MAT325 E-Pack: Numerical Analysis

Cheng Peng

West Chester University

Contents

1 Introduction **7**

2 Calculus Review **9**
2.1 Limits and Continuity 9
2.2 Differentiation 11
2.3 Integration 16

3 Scientific Computing with R **21**
3.1 Vectors and Matrices 22
3.2 Built-in Mathematical Functions and Operators 25
3.3 Graphic Functions in Base R 27
3.4 Rounding and Related Functions 28
3.5 Control Statements 31
3.6 Numerical Example 36
3.7 User Defined Functions 39

4 Basics of Data Representation and Error Analysis **45**
4.1 Data Representation 45
4.2 How data is stored in computer memory 50

5 Error Analysis, Algorithms and Convergence **55**
5.1 Error Analysis 55
5.2 Algorithms and Convergence 58

6 Bisection Method **63**
6.1 The Question 63
6.2 Bisection Method 63
6.3 Error Analysis 65
6.4 Numerical Examples. 66

7 Fixed Point Method **71**
7.1 Some Theories of the Fixed Point Method 71
7.2 Fixed Point Iteration 72
7.3 Error Analysis 74

7.4 Numerical Example 75

8 Newton's Method **77**
8.1 Notations: Big O and Little o 77
8.2 Foundations of Newton Method 78
8.3 Algorithm and Pseudo-code 79
8.4 Error Analysis 81

9 Secant Method **83**
9.1 Secant Method 83
9.2 Secant Algorithm and Implementation 84
9.3 Error Analysis 88
9.4 False Position Method 89

10 Lagrange Interpolation **91**
10.1 Concepts of Interpolation Method 93
10.2 The Lagrange Interpolation 95
10.3 Lagrange Algorithm and Implementation 98
10.4 Error Analysis 102

11 Newton Interpolation **105**
11.1 Some Definitions 105
11.2 Newton Interpolation Polynomial 110
11.3 Error Analysis 114
11.4 Some Remarks of Newton Interpolation 115
11.5 Implicit Loops in R Programming 116

12 Gauss Elimination Method **127**
12.1 Naive Gaussian Elimination Methods 129
12.2 Gaussian Elimination Algorithm 133
12.3 Gauss-Jordan Elimination (`Optional`) 137

13 Determinant and Inversion of Matrices **139**
13.1 Concepts of Matrix: A Review 139
13.2 Determinant of a Square Matrix 143
13.3 Matrix Inversion 146

14 Matrix Factorization **151**
14.1 Elementary and Permutation Matrices 152
14.2 LU Factorization Algorithm 157
14.3 Benefit of LU Factorization 160

15 Least Square Approximation **163**
15.1 Concepts of Least Square Approximation 163
15.2 Approximating One-variable Linear Function 164
15.3 Approximating Multiple-variable Linear Function 167
15.4 Approximating One-variable Polynomial Function 169

16 Concepts of Spline Interpolations **175**
16.1 Linear Splines 177
16.2 Quadratic Spline Interpolation 179

17 Cubic Spline Interpolation **187**
17.1 Cubic Splines 187
17.2 Clamped Splines 198
17.3 Error Analysis 198

18 Newton Method for Nonlinear System and Optimization **201**
18.1 Multivariate Taylor Expansion 201
18.2 Newton Method for Nonlinear Systems 202
18.3 Optimization 206
18.4 A Case Study 214

19 Numerical Integration I **221**
19.1 Numerical Approximation of Derivatives 221
19.2 Trapezoid Method 223
19.3 Simpson's One-third Rule 226

20 Romberg Integration and Gaussian Quadratures **235**
20.1 Richardson Extrapolation 235
20.2 Romberg Integration 240
20.3 Gaussian Quadrature 245
20.4 Remarks 248

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Chapter 1

Introduction

This *E-Pack* is a self-contained homegrown Ebook that contains all topics covered in current MAT 325 (Numerical Analysis) at WCU. All technical terms used in this Ebook are consistent with those used in the *required* textbook.

Several features of eBook are

- It is convenient! Three formats (PDF, HTML, and ePub) of the E-Pack are accessible from different devices.
- It is focused! This E-course pack only **picks** the topics required for this course.
- It is organized! Each week covers one or two topics and weekly work is clearly organized through a separate course website.
- It is more intuitive since the E-pack uses visual aids whenever possible. Animated graphs that are alive in the HTML version are used to demonstrate the limiting process of various approximations.
- Every topic follows the same workflow: explaining the mathematics (including error analysis) and the use case of methodology => developing the algorithm and pseudo-code => translating the pseudo-code to programming language => and applying the methods with examples. The workflow is not dependent on a specific programming language!

Chapter 2

Calculus Review

This note reviews the basics of Calculus which will be used throughout the semester. We will not derive or prove anything in this note.

2.1 Limits and Continuity

The concept of continuity of function is defined based on the concept of the limit of a function at a given point. They are core concepts in numerical analysis. We use ϵ-δ language to summarize the limits and continuity. We only use single-variable functions to describe these concepts.

2.1.1 Limits

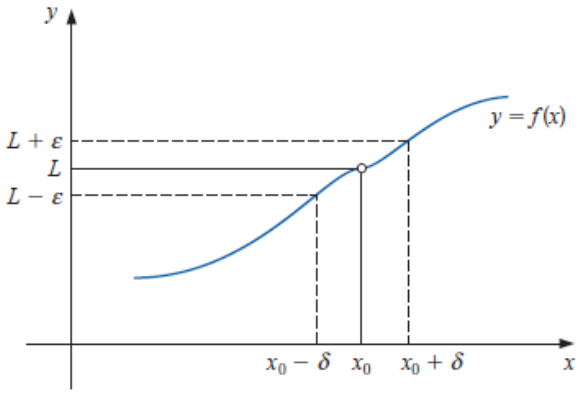
A function f defined on a set X of real numbers has the **limit** L at x_0, written

$$\lim_{x \to x_0} f(x) = L,$$

if, given any real number $\varepsilon > 0$, there exists a real number $\delta > 0$ such that

$$|f(x) - L| < \varepsilon, \quad \text{whenever} \quad x \in X \quad \text{and} \quad 0 < |x - x_0| < \delta.$$

This definition can be graphically explained in the following figure



2.1.2 Continuity

The continuity of a function is defined based on the concept of limit.

Definition: Let f be a function defined on a set X of real numbers and $x_0 \in X$. Then f is continuous at x_0 if

$$\lim_{x \to x_0} f(x) = f(x_0).$$

The function f is continuous on the set X if it is continuous at each number in X.

2.1.3 Convergence of A Sequence

Convergence is one of the fundamental concepts in numerical analysis which is related to the limit of a sequence of real or complex numbers. In numerical analysis, the order of convergence and the rate of convergence of a convergent sequence are quantities that represent how quickly the sequence approaches its limit.

Definition: Let $\{x_n\}_{n=1}^{\infty}$ be an infinite sequence of real numbers. This sequence has the **limit x (converges to x)** if, for any $\epsilon > 0$ there exists a positive integer $N(\epsilon)$ such that $|x_n - x| < \epsilon$, whenever $n > N(\epsilon)$. The notation

$$\lim_{n \to \infty} x_n = x, \text{ or } x_n \to x \text{ as } n \to \infty,$$

means that the sequence $\{x_n\}_{n=1}^{\infty}$ converges to x.

Definition: If f is a function defined on a set X of real numbers and $x_0 \in X$, then the following statements are equivalent:

a. f is continuous at x_0;

b. If $\{x_n\}_{n=1}^{\infty}$ is any sequence in X converging to x_0, then $lim_{n \to \infty} f(x_n) = f(x_0)$.

Remark: The values of above sequences in X could be from both sides of x_0.

https://github.com/pengdsci/MAT325/raw/main/w01/img/w01Note1-1-Comvergence.gif

2.2 Differentiation

In numerical analysis, many numerical algorithms are based on the assumption that the curve of the underlying function is continuous and **smooth** over an interval. The smoothness of a curve is characterized by the concept of differentiation.

2.2.1 Definition

(Average) Rate of Change: The rate of change of a function $f(x)$ over interval $[x, x + \Delta x]$ is defined to be

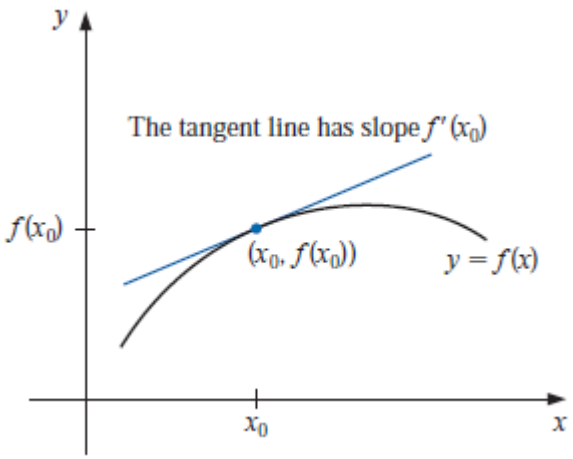
$$\frac{f(x + \Delta x) - f(x)}{\Delta}$$

Geometrically, the average rate of change of $f(x)$ over the interval is the slope of the secant line that passes through the two points on the curve corresponding to the two ending values of the interval.

Definition: Let f be a function defined in an open interval containing x_0. The function f is differentiable at x_0 if

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

exists. The number $f(x_0)$ is called the derivative of f at x_0. A function that has a derivative at each number in a set X is differentiable on X.

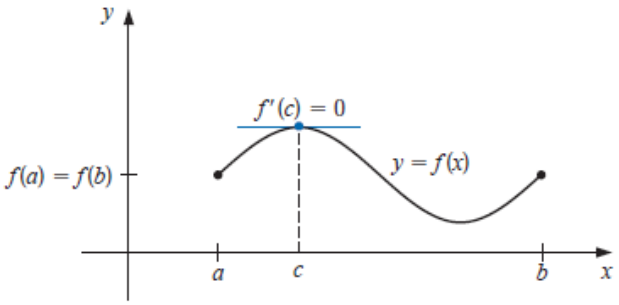


We will assume that you are proficient in using all rules of derivatives. Particularly, the power and chain rules.

2.2.2 Properties

Some of the properties and existence theorems will be used in developing numerical algorithms for optimization.

Rolle's Theorem: Suppose $f \in C[a, b]$ (continuous) and f is differentiable on (a, b). If $f(a) = f(b)$, then a number c in (a, b) exists with $f(c) = 0$.

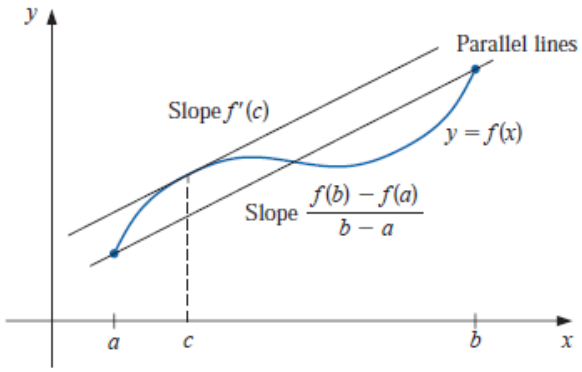


The following mean value theorem is a generalization of Rolle's Theorem.

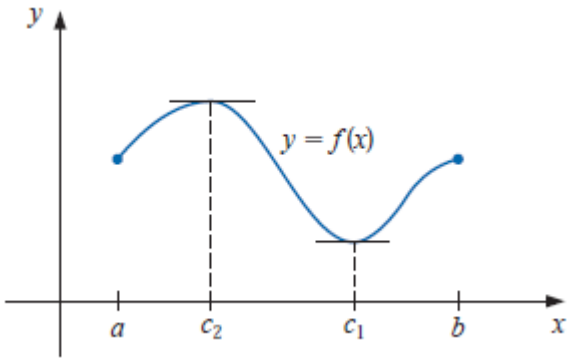
Mean Value Theorem: If $f \in C[a, b]$ and f is differentiable on (a, b), then a number c in (a, b) exists with

$$\frac{f(b) - f(a)}{b - a}.$$

We can visualize the mean value theorem in the following figure.



Extreme Value Theorem: If $f \in C[a, b]$, then $c_1, c_2 \in [a, b]$ exist with $f(c_1) \leq f(x) \leq f(c_2)$, for all $x \in [a, b]$. In addition, if f is differentiable on (a, b), then the numbers c_1 and c_2 occur either at the endpoints of $[a, b]$ or where f is zero.



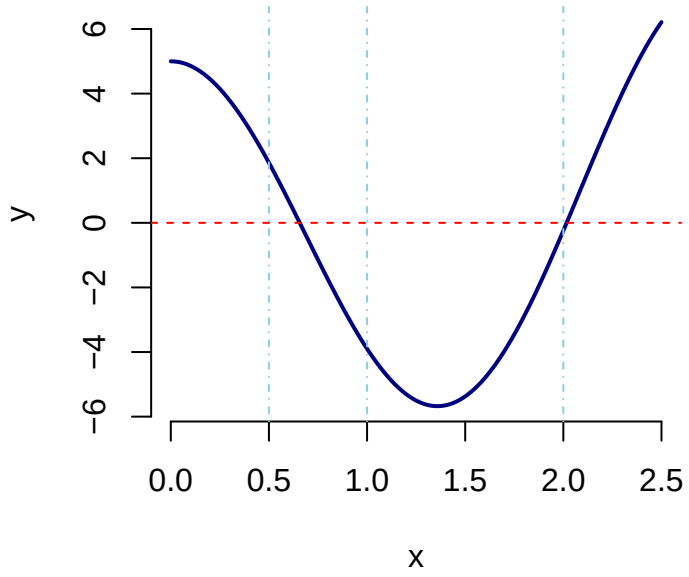
Example 1: Use a computer program to find the absolute minimum and absolute maximum values of

$$f(x) = 5\cos(2x) - 2x\sin(2x)$$

on the intervals $[1, 2]$ and $[0.5, 1]$, respectively.

Solution: We could free online graphing tools such as *WolframAlpha* (https://www.wolframalpha.com/) to sketch the function. We will use **R** to plot the function.

```
x=seq(0, 2.5, by = 0.01)
y=5*cos(2*x) - 2*x*sin(2*x)
plot(x,y, type="l", lwd=2, col="navy", bty="n")
abline(h=0, lty=2, col="red")
abline(v=c(0.5, 1, 2), lty=4, col="skyblue")
```



(a). We can see from the above figure that, the absolute maximum on $[1, 2]$ is $f(2) = 5\cos(2 \times 2) - 2 \times 2\sin(2 \times 2) = -0.2410081$ (see the following code)

```
5*cos(4) - 4*sin(4)
```

```
## [1] -0.24100812
```

The absolute minimum is the solution to $f'(x) = 0$. That is, we need to solve equation $-10\sin(2x) - 2\sin(2x) - 4x\cos(2x) = 0$ that is equivalent to $\tan(2x) = x/3$. This is a nonlinear equation. There is no closed form of the solution. We will introduce various methods to find the root of this equation. For now, we simply call an R function to find the root.

```
fn = function(x) tan(2*x)+x/3  # define the function
root = nleqslv(1.5, fn)$x      # the first list $x in the output is the root
f.min = 5*cos(2*root) - 2*root*sin(2*root)  # finding the absolute minimum
list(root = root, abs.min = f.min)
```

```
## $root
## [1] 1.3582299
##
## $abs.min
## [1] -5.6753013
```

Therefore, the absolute minimum is $f(1.35823 - 5.675301) = -5.675301$.

(2). Since the function is strictly decreasing on $[0.5, 1]$, to find the absolute minimum and maximum of the function on $[0.5, 1]$, we simply evaluate the function at $x = 0.5$ and $x = 1$ (see the following code).

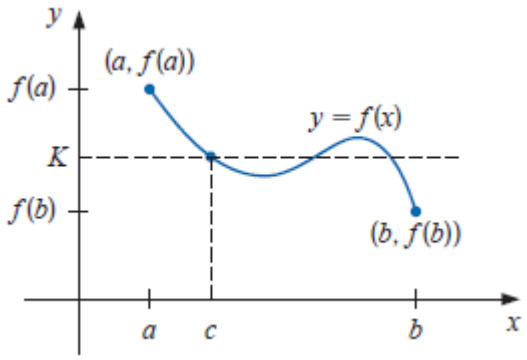
```
list(abs.max = 5*cos(2*0.5) - 2*0.5*sin(2*0.5),
     abs.min = 5*cos(2*1) - 2*1*sin(2*1))
```

```
## $abs.max
## [1] 1.8600405
##
## $abs.min
## [1] -3.899329
```

Therefore, the absolute minimum is $f(1) = -3.899329$ and the absolute maximum $f(0.5) = 1.860041$.

Intermediate Value Theorem: If $f \in C[a, b]$ and K is any number between $f(a)$ and $f(b)$, then there exists a number c in (a, b) for which $f(c) = K$.

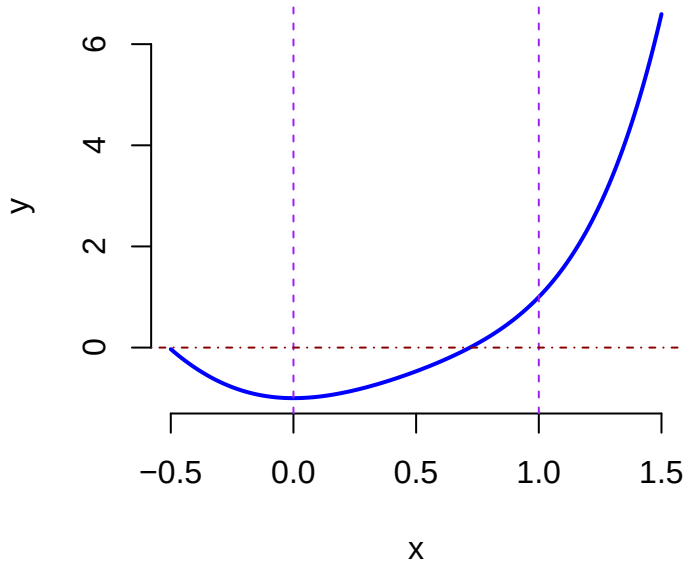
The results in intuitive from the following figure.



Example 2: Show that $x^5 - 2x^3 + 3x^2 - 1 = 0$ has a solution in the interval $[0, 1]$.

Solution: We want to prove the *existence* of a solution in $[0, 1]$. We first sketch the function in the following.

```
x = seq(-0.5, 1.5, by =0.01)
y = x^5-2*x^3 + 3*x^2-1
plot(x,y, type="l", lwd=2, col="blue", bty="n")
abline(v=c(0,1), lty = 2, col = "purple")
abline(h=0, lty = 4, col="darkred")
```



After inspecting the curve, choose $a = 0$ and $b = 1$ and then use the intermediate value theorem. Note that $f(0) = -1$ and $f(1) = 1$. Therefore, $f(x) = 0$ has a solution in $[0, 1]$.

Remark: The intermediate value theorem states that the existence of *at least one solution* in interval $[a, b]$.

2.3 Integration

Numerical integration is one of the major topics in numerical analysis. We focus on the definite integral of the single variable function.

2.3.1 Series vs Sequence

A **sequence** is an arrangement of any objects or a set of numbers in a particular order followed by some rule. If $a_1, a_2, a_3, a_4, \cdots$, denote the terms of a sequence, then $1, 2, 3, 4, \cdots$ denotes the position of the term. A sequence can be defined based on the number of terms i.e. either finite sequence or infinite sequence.

If $a_1, a_2, a_3, a_4, \cdots$ is a sequence, then the corresponding **series** is given by $S_n = a_1 + a_2 + a_3 + \cdots + a_n$ for $n = 1, 2, \cdots,$. If $\lim_{n\to\infty} s_n = A$ (A is finite), then we

call series s_n converges to A.

2.3.2 Definition of Definite Integral

Riemann Integral: The Riemann integral of the function f on the interval $[a, b]$ is the following limit, provided it exists:

$$\int_a^b f(x)dx = \lim_{\max \Delta x_i \to 0} \sum_{i=1}^{n} f(z_i)\Delta x_i,$$

where the numbers $x_0, x_1, \cdots, x_n$ satisfy $a = x_0 \leq x_1 \leq \cdots \leq x_n = b$, where $x_i = x_i - x_{i-1}$, for each $i = 1, 2, \cdots, n$, and z_i is arbitrarily chosen in the interval $[x_{i-1}, x_i]$.

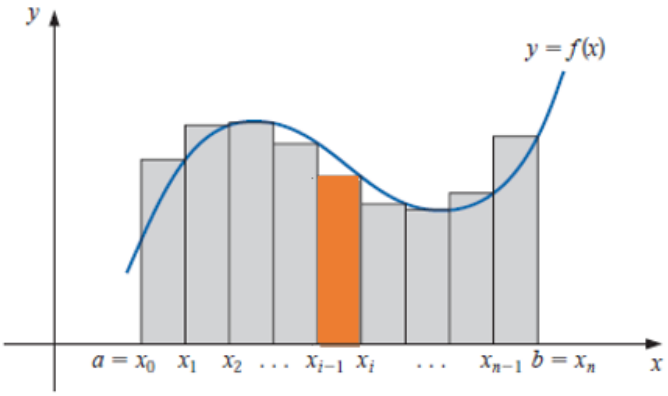
Remark1: The above Riemann integral involves two concepts

1. **Partition of an interval**: $a = x_0 \leq x_1 \leq \cdots \leq x_n = b$ is also called a partition of interval $[a, b]$.
2. $D_n = \sum_{i=1}^{n} f(z_i)\Delta x_i$ is so called Darboux sum.

In practice, we take a simple **equally spaced partition** to evaluate the Riemann integral. To be specific, use the partition such that $x_i = a + i(b-a)/n$. With this equally-spaced partition, the Riemann integral has the following simple form

$$\int_a^b f(x)dx = \lim_{n\to\infty} \frac{b-a}{n} \sum_{i=1}^{n} f(x_i),$$

The geometric display of the Darboux in the above expression is given below



The area of the orange rectangle is the i-th term in the Darboux sum.

The following animated graph shows the process of approximating the integral by the Darboux sum.

https://github.com/pengdsci/MAT325/raw/main/w01/img/w01-GIFRiemann Sum.gif

2.3.3 Taylor Expansion

Since polynomial functions are relatively easier to handle in mathematics. If we want to study the *local* behavior of a complicated function (algebraically), we could use a polynomial to approximate the function locally. The Taylor series is one such polynomial that is used frequently in practice.

Taylos's Theroem: Suppose $f \in C^n[a,b]$, that $f^{(n+1)}$ exists on $[a,b]$, and $x_0 \in [a,b]$. For every $x \in [a,b]$, there exists a number $\xi(x)$ between x_0 and x with

$$f(x) = P_n(x) + R_n(x),$$

where

$$P(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n$$

$$= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k.$$

and

$$R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)^{n+1}.$$

Here $P_n(x)$ is called the **nth Taylor polynomial** for f about x_0, and $R_n(x)$ is called the **remainder term (or truncation error)** associated with $P_n(x)$.

When $x_0 = 0$, the Taylor polynomial is called **Maclaurin polynomial**.

The following animated graph shows the process of Taylor approximation to function $f(x) = e^{-x}\sin(x)$.

https://github.com/pengdsci/MAT325/blob/main/w01/img/w01Note1-1-TaylorExansion02.gif

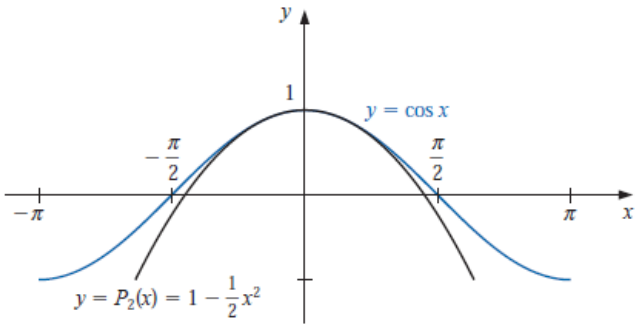
Example: Let $f(x) = cosx$ and $x_0 = 0$. Determine the second Taylor polynomial for f about x_0.

Solution: We use the Taylor theorem to expand $cos(x)$ up two 2nd order

$$cos(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(\xi(x))}(0)}{3!}x^3$$

$$= 1 - \frac{1}{2}x^2 + \frac{1}{6}x^3\sin[\xi(x)].$$

where $\xi(x)$ is in $[0, x]$. The following figure shows the approximation.



Chapter 3

Scientific Computing with R

R was initially created by group of statisticians for data analysis (for free). In the past decade, many people from other disciplines contributed to the continuous development of this program. It is among the top programming language in data science and machine learning and widely used to perform a variety of non-statistical tasks, including data processing, information visualization, data mining, and scientific computing, etc.

During the semester, I will write series of short notes on R to implement numerical algorithms to be covered in the course. I will also write one or two lab notes to cover optimization problems in machine learning and data science.

The following three books focus on using R for scientific computing. You can find one of them as a reference when you make programs for this class.

1. **Introduction to scientific programming and simulation using R**. This book can be found from internet.
2. **Mastering Scientific Computing with R**. WCU library has this eBook. You can access this book using the following link. https://ebookcentral.proquest.com/lib/wcupa/detail.action?pq-origsite=primo&docID=1936749
3. **Using R Numerical Analysis in Science and Engineering**. You can also find this from internet.

For those who programmed in MATLAB, you can check the following page to see the back-to-back syntax comparison between R and MATLAB https://mathesaurus.sourceforge.net/octave-r.html.

R is case sensitive!.

3.1 Vectors and Matrices

Vectors and matrices are two major R objects that will be used frequently in numerical analysis. This section outlines the definition and utilization of vectors and matrices.

3.1.1 Vectors

A vector is a collection of *like* elements without dimensions. The vector element or elements must be of the same types of data (either **character**, **numeric**, or **logical**).

