Statistical Computing

Lecture 1: R basics

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Objectives

- Overview of R
- R nuts and bolts
- Getting data in and out of R
- Subsetting R objects

Overview of R

What is R?

- A freely available language and environment
- Statistical computing and graphics
- Linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc.

Installation

- Install R
- Install Rstudio

Why Rstudio?

- Syntax highlighting
- Able to evaluate R code
  - by line
  - by selection
  - entire file
- Command auto-completion

Design of the R System

- When you download R from CRAN, you get the "base" system - a substantial amount of functionality.

- 10,000 packages on CRAN that have been developed by users and programmers around the world.

- People often make packages available on their personal websites.

- There are a number of packages being developed on repositories like GitHub and BitBucket.

R Nuts and Bolts

Basic Operations

```
1 + 2 + 3
## [1] 6
1 + 2 * 3
## [1] 7

x <- 1
y <- 2
z <- c(x, y)
z
## [1] 1 2

exp(1)
## [1] 2.718282
cos(3.141593)
## [1] -1
log2(1)
## [1] 0
```

R Objects

R has five basic classes of objects:

1. character
2. numeric (real numbers)
3. integer
4. complex
5. logical (True/False)

Numbers

- Numbers in R are generally treated as numeric objects.
- Difference of `1` and `1L`?
- Special number `Inf`. Try `1/Inf`.
- `NaN`: an undefined value (not a number). Try `0/0`. It can also be thought of as a missing value.

Attributes

Attributes can be accessed by `attributes()`. Some examples of R object attributes are:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length

Vectors

The `c()` function can be used to create vectors of objects by concatenating things together.

```
x <- c(0.5, 0.6)  ## numeric
x <- c(TRUE, FALSE)  ## logical
x <- c(T, F)  ## logical
x <- c("a", "b", "c")  ## character
x <- 9:29  ## integer
x <- c(1 + (0+0i), 2 + (0+4i))  ## complex
```

You can also use the `vector()` function to initialize vectors.

```
x <- vector("numeric", length = 10)
x
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Matrices

```
m <- matrix(c(1:6), 2, 3)
attributes(m)
## $dim
## [1] 2 3
dim(m)
## [1] 2 3
t(m)
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
m[1, 2]
## [1] 3
m[1, ]
## [1] 1 3 5
n <- matrix(c(8:13), 2, 3)
cbind(m, n)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    8   10   12
## [2,]    2    4    6    9   11   13
rbind(m, n)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    8   10   12
## [4,]    9   11   13
```

Lists

- Special data structure that matrix could not handle.
  - Data length are not the same.
  - Data type are not the same.

```
l <- list(a = c(1, 2), b = "apple")
attributes(l)
## $names
## [1] "a" "b"
```

Factors

Factors are used to represent categorical data.

```
f <- factor(c("yes", "yes", "no", "yes", "no"))
attributes(f)
## $levels
## [1] "no"  "yes"
##
## $class
## [1] "factor"
```

Data Frames

- A special type of list.
- Unlike matrices – data frames can store different classes of objects in each column.
- They have column names and row names.

```
d <- data.frame(x = 1:10, y = letters[1:10])
attributes(d)
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
names(d)
## [1] "x" "y"
row.names(d)
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Names

Names are very useful for writing readable code and self-describing objects.

```
x <- 1:3
names(x)
## NULL
names(x) <- c("New York", "Seattle", "Los Angeles")
x
##    New York     Seattle Los Angeles
##           1           2           3
names(x)
## [1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
x <- list(`Los Angeles` = 1, Boston = 2, London = 3)
x
## $`Los Angeles`
## [1] 1
##
```

```
## $Boston
## [1] 2
##
## $London
## [1] 3
names(x)
## [1] "Los Angeles" "Boston"      "London"
```

Getting Data in and out of R

Reading and Writing Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces

There are analogous functions for writing data to files.

- `write.table`, for writing tabular data to text files (i.e. CSV) or connections
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a files

There are many R packages that have been developed to read in all kinds of other datasets (e.g., the `readr` package).

Subsetting R objects

How to Subset?

There are three operators that can be used to extract subsets of R objects.

- The `[` operator always returns an object of the same class as the original. It can be used to select multiple elements of an object

- The `[[` operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.

- The `$` operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of `[[`.

Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the `[` operator.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]  ## Extract the first element
## [1] "a"
```

```
x[2]  ## Extract the second element
## [1] "b"
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
x[1:4]
## [1] "a" "b" "c" "c"
x[c(1, 3, 4)]
## [1] "a" "c" "c"
x[x > 2]
## [1] "a" "b" "c" "c" "d" "a"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (*i*,*j*) type indices.

```
x <- matrix(1:6, 2, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

We can access the $(1,2)$ or the $(2,1)$ element of this matrix using the appropriate indices.

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

Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
x[1, ]  ## Extract the first row
## [1] 1 3 5
x[, 2]  ## Extract the second column
## [1] 3 4
```

Subsetting Lists

ists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
x <- list(foo = 1:4, bar = 0.6)
x
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
```

The [[operator can be used to extract *single* elements from a list. Here we extract the first element of the list.

```
x[[1]]
## [1] 1 2 3 4
```

The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
x[["bar"]]
## [1] 0.6
x$bar
## [1] 0.6
```

Subsetting Nested Elements of a List

The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
## Get the 3rd element of the 1st element
x[[c(1, 3)]]
## [1] 14
## Same as above
x[[1]][[3]]
## [1] 14
## 1st element of the 2nd element
x[[c(2, 1)]]
## [1] 3.14
```

Extracting Multiple Elements of a List

The [operator can be used to extract *multiple* elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

Note that `x[c(1, 3)]` is NOT the same as `x[[c(1, 3)]]`.

Remember that the [operator always returns an object of the same class as the original. Since the original object was a list, the [operator returns a list. In the above code, we returned a list with two elements (the first and the third).

Removing NA Values

A common task in data analysis is removing missing values (`NA`s).

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
print(bad)
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
x[!bad]
## [1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
head(airquality)
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
good <- complete.cases(airquality)
head(airquality[good, ])
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Review of this lecture

- Overview of R
- R nuts and bolts
- Getting data in and out of R
- Subsetting R objects

Lab Session 1

Read and Write Data in R

You'll be working with swimming_pools.csv; it contains data on swimming pools in Brisbane, Australia (Source: data.gov.au). The file contains the column names in the first row. It uses a comma to separate values within rows.

1. Try `read.csv()` and `read.table()` to import "swimming_pools.csv" as a data frame with the name `pools`.
2. Try `write.table()`, `dput()`, and `save()` functions to write `pools` to files.
3. Restart R and read your saved data in R.
4. Practice subsetting of a data frame.

References

Chapters 3-10 of the book "R programming for data science".

Statistical Computing

Lecture 2: Managing data frames with the `dplyr` package

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Data Frames

- *data frame* is a key data structure in statistics and in R.
- one observation per row and each column represents a variable
- we need to have good tools for dealing with them.
- you have seen `subset()` function and the use of `[` and `$`
- `dplyr` package is designed to mitigate a lot of complex operations for data frames.

The `dplyr` Package

- by Hadley Wickham of RStudio
- everything `dplyr` does could already be done with base R, but it *greatly* simplifies existing functionality in R.
- it provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames.
- the `dplyr` functions are **very** fast, as many key operations are coded in C++.

`dplyr` Grammar

Some of the key "verbs" provided by the `dplyr` package are

- `select`: return a subset of the columns of a data frame, using a flexible notation
- `filter`: extract a subset of rows from a data frame based on logical conditions
- `arrange`: reorder rows of a data frame
- `rename`: rename variables in a data frame
- `mutate`: add new variables/columns or transform existing variables
- `summarise` / `summarize`: generate summary statistics of different variables in the data frame, possibly within strata
- `%>%`: the "pipe" operator is used to connect multiple verb actions together into a pipeline

Common `dplyr` Function Properties

All of the functions have a few common characteristics. In particular,

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
3. The return result of a function is a new data frame.
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

Installing the dplyr package

```
install.packages("dplyr")
```

After installing the package it is important that you load it into your R session with the `library()` function.

```
library(dplyr)
##
## 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
```

select()

We will use a dataset containing air pollution and temperature data for the city of Chicago in the U.S.

```
chicago <- readRDS("chicago.rds")
```

```
dim(chicago)
## [1] 6940    8
str(chicago)
## 'data.frame':    6940 obs. of  8 variables:
##  $ city      : chr  "chic" "chic" "chic" "chic" ...
##  $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
##  $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
##  $ date      : Date, format: "1987-01-01" "1987-01-02" ...
##  $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
##  $ pm10tmean2: num  34 NA 34.2 47 NA ...
##  $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
##  $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Sometimes you may want to use only a couple of variables out of many.

```
names(chicago)[1:3]
## [1] "city" "tmpd" "dptp"
subset <- select(chicago, city:dptp)
head(subset)
##   city tmpd   dptp
## 1 chic 31.5 31.500
## 2 chic 33.0 29.875
## 3 chic 33.0 27.375
## 4 chic 29.0 28.625
## 5 chic 32.0 28.875
## 6 chic 40.0 35.125
```

Sometimes you may want to drop some variables that are not useful.

```
select(chicago, -(city:dptp))
```

If you wanted to keep every variable that ends with a “2”, we could do

```
subset <- select(chicago, ends_with("2"))
str(subset)
## 'data.frame':    6940 obs. of  4 variables:
##  $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
##  $ pm10tmean2: num  34 NA 34.2 47 NA ...
##  $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
##  $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Or if we wanted to keep every variable that starts with a "d", we could do

```
subset <- select(chicago, starts_with("d"))
str(subset)
## 'data.frame':    6940 obs. of  2 variables:
##  $ dptp: num  31.5 29.9 27.4 28.6 28.9 ...
##  $ date: Date, format: "1987-01-01" "1987-01-02" ...
```

`filter()`

The `filter()` function is used to extract subsets of rows from a data frame.

```
chic.f <- filter(chicago, pm25tmean2 > 30)
str(chic.f)
## 'data.frame':    194 obs. of  8 variables:
##  $ city      : chr  "chic" "chic" "chic" "chic" ...
##  $ tmpd      : num  23 28 55 59 57 57 75 61 73 78 ...
##  $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
##  $ date      : Date, format: "1998-01-17" "1998-01-23" ...
##  $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
##  $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
##  $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
##  $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

```
summary(chic.f$pm25tmean2)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   30.05   32.12   35.04   36.63   39.53   61.50
```

We could for example extract the rows where PM2.5 is greater than 30 *and* temperature is greater than 80 degrees Fahrenheit.

```
chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
select(chic.f, date, tmpd, pm25tmean2)
##          date tmpd pm25tmean2
## 1  1998-08-23   81   39.60000
## 2  1998-09-06   81   31.50000
## 3  2001-07-20   82   32.30000
## 4  2001-08-01   84   43.70000
## 5  2001-08-08   85   38.83750
## 6  2001-08-09   84   38.20000
## 7  2002-06-20   82   33.00000
## 8  2002-06-23   82   42.50000
## 9  2002-07-08   81   33.10000
## 10 2002-07-18   82   38.85000
## 11 2003-06-25   82   33.90000
## 12 2003-07-04   84   32.90000
## 13 2005-06-24   86   31.85714
## 14 2005-06-27   82   51.53750
## 15 2005-06-28   85   31.20000
## 16 2005-07-17   84   32.70000
## 17 2005-08-03   84   37.90000
```

arrange()

The `arrange()` function is used to reorder rows of a data frame according to one of the variables/columns.

Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
chicago <- arrange(chicago, date)
```

We can now check the first few rows

```
head(select(chicago, date, pm25tmean2), 3)
##         date pm25tmean2
## 1 1987-01-01         NA
## 2 1987-01-02         NA
## 3 1987-01-03         NA
```

and the last few rows.

```
tail(select(chicago, date, pm25tmean2), 3)
##            date pm25tmean2
## 6938 2005-12-29    7.45000
## 6939 2005-12-30   15.05714
## 6940 2005-12-31   15.00000
```

Columns can be arranged in descending order too by useing the special `desc()` operator.

```
chicago <- arrange(chicago, desc(date))
```

Looking at the first three and last three rows shows the dates in descending order.

```
head(select(chicago, date, pm25tmean2), 3)
##         date pm25tmean2
## 1 2005-12-31   15.00000
## 2 2005-12-30   15.05714
## 3 2005-12-29    7.45000
tail(select(chicago, date, pm25tmean2), 3)
##            date pm25tmean2
## 6938 1987-01-03         NA
## 6939 1987-01-02         NA
## 6940 1987-01-01         NA
```

How would you do this in base R without `dplyr`?

rename()

Renaming a variable in a data frame in R is surprisingly hard to do! The `rename()` function is designed to make this process easier.

Here you can see the names of the first five variables in the `chicago` data frame.

```
head(chicago[, 1:5], 3)
##   city tmpd dptp       date pm25tmean2
## 1 chic   35 30.1 2005-12-31   15.00000
## 2 chic   36 31.0 2005-12-30   15.05714
## 3 chic   35 29.4 2005-12-29    7.45000
```

Now we rename the awkward variable names.

```
chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
head(chicago[, 1:5], 3)
##   city tmpd dewpoint       date     pm25
## 1 chic   35     30.1 2005-12-31 15.00000
## 2 chic   36     31.0 2005-12-30 15.05714
## 3 chic   35     29.4 2005-12-29  7.45000
```

mutate()

The mutate() function exists to compute transformations of variables in a data frame.

For example, with air pollution data, we often want to *detrend* the data by subtracting the mean from the data.

```
chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
head(chicago)
##   city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2
## 1 chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000
## 2 chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556
## 3 chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222
## 4 chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563
## 5 chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000
## 6 chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944
##   pm25detrend
## 1   -1.230958
## 2   -1.173815
## 3   -8.780958
## 4    1.519042
## 5    7.329042
## 6   -7.830958
```

group_by()

The group_by() function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is.

First, we can create a year varible using as.POSIXlt().

```
chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Now we can create a separate data frame that splits the original data frame by year.

```
years <- group_by(chicago, year)
```

Finally, we compute summary statistics for each year in the data frame with the summarize() function.

```
summarize(years, pm25 = mean(pm25, na.rm = TRUE), o3 = max(o3tmean2,
    na.rm = TRUE), no2 = median(no2tmean2, na.rm = TRUE))
## # A tibble: 19 x 4
##     year  pm25    o3   no2
##    <dbl> <dbl> <dbl> <dbl>
##  1  1987 NaN    63.0  23.5
##  2  1988 NaN    61.7  24.5
##  3  1989 NaN    59.7  26.1
##  4  1990 NaN    52.2  22.6
##  5  1991 NaN    63.1  21.4
##  6  1992 NaN    50.8  24.8
##  7  1993 NaN    44.3  25.8
##  8  1994 NaN    52.2  28.5
##  9  1995 NaN    66.6  27.3
## 10  1996 NaN    58.4  26.4
## 11  1997 NaN    56.5  25.5
## 12  1998  18.3  50.7  24.6
## 13  1999  18.5  57.5  24.7
## 14  2000  16.9  55.8  23.5
## 15  2001  16.9  51.8  25.1
## 16  2002  15.3  54.9  22.7
## 17  2003  15.2  56.2  24.6
## 18  2004  14.6  44.5  23.4
## 19  2005  16.2  58.8  22.6
```

`group_by()`

In a slightly more complicated example, we might want to know what are the average levels of ozone (`o3`) and nitrogen dioxide (`no2`) within quintiles of `pm25`. A slicker way to do this would be through a regression model, but we can actually do this quickly with `group_by()` and `summarize()`.

First, we can create a categorical variable of `pm25` divided into quintiles.

```
qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Now we can group the data frame by the `pm25.quint` variable.

```
quint <- group_by(chicago, pm25.quint)
```

Finally, we can compute the mean of `o3` and `no2` within quintiles of `pm25`.

```
summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE), no2 = mean(no2tmean2,
    na.rm = TRUE))
## # A tibble: 6 x 3
##   pm25.quint     o3   no2
##   <fct>       <dbl> <dbl>
## 1 (1.7,8.7]    21.7  18.0
## 2 (8.7,12.4]   20.4  22.1
## 3 (12.4,16.7]  20.7  24.4
## 4 (16.7,22.6]  19.9  27.3
## 5 (22.6,61.5]  20.3  29.6
## 6 <NA>         18.8  25.8
```

`%>%`

The pipeline operater `%>%` is very handy for stringing together multiple `dplyr` functions in a sequence of operations.

```
third(second(first(x)))
```

This nesting is not a natural way to think about a sequence of operations. The `%>%` operator allows you to string operations in a left-to-right fashion, i.e.

```
first(x) %>% second %>% third
```

`%>%`

Take the example that we just did in the last section where we computed the mean of `o3` and `no2` within quintiles of `pm25`. There we had to

1. create a new variable `pm25.quint`
2. split the data frame by that new variable
3. compute the mean of `o3` and `no2` in the sub-groups defined by `pm25.quint`

That can be done with the following sequence in a single R expression.

```
mutate(chicago, pm25.quint = cut(pm25, qq)) %>% group_by(pm25.quint) %>%
    summarize(o3 = mean(o3tmean2, na.rm = TRUE), no2 = mean(no2tmean2,
        na.rm = TRUE))
## # A tibble: 6 x 3
##   pm25.quint     o3   no2
##   <fct>       <dbl> <dbl>
## 1 (1.7,8.7]    21.7  18.0
## 2 (8.7,12.4]   20.4  22.1
## 3 (12.4,16.7]  20.7  24.4
## 4 (16.7,22.6]  19.9  27.3
## 5 (22.6,61.5]  20.3  29.6
## 6 <NA>         18.8  25.8
```

Summary

The `dplyr` package provides a concise set of operations for managing data frames. With these functions we can do a number of complex operations in just a few lines of code. In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of `group_by()` and `summarize()`.

- `dplyr` can work with other data frame "backends" such as SQL databases. There is an SQL interface for relational databases via the DBI package

- `dplyr` can be integrated with the `data.table` package for large fast tables

The `dplyr` package is handy way to both simplify and speed up your data frame management code. It's rare that you get such a combination at the same time!

Lab Session 1 Cont'd

`dplyr`

You'll be working with the `airquality` in the R package `datasets`. Bear in mind `%>%`.

1. Please return all the rows where Temp is larger than 80 and Month is after May.
2. Please add a new column that displays the temperature in Celsius.
3. Calculate the mean temperature in each month.
4. Remove all the data corresponding to Month = 5, group the data by month, and then find the mean of the temperature each month.

References

Chapter 13 of the book "R programming for data science".

Statistical Computing

Lecture 3: Control structures and functions

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Control structures

Commonly used control structures

- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop (must `break` out of it to stop)
- `break`: break the execution of a loop
- `next`: skip an interation of a loop

`if-else`

The `if-else` combination is probably the most commonly used control structure in R (or perhaps any language). For starters, you can just use the `if` statement.

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

`if-else`

If you have an action you want to execute when the condition is false, then you need an `else` clause.

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

You can have a series of tests by following the initial `if` with any number of `else ifs`.

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

`if-else` Example

```r
## Generate a uniform random number
x <- runif(1, 0, 10)
if (x > 3) {
    y <- 10
} else {
    y <- 0
}
```

Or you can write:

```r
y <- if (x > 3) {
    10
} else {
    0
}
```

`for` Loops

For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```r
for (i in 1:10) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

`for` Loops Example

The following three loops all have the same behavior.

```r
x <- c("a", "b", "c", "d")
for (i in 1:4) {
    ## Print out each element of 'x'
    print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

The `seq_along()` function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object `x`).

```
## Generate a sequence based on length of 'x'
for (i in seq_along(x)) {
    print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

It is not necessary to use an index-type variable.

```
for (letter in x) {
    print(letter)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

Nested `for` loops

`for` loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists).

`while` Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
count <- 0
while (count < 10) {
    print(count)
    count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

While loops can potentially result in infinite loops if not written properly. Use with care!

`repeat` Loops

`repeat` initiates an infinite loop right from the start. The only way to exit a `repeat` loop is to call `break`.

```
x0 <- 1
tol <- 1e-08
repeat {
    x1 <- computeEstimate()

    if (abs(x1 - x0) < tol) {
        ## Close enough?
        break
    } else {
        x0 <- x1
    }
}
```

`next`, `break`

`next` is used to skip an iteration of a loop.

```
for (i in 1:100) {
    if (i <= 20) {
        ## Skip the first 20 iterations
        next
    }
    ## Do something here
}
```

`break` is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for (i in 1:100) {
    print(i)
    if (i > 20) {
        ## Stop loop after 20 iterations
        break
    }
}
```

Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if (you believe) they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the "apply" functions are more useful.

Functions

Functions

- A transition from a mere "user" to a developer!
- Often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions.
- Often written when code must be shared with others or the public.

Your First Function

Functions are defined using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```r
f <- function() {
    cat("Hello, world!\n")
}
f()
```

```
## Hello, world!
```

The last aspect of a basic function is the *function arguments*.

```r
f <- function(num) {
    for (i in seq_len(num)) {
        cat("Hello, world!\n")
    }
}
f(3)
```

```
## Hello, world!
## Hello, world!
## Hello, world!
```

When to Write a Function?

If you find yourself doing a lot of cutting and pasting, that's usually a good sign that you might need to write a function.

This next function returns the total number of characters printed to the console.

```r
f <- function(num) {
    hello <- "Hello, world!\n"
    for (i in seq_len(num)) {
        cat(hello)
    }
    chars <- nchar(hello) * num
    chars
}
meaningoflife <- f(3)
```

```
## Hello, world!
## Hello, world!
## Hello, world!
```

```r
print(meaningoflife)
```

```
## [1] 42
```

Default Values

Try this:

```r
f()
```

We can modify this behavior by setting a *default value* for the argument `num`.

```r
f <- function(num = 1) {
    hello <- "Hello, world!\n"
    for (i in seq_len(num)) {
        cat(hello)
    }
    chars <- nchar(hello) * num
    chars
}
f()  ## Use default value for 'num'
```

```
## Hello, world!
```

```
## [1] 14
```

```r
f(2)  ## Use user-specified value
```

```
## Hello, world!
## Hello, world!
```

```
## [1] 28
```

At this point, we have written a function that

- has one *formal argument* named `num` with a *default value* of 1. The *formal arguments* are the arguments included in the function definition. The `formals()` function returns a list of all the formal arguments of a function
- prints the message "Hello, world!" to the console a number of times indicated by the argument `num`
- *returns* the number of characters printed to the console

Argument Matching

R assigns the first value to the first argument, the second value to second argument, etc. So in the following call to `rnorm()`

```r
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

```r
mydata <- rnorm(100, 2, 1)  ## Generate some data
```

100 is assigned to the `n` argument, 2 is assigned to the `mean` argument, and 1 is assigned to the `sd` argument, all by positional matching.

```r
## Positional match first argument, default for 'na.rm'
sd(mydata)
```

```
## [1] 1.025863
```

```
## Specify 'x' argument by name, default for 'na.rm'
sd(x = mydata)
```

```
## [1] 1.025863
```

```
## Specify both arguments by name
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.025863
```

When specifying the function arguments by name, it doesn't matter in what order you specify them.

```
## Specify both arguments by name
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.025863
```

You can mix positional matching with matching by name.

```
sd(na.rm = FALSE, mydata)
```

```
## [1] 1.025863
```

Here, the `mydata` object is assigned to the `x` argument, because it's the only argument not yet specified.

Below is the argument list for the `lm()` function, which fits linear models to a dataset.

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

The ... Argument

- There is a special argument in R known as the `...` argument, which indicate a variable number of arguments that are usually passed on to other functions.
- The `...` argument is often used when extending another function and you don't want to copy the entire argument list of the original function
- For example, a custom mean function may want to make use of the default `mean()` function along with its entire argument list. The function below changes the default for the `na.rm` argument to the value `na.rm = "TRUE"` (the original default was `na.rm = "FALSE"`).

```
mymean <- function(x, na.rm = TRUE, ...) {
        mean(x, na.rm = na.rm, ...)         ## Pass '...' to 'mean?rnorm' function
```

```r
}
x <- c(1, 2, NA)
mean(x)
mymean(x)
```

Summary

- Functions can be defined using the `function()` directive and are assigned to R objects just like any other R object
- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special `...` argument in a function definition.

Looping Functions

Looping on the Command Line

Writing `for` and `while` loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- `lapply()`: Loop over a list and evaluate a function on each element
- `sapply()`: Same as `lapply` but try to simplify the result
- `apply()`: Apply a function over the margins of an array
- `tapply()`: Apply a function over subsets of a vector
- `mapply()`: Multivariate version of `lapply`

`lapply()`

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the `l` is for "list").

This function takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list()`.

`lapply()` Example 1

Here's an example of applying the `mean()` function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```r
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.4463099
```

Notice that here we are passing the `mean()` function as an argument to the `lapply()` function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses `()` like you do when you are *calling* a function.

`lapply()` Example 2

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100,
    5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.1513343
##
## $c
## [1] 1.099387
##
## $d
## [1] 5.044237
```

`lapply()` Example 3

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.6090059
##
## [[2]]
## [1] 0.6426180 0.2720511
##
## [[3]]
## [1] 0.2748745 0.1650714 0.3495338
##
## [[4]]
## [1] 0.06996351 0.81690008 0.39358289 0.54555616
```

Now how about other arguments?

Here, the `min = 0` and `max = 10` arguments are passed down to `runif()` every time it gets called.

```
x <- 1:4
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 0.07901924
##
## [[2]]
## [1] 8.124458 4.479709
##
## [[3]]
## [1] 5.660471 4.711902 3.588175
##
## [[4]]
## [1] 9.350531 4.850635 8.183975 7.664252
```

`lapply()` Example 4

Here I am creating a list that contains two matrices.

```r
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

- How do you extract the first column of each matrix in the list?
- Now you need an anonymous function for extracting the first column of each matrix.

```r
lapply(x, function(elt) {
    elt[, 1]
})
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

I can also define the function separately.

```r
f <- function(elt) {
    elt[, 1]
}
lapply(x, f)
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

`sapply()`

The `sapply()` function behaves similarly to `lapply()`; the only real difference is in the return value. `sapply()` will try to simplify the result of `lapply()` if possible. Essentially, `sapply()` calls `lapply()` on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

Here's the result of calling `lapply()`.

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100,
    5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] -0.02610719
##
## $c
## [1] 0.9502227
##
## $d
## [1] 5.167316
```

Here's the result of calling `sapply()` on the same list.

```
sapply(x, mean)
```

```
##           a           b           c           d
##  2.50000000 -0.02610719  0.95022269  5.16731559
```

`split()`

The `split()` function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to `split()` are

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

where

- `x` is a vector (or list) or data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

The combination of `split()` and a function like `lapply()` or `sapply()` is a common paradigm in R.

split() Example

Here we simulate some data and split it according to a factor variable. Note that we use the gl() function to "generate levels" in a factor variable.

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
split(x, f)
```

```
## $`1`
##  [1] -0.5386728  0.9295600 -0.4682451 -0.4508130 -0.1778677 -0.5205303
##  [7] -1.5224860 -0.4957441 -1.6650648  0.5737440
##
## $`2`
##  [1] 0.2565500 0.9344672 0.3933274 0.5301347 0.2468453 0.9889854 0.5136659
##  [8] 0.6495695 0.8827517 0.5351771
##
## $`3`
##  [1]  1.38026130  2.38310883  1.53170210  2.61821018  1.17681609
##  [6]  1.37004551  1.14637725 -0.83498703  0.21195133 -0.05818388
```

A common idiom is split followed by an lapply.

```
lapply(split(x, f), mean)
```

```
## $`1`
## [1] -0.433612
##
## $`2`
## [1] 0.5931474
##
## $`3`
## [1] 1.09253
```

Splitting a Data Frame

```
library(datasets)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

We can split the airquality data frame by the Month variable so that we have separate sub-data frames for each month.

```
s <- split(airquality, airquality$Month)
str(s)
```

```
## List of 5
##  $ 5:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
##   ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
##   ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##   ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
##   ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 6:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
##   ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
##   ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
##   ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
##   ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 7:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
##   ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
##   ..$ Wind   : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
##   ..$ Temp   : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
##   ..$ Month  : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 8:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
##   ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
##   ..$ Wind   : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
##   ..$ Temp   : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
##   ..$ Month  : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 9:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
##   ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
##   ..$ Wind   : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
##   ..$ Temp   : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
##   ..$ Month  : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

Then we can take the column means for `Ozone`, `Solar.R`, and `Wind` for each sub-data frame.

```
lapply(s, function(x) {
    colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE)
})
```

```
## $`5`
##     Ozone   Solar.R      Wind
##  23.61538 181.29630  11.62258
##
## $`6`
##     Ozone   Solar.R      Wind
##  29.44444 190.16667  10.26667
##
## $`7`
##      Ozone    Solar.R       Wind
##  59.115385 216.483871   8.941935
##
## $`8`
##      Ozone    Solar.R       Wind
##  59.961538 171.857143   8.793548
```

```
##
## $`9`
##     Ozone   Solar.R      Wind
##  31.44828 167.43333  10.18000
```

Using `sapply()` might be better here for a more readable output.

```r
sapply(s, function(x) {
    colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE)
})
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

`tapply()`

`tapply()` is used to apply a function over subsets of a vector. It can be thought of as a combination of `split()` and `sapply()` for vectors only.

```r
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

The arguments to `tapply()` are as follows:

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- ... contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

`tapply()` Example

Given a vector of numbers, one simple operation is to take group means.

