCity`), loads the adjacency matrix into the matrix `A`, and plots the graph.

```
# this package is required to plot the graph
require(igraph)
```

```
# This loads the adjacency matrix into A and plots it
source("https://drive.google.com/uc?export=download&id=12exaJ8eOwe-BVKYAPOD0RH25HrOP0Rjx")
```



a. Let $B = A + I$ be the ***augmented adjacency matrix***, let $x = (1, 1, \dots, 1)^\top$, and compute Bx, B^2x, B^3x. The entries are nonnegative integers, and they can be interpreted as counting something. What does the i^{th} entry $(B^k x)_i$ count?

```
B = A + diag(39)
x = rep(1, 39)
B%*%x
```

```
##       [,1]
##  [1,]    3
##  [2,]    3
##  [3,]    4
##  [4,]    3
##  [5,]    4
##  [6,]    3
##  [7,]    4
##  [8,]    5
##  [9,]    4
## [10,]    6
## [11,]    5
## [12,]    5
## [13,]    3
## [14,]    5
## [15,]    3
## [16,]    6
## [17,]    3
## [18,]    3
## [19,]    3
## [20,]    4
## [21,]    3
## [22,]    3
## [23,]    4
## [24,]    4
## [25,]    2
## [26,]    4
## [27,]    3
## [28,]    3
## [29,]    3
## [30,]    5
## [31,]    3
## [32,]    5
## [33,]    3
## [34,]    6
## [35,]    5
## [36,]    4
## [37,]    3
## [38,]    4
## [39,]    3
```

```
B%*%B%*%x
```

```
##       [,1]
##  [1,]    9
##  [2,]   10
##  [3,]   15
##  [4,]   11
##  [5,]   14
##  [6,]   12
##  [7,]   17
##  [8,]   23
##  [9,]   18
## [10,]   30
## [11,]   22
## [12,]   26
## [13,]   13
## [14,]   21
## [15,]   11
## [16,]   26
## [17,]   12
## [18,]    9
## [19,]   10
## [20,]   16
## [21,]   10
## [22,]   10
## [23,]   16
## [24,]   13
## [25,]    6
## [26,]   13
## [27,]   12
## [28,]   11
## [29,]   11
## [30,]   22
## [31,]   14
## [32,]   20
## [33,]   13
## [34,]   30
## [35,]   25
## [36,]   18
## [37,]   13
## [38,]   16
## [39,]   11
```

```
B%*%B%*%B%*%x
```

```
##       [,1]
##  [1,]   30
##  [2,]   34
##  [3,]   58
##  [4,]   40
##  [5,]   54
##  [6,]   49
##  [7,]   71
##  [8,]  105
##  [9,]   80
## [10,]  145
## [11,]   99
## [12,]  126
## [13,]   56
## [14,]   87
## [15,]   41
## [16,]  113
## [17,]   47
## [18,]   31
## [19,]   35
## [20,]   62
## [21,]   36
## [22,]   36
## [23,]   64
## [24,]   46
## [25,]   19
## [26,]   49
## [27,]   45
## [28,]   42
## [29,]   44
## [30,]   98
## [31,]   64
## [32,]   87
## [33,]   56
## [34,]  144
## [35,]  115
## [36,]   85
## [37,]   55
## [38,]   68
## [39,]   37
```

CHECK THIS

The i^{th} entry $(B^k x)_i$ counts the accessibility of the node in the matrix. That is, after each iteration, it counts the total number of possible paths that lead to this node after k iterations in the graph. This can be taken to be a measure of how central or accessible this node is in the graph.

b. The sequence Bx, B^2x, B^3x, … should converge to the dominant eigenvector of B. Explain why the dominant eigenvector of the augmented adjacency matrix is a measure of accessibility. If this is not

clear, you should have a look at the article “Linear Algebra in Geography: Eigenvectors of Networks,” (http://www.jstor.org/stable/2689388?seq=1#page_scan_tab_contents) by Philip D. Straffin, Jr. in Mathematics Magazine, November 1980.

The dominant eigenvector is a measure of accessibility because its values correspond to the likely hood that you will end up at a node after k iterations. Essentially, it is a measure of the probability that you will arrive at a given city, and thus can be used as a measure of how accessible a city is. The higher the probability that you will end there, the more accessible it is. The lower the probability that you will end there, the less accessible it is.

c. Is the augmented adjacency matrix B primitive? How do you know?

The augmented adjacency matrix B is not primitive because it is periodic. B can only be primitive if and only if it is aperiodic and irreducible, but since it is periodic it cannot be primitive.

d. Compute the dominant eigenvector of B. Do it two ways: (i) use `R` 's function `eigen` ; (ii) use my power iteration function `PI` . Report the number of steps needed in power iteration so that the answer you get is correct to 2 decimal places in the infinity norm.

```
out = eigen(B)
v = out$vectors[,1] #The dominant eigen value
v
```

```
##  [1] -0.02042804 -0.03165857 -0.08985265 -0.05032510 -0.08545174
##  [6] -0.10063531 -0.14670548 -0.26510519 -0.18677594 -0.40505390
## [11] -0.23101252 -0.34980580 -0.09995430 -0.11717216 -0.03950126
## [16] -0.15520771 -0.05061439 -0.02110440 -0.02290150 -0.05867157
## [21] -0.02626961 -0.03283701 -0.08811611 -0.03704991 -0.01063602
## [26] -0.06852954 -0.07924554 -0.08637114 -0.09335115 -0.23881186
## [31] -0.17585248 -0.20751743 -0.12916393 -0.38140508 -0.23706014
## [36] -0.20105556 -0.07713829 -0.11349877 -0.03030897
```

```
PIout = PI(B, tol=0.005) #Check if correct to two decimal places
PIout$vec
```

```
##             [,1]
##  [1,] 0.02248508
##  [2,] 0.03013338
##  [3,] 0.08396333
##  [4,] 0.04410130
##  [5,] 0.07376296
##  [6,] 0.09024000
##  [7,] 0.12955200
##  [8,] 0.24704853
##  [9,] 0.16792216
## [10,] 0.39100105
## [11,] 0.22281850
## [12,] 0.34751689
## [13,] 0.10402560
## [14,] 0.13804290
## [15,] 0.04732026
## [16,] 0.18676470
## [17,] 0.06441232
## [18,] 0.02907082
## [19,] 0.03181674
## [20,] 0.07605936
## [21,] 0.03535884
## [22,] 0.04102853
## [23,] 0.10174695
## [24,] 0.04507913
## [25,] 0.01368644
## [26,] 0.07261716
## [27,] 0.08131636
## [28,] 0.08437976
## [29,] 0.08618616
## [30,] 0.21927698
## [31,] 0.17135370
## [32,] 0.20781483
## [33,] 0.13220124
## [34,] 0.38483014
## [35,] 0.25865182
## [36,] 0.21369376
## [37,] 0.09407541
## [38,] 0.13083671
## [39,] 0.03507069
```

```
PIout$steps
```

```
## [1] 13
```

The power iteration required 13 steps to get the answer correct within two decimal places in the infinity norm.

e. ***Gould's index of accessibility*** is just the dominant eigenvector of B, normalized so that the entries

sum to 1; i.e., if v is the dominant eigenvector, then Gould's index is

```
v\sum(v)
```

Compute Gould's index for this problem and answer the historians' question.

```
gInd = v/sum(v)
gInd
```

```
##  [1] 0.004218790 0.006538114 0.018556330 0.010393117 0.017647456
##  [6] 0.020783161 0.030297552 0.054749410 0.038572887 0.083651558
## [11] 0.047708607 0.072241745 0.020642520 0.024198345 0.008157783
## [16] 0.032053431 0.010452862 0.004358472 0.004729608 0.012116828
## [21] 0.005425189 0.006781486 0.018197702 0.007651532 0.002196546
## [26] 0.014152691 0.016365754 0.017837331 0.019278839 0.049319323
## [31] 0.036316979 0.042856411 0.026674879 0.078767614 0.048957559
## [36] 0.041521909 0.015930566 0.023439718 0.006259396
```

```
RusCity[35]
```

```
## [1] "Moscow"
```

```
gInd[35]
```

```
## [1] 0.04895756
```

```
max(gInd)
```

```
## [1] 0.08365156
```

From the Gould Index, we can see that Moscow is not the most important city when strickly speaking of centrality of trade routes. However, it is still a very important city when it comes to trade routes, as its normalized value is still one of the highest in the array. As a result, we can conclude that Moscow's rise to prominance was not solely a result of its trade centrality, as it is not the most central city in the set, and must have included other factors, such as political importance. However, I think it would be wrong to say that Moscow's centrality did not play a role in its rise to prominance, as it is still a very central city.