3.1.1.1 Definition

An R vector defined by a built-in R function `c()` (**c** stands for concatenate). The following are examples of basic types of vectors.

```
intVec <- c(1,3,6,7)                    # vector of integers
charVec = c('One','Two','Three')        # vector of characters, string vector
logiVec = c(FALSE, TRUE)                # logical vectors
single.Val.Vec = c("convergent")        # single element character vector
emptyVec = NULL                         # empty vector / null vector
##
intVec                                  # type the of the name of intVec
```

```
## [1] 1 3 6 7
```

3.1.1.2 Use of Vector Index

One can access elements in a vector through index using square bracket `[idx]`. Similar to MATLAB, **R index starts from 1**!

```
exampleVec = c(1, 2, 3, 2, 7, 9, 11, 15, 7, 2)  # use this vector as an example
###
exampleVec[7]             # extract the 7th element in the vector
```

```
## [1] 11
```

```
exampleVec[c(1,2,3)]      # extract the first three elements, c(1,2,3) is the vector of
```

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

```
exampleVec[-7]            # drop the 7th element from the vector
```

```
## [1]  1  2  3  2  7  9 15  7  2
```

```
exampleVec[-c(1,2,3)]     # drop the first three elements
```

```
## [1]  2  7  9 11 15  7  2
```

```
duplVec = exampleVec     # duplicate an existing vector and rename it
## The following code replaces elements with NEW elements
duplVec[c(9,10)] = c(99,100)   # replace elements 7, 4 with 99 and 100 respectively.
duplVec                  # display the modified vector
```

```
##  [1]   1   2   3   2   7   9  11  15  99 100
```

3.1.1.3 Operations Between Vectors

The following examples show the operations commonly used in error analysis.

```
A = c(5, 2, 3, 7, 5, 1, 9)
B = c(3, 2, 4, 8, 2, 7)
## next we define different new vectors using A and B
new01 = c(A,B)     # concatenate A and B
new01              # display new01
```

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

```
new02 = A-5        # subtract 5 from individual element in A
new02
```

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

```
new03 = A^2        # square each individual element in A
new03
```

```
## [1] 25  4  9 49 25  1 81
```

```
new04 = 2*A        # multiply each element of A by 2
new04
```

```
## [1] 10  4  6 14 10  2 18
```

3.1.1.4 Shortcuts for Defining Vectors

There are shortcuts to define patterned vectors (sequence). The following are few examples.

```
seq.vec = seq(1, 100, by = 2)       # This defines a sequence: 1, 3, 5, 7, ..., 99. by = jump
seq.vec
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59
## [38] 75 77 79 81 83 85 87 89 91 93 95 97 99
```

```
seq.vec01 = seq(1, 99, length = 5)  # This defines a sequence with 5 numbers that are equally spa
seq.vec01
```

```
## [1]  1.0 25.5 50.0 74.5 99.0
```

```
seq.vec02 = rep(1, 100)             # This defines a sequence with 100 1s.
seq.vec02
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [56] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
seq.vec03 = 3:10                    # This sequence: 3,4,5,6,7,8,9,10.
seq.vec03
```

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

```
seq.vec04 = letters                 # letters is a built-in vector of all lower case le
seq.vec04
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
```

```
seq.vec05 = LETTERS                 # LETTERS is a built-in vector of all uppercase le
seq.vec05
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T"
```

```
LETTERS[1:7]                        # first 7 uppercase letters.
```

```
## [1] "A" "B" "C" "D" "E" "F" "G"
```

3.1.2 Matrices

R matrices are two dimensional table indexed by two subscripts using $[i, j]$, where i = index of row of the matrix and j = index of the column of the matrix. On can access the matrix using index $[i, j]$. The following are some examples of matrices

```
vec0 = 1:36
m01 = matrix(vec0, ncol = 9, byrow = TRUE)   # this defines a 4x9 matrix, the cells we
m01
```

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

```
m02 = matrix(vec0, nrow = 6, byrow = FALSE)  # this defines a 6x6 square matrix, cells
m02
```

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

```
m03 = matrix(ncol = 5, nrow = 6)        # this defined a 5x6 empty matrix
m03
```

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

```
m03[4,5] = 99                           # replace the element in row 4 and column 5 with 99.
m03
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   NA   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA   99
## [5,]   NA   NA   NA   NA   NA
## [6,]   NA   NA   NA   NA   NA
```

3.2 Built-in Mathematical Functions and Operators

R has built in most of the commonly used mathematical functions and important scalars. The following is a partial list.

3.2.1 Arithmetic Operators

`+` – addition

`-` – subtraction

`*` – multiplication

`/` – division

`^` – raise to the power of

3.2.2 Basic Mathematical Functions

`abs()` - absolute value

`sqrt()` - square root

`round()` - rounding function

`ceiling()` - rounding up

`floor()` - rounding down

`sign()` - sign of a number

`exp()` - natural base exponential function

`log()` - natural base logarithmic function

`log10()` - base 10 logarithmic function

3.2.3 Trigonometry

`sin()` – sine

`cos()` – cosine

`tan()` – tangent

`asin()` – sine inverse

`acos()` – cosine inverse

`atan()` – tangent inverse

3.2.4 Linear Algebra

`+` – element-wise addition

`-` – element-wise subtraction

`*` – element-wise multiplication

`/` – element-wise division

`%*%` – matrix multiplication

`t()` – transpose

`eigen()` – eigenvalues and eigenvectors

`solve()` – inverse of matrix

`rbind()` – combines vectors of observations horizontally into matrix class

`cbind()` – combines vectors of observations vertically into matrix class

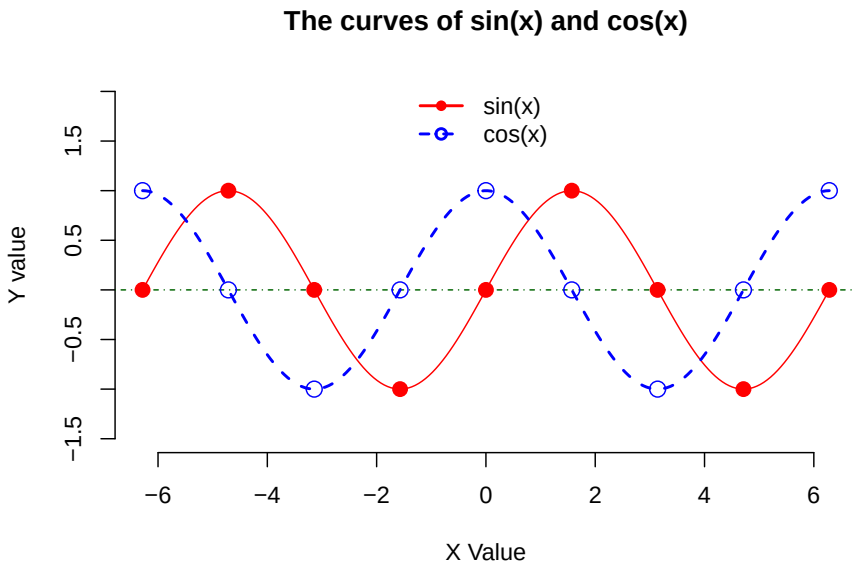
3.3 Graphic Functions in Base R

Base R graphical system contains a set of **high-level plotting functions** such as `plot()`, `hist()`, `barplot()`, etc. and also a set of **low-level functions** that are used jointly with the high-level plotting functions such as `points()`, `lines()`, `text()`, `segments()`, etc. to make a flexible graphical system.

Next we use a simple example to illustrate the basic graphic functions. We draw the curves of two functions $f_1(x) = \sin(x)$ and $f_2(x) = \cos(x)$ over interval $[-2\pi, 2\pi]$.

```
x = seq(-2*pi, 2*pi, length = 200)  # define a sequence of numbers on the interval
f1 = sin(x)                         # evaluate f1 = sin(x) at these values
f2 = cos(x)                         # evaluate f2 = cos(x) at these values
##
plot(x, f1,                         # x- and y- coordinates
        type = "l",                 # lower case l in the quote means line. The default option is
        xlab = "X Value",           # label of x-axis, if not specified, the name of x vector wil
        ylab = "Y value",
        ylim = c(-1.5, 2.0),        # the limits of y-axis, must be a vector of two values. The d
        xlim = c(-2*pi, 2*pi),      # the limits of x-axis, if it is not provided, the min and ma
        col = "red",                # select a color for the line
        lty = 1,                    # line type. 1 = solid, 2 = dash, ....
        cex = 1.2,                  # size, default = 1
        main = "The curves of sin(x) and cos(x)",   # the title of the plot.
        bty = "n"                   # remove the outbox on the plot. The default plot has an oute
     )
abline(h = 0, lty = 4, col = "darkgreen")                       # add a horizontal line at y = 0
lines(x, f2, lty = 2, lwd = 2, col = "blue")                    # cos(x) curve to the existing pl
## add a few special points on both curves
pt.x = seq(-2*pi, 2*pi, length = 9)                             # select 9 x-coordinates to make
pt.y1 = sin(pt.x)                                               #
pt.y2 = cos(pt.x)
## Adding points to the two curves
points(pt.x, pt.y1,                     # add points to f1 = sin(x)
         pch = 16,                      # point types, there 20+ types of points in R
         col = "red",
         cex = 1.5)                     # size of the points, default cex = 1
points(pt.x, pt.y2, pch = 21, col = "blue", cex = 1.5)
## adding legends
legend("top",      # location of the legend, other locations: "bottom", "topleft", "topright" bot
                   # the coordinates of the location to place the legend.
              c("sin(x)", "cos(x)"),          # label of the corresponding curves
              pch = c(16, 21),                # the corresponding point type
              col = c("red", "blue"),         # corresponding color
              lty = c(1, 2),                  # corresponding line type
              lwd = c(2,2),                   # corresponding line width
```

```
       bty = "n")                    # no box placed around the legend
```



3.4 Rounding and Related Functions

There are several R functions we can use to handle decimals and manage significant digits in numerical analysis. A list of a few such functions in the base R.

3.4.1 round()

`round(number,digits)` rounds the number to the number of digits provided. Here is an example of the round function in action.

```
round(1234.56789,3)    # keep three decimal places
```

```
## [1] 1234.568
```

```
round(1234.56789,5)    # R only displays 7 digits by default
```

```
## [1] 1234.5679
```

```
round(1234.56789,0)    # keep no decimal place
```

```
## [1] 1235
```

```
round(1234.56789,-3)   # set three digits before decimal points to zeros
```

```
## [1] 1000
```

```
round(1234.56789,-5)
```

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

3.4.2 Controlling Number of Displayed Digits

There are different ways to change the number of displayed digits. We can change the default `option(digits = 7)` to `option(digits = 10)` to display 10 digits (Caution: this is a global option!).

```
options(digits = 10)
round(1234.56789,3)    # keep three decimal places
```

```
## [1] 1234.568
```

```
round(1234.56789,5)    # R only displays 7 digits by default
```

```
## [1] 1234.56789
```

```
round(1234.56789,6)    # R only displays 7 digits by default
```

```
## [1] 1234.56789
```

The c-style formatting function can also be used to display the desired number of digits.

```
sqrt(2)                    # this will display 10 digits. why?
```

```
## [1] 1.414213562
```

```
sprintf("%.20f", sqrt(2))   # Note this displays a string value
```

```
## [1] "1.41421356237309514547"
```

```
options(digits = 20)        # change options to display 20 digits
sqrt(2)
```

```
## [1] 1.4142135623730951455
```

```
options(digits = 7)         # change the options back to the default
```

3.4.3 `Signif()`

`Signif()` is an R **rounding function** with the format of `signif(number,digits)` and it **rounds** the number to the number of digits provided.

```
signif(1234.566789,4)
```

```
## [1] 1235
```

```
signif(1234.566789,7)
```

```
## [1] 1234.567
```

```
signif(1234.566789,1)
```

```
## [1] 1000
```

The `signif function` round to the given number of digits. Both the round and `signif functions` use standard rounding conventions.

3.4.4 `floor()` and `ceiling()`

`floor()` is a rounding function with the format of `floor(number)` and it rounds the number to **the nearest integer that is less than its value**.

```
floor(3.14159)
```

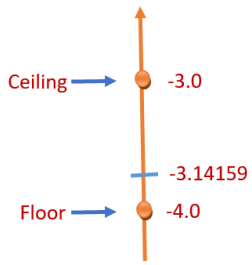
```
## [1] 3
```

```
floor(-3.14159)
```

```
## [1] -4
```

`floor()` just drops the decimal places of a decimal number. In the above example, $-3.14159 = -4 + 0.85841$, `floor()` throws 0.85841 away and only keeps the integral part -4.

`ceiling()` is a rounding function with the format of `ceiling(number)` that rounds the number to **the nearest integer that is greater than its value**.




```
ceiling(3.14159)
```

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

```
ceiling(-3.14159)
```

```
## [1] -3
```

3.4.5 `trunc()`

`trunc()` is a rounding function with the format of `trunc(number)` that **drops all digits after the decimal point**.

```
trunc(3.14159)
```

```
## [1] 3
```

```
trunc(-3.14159)
```

```
## [1] -3
```

3.5 Control Statements

Control statements are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to decide after assessing the variable.

There are 8 types of control statements in R:

- `if` condition
- `if-else` condition
- `for loop`
- nested loops
- `while loop`
- `repeat` and `break` statements
- `return` statement
- `next` statement

We will introduce `if/if-else` statements and `for/while` loops in this note and use them to implement the algorithm with the given pseudo-code given in the lecture note.

3.5.1 Conditional Statements

3.5.1.1 IF Statement

This control structure checks whether the expression provided in parenthesis is true or not. If true, the execution of the statements in braces `{}` continues. The syntax is given by

Syntax:

```
if(expression){
    statements
    ....
    ....
```

```
}
```

Example 1:

```
x <- 100                                  # numerical scalar
##
if(x > 10){                               # conditional statement
print(paste(x, "is greater than 10")) # paste() is a string function. one
                                          # can pass a numerical argument to
                                          # this string function.
}
```

```
## [1] "100 is greater than 10"
```

Example 2:

```
x <- pi                          # numerical scalar
##
if(x == 3.14159){                # conditional statement
print(paste(x, "is pi!"))
}
```

3.5.1.2 IF-ELSE Statement

If the **if condition** is determined as true, it will execute the statements written *inside the if block*. Otherwise, if the **if condition** is not satisfied, it will return the statements written inside the *else block*.

Syntax

```
if (condition) {# The statement will execute
                # if the condition is satisfied.
     statement_1
     ...........
   }
else { # The statement will execute if the condition
       # turns out to be not satisfied.
   statement_2
   ...........
 }
```

Example 3

```
x <- pi                          # numerical scalar
##
if(x == 3.14159){                # conditional statement
   print(paste(x, "is pi!"))
   } else {
   print(paste(x, "is NOT pi!"))  # Caution: x will be printed as a string!
  }
```

```
## [1] "3.14159265358979 is NOT pi!"
```

3.5.1.3 ELSE-IF Statement

It continuously checks certain conditions. If any of the if the condition is satisfied, it will return statements written inside the block. If none of the if the condition is true, it will execute the statements written inside the else block.

Syntax

```
if (condition){
     expression
  } else if (condition){
     expression
  } else if (condition){
     expression
  } else { # The statement will execute if none of the if
           # conditions turn out to be true.
    statement
}
```

Example 4

```
y = 10

if (x > y) {
    print(x)
} else if (y == x) {
    print("x and y are equal")
} else {
    print(y)
}
```

```
## [1] 10
```

3.5.2 Loops in R

A loop is used to perform repetitive tasks. There are three types of loops in R: for-loop, while-loop, and repeat-while-loop. These loops are used with `if-else` like conditional statements to execute statements conditionally.

3.5.2.1 FOR Loops

A **for-loop** will run statements a pre-set number of times n.

Syntax

```
 for (i in 1:n){
  executable statements
 }
```

Example 5

Find 10 factorial (10!). Note that $10! = 0 \times 1 \times 2 \times \cdots \times 9 \times 10$. We only need 10 iterations of the cumulative product to find the value.

```
result = 1  # initial value to start the process of iteration
for(i in 1:10){
  result = result * i
}
cat("\n 10! =", result,".\n")
```

```
##
##  10! = 3628800 .
```

`for-loop` is usually used jointly with other conditional statements defined by `if-else`, `break`, `next`, etc.

Example 6. Let X =c(10 18 42 46 11 29 15 22 20 8 49 47 1 16 23 13 32 27 19 37 31 43 39 24 44 12 3 2 9 7 50 4 30 6 45 38 17 14 26 36 21 41 40 5 33 34 25 28 35 48). We want to select 10 smallest odd numbers and store them in a vector.

```
# define the vector
 X=c(10, 18, 42, 46, 11, 29, 15, 22, 20, 8, 49, 47, 1, 16, 23, 13, 32, 27,
     19, 37, 31, 43, 39, 24, 44, 12, 3, 2, 9, 7, 50, 4, 30, 6, 45, 38, 17,
     14, 26, 36, 21, 41, 40, 5, 33, 34, 25, 28, 35, 48)
# Note that the largest of the first 10 smallest odd number is 19.
oddVec = NULL     # store the first 10 smallest odd numbers
k = 1             # initial index of oddVec
for (i in 1:100){
     xi = X[i]
     I = X[i]%%2         # %% gives the remainder of the division
     if(I == 0){
         next            # skip the current iteration and jump to the next
     } else{
        if(xi > 19) {
           next          # skip the current iteration and jump to the next
       } else{
           oddVec[k] = xi
           if (k == 10) break    # beak the iteration once all
                                 # desired odd numbers are selected!
           k = k + 1             # update the index for the next valid odd number
       }
     }
  }
oddVec
```

```
##  [1] 11 15  1 13 19  3  9  7 17  5
```

3.5.2.2 WHILE Loops

The while loop repeats statements as long as a certain condition is true. Stated another way, the while loop will stop when the condition is false (for example, the user types 0 to exit). Each time the loop starts running, it checks the condition.

See the example in the next section.

3.5.2.3 REPEAT-IF-WHILE Loops

This loop is equivalent to the **DO-WHILE** loop in other languages such as SAS and C++.

The `repeat loop` does not have any condition to **terminate** the loop. We need to put an **exit condition** implicitly with a **break statement** inside the loop.

Syntax:

```
repeat{
  statements...
  if(condition){
    break
  }
}
```

Example 6

```
i = 1
repeat {
  print(i)
  i = i + 1
  if(i >= 5 )
    break
}
```

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

3.6 Numerical Example

Example 4 (Numerical Implementation): The nth Taylor polynomial for $f(x) = e^x$ expanded about $x_0 = 0$ is

$$P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

and the value of e to six decimal places is 2.718282. **Construct an algorithm** to determine the **minimal value of** n required for

$$|e - P_n(1)| < 10^{-5},$$

without using the Taylor polynomial remainder term.

Solution: The objective is to determine the degrees of the Taylor polynomial evaluated at $x = 1$ to approximate e. The input values are (1) tolerance TOL and the initial degree of the Taylor polynomial. The output is the smallest degree of the Taylor polynomial that meets $|e - P_n(1)| < TOL$.

Caution: In general, one should consider including two stopping rules: error tolerance and maximum iterations. In this particular example, the maximum number of iterations is simply the solution to the problem. Therefore, there is one stopping rule: TOL

3.6.1 VERSION 1

This is modified from the pseudo-code from the example in the lecture note.

```
INPUT  initial degree: n,
            tolerance: TOL,

OUTPUT the desired degree N of the polynomial

Step 1. SET   n = 0;
              Pn = 1;
             ERR = exp(1) - 1;
Step 2. WHILE ERR >= TOL DO:
        1.    n = n + 1
              Pn = Pn + 1/n!          # n! = n factorial
             ERR = exp(1) - Pn        # exp(1) = 2.718282
        2. IF |ERR| < TOL DO:
              OUTPUT (N)
              WRITE (tolerance achieved!)
              STOP
           ELSE DO:                 # optional: print out something
```

```
                                # to monitor the iterative process
            WRITE(iteration number)
          ENDIF
        ENDWHILE
```

The following is the R code for implementing the above pseudo-code.

```
 n = 0
 TOL = 10^(-5)
 Pn = 1
 ERR = exp(1) - Pn
# Loop Starts
while(ERR >= TOL){
  n = n + 1
  Pn = Pn + 1/factorial(n)
  ERR = exp(1) - Pn
  if(ERR < TOL){
  cat("\n\n The desired degrees of the Taylor polynomial is ", n+1,". \n")
  cat("Tolerance achieved!\n\n\n")
  } else{
  cat("\n Iteration number: ", n, ". ")
  }
 }
```

```
##
##  Iteration number:  1 .
##  Iteration number:  2 .
##  Iteration number:  3 .
##  Iteration number:  4 .
##  Iteration number:  5 .
##  Iteration number:  6 .
##  Iteration number:  7 .
##
##  The desired degrees of the Taylor polynomial is  9 .
## Tolerance achieved!
```

3.6.2 Version 2

Change the order for updating the degrees of the polynomial and add additional outputs

```
INPUT  initial degree: n,
            tolerance: TOL,

OUTPUT the desired degree N of the polynomial
```

```
Step 1. SET   n = 0;
             Pn = 0;
            ERR = exp(1);
Step 2. WHILE ERR >= TOL DO:
        1.  Pn = Pn + 1/n!          # n! = n factorial
           ERR = exp(1) - Pn        # exp(1) = 2.718282
        2. IF |ERR| < TOL DO:
               OUTPUT (N, Pn, absolute error)
               WRITE (tolerance achieved!)
               STOP
           ELSE DO:                 # optional: print out something
                                    # to monitor the iterative process
               WRITE (iteration number, absolute error)
               n = n + 1
          ENDIF
        ENDWHILE
```

The following is the R code for implementing the above pseudo-code.

```r
 n = 0
 TOL = 10^(-5)
 Pn = 0
 ERR = exp(1)
# Loop Starts
while(ERR >= TOL){
  Pn = Pn + 1/factorial(n)
  ERR = exp(1) - Pn
  if(ERR < TOL){
     cat("\n\nTolerance achieved!\n\n")
     cat("\n degree N = ", n+1, ".")
     cat("\n e = ", exp(1), ".")
     cat("\n Pn =", Pn, ".")
     cat("\n Absolute Error =", ERR, ".")
  } else{
          n = n + 1
     cat("\n Iteration numer: ", n, ". Absolute Error = ", ERR, ".")
  }
 }
```

```
##
##  Iteration numer:  1 . Absolute Error =  1.718282 .
##  Iteration numer:  2 . Absolute Error =  0.7182818 .
##  Iteration numer:  3 . Absolute Error =  0.2182818 .
```

```
##  Iteration numer:  4 . Absolute Error =  0.05161516 .
##  Iteration numer:  5 . Absolute Error =  0.009948495 .
##  Iteration numer:  6 . Absolute Error =  0.001615162 .
##  Iteration numer:  7 . Absolute Error =  0.0002262729 .
##  Iteration numer:  8 . Absolute Error =  2.786021e-05 .
##
## Tolerance achieved!
##
##
##  degree N =  9 .
##  e =  2.718282 .
##  Pn = 2.718279 .
##  Absolute Error = 3.058618e-06 .
```

3.7 User Defined Functions

One of the great strengths of R is that it allows users to extend the capacity of R through user-defined functions. Functions are often used to encapsulate a sequence of expressions that need to be executed repetitively. We have made code to implement bisection and fixed-point methods by specific examples. We may want to write the R function of these implementations so we can use them for other root-finding problems with the appropriate input information.

The syntax of the general user-defined R function has the following form

```
myfunction <- function(arg1, arg2, ... ){
   statements
   return(object)
}
```

`arg1, arg2, ...` are arguments that are passed into the function. The arguments could be any objects such as vectors and other user-defined R functions.

Example 1 Finding the standard deviation of an input vector.

```
sdfun <- function(x) {
  res <- sqrt(sum((x - mean(x))^2) / (length(x) - 1))
  return(res)
}
##
vec = c(1,4,2,6,-3, -5)
sdfun(x = vec)   # or simply sdfun(vec)
```

```
## [1] 4.167333
```

We have defined functions in implementing root-finding methods.

3.7.1 Writing R Functions for Root-finding Methods

As an example, we will write an R function to implement the bisection method for finding the root for any given equation on ver an interval (if it exists). Two versions of R functions will be presented in the following.

3.7.1.1 Function with Numerical Outputs

We simply wrap up the example code in the lecture note to make the following function.

```
num.FixedPoint = function(fn,        # function that satisfies f(x) = x
                          a,         # lower limit of the interval [a, b]
                          b,         # upper limit of the interval [a, b]
                          TOL,       # error tolerance
                          N,         # maximum number of iterations
                          x0,        # initial value of x
                          detail,     # intermediate output
                          ...){
 gfun = fn
 x = x0
 ERR = Inf
 n = 0
##
 while (ERR > TOL){
  n = n + 1
  new.x = gfun(x)
  ERR = abs(new.x - x)
  if(ERR < TOL){
    cat("\n\nThe algorithm converges!")
    cat("\nThe approximate root is:", new.x,".")
    cat("\nThe absolute error is:", ERR, ".")
    cat("\nThe number of iterations is:", n, ".")
    break
  } else{
    if(ERR > 10^7){
        cat("\n\nThe algorithm diverges!")
        break
    } else{
         if(detail == TRUE){
            cat("\nIteration:",n,". Estimated root:", new.x, ". Absolute error:", ERR,"
          }
         x = new.x          # update x value!!!
    }
  }
  if(n == N){
    cat("\n\nThe maximum number of iterations is achieved!")
```

```
    break
  }
 }
}
```

Next, we use several examples.

Example 1 Using the fixed-point method to find the approximate root of $x^3 - 7x + 2 = 0$. To use the fixed-point method, we rewrite the equation into the form $(x^3 + 2)/7 = x$. Then $g(x) = (x^3 + 2)/7$ will be the function to be passed into the function.

```
###
fun0 = function(x) (x^3 + 2)/7
num.FixedPoint(fn = fun0, a = 0, b = 2, TOL = 10^(-6), N = 200, x0 =1.5, detail = TRUE)
```

```
##
## Iteration: 1 . Estimated root: 0.7678571 . Absolute error: 0.7321429 .
## Iteration: 2 . Estimated root: 0.3503903 . Absolute error: 0.4174668 .
## Iteration: 3 . Estimated root: 0.2918598 . Absolute error: 0.0585305 .
## Iteration: 4 . Estimated root: 0.2892659 . Absolute error: 0.002593907 .
## Iteration: 5 . Estimated root: 0.289172 . Absolute error: 9.385572e-05 .
## Iteration: 6 . Estimated root: 0.2891687 . Absolute error: 3.364631e-06 .
##
## The algorithm converges!
## The approximate root is: 0.2891686 .
## The absolute error is: 1.20578e-07 .
## The number of iterations is: 7 .
```

Example 2: Calculate $1/\sqrt{2}$ by using the fixed-point method. Note that $f(x) = x^2 - 1/2$. To use the fixed point method, we need $g(x) = x - x^2 + 1/2$ as the input function.

```
###
fun0 = function(x) x - x^3 + 1/2
num.FixedPoint(fn = fun0, a = -1, b = 2, TOL = 10^(-5), N = 200, x0 =0.7, detail =FALSE)
```

```
##
##
## The algorithm converges!
## The approximate root is: 0.7937048 .
## The absolute error is: 9.128962e-06 .
## The number of iterations is: 83 .
```

3.7.1.2 Function with More Optional Outputs

We include graphics the function created previously.

```
FixedPointAlgR = function(fn,       # function that satisfies f(x) = x
                          a,        # lower limit of the interval [a, b]
                          b,        # upper limit of the interval [a, b]
                          TOL,      # error tolerance
                          N,        # maximum number of iterations
                          x0,       # initial value of x
                          detail=TRUE,   # intermediate numerical outputs
                          graphic=TRUE,  # intermediate graphic outputs
                          x.lim,    # x-axis of graphic window
                          y.lim,    # y-axis of graphic window
                          sleep,    # sleep time between iterations
                          ...){
 gfun = fn      # rename the input function used in the function
 x = x0         # rename the initial value used in the function
 ERR = Inf      # initial error (a big  number to start the iteration)
 n = 0          # initial iteration counter - iterator or index
##
 if(graphic==TRUE){
 xlimit=c(a-0.1*abs(b-a), b+0.1*abs(b-a))               # x-limit in the graphic windo
 xx=seq(a-0.1*abs(b-a), b+0.1*abs(b-a), length = 2000)  # sequence of x values
 yy1 = gfun(xx)                                         # evaluate g(x)
 yy2 = xx                                               # evaluate diagonal
 plot(xx, yy1, ylim = y.lim, xlim = x.lim, type = "l", lwd = 2, lty = 1, col = "blue")
 lines(xx,yy2, lwd = 2, lty = 1, col ="darkred")        # add line y = x
 title("Fixed Point Algorithm Approximation")           # add title to the graphic
 }
##
 while (ERR > TOL){
  Sys.sleep(sleep)                                      # put the system to sleep
  n = n + 1
  new.x = gfun(x)
  ERR = abs(new.x - x)
  if(ERR < TOL){
    cat("\n\nThe algorithm converges!")
    cat("\nThe approximate root is:", new.x,".")
    cat("\nThe absolute error is:", ERR, ".")
    cat("\nThe number of iterations is:", n, ".")
    break
  } else{
    if(ERR > 10^7){
        cat("\n\nThe algorithm diverges!")
        break
    } else{
         if(detail == TRUE){
            cat("\nIteration:",n,". Estimated root:", new.x, ". Absolute error:", ERR,"
```

```
        }
       if(graphic==TRUE){
           segments(new.x, new.x,       new.x,       gfun(new.x), col="purple")
           segments(new.x, gfun(new.x), gfun(new.x), gfun(new.x), col="purple")
         }

        x = new.x          # update x value!!!
    }
  }
  if(n == N){              # checking iteration limit
    cat("\n\nThe maximum number of iterations is achieved!")
    break
  }
 }
}
```

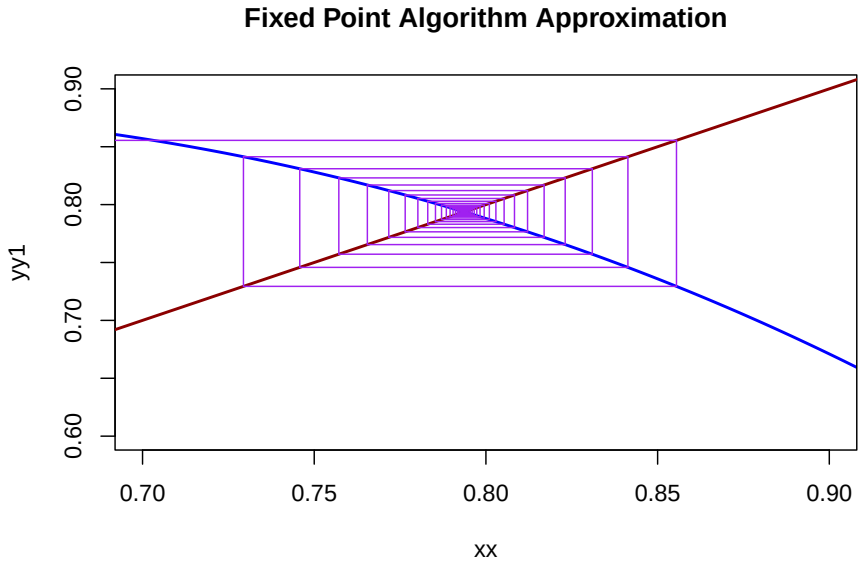
Example 1 (Revisited)

```
###
fun0 = function(x) (x^3 + 2)/7
FixedPointAlgR(fn = fun0, a = 0, b = 1, TOL = 10^(-6), N = 200,
               x0 =0.35, detail = TRUE, graphic = FALSE,
               x.lim=c(0.285, 0.294), y.lim=c(0.28, 0.32), sleep = 0.1)
```

```
##
## Iteration: 1 . Estimated root: 0.2918393 . Absolute error: 0.05816071 .
## Iteration: 2 . Estimated root: 0.2892651 . Absolute error: 0.002574143 .
## Iteration: 3 . Estimated root: 0.289172 . Absolute error: 9.313374e-05 .
## Iteration: 4 . Estimated root: 0.2891687 . Absolute error: 3.33874e-06 .
##
## The algorithm converges!
## The approximate root is: 0.2891686 .
## The absolute error is: 1.196502e-07 .
## The number of iterations is: 5 .
```

Example 2 (Revisited)

```
###
fun0 = function(x) x - x^3 + 1/2
FixedPointAlgR(fn = fun0, a = -1, b = 2, TOL = 10^(-5), N = 200,
               x0 =-1.2, detail =FALSE, graphic = TRUE,
               x.lim=c(0.7, 0.9), y.lim=c(0.6, 0.9), sleep = 0.1)
```




```
##
##
## The algorithm converges!
## The approximate root is: 0.7937047 .
## The absolute error is: 8.942928e-06 .
## The number of iterations is: 85 .
```

Chapter 4

Basics of Data Representation and Error Analysis

This note introduces briefly the concepts of floating point and the basics of error analysis. Before presenting these concepts, we introduce some computer architecture concepts from computer science.