```r
## Simulate some data
x <- c(rnorm(10), runif(10), rnorm(10, 1))
## Define some groups with a factor variable
f <- gl(3, 10)
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```r
tapply(x, f, mean)
```

```
##         1         2         3
## 0.1480750 0.5217016 1.0749164
```

It is equivalent to:

```r
sapply(split(x, f), mean)
```

```
##         1         2         3
## 0.1480750 0.5217016 1.0749164
```

`apply()`

- Used to a evaluate a function over the margins of an array.
- Often used to apply a function to the rows or columns of a matrix.
- Using `apply()` is not really faster than writing a loop, but it works in one line and is highly compact.

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

The arguments to `apply()` are

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be "retained".
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

`apply()` Example

Here I create a 20 by 10 matrix of Normal random numbers. I then compute the mean of each column.

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)  ## Take the mean of each column
```

```
##  [1]  0.162590154  0.264977385 -0.222791353  0.219542710  0.127501619
##  [6] -0.002607522  0.010729913  0.172286913 -0.430984706 -0.086041934
```

I can also compute the sum of each row.

```
apply(x, 1, sum)  ## Take the mean of each row
```

```
##  [1] -1.2495094  0.4610130  0.8709000 -5.7911196  5.9498309 -5.4315689
##  [7] -4.3392107  4.8994311 -1.4722936 -5.6203763 -0.8972293  2.5279202
## [13] -1.2607678  8.3443647 -3.0409513  0.3704913  1.3521787  2.0395559
## [19]  1.3034202  5.2879845
```

- Note that in both calls to `apply()`, the return value was a vector of numbers.
- The `MARGIN` argument essentially indicates to `apply()` which dimension of the array you want to preserve or retain. So when taking the mean of each column, I specify

```
apply(x, 2, mean)
```

Col/Row Sums and Means

For the special case of column/row sums and column/row means of matrices, we have some useful shortcuts.

- `rowSums` = `apply(x, 1, sum)`
- `rowMeans` = `apply(x, 1, mean)`
- `colSums` = `apply(x, 2, sum)`
- `colMeans` = `apply(x, 2, mean)`

The shortcut functions are heavily optimized and hence are *much* faster, but you probably won't notice unless you're using a large matrix. Another nice aspect of these functions is that they are a bit more descriptive. It's arguably more clear to write `colMeans(x)` in your code than `apply(x, 2, mean)`.

Other Ways to Apply

You can do more than take sums and means with the `apply()` function. For example, you can compute quantiles of the rows of a matrix using the `quantile()` function.

```
x <- matrix(rnorm(200), 20, 10)
## Get row quantiles
apply(x, 1, quantile, probs = c(0.25, 0.75))
```

```
##           [,1]       [,2]       [,3]       [,4]       [,5]      [,6]
## 25% -0.8828056 -0.2131614 -0.1716636 -0.6753241 -0.5797748 0.3232881
## 75%  0.5387326  1.1380349  0.4517773  0.4433824  0.6252163 1.1695355
##           [,7]       [,8]       [,9]      [,10]       [,11]      [,12]
## 25% -0.7031755 -0.4029417 -0.7935699 -0.4532591 -0.05619261 -0.6926032
## 75%  0.7559389  1.1000217  0.7200173  0.6407631  0.72531964  0.6539091
##          [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
## 25% -0.6450824 -0.8604518 -0.8428936 -1.0681552 -0.4191486 -0.8107607
## 75%  0.8404527  0.5207244  0.4128027  0.1198969  0.6404508  0.2626132
##          [,19]      [,20]
## 25% 0.04098455 0.03793319
## 75% 0.56164971 1.07131897
```

Notice that I had to pass the `probs = c(0.25, 0.75)` argument to `quantile()` via the `...` argument to `apply()`.

`mapply()`

The `mapply()` function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that `lapply()` and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what `mapply()` is for.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The arguments to `mapply()` are

- `FUN` is a function to apply
- `...` contains R objects to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

The `mapply()` function has a different argument order from `lapply()` because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the `...` argument because we can apply over an arbitrary number of R objects.

`mapply()` Example

For example, the following is tedious to type

`list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))`

With `mapply()`, instead we can do

```
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

This passes the sequence `1:4` to the first argument of `rep()` and the sequence `4:1` to the second argument.

Here's another example for simulating randon Normal variables.

```
noise <- function(n, mean, sd) {
    rnorm(n, mean, sd)
}
## Simulate 5 randon numbers
noise(5, 1, 2)
```

```
## [1]  3.4367350  0.9789327  3.2000309  1.2762248 -3.5700115
```

Here we can use `mapply()` to pass the sequence `1:5` separately to the `noise()` function so that we can get 5 sets of random numbers, each with a different length and mean.

```
x <- mapply(noise, 1000, 1:5, 1:5)
apply(x, 2, mean)
```

```
## [1] 1.024402 2.077510 2.754549 4.063622 5.032008
```

The above call to `mapply()` is the same as

```
invisible(list(noise(1000, 1, 1), noise(1000, 2, 2), noise(1000,
    3, 3), noise(1000, 4, 4), noise(1000, 5, 5)))
```

Vectorizing a Function

- The `mapply()` function can be use to automatically "vectorize" a function.
- It can be used to take a function that typically only takes single arguments and create a new function that can take vector arguments.

Here's an example of a function that computes the sum of squares $\sum_{i=1}^{n}(x_i - \mu)^2/\sigma^2$.

```
sumsq <- function(mu, sigma, x) {
    sum(((x - mu)/sigma)^2)
}
```

This function takes a mean `mu`, a standard deviation `sigma`, and some data in a vector `x`.

In many statistical applications, we want to minimize the sum of squares to find the optimal `mu` and `sigma`. Before we do that, we may want to evaluate or plot the function for many different values of `mu` or `sigma`. However, passing a vector of `mu`s or `sigma`s won't work with this function because it's not vectorized.

```
x <- rnorm(100)  ## Generate some data
sumsq(1:10, 1:10, x)  ## This is not what we want
```

```
## [1] 106.4872
```

Note that the call to `sumsq()` only produced one value instead of 10 values.

However, we can do what we want to do by using `mapply()`.

```
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

```
##  [1] 177.02812 114.38448 104.22752 101.15984  99.96269  99.43270  99.18545
##  [8]  99.07182  99.02601  99.01616
```

There's even a function in R called `Vectorize()` that automatically can create a vectorized version of your function. So we could create a `vsumsq()` function that is fully vectorized as follows.

```
vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
vsumsq(1:10, 1:10, x)
```

```
##  [1] 177.02812 114.38448 104.22752 101.15984  99.96269  99.43270  99.18545
##  [8]  99.07182  99.02601  99.01616
```

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The `split()` function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.

Lab Session 2

In this lab, you will use the temperature data in four cities: Melbourne, Sydney, Brisbane and Cairns. You can download them from https://yanfei.site/docs/sc/data/temp.zip.

1. Please make a function `load.file()` to read a .csv file and transform the first column (a character representing date and time) using `as.POSIXlt` into R time format.
2. Then apply `load.file()` to each filename using `lapply()`.
3. How many rows of data are there for each city?
4. What is the hottest temperature recorded by city?
5. Estimate the autocorrelation function for each city.

References

Chapters 14, 15 and 18 of the book "R programming for data science".

Statistical Computing

Lecture 4: Dealing with text data

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Primary R Functions

The primary R functions for dealing with text data are

- `grep()`, `grepl()`: These functions search for matches of a regular pattern in a character vector. `grep()` returns the indices into the character vector that contain a match or the specific strings that happen to have the match. `grepl()` returns a `TRUE`/`FALSE` vector indicating which elements of the character vector contain a match.
- `regexpr()`, `gregexpr()`: Search a character vector for pattern matches and return the indices of the string where the match begins and the length of the match.
- `sub()`, `gsub()`: Search a character vector for pattern matches and replace that match with another string.
- `substr()`: Extract substrings in a character vector.
- `regexec()`: This function searches a character vector for a pattern, much like `regexpr()`, but it will additionally return the locations of any parenthesized sub-expressions. Probably easier to explain through demonstration.

Text data

We will use a running example using data from homicides in Baltimore City. You can get the file from https://yanfei.site/docs/sc/data/homicides.txt. Original data is from https://homicides.news.baltimoresun.com.

```r
homicides <- readLines("../data/homicides.txt")
## Total number of events recorded
length(homicides)
```

```
## [1] 1571
```

```r
homicides[1]
```

```
## [1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon Nelson</dt><dd class=\"address\
```

```r
homicides[1000]
```

```
## [1] "39.33626300000, -76.55553990000, icon_homicide_shooting, 'p1200', '<dl><dt><a href=\"http://ess
```

We have the latitude and longitude of where the victim was found; then there's the street address; the age, race, and gender of the victim; the date on which the victim was found; in which hospital the victim ultimately died; the cause of death.

`grep()`

Suppose we wanted to identify the records for all the victims of shootings (as opposed to other causes)? How could we do that?

Here I use `grep()` to match the literal `iconHomicideShooting` into the character vector of homicides.

```
g <- grep("iconHomicideShooting", homicides)
length(g)
```

```
## [1] 228
```

Using this approach I get 228 shooting deaths. However, I notice that for some of the entries, the indicator for the homicide "flag" is noted as `icon_homicide_shooting`. It's not uncommon over time for web site maintainers to change the names of files or update files. What happens if we now `grep()` on both icon names using the `|` operator?

```
g <- grep("iconHomicideShooting|icon_homicide_shooting", homicides)
length(g)
```

```
## [1] 1263
```

`grep()`

Another possible way to do this is to `grep()` on the cause of death field, which seems to have the format `Cause: shooting`. We can `grep()` on this literally and get

```
g <- grep("Cause: shooting", homicides)
length(g)
```

```
## [1] 228
```

Notice that we seem to be undercounting again. This is because for some of the entries, the word "shooting" uses a captial "S" while other entries use a lower case "s". We can handle this variation by using a character class in our regular expression.

```
g <- grep("Cause: [Ss]hooting", homicides)
length(g)
```

```
## [1] 1263
```

`grepl()`

The function `grepl()` works much like `grep()` except that it differs in its return value. `grepl()` returns a logical vector indicating which element of a character vector contains the match. For example, suppose we want to know which states in the United States begin with word "New".

```
g <- grepl("^New", state.name)
g
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [28] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [46] FALSE FALSE FALSE FALSE FALSE
```

```
state.name[g]
```

```
## [1] "New Hampshire" "New Jersey"    "New Mexico"
## [4] "New York"
```

Here, we can see that `grepl()` returns a logical vector that can be used to subset the original `state.name` vector.

`regexpr()`

- Both the `grep()` and the `grepl()` functions have some limitations - they don't tell you exactly where the match occurs or what the match is for a more complicated regular expression.
- The `regexpr()` function gives you
  - index into each string where the match begins
  - length of the match for that string.
- `regexpr()` only gives you the *first* match of the string (reading left to right). `gregexpr()` will give you *all* of the matches in a given string if there are is more than one match.

`regexpr()` Example

In our Baltimore City homicides dataset, we might be interested in finding the date on which each victim was found. Taking a look at the dataset

```
homicides[1]
```

```
## [1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon Nelson</dt><dd class=\"address
```

it seems that we might be able to just `grep` on the word "Found". However, the word "found" may be found elsewhere in the entry, such as in this entry, where the word "found" appears in the narrative text at the end.

```
homicides[954]
```

```
## [1] "39.30677400000, -76.59891100000, icon_homicide_shooting, 'p816', '<dl><dt><a href=\"http://esse
```

But we can see that the date is typically preceded by "Found on" and is surrounded by `<dd></dd>` tags, so let's use the pattern `<dd>[F|f]ound(.*)</dd>` and see what it brings up.

```
regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
```

```
##  [1] 177 178 188 189 178 182 178 187 182 183
## attr(,"match.length")
##  [1] 93 86 89 90 89 84 85 84 88 84
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

We can use the `substr()` function to extract the first match in the first string.

```
substr(homicides[1], 177, 177 + 93 - 1)
```

```
## [1] "<dd>Found on January 1, 2007</dd><dd>Victim died at Shock Trauma</dd><dd>Cause: shooting</dd>"
```

Picked up too much information? We need to use the `?` metacharacter to make the regular expression "lazy" so that it stops at the *first* `</dd>` tag.

```
regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:10])
```

```
##  [1] 177 178 188 189 178 182 178 187 182 183
## attr(,"match.length")
##  [1] 33 33 33 33 33 33 33 33 33 33
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Now when we look at the substrings indicated by the `regexpr()` output, we get

```r
substr(homicides[1], 177, 177 + 33 - 1)
```

```
## [1] "<dd>Found on January 1, 2007</dd>"
```

Instead of using `substr()`, `regmatches()` is more handy.

```r
r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
regmatches(homicides[1:5], r)
```

```
## [1] "<dd>Found on January 1, 2007</dd>"
## [2] "<dd>Found on January 2, 2007</dd>"
## [3] "<dd>Found on January 2, 2007</dd>"
## [4] "<dd>Found on January 3, 2007</dd>"
## [5] "<dd>Found on January 5, 2007</dd>"
```

`sub()` and `gsub()`

Sometimes we need to clean things up or modify strings by matching a pattern and replacing it with something else. For example, how can we extract the date from this string?

```r
x <- substr(homicides[1], 177, 177 + 33 - 1)
x
```

```
## [1] "<dd>Found on January 1, 2007</dd>"
```

We want to strip out the stuff surrounding the "January 1, 2007" portion. We can do that by matching on the text that comes before and after it using the `|` operator and then replacing it with the empty string.

```r
sub("<dd>[F|f]ound on |</dd>", "", x)
```

```
## [1] "January 1, 2007</dd>"
```

Notice that the `sub()` function found the first match (at the beginning of the string) and replaced it and then stopped. However, there was another match at the end of the string that we also wanted to replace. To get both matches, we need the `gsub()` function.

```r
gsub("<dd>[F|f]ound on |</dd>", "", x)
```

```
## [1] "January 1, 2007"
```

The `sub()` and `gsub()` functions can take vector arguments so we don't have to process each string one by one.

```r
r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
m <- regmatches(homicides[1:5], r)
m
```

```
## [1] "<dd>Found on January 1, 2007</dd>"
## [2] "<dd>Found on January 2, 2007</dd>"
## [3] "<dd>Found on January 2, 2007</dd>"
## [4] "<dd>Found on January 3, 2007</dd>"
## [5] "<dd>Found on January 5, 2007</dd>"
```

```
d <- gsub("<dd>[F|f]ound on |</dd>", "", m)
## Nice and clean
d
```

```
## [1] "January 1, 2007" "January 2, 2007" "January 2, 2007"
## [4] "January 3, 2007" "January 5, 2007"
```

Finally, it may be useful to convert these strings to the `Date` class so that we can do some date-related computations.

```
as.Date(d, "%B %d, %Y")
```

```
## [1] "2007-01-01" "2007-01-02" "2007-01-02" "2007-01-03"
## [5] "2007-01-05"
```

`regexec()`

The `regexec()` function works like `regexpr()` except it gives you the indices for parenthesized sub-expressions. For example, take a look at the following expression.

```
regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])
```

```
## [[1]]
## [1] 177 190
## attr(,"match.length")
## [1] 33 15
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Here's the overall expression match.

```
substr(homicides[1], 177, 177 + 33 - 1)
```

```
## [1] "<dd>Found on January 1, 2007</dd>"
```

And here's the parenthesized sub-expression.

```
substr(homicides[1], 190, 190 + 15 - 1)
```

```
## [1] "January 1, 2007"
```

All this can be done much more easily with the `regmatches()` function.

```
r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1:2])
regmatches(homicides[1:2], r)
```

```
## [[1]]
## [1] "<dd>Found on January 1, 2007</dd>"
## [2] "January 1, 2007"
##
## [[2]]
## [1] "<dd>Found on January 2, 2007</dd>"
## [2] "January 2, 2007"
```

`regexec()`

As an example, we can make a plot of monthly homicide counts. First we need a regular expression to capture the dates.

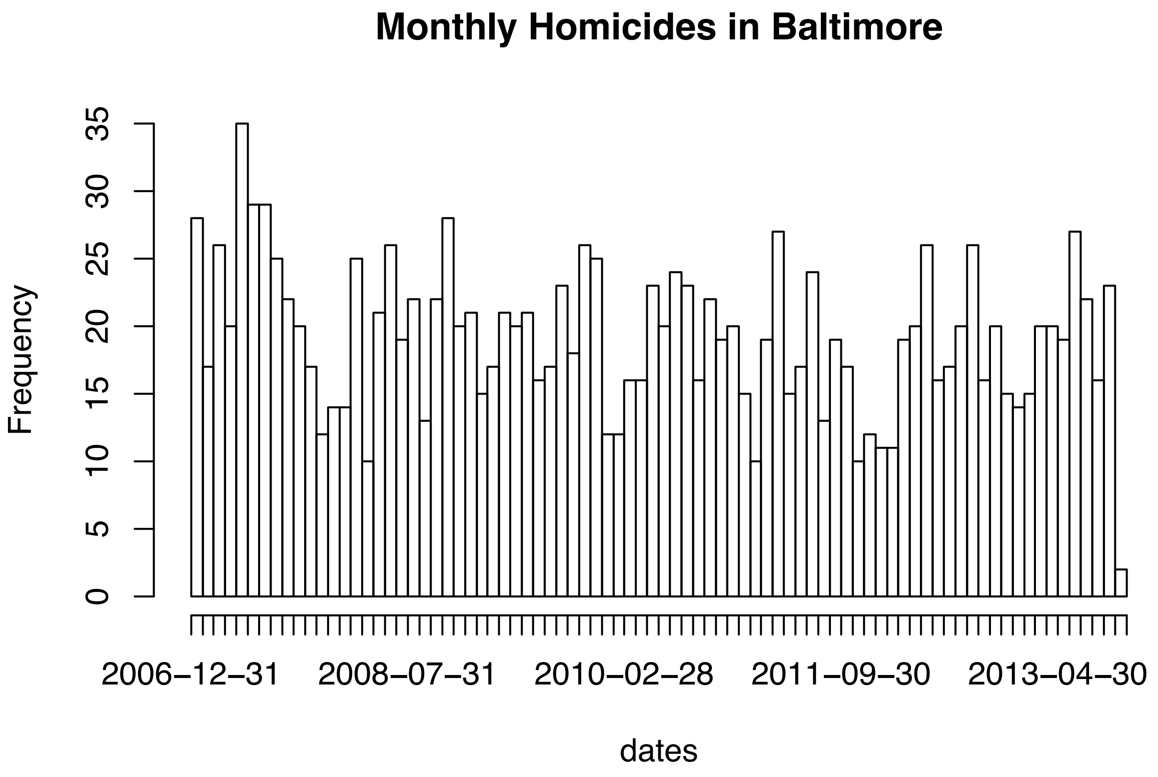
```r
r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides)
m <- regmatches(homicides, r)
```

Then we can loop through the list returned by `regmatches()` and extract the second element of each (the parenthesized sub-expression).

```r
dates <- sapply(m, function(x) x[2])
```

Finally, we can convert the date strings into the `Date` class and make a histogram of the counts.

```r
invisible(dates <- as.Date(dates, "%B %d, %Y"))
hist(dates, "month", freq = TRUE, main = "Monthly Homicides in Baltimore")
```



We can see from the picture that homicides do not occur uniformly throughout the year and appear to have some seasonality to them.

Summary

The primary R functions for dealing with regular expressions are

- `grep()`, `grepl()`: Search for matches of a regular expression/pattern in a character vector
- `regexpr()`, `gregexpr(): Search a character vector for regular expression matches and return the indices where the match begins; useful in conjunction with`regmatches()‘
- `sub()`, `gsub()`: Search a character vector for regular expression matches and replace that match with another string

- `regexec()`: Gives you indices of parethensized sub-expressions.

References

Chapter 19 of the book “R programming for data science”.

Statistical Computing

Lecture 5: Debugging and profiling R code

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Debugging

Something's Wrong!

R has a number of ways to indicate to you that something's not right.

- `message`: A generic notification/diagnostic message produced by the `message()` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues. Warnings are generated by the `warning()` function
- `error`: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the `stop()` function.
- `condition`: A generic concept for indicating that something unexpected has occurred; programmers can create their own custom conditions if they want.

Try `log(-1)`.

Example

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

Now try `printmessage(1)` and `printmessage(NA)`. What happened? How to fix this problem?

Solution

```
printmessage2 <- function(x) {
    if (is.na(x))
        print("x is a missing value!") else if (x > 0)
        print("x is greater than zero") else print("x is less than or equal to zero")
    invisible(x)
}
```

Now try `printmessage2(NA)`.

Example

Now try the following and see what happened.

```r
x <- log(c(-1, 2))
printmessage2(x)
```

Solution

1. allowing vector arguments.
2. vectorizing the `printmessage2()` function to allow it to take vector arguments.

For the first solution, we check the length of the input.

```r
printmessage3 <- function(x) {
    if (length(x) > 1L)
        stop("'x' has length > 1")
    if (is.na(x))
        print("x is a missing value!") else if (x > 0)
        print("x is greater than zero") else print("x is less than or equal to zero")
    invisible(x)
}
```

Now try `printmessage3(1:2)`.

For the second solution, vectorizing the function can be accomplished easily with the `Vectorize()` function.

```r
printmessage4 <- Vectorize(printmessage2)
out <- printmessage4(c(-1, 2))
```

Figuring Out What's Wrong

The primary task of debugging any R code is correctly diagnosing what the problem is. Some basic questions you need to ask are

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Debugging Tools in R

R provides a number of tools to help you with debugging your code.

- `traceback()`: prints out the function call stack after an error occurs; does nothing if there's no error
- `debug()`: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- `browser()`: suspends the execution of a function wherever it is called and puts the function in debug mode
- `trace()`: allows you to insert debugging code into a function a specific places
- `recover()`: allows you to modify the error behavior so that you can browse the function call stack

These functions are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting `print()` or `cat()` statements in the function.

Using `traceback()`

- The `traceback()` function prints out the *function call stack* after an error has occurred.
- For example, you may have a function `a()` which subsequently calls function `b()` which calls `c()` and then `d()`. If an error occurs, it may not be immediately clear in which function the error occurred. The `traceback()` function shows you how many levels deep you were when the error occurred.

```
mean(x)
traceback()
```

Example

```
f <- function(x) {
  r <- x - g(x)
  r
}
g <- function(y) {
  r <- y * h(y)
  r
}
h <- function(z) {
  r <- log(z)
  if (r < 10)
    r^2
  else r^3
}
```

- Try `f(-1)`.
- Try `traceback()`.
- Looking at the traceback is useful for figuring out roughly where an error occurred but it's not useful for more detailed debugging. For that you might turn to the `debug()` function.

Using `debug()`

- The `debug()` function initiates an interactive debugger (also known as the "browser" in R) for a function.
- With the debugger, you can step through an R function one expression at a time to pinpoint exactly where an error occurs.
- The `debug()` function takes a function as its first argument. For debugging the `f()` function, try `debug(f)`.
- Now, every time you call the `f()` function it will launch the interactive debugger. To turn this behavior off you need to call the `undebug()` function.

Commands in `debug()`

The debugger calls the browser at the very top level of the function body. From there you can step through each expression in the body. There are a few special commands you can call in the browser:

- `n` executes the current expression and moves to the next expression
- `c` continues execution of the function and does not stop until either an error or the function exits
- `Q` quits the browser

Example 1

Try `f(-1)`.

- While you are in the browser you can execute any other R function that might be available to you in a regular session.
- You can use `ls()` to see what is in your current environment (the function environment) and `print()` to print out the values of R objects in the function environment.
- You can turn off interactive debugging with the `undebug()` function.

Example 2

```r
SS <- function(mu, x) {
  d <- x - mu
  d2 <- d ^ 2
  ss <- sum(d2)
  ss
}
debug(SS)
SS(1, rnorm(100))
```

Using `recover()`

- The `recover()` function can be used to modify the error behavior of R when an error occurs. Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.
- With `recover()` you can tell R that when an error occurs, it should halt execution at the exact point at which the error occurred. That can give you the opportunity to poke around in the environment in which the error occurred. This can be useful to see if there are any R objects or data that have been corrupted or mistakenly modified.

```r
options(error = recover)    ## Change default R error behavior
f(-1)
```

- The `recover()` function will first print out the function call stack when an error occurrs. Then, you can choose to jump around the call stack and investigate the problem. When you choose a frame number, you will be put in the browser (just like the interactive debugger triggered with `debug()`) and will have the ability to poke around.

Summary

- There are three main indications of a problem/condition: `message`, `warning`, `error`; only an `error` is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions

- Debugging tools are not a substitute for thinking!

Profiling R

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

—*Donald Knuth*

R Profiler

- R comes with a profiler to help you optimize your code and improve its performance.
- In generall, it's usually a bad idea to focus on optimizing your code at the very beginning of development.
- Rather, in the beginning it's better to focus on translating your ideas into code and writing code that's coherent and readable.
- The problem is that heavily optimized code tends to be obscure and difficult to read, making it harder to debug and revise. Better to get all the bugs out first, then focus on optimizing.

What to optimize

- Optimizing the parts of your code that are running *slowly*
- How do we know what parts those are? ⇒ This is what the profiler is for.
- Profiling is a systematic way to examine how much time is spent in different parts of a program.
- Often you might write some code that runs fine once. But then later, you might put that same code in a big loop that runs 1,000 times. Now the original code that took 1 second to run is taking 1,000 seconds to run! Getting that little piece of original code to run faster will help the entire loop.

Basic principles

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don't guess.
- If you're going to be scientist, you need to apply the same principles here!

Using `system.time()`

The `system.time()` function computes the time (in seconds) needed to execute an expression. The function returns:

- *user time*: time charged to the CPU(s) for this expression
- *elapsed time*: "wall clock" time, the amount of time that passes for *you* as you're sitting there

The elapsed time may be *greater than* the user time if the CPU spends a lot of time waiting around.

```r
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
```

```
##    user  system elapsed
##   0.042   0.006   4.513
```

The elapsed time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them).

```r
library(plyr)
library(doMC)
```

```
## Loading required package: foreach

## Loading required package: iterators

## Loading required package: parallel
```

```r
registerDoMC(cores = detectCores())
system.time(aaply(1:10000, 1, function(x) rnorm(1, mean = x),
    .parallel = TRUE))
```

```
##    user  system elapsed
##   3.715   0.766   2.465
```

Timing Longer Expressions

You can time longer expressions by wrapping them in curly braces within the call to `system.time()`.

```r
system.time({
    n <- 1000
    r <- numeric(n)
    for (i in 1:n) {
        x <- rnorm(n)
        r[i] <- mean(x)
    }
})
```

```
##    user  system elapsed
##   0.098   0.003   0.101
```

If your expression is getting pretty long (more than 2 or 3 lines), it might be better to either break it into smaller pieces or to use the profiler. The problem is that if the expression is too long, you won't be able to identify which part of the code is causing the bottleneck.

The R Profiler

- Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time.
- However, this approach assumes that you already know where the problem is and can call `system.time()` on it that piece of code.
- What if you don't know where to start?
- This is where the profiler comes in handy.

The R Profiler

- The `Rprof()` function starts the profiler in R.
- In conjunction with `Rprof()`, we will use the `summaryRprof()` function which summarizes the output from `Rprof()`.

- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function.
- By default it will write its output to a file called Rprof.out. You can specify the name of the output file if you don't want to use this default.
- Once you call the Rprof() function, everything that you do from then on will be measured by the profiler.
- The profiler can be turned off by passing NULL to Rprof().

Example

Now let's play around some data with the profiler running.

```
Rprof()
data(diamonds, package = "ggplot2")
plot(price ~ carat, data = diamonds)
m <- lm(price ~ carat, data = diamonds)
abline(m, col = "red")
Rprof(NULL)
```

Using summaryRprof()

The summaryRprof() function tabulates the R profiler output and calculates how much time is spent in which function. There are two methods for normalizing the data.

- "by.total" divides the time spend in each function by the total run time
- "by.self" does the same as "by.total" but first subtracts out time spent in functions above the current function in the call stack.

Now try summaryRprof() and interprete the results.

Interactive Visualizations for Profiling R Code

profvis is a tool for helping you to understand how R spends its time.