Problem 4

Note: This is Exercise 4 from Activity 24

Use the SVD to find a least squares solution to $Ax = b$, where $b = (15, 12, 8, 5, 7)^T$ and

$$A = \begin{pmatrix} 1 & -1 & 2 \\ 1 & 1 & 4 \\ 0 & 0 & 4 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix}.$$

a. First do it with the reduced SVD (this is what `R` will give back if you use `svd(A)`). You should find x_*, b_*, and the residual vector $r = b - b_*$. Compute $||r||$.

```
A = cbind(c(1, 1, 0, 1, 1), c(-1, 1, 0, 1, 1), c(2, 4, 4, 2, 1))
b = c(15, 12, 8, 5, 7)

svdRes = svd(A)
u = svdRes$u
v = svdRes$v
d = svdRes$d

d = diag(d)
d
```

```
##          [,1]     [,2]     [,3]
## [1,] 6.617029 0.000000 0.000000
## [2,] 0.000000 1.916442 0.000000
## [3,] 0.000000 0.000000 1.241844
```

```
psuedoInv = solve(t(d)%*%d)%*%t(d)
psuedoInv
```

```
##          [,1]      [,2]     [,3]
## [1,] 0.1511252 0.0000000 0.000000
## [2,] 0.0000000 0.5218003 0.000000
## [3,] 0.0000000 0.0000000 0.805254
```

```
xstar = v %*% psuedoInv %*% t(u) %*% b
xstar
```

```
##          [,1]
## [1,]  7.201613
## [2,] -3.830645
## [3,]  1.983871
```

```
bstar = A %*% xstar
bstar
```

```
##           [,1]
## [1,] 15.000000
## [2,] 11.306452
## [3,]  7.935484
## [4,]  7.338710
## [5,]  5.354839
```

```
r = b-bstar

vnorm(r)
```

```
## [1] 2.943007
```

b. Now use the full SVD to do it (you do this by telling `svd` how many vectors you want in `U` by using `nu = ???`). When you do it this way you have to be careful about dividing by zero if A is not full rank, but the upside is that you can calculate $||r||$ using the vector $c = \bar{U}^T b$, without having to compute x_* or r.

```
fsvd = svd(A, nu = 5)
fu = fsvd$u
fv = fsvd$v
fd = fsvd$d

c = t(fu)%*%b
y = c/fd
```

```
## Warning in c/fd: longer object length is not a multiple of shorter object
## length
```

```
xstar = fv %*% y[1:3]

bstar = A %*% xstar

r = b-bstar

xstar
```

```
##           [,1]
## [1,]  7.201613
## [2,] -3.830645
## [3,]  1.983871
```

```
bstar
```

```
##           [,1]
## [1,] 15.000000
## [2,] 11.306452
## [3,]  7.935484
## [4,]  7.338710
## [5,]  5.354839
```

```
vnorm(r)
```

```
## [1] 2.943007
```

Note: in practice, you would always compute the least squares solution with the reduced SVD, not the full one.

Problem 5

Note: This is Exercise 2 from Activity 26

```
library(jpeg)
```

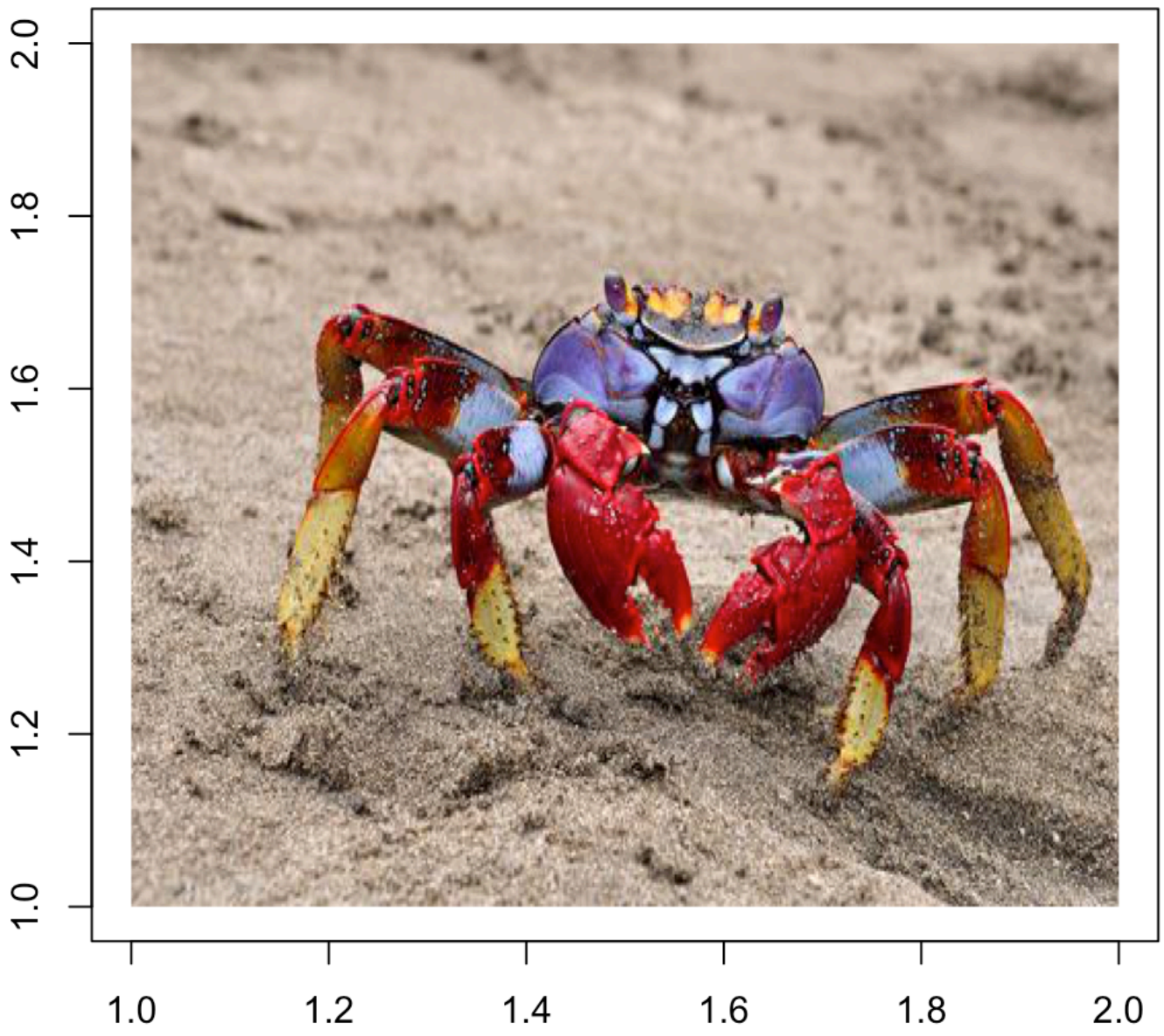
I know before this beautiful last weekend a lot of us had been feeling crabby about the weather, so we are going to do analysis on the crabby.jpg image. Make sure to dowload and save this in the same folder as your markdown file.

```
ColorImg = readJPEG("crabby.jpg")
dim(ColorImg)
```

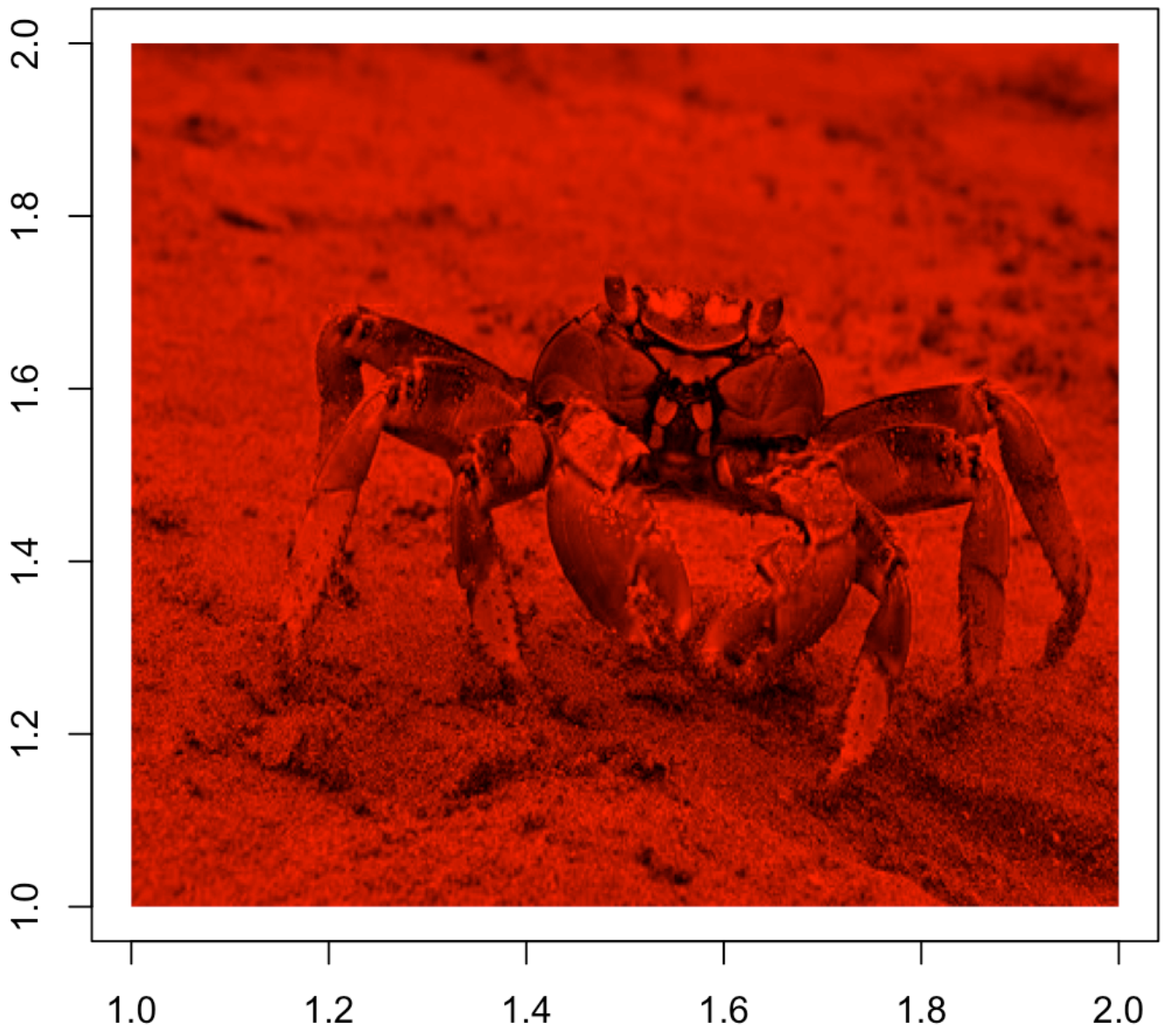
```
## [1] 318 480   3
```

`ColorImg` is a $318 \times 480 \times 3$ data cube. Think about what each entry is recording. Looking at a few images may give you a hint (Carefully look over the code below). The `imPlot` function can plot an image:

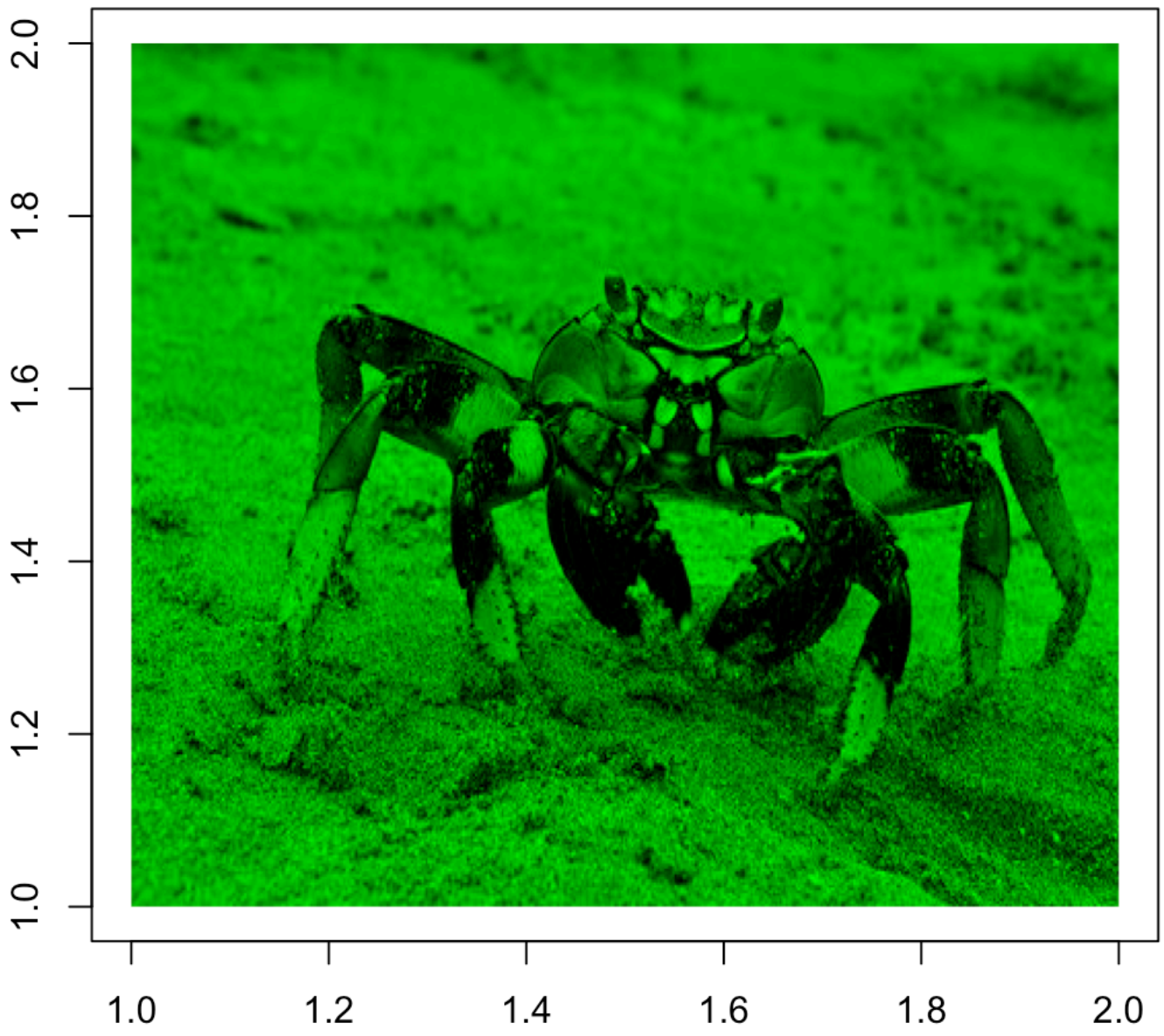
```
imPlot = function(img,...) {
  plot(1:2, type='n',xlab=" ",ylab= " ",...)
  rasterImage(img, 1.0, 1.0, 2.0, 2.0)
}

imPlot(ColorImg,main="Crabby")
```



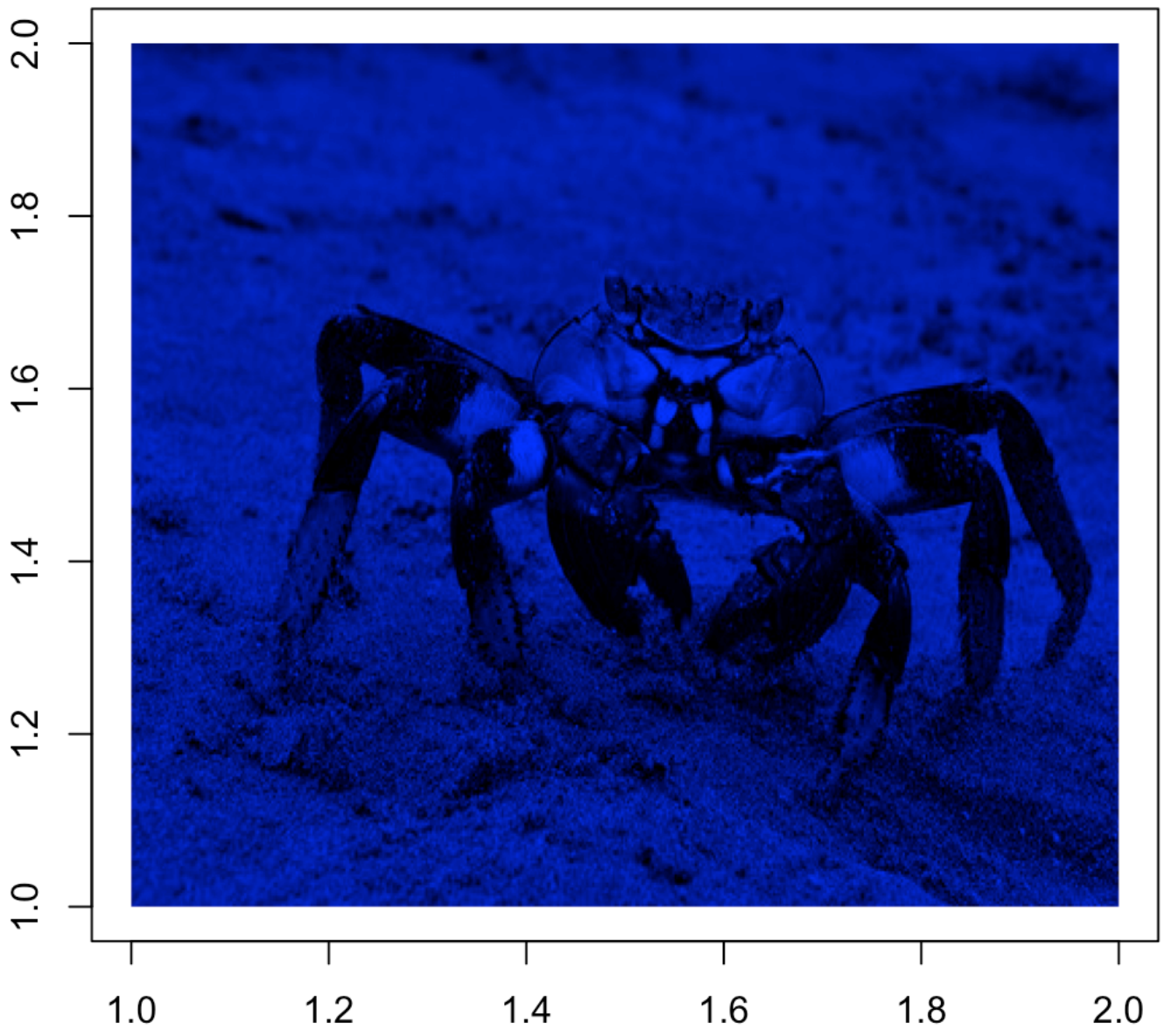

```
Red=ColorImg
Red[,,2]=Red[,,3]=0
imPlot(Red,main="Red Crabby")
```




```
Green=ColorImg
Green[,,1]=Green[,,3]=0
imPlot(Green,main="Green Crabby")
```