Bit is the fundamental unit of memory inside a computer, which is short for **binary digit**. **Each bit** of data corresponds to a '0' or '1'.

The bit is the smallest unit of data that a computer processes, but a single bit are too small to represent much data.

The smallest practical unit for expressing information is the byte, which is made up of eight bits.

A single byte consists of **eight bits**. That is, a single byte can represent 256 ($= 2^8$) combinations of data.

4.1 Data Representation

Data is a broad concept. All recorded information is called data. We briefly describe how data is represented and stored in computer systems. In this numerical analysis class, we focus on how numbers are represented and stored in computers.

4.1.1 Number Systems

We use a decimal system (base 10) for counting and computations. Computers use binary (base 2) number systems, as they are made from binary digital components (known as transistors) operating in two states - on (encoded as 1) and off (encoded as 0). In computing, the hexadecimal (base 16) or octal (base 8) number systems are also used as a compact form for representing binary numbers. Next, we briefly outline decimal, binary, and hexadecimal number systems. Every representation has three components: **digits**, **base**, and **exponent**.

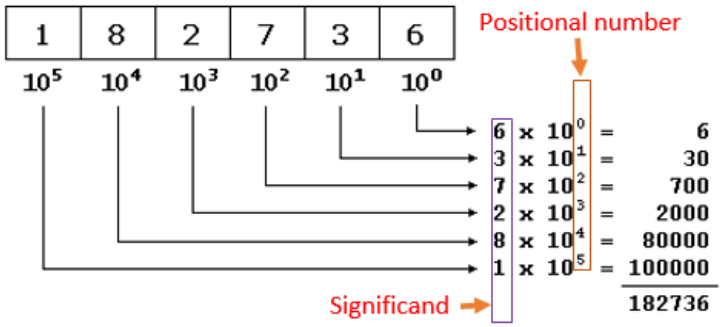
4.1.1.1 Decimal (Base 10) System

The decimal number system has ten symbols: $0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9, called **digits**. It uses positional notation. That is, the least-significant digit (right-most digit) is of the order of 10^0 (units or ones), the second right-most digit is of the order of 10^1 (tens), the third right-most digit is of the order of 10^2 (hundreds), and so on. The exponents of the base are called **positional numbers**.

Example 1: The base 10 representation of integer 182736 is given the following form

$$182736 = 1 \times 10^5 + 8 \times 10^4 + 2 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 6 \times 10^0.$$

The following figure explains the above representation.



To avoid confusion, we use $182736D$ or 182736_{10} to denote 182736 to be a decimal number in case multiple number systems are used at the same time.

4.1.1.2 Binary (base 2) System

The binary number system is used by all computers. The binary number system is a base-2 number system, therefore there are two valid digits: 0 and 1. The

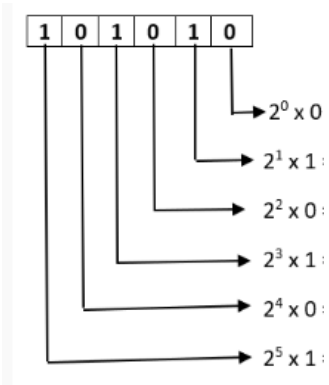
binary number system has two symbols: 0 and 1, called bits. It is also a positional notation.

Example 2: Write the base 2 representation of binary integer 101010B.

Solution The base 2 representation is given in the following.

$$101010B = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

where suffix B denotes the binary number. The following figure explains the above representation.



4.1.1.3 Conversion Between Number Systems

Computers use the binary system to store numbers. How to convert a number from one system to the other? The following examples illustrate the idea of conversion.

Example 3 Convert 12045D to a binary number.

Solution: All we need to so is to represent 205D in a power series with base 2 in the following

$$205D = 2^7D + 2^6D + 2^3D + 2^2D + 1D$$

$$= 1\times2^7D+1\times2^6D+0\times2^5D+0\times2^4D+1\times2^3D+1\times2^2D+0\times2^1D+1\times2^0D = 11001101B$$

Example 4: Convert 101011B into a decimal system.

Solution: Use the same idea in the previous example to complete the conversion.

$$101011B = 1 \times 2^5D + 0 \times 2^4D + 1 \times 2^3D + 0 \times 2^2D + 1 \times 2^1D + 1 \times 2^0$$

$$= 32D + 0D + 8D + 0D + 2D + 1D = 43D.$$

Example 5: Express 3.25D (floating value) into a binary (floating) value.

Solution: We represent the integral and decimal parts into binary numbers respectively.

- **Integral part**: $3_{10} = (1 \times 2^1 + 1 \times 2^0)_{10} = (11)_2$.
- **Decimal part**: $.25_{10} = (0 \times 2^{-1} + 1 \times 2^{-2} = (.01)_2$.

Therefore, $3.25_{10} = 11.01_2$.

Comment: converting the fractional part to binary (base 2 representation) is a little cumbersome. The following is a programmatically convenient tabular algorithm:

Fractional part		Binary digits	Positional number
0.25 x 2 = 0.5	0.5 < 1	0	2^{-1}
0.5 x 2 = 1	1 ≥ 1, 1 − 1 = 0, stop!	1	2^{-2}
			2^{-3}

Example 6 Express 21.36D (decimal representation) into a binary (representation) value.

Solution: We still convert the integral and fraction parts separately.

- **Integral Part**: $21 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (10101)_2$.
- **Fractional Part**: The binary conversion of 0.36 is based on the following table.

Fractional part		Binary digits	Positional number
0.36 x 2 = 0.72	0.72 < 1	0	2^{-1}
0.72 x 2 = 1.44	1.44 ≥ 1, 1.44 − 1 = 0.44	1	2^{-2}
0.44 x 2 = 0.88	0.88 < 1	0	2^{-3}
0.88 x 2 = 1.76	1.76 ≥ 1, 1.76 − 1 = 0.76	1	2^{-4}
0.76 x 2 = 1.52	1.52 ≥ 1, 1.52 − 1 = 0.52	1	2^{-5}
0.52 x 2 = 1.04	1.04 ≥ 1, 1.04 − 1 = 0.04	1	2^{-6}
0.04 x 2 = 0.08	0.08 < 1	0	2^{-7}
0.08 x 2 = 0.16	0.16 < 1	0	2^{-8}
0.16 x 2 = 0.32	0.32 < 1	0	2^{-9}
0.32 x 2 = 0.64	0.64 < 1	0	2^{-10}
0.64 x 2 = 1.28	1.28 ≥ 1, 1.28 − 1 = 0.28	1	2^{-11}
0.28 x 2 = 0.56	0.56 < 1	0	2^{-12}
0.56 x 2 = 1.12	1.12 ≥ 1, 1.12 − 1 = 0.12	1	2^{-13}
0.12 x 2 = 0.24	0.24 < 1	0	2^{-14}
0.24x 2 = 0.48	0.48 < 1	0	2^{-15}
0.48 x 2 = 0.96	0.96 < 1	0	2^{-16}
0.96 x 2 = 1.92	1.92 ≥ 1, 1.92 − 1 = 0.92	1	2^{-17}
0.92 x 2 = 1.84	1.84 ≥ 1, 1.84 − 1 = 0.84	1	2^{-18}
0.84 x 2 = 1.68	1.68 ≥ 1, 1.68 − 1 = 0.68	1	2^{-19}
0.68 x 2 = 1.36	1.36 ≥ 1, 1.36 − 1 = **0.36**	1	2^{-20}
	Binary	**0**	2^{-21}
	digit	**1**	2^{-22}
	pattern	**0**	2^{-23}
	starts	**1**	2^{-24}
	repeating	**1**	2^{-25}
		...	...

From the table we have

$$0.36_{10} = (\overbrace{01011100001010001111}^{20\text{ digits}}\cdots\overbrace{01011100001010001111}^{20\text{ digits}}\cdots)_2$$

That is, the resultant binary number has infinite digits because the sequence of digits (01011100001010001111) will repeat infinitely many times.

Therefore, $21.36_{10} = 10101.\overline{01011100001010001111}_2$.

Remark: This examples shows that 0.36 has infinite representations in binary. The bits go on forever; no matter how many of those bits you store in a computer, you will never end up with the binary equivalent of decimal 0.36.

4.1.1.4 Scientific Notation

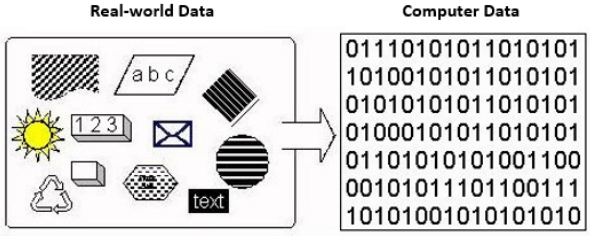
We have outlined the base-10 and base-2 representation of given numbers. We also use **scientific notation** in base-10 representation.

Example 7: Write the scientific notation of 21.36_{10} and its binary representation (see the second part of example 6).

Solution: The scientific notation of 21.36_{10} denoted by $21.36 = 2.136 \times 10^1$ or written as $2.136E1$. In binary representation, $21.36_{10} = 10101.\overline{0101110000101000111 1}_2 = 1.0101010111000010100011111_2 \times 2^4$

4.2 How data is stored in computer memory

Human beings use decimal (base 10) and duodecimal (base 12) number systems for counting and measurements (probably because we have 10 fingers and two big toes). Computers use binary (base 2) number systems, as they are made from binary digital components (known as transistors) operating in two states - **on** and **off**.

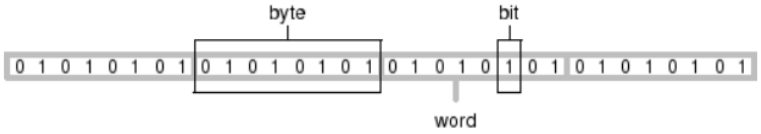


We will not go into details about how all kinds of data are stored in a computer. But it is necessary to have some conceptual understanding of how numbers are stored in a computer so that we will better understand various sources of errors.

Computer memory is the collection of many groups that contain a certain number of bits. As mentioned earlier,

- a collection of 8 bits is called a byte, and
- a collection of 4 bytes, or 32 bits, is called a word.

Each individual data value in a data set is usually stored using one or more bytes of memory, but at the lowest level, any data stored on a computer is just a large collection of bits. The following is an illustrative example that explains how a number is stored using a 32-bit layout (also called **bit string**) memory.



4.2.1 How integers are stored in memory?

The most computer system uses 32-bit (or 64-bit) layout to store integers. The previous figure is a 32-bit string layout. The binary integer stored is

$$(\overbrace{01010101010101010101010101010101}^{32\text{ digits}})_2 = (2^{30}+2^{28}+2^{26}+\cdots+2^2+2^0)_{10} = 1431655765_{10}$$

If we store only unsigned integers, the biggest integer we store in the 32-bit layout is $2^{32}-1 = 4294967295$. The smallest integer is 0. If we consider both positive and positive integers, the use one bit (utmost left-hand side of the bit string) to denote the sign of the integer (0 indicates positive and 1 indicates negative). Then the smallest and the biggest integers that can be stored are $-2^{31}-1 = -2147483647$ and $2^{31}-1 = +2147483647$.

4.2.2 Floating-point Numbers

We first describe the concepts of real numbers and floating-point numbers.

- **Floating-point numbers** are any numbers with a decimal point in them. For example, 3.14159 is a floating-point number.
- **A real number** is the "analog" version of a floating-point number – It **has infinite precision** and **is continuous**.
- The **difference** between *reals* and *floating-point numbers* is that the latter has limited precision – it really depends on the abilities of the computer. **Floating-point numbers are essentially approximations of real numbers**.

Example 8: Let's binary representation and floating-point representation of real number 0.1_{10}.

Answer: Using the same method used in **example 7** to represent 0.1_{10}, we have

$$0.1_{10} = 0.0\,\underbrace{\overline{0011}}_{4\text{ digits}}{}_2$$

It has infinite representation in binary representation. When it is stored in a computer system, only finite binary digits are used due to the precision of the computer.

Example 9: Find the difference $3_{10} \times 0.1_{10} - 0.3_{10}$.

Answer: The difference is 0 if calculated manually. We have shown that 0.1_{10} has an infinite binary representation. However, $0.3_{10} = (0.01001100110)_2$. When calculating the above difference in a computer, it will give a non-zero number. The result from **R** in the following

```
3*0.1 - 0.3
```

```
## [1] 5.551115e-17
```

4.2.3 How Floating-point Numbers Are Stored in Memory?

IEEE 754-1985 represents numbers in binary, providing definitions for four levels of precision, of which the two most commonly used are single (32-bit) and double precision (64-bit) layouts.

Level	Width	Range at full	Precision
Single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	Approximately 7 decimal digits
Double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	Approximately 16 decimal digits

The equation for converting a 32-bit pattern into a floating point number is given by

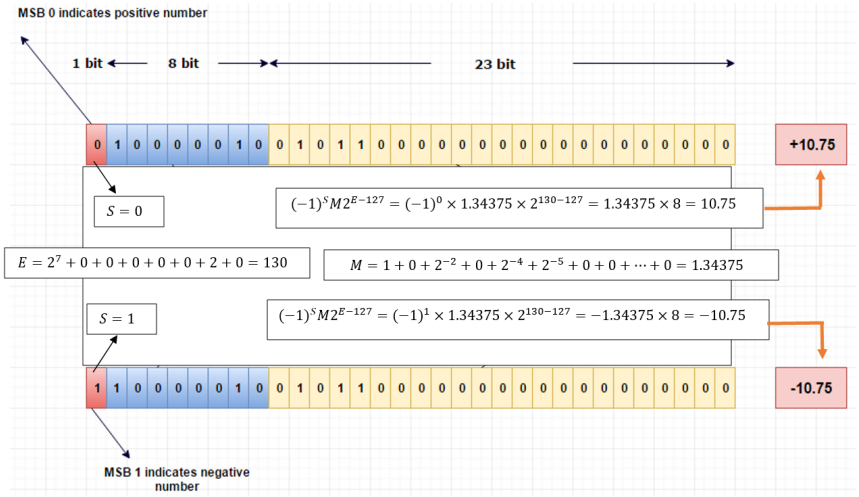
$$(-1)^S \times M \times 2^{E-127}$$

where S is the value of the *sign bit*, M is the value of the *mantissa* (fractional part), and E is the value of component. In 32-bit layout, 23 bits are allocated to *mantissa*. The general form of M is explicitly by

$$M_{10} = 1 + m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2^{-3} + \cdots + m_1 2^{-22}$$

where m_i $(i = 0, 1, 2, \cdots, 22$ is either 1 or 0. Note that -127 in the conversion formula reflects the total number of alternatives based on the allocated 8 bits $(2^8 - 1 = 127)$.

The examples given below show how $\pm 10.75_{10}$ are stored in a 32-bit layout system.



Of the 64 bits used to store a double precision number, bits 0 through 51 are the *mantissa*, bits 52 through 62 are the *exponent*, and bit 63 is the *sign* bit.

The main take-away is that storing real number to a computer system involves potential rounding error (due to approximation)!

4.2.4 Basic Types Errors in Error Analysis

Let P is the true value and P^* an approximated value. Three types of errors will be used from time to time throughout the semester.

- The **actual error** is defined to be $P - P^*$.
- The **absolute error** is defined to be $|P - P^*|$
- The **relative error** is defined to be $|P - P^*|/P$, $p \neq 0$. The relative errors are frequently used in error analysis throughout this semester.
- **Significant digit**:

The number P^* is said to approximate P to t **significant digits** if t is the largest non-negative number such that

$$\frac{|P - P^*|}{P} < 1.23 \times 10^{-t}$$

Chapter 5

Error Analysis, Algorithms and Convergence

We briefly introduce sources of errors in numerical analysis and concepts of convergence of algorithms.

5.1 Error Analysis

We briefly outline the sources and types of errors.

5.1.1 Understanding Numerical Error

We have seen that every computerized representation of real numbers with fractional parts is forced to employ rounding and other approximations. Rounding, however, represents one of many sources of error in numerical systems.

Rounding or truncation error comes from rounding and other approximations used to deal with the fact that we can only represent a finite set of values using most computational number systems. ** For example,**, it is impossible to write π exactly as an IEEE 754 floating-point value, so in practice, its value is truncated after a finite number of digits.

Discretization error comes from our computerized adaptations of calculus, physics, and other aspects of continuous mathematics. **For example**, a numerical system might attempt to approximate the derivative of a function $f(t)$ using divided differences:

$$f'(t) \approx \frac{f(t+\epsilon) - f(t)}{\epsilon}$$

for some fixed choice of ϵ. We must use a finite $\epsilon > 0$ rather than taking a limit

as $\epsilon \to 0$, the resulting value for $f'(t)$ is only accurate to some number of digits. This results in a discretization error.

Modeling These errors arise during the modeling process when scientists ignore effecting factors in the model to simplify the problem. Also, these errors are known as formulation errors.

5.1.2 Error Classification

Let $P = 1.354595$ is the true value and $P^* = 1.354675$ an approximated value. Three types of errors will be used from time to time throughout the semester.

- The **actual error** is defined to be $P - P^* = 1.354595 - 1.354675 = -0.00008 = -8 \cdot 10^{-5}$.
- The **absolute error** is defined to be $|P - P^*| = |1.354595 - 1.354675| = 0.00008 = 8 \cdot 10^{-5}$
- The **relative error** is generally defined to be $|P - P^*|/P$, $p \neq 0$. With the given values of P and P^*, we have

$$\frac{|P - P^*|}{|P|} = \frac{|1.354595 - 1.354675|}{|1.354595|} = 5.905475 \times 10^{-05}$$

The relative errors are frequently used in error analysis throughout this semester.

5.1.3 Approximation Significant Digits (Figures)

The number P^* is said to approximate P to t **significant digits** if t is the **largest non-negative number** such that

$$\frac{|P - P^*|}{|P|} < 5 \times 10^{-t}$$

Example 1: We still use $P = 1.354595$ as the true value and $P^* = 1.354675$ an approximated value. We have calculated

$$\frac{|P - P^*|}{|P|} = \frac{|1.354595 - 1.354675|}{|1.354595|} = 5.905475 \times 10^{-05}$$

Since $5 \times 10^{-05} < 5.905475 \times 10^{-05} < 5 \times 10^{-04}$. Therefore, by the definition, the approximation significant digit is 4.

Example 2 (refer to *Example 5* of the textbook, page 25.) Let $p = 0.54617$ and $q = 0.54601$. Use four-digit arithmetic to approximate $p - q$ and determine the absolute and relative errors using (a) rounding and (b) chopping.

Solution: The exact value of $r = p - q$ is $r = 0.00016$.

(a): Suppose the subtraction is performed using four-digit rounding arithmetic. Rounding p and q to four digits gives $p^* = 0.5462$ and $q^* = 0.5460$, respectively, and $r^*p^* - q^* = 0.0002$ is the four-digit approximation to r. Since

$$\frac{|r - r^*|}{|r|} = \frac{|0.00016 - 0.0002|}{|0.00016|} = 0.25 = 2.5 \times 10^{-1} < 5 \times 10^{-1}.$$

By the definition, the result has only **one significant digit**, whereas p^* and q^* were accurate to four and five significant digits, respectively.

(b). If chopping is used to obtain the four digits, the four-digit approximations to p, q, and r are $p^* = 0.5461$, $q^* = 0.5460$, and $r^* = p^* - q^* = 0.0001$. This gives

$$\frac{|r - r^*|}{|r|} = \frac{|0.00016 - 0.0001|}{|0.00016|} = 0.375 = 3.75 \times 10^{-1} < 5 \times 10^{-1}.$$

which also results in only one significant digit of accuracy.

5.1.4 Nested Arithmetic

Accuracy loss **due to round-off error** can also be reduced by rearranging calculations, as shown in the next example.

Example 3: [examples 6 and 7 of the textbook, pages 25-27]. Consider polynomial $f(x) = x^3 - 6.1x^2 + 3.2x + 1.5$. Evaluate $f(x)$ at $= x = 4.71$ using three-digit arithmetic.

Solution: We calculate approximations

- Exact value: $f(x) = 4.71^3 - 6.1 \times 4.71^2 + 3.2 \times 4.71 + 1.5 = -14.263899$
- Rounding term-wise: $f(x) = 104 - 135 + 15.1 + 1.5 = -14.4$
- Nest rounding: $f(x) = ((4.71 - 6.1) \times 4.71 + 3.2) \times 4.71 + 1.5 = (-6.54 + 3.2) \times 4.71 + 1.5 = -15.7 + 1.5 = 14.2$.

The above evaluation can be done using the following R code.

```
x = 4.71
f.exact = x^3 - 6.1* x^2 + 3.2*x + 1.5
f.3digit = signif(x^3,3) - signif(6.1* x^2,3) + signif(3.2*x,3) + 1.5
f.3digit.nest = signif((signif((x-6.1)*x,3) + 3.2)*x,3) + 1.5
cbind(f.exact=f.exact, f.3digit=f.3digit, f.3digit.nest=f.3digit.nest)
```

```
##       f.exact f.3digit f.3digit.nest
## [1,] -14.2639    -14.4         -14.3
```

The relative errors of the two different approximations are

$$\text{Non-nest approximation} = \frac{|-14.2639 - (-14.4)|}{|-14.2639|} \approx 0.00954157$$

$$\text{Nest approximation} = \frac{|-14.2639-(-14.3)|}{|-14.2639|} \approx 0.002530865$$

Therefore, the nested arithmetic yields less approximation error.

Polynomials should always be expressed in the nested form before performing an evaluation because this form minimizes the number of arithmetic calculations. The error in the Illustration is due to the reduction in computations from four multiplications and three additions to two multiplications and three additions. One way to reduce round-off error is to reduce the number of computations.

5.2 Algorithms and Convergence

The objective of numerical analysis is to solve continuous problems using numeric approximation but accurate numeric solutions. Numerical methods (or algorithms) are used in cases where the exact solution is impossible or prohibitively expensive to calculate.

5.2.1 Algorithm and Psuedo-code

An algorithm is a procedure that describes, in an **unambiguous** manner, a **finite** sequence of steps to be performed in a specified order. The object of the algorithm is to implement a procedure to solve a problem or approximate a solution to the problem.

Pseudo-code is a programmatic description of an algorithm that does not require any strict programming language syntax or underlying technology considerations. It is a rough draft of a program. Pseudo-code summarizes a program's flow, which is something like

Algorithm 1: Newton Method

1 Set an initial guess x_0;
2 Set the number of steps n_{max};
3 Set $n = 1$;
4 **do**
5 $\quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$;
6 $\quad n = n + 1$;
7 **while** $n \leq n_{max}$;

5.2.2 Rules of Pseudo-code

The purpose of using pseudo-code is an efficient key principle of an algorithm. It is used in planning an algorithm by sketching out the structure of the program before the actual coding takes place. The basic rules of writing pseudo-code are

Rule 1: *Write only one statement per line*

Each statement in the pseudo-code should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudo-code

Rule 2: *Capitalize initial keyword*

There are just a few keywords we will use: **WRITE, OUTPUT, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL**

Rule 3: *Indent to show hierarchy*

We will use a particular indentation pattern in each of the design structures:

SEQUENCE: keep statements that are "stacked" in sequence all starting in the same column.

SELECTION: indent the statements that fall inside the selection structure, but not the keywords that form the selection.

LOOPING: indent the statements that fall inside the loop, but not the keywords that form the loop.

Rule 4: *End multi-line structures*

See how the **IF/ELSE/ENDIF** is constructed in the next example. The **ENDIF** (or END whatever) always is in line with the IF (or whatever starts the structure).

Rule 5: *Keep statements language independent*

Pseudo-code should not be tied to any programming language. It can be implemented in any language, whether it's C++, Java, Python, MATLAB, R, or any other programming language.

Example 4: The nth Taylor polynomial for $f(x) = e^x$ expanded about $x_0 = 0$ is

$$P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

and the value of e to six decimal places is 2.718282. **Construct an algorithm** to determine the **minimal value of** n required for

$$|e - P_n(1)| < 10^{-5},$$

without using the Taylor polynomial remainder term.

Solution: The objective is to determine the degrees of the Taylor polynomial evaluated at $x = 1$ to approximate e. The input values are (1) tolerance TOL

and the initial degree of the Taylor polynomial. The output is the smallest degree of the Taylor polynomial that meets $|e - P_n(1)| < TOL$.