```
library(profvis)

profvis({
  data(diamonds, package = "ggplot2")
  plot(price ~ carat, data = diamonds)
  m <- lm(price ~ carat, data = diamonds)
  abline(m, col = "red")
})
```

Note that Rstudio includes integrated support for profiling with profvis.

Summary

- Rprof() runs the profiler for performance of analysis of R code
- summaryRprof() summarizes the output of Rprof() and gives percent of time spent in each function (with two types of normalization)

- Good to break your code into functions so that the profiler can give useful information about where time is being spent
- RStudio includes integrated support for profiling with `profvis`
- C or Fortran code is not profiled

Lab Session 4

In this lab, write your own code, enjoy the tools of debugging and profiling and write a short report of optimizing your code.

References

Chapters 20 and 21 of the book “R programming for data science”.

Statistical Computing

Lecture 6: Newton Method

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Optimisation in Business

Many problems in business require something to be minimized or maximized.

- Maximizing Revenue
- Minimizing Costs
- Minimizing Delivery Time
- Maximizing Financial Returns

Optimisation in Statistics

- Maximum Likelihood
- Least Squares
- Method of Moments
- Posterior Mode

Optimisation

- Suppose we want to find a minimum or maximum of a function $f(x)$.
- Sometimes $f(x)$ will be very complicated.
- Are there computer algorithms that can help?
- Yes!

Root Finding

Root Finding

- Consider the problem of finding the root of a function.
- For the function $f(x)$, the root is the point x^* such that $f(x^*) = 0$.
- An algorithm for solving this problem was proposed by Newton and Raphson nearly 500 years ago.

Newton-Raphson Method

- Newton-Raphson is an iterative method that begins with an initial guess of the root.
- The method uses the derivative of the function $f^{'}(x)$ as well as the original function $f(x)$.
- When successful, it converges (usually) rapidly, but may fail as any other root-finding algorithm.

Newton-Raphson Method

The method tries to solve an equation in the form of $f(x) = 0$ through the iterative procedure:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Please think about **Taylor expansion**.

Stopping Rule

- With each step the algorithm should get closer to the root.
- However, it can run for a long time without reaching the exact root.
- There must be a stopping rule otherwise the program could run forever.
- Let ϵ be an extremely small number e.g. 1×10^{-10} called the **tolerance level**.
- If $|f(x^*)| < \epsilon$ then the solution is close enough and there is a root at x^*.

Example

- Now find the root of $f(x) = x^2 - 5$ using Newton method by hand.
- Tell about its geometric interpretation.

Newton-Raphson Method

1. Select an initial guess x_0, and set $n = 0$.
2. Set $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.
3. Evaluate $|f(x_{n+1})|$.
    1. If $|f(x_{n+1})| < \epsilon$, then stop;
    2. Otherwise set $n = n + 1$ and go back to step 2.

Lab Session 5

1. Write R code to find the root of $x^2 = 5$.
2. Now use your Newton-Raphson code to find the root of $f(x) = \sqrt{|x|}$. Try initial value 0.25.
3. Now use your Newton-Raphson code to find the root of $xe^{-x^2} = 0.4(e^x + 1)^{-1} + 0.2$. Try initial values 0.5 and 0.6.

What did you learn from mistakes?

You can use `D()`

```
newton.raphson <- function(f, init.value, df = NULL, tol = 1e-5, maxiter = 1000) {
  if (is.null(df)) {
    df <- D(f, 'x')
  }
  niter <- 0
  diff <- tol + 1
  x <- init.value
  while (diff >= tol && niter <= maxiter) {
    niter <- niter + 1
    fx <- eval(f)
    dfx <- eval(df)
    if (dfx == 0) {
      warning("Slope is zero: no further improvement possible.")
      break
    }
    diff <- -fx/dfx
    x <- x + diff
    diff <- abs(diff)
  }
  if (niter > maxiter) {
    warning("Maximum number of iterations 'maxiter' was reached.")
  }
  return(list(root = x, f.root = fx, niter = niter, estim.prec = diff))
}
```

You can also use `deriv()`

```
newton.raphson <- function(ftn, x0, tol = 1e-9, max.iter = 100) {
  # Newton_Raphson algorithm for solving ftn(x)[1] == 0
  # we assume that ftn is a function of a single variable that returns
  # the function value and the first derivative as a vector of length 2
  #
  # x0 is the initial guess at the root
  # the algorithm terminates when the function value is within distance
  # tol of 0, or the number of iterations exceeds max.iter

  # initialise
  x <- x0
  fx <- ftn(x)
  iter <-  0

  # continue iterating until stopping conditions are met
  while ((abs(fx[1]) > tol) && (iter < max.iter)) {
    x <- x - fx[1]/fx[2]
    fx <- ftn(x)
    iter <-  iter + 1
    cat("At iteration", iter, "value of x is:", x, "\n")
  }

  # output depends on success of algorithm
  if (abs(fx[1]) > tol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(x)
  }
}

f <- expression(x^2 - 5)
df <- deriv(f, 'x', func = TRUE)
ftn <- function(x){
  dfx <- df(x)
  f <- dfx[1]
  gradf <- attr(dfx, 'gradient')[1,]
  return(c(f, gradf))
}

newton.raphson(ftn, 1)
```

Other Examples

Now try these functions:

1. $f(x) = x^3 - 2x^2 - 11x + 12$, try starting values 2.35287527 and 2.35284172.
2. $f(x) = 2x^3 + 3x^2 + 5$, try starting values 0.5 and 0.

Learn from Mistakes

- Newton-Raphson does not always converge.
- For some functions, using some certain starting values leads to a series that converges, while other starting values lead to a series that diverges.
- For other functions different starting values converge to different roots.
- Be careful when choosing the initial value.
- Newton-Raphson doesn't work if the first derivative is zero.

Conclusion

- Why did we spend so much time on finding roots of an equation?
- Isn't this topic meant to be about optimization?
- Can we change the algorithm slightly so that it works for optimization?

Optimisation

Finding a Maximum or Minimum

- Suppose we want to find an minimum or maximum of a function $f(x)$ (think about maximum likelihood estimation).
- Find the derivative $f^{'}(x)$ and find x^* such that $f^{'}(x^*) = 0$.
- This is the same as finding a root of the first derivative. We can use the Newton Raphson algorithm on the first derivative.

Newton Raphson algorithm for finding local maxima or minima

1. Select an initial guess x_0, and set $n = 0$.
2. Set $x_{n+1} = x_n - \frac{f^{'}(x_n)}{f^{''}(x_n)}$.
3. Evaluate $|f^{'}(x_{n+1})|$.
    1. If $|f^{'}(x_{n+1})| < \epsilon$, then stop;
    2. Otherwise set $n = n + 1$ and go back to step 2.

Different Stopping Rules

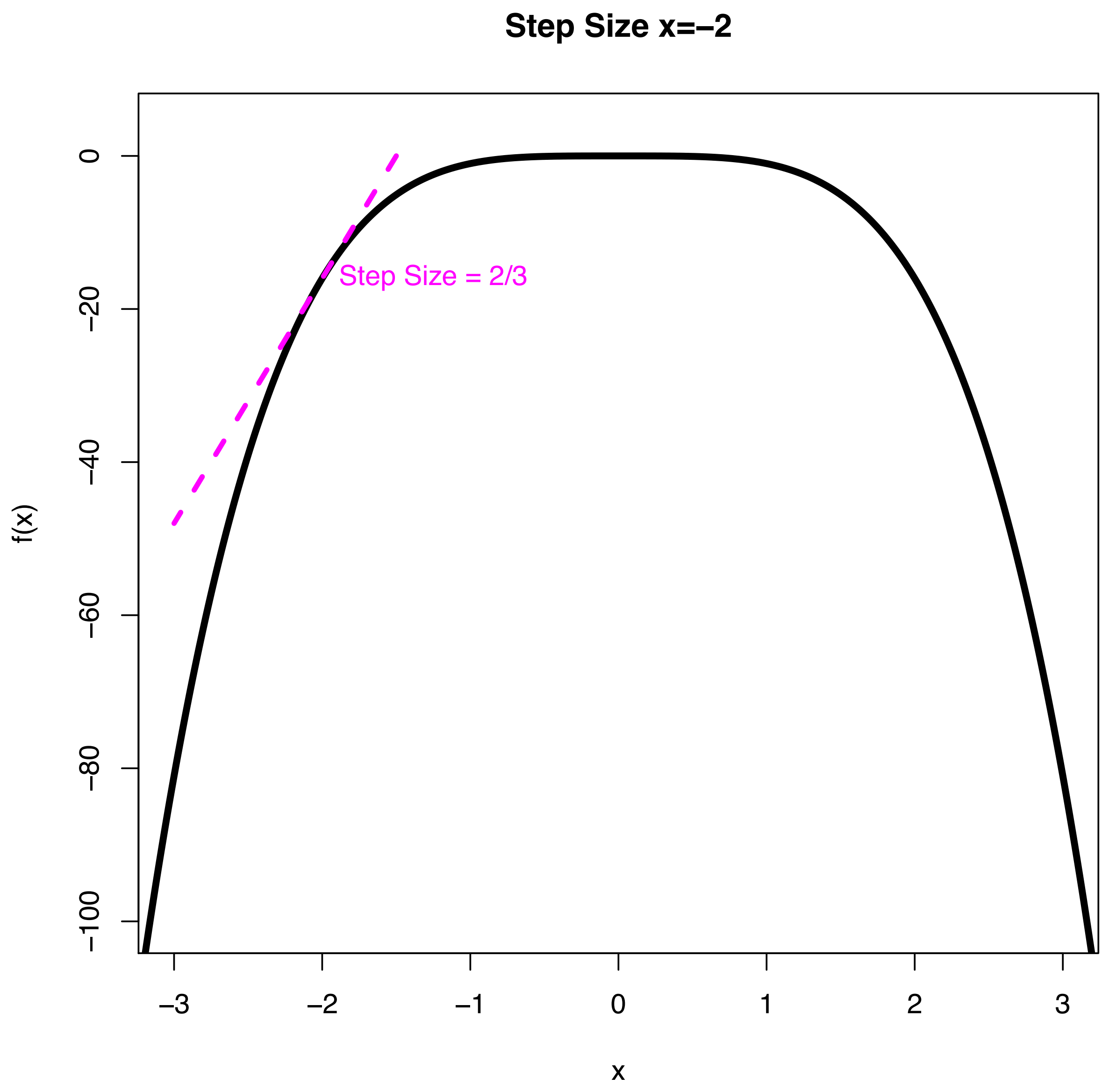
Three stopping rules can be used.

- $|f^{'}(x_{n+1})| < \epsilon$.
- $|x_n - x_{n-1}| < \epsilon$.
- $|f(x_n) - f(x_{n-1})| < \epsilon$.

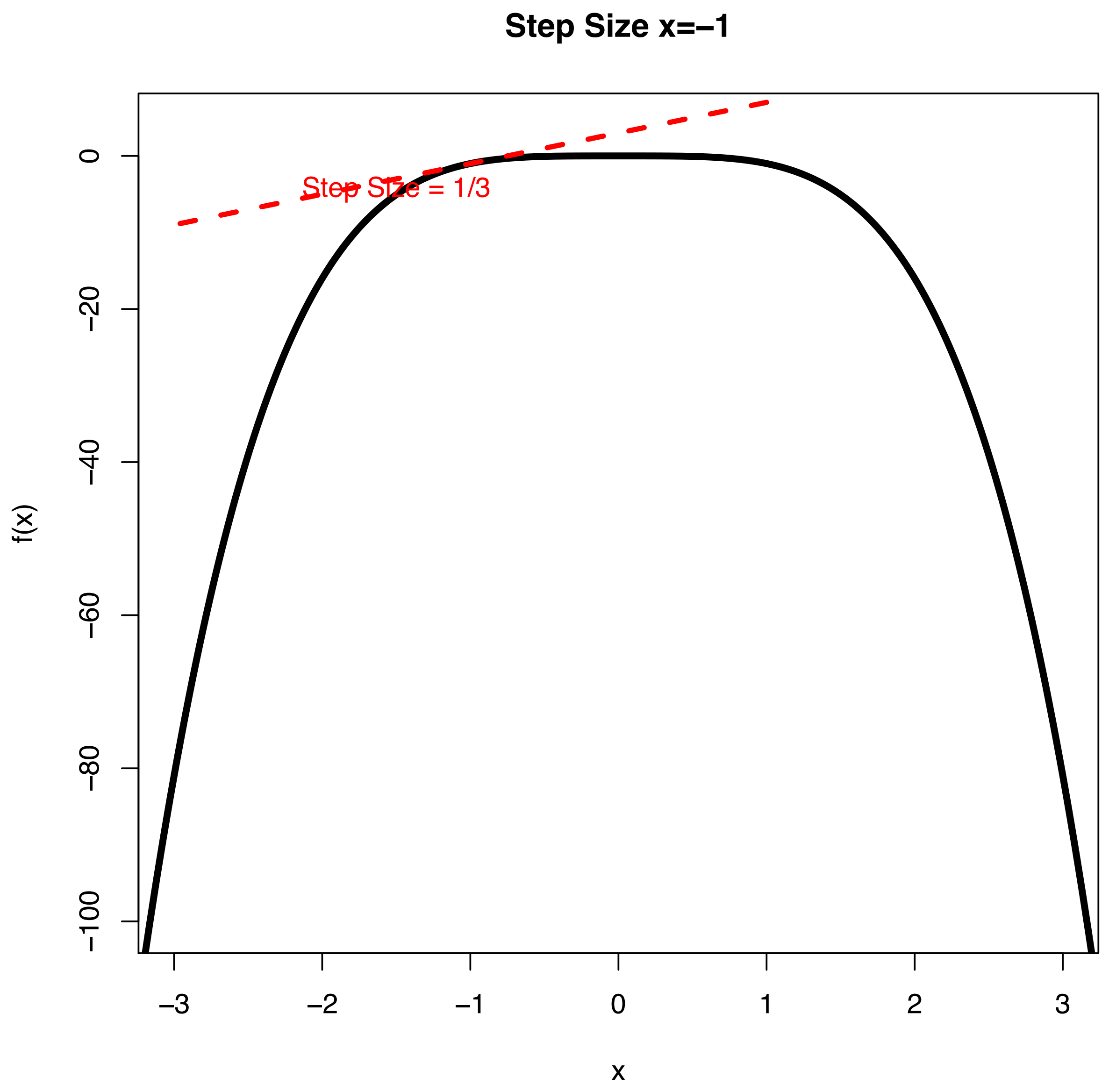
Intuition

- Focus the step size $-\frac{f'(x)}{f''(x)}$.
- The signs of the derivatives control the direction of the next step.
- The size of the derivatives control the size of the next step.
- Consider the concave function $f(x) = -x^4$ which has $f'(x) = -4x^3$ and $f''(x) = -12x^2$. There is a maximum at $x^* = 0$.

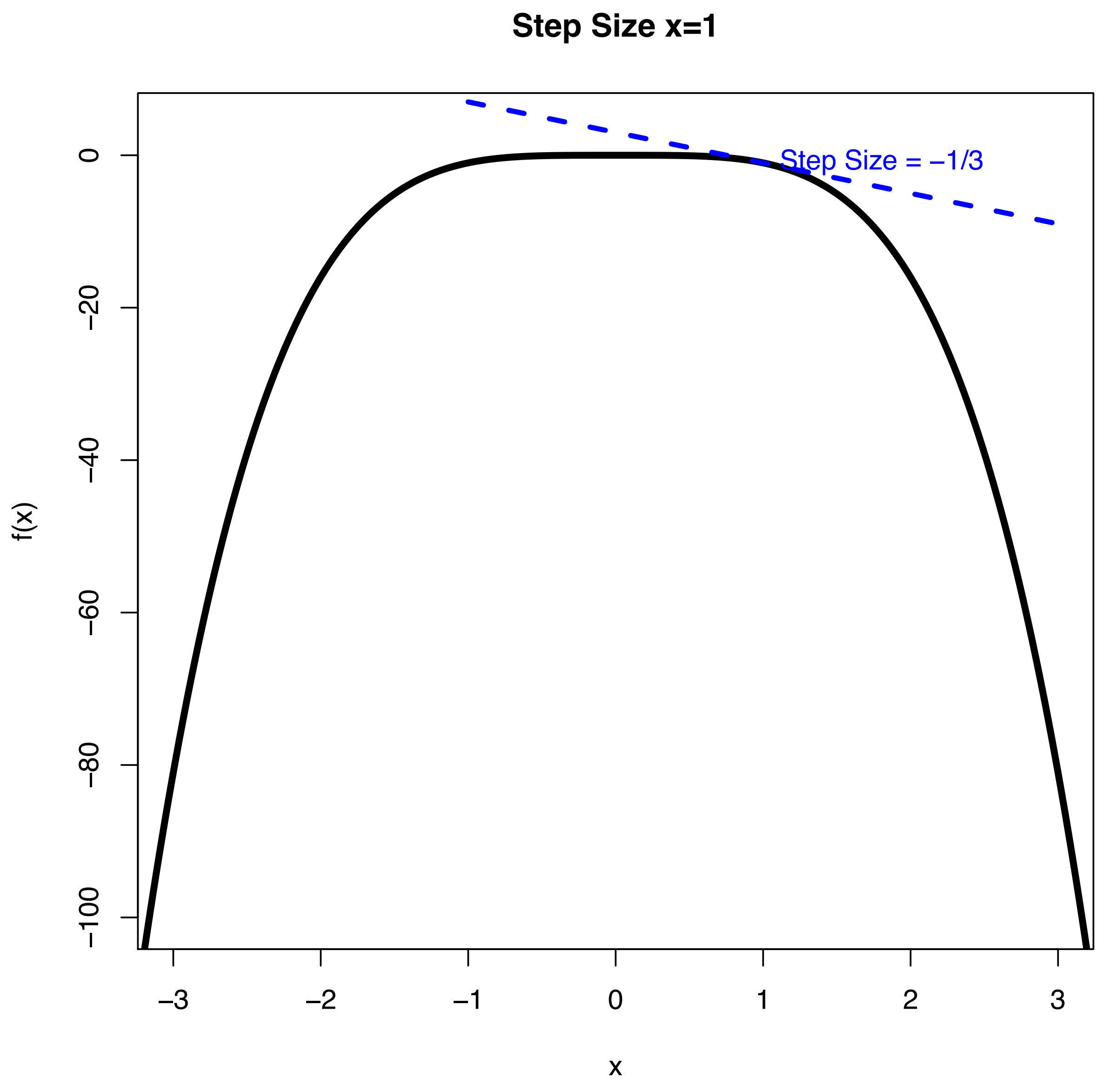
Role of first derivative



Role of first derivative



Role of first derivative



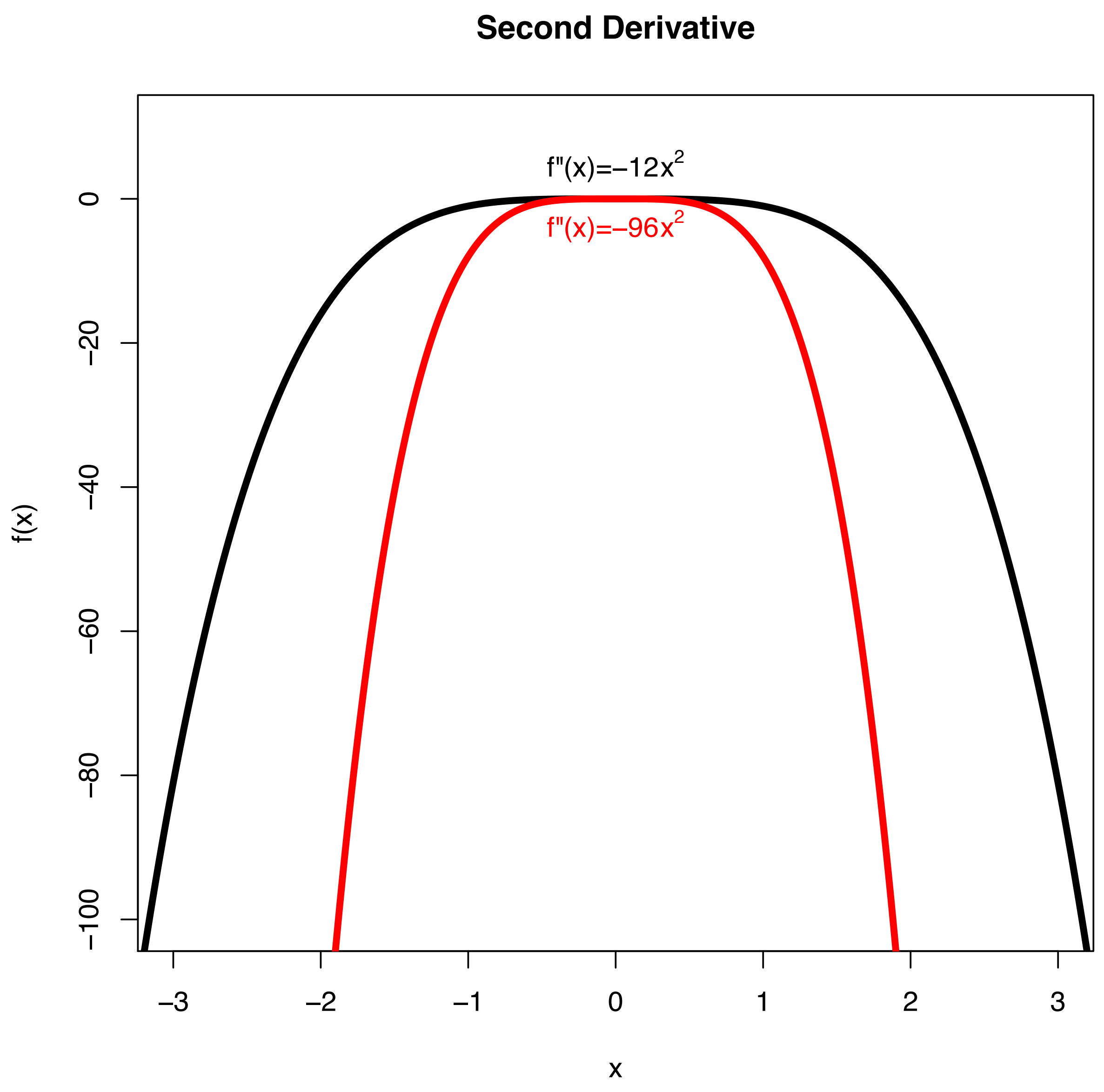
Role of first derivative

- If $f''(x)$ is negative the function is locally concave, and the search is for a local maximum.
- To the left of this maximum $f'(x) > 0$.
- Therefore $-\frac{f'(x)}{f''(x)} > 0$.
- The next step is to the right.
- The reverse holds if $f'(x) < 0$.
- Large absolute values of $f'(x)$ imply a steep slope. A big step is needed to get close to the optimum. The reverse hold for small absolute value of $f'(x)$.

Role of first derivative

- If $f''(x)$ is positive the function is locally convex, and the search is for a local minimum.
- To the left of this maximum $f'(x) < 0$.
- Therefore $-\frac{f'(x)}{f''(x)} > 0$.
- The next step is to the right.
- The reverse holds if $f'(x) > 0$.
- Large absolute values of $f'(x)$ imply a steep slope. A big step is needed to get close to the optimum. The reverse hold for small absolute value of $f'(x)$.

Role of second derivative



Role of second derivative

- Together with the sign of the first derivative, the sign of the second derivative controls the direction of the next step.
- A larger second derivative (in absolute value) implies a larger curvature.
- In this case smaller steps are need to stop the algorithm from overshooting.
- The opposite holds for a small second derivative.

Multidimensional Optimization

Functions with more than one input

- Most interesting optimization problems involve multiple inputs.
  - In determining the most risk efficient portfolio the return is a function of many weights (one for each asset).
  - In least squares estimation for a linear regression model, the sum of squares is a function of many coefficients (one for each regressor).
- How do we optimize for functions $f(x)$ where x is a vector?

Derviatives

- Newton's algorithm has a simple update rule based on first and second derivatives.
- What do these derivatives look like when the function is $y = f(x)$ where y is a scalar and $\mathbf{x}$ is a $d \times 1$ vector?

First derivative

Simply take the partial derivatives and put them in a vector

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_d} \end{pmatrix}$$

This is called the gradient vector.

An example

The function $y = x_1^2 - x_1x_2 + x_2^2 + e^{x_2}$ has gradient vector

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{pmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 + e^{x_2} \end{pmatrix}.$$

Second derivative

Simply take the second order partial derivatives. This will give a matrix

$$\frac{\partial y}{\partial \mathbf{x} \partial \mathbf{x}'} = \begin{pmatrix} \frac{\partial^2 y}{\partial x_1^2} & \frac{\partial^2 y}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 y}{\partial x_1 \partial x_d} \\ \frac{\partial^2 y}{\partial x_2 \partial x_1} & \frac{\partial^2 y}{\partial x_2^2} & \cdots & \frac{\partial^2 y}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 y}{\partial x_d \partial x_1} & \frac{\partial^2 y}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 y}{\partial x_d^2} \end{pmatrix}.$$

This is called the Hessian matrix.

An example

The function $y = x_1^2 - x_1 x_2 + x_2^2 + e^{x_2}$ has Hessian matrix

$$\frac{\partial y}{\partial \mathbf{x} \partial \mathbf{x}'} = \begin{pmatrix} 2 & -1 \\ -1 & 2 + e^{x_2} \end{pmatrix}$$

Newton's algorithm for multidimensional optimization

We can now generalise the update step in Newton's method:

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \left(\frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x} \partial \mathbf{x}'} \right)^{-1} \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

Now write code to minimise $y = x_1^2 - x_1 x_2 + x_2^2 + e^{x_2}$.

Maximum likelihood Estimate for linear models

Assume you want to make a regression model

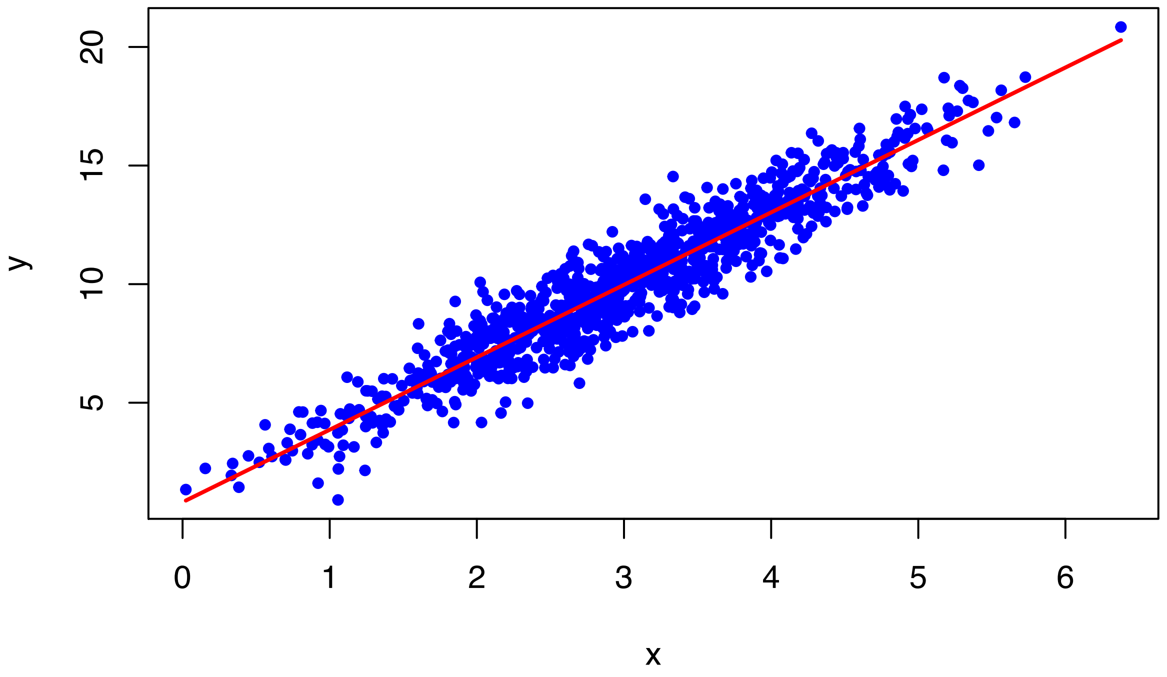
$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i, \text{ where } \epsilon_i \sim N(0, 1).$$

- How do we estimate the parameters?
- Write down the log likelihood function with respect to the unknown parameters.
- Write down the gradient for the log likelihood function.
- Write down the Hessian for the log likelihood function.
- Use your newton function to obtain the best parameter estimate.

Optimizing Using Newton's Method

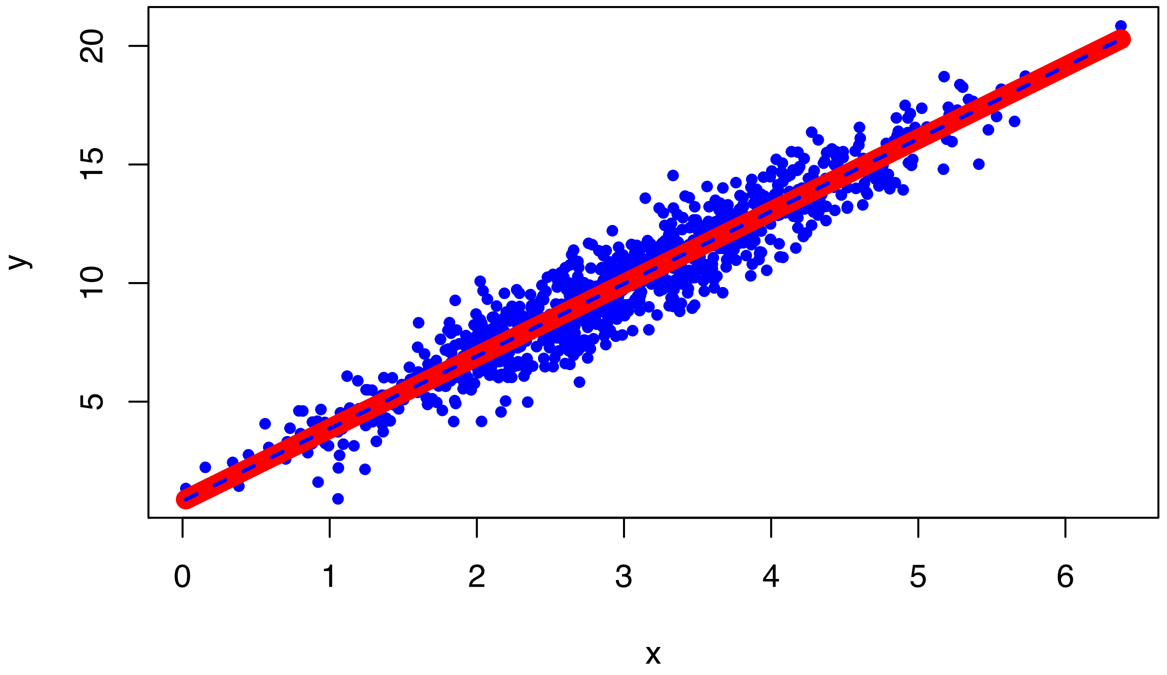
```r
## Generate some data
beta0 <- 1
beta1 <- 3
sigma <- 1
n <- 1000
x <- rnorm(n, 3, 1)
y <- beta0 + x * beta1 + rnorm(n, mean = 0, sd = sigma)
# plot(x, y, col = 'blue', pch = 20)
```

```r
## Make the log normal likelihood function
func = function(beta) {
    sum((y - beta[1] - beta[2] * x)^2)
}

grad = function(beta) {
    matrix(c(sum(-2 * (y - beta[1] - beta[2] * x)), sum(-2 *
        x * (y - beta[1] - beta[2] * x))), 2, 1)
}

hess = function(beta) {
    matrix(c(2 * length(x), 2 * sum(x), 2 * sum(x), 2 * sum(x^2)),
        2, 2)
}

## The optimization
source("./RCode/newton.R")
optimOut <- newton(function(beta) {
    list(func(beta), grad(beta), hess(beta))
}, c(-4, -5))
beta0Hat <- optimOut[1]
beta1Hat <- optimOut[2]
yHat <- beta0Hat + beta1Hat * x

## Plot
plot(x, y, pch = 20, col = "blue")
points(sort(x), yHat[order(x)], type = "l", col = "red", lwd = 2)
```



Comparison with OLS