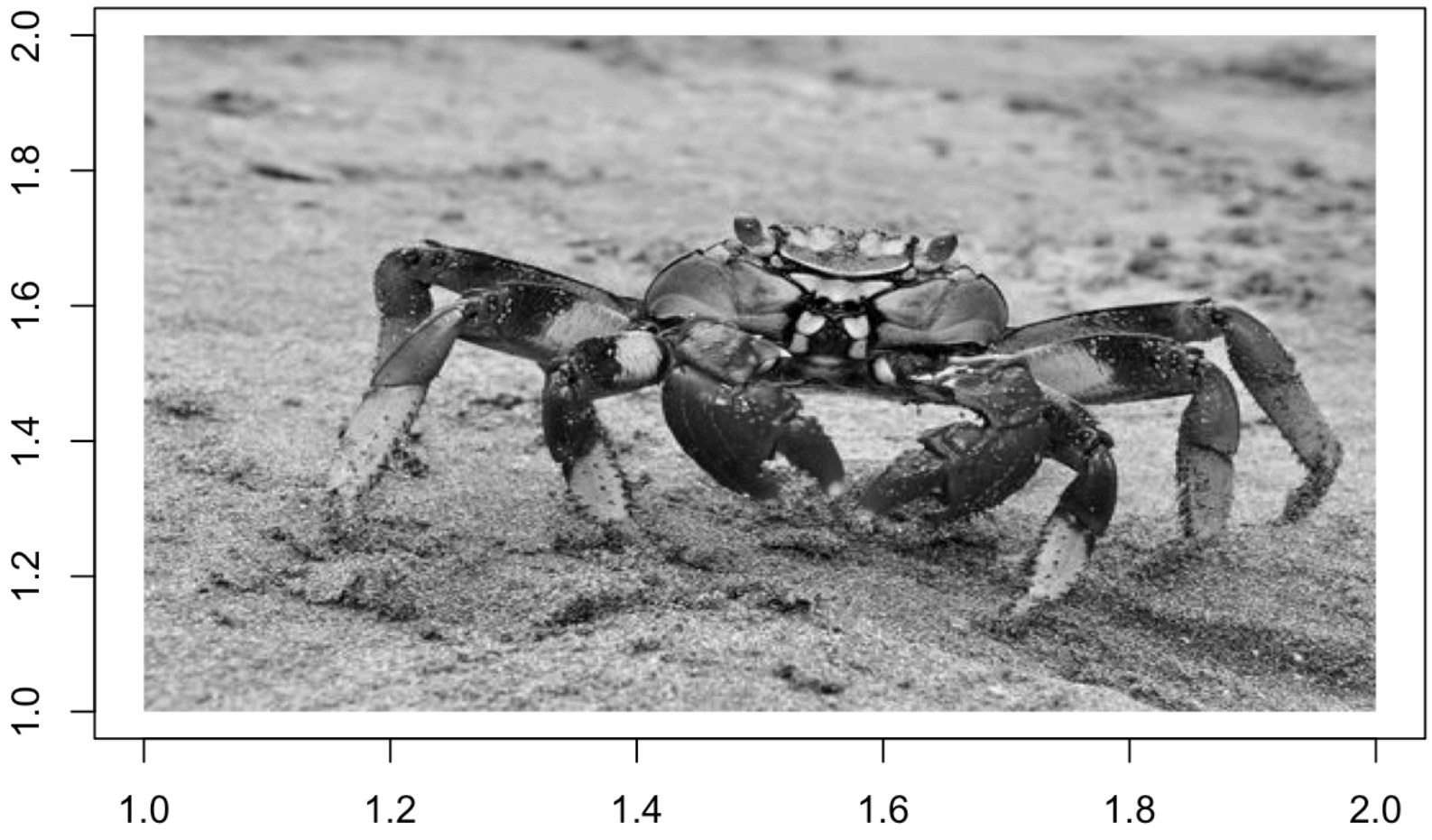



```
Blue=ColorImg
Blue[,,1]=Blue[,,2]=0
imPlot(Blue,main="Blue Crabby")
```



Each of the three sheets corresponds to the amount of red, green, and blue values within an image! Let's convert this color image to a grayscale image (How would below change if we were instead to do the analysis on the color image?). Think what the following is doing:

```
img=0.2989*ColorImg[,,1]+0.5870*ColorImg[,,2]+0.1140*ColorImg[,,3]
imPlot(img,main="Crabby")
```



To store the image, we need to store 152,640 floating point numbers. Our objective in this section is to compress this image using the SVD.

a. Write a function `approxImg` that uses your `SVDApprox` function to do the following:

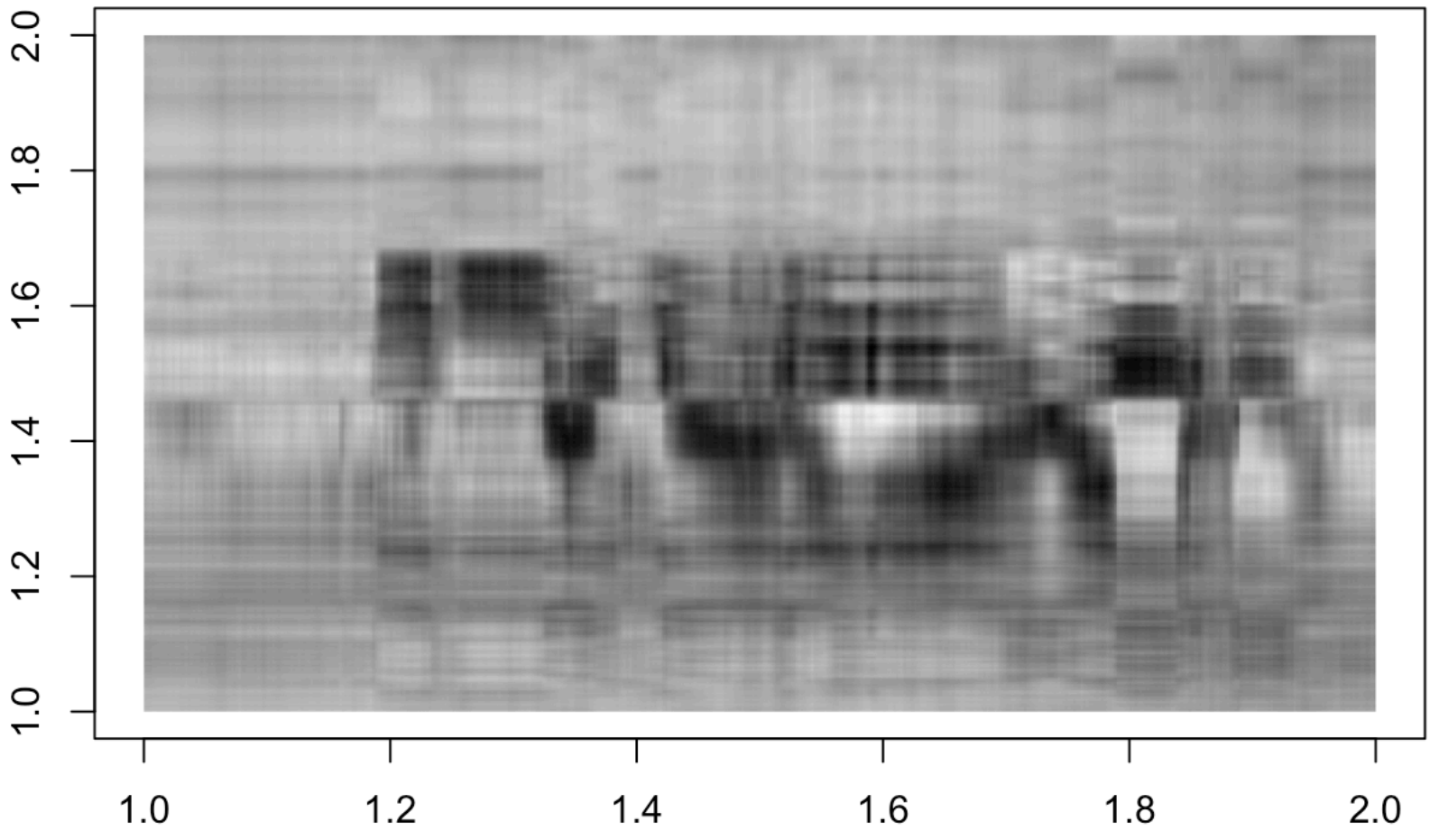
```
SVDApprox = function(A, k){
  out = svd(A)
  u = out$u
  v = out$v
  o = out$d

  #Ak = o[1:k]*u[,1:k]%*%t(v[,1:k])

  for(i in 1:k) {
    if(i == 1) {
      Ak = o[1]*u[,1]%*%t(v[,1])
    } else {
      Ak = Ak + o[i]*u[,i]%*%t(v[,i])
    }
  }
  return(Ak)
}
```

- Takes an image and a rank k as the two inputs
- Computes the best rank k approximation to the image
- Thresholds the approximation by setting all values below zero back to zero and all values above one back to one
- Plots the approximate image