Caution: In general, one should consider to include two stopping rules: error tolerance and maximum iterations. In this particular example, the maximum number of iterations is simply the solution of the problem. Therefore, there is one stopping rule: TOL

```
INPUT  initial degree: n,
           tolerance: TOL,

OUTPUT the desired degree N of the polynomial

Step 1. SET   n = 0;
           SUM = 0;
           ERR = exp(1);
Step 2. WHILE ERR > TOL DO:
        1. SUM = SUM + 1/n!         # n! = n factorial
           ERR = exp(1) - SUM       # exp(1) = 2.718282
        2. IF |ERR| < TOL DO:
              OUTPUT (N)
              WRITE (tolerance achieved!)
              STOP
           ELSE DO:
              n = n + 1
          ENDIF
        ENDWHILE
```

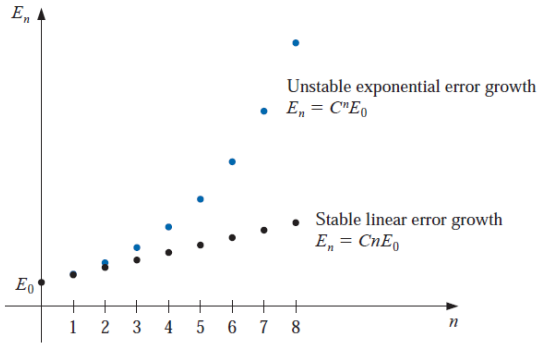
5.2.3 Stability of Algorithms

Roughly speaking, the stability of an algorithm measures how good the algorithm is at solving problems to achievable accuracy. In practice, there could have several algorithms for solving one problem and some algorithms are better than others. Those algorithms that get unnecessarily inaccurate answers are called unstable.

To further consider the subject of round-off error growth and its connection to algorithm stability, suppose an error with magnitude $E_0 > 0$ is introduced at some stage in the calculations and that the magnitude of the error after n subsequent operations is denoted by E_n. The two cases that arise most often in practice are defined as follows.

Growth of Error: Suppose that $E_0 > 0$ denotes an error introduced at some stage in the calculations and E_n represents the magnitude of the error after n subsequent operations.

- If $E_n \approx CnE_0$, where C is a constant independent of n, then the growth of error is said to be **linear**.
- If $En \approx C^n E_0$, for some $C > 1$, then the growth of error is called **exponential**.



5.2.4 Rates of Convergence

Since iterative techniques involving sequences are often used, this section concludes with a brief discussion of some terminology used to describe the rate at which **convergence** occurs. In general, we would like the technique to converge as rapidly as possible. The following definition is used to compare the convergence rates of sequences.

Convergence Sequence: Suppose $\{\beta_n\}_{n=1}^{\infty}$ is a sequence known to converge to zero, and $\{\alpha_n\}_{n=1}^{\infty}$ converges to a number α. If a positive constant K exists with

$$|\alpha_n - \alpha| \leq K|\beta_n|, \text{ for large } n,$$

then we say that $\{\alpha_n\}_{n=1}^{\infty}$ converges to α with rate, or order, of convergence $O(\beta_n)$. (This expression is read "big oh of β_n".) It is indicated by writing $\alpha_n = \alpha + O(\beta_n)$.

Remark: Since sequence, $\{1/n^p\}_{n=1}^{\infty}$ (for some p) is a simple sequence, it is usually used as the base sequence to define the rate of convergence. In other words, we are interested in the largest value of p such that $\alpha_n = \alpha + O(1/p^n)$.

Example 5: Consider two sequences $\{\alpha_n\}_{n=1}^{\infty}$ and $\{\beta_n\}_{n=1}^{\infty}$ where

$$\alpha_n = \frac{n+1}{n^2} \text{ and } \beta_n = \frac{n+3}{n^3}.$$

What is the rate of convergence of the two sequences?

Solution: Note that

$$\lim_{n\to\infty} \alpha_n = \lim_{n\to\infty} \frac{n+1}{n^2} = 0 \text{ and } \lim_{n\to\infty} \beta_n = \lim_{n\to\infty} \frac{n+3}{n^3} = 0.$$

Therefore, $\alpha_n = 0 + O(\frac{1}{n})$ and $\beta_n = 0 + O(\frac{1}{n^2})$. In other words, the convergence rate of $\{\alpha_n\}$ and $\{1/n\}$ are the same, and $\{\beta_n\}$ and $\{1/n^2\}$ are the same.

Chapter 6

Bisection Method

There are different methods for root finding. The bisection method discussed in this note is useful for finding a root of single variable functions that satisfy certain assumptions.

6.1 The Question

The general question to answer: let $f(x)$ be a single variable continuous function defined on D, a sub set of R. Assume there is at least one root for equation $f(x) = 0$ on the closed interval [a, b].

Goal: using a numerical method to locate the root.

Several methods can be used to find the root of a given single variable equation satisfying certain regular conditions in the next few lectures. Most of these methods are direct applications of some concepts we have already covered in elementary calculus courses.

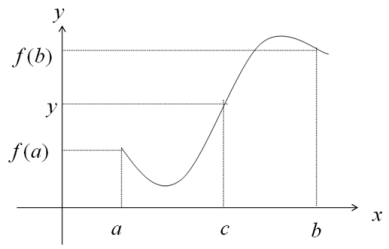
We follow the same process throughout the semester to do numerical analysis:

- doing some mathematics
- developing an algorithm
- performing error analysis
- implementing the algorithm using a programming language

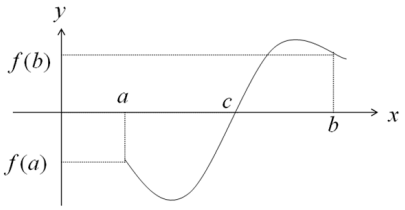
6.2 Bisection Method

The bisection method is developed based on the **intermediate value theorem (IVT)**. If a function $y = f(x)$ is continuous for all x in the closed interval $[a, b]$, and y_0 is a number between $f(a)$ and $f(b)$, then there is a number $x = c$ in (a, b) that satisfies $f(c) = y_0$.

This definition can be graphically explained in the following figure



As a special case, if we have two distinct values $x = a$ and $x = b$ that satisfy $f(a)f(b) < 0$. Then by the intermediate value theorem, there exists a $c \in [a, b]$ such that $f(c) = 0$. This means, $f(x) = 0$ has root c in $[a, b]$.



6.2.1 The Logic of Bisection Method

The method calls for a repeated halving of sub-intervals of $[a, b]$ and, at each step, locating the half interval that contains the root r (i.e., $f(r) = 0$).

https://github.com/pengdsci/MAT325/raw/main/w03/img/w03-bisectionMethod.gif

6.2.2 Bisection Algorithm

Based on the logic of the bisection method, we develop the following pseudo-code to find the root of a given equation $f(x) = 0$ on the interval $[a, b]$ assuming that $f(a)f(b) < 0$.

```
INPUT: ending values: a, b;
           tolerance: TOL
      max. iteration: N
OUTPUT:  approximate root and errors
         error/warning messages
         intermediate outputs

STEP 1.  SET n = 0
         ERR = |b - a|
STEP 2.  WHILE ERR > TOL DO:
         STEP 3  n = n + 1               (updating iteration index)
                 SET m = a + (b - a)/2   (mid-point)
```

```
STEP 4. IF f(m)f(b) < 0 DO:
            a = m                (updating ending value)
            b = b                (not necessary...)
        ENDIF
STEP 5. IF f(m)f(b) > 0 DO:
           a = a
           b = m
        ENDIF
STEP 6. ERR = |b - a|
        IF ERR < TOL DO:
            OUTPUT (p = m, other information)
            STOP
        ENDIF
        IF ERR > TOL DO:
           OUTPUT (intermediate info)
        ENDIF
STEP 7. IF n = N DO:
           OUTPUT (message on iteration limit)
           STOP
        ENDIF
ENDWHILE
```

6.3 Error Analysis

Given that we have an initial bound on the problem $[a, b]$, then the maximum error of using either a or b as our approximation is $h = b - a$. Because we halve the width of the interval with each iteration, the error is reduced by a factor of 2, and thus, the error after n iterations will be $h/2^n$. Let r be the true root of $f(x) = 0$ and p_n is the approximate root at n-th iteration. The absolute error of the bisection method is given by

$$|r - p_n| = \frac{b - a}{2^n}.$$

Next, we introduce the order of convergence that is defined in Section 2.4 of the textbook.

2.4 Error Analysis for Iterative Methods

In this section we investigate the order of convergence of functional iteration schemes and, as a means of obtaining rapid convergence, rediscover Newton's method. We also consider ways of accelerating the convergence of Newton's method in special circumstances. First, however, we need a new procedure for measuring how rapidly a sequence converges.

Order of Convergence

Definition 2.7 Suppose $\{p_n\}_{n=0}^{\infty}$ is a sequence that converges to p, with $p_n \neq p$ for all n. If positive constants λ and α exist with

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}} = \lambda,$$

then $\{p_n\}_{n=0}^{\infty}$ **converges to p of order α, with asymptotic error constant λ.** ■

An iterative technique of the form $p_n = g(p_{n-1})$ is said to be of *order* α if the sequence $\{p_n\}_{n=0}^{\infty}$ converges to the solution $p = g(p)$ of order α.

In general, a sequence with a high order of convergence converges more rapidly than a sequence with a lower order. The asymptotic constant affects the speed of convergence but not to the extent of the order. Two cases of order are given special attention.

(i) If $\alpha = 1$ (and $\lambda < 1$), the sequence is **linearly convergent**.

(ii) If $\alpha = 2$, the sequence is **quadratically convergent**.

The next illustration compares a linearly convergent sequence to one that is quadratically convergent. It shows why we try to find methods that produce higher-order convergent sequences.

Based on the above definition of the order of convergence, we have

$$\frac{|p_{n+1} - r|}{|p_n - r|^1} = \frac{(b-a)/2^{n+1}}{(b-a)/2^n} = \frac{1}{2}.$$

That is $\alpha = 1$. Therefore, the order of convergence order of the error is **linear**.

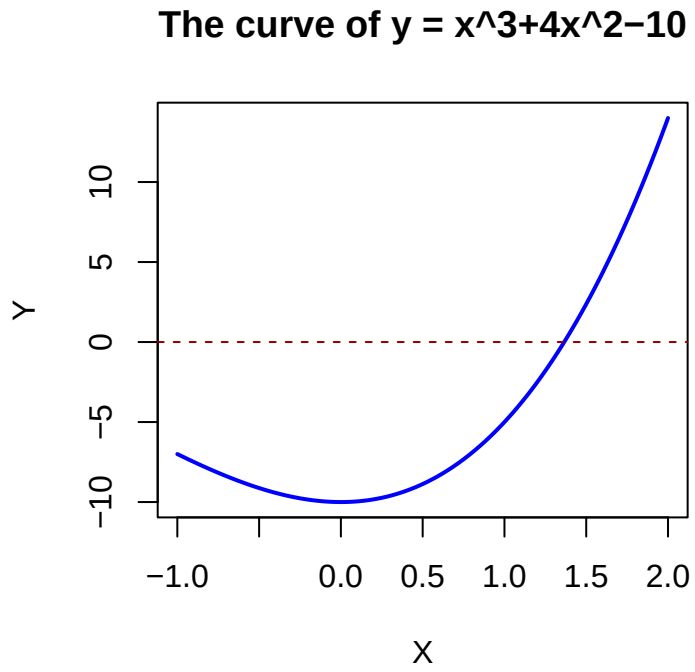
6.4 Numerical Examples.

We will present two examples to demonstrate the application of the bisection method.

Example 1: Find a root of equation $x^3 + 4x^2 - 10 = 0$.

Solution: Since we are not given the interval to apply the bisection method. We first sketch $f(x) = x^3 + 4x^2 - 10$ and $y = 0$. The x-coordinate of the intersection of the two curves is the solution to the original equation.

```
x= seq(-1, 2, length = 200)
y = x^3+4*x^2-10
#
plot(x, y,
     type = "l",
     lty = 1,
     lwd = 2,
     col = "blue",
     xlab = "X",
     ylab = "Y",
     main = "The curve of y = x^3+4x^2-10")
abline(h = 0, lty = 2, lwd = 1, col = "darkred")
```



Based on the above curve, we choose $[a, b] = [1, 2]$ to implement the bisection algorithm with the following code.

```
## we first define the R function based on the given equation
fn = function(x){
   f.val =   x^3 + 4*x^2 - 10
   f.val
}
```

```
## INPUT INFO
  a = 0
  b = 2
  TOL = 10^(-6)            # tolerance limit
  N = 200                   # max iterations
## Initialization
  n  = 0
  ERR = abs(b - a)
## Loop begins
while(ERR > TOL){
  n = n + 1
  m = a + (b - a) / 2      # midpoint of interval [a, b]
  if (fn(m)*fn(b) < 0){    # new interval should be [m, b]
      a = m                # m will be the new a
      b = b
  } else {                 # new interval should be [a,m]
      a = a
      b = m                # m will be the new 'b'
   }
  ERR = abs(b - a)         # updated absolute error
  ## Output information
  if(ERR < TOL){  # if the absolute error is less than TOL, print out results
    cat("\nThe algorithm converges!")
    cat("\nThe number of iteration n =", n, ".")
    cat("\nThe approximate root r =", (a + b)/2,".")
    cat("\nThe absolute error ERR =", ERR,".")
    break
  } else {            ## output of intermediate iteration
    cat("\nIteration:",n,". The absolute error ERR =", ERR, ".")
    }
  ### Test attainment to the max iteration
  if(n == N){
    cat("\n\n Iteration limit reached. The algorithm diverges!")
    break
  }
}
```

```
##
## Iteration: 1 . The absolute error ERR = 1 .
## Iteration: 2 . The absolute error ERR = 0.5 .
## Iteration: 3 . The absolute error ERR = 0.25 .
## Iteration: 4 . The absolute error ERR = 0.125 .
## Iteration: 5 . The absolute error ERR = 0.0625 .
## Iteration: 6 . The absolute error ERR = 0.03125 .
## Iteration: 7 . The absolute error ERR = 0.015625 .
```

```
## Iteration: 8 . The absolute error ERR = 0.0078125 .
## Iteration: 9 . The absolute error ERR = 0.00390625 .
## Iteration: 10 . The absolute error ERR = 0.001953125 .
## Iteration: 11 . The absolute error ERR = 0.0009765625 .
## Iteration: 12 . The absolute error ERR = 0.0004882812 .
## Iteration: 13 . The absolute error ERR = 0.0002441406 .
## Iteration: 14 . The absolute error ERR = 0.0001220703 .
## Iteration: 15 . The absolute error ERR = 6.103516e-05 .
## Iteration: 16 . The absolute error ERR = 3.051758e-05 .
## Iteration: 17 . The absolute error ERR = 1.525879e-05 .
## Iteration: 18 . The absolute error ERR = 7.629395e-06 .
## Iteration: 19 . The absolute error ERR = 3.814697e-06 .
## Iteration: 20 . The absolute error ERR = 1.907349e-06 .
## The algorithm converges!
## The number of iteration n = 21 .
## The approximate root r = 1.36523 .
## The absolute error ERR = 9.536743e-07 .
```

Chapter 7

Fixed Point Method

Goal: Find the root of equation $f(x) = 0$ over interval $[a, b]$.

Definition of Fixed-point Problem: The number p is a fixed point for a given function g if $g(p) = p$. In other words, if function $g(x)$ has a fixed point p, then p is a root of equation $g(x) - x = 0$.

Root-finding versus Fixed-point Problems: we could convert root finding problem $f(x) = 0$ to fixed-point problem $g(x) = af(x) + x = x$ for any real number $a \neq 0$.

This section discusses how to approximate the root p using the **Fixed-point Method**.

7.1 Some Theories of the Fixed Point Method

Theorem 1 (Existence of fixed point)

Suppose that $g(x)$ is a continuous function that maps its domain, D, onto a subset of itself, $S = g(D)$, i.e., $g(x) \in C[a, b]$, such that

$$g(x) : [a, b] \to S \subset [a, b]$$

Then $g(x)$ has a fixed point in $[a, b]$.

Proof: If $g(a) = a$ or $g(b) = b$, we are done.

Assume $g(a) \neq a$ and $g(b) \neq b$. Since $g([a,b]) \subset [a,b]$, we have $g(a) > a$ and $g(b) < b$. Let $h(x) = g(x) - x$. Since $g(x) \in C[a,b]$, so is $h(x) \in C[a,b]$. Observe that $h(a) = g(a) - a > 0$, $h(b) = g(b) - b < 0$. By intermediate value theorem, there exists at least one value r in $[a,b]$ such that $h(r) = 0$. That implies that $g(r) = r$. This completes the proof.

Corollary: Every continuous bounded function on the real numbers has a fixed point.

Theorem 2 (Uniqueness of fixed point, also called Contraction Mapping Theorem)

Suppose that $g(x)$ is a continuous function that maps its domain, D, onto a subset of itself, $S = g(D)$, i.e., $g(x) \in C[a,b]$, such that

$$g(x) : [a,b] \to S \subset [a,b]$$

Suppose further that there exists some positive constant $0 < K < 1$ such that $|g'(x)| < K$ for all x in $[a,b]$. Then $g(x)$ has a unique fixed point p in $[a,b]$.

Proof: Assume there at least two fixed points, say p_1 and p_2 in $[a,b]$, such that $g(p_1) = p_1$ and $g(p_2) = p_2$ with $p_1 < p_2$. That is, $g(p_2) - g(p_1) = p_2 - p_1$.

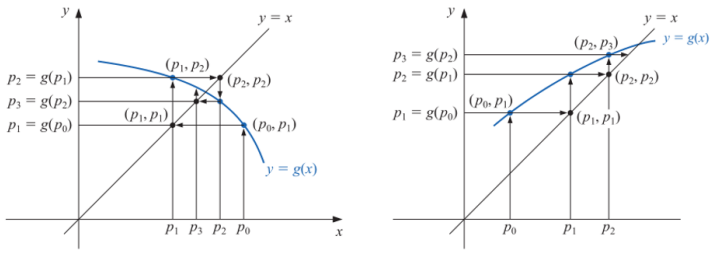
On the other hand, the mean value theorem indicated that $g(p_2) - g(p_1) = g'(c)(p_2 - p_1)$ for some c in $[p_1, p_2]$. Since , we have $g(p_2) - g(p_1) < p_2 – p_1$. This **contradicts** with $g(p_2) - g(p_1) = p_2 - p_1$. Therefore, we have proved the uniqueness of the fixed point. The proof is completed.

7.2 Fixed Point Iteration

With the theory developed previously, we focus on the following recursive equation for finding the fixed point numerically.

Choose an initial approximation p_0, generate sequence $\{p_n\}_{n=0}^{\infty}$ by $p_n = g(p_{n-1})$. If the sequence converges to p and $g(x)$ is continuous at p, then

$$p = \lim_{n\to\infty} p_n = \lim_{n\to\infty} g(p_{n-1}) = g(p)$$



The following animated graph demonstrates the process of searching the fixed point.

https://github.com/pengdsci/MAT325/raw/main/w03/img/Fixed_point_anime.gif

Fixed-point Iteration Algorithm

```
INPUT    f(x),
         initial p0,
         TOL
         N                        (max iterations)
STEP 1.  ERR = infinity           (initial error - must be a big number)
         x                        (initial value)
         n = 0                    (initial iterator)
STEP 2.  WHILE ERR > TOL DO:
         STEP 3. n = n + 1
         STEP 4. new.x = f(x)
                 ERR = new.x - x
                 IF ERR < TOL DO:
                    OUTPUT (results)
                    STOP
                  ENDIF
                 IF ERR > TOL DO:
                   OUTPUT (intermediate results)
                   x = new.x      (update x value)
                 ENDIF
                 IF n == N DO:
                   OUTPUT ()
                   STOP
                 ENDIF
```

7.3 Error Analysis

By the Lipschitz condition in the definition of **Contraction Mapping Theorem**, the approximated error is easily obtained as $|p_{n+1} - p_n| \leq K^n|p_1 - p_0|$. Next, we present a result that gives the bound the true error.

Theorem 3 (Error Estimation)

If fixed point iteration is terminated after $n > 1$ steps then the error is limited by

$$|p_n - p| \leq \frac{K^n|p_1 - p_0|}{1 - K}$$

where $0 < K < 1$ is the Lipschitz constant.

Proof We use mathematical induction to prove this theorem. For $n = 1$, we need to show that

$$|p_1 - p| \leq \frac{K}{K - 1}|p_1 - p_0|.$$

To this end, we use the Mean Value Theorem, there exists a constant $c \in [\min\{p, p_0\}, \max\{p, p_0\}]$ such that

$$|g'(c)| = \left|\frac{f(p_0) - f(p)}{p_0 - p}\right| = \left|\frac{p_1 - p}{p_0 - p}\right| \leq K.$$

That is, $|p_1 - p| \leq K|p_0 - p|$. Using the triangular inequality, we have

$$|p_1 - p| \leq K|p_0 - p| = K|p_0 - p_1 + p_1 - p| \leq K|p_1 - p_0| + K|p_1 - p|$$

Therefore

$$|p_1 - p| \leq \frac{K}{K - 1}|p_1 - p_0|.$$

We assume that original inequality is true when $n = m$, i.e.,

$$|p_m - p| \leq \frac{K^m|p_1 - p_0|}{1 - K}.$$

We want to show that

$$|p_{m+1} - p| \leq \frac{K^{m+1}|p_1 - p_0|}{1 - K}.$$

We still use the MVT on interval $[\min\{p, p_m\}, \max\{p, p_m\}]$ to get

$$|g'(c)| = \left|\frac{f(p_m) - f(p)}{p_m - p}\right| = \left|\frac{p_{m+1} - p}{p_m - p}\right| \leq K.$$

where $c \in [\min\{p, p_m\}, \max\{p, p_m\}]$. Therefore,

$$|p_{n+1} - p| \leq K|p_n - p|$$

and

$$|p_{m+1} - p| \leq K|p_m - p_0| = \frac{K^{m+1}}{1 - K}|p_1 - p_0|.$$

This completes the proof.

Corollary (Order of Convergence)

If the fixed-point method converges, it has a linear convergence order.

The proof has already been given in the proof of the above theorem.

7.4 Numerical Example

We know that there is a solution for the equation $x^3 - 7x + 2 = 0$ in $[0, 1]$. We rewrite the equation in the form $x = (x^3 + 2)/7$ and denote the process $x_{n+1} = (x_n^3 + 2)/7$. We see from the following figures that if $0 \leq x_0 \leq 1$ then (x_n) converges to a root of the above equation. We also note that if we start with (for example) $x_0 = 2.5$ then the recursive process does not converge.

```
## input values
TOL = 10^(-6)
  M = 200
####
gfun = function(x) (x^3 + 2)/7
####
 a = 0
 b = 2
 x = 2        # initial value of x
 n = 0         # initializing iterator
 ERR = Inf     # initial error
##
 while (ERR > TOL){
  n = n + 1
  new.x = gfun(x)
```

```
  ERR = abs(new.x - x)
  if(ERR < TOL){
    cat("\n\nThe algorithm converges!")
    cat("\nThe approximate root is:", new.x,".")
    cat("\nThe absolute error is:", ERR, ".")
    cat("\nThe number of iterations is:", n, ".")
    break
  } else{
    if(ERR > 10^7){
        cat("\n\nThe algorithm diverges!")
        break
    } else{
         cat("\nIteration:",n,". Estimated root:", new.x, ". Absolute error:", ERR,".")
         x = new.x               # update x value!!!
    }
  }
  if(n == M){
    cat("\n\nThe maximum number of iterations is achieved!")
    break
  }
 }
```

```
##
## Iteration: 1 . Estimated root: 1.428571 . Absolute error: 0.5714286 .
## Iteration: 2 . Estimated root: 0.7022074 . Absolute error: 0.726364 .
## Iteration: 3 . Estimated root: 0.3351793 . Absolute error: 0.3670281 .
## Iteration: 4 . Estimated root: 0.2910937 . Absolute error: 0.04408562 .
## Iteration: 5 . Estimated root: 0.289238 . Absolute error: 0.001855685 .
## Iteration: 6 . Estimated root: 0.289171 . Absolute error: 6.696095e-05 .
## Iteration: 7 . Estimated root: 0.2891686 . Absolute error: 2.400242e-06 .
##
## The algorithm converges!
## The approximate root is: 0.2891685 .
## The absolute error is: 8.601697e-08 .
## The number of iterations is: 8 .
```

Chapter 8

Newton's Method

We have introduced bisection and fixed-point methods for finding the root of single-variable equations over a pre-selected interval. Both methods have a linear convergence rate (if the error sequence converges). This note introduces the well-known Newton method for finding the root of non-linear equations. We will see that the Newton method has a quadratic convergence rate (if converges). Unlike the bisection method, this method can be extended to multi-variable nonlinear systems (same as the fixed-point method).

8.1 Notations: Big O and Little o

We have introduced the concept of convergence rate at which some function changes as its argument grows (or shrinks), without worrying too much about the detailed form. This is what the O(·) and o(·) notation are. We now give a little more detail about these notations.

A function $f(n)$ is "of constant order", or "of order 1" when there exists some non-zero constant c such that

$$\frac{f(n)}{c} \to 1$$

as $n \to 1$; equivalently, since c is a constant, $f(n) \to c$ as $n \to 1$. It doesn't matter how big or how small c is, just so long as there is some such constant. We then write

$$f(n) = O(1)$$

and say that "the proportionality constant c gets absorbed into the big O". For example, if $f(n) = 37$, then $f(n) = O(1)$. But if $g(n) = 37(1 - 2/n)$, then $g(n) = O(1)$

The other orders are defined recursively. Saying

$$g(n) = O(f(n))$$

means

$$\frac{g(n)}{f(n)} = O(1), \text{ or } \frac{g(n)}{f(n)} \to c,$$

as $n \to \infty$. This is equivalently to say that $g(n)$ is **of the same order** as $f(n)$, and they **grow at the same rate**!

Example 1: a quadratic function $a_1n^2 + a_2n + a_3 = O(n^2)$, no matter what the coefficients are. On the other hand, $b_1n^{-2} + b_1n^{-1}isO(n^-1)$.

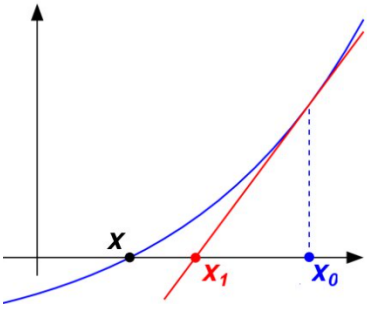
Big-O means "is of the same order as". The corresponding little-o means "is ultimately smaller than": $f(n) = o(1)$ means that $f(n)/c \in 0$ for any constant c. Reccursively, $g(n) = o(f(n))$ means $g(n)/f(n) = o(1)$, or $g(n)/f(n) \to 0$. We also read $g(n) = o(f(n))$ as "$g(n)$ is ultimately negligible compared to $f(n)$".

8.2 Foundations of Newton Method

The Newton method is formulated based on the Taylor series.

8.2.1 The Algorithmic Logic

Let's consider a general function $f(x)$. For the starting point x_0, the slope of the tangent line at the point $(x_0, f(x_0))$ is $f'(x_0)$ so the equation of the tangent line is $y - f(x_0) = f'(x_0)(x - x_0)$. We look at the intersection between the tangent line and x-axis: $(x_1, 0)$



where x_1 is the root of $0 - f(x_0) = f'(x_0)(x - x_0)$. solving the equation, we have $x_1 = x_0 - f(x_0)/f'(x_0)$. In the above figure, we can see x_1 is closer to the true root x. If we draw the tangent line at $(x_1, f(x_1))$ and look at the

intersection between the x-axis and this tangent line, the x-coordinate $x_2 = x_1 - f(x_1)/f'(x_1)$.

https://github.com/pengdsci/MAT325/raw/main/w04/img/w04-NewtonIterationGIF.gif

Starting with x_1 and repeating this process we have $x_2 = x_1 - f(x_1)/f'(x_1)$, we get $x_3 = x_2 - f(x_2)/f'(x_2)$; and so on.