```
myLM <- lm(y ~ x)
myLMCoef <- myLM$coefficients
yHatOLS <- myLMCoef[1] + myLMCoef[2] * x
plot(x, y, pch = 20, col = "blue")
points(sort(x), yHat[order(x)], type = "l", col = "red", lwd = 10)
points(sort(x), yHatOLS[order(x)], type = "l", col = "blue",
    lty = "dashed", lwd = 2, pch = 20)
```



Lab Session 7

Use Newton's method to find the maximum likelihood estimate for the coefficients in a logistic regression. The steps are:

1. Write down likelihood function.
2. Find the gradient and Hessian matrix.
3. Code these up in R.
4. Simulate some data from a logistic regression model and test your code.

References

Chapters 10 and 12 of the book "Introduction to Scientific Programming and Simulation Using R".

Statistical Computing

Lecture 7: Quasi-Newton Methods

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Quasi-Newton Methods

- One of the most difficult parts of the Newton method is working out the derivatives especially the Hessian.
- However methods can be used to approximate the Hessian and also the gradient.
- These are known as Quasi-Newton Methods.
- In general they will converge slower than pure Newton methods.

The BFGS Algorithm

Introduced over several papers by Broyden, Fletcher, Goldfarb and Shanno. It is the most popular Quasi-Newton algorithm.

- Recall Newton iteration:
$$x_{n+1} = x_n - f^{''}(x_n)^{-1}f^{'}(x_n).$$
- Is there some matrix to replace $f^{''}(x_n)$ or $f^{''}(x_n)^{-1}$?
- Can we use a revised iteration: $x_{n+1} = x_n - B_n^{-1}f^{'}(x_n)$, where B_n is simpler to compute but still allows the algorithm to converge quickly?

The BFGS Algorithm

- The idea with Quasi-Newton is to find a solution B_n to the problem
$$f^{'}(x_n) - f^{'}(x_{n-1}) = B_n(x_n - x_{n-1}).$$
- Let $y_n = f^{'}(x_n) - f^{'}(x_{n-1})$ and $s_n = x_n - x_{n-1}$, one updating procedures for B_n:
$$B_n = B_{n-1} + \frac{y_n y_n^{'}}{y_n^{'} s_n} - \frac{B_{n-1}s_n s_n^{'} B_{n-1}^{'}}{s_n^{'} B_{n-1} s_n}.$$

The L-BFGS-B Algorithm

- The R function `optim()` also has a variation called L-BFGS-B.
- The L-BFGS-B uses less computer memory than BFGS and allows for box constraints.

Box Constrains

- Box constraints have the form
$$l_i \leq x_i \leq u_i, \ \forall i.$$
- In statistics this can be very useful. Often parameters are constrained.
  - Variance must be greater than 0.
  - For a stationary AR(1), coefficients must be between -1 and 1.
  - Weights in a portfolio must be between 0 and 1.

`optim()` in R

- `optim()` requires at least two inputs.
  - Initial values
  - The function that needs to be optimized
- By default it minimises a function.

- A function that computes the gradient vector can also be provided.

- The optimization method can be set (choices include BFGS, L-BFGS-B and Nelder-Mead) .
- Lower and upper bounds can be set through the arguments lower and upper if the L-BFGS-B method is used.

`optim()` in R

- Further arguments can be passed in an argument called `control`.
- Some things that can be included in this list are
  - Maximum number of iterations (`maxit`)
  - Information about the algorithm (`trace`)
  - How often to display information about the algorithm (`REPORT`)

`optim()` in R

- The result of optim can be saved in an object that is a list containing
  - The value of the function at the turning point (`value`)
  - The optimal parameters (`par`)
  - Useful information about whether the algorithm has converged (`convergence`)
- For all algorithms convergence = 0 if the algorithm has converged (slightly confusing).

Lab Session 8

Use `optim()` to carry out maximum likelihood for the Logistic regression model.

References

Chapter 3.3 of the book “Advanced Statistical Computing”.

Statistical Computing

Lecture 8: Derivative Free Methods

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Motivation

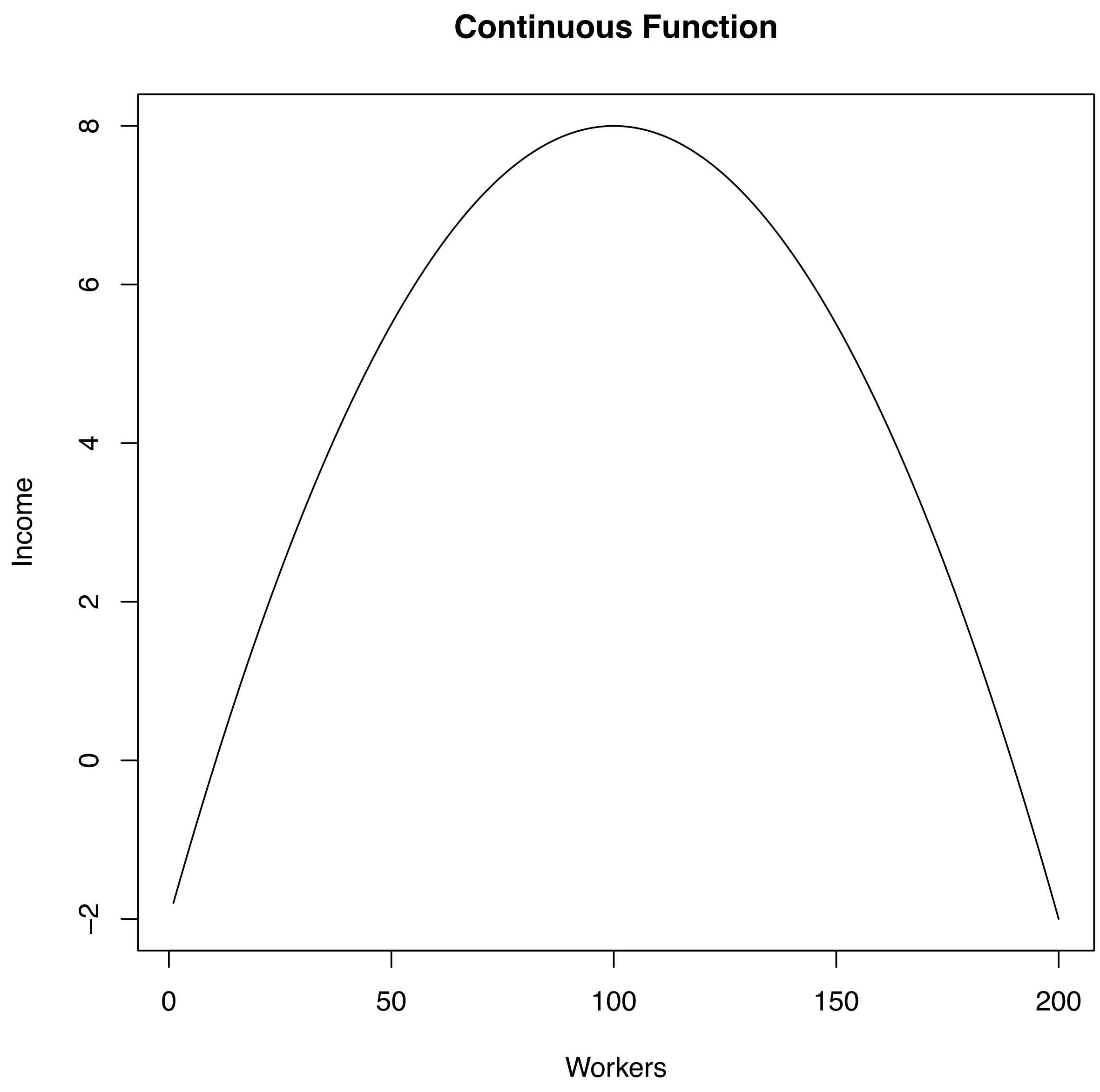
Discontinuous Functions

- The Newton Method requires first and second derivatives.
- If derivatives are not available the they can be approximated by Quasi-Newton methods.
- What if the derivatives do not exist?
- This may occur if there are discontinuities in the function.

Business Example

- Suppose the aim is to optimize income of the business by selecting the number of workers.
- In the beginning adding more workers leads to more income for the business.
- If too many workers are employed, they may be less efficient and the income of the company goes down.

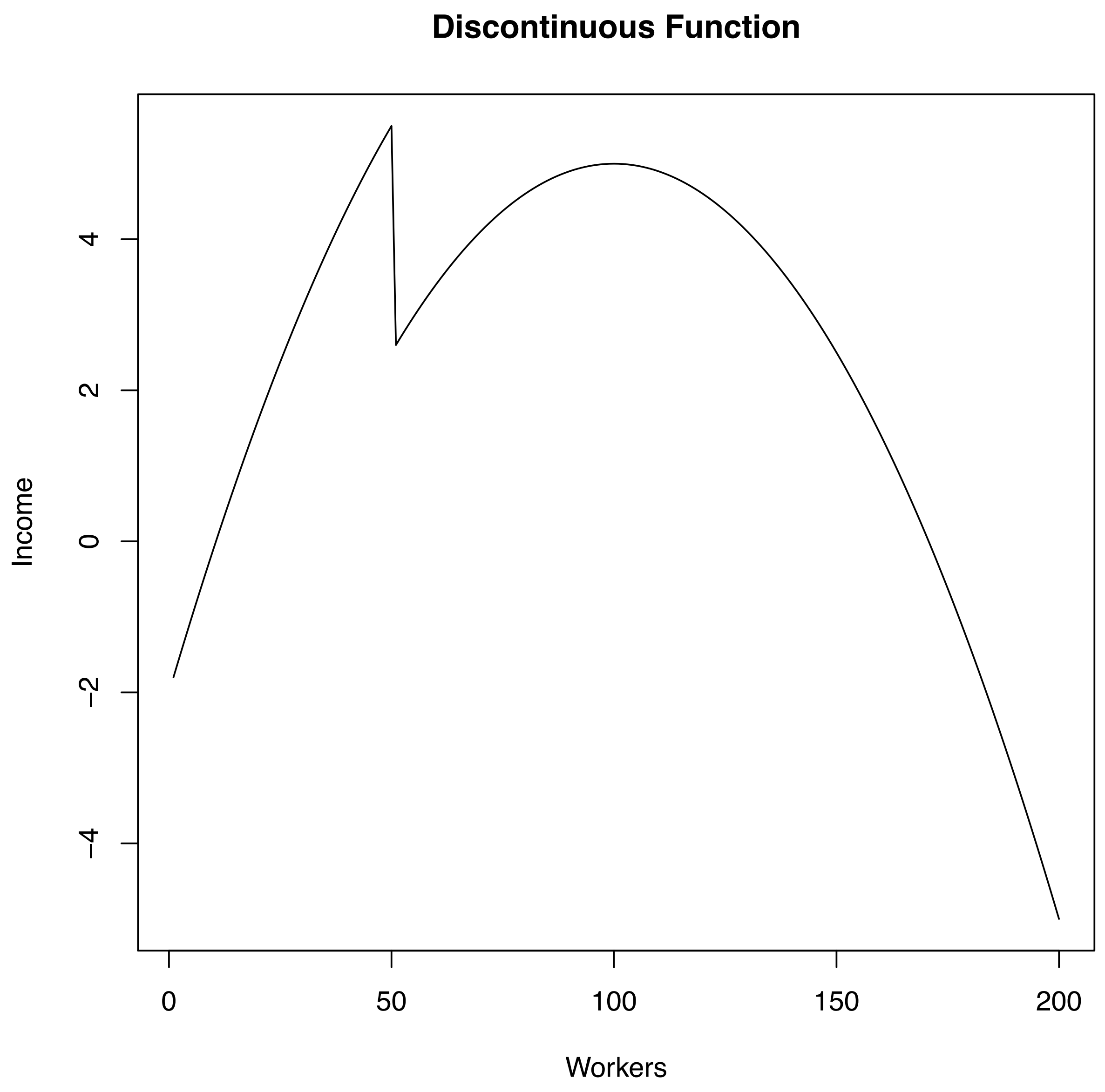
Business Example



Business Example

- Now suppose that there is a tax that the company must pay.
- Companies with less than 50 workers do not pay the tax.
- Companies with more than 50 workers do pay the tax.
- How does this change the problem?

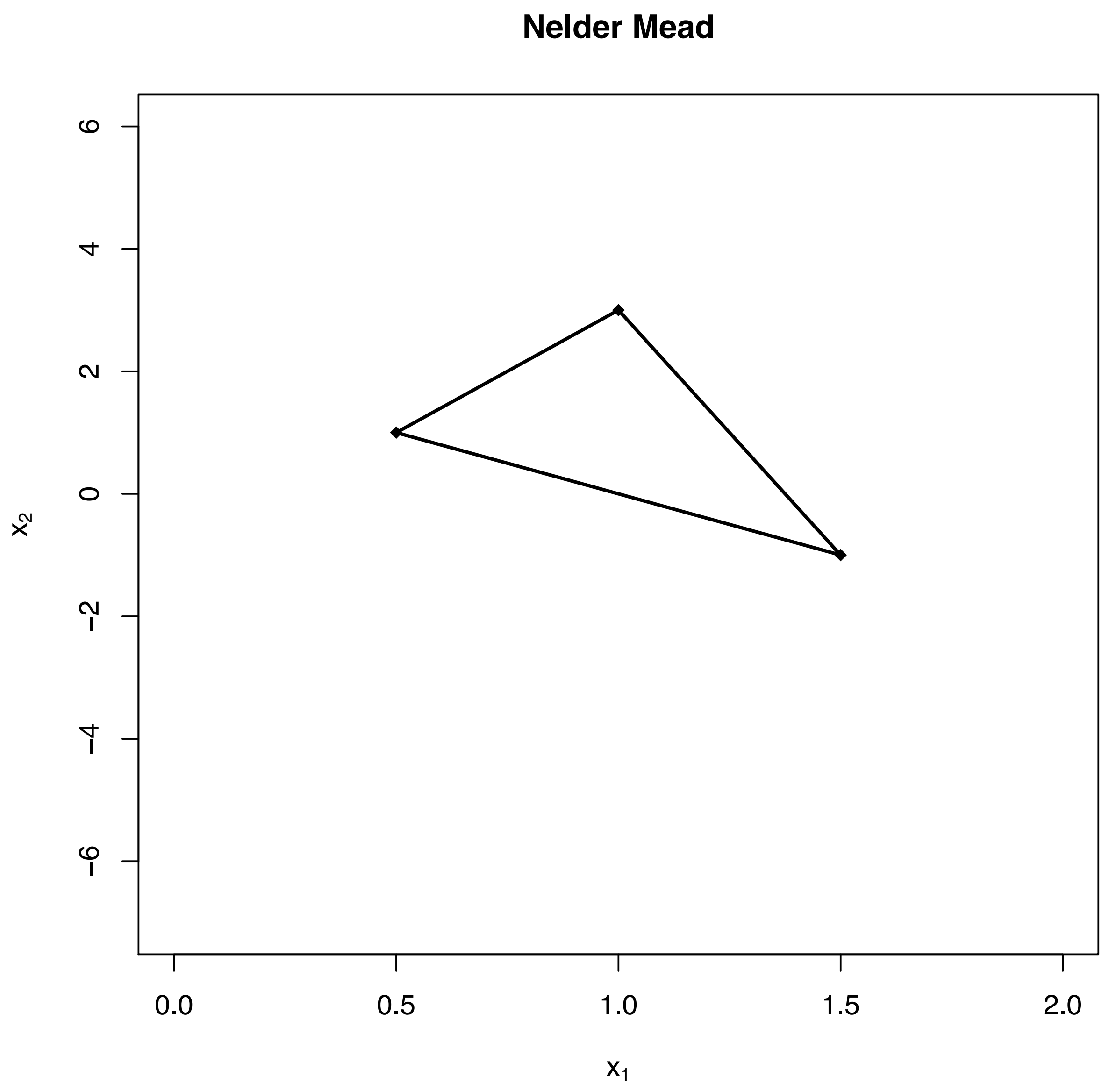
Business Example



The Nelder Mead Algorithm

- The Nelder Mead algorithm is robust even when the functions are discontinuous.