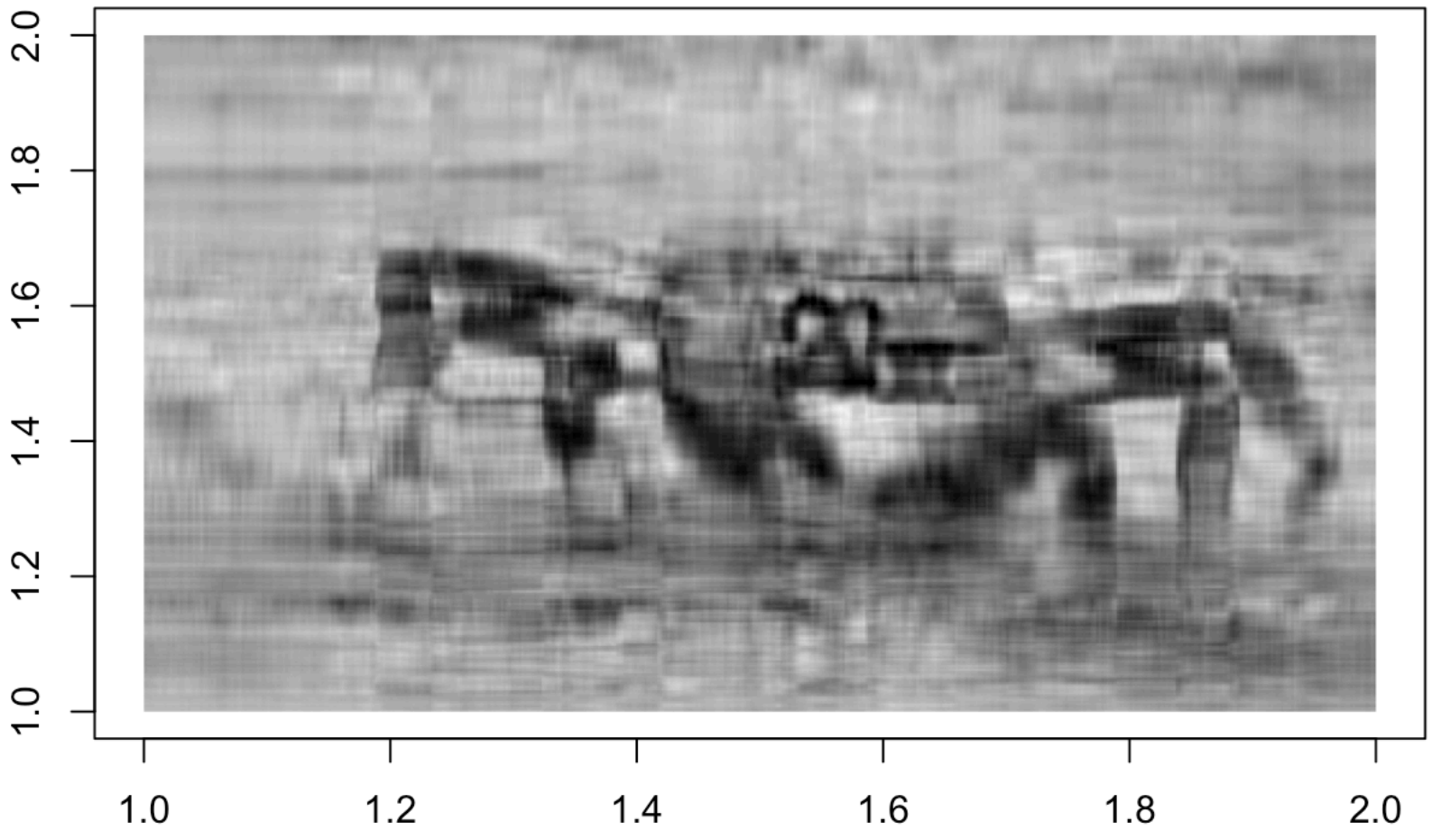
```
approxImg = function(image, k) {
  approx = SVDApprox(image, k)
  for(i in 1:nrow(approx)) {
    for(j in 1:ncol(approx)) {
      if(approx[i, j] > 1) {
        approx[i, j] = 1
      }
      if(approx[i, j] < 0) {
        approx[i, j] = 0
      }
    }
  }
  return(approx)
}
```

b. Test your `approxImg` function on the Crabby image with $k = 5, 10, 25, 50, 100$.

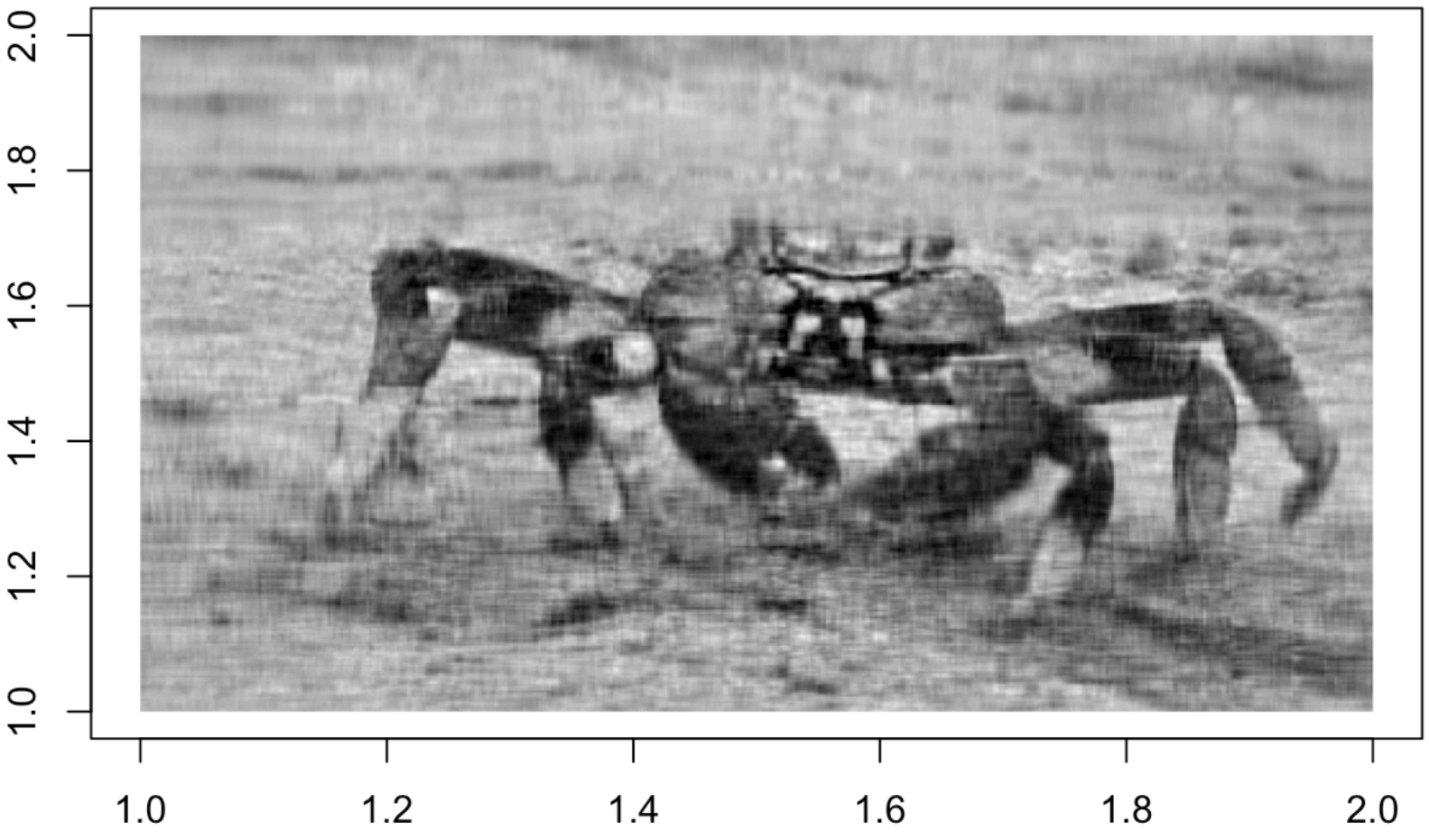
```
k5 = approxImg(img, 5)
imPlot(k5)
```



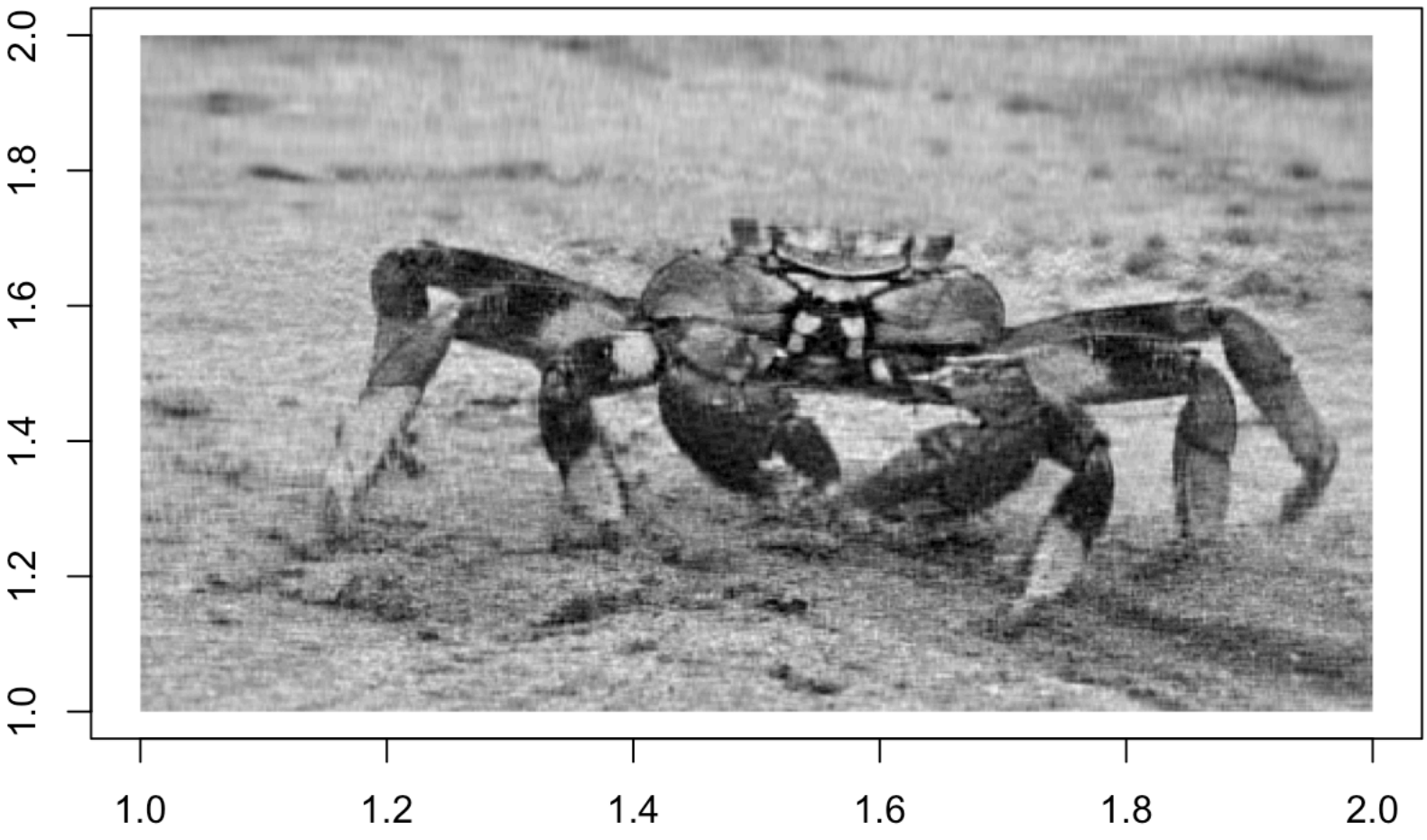

```
k10 = approxImg(img, 10)
imPlot(k10)
```