8.2.2 Initial Starting Value Matters

Here are a few examples with different starting values. We can see the number of iterations needed to achieve the error tolerance.

Example 2. Find the root of equation $f(x) = x^3 - x + 3 = 0$ using various initial starting values.

Case 1: $x_0 = -1$. The algorithm converges after 6 iterations.

Case 2: $x_0 = -0.1$. The algorithm converges after 33 iterations.

Case 3: $x_0 = 0$. The algorithm **diverges** with the initial value $x_0 = 0$!

8.3 Algorithm and Pseudo-code

Assume that $f(x) \in C^2[a, b]$. Let $x_0 \in [a, b]$ be an approximation to p, **the root of** $f(x) = 0$, such that $f(x_0) \neq 0$ and $|p - x_0|$ is "small."

Consider the first Taylor polynomial for $f(x)$ expanded about x_0 and evaluated at $x = p$.

$$f(p) = f(x_0) + (p - x_0)f'(x_0) + \frac{(p - x_0)^2}{2} f''(\xi(p))$$

where $\xi(p)$ is some number in $[\min x_0, p, \max p, x_0]$. Since $f(p) = 0$ and $|p - x_0|$ is "small", therefore, $0 \approx f(x_0) - (p - x_0)f'(x_0)$. This yields

$$p \approx x_0 - \frac{f(x_0)}{f'(x_0)} \to x_1$$

As demonstrated in the previous section, continuing this process, we have $\{x_n\}_{n=0}^{\infty}$, where

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ for } n \geq 0,$$

to approximate the root of equation $f(x) = 0$.

Pseudo-code of Newton Method:

```
INPUT:   initial x0;
         TOL;
         M = maximum iterations.
         f(x)
         f'(x)
OUTPUT:  Approximated root and optional information.

STEP 1:  n = 0    (initial counter)
         x = x0   (initial value)
         ERR = |f(x)/f'(x)|
STEP 2: WHILE ERR > TOL DO:
           n = n + 1
           x = x -f(x)/f'(x)
           ERR = |f(x)/f'(x)|
           IF ERR < TOL DO:
              OUTPUT (result and related info)
              STOP
           ENDIF
           IF ERR >= TOL DO:
              OUTPUT (intermediate info and messages)
           ENDIF
           IF n = M DO:
              OUTPUT (message: max iterations achieved!)
              STOP
           ENDIF
        ENDWHILE
```

Implementation with R

The following code is developed based on the following example.

Example 2 (Revisited): Find the root of equation $f(x) = x^3 - x + 3 = 0$.

```
# Define f(x) and f'(x)

fn = function(x) x^3 - x +3
dfn = function(x) 3*x^2 - 1

# initial values
n = 0
x = -1
M = 200
TOL = 10^(-6)
ERR = abs(fn(x)/dfn(x))
# loop begins
while(ERR > TOL){
```

```
  n = n + 1
  x = x - fn(x)/dfn(x)
  ERR = abs(fn(x)/dfn(x))
  if(ERR < TOL){
     cat("\n\nAlgorithm converges!")
     cat("\nThe approximated root:", x, ".")
     cat("\nThe absolute error:", ERR, ".")
     cat("\nThe number of iterations n =",n,".")
     break
    } else{
      cat("\nIteration n =",n, ", approximate root:",x,", absolute error:", ERR,".")
    }
    if (n ==M){
      cat("\n\nThe maximum iterations attained!")
      cat("\nThe algorithm did not converge!")
      break
    }
}
```

```
##
## Iteration n = 1 , approximate root: -2.5 , absolute error: 0.5704225 .
## Iteration n = 2 , approximate root: -1.929577 , absolute error: 0.2217111 .
## Iteration n = 3 , approximate root: -1.707866 , absolute error: 0.03530793 .
## Iteration n = 4 , approximate root: -1.672558 , absolute error: 0.0008580914 .
##
## Algorithm converges!
## The approximated root: -1.6717 .
## The absolute error: 5.002863e-07 .
## The number of iterations n = 5 .
```

8.4 Error Analysis

Assume that $f(x) \in C^2[a,b]$ is continuous and p is a simple zero of $f(x)$ so that $f(p) = 0 \neq f'(p)$. From the definition of the Newton iteration, we have

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

Using Taylor expansion, we have

$$f(x_n) = f'(p)(x_n - p) + \frac{1}{2}f''(\xi(p))(x_n - p)^2 = f'(p)e_n + \frac{1}{2}f''(\xi(p))e_n^2,$$

where $\xi(p)$ is between x_n and p. Therefore,

$$e_{n+1} = e_n - \frac{f'(p)e_n + \frac{1}{2}f''(\xi(p))e_n^2}{f'(p)} = \frac{f''(\xi(p))}{2f'(p)}e_n^2,$$

that is,

$$\frac{e_{n+1}}{e_n^2} = \frac{f''(\xi(p))}{2f'(p)}.$$

Theorem: Assume $f(x)$ is a continuous function with a continuous second derivative, that is defined on an interval $I = [p-\delta, p+]delta$, with $\delta > 0$. Assume that $f(p) = 0$, and that $f''(p) \neq 0$. Assume that there exists a constant M such that

$$\left|\frac{f''(x)}{f'(y)}\right| \leq M, \text{ for } x, y \in I$$

If x_0 is sufficiently close to the root p, i.e., if $|x_0 - p| \leq \min\{\delta, 1/M\}$, then the sequence $\{x_n\}$ defined in Newton Method converges to the root p with a quadratic convergence order.

Chapter 9

Secant Method

Recall that Newton's method uses Taylor expansion to derive the functional recursive relationship between adjacent approximated roots

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \text{ for } n = 0, 1, \cdots.$$

where $f'(x_n)$ is the slope of the tangent line passing through $x = x_n$. If we use the slope of a secant line that passes through points $(x_n, f(x_n))$ and $(x_{n-1}, f(x_{n-1}))$, we can use the x-coordinates of the intersection between the secant line and x-axis to approximate the root of $f(x) = 0$.

9.1 Secant Method

Assume we have two distinct initial values $x = x_0$ and $x = x_1$. Then slope of the secant line passing through A$(x_0, f(x_0))$ and B$(x_1, f(x_1))$ is

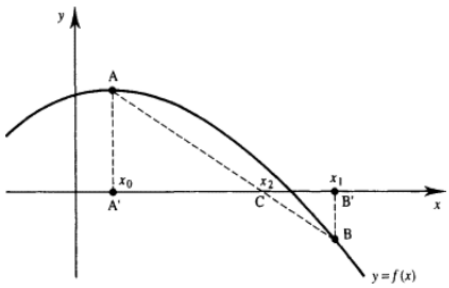
$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} \approx f'(x_1) \text{ when } |x_1 - x_0| \text{ is small.}$$

The secant method uses the x-coordinate of the intersection of the secant line

$$f(x) = f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1)$$

and $f(x) = 0$ (equation of the x-axis). Solving for x, we have

$$x = x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1) \equiv x_2.$$



In general, the recursive relationship between approximated roots of the **secant method** is given by

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n), \text{ for } n = 0, 1, \cdots$$

https://github.com/pengdsci/MAT325/raw/main/w05/img/w05-betterSecantAnimation.gif

9.2 Secant Algorithm and Implementation

We develop the following pseudo-code of the secant method.

```
INPUT:  f(x)            (satisfying f(x) = 0)
        x0              (initial value 1)
        x1              (initial value 2)

STEP 1: x0
        x1              (f(x0)*f(x1) must be negative)
        M = 200
        TOL = 10^(-6)
        n = 0
        ERR = |x1 - x0|
STEP 2: WHILE ERR > TOL DO
           n = n + 1
           new.x = x1 - ((x1-x0)/(f(x1)-f(x0)))*f(x1)
           ERR = |new.x - x1|
           IF ERR < TOL DO:
              OUTPUT        (results and optional relevant info)
              STOP
           ENDIF
           IF ERR >= TOL DO:
              OUTPUT        (message or intermediate outputs)
              x1 = new.x    (update)
              x0 = x1
           ENDIF
           IF n == M DO:
```

```
            OUTPUT          (warning messages)
            STOP
          ENDIF
        ENDWHILE
```

9.2.1 Implementation with R

We next write an R function to implement the secant method.

```
########################################
##     Root Finding: Secant Method
########################################
Secant.Method = function(fn,              # input function
                  TOL,                    # error tolerance
                  max.iter,               # max allowed iterations
                  x1,                     # initial value #1
                  x2                      # initial value #2
                  ){
  ctr = 0                 # counter of iteration
  ERR = abs(x2 - x1)      # initial error - width of initial interval
  # Define a data frame (data table) to store the output of each iteration
  ERR.table =  data.frame(Iteration = 1:max.iter,
                          Est.root = rep(NA, max.iter),
                          Abs.error = rep(NA, max.iter))
  while(ERR > TOL){
      ctr = ctr + 1
      new.x = x2 - fn(x2) * (x2 - x1) / (fn(x2) - fn(x1))
      ERR = abs(new.x - x2)
      if(ERR < TOL){
         ERR.table[ctr,] = c(ctr, new.x, ERR)
         break
         } else{
            ERR.table[ctr,] = c(ctr, new.x, ERR)
            # updating the two values. CAUTION: order matters
            x1 = x2
            x2 = new.x
         }
        if(ctr == max.iter){
         #cat("\n\nThe maximum number of iterations attained!\n\n\n")
         break
       }
   }                      # close the while-loop
  if(ctr==max.iter){
    pander(data.frame(message = "The maximum number of iterations attained!"))
  } else{
  na.omit(ERR.table)      # delete rows with NAs (missing values)
```

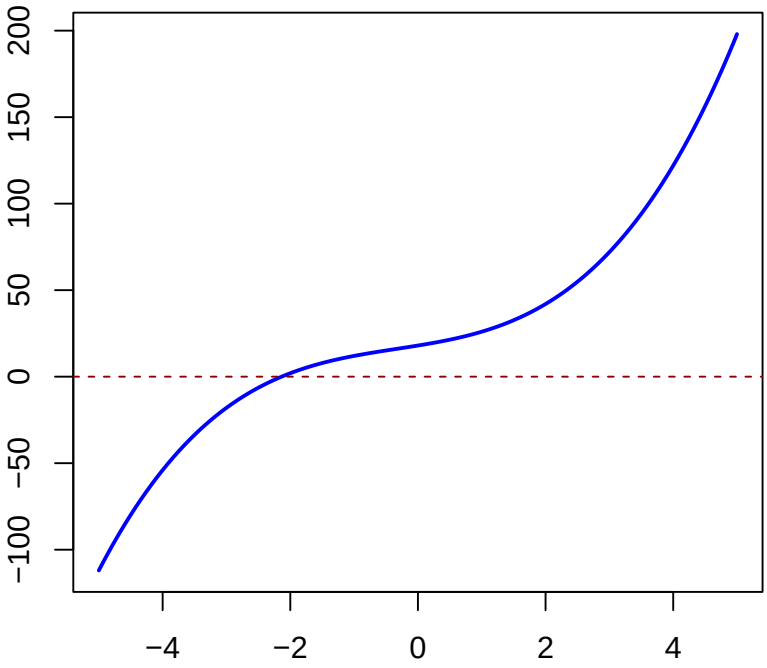
```
  }
}                          # close the function environment
```

9.2.2 Numerical Examples

Example 1: Find a root of equation $x^3 + x^2 + 6x + 18 = 0$.

Solution: we use the above R function of the Newton method to find the approximated root of the equation.

```
# define the function f(x) that satisfies f(x) = 0
example01.func = function(x){x^3+x^2+6*x+18 }
###
xx = seq(-5,5, length=500)     # 500 evenly x-values evenly spread on [-5, 5]
yy = example01.func(xx)        # the corresponding y values
plot(xx, yy, type = "l", xlab ="", ylab="", main="", lwd = 2, col = "blue")
abline(h=0, col = "darkred", lty = 2)
```



Based on the above graph, we search a root over $[-5, 5]$ in the follwoing function call.

```
# call the function
error.matrix = Secant.Method(fn = example01.func,       # input function
                      TOL = 10^(-8),                 # error tolerance
                      max.iter = 5,                 # max allowed iterations
                      x1 = -5,                       # initial value #1
                      x2 = 5)                        # initial value #2
pander(error.matrix)
```

```
## Warning in pander.default(error.matrix): No pander.method for "knit_asis", reverting to defaul
```

message
The maximum number of iterations attained!

•

The error plot is given by

```
if(length(dim(error.matrix))>0){
  ###
  Error = error.matrix$Abs.error
  nitr = length(Error)
  plot(1:nitr, Error, type = "l", lwd = 2, col = "blue",
             main="Error Plot",
             xlim = c(0,nitr+1),
             ylim = c(0, max(Error)),
             xlab = "Iteration Numbers",
             ylab = "Absolute Error",
             cex.main = 0.8,
             col.main = "darkred"
       )
  }else{
  pander(data.frame(meesage ="The approximation error is unavailable!"))
}
```

meesage
The approximation error is unavailable!

Practice Exercise: find the solution to $0.8(x+0.5)^3 - 1 = 0$ 0n $[0.25, 2.75]$.

9.3 Error Analysis

Let $e_n = x_n - p$, then $e_n - e_{n-1} = x_n - x_{n-1}$. From the definition of the secant method we have

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

With some algebraic manipulation, we can express e_{n+1} as

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

Note that

$$\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \approx \frac{1}{f'(p)}$$

After expanding $f(x_n)$ and $f(x_{n-1})$ at p, we have

$$\frac{f(x_n)/e_n - f(x_{n-1})/e_{n-1}}{x_n - x_{n-1}} \approx \frac{f''(p)}{2}$$

Therefore,

$$e_{n+1} \approx \frac{f''(p)}{2f'(p)} e_n e_{n-1}$$

Consequently,

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n e_{n-1}} = \frac{f''(p)}{2f'(p)} = C_0$$

To find the order of convergence, we assume that $e_{n+1} = C_n e_n^\alpha$ where $\lim_{n\to\infty} C_n = C$. Then

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n e_{n-1}} = \lim_{n\to\infty} \frac{C[Ce_{n-1}^\alpha]^\alpha}{Ce_{n-1}^\alpha e_{n-1}} = \lim_{n\to\infty} C^\alpha e_{n-1}^{\alpha^2-\alpha-1} = C_0.$$

This implies that

$$\alpha^2 - \alpha - 1 = 0$$

The positive root of the above equation is

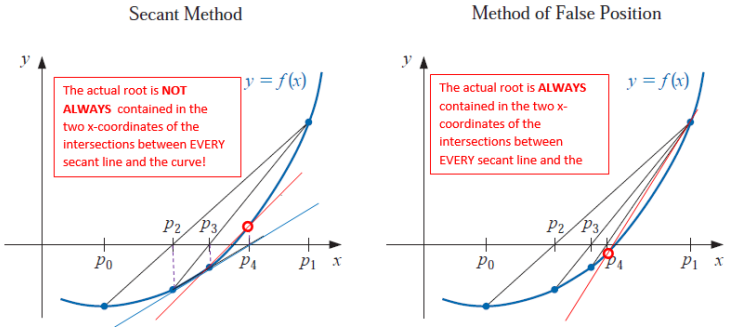
$$\alpha = \frac{1+\sqrt{5}}{2} \approx 1.62$$

Therefore, the convergence order for the secant method is between linear and quadratic orders – we call this **super-linear** convergence!

9.4 False Position Method

The method of False Position (also called Regula Falsi) generates approximations using the x-coordinates of successive secant lines defined prior approximated roots in such a ways that the actual roots is always $[x_n, x_{n-1}]$ in the n-th iteration. **Programmatically, it is the secant method with an additional control statement.**

> First choose initial approximations p_0 and p_1 with $f(p_0) \times f(p_1) < 0$. The approximation p_2 is chosen in the same manner as in the Secant method, as the x-intercept of the line joining $(p_0, f(p_0))$ and $(p_1, f(p_1))$. To decide which secant line to use to compute p_3, consider $f(p_2) \times f(p_1)$, or more correctly $\text{sgn}f(p_2) \times \text{sgn}f(p_1)$
>
> - If $\text{sgn}f(p_2) \times \text{sgn}f(p_1) < 0$, then p_1 and p_2 bracket a root. Choose p_3 as the x-intercept of the line joining $(p_1, f(p_1))$ and $(p_2, f(p_2))$.
> - If not, choose p_3 as the x-intercept of the line joining $(p_0, f(p_0))$ and $(p_2, f(p_2))$, and then interchange the indices on p_0 and p_1.



The following is an animated graph showing the process of search the root using false position method.

https://github.com/pengdsci/MAT325/raw/main/w05/img/w05-regula.gif

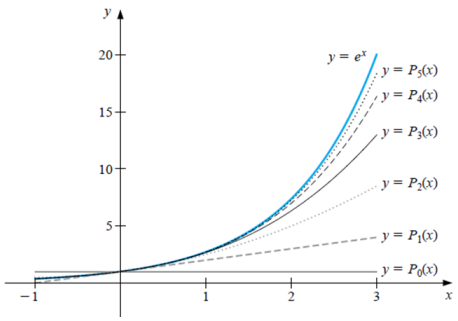
Finally, the order of convergence of **false position method** is same as the **secant method**.

Chapter 10

Lagrange Interpolation

In many computational applications, one must approximate an intractable real-valued function $f(x)$ with a computationally tractable function $\widehat{f}(x)$. Broadly speaking, there are two types of function approximation problems that arise often in real-world applications: interpolation and functional equation problems.

Starting from this note, we will focus on numerical approximation problems via interpolation. We have noticed that a given continuous function can be approximated by a polynomial function. The following figure shows that $y = e^x$ can be approximated by Taylor polynomials reasonably well.



where

$$P_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}.$$

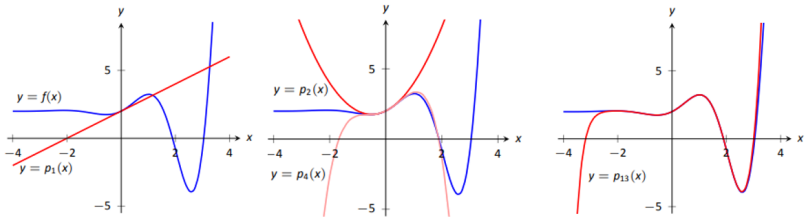
Note that the Taylor expansion of $y = e^x$ is given by

$$y = e^x = P_n(x) + R_n(x)$$

With

$$R_n(x) = \frac{e^{\xi} x^{n+1}}{(n+1)!} \quad \text{for some } \xi \in (0, x).$$

It is not surprising that, as n gets bigger, $P_n(x)$ gets closer to $y = e^x$. The next figure gives the curve of a function that is *more complex* than e^x. We can see a similar pattern as seen in the above figure.



$$p_{13} = \frac{16901x^{13}}{6227020800} + \frac{13x^{12}}{1209600} - \frac{1321x^{11}}{39916800} - \frac{779x^{10}}{1814400} - \frac{359x^9}{362880} + \frac{168\ x^8}{\cancel{240}\ 40320} + \frac{139x^7}{5040} + \frac{\overset{22}{\cancel{11}}x^6}{\cancel{360}\ 720} - \frac{19x^5}{120} - \frac{12\ x^4}{\cancel{2}\ 24} - \frac{x^3}{6} + \frac{\overset{2}{x^2}}{2} + x + 2.$$

The approximation error is summarized in the following Theorem (will not prove it in this class)

Theorem. Let $f(x)$ be a real-valued function that has continuous derives up to order $n + 1$, Then the remainder of the Taylor expansion at $x = a$ (i.e., approximation error) can be expressed in the following integral form

$$R_n^a[f(x)] = \int_a^x \frac{f^{(n+1)}(t)}{n!}(t-a)^n dt.$$

In the above two examples, the underlying function $f(x)$ was expanded at $x = 0$ (i.e., Maclaurin expansion). In general, Taylor expansion (approximation) uses values of the function and its derivatives: $f(a), f'(a), f^{(2)}(a), \cdots, f^{(n)}$ and $f^{(n+1)(\xi)}$ where $\xi \in (a, x)$.

We can also consider Taylor expansion as a linear combination of basis functions $\{1, x, x^2, x^3, \cdots, x^n, \cdots\}$.

There are several obvious disadvantages of the Taylor polynomial approximation:

- $y = f(x)$ must be explicitly given and is n-th order differentiable. That is, to get an n-th degree Taylor polynomial, we need to assume $f(x)$ to have an n-th order derivative.
- The approximation is very well in the neighborhood of $x = a$ at which the function $y = f(x)$ is expanded (via Taylor expansion) but is poor far away from the neighborhood.

A natural question: *whether we can sample a set of points on the curve of* $y = f(x)$ *and then find a lower degree (than Taylor) polynomial for approximating* $y = f(x)$ *such that* $P_n(x_i) = f(x_i)$.

The answer to the question is **YES**. Several methods using this idea will be introduced in the next few notes.

10.1 Concepts of Interpolation Method

A function is said to **interpolate** a set of data points if it passes through those points.

Definition: The function $y = f(x)$ interpolates the data points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, x_n)\}$ if $y_i = P_n(x_i)$ for each $1 \leq i \leq n$.

Since $f(x)$ is a function; x_i's must be all distinct in order for a function to pass through them.

10.1.1 Data-fitting / Interpolation:

For the following given points samples from an **unknown function** $f(x)$:

x	x_0	x_1	x_2	$\cdots$	x_n
y	y_0	y_1	y_2	$\cdots$	y_n

and we try to find a polynomial $P_n(x)$ of degree $\leq n$ for which,

$$P_n(x_i) = y_i, \quad \text{for} \quad 0 \leq i \leq n.$$

such a polynomial is said to interpolate the data (data fitting). This type of question is very common in almost areas that produce data.

Existence of Polynomial Interpolation: if $\{x_0, x_1, x_2, \cdots, x_n\}$ are distinct real numbers, then for arbitrary values $\{y_0, y_1, y_2, \cdots, y_n\}$ there is a unique polynomial $P_n(x)$ of degree $\leq n$ such that

$$P_n(x_i) = y_i, \quad \text{for } 0 \leq i \leq n.$$

Proof. For any polynomial $P_n(x)$ of degree $\leq n$, we have the following form :

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

To determine the polynomial $P_n(x)$ is to find the coefficient a_i's. We will use the interpolation condition $P_n(x_i) = y_i, \quad \text{for } 0 \le i \le n.$

$$\begin{cases} a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_0^3 + \cdots + a_n x_0^n = y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 + \cdots + a_n x_1^n = y_1 \\ a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3 + \cdots + a_n x_2^n = y_2 \\ \qquad\qquad\qquad \cdots \\ a_0 + a_1 x_n + a_2 x_n^2 + a_3 x_n^3 + \cdots + a_n x_n^n = y_n \end{cases}$$

Note that $\{a_0, a_1, a_2, \cdots, a_n\}$, are unknown. We rewrite it in the following matrix form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^n \\ 1 & x_3 & x_3^2 & x_3^3 & \cdots & x_3^n \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}.$$

Since the coefficient matrix is a **Vandermonde matrix**, it is nonsingular if and only if $\{x_0, x_1, x_2, \cdots, x_n\}$ are distinct. This imply that the system exists a unique solution $(a_0, a_1, a_2, \cdots, a_n)^T$ if and only if $\{x_0, x_1, x_2, \cdots, x_n\}$ are distinct since the determinant of that matrix is

$$\prod_{1 \le i < j \le n} (x_i - x_j)$$

Hence, there exists a unique polynomial $P_n(x)$ of degree $\le n$ if $\{x_0, x_1, x_2, \cdots, x_n\}$ are distinct.

10.1.2 Functional Equation (Curve Approximation)

Another type of interpolation problem is formulated as follows: given a set of $\{x_0, x_1, x_3, \cdots, x_n\}$ and a continuous function $f(x)$, find a polynomial $P_n(x)$ of degree less than or equal to n such that $P_n(x_i) = f(x_i)$ for $0 \le i \le n$.

The Newton interpolation is one type of this problems. We will introduce this method in a subsequent note.

10.2 The Lagrange Interpolation

The basic idea of Lagrange interpolation is approximate a function by using a linear combination of Lagrange basis polynomials defined based on a given set of distinct points $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ sampled from a curve of a function $f(x)$ with an unknown analytic expression. The given points are called interpolation nodes.

Next, we use several special interpolations to illustrate the construction of Lagrange basis polynomials.

10.2.1 Linear Lagrange Interpolation

Assume that we are given two distinct points $\{(x_0, y_0), (x_1, y_1)\}$. The objective is to find an interpolation "*polynomial*" that passes through the two points. Intuitively, we use the following two-point form

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

The above function is linear (degree 1 polynomial). We use

$$p_1(x) = \frac{y_1 - y_0}{x - x_0}(x - x_0) + y_0.$$

Next, we re-express the above degree one polynomial in the following

$$p_1(x) = y_1 \frac{x - x_0}{x_1 - x_0} + y_0 \left(1 - \frac{x - x_0}{x_1 - x_0}\right) = y_1 \frac{x - x_0}{x_1 - x_0} + y_0 \frac{x - x_1}{x_0 - x_1}$$

We denote

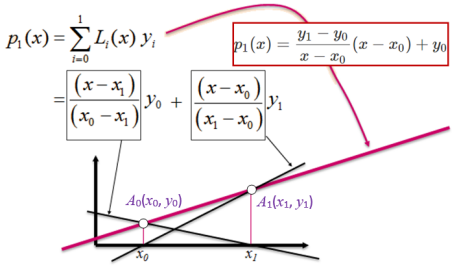
$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \quad \text{and} \quad L_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

$L_0(x)$ and $L_1(x)$ are both degree-one polynomials. They are called **Lagrange Basis Polynomials with degree 1**.

Observations of Lagrange Basis Polynomials: For $i, j = 0, 1$,

$$L_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$$

The degree-one interpolation polynomial is expressed as $p_1(x) = y_0 L_0(x) + y_1 L_1(x)$. The following figure illustrates how the interpolated polynomial is expressed as the linear combination of the **Lagrange Basis Polynomials**.



We can check that $p_1(x_0) = y_0$ and $p_1(x_1) = y_1$..

10.2.2 Quadratic Lagrange Interpolation

Quadratic Lagrange interpolation assumes that three distinct points were sampled from the curve.

x	x_0	x_1	x_2
y	y_0	y_1	y_2

The objective is to find a polynomial $p_2(x)$ of degree ≤ 2 such that

$$p_2(x_i) = y_i, \quad \text{for } i = 1, 2, 3.$$

we construct the basis $L_0(x), L_1(x), L_2(x)$ such that

$$L_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$$

We only construct the basis function $L_0(x)$ associated with the point x_0. Since x_1 and x_2 are zeros of $L_0(x)$, it should have the following form

$$L_0(x) = c(x - x_1)(x - x_2).$$

Since $L_0(x_0) = 1$, which implies that

$$c = \frac{1}{(x_0 - x_1)(x_0 - x_2)}$$

Therefore,

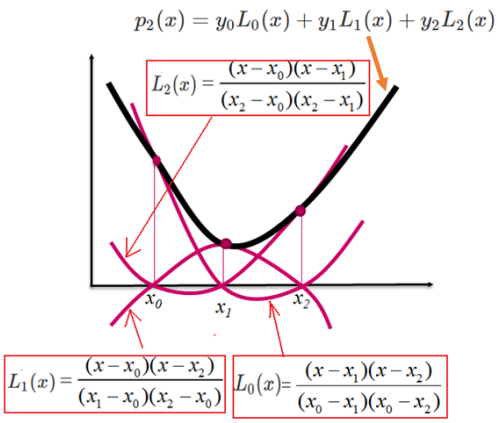
$$L_0 = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}$$

Similarly, we can construct $L_1(x)$ and $L_2(x)$ in the following

$$L_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \quad \text{and} \quad L_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

Hence the interpolation polynomial is as follows

$$p_2(x) = y_0L_0(x) + y_1L_1(x) + y_2L_2(x)$$



10.2.3 General Lagrange Interpolation

Assume now that we are given the following distinct points

x	x_0	x_1	x_2	$\cdots$	x_n
y	y_0	y_1	y_2	$\cdots$	y_n

then a unique polynomial $p_n(x)$ of degree at most n exists with

$$p_n(x) = y_k$$

This polynomial is explicitly defined as follows

$$p_n(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + \cdots + y_n L_{n,n}(x),$$

where

$$L_{n,k} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x_k - x_n)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}.$$

10.3 Lagrange Algorithm and Implementation

The algorithm of the Lagrange interpolation involves two nested iterative processes:

- Approximated individual basis polynomial and evaluate it at a given x-value (including the x-coordinates in the approximating notes);
- Estimated the set of estimated polynomials with the approximated value of Pn(x).