- The idea is based on evaluating the function at the vertices of an n-dimensional simplex where n is the number of input variables into the function.
- For two dimensional problems the n-dimensional simplex is simply a triangle, and each corner is one vertex
- In general there are n + 1 vertices.

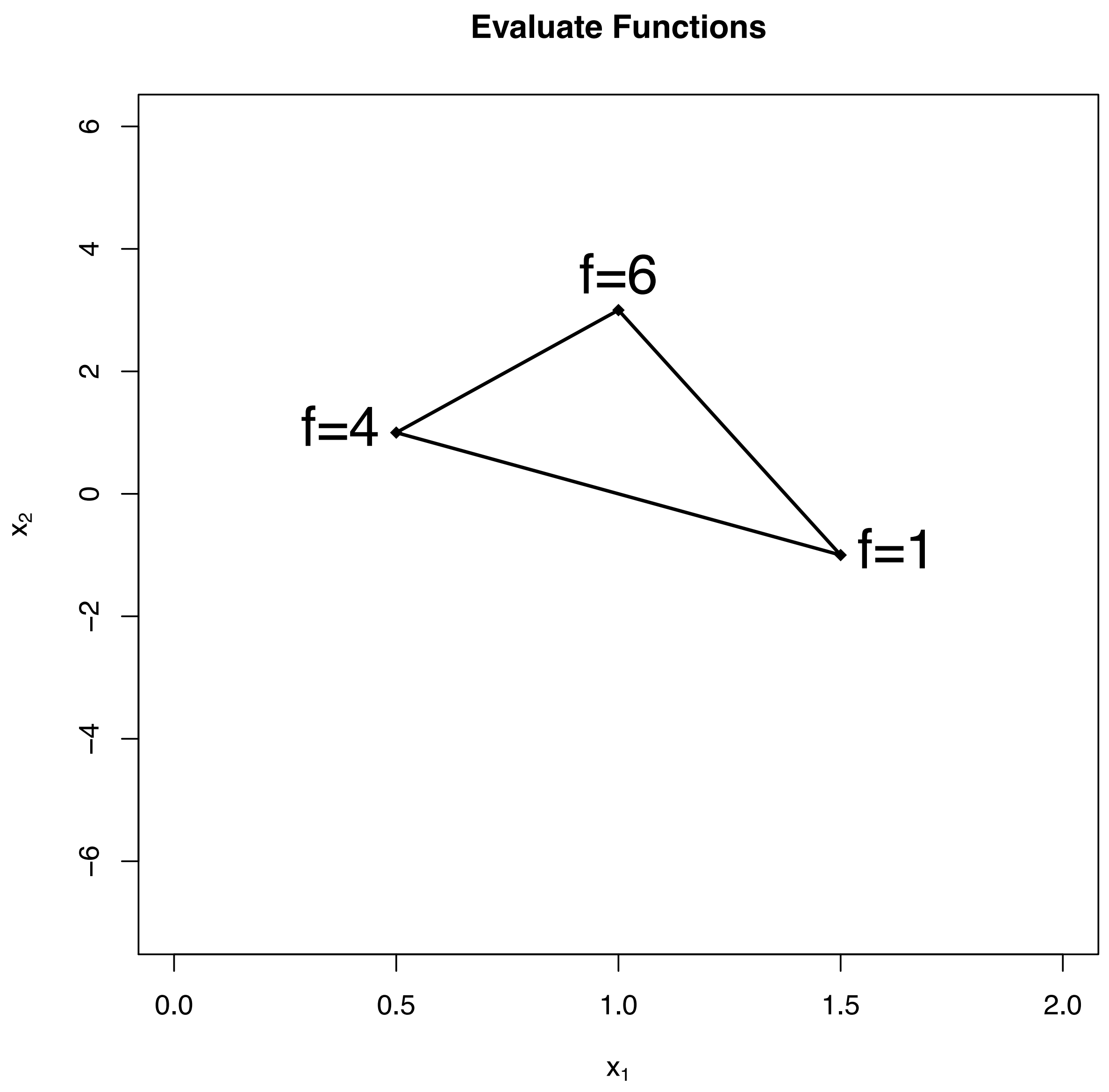
A 2-dimensional simplex



Step 1: Evaluate Function

- For each vertex $\mathbf{x}_j$ evaluate the function $f(\mathbf{x}_j)$
- Order the vertices so that
$$f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \ldots \leq f(\mathbf{x}_{n+1}).$$
- Suppose that the aim is to minimize the function, then $f(\mathbf{x}_{n+1})$ is the worst point.
- The aim is to replace $f(\mathbf{x}_{n+1})$ with a better point.

A 2-dimensional simplex



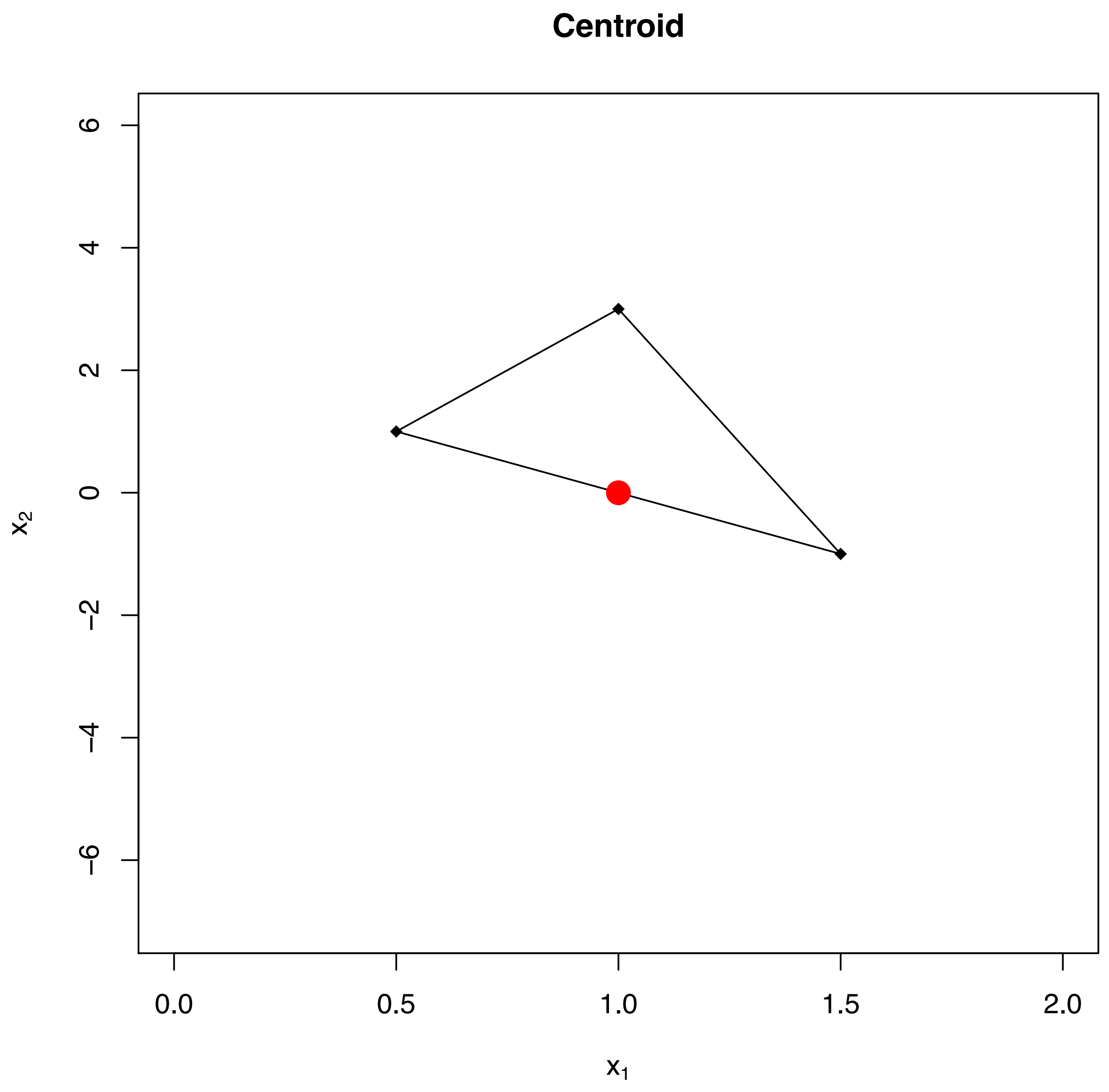
Step 2: Find Centroid

- After eliminating the worst point $\mathbf{x}_{n+1}$, compute the centroid of the remaining n points

$$\mathbf{x}_0 = \frac{1}{n}\sum_{j=1}^{n}\mathbf{x}_j.$$

- For the 2-dimensional example the centroid will be in the middle of a line.

Find Centroid



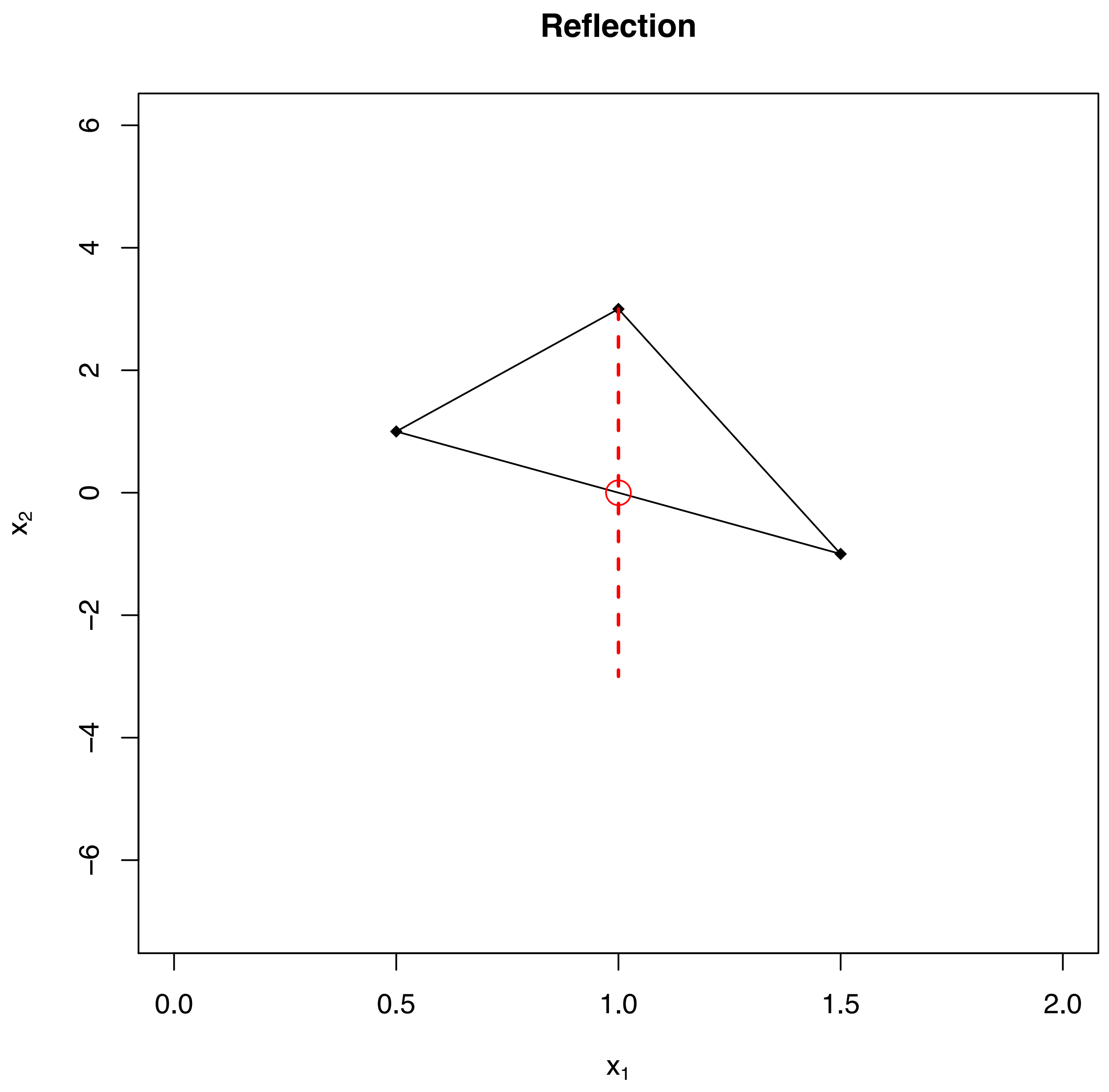
Step 3: Find reflected point

- Reflect the worst point around the centroid to get the reflected point.
- The formula is:

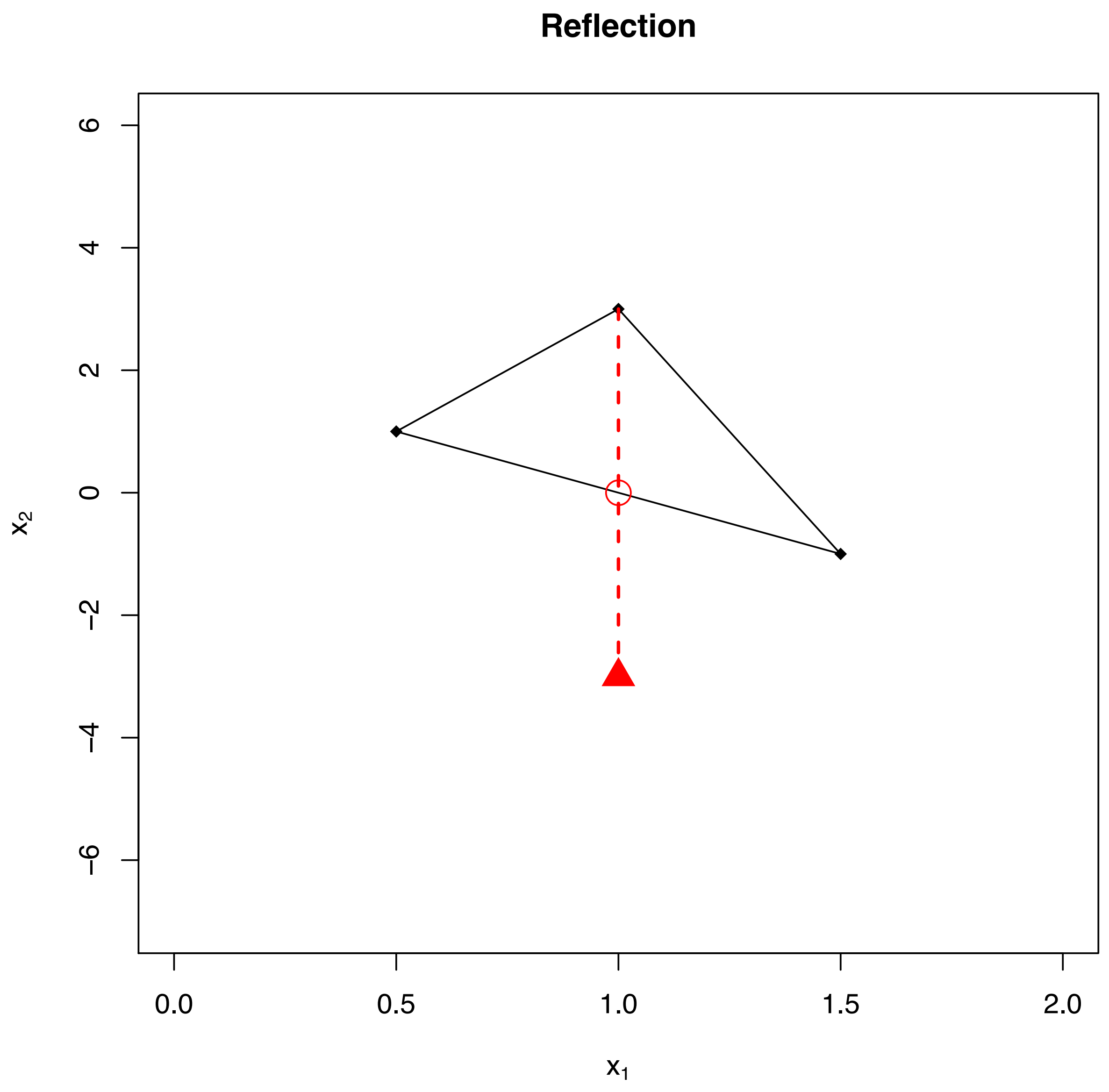
$$\mathbf{x}_r = \mathbf{x}_0 + \alpha(\mathbf{x}_0 - \mathbf{x}_{n+1}).$$

- A common choice is $\alpha = 1$.
- In this case the reflected point is the same distance from the centroid as the worst point.

Find Reflected Point



Find Reflected Point

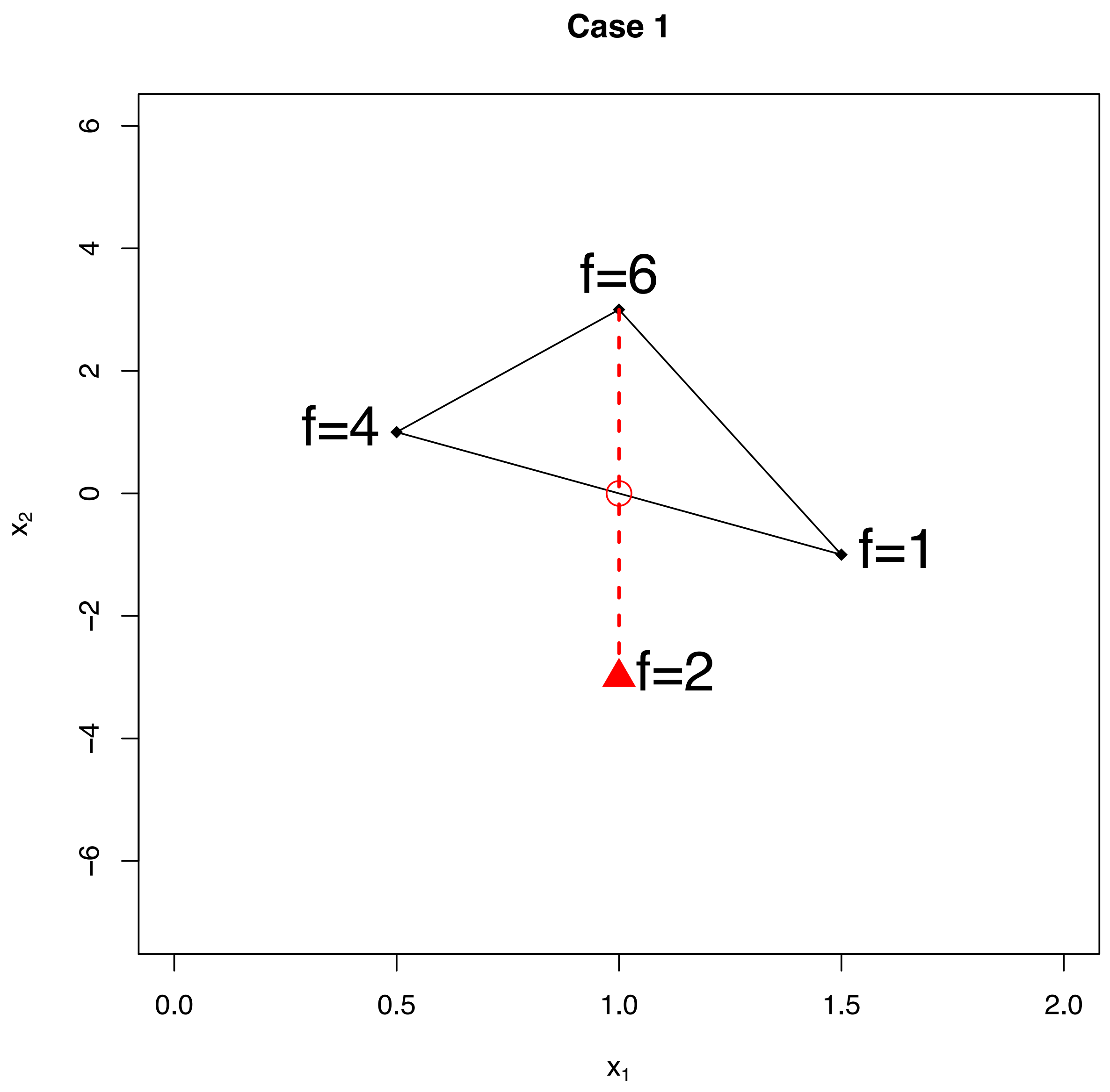


Three cases

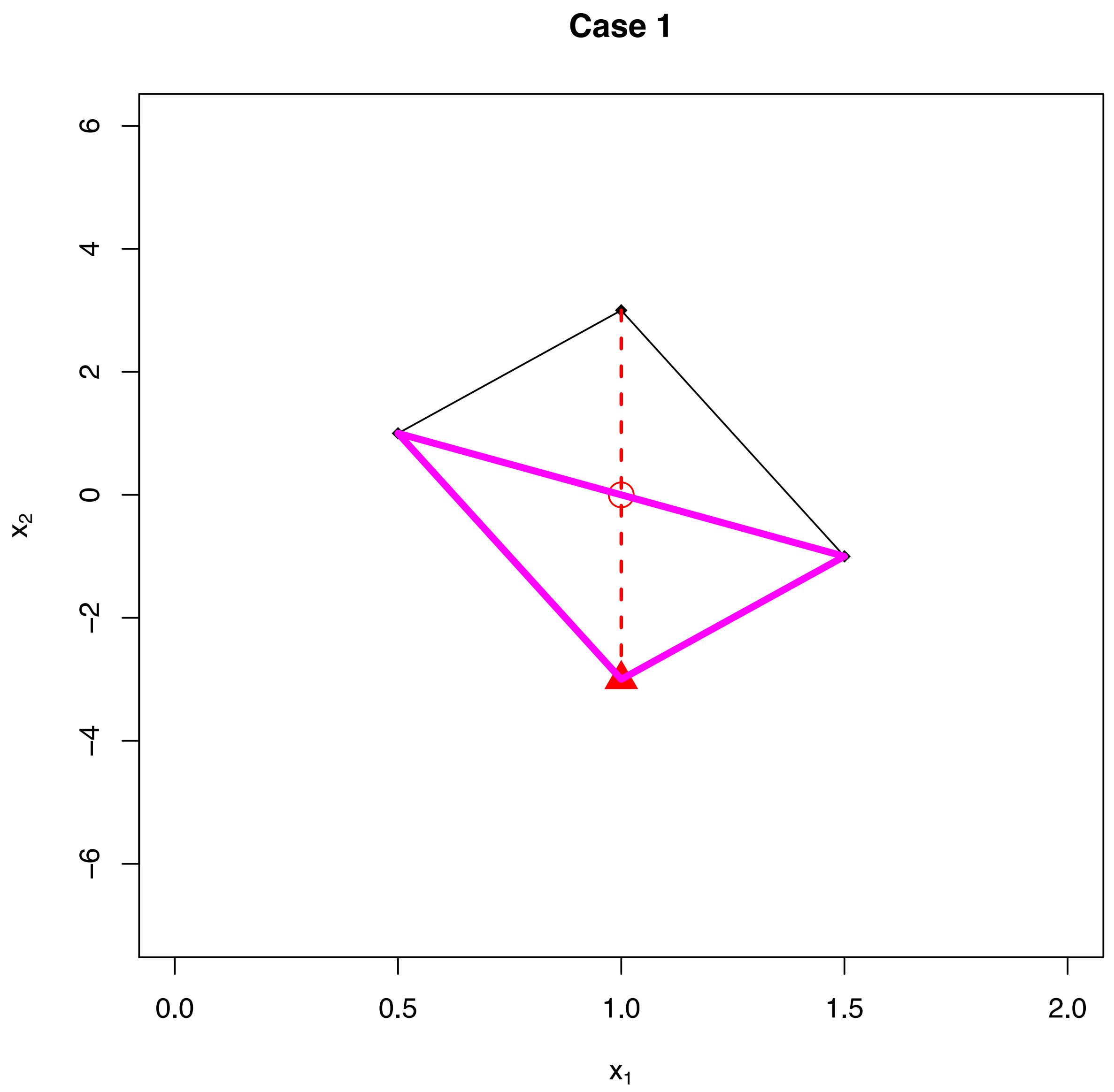
1. $f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$
   - $\mathbf{x}_r$ is neither best nor worst point
2. $f(\mathbf{x}_r) < f(\mathbf{x}_1)$
   - $\mathbf{x}_r$ is the best point
3. $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$
   - $\mathbf{x}_r$ is the worst point

Case 1

In Case 1 a new simplex is formed with $\mathbf{x}_{n+1}$ replaced by the reflected point $\mathbf{x}_r$. Then go back to step 1.



Case 1



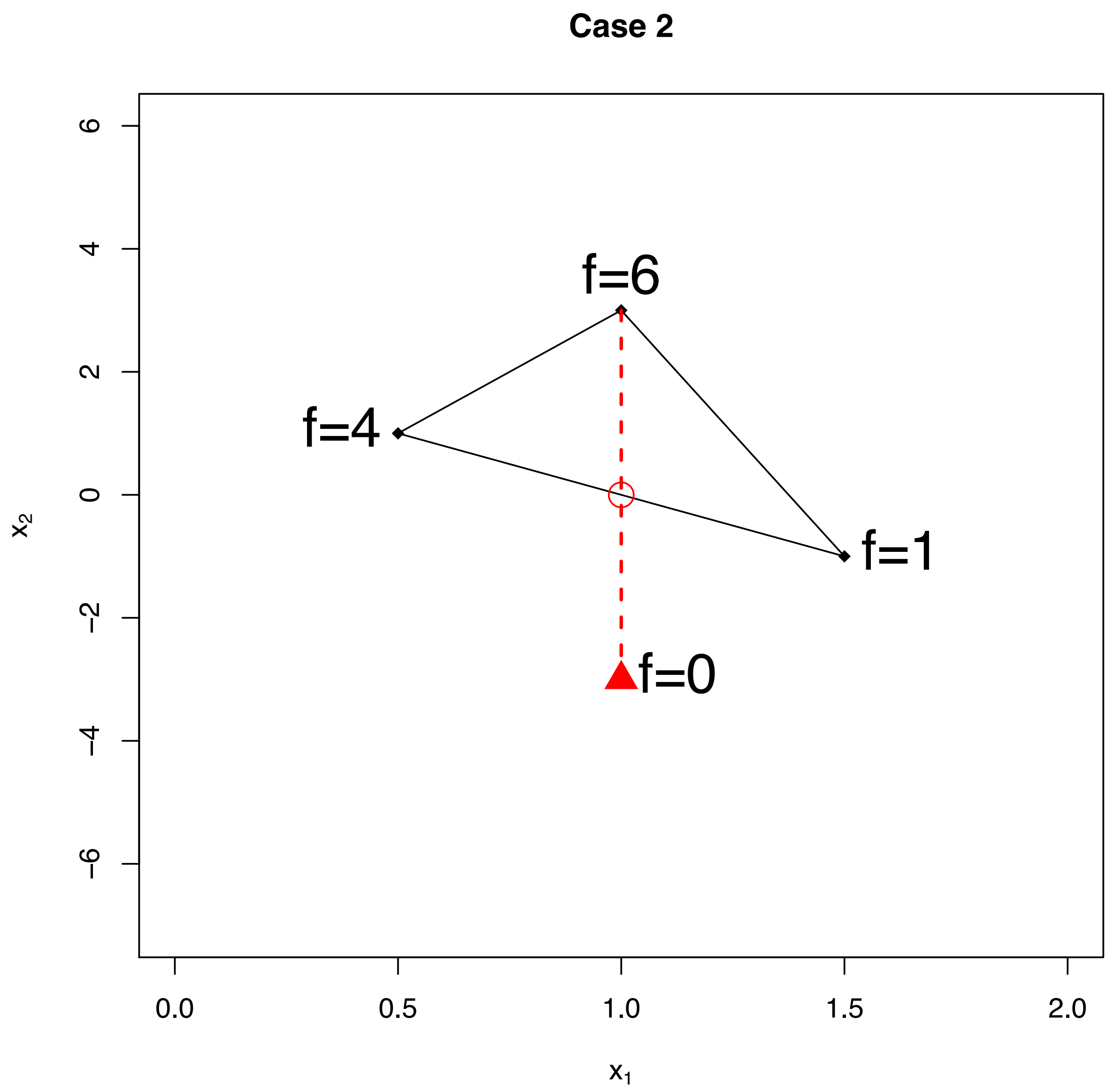
Case 2

In Case 2, $\mathbf{x}_r < \mathbf{x}_1$. A good direction has been found so we expand along that direction

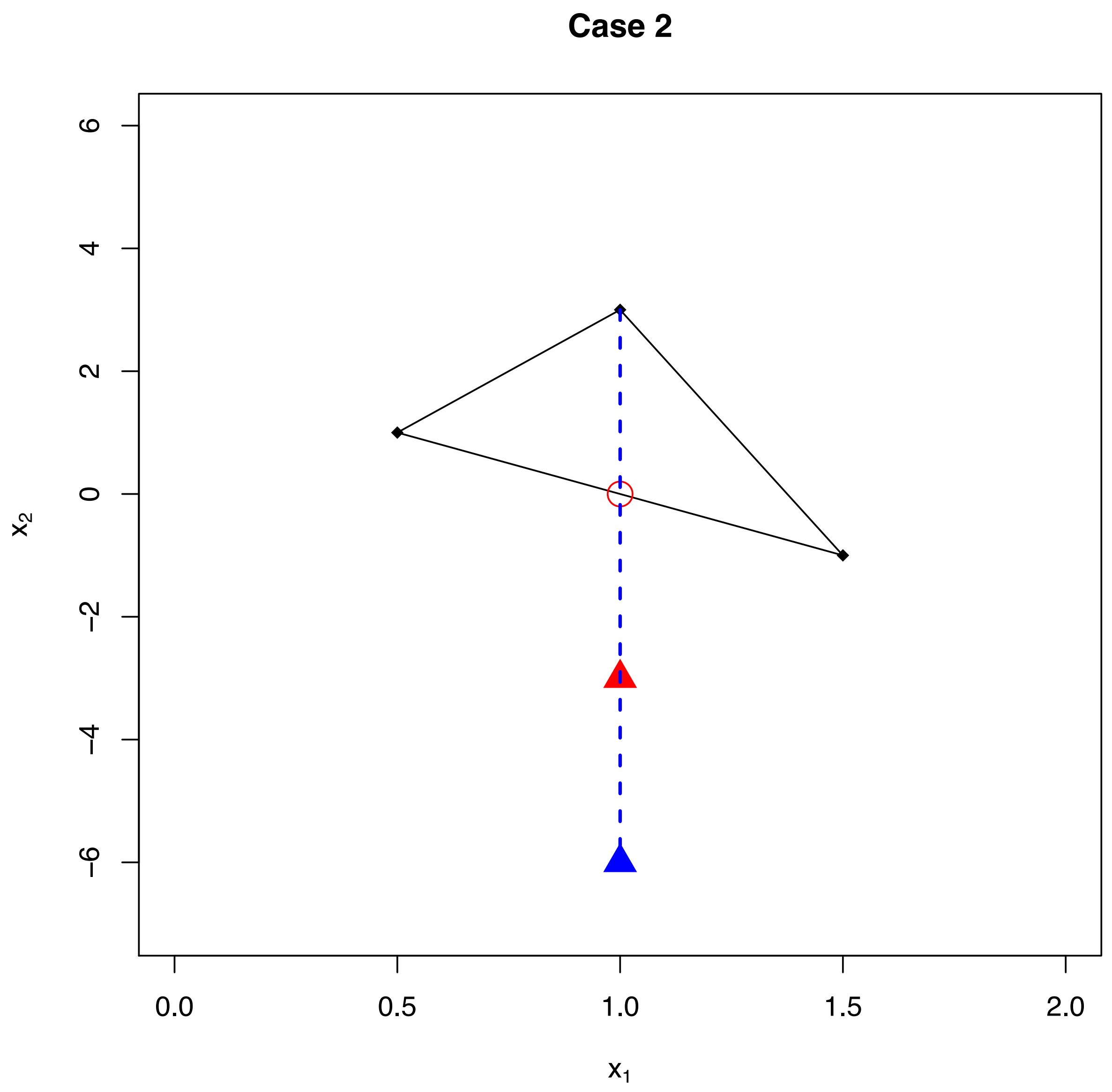
$$\mathbf{x}_e = \mathbf{x}_0 + \gamma(\mathbf{x}_r - \mathbf{x}_0).$$

A common choice is $\gamma = 2$

Case 2



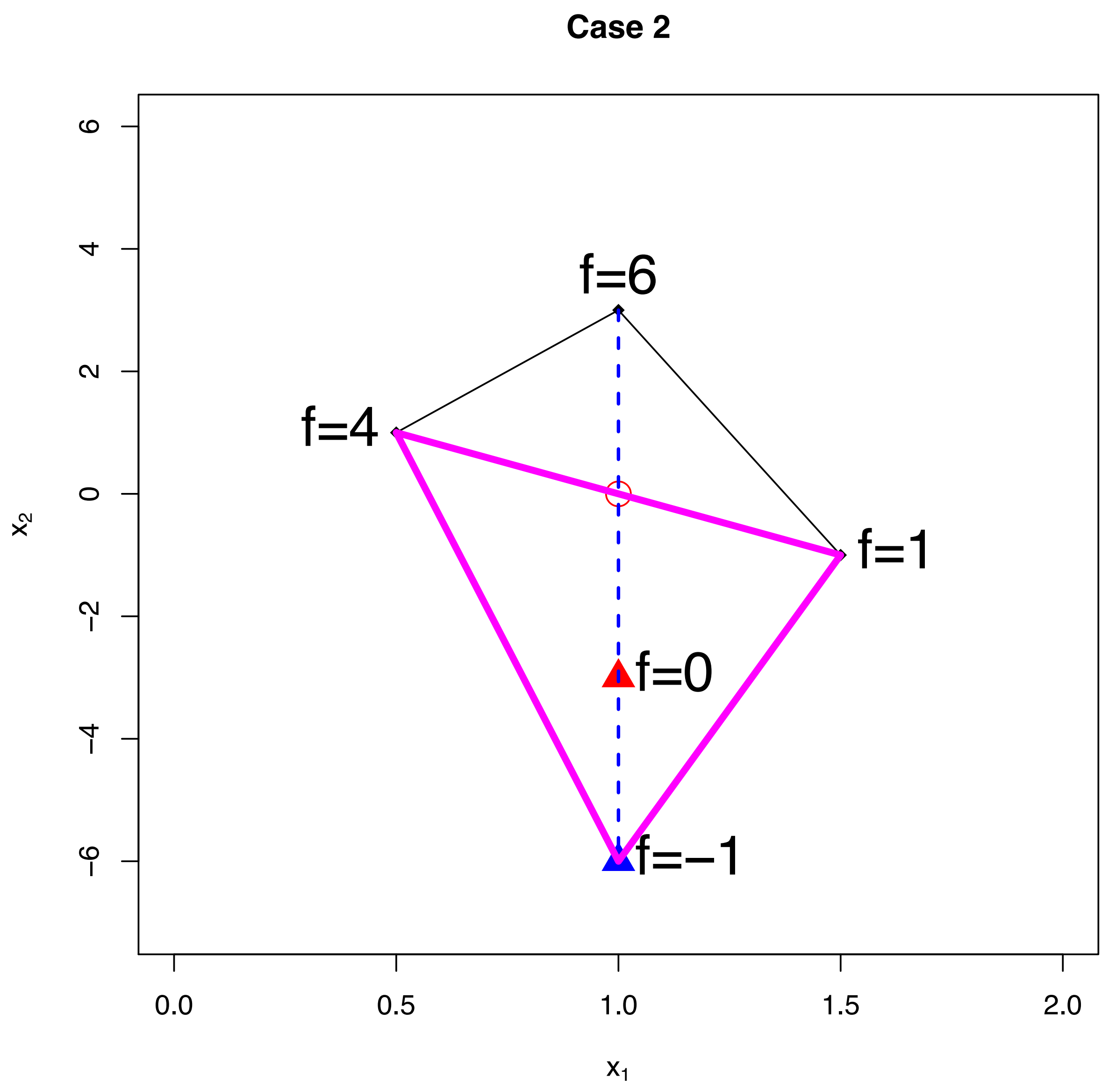
Case 2



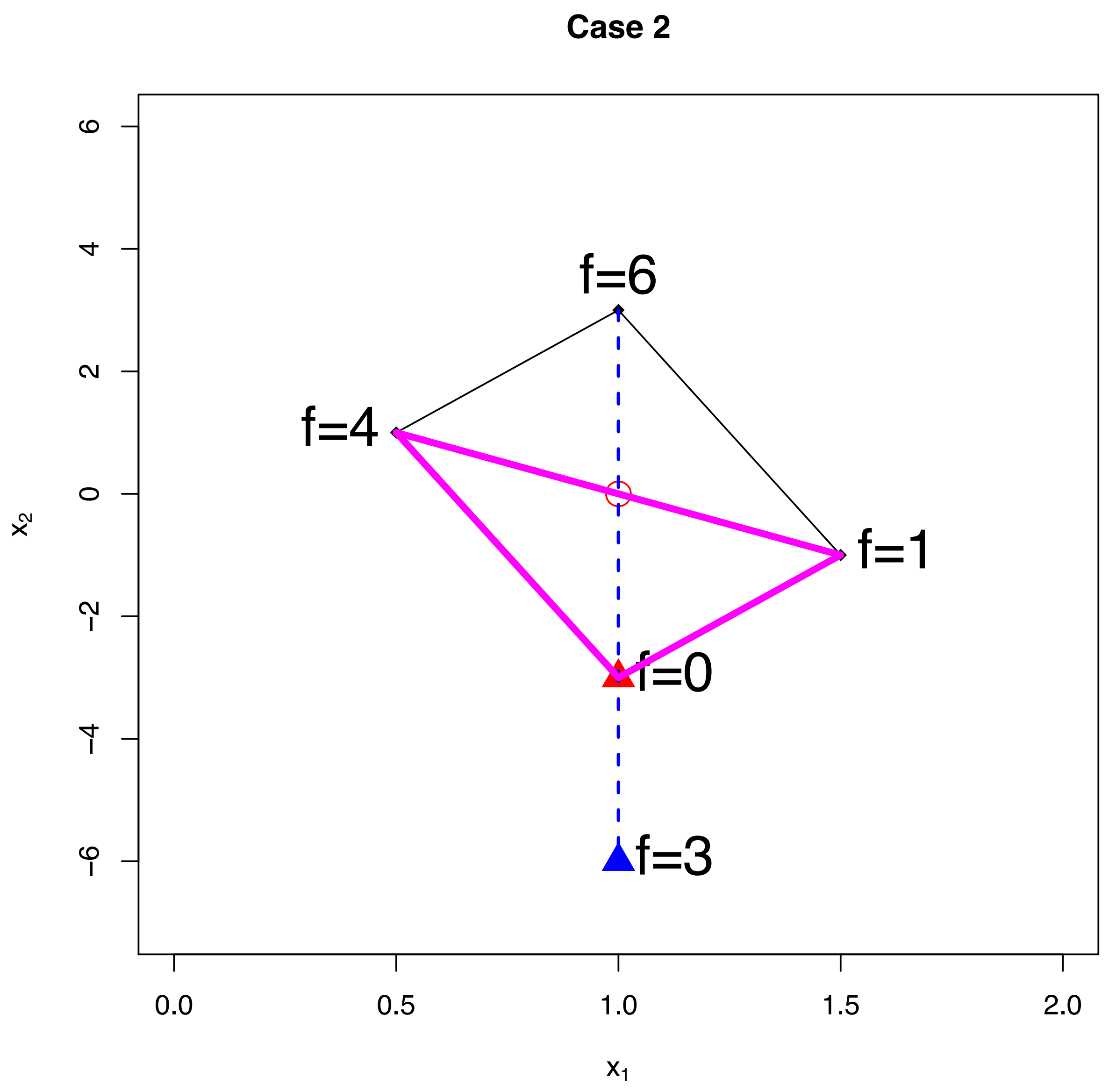
Choosing the Expansion Point

- Evaluate $f(\mathbf{x}_e)$.
- If $f(\mathbf{x}_e) < f(\mathbf{x}_r)$:
  - The expansion point is better than the reflection point. Form a new simplex with the expansion point
- If $f(\mathbf{x}_r) \leq f(\mathbf{x}_e)$:
  - The expansion point is not better than the reflection point. Form a new simplex with the reflection point.

Keep Expansion Point



Keep Relection Point



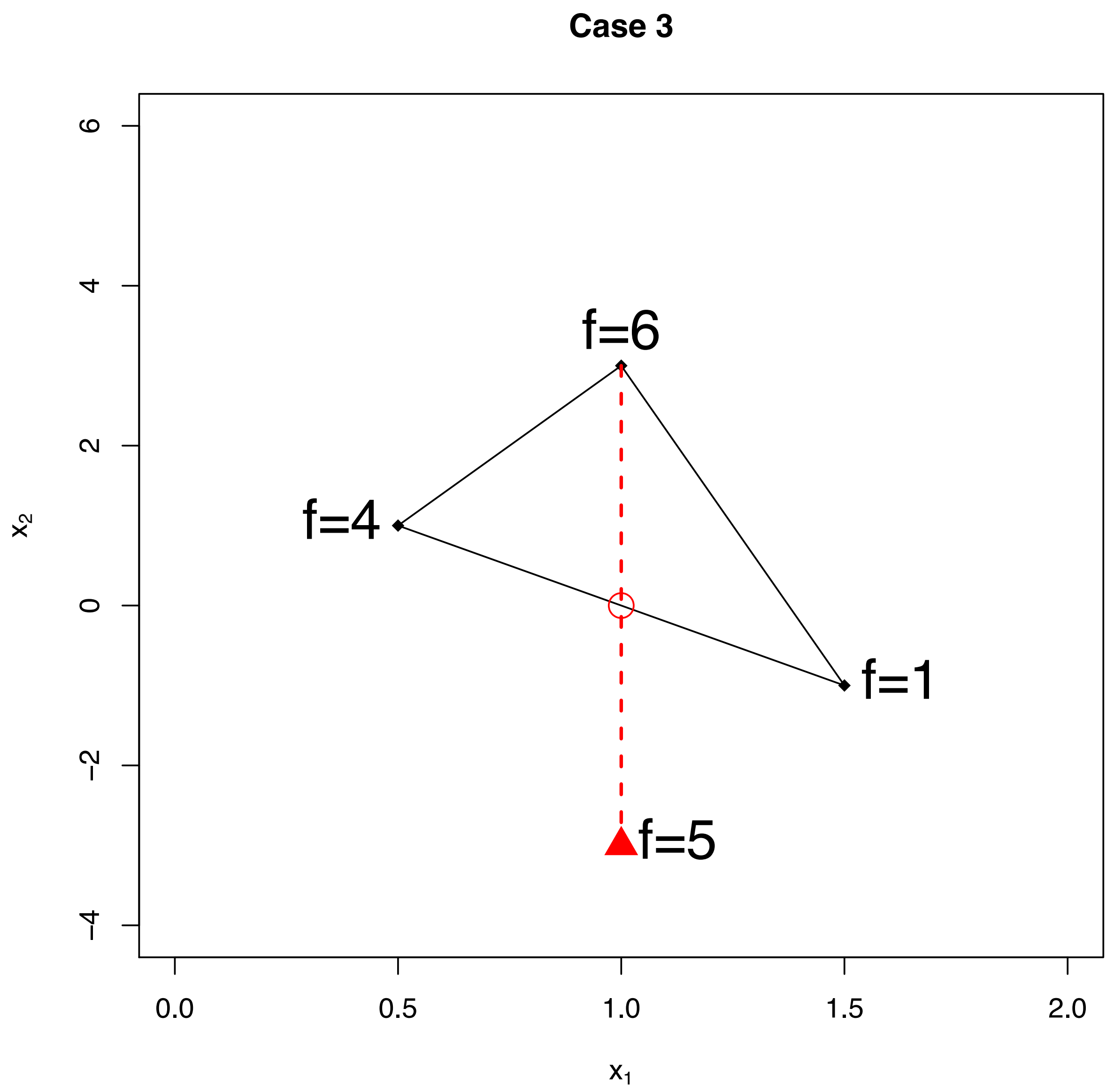
Case 3

Case 3 implies that there may be a valley between $\mathbf{x}_{n+1}$ and $\mathbf{x}_r$ so find the contracted point. A new simplex is formed with the contraction point if it is better than $\mathbf{x}_{n+1}$

$$\mathbf{x}_c = \mathbf{x}_0 + \rho(\mathbf{x}_{n+1} - \mathbf{x}_0)$$

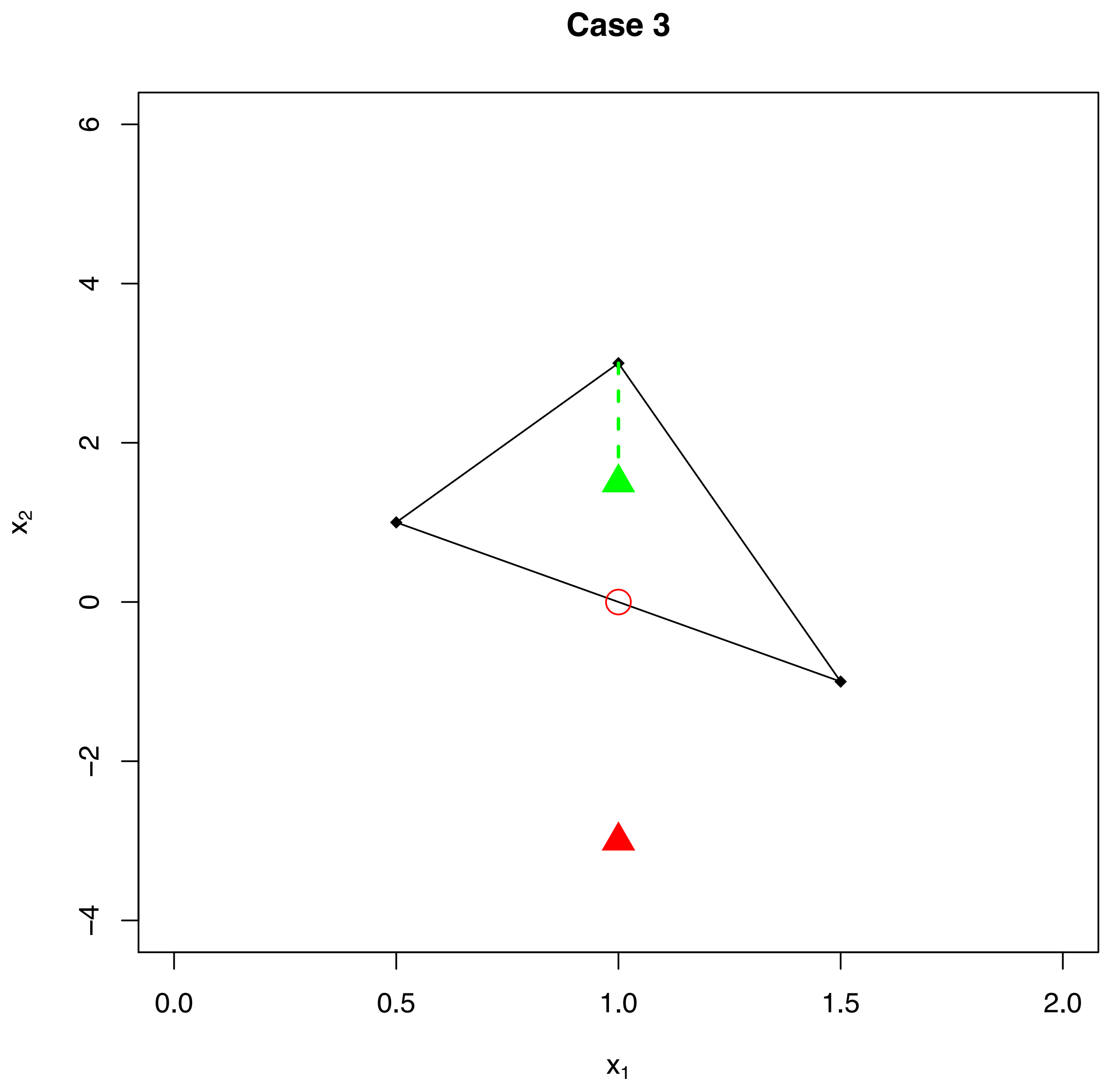
A common choice is $\rho = 0.5$

Case 3

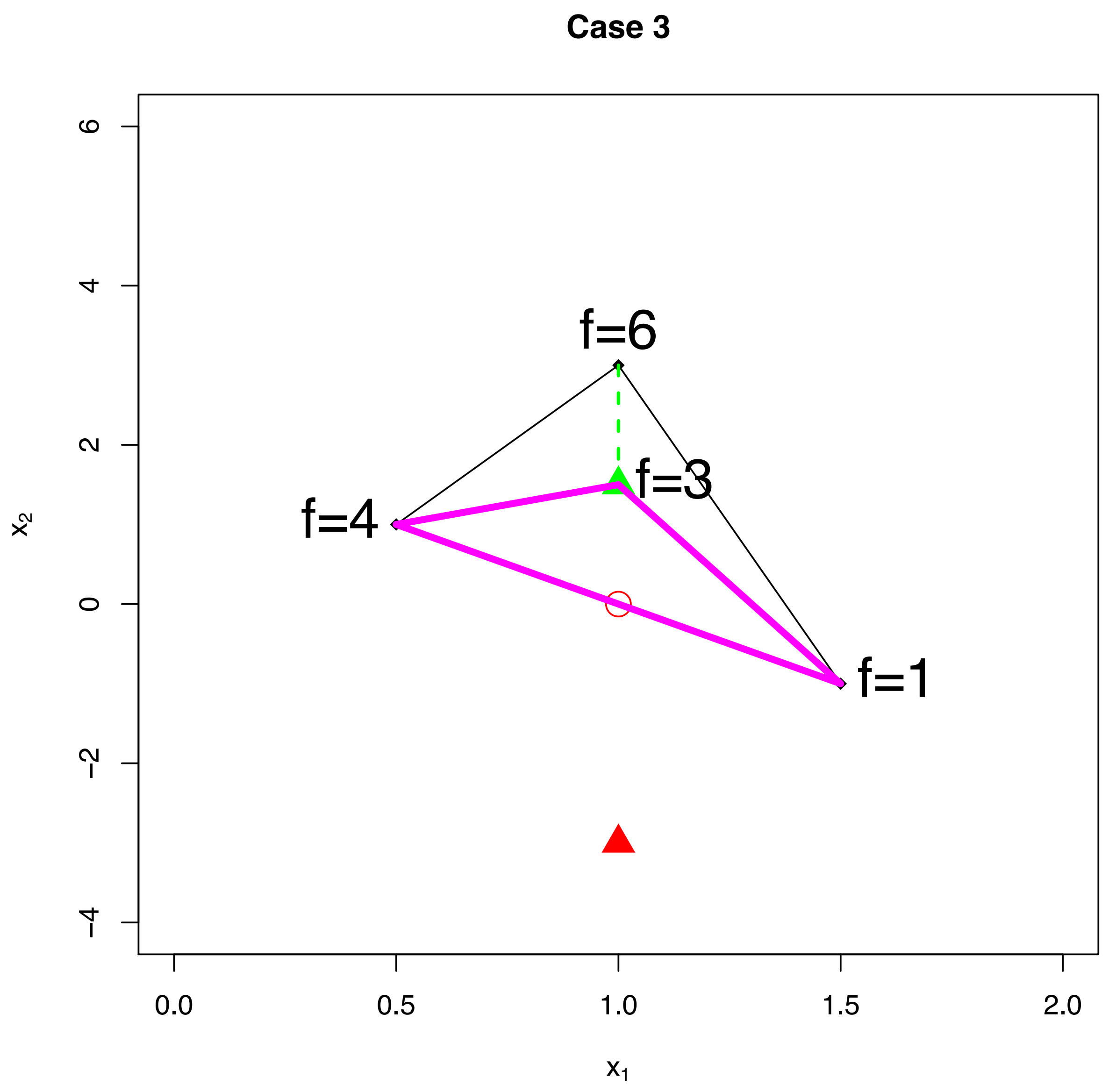


Valley

Find Contraction point



New Simplex



Shrink

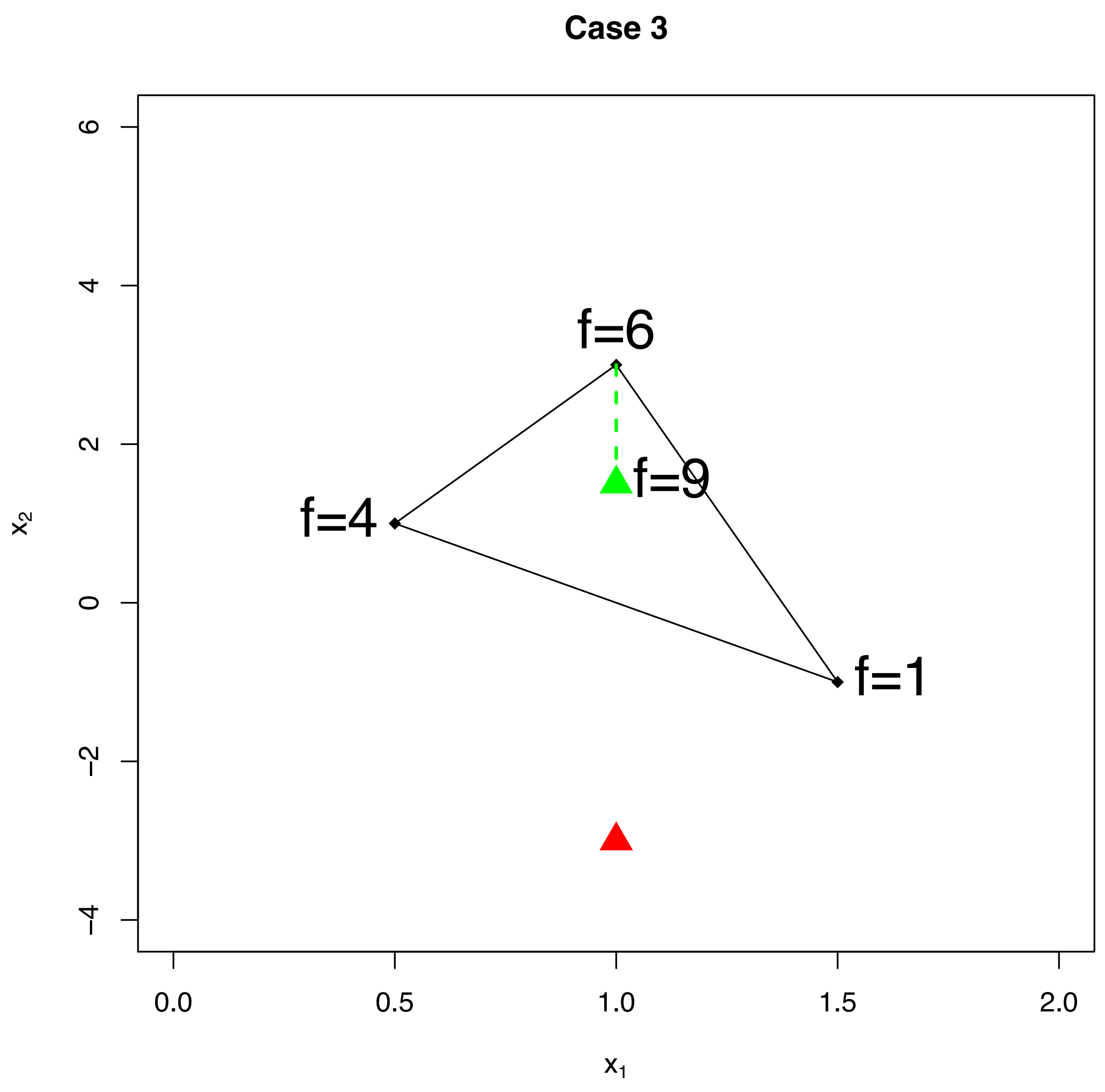
If $f(\mathbf{x}_{n+1}) \leq f(\mathbf{x}_c)$ then contracting away from the worst point does not lead to a better point. In this case the function is too irregular a smaller simplex should be used. Shrink the simplex

$$\mathbf{x}_i = \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1)$$

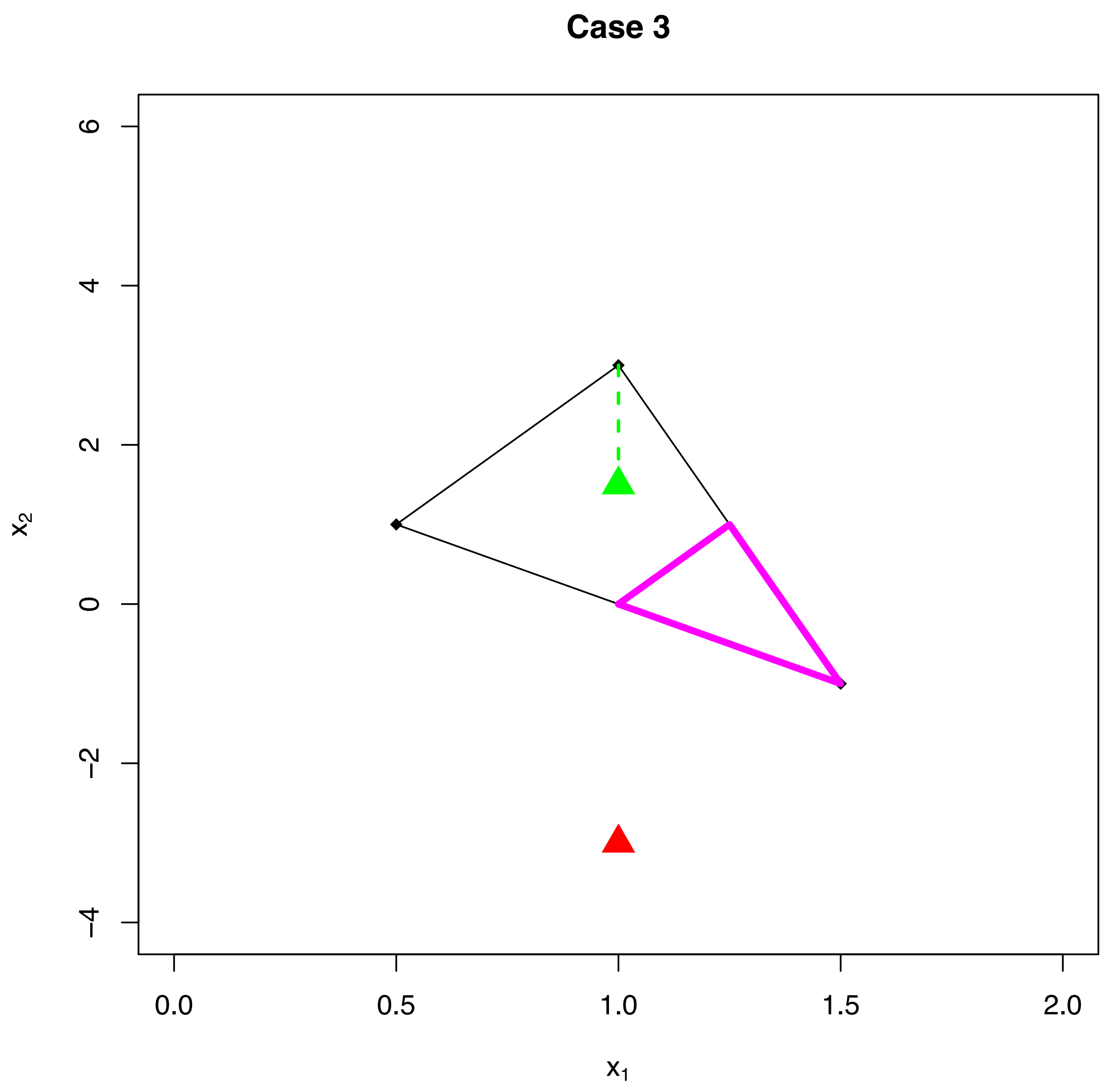
A popular choice is $\sigma = 0.5$.

Egg Carton

Contraction Point is worst



New Simplex



Summary

- Order points
- Find centroid
- Find reflected point
- Three cases:
    1. Case 1 ($f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$): Keep $\mathbf{x}_r$
    2. Case 2 ($f(\mathbf{x}_r) < f(\mathbf{x}_1)$): Find $\mathbf{x}_e$.
        - If $f(\mathbf{x}_e) < f(\mathbf{x}_r)$ then keep $\mathbf{x}_e$
        - Otherwise keep $\mathbf{x}_r$
    3. Case 3 ($f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$): Find $\mathbf{x}_c$

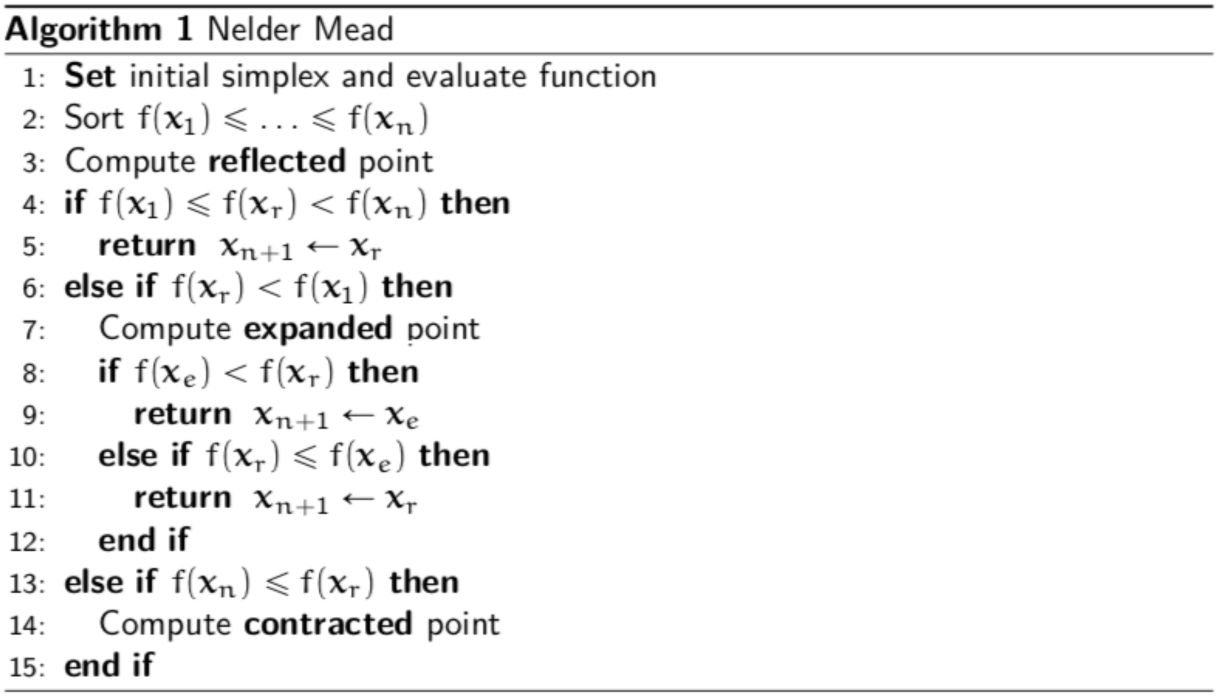
- If $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$ then keep $\mathbf{x}_c$
- Otherwise shrink

Coding Nelder Mead

Your task

- Find the minimum of the function $f(\mathbf{x}) = x_1^2 + x_2^2$
- Use a triangle with vertices $(1, 1)$, $(1, 2)$, $(2, 2)$ as the starting simplex
- Don't worry about using a loop just yet. Try to get code that just does the first iteration.
- Don't worry about the stopping rule yet either

Use pseudo-code



Algorithm 1 Nelder Mead