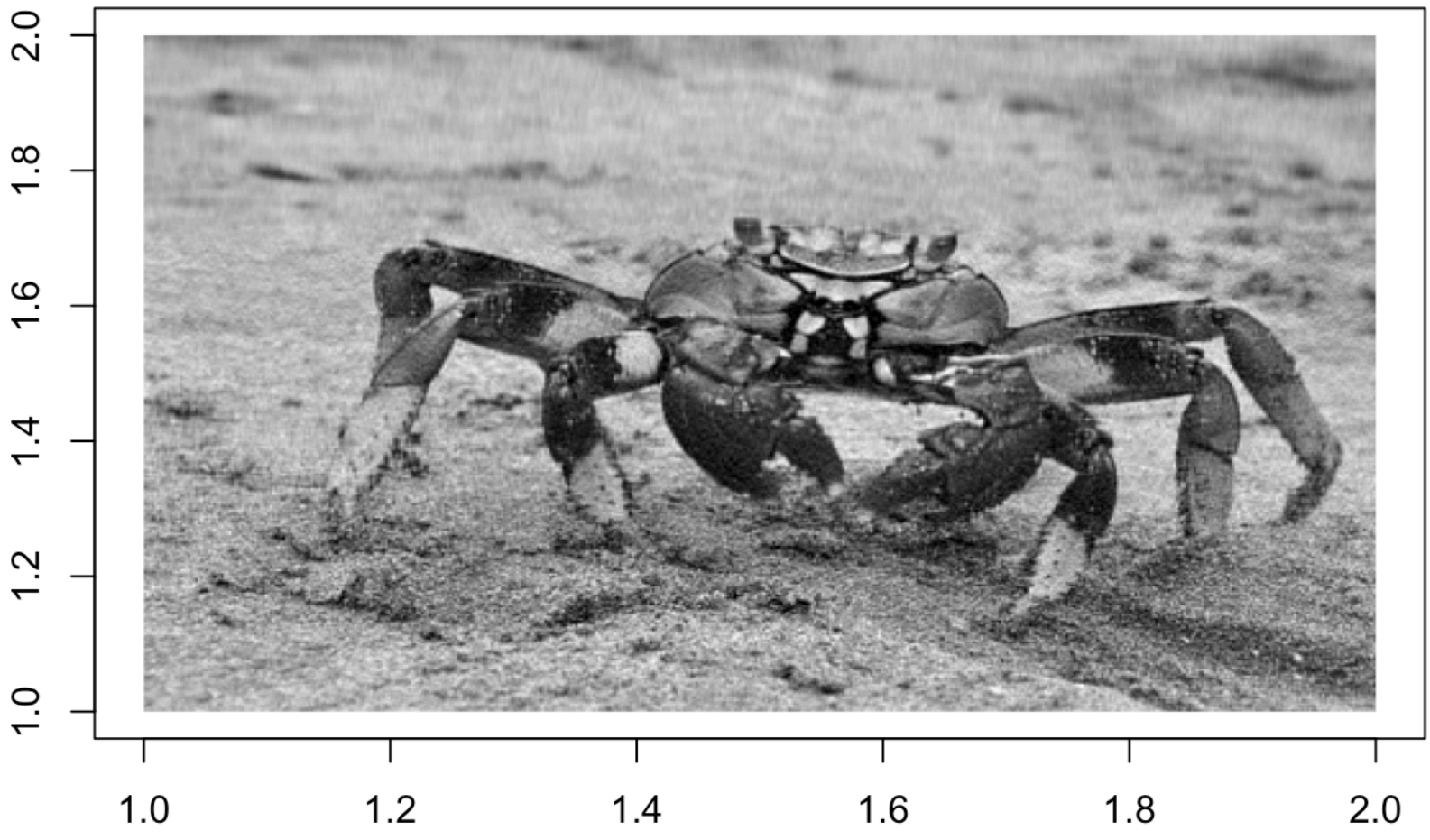

```
k25 = approxImg(img, 25)
imPlot(k25)
```




```
k50 = approxImg(img, 50)
imPlot(k50)
```




```
k100 = approxImg(img, 100)
imPlot(k100)
```



c. For each approximation level k, how many floating point numbers do you need to store? For each k, compute the compression ratio, which is the number of floating point numbers needed to store the approximate image divided by the number of floating point numbers needed to store the original image .

```
m = 318
n = 480

k = c(5, 10, 25, 50, 100)

for(i in 1:length(k)) {
  print("Numbers needed to store: ")
  print((m+n+1)*k[i])
  print("Compression Ratio: ")
  print(((m+n+1)*k[i])/(m*n))
}
```

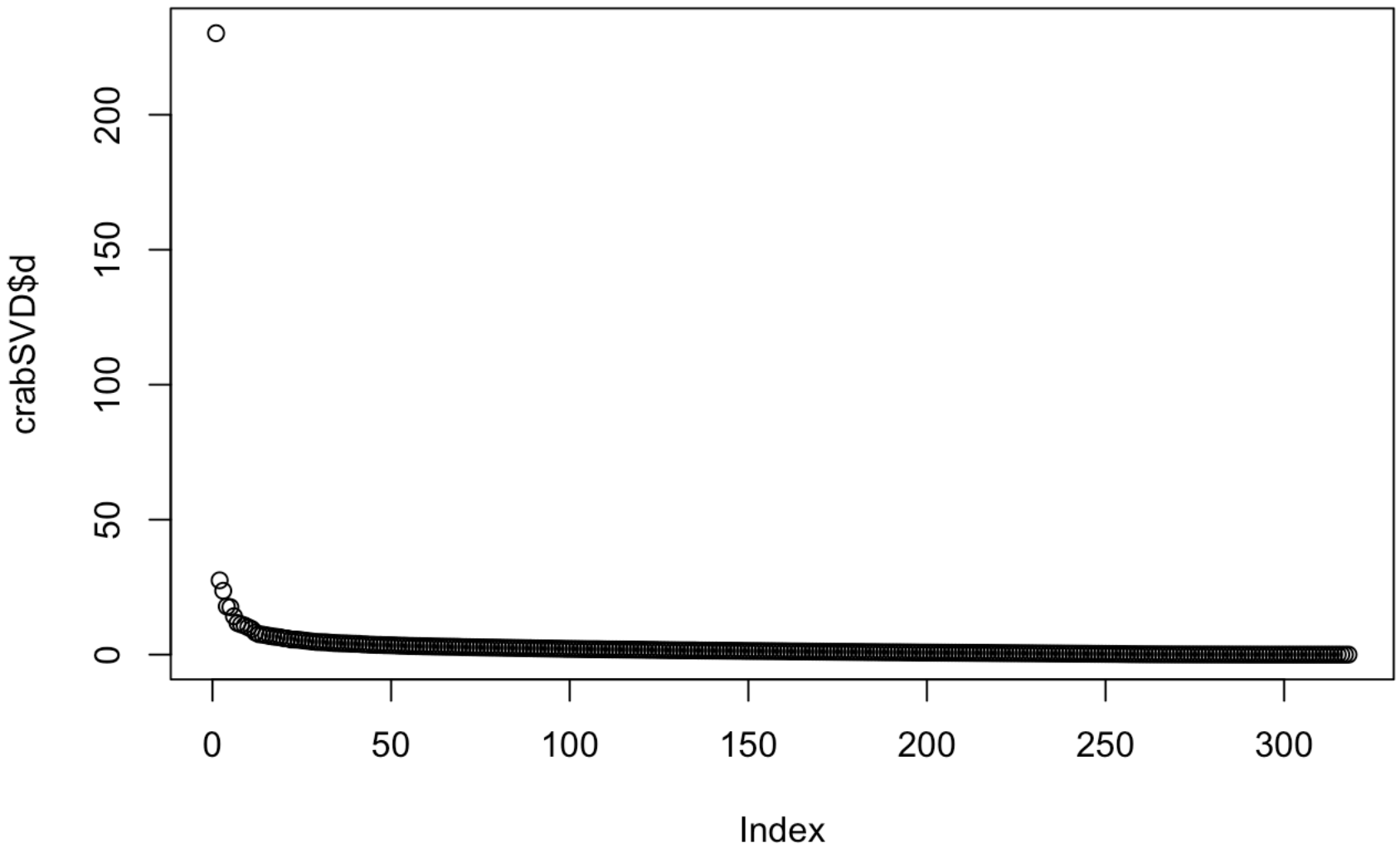
```
## [1] "Numbers needed to store: "
## [1] 3995
## [1] "Compression Ratio: "
## [1] 0.02617269
## [1] "Numbers needed to store: "
## [1] 7990
## [1] "Compression Ratio: "
## [1] 0.05234539
## [1] "Numbers needed to store: "
## [1] 19975
## [1] "Compression Ratio: "
## [1] 0.1308635
## [1] "Numbers needed to store: "
## [1] 39950
## [1] "Compression Ratio: "
## [1] 0.2617269
## [1] "Numbers needed to store: "
## [1] 79900
## [1] "Compression Ratio: "
## [1] 0.5234539
```

d. Recall that the optimal approximation error by a rank k matrix (with the error measured by the Frobenius norm) is $||E_k||_F = ||A - A_k||_F = \sum_{i=k+1}^{r} \sigma_i^2$‘. We can use this to define the *relative error* as

$$\left(\frac{\sum_{i=k+1}^{r} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \right)^{\frac{1}{2}}.$$