10.3.1 Pseudo-code

The pseudo-code is given by:

```
INPUT: x1, x2, ... ,xn
       y1, y2, ... ,yn
        (or f(x1), f(x2), ..., f(xn))
       pred.x

OUTPUT: return Pn(x)

STEP 1: set initial values
        Pn = 0     (initial value of interpolated polynomial)
        LP = 1     (vector with all 1s)

Step 2: FOR i = 1, 2, ..., n. DO
        STEP 3: FOR j = 1, 2, ..., n. DO
                    IF i != j DO:
                       LP = LP*(pred.x-xj)/(xi-xj)
                    ENDIF
                ENDFOR
        STEP 4  Pn = LP*yi + Pn
        ENDFOR
STEP 5: OUTPUT Pn
```

10.3.2 R Function with Scalar Input

The next function takes only a single x value and returns the value of the approximated polynomial at the provided x value.

```
#########################################################
##        Lagrange Interpolation
#########################################################
LagrangeInterpolation =function(
                        pred.x,       # scalar x for eval Pn()
                        fn = NULL,    # input function or
                        yvec = NULL, # input y-coordinates
                        xvec          # input x-coordinates
                        ){
    #
    if(length(yvec) == 0) yvec = fn(xvec) #
    n = length(xvec)       # input x-coordinates
    Pn = 0
    for (i in 1:n){
        LP = 1
        for (j in (1:n)[-i]){
             LP = LP * (pred.x - xvec[j])/(xvec[i] - xvec[j])
            }
        Pn = Pn + LP * yvec[i]
      }
   Pn
  }
```

Example 1: Find a Lagrange polynomial to approximate the function $f(x) = e^x \cos(3x)$ and estimate the value of $f(x)$ at $x = 0.5$ and 0.3 respectively.

Solution: We use the above R function to estimate $f(x)$ at $x = 0.5$ and 0.3. We also print out the true values $f(x)$ at $x = 0.5$ and 0.3 for comparison.

```
fn=function(x) exp(x)*cos(3*x)
approx.val0.3 = LagrangeInterpolation(fn=fn, xvec=c(0, 0.3, 0.6), pred.x = 0.3)
approx.val0.5 = LagrangeInterpolation(fn=fn, xvec=c(0, 0.3, 0.6), pred.x = 0.5)
true.val = fn(c(0.3,0.5))
pander(cbind(approx.val0.3 = approx.val0.3, true.val0.3 = true.val[1],
             approx.val0.5 = approx.val0.5, true.val0.5 = true.val[2]))
```

approx.val0.3	true.val0.3	approx.val0.5	true.val0.5
0.8391	0.8391	0.1251	0.1166

10.3.3 R Function with Vector Input

```
#########################################################
##        Lagrange Interpolation
#########################################################
Lagrange.Interpolation.Vector =function(
                        pred.x,      # vector x for eval Pn()
                        fn = NULL,   # input function or
                        yvec = NULL, # input y-coordinates
                        xvec         # input x-coordinates
                        ){
  #
  if(length(yvec) == 0) yvec = fn(xvec) #
  n = length(xvec)       # input x-coordinates
  m = length(pred.x)     # number of input x values
  PV = rep(0, m)
  for (k in 1:m){
     Pn = 0
     for (i in 1:n){
         LP = 1
         for (j in (1:n)[-i]){
              LP = LP * (pred.x[k] - xvec[j])/(xvec[i] - xvec[j])
             }
         Pn = Pn + LP * yvec[i]
       }
     PV[k] = Pn
    }
   PV
  }
```

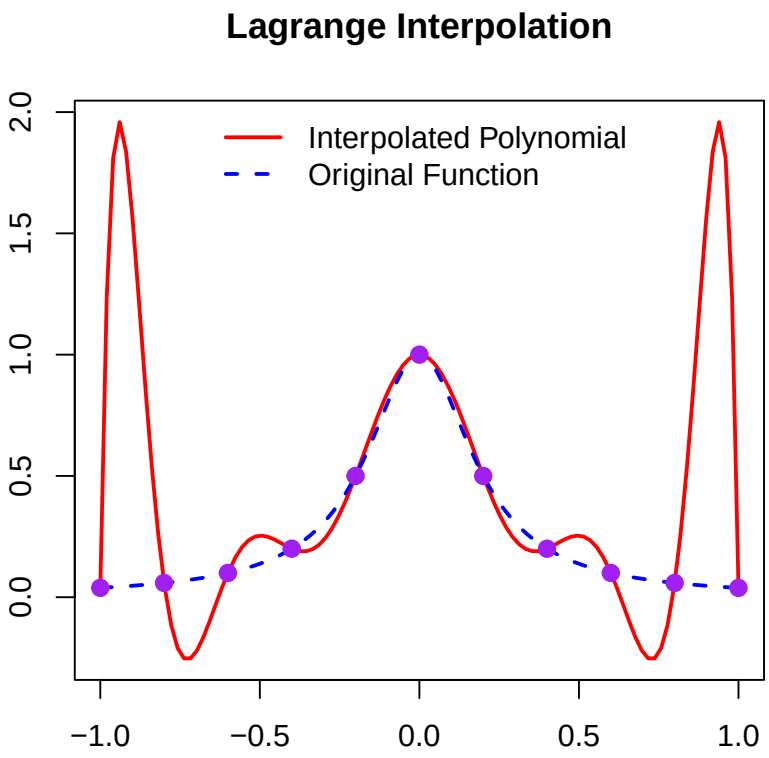
```
fn=function(x) exp(x)*cos(3*x)
approx.value = Lagrange.Interpolation.Vector(fn=fn, xvec=c(0, 0.3, 0.6),
                                             pred.x = c(0.3, 0.5))
true.value = fn(c(0.3, 0.5))
pander(rbind(true.value=true.value, approx.value = approx.value))
```

true.value	0.8391	0.1166
approx.value	0.8391	0.1251

Example 3: Consider Lagrange interpolation approximation of $f(x) = \frac{1}{1+25x^2}$. The x-nodes used in the approximation are (-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1). Plot the curves of $f(x)$ and $P_{10}(x)$.

Solution: Based on the given information, we use the above vector-based function to find the P_n(x) and create a sequence of 100 x-values from [-1, 1] that are equally spaced.

```
fn=function(x) 1/(1 + 25*x^2)
pred.x = seq(-1, 1, length = 100)
xvec = c(-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1)
##
approx.value = Lagrange.Interpolation.Vector(fn=fn, xvec=xvec,
                                             pred.x = pred.x)
true.value = fn(pred.x)
plot(pred.x, approx.value, type = "l", col = "red", lwd = 2, lty = 1,
            main = "Lagrange Interpolation",
             xlab = "",
             ylab = "")
lines(pred.x, true.value, lty = 2, lwd = 2, col = "blue")
points(xvec, fn(xvec), pch = 19, col = "purple", cex = 1.2)
legend("top", c("Interpolated Polynomial", "Original Function"),
       lwd=rep(2,2), lty=1:2, col = c("red", "blue"), bty="n")
```



10.4 Error Analysis

It is important to understand the nature of the error term when the Lagrange polynomial is used to approximate a continuous function $f(x)$.

We can see from the expression of Lagrange basis polynomials that the error term of Lagrange interpolation should be similar to that for the Taylor polynomial, except that the factor $(x - x_0)^{n+1}$ is replaced with the product $(x - x_0)(x - x1)\cdots(x - x_n)$. This is expected because interpolation is exact at each of the $n+1$ nodes x_k , where we have error term $E_n(x_k) = f(x_k) - P_n(x_k) = y_k - y_k = 0$ for $k = 0, 1, 2, \cdots, n$.

The following theorem specifies the error term of the Lagrange interpolation.

Theorem:Assume that $f \in C^{n+1}[a, b]$ and that $x_0, x_1, \cdots, x_n \in [a, b]$ are $n + 1$ nodes. If $x \in [a, b]$, then

$$f(x) = P_n(x) + E_n(x)$$

where $P_n(x)$ is a polynomial used to approximate $f(x)$

$$f(x) \approx P_n(x) = \sum_{j=0}^{n} f(x_j)L_j(x) = \sum_{j=0}^{n} y_j L_j(x)$$

The error term $E_n(x)$ has the form

$$E_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)$$

for some value $c = c(x)$ that lies in the interval $[a, b]$.

Proof (I only prove the special case of $n = 2$, i.e., with 3 interpolation nodes).

For given two nodes x_0, x_1x_2 and an arbitrarily chosen x, If $x = x_j(j = 0, 1, 2)$, then error $E_2(x) = 0$.

We now assume that the arbitrarily chosen value $x \neq x_j(j = 0, 1, 2)$.

Denote $w(t) = (t - x_0)(t - x_1)(t - x_2)$ and $\lambda = [f(x) - P_2(x)]/w(x)$. Clearly, $w(x) \neq 0$. We now define an auxiliary function $\phi(t) = f(t) - P_2(t) - \lambda w(t)$. Apparently, $\phi(x) = 0$, $\phi(x_j) = 0$ (for $j = 0, 1, 2$), and $\phi \in C^3[a, b]$.

since $f \in C^3[a, b]$ and $\phi(t) = 0$ has four roots, x, x_0, x_1, x_2 in $[a, b]$, there are 3 roots for $\phi'(t) = 0$ in the open interval (a, b) according to Rolle's Theorem (special case of the Mean Value Theorem). It follows that there are 2 roots for $\phi''(t) = 0$ in (a, b) and 1 root for $\phi'''(t) = 0$ in (a, b).

Therefore, $\exists c = c(x)$ so that $c(x) \in (a, b)$ and $\phi'''(c) = 0$. Note that, $\phi'''(t) = f'''(t)–P_2'''(t)–\lambda w'''(t)$.

From the definition of $P_2(t)$ and $w(t)$ we know that $P_2'''(t) = 0$ and $w'''(t) = 3!$. Plugging these values into the above equation, we have $\lambda'''(t) = f'''(t) - \lambda 3!$. Consequently, $f'''(c) - \lambda 3! = 0$. Recall that $\lambda = [f(x) - P_2(x)]/w(x)$ and $w(x) = (x - x_0)(x - x_1)(x - x_2)$. We rewrite $f'''(c) - \lambda 3! = 0$ as follows

$$f'''(c) - \frac{3![f(x) - P2(x)]}{(x - x0)(x - x1)(x - x2)} = 0$$

.

Therefore,

$$f(x) - P_2(x) = \frac{f'''(c)}{3!}(x - x_0)(x - x_1)(x - x_2).$$

The proof is completed.

The following corollary gives the error bound explicitly.

Corollary: Assume that $f \in C^{n+1}[a,b]$ and $|f^{(n+1)}(x)| \leq M$ for all $x \in [a,b]$. Assume also that nodes $x_0, x_1, \cdots, x_n \in [a,b]$ are equally spaced. If $x \in [a,b]$. Let $P_n(x)$ be the unique interpolating polynomial with degree $\leq n$ at the aforementioned equally spaced nodes. If $x \in [a,b]$, then

$$|f(x) - P_n(x)| \leq \frac{M}{4(n+1)} \left(\frac{b-a}{n}\right)^{n+1}.$$

Remark: If the $(n+1)$-th derivative is not uniformly bounded, the error bound in the above corollary should be in the following form

$$|f(x) - P_n(x)| \leq \frac{\max_{c\in[a,b]} |f^{(n+1)}(c)|}{4(n+1)} \left(\frac{b-a}{n}\right)^{n+1}.$$

Example 4: In $f(x) = \cos(x)$, the (n+1)-th derivative is uniformly bounded, one can force the error arbitrarily small by choosing the number of nodes.

Example 5: In $f(x) = 1/(1+x2)$, the derivative cannot be uniformly bounded. Increasing the number of nodes may not decrease the error.

Chapter 11

Newton Interpolation

Let $f(x)$ be a function whose values are known or can be calculated at a set of points (nodes) $\{x_0.x_1, \cdots, x_n\}$. Assume that these points are distinct, but NOT necessarily be ordered on the real line. There exists a polynomial $p_n(x)$ of degree at most n that interpolates $f(x)$ at $n+1$ nodes:

$$p_n(x) = f(x_i), \quad 0 \leq i \leq n.$$

We have discussed the Lagrange form of the interpolation polynomial. In this note, we introduce Newton's form of interpolation polynomial.

11.1 Some Definitions

We introduce several definitions related to the divided differences and how to calculate the dived difference manually and programmatically.

11.1.1 Definitions

The Newton form basis polynomials is defined in the following

$$\begin{array}{lcl} q_0(x) & = & a \\ q_1(x) & = & x - x_0 \\ q_2(x) & = & (x - x_0)(x - x_1) \\ \cdots & \cdots & \cdots \\ q_n(x) & = & (x - x_0)(x - x_1)\cdots(x - x_{n-1}) \end{array}$$

The $q_0(x), q_1(x), \cdots, 1_n(x)$ is basis of span $\{x^0, x^1, x^2, \cdots, x^n\}$.

The Newton Form Polynomial is defined as

$$p_n(x) = \sum_{i=0}^{n} c_i q_i(x).$$

Divided Differences are defined based on the coordinates of a set of given points on the curve of $f(x)$.

x	x_0	x_1	x_2	$\cdots$	x_n
$f(x)$	$f(x_0)$	$f(x_1)$	$f(x_2)$	$\cdots$	$f(x_n)$

For $i, j, k = 0, 1, 2, \cdots, n$,

- The zero-th order *divided difference* is $f[x_i] = f(x_i)$.
- The second order *divided difference* based on distinct nodes x_i and x_j is defined as
$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}.$$
  - This is the slope of the secant line that passes the two given points on the curve of $f(x)$.
  - It is also used to approximate the derivative of the function over interval $[x_i, x_j]$ (assuming $x_i < x_j$) if it exists.
- The third order *divided difference* based on the distinct nodes $x_i < x_j < x_k$ is defined as
$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k}$$
- The high order *divided difference* based on nodes $x_0 < x_1 < \cdots < x_n$ is similarly defined as
$$f[x_0, x_1, \cdots, x_n] = \frac{f[x_0, x_1, \cdots, x_{n-1}] - f[x_1, \cdots, x_n]}{x_0 - x_n}$$

11.1.2 Calculation of Divided Difference

The Newton interpolation is defined based on the divided difference. We first look at the structure of the divided difference and then develop an algorithm to compute the divided difference programmatically.

- *Iterative Nature of Divided Differences*: Consider the following given points on the curve of $f(x)$

x	x_0	x_1	x_2	x_3
$f(x)$	$f(x_0)$	$f(x_1)$	$f(x_2)$	$f(x_3)$

- Three *first-order divided differences* are defined based on the given 4 nodes. The first one based

$$f[x_0, x_1] = \frac{f(x_0) - f(x_1)}{x_0 - x_1}, \quad f[x_1, x_2] = \frac{f(x_1) - f(x_2)}{x_1 - x_2}, \quad f[x_2, x_3] = \frac{f(x_2) - f(x_3)}{x_2 - x_3},$$

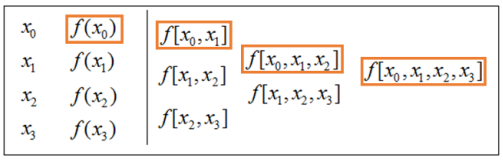
- Two second order *divided differences* are defined by

$$f[x_0, x_1, x_2] = \frac{f[x_0, x_1] - f[x_1, x_2]}{x_0 - x_2} = \frac{\frac{f(x_0)-f(x_1)}{x_0-x_1} - \frac{f(x_1)-f(x_2)}{x_1-x_2}}{x_0 - x_2},$$

$$f[x_1, x_2, x_3] = \frac{f[x_1, x_2] - f[x_2, x_3]}{x_1 - x_3} = \frac{\frac{f(x_1)-f(x_2)}{x_1-x_2} - \frac{f(x_2)-f(x_3)}{x_2-x_3}}{x_1 - x_3}$$

- The third-order divided difference is defined by

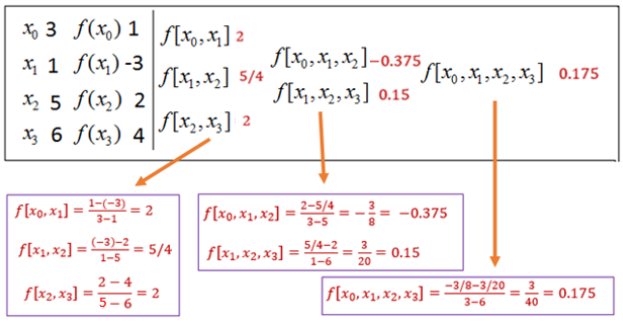
$$f[x_0, x_1, x_2, x_3] = \frac{f[x_0, x_1, x_2] - f[x_1, x_2, x_3]}{x_0 - x_3}$$



Example 1: Calculate all divided differences based on the following given data table

x	3	1	5	6
$f(x)$	1	-3	2	4

Based definition, the divided differences are calculated and summarized in the following table.



11.1.3 Algorithm of Calculating Divided Differences

We re-organize the above calculation in the following matrix and use the logic to develop the pseudo-code for calculating the divided differences.

Input Nodes $x[i]$

3	1	5	6

Divided Difference Matrix $A[i,j]$

$f[x_1]$	1	-3	2	4
$f[x_1,x_2]$	$\frac{f[x_1]-f[x_2]}{x_1-x_2}$	$\frac{f[x_2]-f[x_3]}{x_2-x_3}$	$\frac{f[x_3]-f[x_4]}{x_3-x_4}$	i = 2, j = 1, 2, 3 diff. x = x[j] − x[j + i − 1]
$f[x_1,x_2,x_3]$	$\frac{f[x_1,x_2]-f[x_2,x_3]}{x_1-x_3}$	$\frac{f[x_2,x_3]-f[x_3,x_4]}{x_2-x_4}$	i = 3, j = 1,2 diff. x = x[j] − x[j + i − 1]	
$f[x_1,x_2,x_3,x_4]$	$\frac{f[x_1,x_2,x_3]-f[x_2,x_3,x_4]}{x_1-x_4}$	i = 4, j = 1 diff. x = x[j] − x[j + i − 1]		

Divided Algorithm

```
INPUT: vec.x              (nodes)
       vec.y (or fn() )
       pred.x

OUTPUT: pn(pred.x)

STEP 1: Define a square zero matrix (array): A
STEP 2: Store vec.y in the first row of A
STEP 3: FOR i from 2 to ncol DO:      (Caution: 2nd row!)
            FOR j from 1 to (ncol - i + 1) DO:
                denominator = vec.x[j] - vec.x[j+i-1]
                numerator = A[i-1,j] - A[i-1, j+1]      (using previous row of A)
                A[i,j] = numerator / denominator        (next order divided difference)
            ENDFOR
        ENDFOR
STEP 4: RETURN A
```

```
Divided.Dif = function(
        vec.x,           # input nodes:
        vec.y = NULL,    # one of vec.y and fn must be given
        fn = NULL,
        pred.x           # scalar x for predicting pn(pred.x)
         ){
   n = length(vec.x)
   if (length(vec.y) == 0) vec.y = fn(vec.x) #
   node.x = vec.x
   A = matrix(c(rep(0,n^2)), nrow = n, ncol = n, byrow = TRUE)
   A[1,] = vec.y      # fill the first row with vec.y
   #
   for(i in 2:(n)){
     for(j in 1:(n-i+1)){
      denominator = vec.x[j] - vec.x[j+1+(i-2)]
      numerator = A[i-1,j]- A[i-1,j+1]
      A[i,j] = numerator/denominator
      }
    }
  A
 }
```

```
pander(Divided.Dif(
        vec.x = c(3,1,5,6),    # input nodes:
        vec.y = c(1,-3,2,4),   # one of vec.y and fn must be given
        fn = NULL
        )
)
```

1	-3	2	4
2	1.25	2	0
-0.375	0.15	0	0
0.175	0	0	0

Example 2: (Example 1 of Burden and Faires' textbook, 9th edition, page 127) Complete the divided difference table for the following data.

x	y
1.0	0.7651977
1.3	0.6200860
1.6	0.4554022
1.9	0.2818186
2.2	0.1103623

We use the above function to calculate the divided differences.

```
pander(Divided.Dif(
        vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
        vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
        fn = NULL
        )
    )
```

0.7652	0.6201	0.4554	0.2818	0.1104
-0.4837	-0.5489	-0.5786	-0.5715	0
-0.1087	-0.04944	0.01182	0	0
0.06588	0.06807	0	0	0
0.001825	0	0	0	0

The resulting table is the same as the one obtained in the table of the textbook (except for rounding errors).

11.2 Newton Interpolation Polynomial

From the definition of the divided difference, for any x and x_0, we have

$$f[x, x_0] = \frac{f(x) - f(x_0)}{x - x_0}$$

Solving for $f(x)$, we have

$$f(x) = f(x_0) + f[x, x_0](x - x_0).$$

Similarly, for x_0 and x_1, the second order divided difference is given by

$$f[x, x_0, x_1] = \frac{f[x, x_0] - f[x_0, x_1]}{x - x_1}$$

Therefore,

$$f[x, x_0] = f[x_0, x_1] + f[x, x_0, x_1](x - x_1).$$

That is,

$$f(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x, x_0, x_1](x - x_0)(x - x_1).$$

Repeating the above calculation, we have

$$f(x) = f[x_0]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+\cdots+f[x,x_0,\cdots,x_n](x-x_0)\cdots(x-x_{n-1})$$

11.2.1 Definition of Newton Interpolation

Based on the above derivation, we define **Newton Form Interpolation Polynomial** to be of the following form

$$N_n(x) = f[x_0]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+\cdots+f[x_0,\cdots,x_n](x-x_0)\cdots(x-x_{n-1})$$

We can see that $N_n(x)$ passes all interpolating points:

$$N_n(x_0) = f(x_0)$$

$$N_n(x_1) = f(x_0) + f[x_0,x_1](x_1-x_0) = f(x_0) + \frac{f(x_1)-f(x_0)}{x_1-x_0}(x_1-x_0) = f(x_1)$$

In general,

$$N_n(x_k) = f(x_k), \quad \text{for} \quad 0 \le k \le n.$$

11.2.2 Newton Interpolation Algorithm

The associated **divided differences** in the above **Newton Interpolation Polynomial** are returned in the first column of the function `Divided.Dif()`. To write the algorithmic function

Newton Interpolation Algorithm 2

```
INPUT:  vec.x
        vec.y or fn
        pred.x     (for prediction)
OUTPUT: pred.y     (pn(pred.x))

STEP 1: initialization:
        Divided.Dif          (function call)
        Pn = 0

STEP 2: FOR i from 1 to n DO:
          cumProd = 1             (initial value for the cumulative product)
          FOR j from 1 to i DO:
```

```
            cumProd = cumProd*(pred.x-vec.x[j])
          ENDFOR
          Pn = Pn + Divided.Dif[i]*cumProd
        ENDFOR
STEP 3: RETURN(Pn)
```

Next, we write an R function to implement the Newton interpolation polynomial.

```
NewtonInterpolation = function( vec.x,            # input interpolation nodes
                                vec.y = NULL,
                                fn = NULL,        # either vec.y or fn must be provide
                                pred.x            # single value of x for prediction
                              ){
   if(length(vec.y) ==0) vec.y = fn(vec.x)
   DivDif = Divided.Dif(vec.x, vec.y)[,1]       # the values in the first column of th
   n = length(vec.x)
   Nn = vec.y[1]                    # f[xo]
   for (i in 1:(n-1)){              # Must be n - 1 according to the last term in the po
     cumProd = 1                    # initial value to calculate the cumulative product
     for(j in 1:i){                 # forward difference formula
       cumProd = cumProd*(pred.x-vec.x[j])   # updating the cumulative product in the
     }
     Nn = Nn + DivDif[i+1]*cumProd    # adding high order terms alliteratively to the
   }
  Nn                                # return the value of the Newton polynomial
}
```

Example 3 (Continuation of *example 2*). We evaluate the Newton interpolation polynomial $p_4(x)$ at $x = 1.1, 1.6$ and 2.0, respectively. Recall the given data points are

x	y
1.0	0.7651977
1.3	0.6200860
1.6	0.4554022
1.9	0.2818186
2.2	0.1103623

Solution: We use the above R function to evaluate the function at the two given nodes.

```
pred.1.6 = NewtonInterpolation(vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
                    vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = 1.6)
pred.1.1 = NewtonInterpolation(vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
```

```
                    vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = 1.1)
pred.2.0 = NewtonInterpolation(vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
                    vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = 2.0)

pander(cbind(pred.1.6=pred.1.6, pred.1.1=pred.1.1, pred.2.0 = pred.2.0))
```

pred.1.6	pred.1.1	pred.2.0
0.4554	0.7196	0.2239

The results are the same as those obtained in the textbook Burden and Faires (9th edition, page 131).

R Function: Vector Enabled Newton Interpolated Polynomial

Next, we modify the previous R function to take a vector of input x-values for prediction just like other R functions.

```
##############################################################################
##  Newton Interpolated Polynomial Approximation: vector-enabled input
##############################################################################

Newton.Interpolation = function( vec.x,             # input interpolation nodes
                                 vec.y = NULL,
                                 fn = NULL,         # either vec.y or fn must be provided
                                 pred.x             # VECTOR INPUT!!!
                               ){
   if(length(vec.y) ==0) vec.y = fn(vec.x)
   DivDif = Divided.Dif(vec.x, vec.y)[,1]       # the values in the first column of the div dif m
   n = length(vec.x)
   #############
   m = length(pred.x)
   NV = rep(0, m)                  # values of Nn(pred.x)
   for(k in 1:m) {
   #################
   Nn = vec.y[1]                   # f[xo]
   for (i in 1:(n-1)){             # Must be n - 1 according to the last term in the polynomial
     cumProd = 1                   # initial value to calculate the cumulative product
     for(j in 1:i){                # forward difference formula
       cumProd = cumProd*(pred.x[k]-vec.x[j])   # updating the cumulative product in the inner lo
     }
      Nn = Nn + DivDif[i+1]*cumProd    # adding high order terms alliteratively to the Nn(x)
    }
    NV[k] = Nn                                  # return the value the Newton polynomial
```

```
   }
 NV
}
```

```
pred.x = c(1.6, 1.1, 2.0)   # pred.x is the argument is a local variable!
pred.NIP = Newton.Interpolation(vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
                    vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = c(1.6, 1.1, 2.0))
pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP))
```

pred.x	pred.NIP
1.6	0.4554
1.1	0.7196
2	0.2239

11.3 Error Analysis

Using the generalized Rolle theorem repeatedly on the expression of the divided difference, have the following result.