```
Set initial simplex and evaluate function
Sort $f(x_1) \leqslant \ldots \leqslant f(x_n)$
Compute reflected point
if $f(x_1) \leqslant f(x_r) < f(x_n)$ then
    return $x_{n+1} \leftarrow x_r$
else if $f(x_r) < f(x_1)$ then
    Compute expanded point
    if $f(x_e) < f(x_r)$ then
        return $x_{n+1} \leftarrow x_e$
    else if $f(x_r) \leqslant f(x_e)$ then
        return $x_{n+1} \leftarrow x_r$
    end if
else if $f(x_n) \leqslant f(x_r)$ then
    Compute contracted point
end if
```

Stopping Rule for Nelder Mead

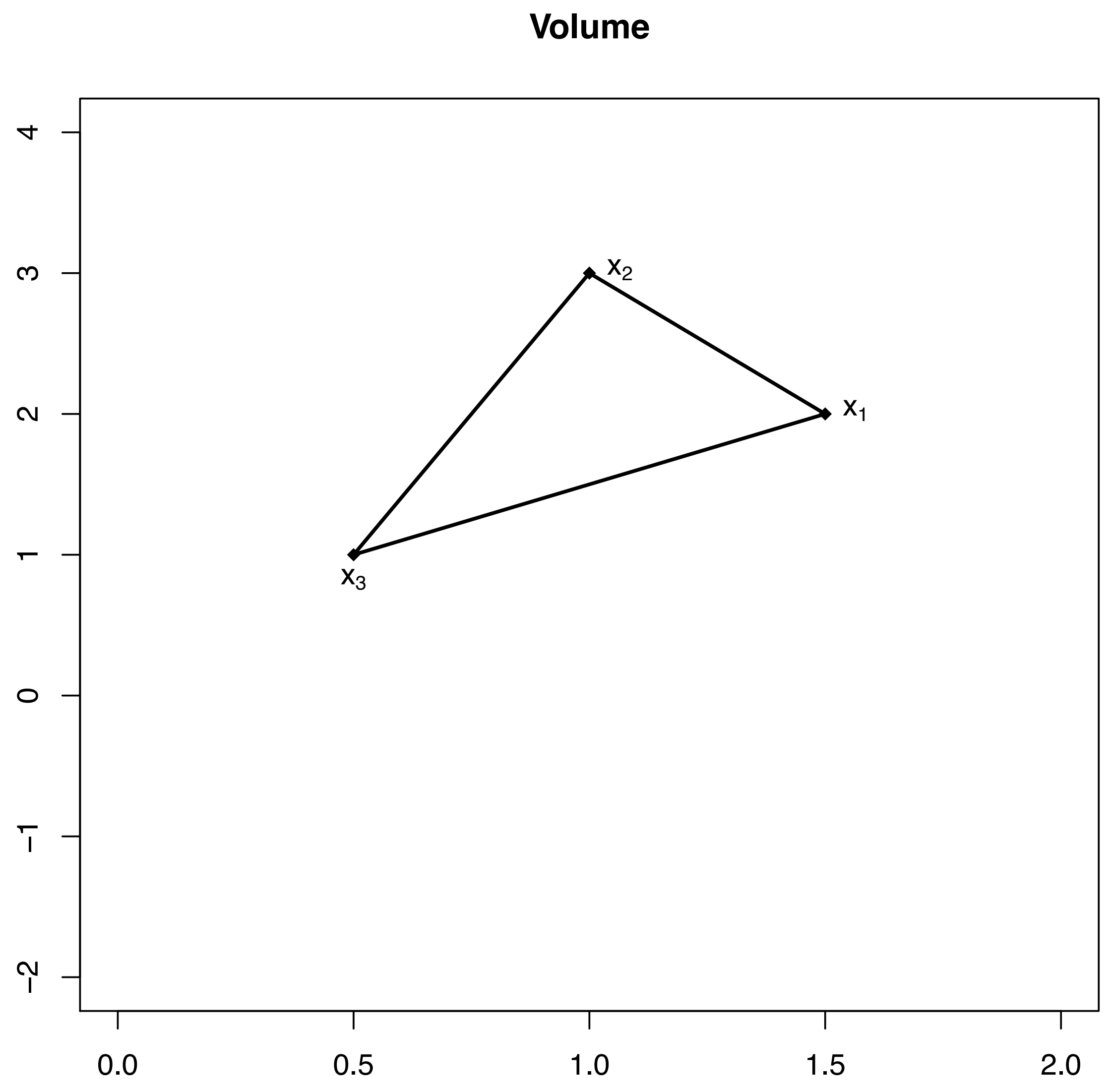
- As Nelder Mead gets close to (or reaches) the minimum, the simplex gets smaller and smaller.
- One way to know that Nelder Mead has converged is by looking at the volume of the simplex.
- To work out the volume requires some understanding between the relationship between matrix algebra and geometry.

Stopping Rule for Nelder Mead

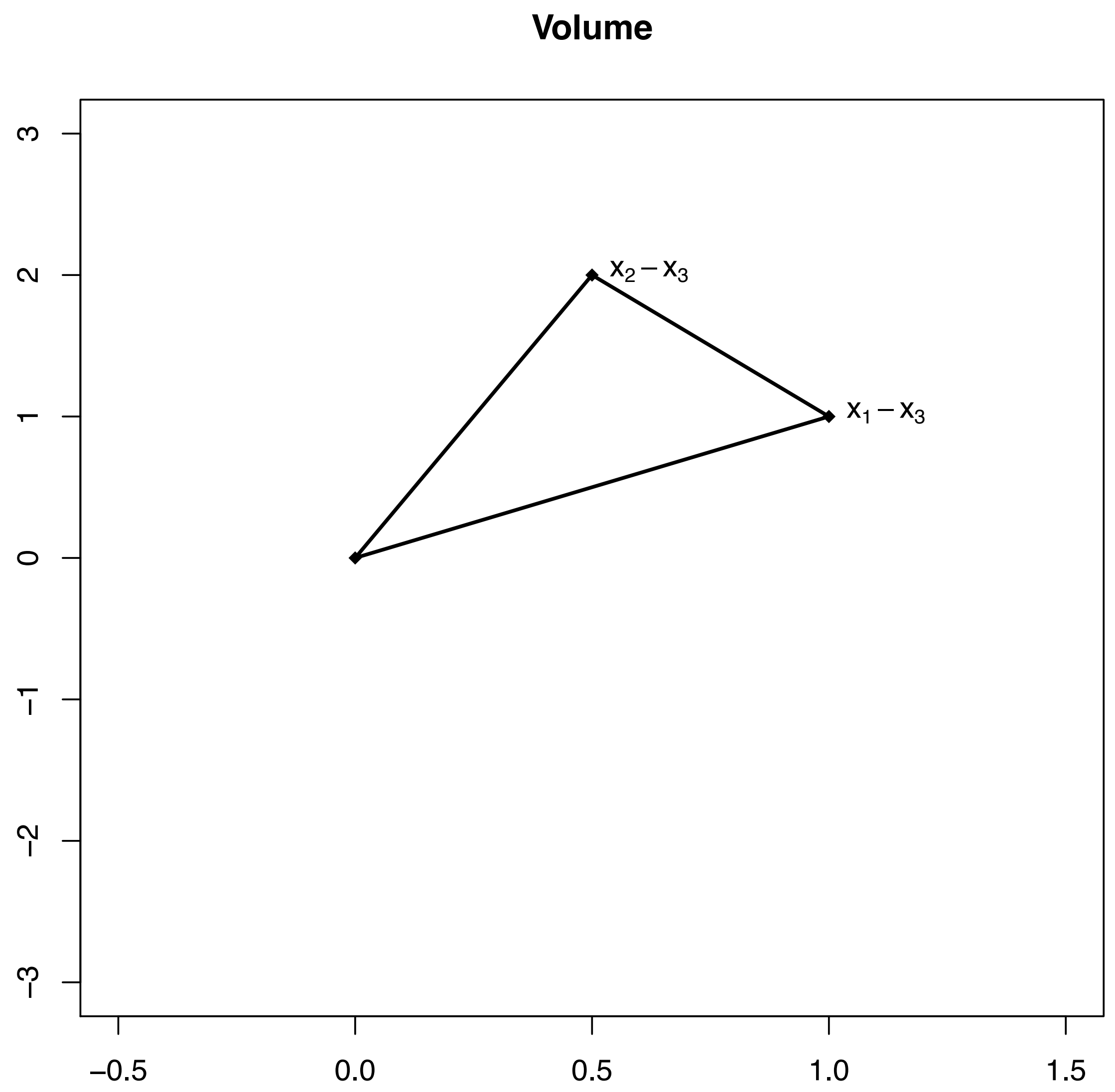
- Choose the vertex $\mathbf{x}_{n+1}$ (although choosing any other vertex will also work)

- Build the matrix $\tilde{X} = (\mathbf{x}_1 - \mathbf{x}_{n+1}, \mathbf{x}_2 - \mathbf{x}_{n+1}, \ldots, \mathbf{x}_n - \mathbf{x}_{n+1})$
- The volume of the simplex is $\frac{1}{2}|det(\tilde{X})|$

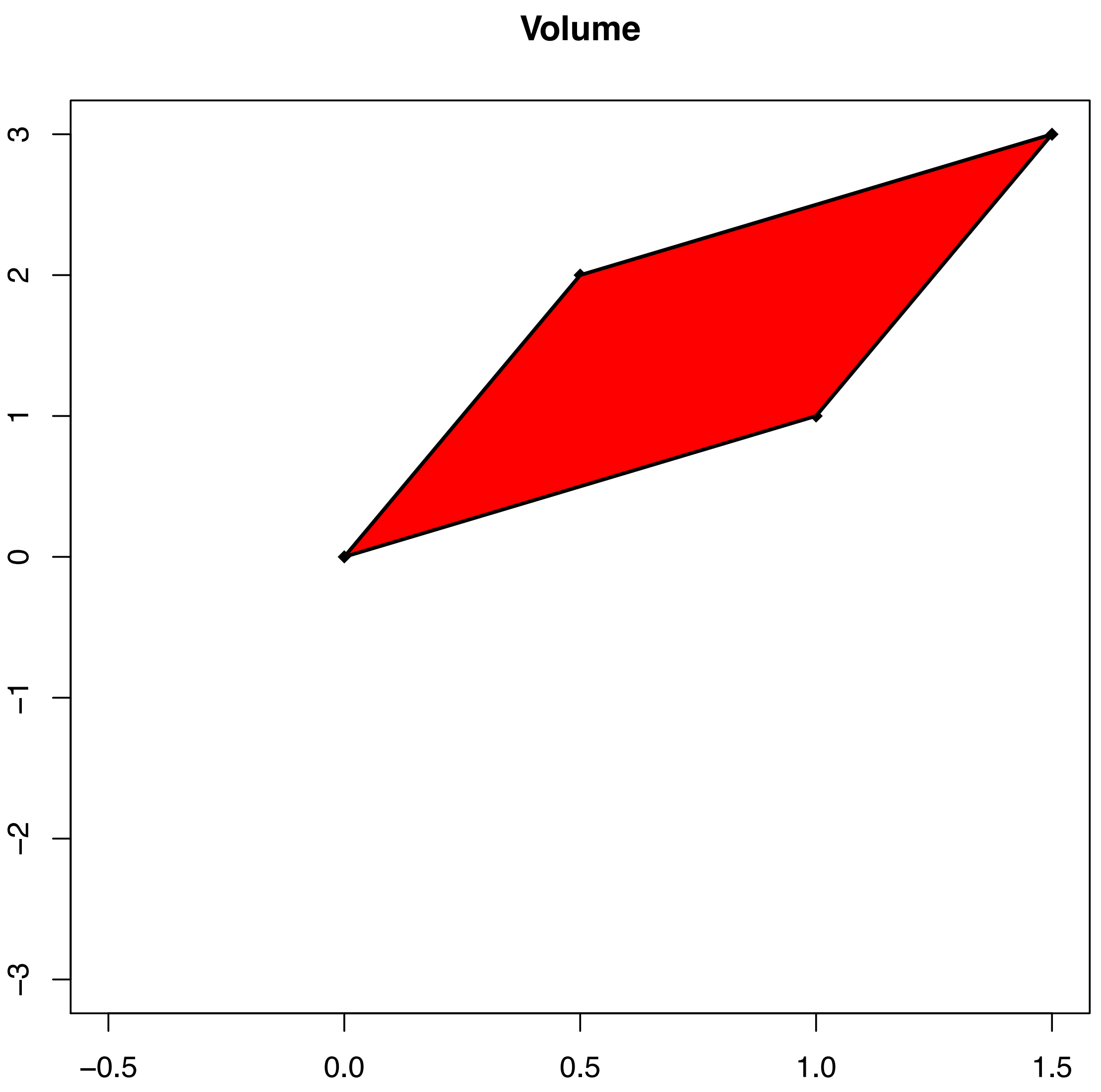
Why?



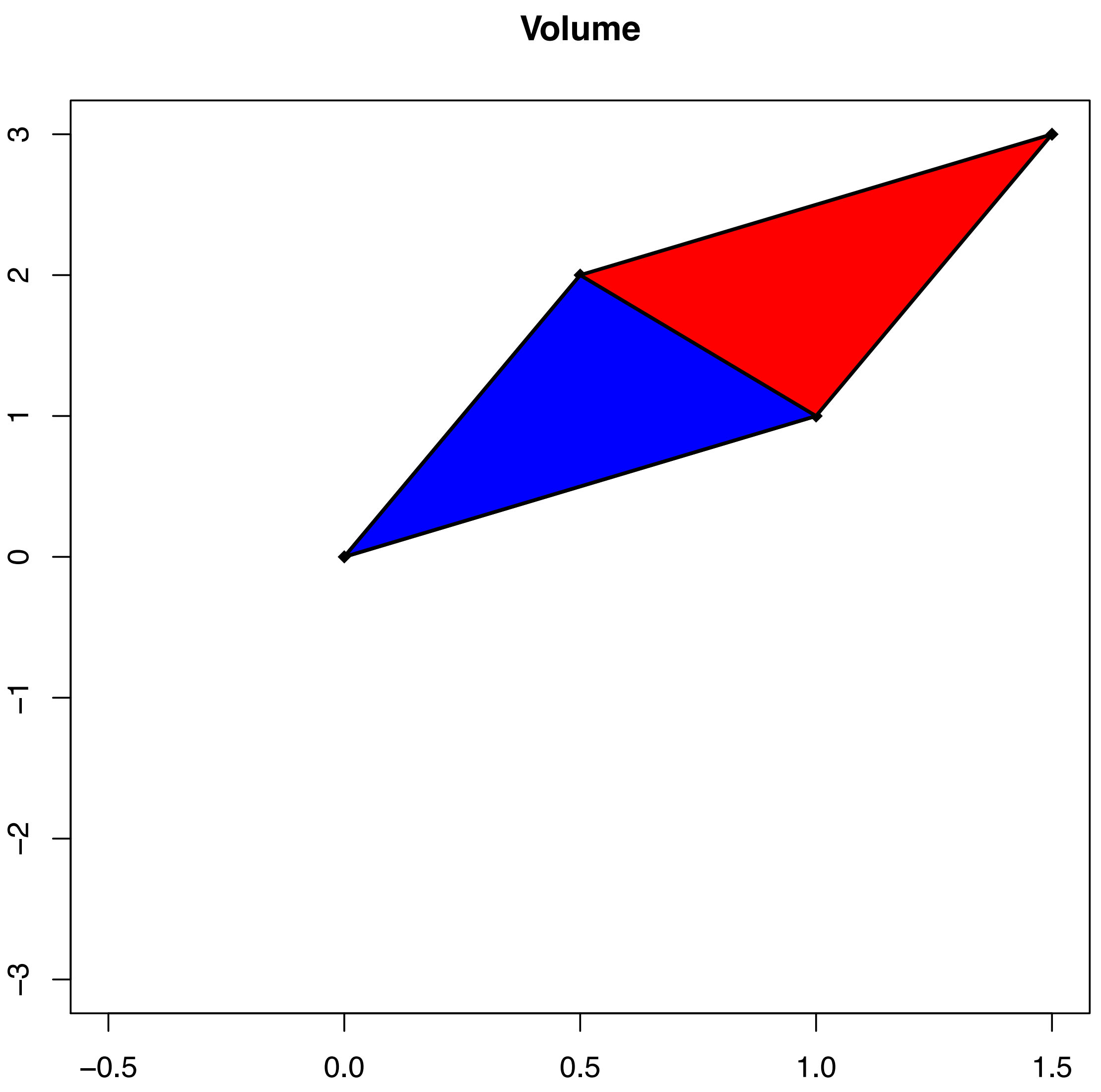
Traslate



Determinant=Area of Trapezoid



Triangle=Half Trapezoid



Alternative formula

Some of you may have learnt the formula for the area of a triangle as:

$$\frac{1}{2}\left|det\begin{pmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3\\ 1 & 1 & 1\end{pmatrix}\right|$$

The two approaches are equivalent.

Nelder Mead in `optim()`

- Nelder Mead is the default algorithm in the R function `optim()`
- It is generally slower than Newton and Quasi-Newton methods but is more stable for functions that are not smooth.
- Including the argument `control=list(trace, REPORT=1)` will print out details about each step of the algorithm.
- Slight different terminology is used for example 'expansion' is called 'extension'

Summary

- This is the end of the optimization topic.
- You should now be familiar with
  - Newton's Method
  - Quasi Newton Method
  - Nelder Mead
- Hopefully you also improved your coding skills!

Summary

Some important lessons:

- If you can evaluate derivatives and Hessians then do so when implementing Newton and Quasi-Newton methods.
- If there are discontinuities in the function then Nelder Mead may work better.
- In any case the best strategy is to optimize using more than one method to check that results are robust.
- Also pay special attention to starting values. A good strategy is to check that results are robust to a few different choices of starting values.

Statistical Computing

Lecture 9: Eigenanalysis

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Why numerical linear algebra?

- Difference between linear algebra and **applied** linear algebra.
- Think about linear algebra in Statistics.
- In curve fitting.
- In image processing.
- In signal processing.
- etc.

We need to know numerical techniques!

Eigenanalysis

- In general, a matrix acts on a vector by changing both its magnitude and its direction.
- However, a matrix may act on certain vectors by changing only their magnitude, and leaving their direction unchanged (or reversing it).
  - these vectors are the eigenvectors of the matrix
  - the scaling factor is the eigenvalue.

Example

- Think about the linear transformation $\mathbf{y} = \mathbf{A}\mathbf{x}$, where

$$\mathbf{A} = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix},\ \mathbf{x} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix},\ \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}.$$

- What is its geometric interpretation?
- What happens when $\theta = \frac{\pi}{4}$?
- When $\theta = \frac{\pi}{2}$?
- When $\theta = 0$?

Example

- What if $\mathbf{A}$ is not diagnal?
- How to find its eigenvalues and eigenvectors?
- If $\mathbf{A}$ is a real symmetric matrix
  - Only real eigenvalues
  - n distinct linearly independent eigenvectors
  - pairwise orthogonal
  - $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{\mathbf{T}}$

- When A is diagonalisable?
- If a diagonalisation doesn't exist, there is always a triangularisation via Schur Decomposition: $\mathbf{A} = \mathbf{QSQ^T}$.

- Let us revisit together the properties of eigenvalues and eigenvectors.

What does eigenanalysis help to do?

- Let's try to understand what $\mathbf{Ax} = \lambda\mathbf{x}$ is really asking.
- Can we find a pair of λ and $\mathbf{x}$ such that when a matrix $\mathbf{A}$ is applied to $\mathbf{x}$, it doesn't change the direction and just scales the vector?
- If we can find such a pair, then everytime we do something with $\mathbf{Ax}$ in some mathematical operation, we can replace it with $\lambda\mathbf{x}$.

What does eigenanalysis help to do?

- Consider $\mathbf{A} = \begin{pmatrix} 5 & -1 \\ -2 & 4 \end{pmatrix}$ and $\mathbf{x} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$.
  - what if we now want to calculate $\mathbf{A^{20}x}$?
  - what if we now want to calculate $\mathbf{A^{-1}x}$?
- Computationally, we would rather work with scalars than matrices and this is what eigenanalysis helps us do.
- But what if we are not lucky enough to be asked to multiply a matrix by one of its eigenvectors?

Advantages of eigenanalysis

- It enables us to replace a matrix with a scalar when we have the opportunity to change our coordinate system to the eigenvectors of a matrix.
- We can express any vector in terms of this new coordinate system.
- We can use the fact that $\mathbf{Ax} = \lambda\mathbf{x}$ to simplify calculations.

Application of Eigenanalysis: Google

How to know page rank?

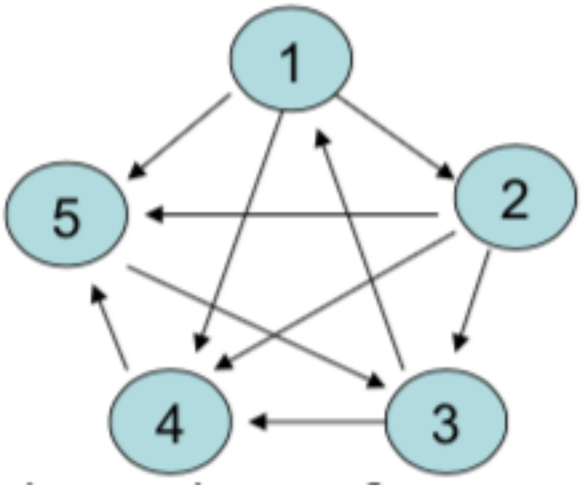
- How does the search engine know which pages are the most important?
- Google assigns a number to each individual page, expressing its importance.
- This number is known as the PageRank and is computed via the eigenvalue problem $Aw = \lambda w$, where A is based on the link structure of the Internet.

Google's pagerank

- Suppose we have a set of webpages W, with $|W| = n$ as the number of webpages.
- We define a connectivity (adjacency) matrix A as follows: $A_{i,j}$ is 1 if there is a hyperlink from page i to page j and 0 otherwise.
- Typically A is huge (27.5 billion x 27.5 billion in 2011), but extremely sparse (lots of zero values)

A small example

- Consider the small set of five webpages.



- The connectivity matrix is

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A small example

- The ranking of page i is proportional to the sum of the rankings of all pages that link to i: $r_4 = \alpha(r_1 + r_2 + r_3)$.
- So this is a system of $n = 5$ linear equations:

$$r = \alpha A^T r, \text{ or } A^T r = (1/\alpha)r.$$

- The ranking vector is treated like a probability of relevance, so we need to then rescale so that $\Sigma_{i=1}^n r_i = 1$.
- Go to R and compute pageranks.

What you need to realise for now

- Finding the eigenvalues and eigenvectors of a massive matrix is computationally challenging though (don't try to solve the characteristic polynomial!)
- and you will learn numerical techniques later.

Statistical Computing

Lecture 10: Singular Value Decomposition

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Non-square matrices

- Recall that if the matrix A is square (real or complex) then a diagonalisation may exist.
  - This is clearly very useful for easy calculation of many important problems as we saw last week.
  - If a diagonalisation doesn't exist, then there is always a triangularisation via Schur Decomposition.
- But non-square matrices don't have eigenvalues, so what can we do?
- You are about to learn the most useful diagonal decomposition that works for all matrices: Singular Value Decomposition.

Singular values

- Singular values are the square roots of the eigenvalues of A^TA which is square and symmetric
- The singular vectors (u and v) come in a pair for each singular value σ, such that

$$Av = \sigma u.$$

Generalising Eigen-Decomposition

- Eigendecomposition involves only one eigenvector for each eigenvalue (including multiplicities), stored in an orthogonal matrix Q, with eigenvalues on the diagonal of the matrix Λ, so that $A = Q\lambda Q^T$.
- We can generalise this now that we have singular vectors u and v for each singular value σ.

Singular Value Decomposition (SVD)

For $A \in \mathcal{R}^{m\times n}$, there exists orthogonal matrices

$$U = [u_1, \cdots, u_m] \in \mathcal{R}^{m\times m}$$

and

$$V = [v_1, \cdots, v_n] \in \mathcal{R}^{n\times n}$$

such that

$$U^TAV = \Sigma = \text{diag}\{\sigma_1, \cdots, \sigma_p\} \in \mathcal{R}^{m\times n},$$

with $p = \min\{m, n\}$ and $\sigma_1 \geq \cdots \geq \sigma_p \geq 0$.

Rearranging, we have

$$A = U\Sigma V^T$$

.

SVD

Try `svd()` in R.

Some properties of SVD

- σ_i are singular values of A.
- The non-zero singular values of A are the square roots of the non-zero eigenvalues of both $A^T A$ and AA^T.
- The rank of a matrix is equal to the number of non-zero singular values.
- The condition number measures the degree of singularity of $A^T A$:

$$\kappa = \frac{\text{max singular value}}{\text{min singular value}}.$$

Summary

- SVD: Decomposition of any matrix A.
- It works by eigendecomposition of $A^T A$ (or AA^T) which is square and symmetric.
- We are now able to associate an orthogonal diagonal form with every matrix, and easily calculate useful properties of the matrix.
- Over the next few lectures we will look at the fantastic applications of SVD.

Lab session

Peek into SVD and PCA in R, illustrate their relationship and write a short report.

Statistical Computing

Lecture 11: Basic Applications of SVD

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Applications of SVD

- In data compression, we start with a matrix A that contains perfect data, and we try to find a (lower-rank) approximation to the data that seeks to capture the principal elements of the data.
  - we sacrifice only the less important parts that don't degrade data quality too much, in order to gain compression.
- In noise filtering, we start with a matrix A that contains imperfect data, and we try to find a (lower-rank) approximation to the data that seeks to capture the principal elements of the data.
  - we give up only the less important parts that are typically noise.
- So both these tasks are related, and rely on the SVD to find a suitable lower-ranked approximation to a matrix

Matrix Approximation

- Recall that the SVD of a matrix A in $\mathcal{R}_{m\times n}$ decomposes the matrix into

$$A = U\Sigma V^T = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} D & \\ & 0 \end{pmatrix} \begin{pmatrix} V_1^T & V_2^T \end{pmatrix} = U_1^T D V_1^T$$

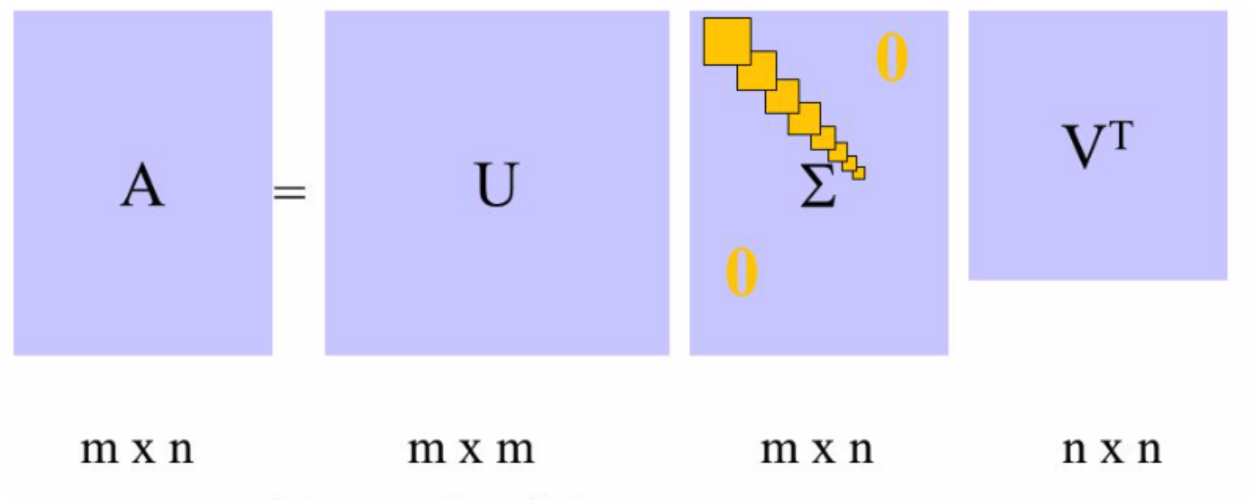
  where $D = \text{diag}\{\alpha_1, \cdots, \alpha_r\} \in \mathcal{R}^{r\times r}$ with $\sigma_r > 0$ for $r = \text{rank}(A)$.

- A can also be expressed as a sum of outer products, with each sum being a rank 1 matrix of dimension $m \times n$

$$A = \Sigma_{i=1}^r \sigma_i u_u v_i^T = \alpha_1 u_1 v_1^T + \cdots + \alpha_r u_r v_r^T.$$

- We can truncate this sum when we feel that the singular values are so small that they are not contributing much.

SVD components



Matrix Approximation Error

- So if we truncate the sum after k singular values, then we are approximating A with A_k

$$A_k = \Sigma_{i=1}^{k\leq r} \sigma_i u_u v_i^T = \alpha_1 u_1 v_1^T + \cdots + \alpha_k u_k v_k^T.$$

- The error in the approximation is σ_{k+1}.
- We have $\text{rank}(A_k) = k < r = \text{rank}(A)$.
- By approximating A with A_k, we have saved memory:
  - A requires us to store $m \times n$ numbers
  - A_k requires us to store ??? numbers?

– Significant savings for large matrices, and/or small k.

Image Compression

Image Compression

- Suppose we have a grayscale image (128×128 pixels).
  - We can use a matrix to represent this image.
- If we have a colour image, it has three matrices, with the same size as the image. Each matrix represents a color value that comprises the RGB color scale.
- Each pixel is represented by an integer from 0-255.
- Next, we can decompose the matrix by SVD.
- By eliminating small singular values, we can approximate the matrix.
  - Choose the value of k for the low-rank approximation A_k.
  - Plotting the singular values might help identify where there is a noticeable drop in significance

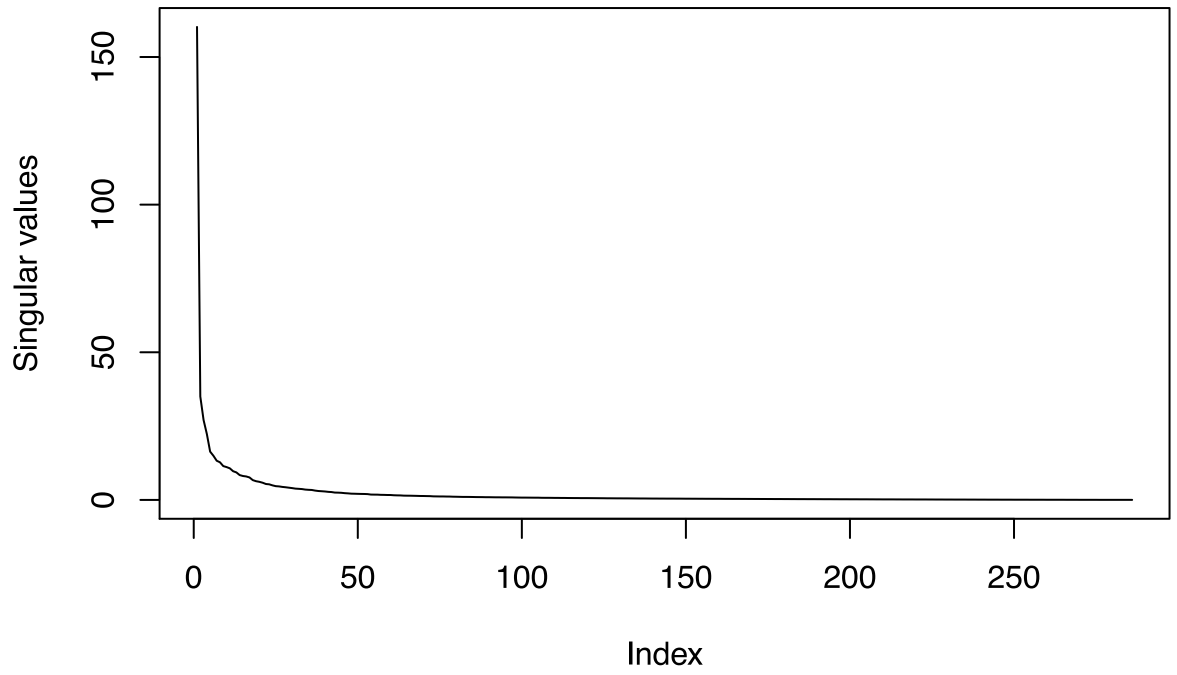
Reconstructing approximated image

- Suppose we have chosen the value of k = number of singular values we wish to retain. - We can generate a new image matrix by expanding A using the SVD (first k singular values only).
- If you want to use colour images, do it for R, G, B matrices separately and then reassemble.

Data compression in R

Here are four images with rank 286, 200, 116 and 32. How many numbers to store for each of them?

Plot of singular values



Lab session

- Like in today's lecture, take a high resolution image of yourself, and produce a sequence of low rank approximations.
- How much can you save?

Noise reduction

Noise reduction

- The SVD also has applications in digital signal processing
- The central idea is to let a matrix A represent the noisy signal, compute the SVD, and then discard small singular values of A.

- It can be shown that the small singular values mainly represent the noise, and thus the rank k matrix A_k represents a filtered signal with less noise.

Application to Signal Separation

- Suppose we have P observed signals $m_i(t)$ which are linear combinations of r source signals $s_i(t)$, corrupted by noise signals $n_i(t)$. We have a time series of data from T time periods
- This can be written as:

$$m_i(t) = \alpha_{i1}s_1(t) + \cdots + a_{ir}s_r(t) + n_i(t),$$

  where $t = 1, \cdots, T$ and $i = 1, \cdots, P$.
- Write the equivalent matrix representation.

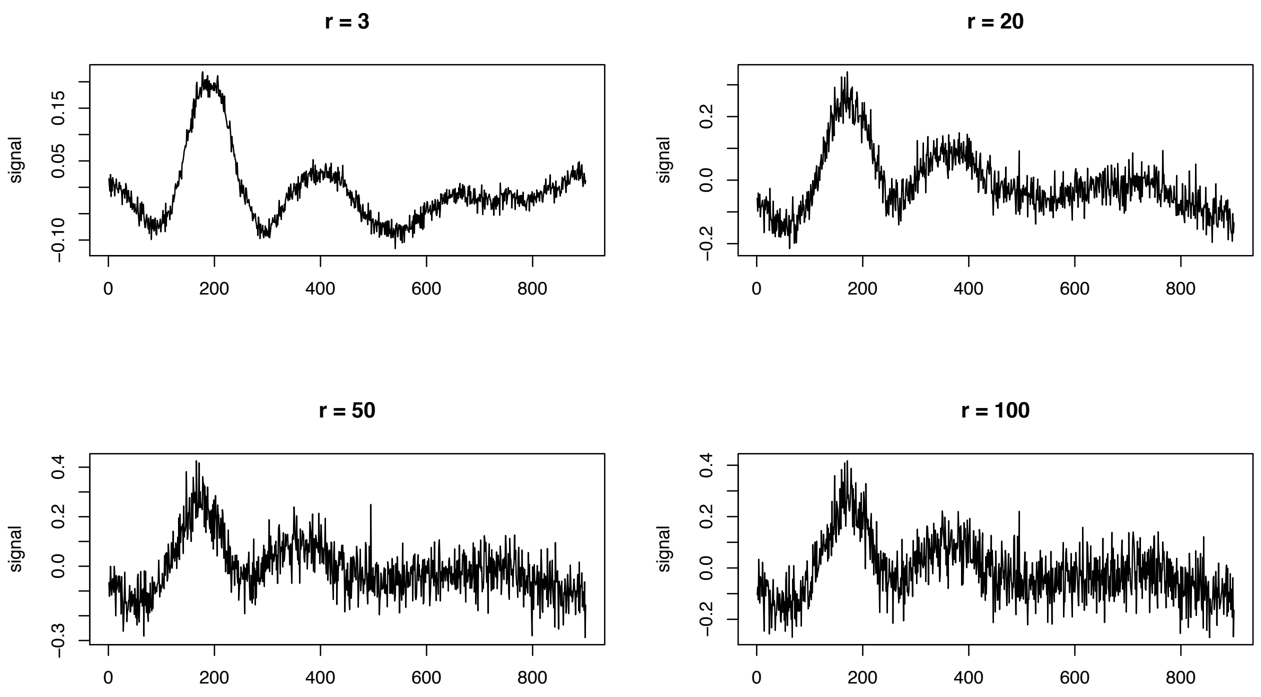
SVD of signal matrix M

- We can decompose M using SVD: $M = U\Sigma V^T$.
- If the signal compared to the noise is sufficiently strong, we are likely to find clear distinction between the singular values due to the signal and those due to noise

$$U = (U_s, U_n),\ \Sigma = \begin{pmatrix} \Sigma_s & \\ & \Sigma_n \end{pmatrix},\ V^T = \begin{pmatrix} V_s \\ V_n \end{pmatrix}$$

Results of noise reduction