Note that a higher proportion of energy in the first k singular values leads to a lower relative error. Therefore, matrices A that have a faster decay in the singular values will be easier to approximate by lower rank matrices. Plot the singular values of the Crabby image. Then compute the relative error for $k = 5, 10, 25, 50, 100$.

```
crabSVD = svd(img)

plot(crabSVD$d)
```




```
k = c(5, 10, 25, 50, 100)
r = ncol(crabSVD$u) #rank of A

for(i in 1:length(k)) {
  upper = 0
  for(j in (k[i]+1):r) { #Rank of A
    upper = upper + crabSVD$d[j]^2
  }
  lower = 0
  for(m in 1:r) {
    lower = lower + crabSVD$d[m]^2
  }
  print("Reletive error: " )
  print(sqrt(upper/lower))
}
```

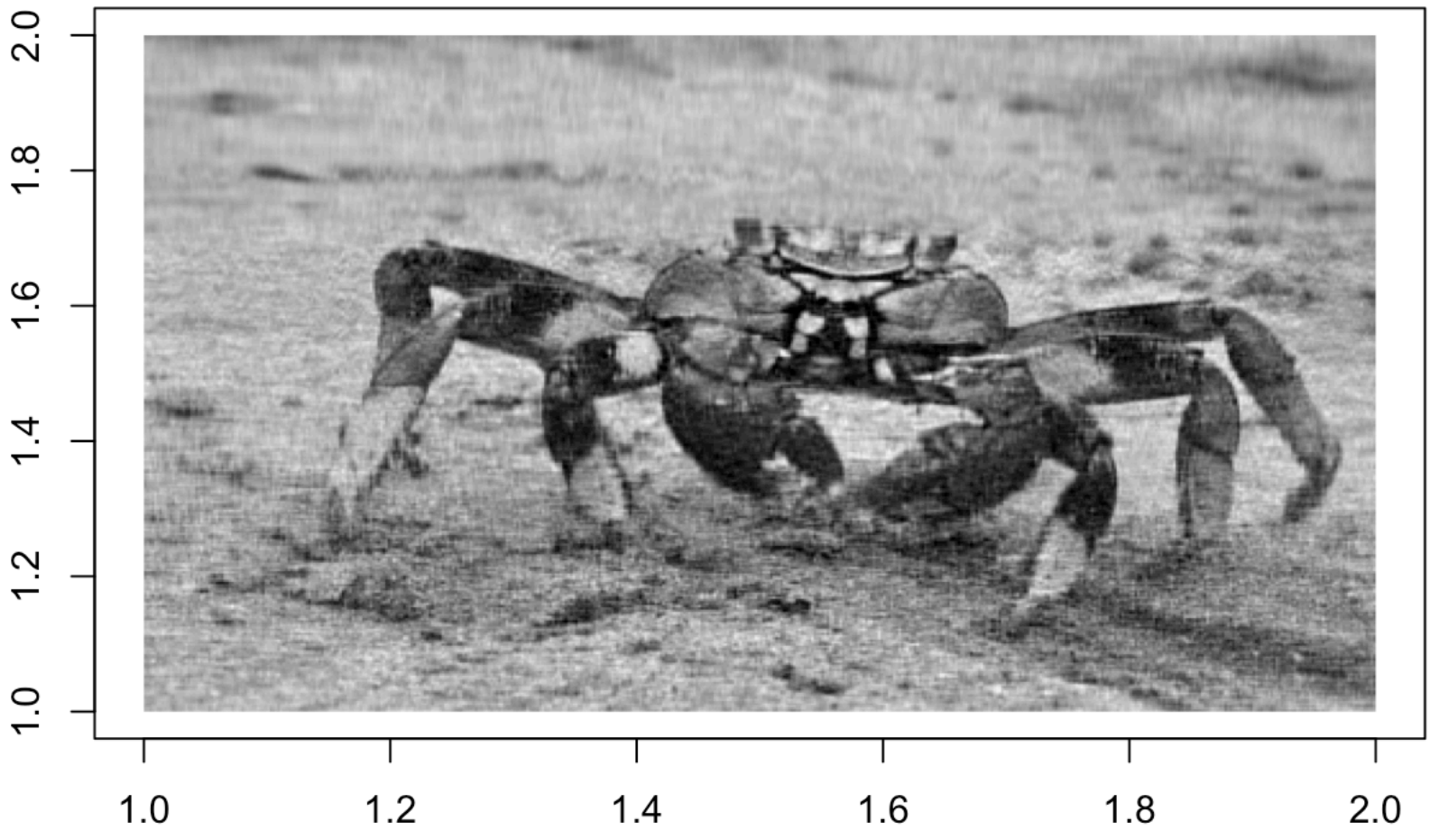
```
## [1] "Reletive error: "
## [1] 0.2047064
## [1] "Reletive error: "
## [1] 0.1731957
## [1] "Reletive error: "
## [1] 0.1340124
## [1] "Reletive error: "
## [1] 0.1017488
## [1] "Reletive error: "
## [1] 0.0622827
```

e. One method to decide on the approximation rank is to choose k such that the relative error is below a given threshold (say 31.6%, which corresponds to having 90% of the energy of all r singular values in the first k singular values). For the Crabby image, let's choose the threshold to be 99.5% of energy, so find k^* such that $\sum_{i=1}^{k^*} \sigma_i^2 \geq 0.995^2 * \sum_{i=1}^{r} \sigma_i^2$, and plot the best approximation of the image with rank k^*. Compute the relative error for k^*.

```
sum = 0
for(i in 1:r) {
  sum = sum + crabSVD$d[i]^2
}
threshold = sum * 0.995^2

kStar = 0
sigmaK = 0
while(sigmaK < threshold) {
  kStar = kStar + 1
  sigmaK = 0
  for(i in 1:kStar) {
    sigmaK = sigmaK + crabSVD$d[i]^2
  }
}

kStar
```

```
## [1] 52
```

```
k52 = approxImg(img, 52)
imPlot(k52)
```




```
upper = 0
for(j in 53:r) { #Rank of A
  upper = upper + crabSVD$d[j]^2
}
lower = 0
for(m in 1:r) {
  lower = lower + crabSVD$d[m]^2
}
print("Reletive error for k = 52: " )
```

```
## [1] "Reletive error for k = 52: "
```

```
print(sqrt(upper/lower))
```

```
## [1] 0.09975444
```

Problem 6

Note: This is Exercise 3 from Activity 26

The SVD can be used to take a two-dimensional or three-dimensional snapshot of high-dimensional data, so that dimensionally-challenged human beings can see it. In this problem, you will use the top two singular values to project some data down to 2-dimensional space where you can see it.

Here is a data set on cereals:

```
require(foreign)
```

```
cereal = read.dta("http://statistics.ats.ucla.edu/stat/data/cerealnut.dta")
A =  as.matrix(cereal[,2:9])
print(A)
```

```
##    calories protein fat  Na fiber carbs sugar   K
## 1       110       6   2 290   2.0  17.0     1 105
## 2       110       1   1 180   0.0  12.0    13  55
## 3       110       3   1 250   1.5  11.5    10  90
## 4       110       2   1 260   0.0  21.0     3  40
## 5       110       2   1 180   0.0  12.0    12  55
## 6       130       3   2 170   1.5  13.5    10 120
## 7       100       3   2 140   2.5  10.5     8 140
## 8       110       2   1 200   0.0  21.0     3  35
## 9       140       3   1 190   4.0  15.0    14 230
## 10      110       1   1 140   0.0  13.0    12  25
## 11      110       2   1 200   1.0  16.0     8  60
## 12       70       4   1 260   9.0   7.0     5 320
## 13      110       2   0 125   1.0  11.0    14  30
## 14      100       2   0 290   1.0  21.0     2  35
## 15      110       1   0  90   1.0  13.0    12  20
## 16      160       3   2 150   3.0  17.0    13 160
## 17      120       2   1 190   0.0  15.0     9  40
## 18      140       3   2 220   3.0  21.0     7 130
## 19       90       3   0 170   3.0  18.0     2  90
## 20      100       3   0 320   1.0  20.0     3  45
## 21      120       3   1 210   5.0  14.0    12 240
## 22      110       2   0 290   0.0  22.0     3  35
## 23      110       6   0 230   1.0  16.0     3  55
## 24      100       4   2 150   2.0  12.0     6  95
## 25       50       1   0   0   0.0  13.0     0  15
```

To perform this projection, we work with the covariance matrix C of A. Compute this as follows:

- For each column of A subtract off the mean of that column. Then each entry is the difference from the mean of that feature

```
cereal = read.dta("http://statistics.ats.ucla.edu/stat/data/cerealnut.dta")
A =  as.matrix(cereal[,2:9])
print(A)
```

```
##    calories protein fat  Na fiber carbs sugar   K
## 1       110       6   2 290   2.0  17.0     1 105
## 2       110       1   1 180   0.0  12.0    13  55
## 3       110       3   1 250   1.5  11.5    10  90
## 4       110       2   1 260   0.0  21.0     3  40
## 5       110       2   1 180   0.0  12.0    12  55
## 6       130       3   2 170   1.5  13.5    10 120
## 7       100       3   2 140   2.5  10.5     8 140
## 8       110       2   1 200   0.0  21.0     3  35
## 9       140       3   1 190   4.0  15.0    14 230
## 10      110       1   1 140   0.0  13.0    12  25
## 11      110       2   1 200   1.0  16.0     8  60
## 12       70       4   1 260   9.0   7.0     5 320
## 13      110       2   0 125   1.0  11.0    14  30
## 14      100       2   0 290   1.0  21.0     2  35
## 15      110       1   0  90   1.0  13.0    12  20
## 16      160       3   2 150   3.0  17.0    13 160
## 17      120       2   1 190   0.0  15.0     9  40
## 18      140       3   2 220   3.0  21.0     7 130
## 19       90       3   0 170   3.0  18.0     2  90
## 20      100       3   0 320   1.0  20.0     3  45
## 21      120       3   1 210   5.0  14.0    12 240
## 22      110       2   0 290   0.0  22.0     3  35
## 23      110       6   0 230   1.0  16.0     3  55
## 24      100       4   2 150   2.0  12.0     6  95
## 25       50       1   0   0   0.0  13.0     0  15
```