Theorem: Let $x_0, \cdots, x_{n-1}$, x be $n+1$ distinct points. Let $a = \min(x_0, \cdots, x_{n-1}, x)$ and $b = \max(x_0, \cdots, x_{n-1}, x)$. Assume that $f(x)$ has a continuous derivative of order n in the interval (a, b). Then

$$f[x_0, x_1, \cdots, x_{n-1}, x] = \frac{f^{(n)}(\xi)}{n!}$$

where $\xi \in (a, b)$.

Proof: Let $P_{n+1}(y)$ be the interpolated polynomial at y for given nodes $\{x_0, x_1, x_2, \cdots, x_{n-1}, x\}$ such that $P_{n+1}(x_i) = f(x_i)$ for $i = 0, 1, 2, \cdots, n-1$, and $P_{n+1}(x) = f(x)$. From the construction of Newton interpolating polynomial $P_n(x)$, we know that

$$P_n(y) = P_{n-1}(x) + f[x_0, x_1, x_2, \cdots, x_{n-1}, x](x - x_0) \cdots (x - x_{n-1})$$

Apparently,

$$f(x) = P_n(x) = P_{n-1}(x) + f[x0, x1, x2, \cdots, x_{n-1}, x](x - x_0) \cdots (x - x_{n-1})$$

Using the theorem introduced in the last unit and the uniqueness of the app, we have

$$f(x) = P_{n-1}(x) + \frac{f^{(n)}(\xi)}{n!}(x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

Using the same arguments in the Lagrange interpolation polynomial, we can establish the error bound for the Newton form interpolation polynomials.

11.4 Some Remarks of Newton Interpolation

First of all, both Lagrange and Newton interpolation polynomials introduced earlier can be viewed as a linear combination of basis polynomials $\{x^0, x^1, x^2, x^2, \cdots, x^n\}$. In fact, Lagrange and Newton interpolation polynomials are two different algebraic representations of the same polynomial.

That is, $P_n(x)$ and $N_n(x)$ be Lagrange and Newton interpolation polynomials based on $(x_0, y_0), (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$. We express both $P_n(x)$ and $N_n(x)$ in terms of basis polynomials $\{x^0, x^1, x^2, x^2, \cdots, x^n\}$ as

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \quad \text{and} \quad N_n(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n$$

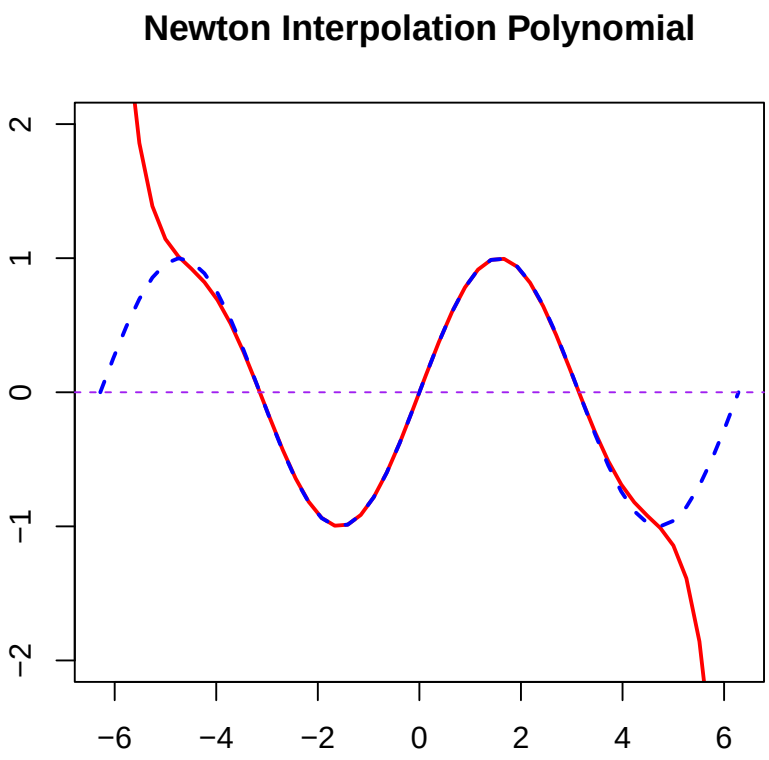
Since $P_n(x_i) = y_i = N_n(x_i)$ for $i = x_0, x_1, \cdots, x_n$, we can easily show that $a_j = b_j$ for $j = 0, 1, 2, \cdots, n$. Therefore, the error bounds of both Lagrange and Newton interpolation polynomials are the same.

For the Newton form interpolation polynomial, we only need to add more terms if additional interpolation nodes are added to the existing ones. Since the divided differences are independent of the newly added nodes. But for the Lagrange form, we need to restart the program. This is a unique feature of the Newton interpolation polynomial.

Example 4. Use the Newton Interpolation polynomial to approximate $f(x) = \sin(x)$ with interpolating nodes $x = -\pi, -0.75\pi, -0.5\pi, 0.25\pi, 0, 0.25\pi, 0.5\pi, 0.75\pi, \pi$.

Solution: We plot both $f(x) = \sin(x)$ and $N_8(x)$ over interval $[-2\pi, 2\pi]$ and see the performance of the approximation.

```
nodes = c(-1.5*pi, -pi, -0.5*pi, -0.25*pi, 0, 0.25*pi, 0.5*pi,  pi, 1.5*pi)
####
xx = seq(-2*pi, 2*pi, length = 50)
yy = sin(xx)
Nn = Newton.Interpolation(vec.x = nodes,
                          vec.y = sin(nodes),
                          pred.x = xx)
plot(xx, Nn, xlab = "", ylab = "", type = "l", lwd = 2, col = "red", ylim = c(-2,2),
        main = "Newton Interpolation Polynomial")
lines(xx, yy, lty =2, col = "blue", lwd = 2)
abline(h = 0, lty = 2, col = "purple")
```



11.5 Implicit Loops in R Programming

This lab note introduces how to reduce loops through vectorization. For any vectorized language, there are different extensions such as user-defined functions and routines to carry vectorized operation instead of element-wise operation. Most R functions are vectorized. We will use some examples to illustrate how a loop can be avoided if vectorization is available.

11.5.1 Implicit in R Primitive Functions

R was written in C. It has a long list of primitive functions written in C. Most of these functions are vectorized. Calling an internal vectorized function is the same as performing an implicit loop.

Example 1: Consider the sum of two matrices with the same dimension. Define

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 7 & 9 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 5 & 2 & 7 \\ 4 & 1 & 6 \end{pmatrix}$$

We find the sum of A and B

$$A + B = \begin{pmatrix} 6 & 4 & 10 \\ 7 & 8 & 15 \end{pmatrix}$$

Method 1: Using explicit loops. Since the sum of compatible matrices is an element-wise operation, in order to access individual elements, we need to use two indexes - one for the row and one for the column. We use the double loops to calculate the sum of two matrices in the following code.

```
loopSum = function(A,B){
    sumAB = matrix(0, nrow = dim(A)[1], ncol = dim(A)[2])
        for(i in 1:dim(A)[1]) {
            for (j in 1:dim(A)[2]){
                sumAB[i,j] = A[i,j] + B[i,j]
             }
         }
    sumAB
 }
```

We can also use R primitive function + to perform the matrix summation.

```
vectorSum = function(A,B) {A + B}
```

```
A = matrix(c(1,2,3,3,7,9), ncol = 3, byrow = TRUE)
B = matrix(c(5,2,7,4,1,6), ncol = 3, byrow = TRUE)
```

```
start <- Sys.time()
loopSum(A,B)
```

```
##      [,1] [,2] [,3]
## [1,]    6    4   10
## [2,]    7    8   15
```

```
print( Sys.time() - start )
```

```
## Time difference of 0.01143289 secs
```

```
start <- Sys.time()
vectorSum(A,B)
```

```
##      [,1] [,2] [,3]
## [1,]    6    4   10
## [2,]    7    8   15
```

```
print( Sys.time() - start )
```

```
## Time difference of 0.007189035 secs
```

Example 2: Summation of large matrices.

```
A0 = matrix(runif(10000000), ncol = 5000)
B0 = matrix(runif(10000000), ncol = 5000)
```

```
start <- Sys.time()
AplusB.lp = loopSum(A0,B0)
print( Sys.time() - start )
```

```
## Time difference of 1.090455 secs
```

```
start <- Sys.time()
AplusB.vec = vectorSum(A0,B0)
print( Sys.time() - start )
```

```
## Time difference of 0.1106699 secs
```

The obvious benefits of using an implicit vectorized primitive function to perform matrix operation:

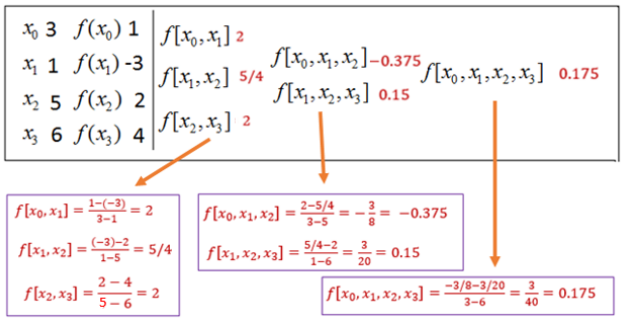
1. The code is simple.

2. It is faster.

11.5.2 Some Vectorized Primitive Functions

Since the numerator and denominator are defined as the difference between adjacent terms. We can vectorize these differences using the R function **diff()** that computes the difference between pairs of consecutive elements of a numeric vector.

Example 2: Consider `vec.x = (1, 1.3, 1.6, 1.9, 2.2)` and `vec.y = (0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623)`. We use the **diff()** to calculate the divided differences in the following table step by step.



The following are manual steps for calculating the divided differences in the above table.

- *Step 1*: zero-th order divided differences ($i = 1$)

```
vec.x = c(3,1,5,6)
vec.y = c(1,-3,2,4)
n = length(vec.x)
```

- *Step 2*: The first order divided differences (i = 2)

```
i=2
## divided difference
i2.y = diff(vec.y)
i2.x = vec.x[-(1:(i-1))] - vec.x[-((n+2-i):n)]
i2.divDif = i2.y/i2.x
cbind(i2.y = i2.y, i2.x = i2.x, i2.divDif = i2.divDif)
```

```
##      i2.y i2.x i2.divDif
## [1,]   -4   -2      2.00
## [2,]    5    4      1.25
## [3,]    2    1      2.00
```

- *Step 3*: The first order divided differences (i = 3)

```
i = 3
i3.y = diff(i2.divDif)    # Caution
i3.x = vec.x[-(1:(i-1))] - vec.x[-((n+2-i):n)]
i3.divDif = i3.y/i3.x
cbind(i3.y = i3.y, i3.x = i3.x, i3.divDif = i3.divDif)
```

```
##       i3.y i3.x i3.divDif
## [1,] -0.75    2    -0.375
## [2,]  0.75    5     0.150
```

- *Step 4*: The first order divided differences (i = 4)

```
i = 4
i4.y = diff(i3.divDif)
```

```
i4.x = vec.x[-(1:(i-1))] - vec.x[-((n+2-4):n)]
i4.divDif = i4.y/i4.x
cbind(i4.y = i4.y, i4.x = i4.x, i4.divDif = i4.divDif)
```

```
##       i4.y i4.x i4.divDif
## [1,] 0.525    3     0.175
```

11.5.3 Divided Difference Variants

Recall the definitions of the divided difference in the Newton form interpolation.

$$N_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots + f[x_0, \cdots, x_n](x - x_0) \cdots (x - x_{n-1})$$

x	$f(x)$	First divided differences	Second divided differences	Third divided differences
x_0	$f[x_0]$			
		$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$		
x_1	$f[x_1]$		$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$	
		$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}$		$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{x_3 - x_0}$
x_2	$f[x_2]$		$f[x_1, x_2, x_3] = \frac{f[x_2, x_3] - f[x_1, x_2]}{x_3 - x_1}$	
		$f[x_2, x_3] = \frac{f[x_3] - f[x_2]}{x_3 - x_2}$		$f[x_1, x_2, x_3, x_4] = \frac{f[x_2, x_3, x_4] - f[x_1, x_2, x_3]}{x_4 - x_1}$
x_3	$f[x_3]$		$f[x_2, x_3, x_4] = \frac{f[x_3, x_4] - f[x_2, x_3]}{x_4 - x_2}$	
		$f[x_3, x_4] = \frac{f[x_4] - f[x_3]}{x_4 - x_3}$		$f[x_2, x_3, x_4, x_5] = \frac{f[x_3, x_4, x_5] - f[x_2, x_3, x_4]}{x_5 - x_2}$
x_4	$f[x_4]$		$f[x_3, x_4, x_5] = \frac{f[x_4, x_5] - f[x_3, x_4]}{x_5 - x_3}$	
		$f[x_4, x_5] = \frac{f[x_5] - f[x_4]}{x_5 - x_4}$		
x_5	$f[x_5]$			

```
NewtonInterp = function(xvec,
                        yvec = NULL,
                        fn = NULL,
                        pred.x
                        ){
 if(length(yvec)==0) yvec = fn(xvec)
 n=length(xvec)
 DivDiff = rep(0,n)          # zero vector to store divided differences
 NewtonBasis = rep(0,1)      # zero vector to store Newton form basis polynomial
 DivDiff[1]=yvec[1]          # 1st order divided difference loaded to the first
 NewtonBasis[1]=1            # zero-degree basis polynomial
 old.NewtonBasis = 1         # initialize Newton basis polynomial for updating newer ba
 ##
 for (i in 2:n){
    NewtonBasis[i] = old.NewtonBasis*(pred.x-xvec[i-1])  # updating Basis polynomial
    dfx = xvec[-(1:(i-1))] - xvec[-((n-(i-2)):n)]         # denominator in the divided
    dy = diff(yvec)              # difference of lower order divided difference for th
    ##
```

```
    DivForm = dy/dfx                    # new vector of divided differences
    DivDiff[i]=DivForm[1]               # pick the top one store in the vector of DivDiff
    yvec = DivForm                      # updating for operation in the next row
    old.NewtonBasis = NewtonBasis[i]    # updating Newton basis polynomial
   }
 Nx=sum(DivDiff*NewtonBasis)            # predicted y value of the given pred.x
 list(Pred.y = Nx, DividedDifference = DivDiff, NewtonBasis = NewtonBasis)
}
```

Example 3: Reproduce the result in the above illustrative table.

```
vec.x = c(3,1,5,6)
vec.y = c(1,-3,2,4)
example1 = NewtonInterp(yvec = vec.y, xvec = vec.x, pred.x = 5)
example1
```

```
## $Pred.y
## [1] 2
##
## $DividedDifference
## [1]  1.000  2.000 -0.375  0.175
##
## $NewtonBasis
## [1] 1 2 8 0
```

```
sum(example1$DividedDifferenc* example1$NewtonBasis)
```

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

Example 5: Reproduce *Example 1* of Burden and Faires' textbook, 9th edition, page 127) Complete the divided difference table for the following data.

x	y
1.0	0.7651977
1.3	0.6200860
1.6	0.4554022
1.9	0.2818186
2.2	0.1103623

```
vec.x = c(1, 1.3, 1.6, 1.9, 2.2)
vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623)
example2 = NewtonInterp(yvec = vec.y, xvec = vec.x, pred.x = 1)
example2
```

```
## $Pred.y
## [1] 0.7651977
```

```
##
## $DividedDifference
## [1]  0.765197700 -0.483705667 -0.108733889  0.065878395  0.001825103
##
## $NewtonBasis
## [1] 1 0 0 0 0
```

```
sum(example2$DividedDifferenc* example2$NewtonBasis)
```

```
## [1] 0.7651977
```

11.5.4 Vectorizing NewtonInterp()

Next, we vectorize the above R function for Newton interpolation polynomial so that it can take a vector input. We will also compare this new function with the two R functions created in the lecture note.

```
Vectorizing.Newton = function(xvec,
                              yvec = NULL,
                              fn = NULL,
                              pred.x     # numerical vector or scalar input
                         ){
 if(length(yvec) == 0) yvec = fn(xvec)
 n = length(xvec)
 m = length(pred.x)             # dimension of the input vector for prediction
 NewtonPolynomial = rep(0, m)  # predicted value of the Newton interpolation polynomia
 for (k in 1:m){
    yvec0 = yvec     # CAUTION: Must be REINSTATED! diff(yvec) will change its origina
    DivDiff = rep(0,n)          # zero vector to store divided differences
    NewtonBasis = rep(0,1)      # zero vector to store Newton form basis polynomial
    DivDiff[1] = yvec[1]        # 1st order divided difference loaded to the first
    NewtonBasis[1] = 1          # zero-degree basis polynomial
    old.NewtonBasis = 1         # initialize Newton basis polynomial for updating the n
    ##
    for (i in 2:n){
       NewtonBasis[i] = old.NewtonBasis*(pred.x[k]-xvec[i-1])  # updating basis polyno
       dfx = xvec[-(1:(i-1))] - xvec[-((n-(i-2)):n)]     # denominator in the divided
       dy = diff(yvec0)          # difference of lower order divided difference for the
       DivForm = dy/dfx          # new vector of divided differences
       #cat("\n\n Inner loop:",i,". dfx =",dfx, ". dy =", dy, ". ")
       DivDiff[i] = DivForm[1]  # pick 1st component to store in vector DivDiff
       yvec0 = DivForm           # updating for operation in the next row
       old.NewtonBasis = NewtonBasis[i]
      }
      NewtonPolynomial[k] = sum(DivDiff*NewtonBasis)
      #cat("\n\n",k,"-th Pred.y:",NewtonPolynomial,", Diff:", DivDiff, ", Basis:", New
     }
```

```
 NewtonPolynomial
}
```

```
vec.x = c(1, 1.3, 1.6, 1.9, 2.2)
vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623)
Vectorizing.Newton(yvec = vec.y, xvec = vec.x, pred.x = c(1, 2))
```

```
## [1] 0.7651977 0.2238754
```

11.5.5 Code Efficiency with Implivit Loops

For comparison, we copy the two functions in the lecture note.

```
Divided.Dif = function(
        vec.x,          # input nodes:
        vec.y = NULL,   # one of vec.y and fn must be given
        fn = NULL,
        pred.x          # scalar x for predicting pn(pred.x)
         ){
   n = length(vec.x)
   if (length(vec.y) == 0) vec.y = fn(vec.x) #
   node.x = vec.x
   A = matrix(c(rep(0,n^2)), nrow = n, ncol = n, byrow = TRUE)
   A[1,] = vec.y     # fill the first row with vec.y
   #
   for(i in 2:(n)){
     for(j in 1:(n-i+1)){
      denominator = vec.x[j] - vec.x[j+1+(i-2)]
      numerator = A[i-1,j]- A[i-1,j+1]
      A[i,j] = numerator/denominator
      }
    }
  A
 }
```

```
##############################################################################
##  Newton Interpolated Polynomial Approximation: vector-enabled input
##############################################################################

Looping.Newton = function( vec.x,            # input interpolation nodes
                           vec.y = NULL,
                           fn = NULL,        # either vec.y or fn must be provided
                           pred.x            # VECTOR INPUT!!!
                          ){
   if(length(vec.y) ==0) vec.y = fn(vec.x)
   DivDif = Divided.Dif(vec.x, vec.y)[,1]       # the values in the first column of the div dif m
   n = length(vec.x)
```

```
   ############
   m = length(pred.x)
   NV = rep(0, m)                  # values of Nn(pred.x)
   for(k in 1:m) {
   ################
   Nn = vec.y[1]                   # f[xo]
   for (i in 1:(n-1)){             # Must be n - 1 according to the last term in the po
     cumProd = 1                   # initial value to calculate the cumulative product
     for(j in 1:i){                # forward difference formula
       cumProd = cumProd*(pred.x[k]-vec.x[j])   # updating the cumulative product in t
     }
      Nn = Nn + DivDif[i+1]*cumProd    # adding high order terms alliteratively to the
    }
    NV[k] = Nn                                  # return the value the Newton polynomi
   }
 NV
}
```

The following code shows the computational time the above two functions take in the approximation.

```
start <- Sys.time()
pred.x = c(1.6, 1.1, 2.0)   # pred.x is the argument is a local variable!
pred.NIP0 = Vectorizing.Newton(xvec = c(1, 1.3, 1.6, 1.9, 2.2),
                    yvec = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = c(1.6, 1.1, 2.0))
pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP0))
```

pred.x	pred.NIP
1.6	0.4554
1.1	0.7196
2	0.2239

```
print( Sys.time() - start )
```

```
## Time difference of 0.01160407 secs
```

```
start <- Sys.time()
pred.x = c(1.6, 1.1, 2.0)   # pred.x is the argument is a local variable!
pred.NIP = Looping.Newton(vec.x = c(1, 1.3, 1.6, 1.9, 2.2),
                    vec.y = c(0.7651977, 0.6200860, 0.4554022, 0.2818186, 0.1103623),
                    pred.x = c(1.6, 1.1, 2.0))
pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP))
```

pred.x	pred.NIP
1.6	0.4554
1.1	0.7196
2	0.2239

```
print( Sys.time() - start )
```

```
## Time difference of 0.02802587 secs
```

Chapter 12

Gauss Elimination Method

We discuss numerical solutions to the linear system of equations. Consider the following general linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad & \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

The goal is to find the solution to this system. There are different methods to solve the linear system. One way is to re-express the linear system into a matrix equation and then solve the matrix equation.

Example 1 Consider the following linear system of functions

$$\begin{aligned} 3x_1 - x_2 + 2x_3 &= 8 \\ 2x_1 - 2x_2 + 3x_3 &= 2 \\ 4x_1 + x_2 - 4x_3 &= 9 \end{aligned}$$

We can re-write the above system of linear equations into the following matrix form

$$\underbrace{\begin{bmatrix} 3 & -1 & 2 \\ 2 & -2 & 3 \\ 4 & 1 & -4 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} 8 \\ 2 \\ 9 \end{bmatrix}}_{\mathbf{B}}$$

We denote

$$\mathbf{A} = \begin{bmatrix} 3 & -1 & 2 \\ 2 & -2 & 3 \\ 4 & 1 & -4 \end{bmatrix}$$

be the coefficient matrix.

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

the vector of unknown, and

$$\mathbf{B} = \begin{bmatrix} 8 \\ 2 \\ 9 \end{bmatrix}$$

Then, the matrix form of the equation is equivalent to $\mathbf{AX} = \mathbf{B}$. If the inverse of $\mathbf{A}$, denoted by $\mathbf{A}^{-1}$, exists, we multiply $\mathbf{A}^{-1}$ to the left of the matrix equation to get the solution of the linear system in the following

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$$

Finding the inverse of the matrix could be a challenge when the system is large. The R built-in function `solve()` to calculate the inverse of a matrix (if it exists). For example, the inverse of $\mathbf{A}$ can be found using the following code.

```
A = matrix(c(3, -1, 2, 2, -2, 3, 4, 1, -4), ncol = 3, byrow = TRUE)
A.inv = solve(A)
A.inv        # print out the
```

```
##           [,1]       [,2]        [,3]
## [1,] 0.3333333 -0.1333333  0.06666667
## [2,] 1.3333333 -1.3333333 -0.33333333
## [3,] 0.6666667 -0.4666667 -0.26666667
```

```
# matrix multiplication
A%*%A.inv    # expected to be the identity matrix
```

```
##      [,1]          [,2]          [,3]
## [1,]    1 -2.220446e-16 -2.220446e-16
## [2,]    0  1.000000e+00 -1.110223e-16
## [3,]    0  0.000000e+00  1.000000e+00
```

Two ways to find the solution to the system

```
B = c(8,2,9)
X.1 = solve(A)%*%B  # returns a matrix form solution
X.1
```

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

```
X.2 = solve(A,B)   # returns a vector form solution
X.2
```

```
## [1] 3 5 2
```

12.1 Naive Gaussian Elimination Methods

Linear algebra texts discuss elimination methods to find the solution. The method has two processes: elimination and backward substitution. We use the equation in the above **Example 1** to review the elimination method.

Recall the linear system

$$\begin{array}{rcl} 3x_1 - x_2 + 2x_3 & = & 8 \\ 2x_1 - 2x_2 + 3x_3 & = & 2 \\ 4x_1 + x_2 - 4x_3 & = & 9 \end{array}$$

We define the following augmented matrix.

$$\mathbf{A} = \left[\begin{array}{cccc} 3 & -1 & 2 & \vdots & 8 \\ 2 & -2 & 3 & \vdots & 2 \\ 4 & 1 & -4 & \vdots & 9 \end{array}\right]$$

Solution We show the detailed steps of forward elimination and backward substitution.

Forward Elimination

We use R_1, R_2, and R_3 to denote the three rows of the above-augmented matrix and then find the solution in the following $r = 3 - 1$ steps:

Step 1 The equation in the first row is called **pivot equation** and $a_{11} = 3$ is called **pivot element**. We first do the following row operations so that the elements in the first column of the resulting **new augmented** matrix are all zeros except for the one in the *pivot* element.

1.1. Multiply $-a_{21}/a_{11}$ to R_1 and add it to R_2 to update R_2. **Row 2**: $i = 2$

$$R_2 \leftarrow R_2 - (a_{21}/a_{11}) \times R_1 = (a_{21}, a_{22}, a_{23}, b_2) - \frac{a_{21}}{a_{11}}(a_{11}, a_{12}, a_{13}, b_1)$$

$$= (a_{21} - \frac{a_{21}}{a_{11}} \times a_{11}, a_{22} - \frac{a_{21}}{a_{11}} \times a_{12}, a_{23} - \frac{a_{21}}{a_{11}} \times a_{11}, b_2 - \frac{a_{21}}{a_{11}} \times a_{11})$$

$$= (2, -2, 3, 2) - \frac{2}{3}(3, -1, 2, 8) = (0, -4/3, 5/3, -10/3)$$

The above step can be computed using the implicit loop through vector operation in any vectorized programming language

1.2 Multiply $-4/3$ to R_1 and add it to R_3 to update R_3. **Row 3**: $i = 3$

$$R_3 \leftarrow R_3 - (a_{31}/a_{11}) \times R1 = (a_{31}, a_{32}, a_{33}, b_3) - \frac{a_{31}}{a_{11}}(a_{11}, a_{12}, a_{13}, b_1)$$

$$= (4, 1, -4, 9) - \frac{4}{3}(3, -1, 2, 8) = \left(0, \frac{7}{3}, -\frac{20}{3}, -\frac{5}{3}\right)$$

After this step, we have the updated augmented matrix for the next step elimination.

$$\left[\begin{array}{cccc:c} 3 & -1 & 2 & \vdots & 8 \\ 0 & -4/3 & 5/3 & \vdots & -10/3 \\ 0 & 7/3 & -20/3 & \vdots & -5/3 \end{array}\right]$$

Step 2: The equation in the first row is called *pivot equation* and $a_{22} = -4/3$ (**CAUTION**: *using the above updated augmented matrix!*)is called pivot element. We next do row operation: $R_3 - \frac{a_{32}}{a_{22}} \times R_2$ so that the resulting new augmented matrix. **Row 3**: $r = 3$.

$$R_3 \leftarrow R_3 - \frac{a_{32}}{a_{22}} \times R_2 = (a_{31}, a_{32}, a_{33}, b_3) - \frac{a_{32}}{a_{22}}(a_{21}, a_{22}, a_{23}, b_2)$$

$$= \left(0, \frac{7}{3}, -\frac{20}{3}, -\frac{5}{3}\right) - \frac{7/3}{-4/3}\left(0, -\frac{4}{3}, \frac{5}{3}, -\frac{-10}{3}\right) = \left(0, \quad 0, \quad -\frac{15}{4}, -\frac{15}{2}\right)$$

With the above row operation, we have the following updated augmented matrix that has a triangular coefficient matrix.

$$\left[\begin{array}{cccc:c} 3 & -1 & 2 & \vdots & 8 \\ 0 & -4/3 & 5/3 & \vdots & -10/3 \\ 0 & 0 & -15/4 & \vdots & -15/2 \end{array}\right].$$

If we re-write the above-augmented matrix into the linear system of equations, we have

$$\begin{cases} 3x_1 & -x_2 & 2x_3 & = & 8 \\ & -\frac{4}{3}x_2 & \frac{5}{3}x_3 & = & -\frac{10}{3} \\ & & -\frac{15}{4}x_3 & = & -\frac{15}{2} \end{cases}$$

Backward Substitution

Next, we do backward substitution to find the solution to the system. The general backward substitution is given by

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{3} a_{ij} x_j \right),$$

where $i = 3-1, 3-2$. However $i = 3$

$$x_3 = \frac{b_3}{a_{33}} = \frac{-15/2}{-15/4} = 2.$$

We start from **row 3** of the last augmented matrix:

Row #3: $r = 3$:

$$x_3 = \frac{b_3}{a_{33}} = \frac{-15/2}{-15/4} = 2.$$

Row #2: $r = 2$

$$x_2 = \frac{1}{a_{22}} \left(b_2 - a_{23} x_3 \right) = -\frac{1}{4/3} \left(-\frac{10}{3} - \frac{5}{3} \times 2 \right) = \frac{3}{4} \frac{20}{3} = 5.$$

Row #1: $r = 1$

$$x_1 = \frac{1}{a_{11}} \left(b_1 - [a_{12} x_2 + a_{13} x_3] \right) = \frac{1}{3} \left(8 - [-1 \times 5 + 2 \times 2] \right) = \frac{1}{3}(8 - (-1)) = 3.$$

Therefore, the solution to the above system linear equation is $(x_1, x_2, x_3) = (3, 5, 2)$.