```r
library(bootSVD)
set.seed(1)
Y <- simEEG(n = 100, centered = TRUE, propVarNoise = 0.3, wide = TRUE)
svdY <- svd(Y)
par(mfrow = c(2, 2))
for (j in c(3, 20, 50, 100)) {
    a = svdY$u[, 1:j] %*% diag(svdY$d[1:j]) %*% t(svdY$v[, 1:j])
    plot(a[1, ], type = "l", xlab = "", ylab = "signal", main = paste("r = ",
        j, sep = ""))
}
```



Curve fitting

Curve fitting

- Curve fitting seeks to approximate the data by fitting a curve that minimises the sum of the squared residual errors.
- Over-determined linear system.
- No exact solutions.
- We find an approximation.

Linear regression

For a linear regression,

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon, \ \epsilon \sim N(0, \Sigma).$$

what is its OLS solution?

Numerical Issues

- Numerically, there can be problems with solving for β via the normal equations.
  - The inverse of $X^T X$ might not exist
  - It is not computationally efficient to invert it for large dimensions
  - $X^T X$ maybe ill-conditioned
- The SVD can help us by providing a numerically stable pseudo-inverse, and the opportunity to identify (through the singular values) any ill-conditioning.

Pseudo-inverse

Recall that the SVD of a matrix A in $\mathcal{R}_{m\times n}$ decomposes the matrix into

$$A = U\Sigma V^T = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} D & \\ & 0 \end{pmatrix} \begin{pmatrix} V_1^T & V_2^T \end{pmatrix} = U_1 D V_1^T$$

where $D = \text{diag}\{\alpha_1, \cdots, \alpha_r\} \in \mathcal{R}^{r\times r}$ with $\sigma_r > 0$ for $r = \text{rank}(A)$.

- The pseudo-inverse, or generalised inverse, of A can always be calculated, even if A is not invertible.

$$A^+ = V_1 D^{-1} U_1^T$$

- A^+ is $n \times m$.

Linear regression

For

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon,\ \epsilon \sim N(0, \Sigma),$$

the solution is

$$\beta = X^+ y = V_1 D^{-1} U_1^T y = \Sigma_{i=1}^r \frac{v_i u_i^T y}{\sigma_i}.$$

A small singular value means trouble. A lower rank approximation should be more reliable.

An example of using generalised inverse