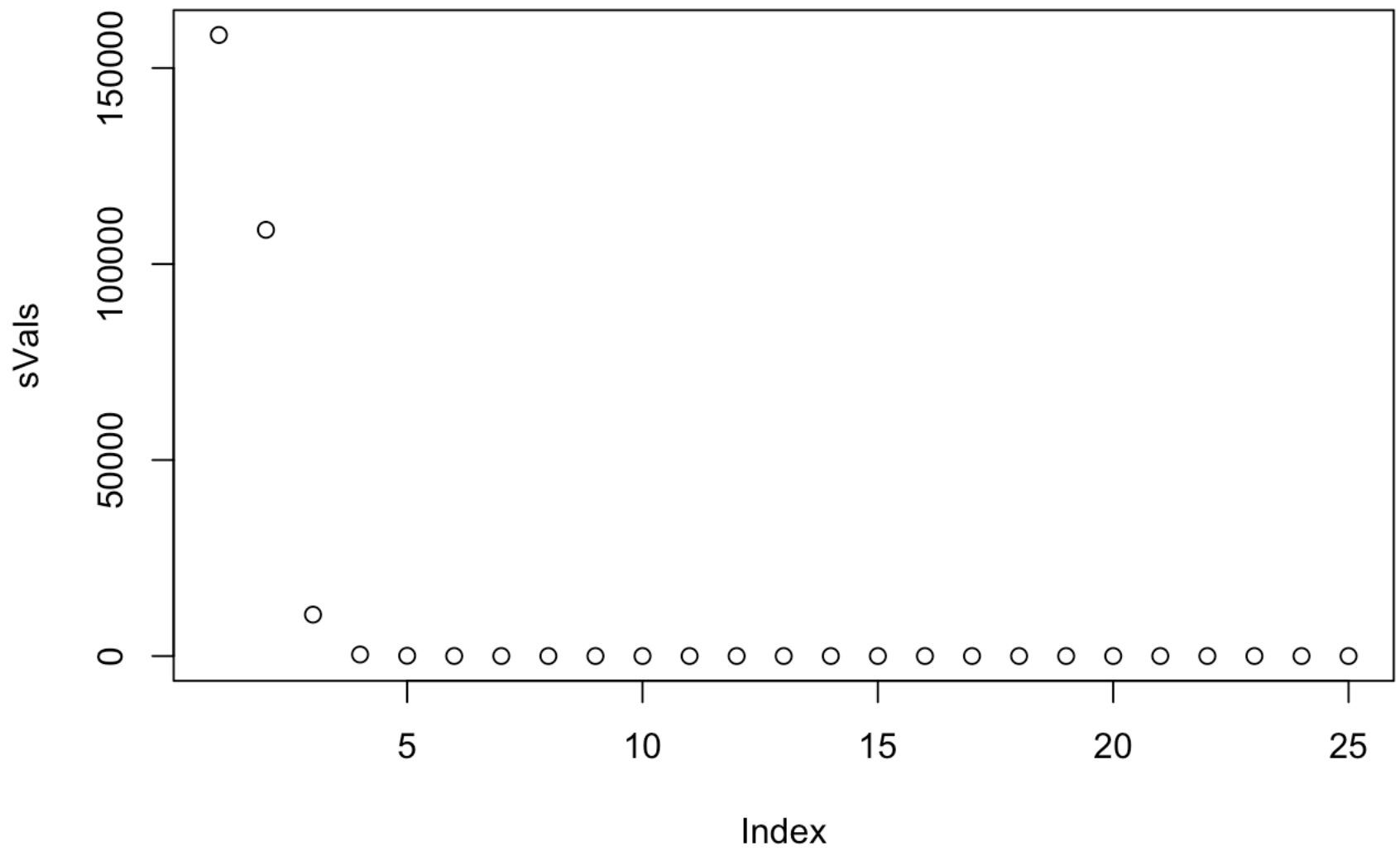
```
for(i in 1:ncol(A)) {
  A[,i] = A[,i] - sum(A[,i])/nrow(A)
}
```

- Now compute the matrix $C = AA^T$. This is the covariance matrix. It measures how well each of the subjects are correlated. The ij-entry is the dot product of cereal i's data with cereal j's data, so it is (roughly) the cosine of the angle between them

```
C = A%*%t(A)
```

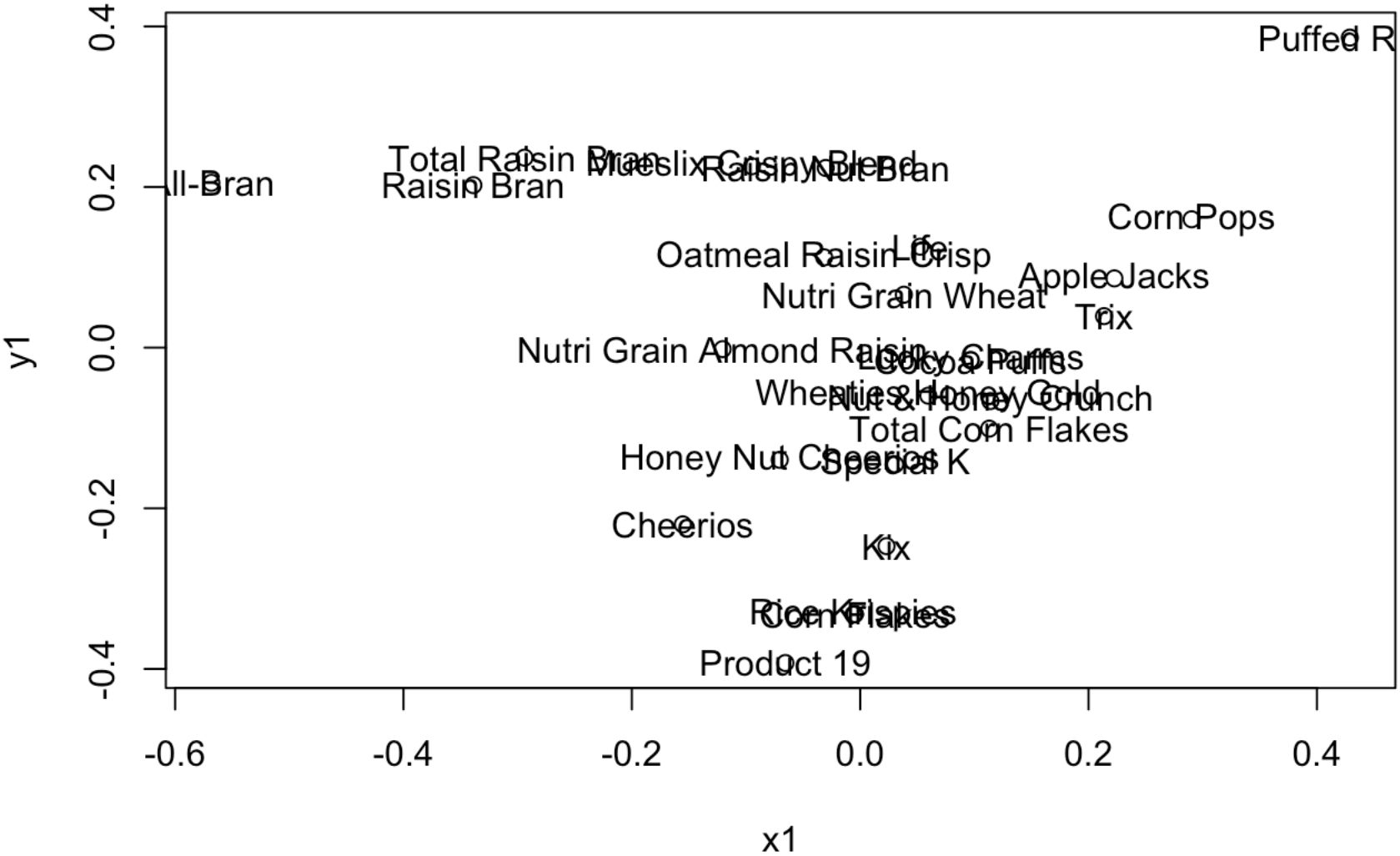
a. Plot the singular values of the matrix C. You should see that there are 2 singular values that stand out from the rest.

```
# out = eigen(C)
# vals = out$values
# sVals = sqrt(vals)
# sVals
# plot(sVals)

sOut = svd(C)
sVals = sOut$d
plot(sVals)
```



b. Use the vectors $x = v_1$ and $y = v_2$ from the SVD as the x and y coordinates of points in the plane. Plot these points. Label the i^{th} point with the i^{th} brand of cereal. To do this, you can use the following command after your plotting command:

```
#sOut = svd(C)
x1 = sOut$v[,1]
y1 = sOut$v[,2]

plot(x1, y1)
text(x1, y1, label = cereal$brand)
```



c. This method should group like cereals next to one another in the plane. Discuss whether you think this is happening.

I think that this method groups like cereals next to one another. By looking at the plot, we can see that the top left seems to be sugary cereals, while the bottom seems to be plain cereals. The left also seems to be mostly brans and the cluster in the middle is almost exclusively wheat cereals. As a result, similar cereals seem to be grouped near one another.

d. Note: You could also have used the SVD of $A^\top$ instead of the SVD of C. Why is this? Provide a quick proof.

We can just use $A^\top$ instead of C because this SVD cereals decomposition relies on the V eigenvectors, which will be the same in the SVD for $A^\top$ as the SVD in C.