```
X <- matrix(c(1, 2, 4, 8, 3, 6, 9, 12, -11, -22, -32, -40), 4,
    3)
y <- 2 * X[, 1] - 7 * X[, 2] + 1e-04 * (rnorm(4) - 0.5)
library(MASS)
beta <- ginv(X) %*% y  # write your own function to find generalised inverse
t(y - X %*% beta) %*% (y - X %*% beta)
```

```
##              [,1]
## [1,] 3.845924e-08
```

Lab session

- Use generalised inverse to solve a linear regression problem, and compare with the `lm()` function in R.

Statistical Computing

Lecture 12: Numerical Algorithms for Eigenanalysis

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Finding eigenvalues and eigenvectors

- So far, we have only considered the characteristic polynomial approach to find the eigenvalues of a matrix
- Once we have the eigenvalues, we have been solving the homogeneous equation to find the corresponding eigenvectors
- The process is: find the eigenvalues first, and then find the corresponding eigenvectors

Practicalities

- Unfortunately, this is an impractical approach for $n > 4$
- We will bypass the characteristic polynomial and now take a different approach
  - The Power Method
  - QR decomposition

The Power Method

The Power Method

- The Power Method finds the dominant eigenvalue λ_1 and corresponding dominant eigenvector v_1 of a matrix A
- The dominant eigenvalue is the one with the largest modulus (absolute value for real eigenvalues)
- The Power Method is an iterative approach that generates a sequence of scalars that converge to λ_1 and a sequence of vectors that converge to v_1
- The Power Method works well (converges quickly) when the dominant eigenvalue is clearly dominant

The Power Method

- It works by starting with an initial vector x_0, transforming to $x_1 = Ax_0$, transforming x_1 to $x_2 = Ax_1$, etc.

$$x_1 = Ax_0;\ x_2 = Ax_1 = A^2x_0; \cdots ; x_k = A^k x_0.$$

- As $k \to \infty$, $x_k \to v_1$.

Proof of convergence

- Assume λ_1 is the dominant eigenvalue.
- $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$.
- Assume there are n independent eigenvectors $v_1, \cdots , v_n$.
- $x_0 = c_1v_1 + \cdots + c_nv_n$.

- $x_k = ?$

Estimation of eigenvalues

So once we have an eigenvector estimate we can quickly estimate the corresponding eigenvalue

$$Ax = \lambda x \Rightarrow x^T A x = \lambda x^T x \Rightarrow \lambda = \frac{x^T A x}{x^T x} = \frac{x^T A x}{||x||^2} = q^T A q.$$

Practical Power method (normalised)

Since the components of x_k just get larger and larger as the Power Method iterates, and we really just want to know the direction of v_1 (not its magnitude), we normalise each x_k.

So the Power Method can be summarised as:

- Set initial vector $q_0 = x_0/(||x_0||)$.
- Repeat for k
  - Compute $x_k = Aq_{k-1}$
  - Normalise $q_k = x_k/||x_k||$
  - Estimate $\lambda_k = q_k^T A q_k$

Comments

- The Power method is not expected to converge if the matrix A is not diagonalisable
- Convergence rate depends on how dominant λ_1 is
- Google uses it to calculate the PageRank and Twitter uses it to show users recommendations of who to follow.
- And for non-dominant eigenvalues/vectors?

Lab session

Now use The Power Method to redo your google pagerank problem

QR decomposition

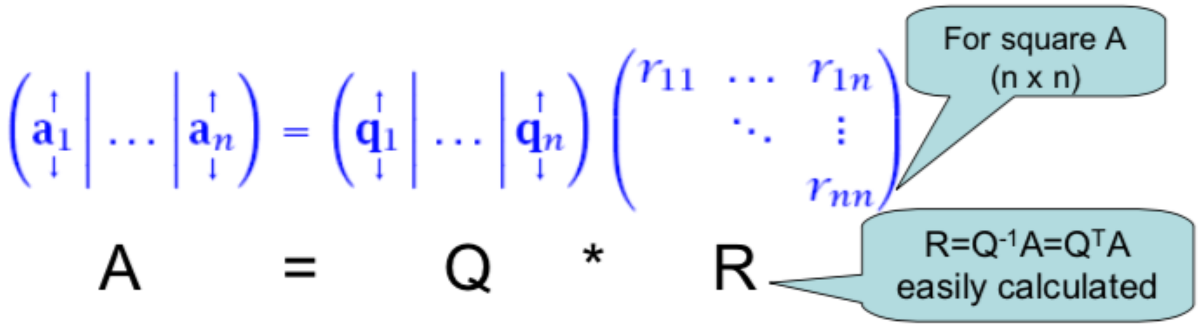
Eigenvalue Revealing Decomposition

- It would be nice if we could get our matrix A into an **eigenvalue revealing decomposition** like Schur decomposition $A = QSQ^T$
- So we can read off the eigenvalues (all of them) from the diagonal
- We will do it iteratively using QR decomposition: $A = QR$
- QR decomposition is not an eigenvalue revealing decomposition, but it will help us with our aim
- QR decomposition can be done with Gram-Schmidt Orthogonalisation (GSO) algorithm

GSO algorithm

Any set of basis vectors $(a_1, a_2, \cdots, a_n)$ can be transformed to an orthonormal basis $(q_1, q_2, \cdots, q_n)$ by:

GSO $\to$ QR



QR

For any $m \times n$ matrix A, we can express A as $A = QR$.

- Q is $m \times m$, orthogonal
- R is $m \times n$, upper triangular

For (non-square) tall matrices A with $m > n$, the last $(m - n)$ rows of R are all zero, so we can express A as:

$$A = QR = (Q_1, Q_2) \begin{pmatrix} R_1 \\ \mathbf{0} \end{pmatrix} = Q_1 R_1.$$

QR algorithm

This algorithm computes an upper triangular matrix S and a unitary matrix Q such that $A = QSQ^T$ is the Schur decomposition of A.

1. Set $A_0 := A$.
2. for $k = 1, 2, \cdots$,
    - $A_{k-1} = Q_{k-1} R_{k-1}$.
    - $A_k := R_{k-1} Q_{k-1}$.
3. Set $S := A_\infty$.

Lab session

- Go to R code up QR algorithm.
- Use QR algorithm on $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.

Solution to $Ax = b$ by QR

If we have $A = QR$ or (even better) the economy form $A = Q_1 R_1$, then the linear system $Ax = b$ can be easily solved:

$$Ax = b$$

$$(Q_1 R_1)x = b$$

$$R_1 x = Q_1^T b$$

and x is found through back substitution.

Computation of the SVD by QR

Just as we can use $A = QR$ to avoid calculating $A^T A$ in the normal equations, we can also use QR decomposition to solve the eigenvalue problems for $A^T A$ and AA^T to obtain the SVD of A.

Lab session

- Use QR decomposition to write your own `svd` function in R.
- For linear regression, compare svd, pseudo-inverse and QR decomposition in R.

Statistical Computing

Lecture 13: Image Recognition Based on SVD

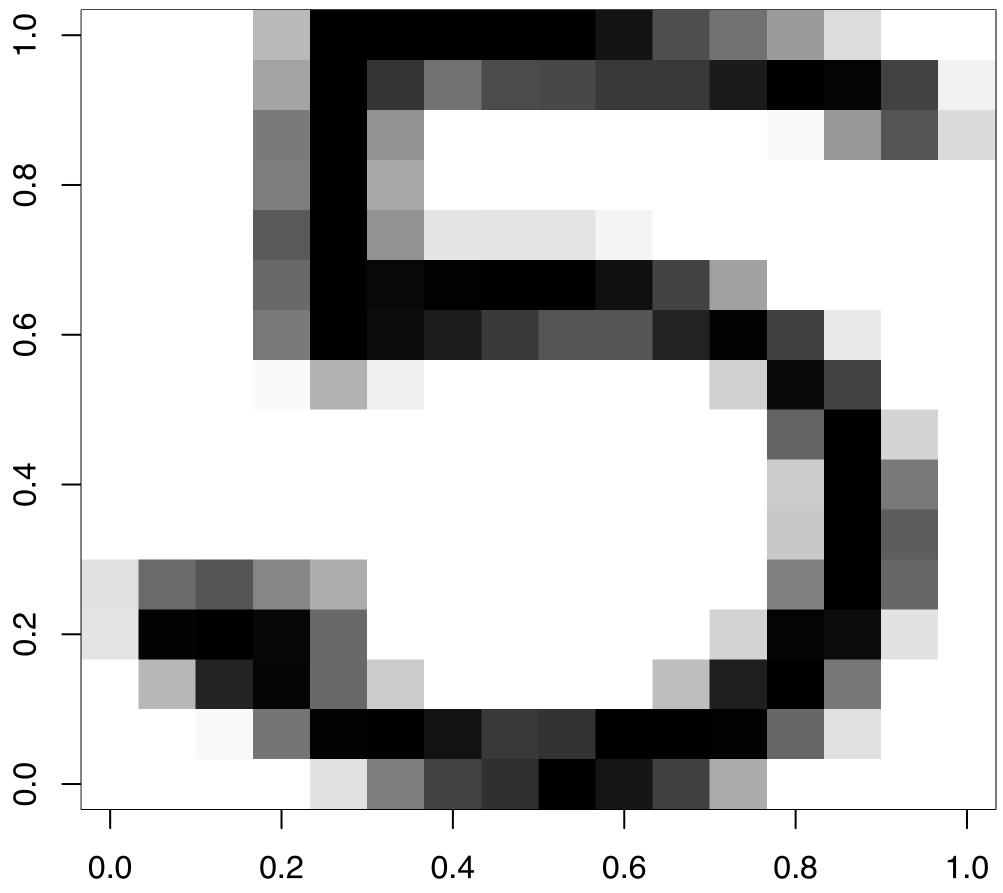
Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

Classification of handwritten digits

Read images in R

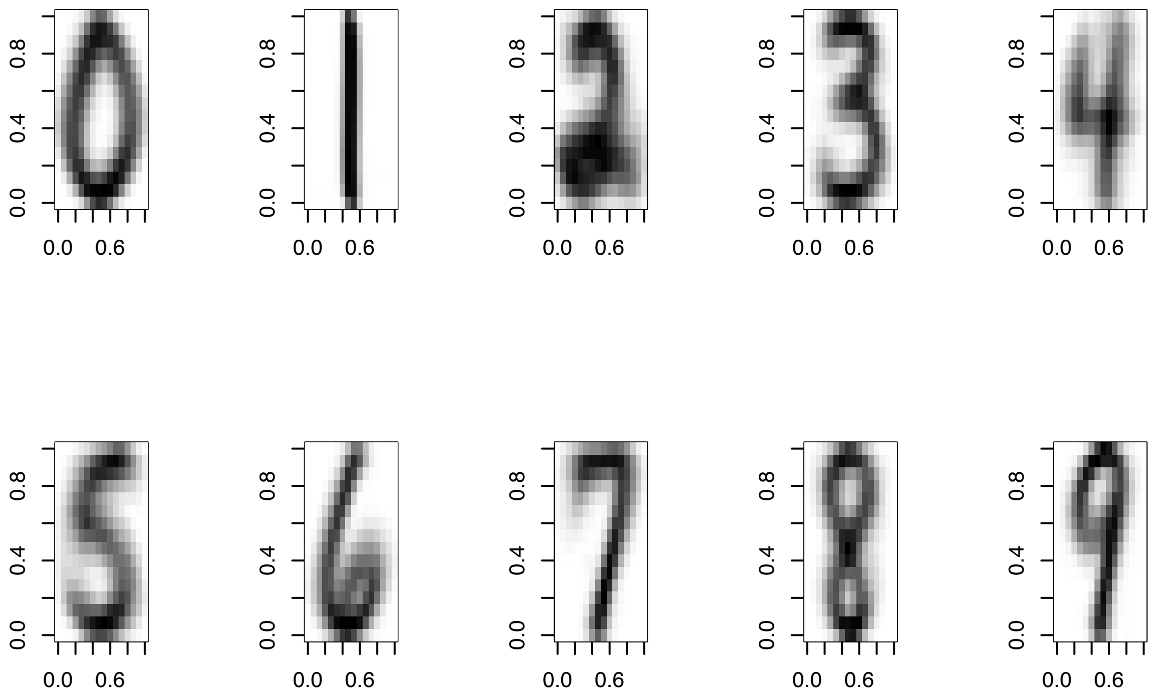
```
## The image matrix for training sample. 256x1707
azip <- read.table("azip.dat")

## The true digits given in the training sample. length = 1707
dzip <- as.numeric(read.table("dzip.dat"))

## The testing image matrix. 256x2007
testzip <- read.table("testzip.dat")

## The true digits for the testing sample. length = 2007
dtest <- read.table("dtest.dat")

## Display the image
i <- 120
image(matrix(azip[, i], ncol = 16)[, 16:1], col = gray(255:0/255))
```



The naive method

The naive method is to check the distance from each test image to the mean of training image.

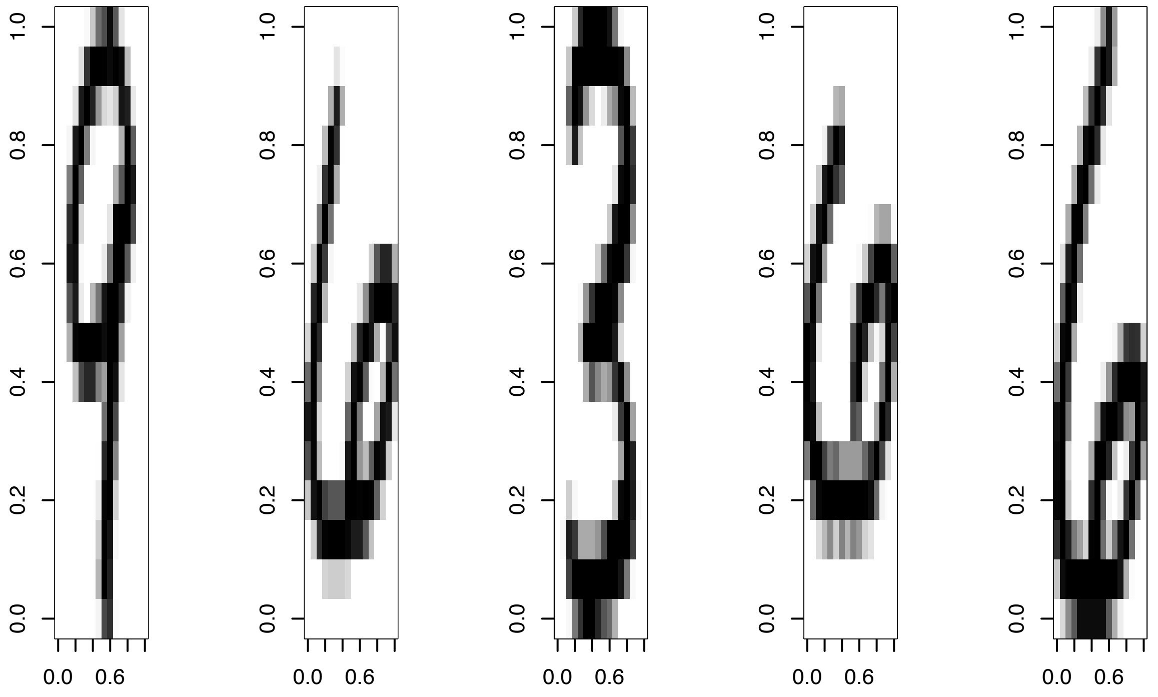
```r
## The mean of training sample of a single digit
digits <- 0:9  # The possible digits in the US postal code
img.mean <- matrix(0, 256, length(digits))

for (i in digits) {
    idx <- (i == dzip)  # the location indicator for the ith digit
    imgi <- azip[, idx, drop = FALSE]
    imgi.mean <- rowMeans(imgi)
    img.mean[, i + 1] <- imgi.mean
}

## Plot the mean image
par(mfrow = c(2, 5))
for (i in 1:10) {
    image(matrix(img.mean[, i], ncol = 16)[, 16:1], col = gray(255:0/255))
}
```



The naive method

- Now it is the time to check the testing sample to the mean of the training sample. We pick the first five testing digits.
- We find the first, third and the fifth are rather easy to classify by eyeballs. But the second and fourth ones are particular difficult.

```
## Sketch a distance function to compute the Euclidean
## distance between two matrices in row wise.
rdist <- function(X, Y) {
    dim.X <- dim(X)
    dim.Y <- dim(Y)
    sum.X <- matrix(rowSums(X^2), dim.X[1], dim.Y[1])
    sum.Y <- matrix(rowSums(Y^2), dim.X[1], dim.Y[1], byrow = TRUE)
    dist0 <- sum.X + sum.Y - 2 * tcrossprod(X, Y)
    out <- sqrt(dist0)
    return(out)
}

## For an unknown testing digit image, compare the distance to
## the means
test.sample <- 1:5

## Let's first plot those testing image
par(mfcol = c(ceiling(length(test.sample)/5), 5))  # five columns

for (i in test.sample) {
    image(matrix(testzip[, i], ncol = 16)[, 16:1], col = gray(255:0/255))
}
```



```
## Calculate the distance from testing sample to the mean in
## the training sample.
img.dist <- rdist(t(testzip[, test.sample]), t(img.mean))


## The classification results by the naive method
apply(img.dist, 1, which.min) - 1
```

```
## V1 V2 V3 V4 V5
##  9  2  3  2  6
```

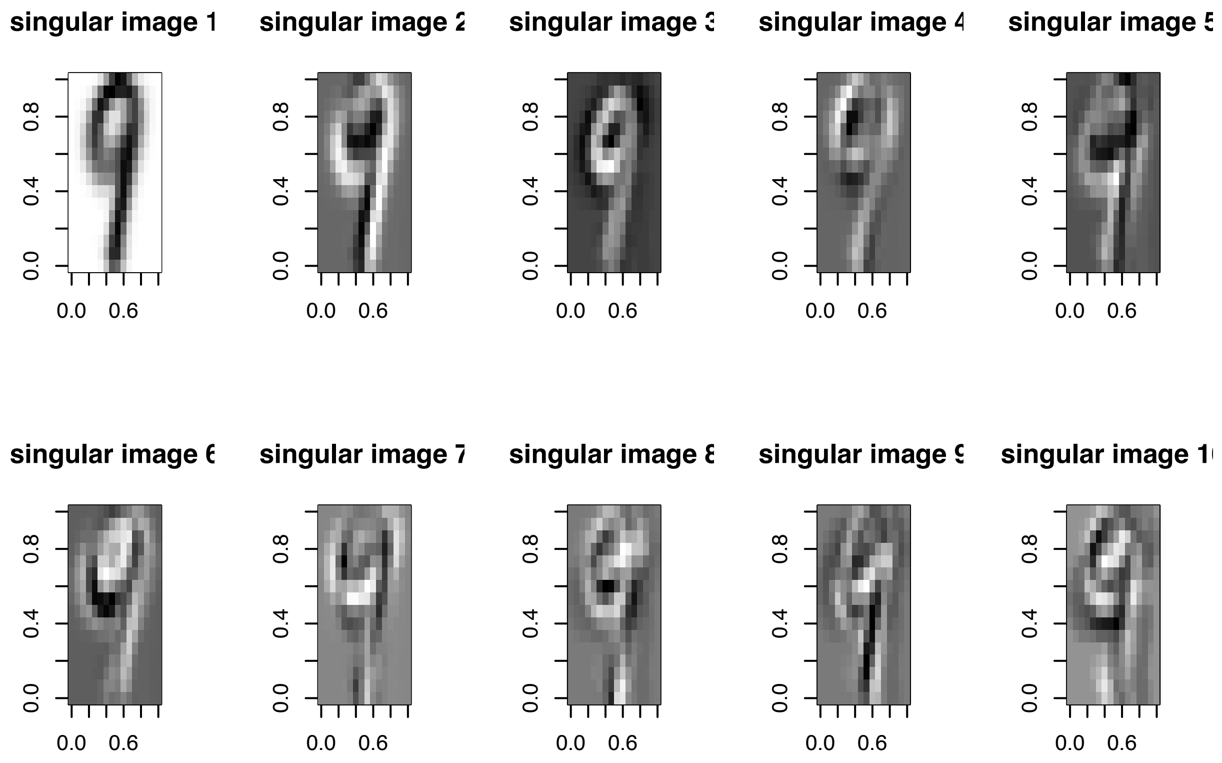
The SVD method

- We pick the digit 9 as an example in this method and plot the first ten singular image from the SVD decomposition.
- We first use four bases, which yields the correct specification as follows. We also tries to classify other digits which gives robust results. But when we increase more basis function, there comes the risk of overfitting.
- It maybe not a good idea to use all the bases but one can always pick up the bases according to the first kth largest eigen values.

The SVD method

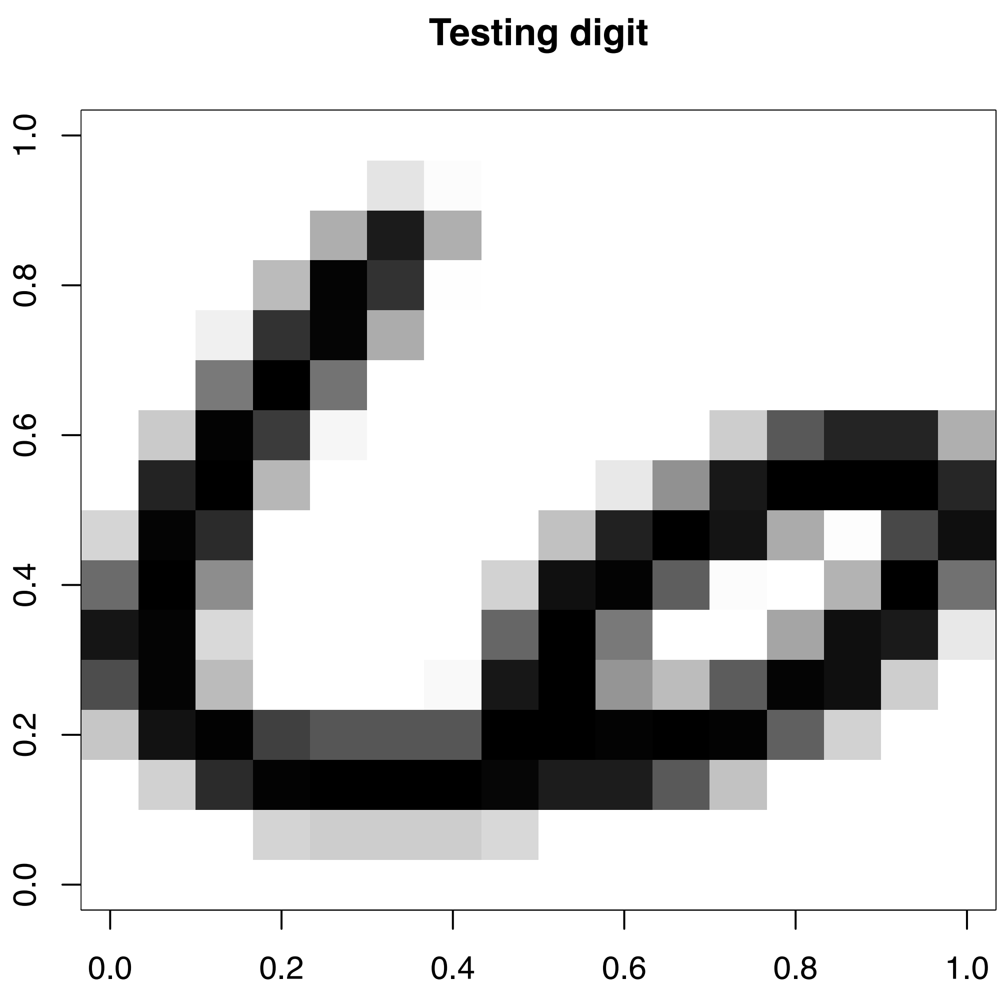
```
## Compute the singular matrix of a single digit in the
## training sample
digit <- 9

## Subtract the matrix for that digit
img.mat <- azip[, digit == dzip, drop = FALSE]
img.matSVD <- svd(img.mat)
```

```
## Plot the singular matrix under different basis.
par(mfrow = c(2, 5))
for (i in 1:10) {
    image(matrix(img.matSVD$u[, i], 16)[, 16:1], col = gray(255:0/255),
        main = paste("singular image ", i, sep = ""))
}
```



Testing based on SVD

```
## Do the least square method with different basis and find
## the minimal residuals.

## The testing digit matrix
test.idx <- 2
image(matrix(testzip[, test.idx], 16)[, 16:1], col = gray(255:0/255),
    main = paste("Testing digit"))
```



```
resid.norm <- matrix(NA, 10, 1, dimnames = list(0:9, "resid"))
for (i in 0:9) {
    img.mat <- azip[, i == dzip, drop = FALSE]
    img.matSVD <- svd(img.mat)
    # basis.max <- ncol(img.matSVD$u)
    basis.max <- 4
    resid.norm[i + 1, ] <- norm(matrix(lm(testzip[, test.idx] ~
        0 + img.matSVD$u[, 1:basis.max])$resid), "F")
}
resid.norm
```

```
##       resid
## 0 11.815495
## 1 12.536173
## 2 11.388341
## 3 12.706473
## 4 12.141602
## 5 12.926655
## 6  9.455279
## 7 12.577303
## 8 12.061615
## 9 12.551339
```

The SVD method

- We will find out when we overfit (see the plot of classification success as a function of the number of basis vectors.)
- To see this, we loop over all testing observations and number of bases from 1 to 88, and then count the correct specification numbers.

Statistical Computing

Lecture 14: SVD in Text Mining

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University http://yanfei.site

What is text mining?

Raw human written text ⇒ Structured information

- The biggest difference between text mining and general data analysis is that it deals with text data, instead of numeric values.
- Sometimes text mining is called 'Natural Language Processing (NLP)', especially in computer science.
- Most text mining methods are based on word frequency in real world.

What do you usually see in text mining?

Concepts in text mining

- Corpus
  - a collection of documents (e.g., a collection of different job description documents)
- Word segment
  - segment each text into words
  - stopwords: common words that generally do not contribute to the meaning of a sentence, at least for the purposes of information retrieval and natural language processing. These are words such as the and a. Most search engines will filter out stopwords from search queries and documents in order to save space in their index.
- DocumentTermMatrix
  - Each row is a document, while each column shows word frequencies of the corresponding word.
  - This is the very basic data structure for text mining.
- TermDocumentMatrix
- Text clustering
  - Group similar documents together according to their similarities.
- Topic models
  - Find topics which the corpus is talking about.

SVD in text mining

Latent Semantic Analysis (LSA)

- Extract relationships between the documents and terms assuming that terms that are close in meaning will appear in similar (i.e., correlated) pieces of text.
- LSA leverages a singular value decomposition (SVD) factorization of a term-document matrix to extract these relationships.

$$A = U\Sigma V^T.$$

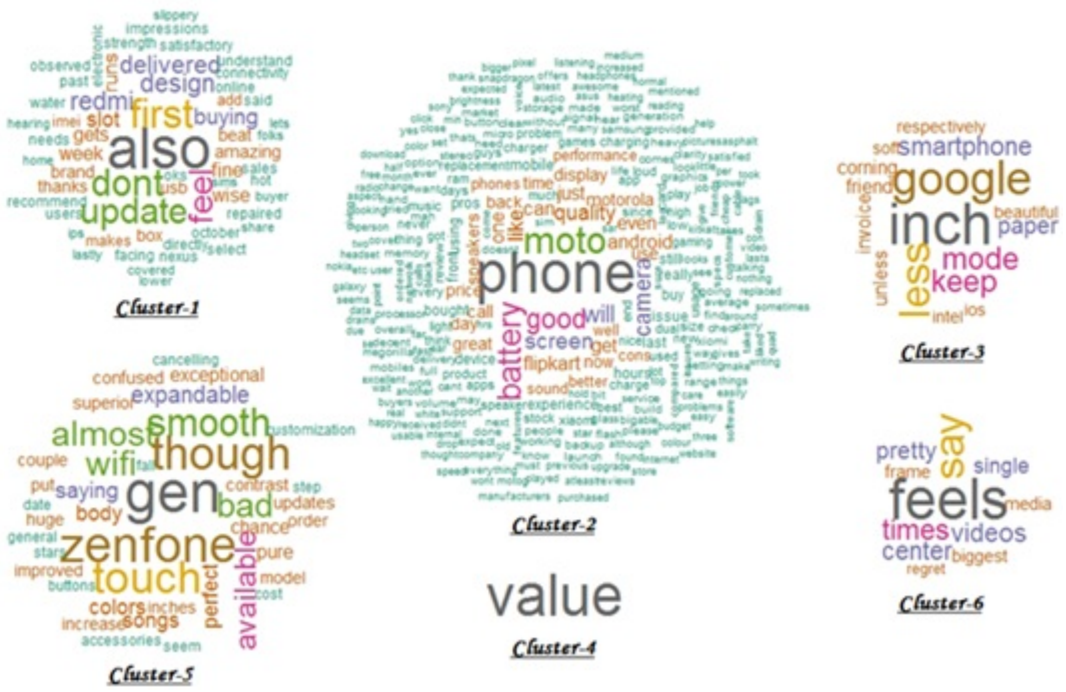
- U contains the eigenvectors of the term correlations, AA^T.

- V contains the eigenvectors of the document correlations, $A^T A$.

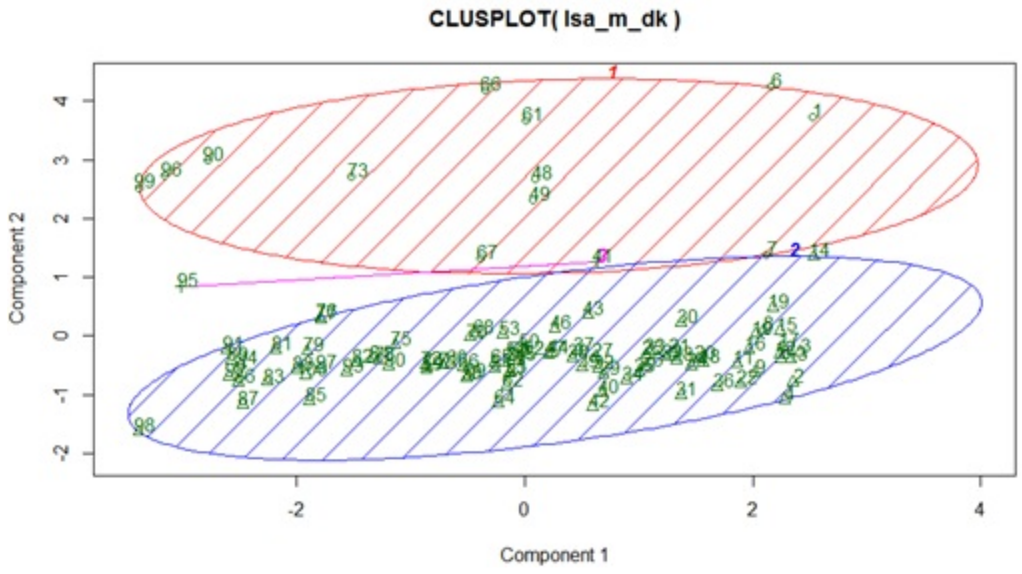
LSA to the Rescue!

- LSA often remediates the curse of dimensionality problem in text analytics:
  - The matrix factorization has the effect of combining columns, potentially enriching signal in the data.
  - By selecting a fraction of the most important singular values, LSA can dramatically reduce dimensionality.
- SVD is effective and is a staple of text analytics pipelines!

LSA applications - Term similarity



LSA applications - Document similarity


CLUSPLOT(lsa_m_dk)
Component 1
Component 2