Remark: We re-arrange the following linear system of linear equations

$$\begin{cases} 3x_1 & -x_2 & 2x_3 & = & 8 \\ & -\frac{4}{3}x_2 & \frac{5}{3}x_3 & = & -\frac{10}{3} \\ & & -\frac{15}{4}x_3 & = & -\frac{15}{2} \end{cases}$$

to get the following equivalent linear system

$$\left\{\begin{array}{cccc} -\frac{15}{4}x_3 & & = & -\frac{15}{2} \\ \frac{5}{3}x_3 \quad -\frac{4}{3}x_2 & & = & -\frac{10}{3} \\ 2x_3 \quad -x_2 \quad +3x_1 & = & 8 \end{array}\right.$$

That can be re-written in the following matrix form

$$\left[\begin{array}{ccc} -\frac{15}{4} & 0 & 0 \\ \frac{5}{3} & -\frac{4}{3} & 0 \\ 2 & -1 & 3 \end{array}\right]\left[\begin{array}{c} x_3 \\ x_2 \\ x_1 \end{array}\right] = \left[\begin{array}{c} -\frac{15}{2} \\ -\frac{10}{3} \\ 8 \end{array}\right]$$

The augmented matrix of the above linear system of equations can be written as

$$\left[\begin{array}{ccc:c} -\frac{15}{4} & 0 & 0 & -\frac{15}{2} \\ \frac{5}{3} & -\frac{4}{3} & 0 & -\frac{10}{3} \\ 2 & -1 & 3 & 8 \end{array}\right]$$

We can repeat the Gaussian **forward elimination** with the above-augmented matrix:

1. pivot element $a_{11} = -\frac{15}{4}$

$$R_2 \leftarrow R_2 - \frac{a_{21}}{a_{11}}R_1 = \left(\frac{5}{3}, -\frac{4}{3}, 0, -\frac{10}{3}\right) - \frac{5/3}{-15/4}\left(-\frac{15}{4}, 0, 0, -\frac{15}{2}\right) = \left(0, -\frac{4}{3}, 0, -\frac{20}{3}\right)$$

$$R_3 \leftarrow R_3 - \frac{a_{31}}{a_{11}}R_1 = (2, -1, 3, 8) - \frac{2}{-15/4}\left(-\frac{15}{4}, 0, 0, -\frac{15}{2}\right) = (0, -1, 3, 4)$$

We obtain the following updated augmented matrix

$$\left[\begin{array}{ccc:c} -\frac{15}{4} & 0 & 0 & -\frac{15}{2} \\ 0 & -\frac{4}{3} & 0 & -\frac{20}{3} \\ 0 & -1 & 3 & 4 \end{array}\right]$$

We need one more row operation on the above-augmented matrix

$$R_3 \leftarrow R_3 - \frac{a_{32}}{a_{22}}R_2 = (0, -1, 3, 4) - \frac{-1}{-4/3}\left(0, -\frac{4}{3}, 0, -\frac{20}{3}\right) = (0, 0, 3, 9)$$

The final augmented matrix has the following form

$$\left[\begin{array}{cccc:c} -\frac{15}{4} & 0 & 0 & \vdots & -\frac{15}{2} \\ 0 & -\frac{4}{3} & 0 & \vdots & -\frac{20}{3} \\ 0 & 0 & 3 & \vdots & 9 \end{array}\right].$$

Its equivalent linear system is given by

$$\left\{\begin{array}{rcc} -\frac{15}{4}x_3 & = & -\frac{15}{2} \\ -\frac{4}{3}x_2 & = & -\frac{20}{3} \\ 3x_1 & = & 9 \end{array}\right.$$

Therefore, the solution to the original system of equation is (using vector operation - pair-wise division)

$$(x_3, x_2, x_1) = \frac{(-15/2, -20/3, 9)}{(-15/4, -4/3, 3)} = (2, 5, 3)$$

We will apply the logic used in this example to implement the **Gaussian Elimination method** (using *backward substitution*).

12.2 Gaussian Elimination Algorithm

In the elimination step, the first row of the augmented matrix is the basis, and all other elements (cells) in the augmented matrix are updated iteratively using the algorithm using the recursive formula

$$R_i \leftarrow (a_{i1}, a_{i2}, \cdots, a_{in}, b_i) - \frac{a_{ik}}{a_{kk}}(a_{k1}, a_{k2}, \cdots, a_{kn}, b_k)$$

where

- $k = 1, 2, \cdots, n-1$, the iteration index for selecting the *pivot element*.
- $i = k+1, k+2, ...n$, the iterator of the row of the augmented matrix.

We focus on Gaussian row elimination. The Gaussian forward elimination and backward substitution (Gauss-Jordan) are optional.

The algorithm of Gaussian is relatively simple compared with the iterative algorithms we learned earlier. We only need to convert the following recursive equation to an algorithm with double loops to accomplish the Gaussian forward elimination.

Gaussian Forward Elimination Algorithm

```
INPUT: A    (augmented matrix)

OUTPUT: A0  (row-echelon matrix)

STEP 1: find n = # of row

STEP 2: FOR i = 1 TO n
        STEP 3:  IF A[i,i] == 0 DO:
                    find j with A[j,i] != 0
                    swap row i and row j
                 ENDIF

        STEP 4:  IF A[i,i] != 0 DO:
                    FOR j = i + 1 TO n DO:
                       A[j,] = A[j,]-(A[j,i]/A[i,i])*A[i,]
                    ENDFOR
                 ENDIF
STEP 5: RETURN A0
```

The vector operation was used in **Step 4** of the pseudo-code. Row-swapping in **step 3** could also involve vector operation.

Next, we write R code to implement the pseudo-code.

```r
Gauss.Elimination = function(A){
  # Input A: augmented matrix. Make sure that he diagonal
  #          elements of the coefficient matrix are non-zero!
  A0 = A
  n = dim(A0)[1]                 # number of rows
  for(i in 1:(n-1)){             # iterator fo pivot element A0[i,i]
                                 # i = 1, 2, ..., n-1
      #~~~~~~~~~~~~~~~~~~~~   Make the function robust  ~~~~~~~~~~~~~~~~~~~~~~~~#
      if(A0[i,i] == 0){
          non.0 = which(A0[,i] != 0)
              #cat("\n\n i =",i, ", non.0 =", non.0,". (i+1):n =", (i+1):n, ".")
              id = intersect(non.0, (i+1):n)[1] # which() equiv to find() in MATLAB
              if(!is.na(id)){
              tempi = A0[i,]                    # put the i-th row in a place-holder
              A0[i,] = A0[id,]                  # row swapping
              A0[id,] = tempi
            }
      }
      #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
       if(A0[i,i] != 0){    # A0[i,i] is in the denominator of recursive equation
          for(j in (i+1):n){  # to make the augmented matrix in the row echelon form
             #cat("\n\n j =",j,".")
             A0[j,] = A0[j,] - (A0[j,i]/A0[i,i])*A0[i,]
```

```
            }
          }
        }
    A0
}
```

We now look at several examples using the above R function `Gauss.Elimination()`.

Example 2: Find the solution of the following system of linear equations.

$$\begin{aligned} x - 3y + 4z &= 1 \\ 2w - 2x + y &= -1 \\ 2w - x - 2y + 4z &= 0 \\ -6w + 4x + 3y - 8z &= 1 \end{aligned}$$

Solution: Performing Gaussian eliminations (i.e., the series of row operations), we have

$$\begin{bmatrix} 0 & 1 & -3 & 4 & 1 \\ 2 & -2 & 1 & 0 & -1 \\ 2 & -1 & -2 & 4 & 0 \\ -6 & 4 & 3 & -8 & 1 \end{bmatrix} \xrightarrow{r_1 \leftrightarrow r_2} \begin{bmatrix} 2 & -2 & 1 & 0 & -1 \\ 0 & 1 & -3 & 4 & 1 \\ 2 & -1 & -2 & 4 & 0 \\ -6 & 4 & 3 & -8 & 1 \end{bmatrix}$$

$$\xrightarrow[3r_1 \text{ added to } r_4]{-r_1 \text{ added to } r_3} \begin{bmatrix} 2 & -2 & 1 & 0 & -1 \\ 0 & 1 & -3 & 4 & 1 \\ 0 & 1 & -3 & 4 & 1 \\ 0 & -2 & 6 & -8 & -2 \end{bmatrix}$$

$$\xrightarrow[2r_2 \text{ added to } r_4]{-r_2 \text{ added to } r_3} \begin{bmatrix} 2 & -2 & 1 & 0 & -1 \\ 0 & 1 & -3 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
##
A1 = matrix(c(0,1,-3,4,1,2,-2,1,0,-1,2,-1,-2,4,0,-6,4,3,-8,1), ncol =5, byrow = TRUE)
Gauss.Elimination(A = A1)
```

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

Example 3 Solve the following system of linear equations.

$$\begin{aligned} x + y - 3z &= 4 \\ 2x + y - z &= 2 \\ 3x + 2y - 4z &= 7 \end{aligned}$$

Solution: Performing Gaussian eliminations (i.e., the series of row operations), we have

$$\begin{bmatrix} 1 & 1 & -3 & 4 \\ 2 & 1 & -1 & 2 \\ 3 & 2 & -4 & 7 \end{bmatrix} \xrightarrow[-3r_1 \text{ added to } r_3]{-2r_1 \text{ added to } r_2} \begin{bmatrix} 1 & 1 & -3 & 4 \\ 0 & -1 & 5 & -6 \\ 0 & -1 & 5 & -5 \end{bmatrix}$$

$$\xrightarrow{-r_2 \text{ added to } r_3} \begin{bmatrix} 1 & 1 & -3 & 4 \\ 0 & -1 & 5 & -6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
A1 = matrix(c(1,1,-3,4,2,1,-1,2,3,2,-4,7), ncol = 4, byrow = TRUE)
Gauss.Elimination(A = A1)
```

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

Example 4 Solve the following system of linear equations.

$$\begin{aligned} \tfrac{1}{4}a + \tfrac{1}{2}b + c &= \tfrac{23}{4} \\ a + b + c &= 7 \\ 4a + 2b + c &= 2 \end{aligned}$$

Solution: Performing Gaussian eliminations (i.e., the series of row operations), we have

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & 1 & \frac{23}{4} \\ 1 & 1 & 1 & 7 \\ 4 & 2 & 1 & 2 \end{bmatrix} \xrightarrow{4r_1} \begin{bmatrix} 1 & 2 & 4 & 23 \\ 1 & 1 & 1 & 7 \\ 4 & 2 & 1 & 2 \end{bmatrix}$$

$$\xrightarrow[-4r_1 \text{ added to } r_3]{-r_1 \text{ added to } r_2} \begin{bmatrix} 1 & 2 & 4 & 23 \\ 0 & -1 & -3 & -16 \\ 0 & -6 & -15 & -90 \end{bmatrix}$$

$$\xrightarrow[-r_2]{-6r_2 \text{ added to } r_3} \begin{bmatrix} 1 & 2 & 4 & 23 \\ 0 & 1 & 3 & 16 \\ 0 & 0 & 3 & 6 \end{bmatrix}$$

```
A1 = matrix(c(1/4, 1/2, 1, 23/4, 1,1,1,7,4,2,1,2), ncol = 4, byrow = TRUE)
Gauss.Elimination(A = A1)
```

```
##      [,1] [,2] [,3]   [,4]
## [1,] 0.25  0.5    1   5.75
## [2,] 0.00 -1.0   -3 -16.00
```

```
## [3,] 0.00  0.0    3   6.00
```

12.3 Gauss-Jordan Elimination (Optional)

```
####
Gauss.Jordan = function(A){
  # A is the augmented matrix
  A1 = Gauss.Elimination(A)           # perform Gauss elimination
  n = length(A1[,1])
  id.0 = rep(1,n)                     # indicator of zero A[i,i]
  for (i in 1:n){
    if(A1[i,i] == 0) id.0[i] = 0
  }
  if(sum(id.0) == n){                  # A[i,i] != 0
        k = n
        A2 = A1[k:1, c(k:1,(n+1))]     # re-arrange rows and columns
        A3 = Gauss.Elimination(A2)     # perform Gauss elimination
        A4 = A3[k:1, c(k:1,(n+1))]     # backward re-arrangement
      } else{                          # if exists A[i,i] =0
        k = which(id.0==0)[1]-1   # select rows and columns for re-arrangement
        A2 = A1[c(k:1,(k+1):n), c(k:1,(k+1):(n+1))]
        A3 = Gauss.Elimination(A2)
        A4 = A3[c(k:1,(k+1):n), c(k:1,(k+1):(n+1))]
      }
   A4
}
```

Example 5 (continuation of **Example 4**)

We can perform additional row operations (Gaussian-Jordan) to get

$$\begin{bmatrix}1&2&4&23\\0&1&3&16\\0&0&3&6\end{bmatrix} \xrightarrow[\frac{1}{3}r_3]{-r_3 \text{ added to } r_2} \begin{bmatrix}1&2&4&23\\0&1&0&10\\0&0&1&2\end{bmatrix} \xrightarrow{-4r_3 \text{ added to } r_1} \begin{bmatrix}1&2&0&15\\0&1&0&10\\0&0&1&2\end{bmatrix} \xrightarrow{-2r_2 \text{ added to } r_1} \begin{bmatrix}1&0&0&-5\\0&1&0&10\\0&0&1&2\end{bmatrix}$$

```
A1 = matrix(c(1/4, 1/2, 1, 23/4, 1,1,1,7,4,2,1,2), ncol = 4, byrow = TRUE)
Gauss.Jordan(A = A1)
```

```
##      [,1] [,2] [,3]   [,4]
## [1,] 0.25    0    0  -1.25
```

```
## [2,] 0.00   -1    0 -10.00
## [3,] 0.00    0    3   6.00
```

Example 6 (continuation of **Example 2**) Gauss-Jordan elimination

```
A6 = matrix(c(0,1,-3,4,1,2,-2,1,0,-1,2,-1,-2,4,0,-6,4,3,-8,1), ncol =5, byrow = TRUE)
Gauss.Jordan(A = A6)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    0   -5    8    1
## [2,]    0    1   -3    4    1
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
```

Example 7 (continuation of **Example 3**) Gauss-Jordan form

```
A7 = matrix(c(1,1,-3,4,2,1,-1,2,3,2,-4,7), ncol = 4, byrow = TRUE)
Gauss.Jordan(A = A7)
```

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

Example 8 (continuation of **Example 4**) Gauss-Jordan form

```
A8 = matrix(c(1/4, 1/2, 1, 23/4, 1,1,1,7,4,2,1,2), ncol = 4, byrow = TRUE)
Gauss.Jordan(A = A8)
```

```
##      [,1] [,2] [,3]   [,4]
## [1,] 0.25    0    0  -1.25
## [2,] 0.00   -1    0 -10.00
## [3,] 0.00    0    3   6.00
```

Chapter 13

Determinant and Inversion of Matrices

We have introduced the Gaussian elimination method to solve the linear system of equations. This note focuses on the algorithm for finding the inverse of square matrices and then use the algorithm to solve matrix equations,

13.1 Concepts of Matrix: A Review

We first define some special matrices and then review the basic properties of matrices.

13.1.1 Definitions of Some Special Matrices

A general form of a $n \times m$ matrix is given by

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1.2} & \cdots & a_{n-1,m} \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

That n rows and m columns. If $n = m$, the corresponding matrix is called the **square matrix**. If all off-diagonal elements of a square matrix are equal to zero, the square matrix is called **diagonal matrix**

$$D_{n\times n} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

A very important special diagonal matrix with all diagonal elements being 1 is called **identity matrix** and has the following form

$$I_{n\times n} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

The following square matrix is called a symmetric matrix

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1,n-1} & a_{2,n-1} & \cdots & a_{n,n-1} \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}$$

A square matrix is called lower triangular if all the entries above the main diagonal are zero. Similarly, a square matrix is called upper triangular if all the entries below the main diagonal are zero. S and T represent lower and upper triangular matrices respectively.

$$S_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \quad \text{and} \quad T_{n\times n} = \begin{bmatrix} b_{11} & 0 & \cdots & 0 \\ b_{21} & b_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & 0 \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

13.1.2 Binary Operations

We perform matrix addition and subtraction, the two matrices must have the same dimensions.

Addition and Subtraction

Let

$$A_{n\times m}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1.2} & \cdots & a_{n-1,m} \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \text{ and } B_{n\times m}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n-1,1} & b_{n-1.2} & \cdots & b_{n-1,m} \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix}$$

$$A_{n\times m}\pm B_{n\times m}=\begin{bmatrix} a_{11}\pm b_{11} & a_{12}\pm b_{12} & \cdots & a_{1m}\pm b_{1m} \\ a_{21}\pm b_{21} & a_{21}\pm b_{22} & \cdots & a_{21}\pm b_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,1}\pm b_{n-1,1} & a_{n-1,1}\pm b_{n-1.2} & \cdots & a_{n-1,1}\pm b_{n-1,m} \\ a_{n1}\pm b_{n1} & a_{n1}\pm b_{n2} & \cdots & a_{nm}\pm b_{nm} \end{bmatrix}$$

Multiplication

When multiplying two matrices, the two matrices must be compatible. That is the number of **columns** of the first (left) matrix MUST be equal to the number of **rows** in the second (right) matrix. To illustrate this compatibility, we define

$$C_{p\times q}=\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ c_{p-1,1} & c_{p-1,2} & \cdots & c_{p-1,m} \\ c_{p1} & c_{p2} & \cdots & c_{pq} \end{bmatrix} \text{ and } D_{q\times k}=\begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1k} \\ d_{21} & d_{22} & \cdots & d_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ d_{q-1,1} & d_{q-1.2} & \cdots & d_{q-1kk} \\ d_{q1} & d_{q2} & \cdots & d_{qk} \end{bmatrix}$$

The product of the above two matrices is defined to be

$$M_{p\times k}=\begin{bmatrix} c_{11}\times d_{11}+c_{12}\times d_{21}+\cdots+c_{1q}\times d_{q1} & \cdots & c_{11}\times d_{1k}+c_{12}\times d_{2k}+\cdots+c_{1q}\times d_{qk} \\ c_{21}\times d_{11}+c_{22}\times d_{21}+\cdots+c_{2q}\times d_{q1} & \cdots & c_{21}\times d_{1k}+c_{22}\times d_{2k}+\cdots+c_{2q}\times d_{qk} \\ \vdots & \vdots & \vdots \\ c_{p-1,1}\times d_{11}+c_{p-1,2}\times d_{21}+\cdots+c_{p-1,q}\times d_{q1} & \cdots & c_{p-1,1}\times d_{1k}+c_{p-1,2}\times d_{2k}+\cdots+c_{p-1,q}\times d_{qk} \\ c_{p1}\times d_{11}+c_{p2}\times d_{21}+\cdots+c_{pq}\times d_{q1} & \cdots & c_{p1}\times d_{1k}+c_{p2}\times d_{2k}+\cdots+c_{pq}\times d_{qk} \end{bmatrix}$$

In general, multiplication is NOT commutative. That is, $A\times B\neq B\times A$.

Example 1: Define

$$C_{3\times 4}=\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 3 & 5 & 2 \end{bmatrix} \text{ and } D_{4\times 3}=\begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 2 \\ 0 & 1 & 3 \\ 1 & 2 & 5 \end{bmatrix}$$

Then

$$C_{3\times4} \times D_{4\times3} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 3 & 5 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 2 \\ 0 & 1 & 3 \\ 1 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 12 & 16 & 37 \\ 18 & 14 & 33 \\ 13 & 16 & 35 \end{bmatrix}.$$

However,

$$D_{4\times3} \times C_{3\times4} = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 2 & 2 \\ 0 & 1 & 3 \\ 1 & 2 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 3 & 5 & 2 \end{bmatrix} = \begin{bmatrix} 10 & 19 & 28 & 17 \\ 13 & 18 & 23 & 18 \\ 7 & 12 & 17 & 7 \\ 14 & 23 & 32 & 16 \end{bmatrix}.$$

Division is not defined in matrix algebra. For example, Even with two square matrices P_{nn} and Q_{nn}, $P \div Q$ is NOT defined! The concept of the inverse of a square matrix is analogous to the concept of the reciprocal of a number. The detail of the inverse matrix will be discussed in the subsequent sections.

13.1.3 Unary Operations

Addition, subtraction, and multiplication involve two matrices. Next look at some unary operations in matrix algebra.

Transpose. The matrix that is resulting from a given matrix B after changing or reversing its rows to columns and columns to rows is called the transpose of a matrix B. For example,

$$D_{3\times4} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 3 & 5 & 2 \end{bmatrix} \text{ and } D^T_{4\times3} = \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \\ 4 & 1 & 2 \end{bmatrix}$$

Note that **transpose** is defined for any matrices (square or non-square) matrices.

Trace of A Matrix: The trace is the sum of all diagonal elements of a **square matrix**.

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1.2} & \cdots & a_{n-1,n} \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \text{ then } \text{Trace}(A_{n\times n}) = \sum_{i=1}^{n} a_{ii}$$

13.2 Determinant of a Square Matrix

The determinant of a matrix is the scalar value or number calculated using a square matrix. The calculation of a general square matrix is not straightforward, an iterative algorithm is needed. We will not introduce these algorithms in this class. But for triangular matrices, we can use the mathematical induction method to show the following theorem

Theorem: Let T_n be an upper triangular matrix of order n.

$$T_n = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Let $\det(T_n)$ be the determinant of T_n. Then $\det(T_n)$ is equal to the product of all the diagonal elements of T_n. That is

$$\det(T_n) = \sum_{i=1}^{n} a_{ii}.$$

Sketch of Proof: We use mathematical induction to prove this theorem.

1. The determinant is a_{11}, which is clearly also the diagonal element. That is, $\det(T_1) = a_{11}$. This establishes the induction basis.
2. Assume that $\det(T_k) = \sum_{i=1}^{k} a_{ii}$ for

$$T_k = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ 0 & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{kk} \end{bmatrix}.$$

We now consider

$$T_{k+1} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1,k+1} \\ 0 & a_{22} & \cdots & a_{2k} & a_{2,k+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{kk} & a_{k,k+1} \\ 0 & 0 & \cdots & 0 & a_{k+1,k+1} \end{bmatrix}.$$

Expanding above matrix by the $(k+1)$-th row, we have

$$T_{k+1} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} & a_{1,k+1} \\ 0 & a_{22} & \cdots & a_{2k} & a_{2,k+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nk} & a_{k,k+1} \\ 0 & 0 & \cdots & 0 & a_{k+1,k+1} \end{bmatrix} = a_{k+1,k+1} \times \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ 0 & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{kk} \end{bmatrix}.$$

Therefore,

$$\det(T_{k+1}) = \det(a_{k+1,k+1}T_k) = a_{k+1,k+1}\det(T_k) = a_{k+1,k+1}\sum_{i=1}^{k} a_{ii} = \sum_{i=1}^{k+1} a_{ii}.$$

This completes the proof.

We have introduced the Gaussian elimination algorithm that can be used for the triangulation of a square matrix. Therefore, we can use the R function we developed earlier to find the determinant of square matrices.

```
DET = function(A){
  # Input A: Must be a square matrix
  A0 = A
  n = dim(A0)[1]                  # number of rows
  for(i in 1:(n-1)){              # iterator fo pivot element A0[i,i]
                                  # i = 1, 2, ..., n-1
      #~~~~~~~~~~~~~~~~~~~~   Make the function robust  ~~~~~~~~~~~~~~~~~~~~~~~~#
      if(A0[i,i] == 0){
          non.0 = which(A0[,i] != 0)
              #cat("\n\n i =",i, ", non.0 =", non.0,". (i+1):n =", (i+1):n, ".")
              id = intersect(non.0, (i+1):n)[1] # which() equiv to find() in MATLAB
              if(!is.na(id)){
              tempi = A0[i,]                    # put the i-th row in a place-holder
              A0[i,] = A0[id,]                  # row swapping
              A0[id,] = tempi
            }
```

```
      }
      #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
       if(A0[i,i] != 0){    # A0[i,i] is in the denominator of recursive equation
          for(j in (i+1):n){  # to make the augmented matrix in the row echelon form
             #cat("\n\n j =",j,".")
             A0[j,] = A0[j,] - (A0[j,i]/A0[i,i])*A0[i,]
           }
        }
      }
     DET = prod(diag(A0))    # prod() - vectorized multiplication, diag() extracts a[i,i]
     if(dim(A)[1] != dim(A)[2]) {
        cat("\n\nNeed a square matrix!")
      } else{
       list(Triangular.Metrix = round(A0,4), Determinant = round(DET,4))
      }
 }
```

Example 3. Let

$$A = \begin{bmatrix} 1 & 9 & 8 & 7 \\ 3 & 8 & 3 & 8 \\ 7 & 2 & 7 & 7 \\ 4 & 3 & 2 & 6 \end{bmatrix}.$$

The determinant of A can be calculated using R in the following

```
A = matrix(c(1,9,8,7,3,8,3,8,7,2,7,7,4,3,2,6), ncol = 4, byrow = TRUE)
DET(A)
```

```
## $Triangular.Metrix
##      [,1] [,2]     [,3]     [,4]
## [1,]    1    9   8.0000   7.0000
## [2,]    0  -19 -21.0000 -13.0000
## [3,]    0    0  18.4211  -0.2632
## [4,]    0    0   0.0000   0.6714
##
## $Determinant
## [1] -235
```

The determinant $det(A) = -235$.

Example 4: Let

$$C_{3\times 4} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

```
C=matrix(c(1,2,3,4,4,3,2,1,1,3,5,2), ncol = 4, byrow = TRUE)
DET(C)
```

```
##
##
## Need a square matrix!
```

13.3 Matrix Inversion

Inverse of a square matrix Let A be a square matrix of order $n \times n$ and let I_n be an identity matrix of same order. If there exists a square matrix B, such that $AB = BA = I$, then the matrix A is said to be invertible. The matrix B is called inverse of A, that is denoted by A^{-1}.

We can use the Gauss-Jordan algorithm to find the inverse of a square matrix if the inverse exists.

Example 5 Find the inverse of the following matrix

$$A = \begin{bmatrix} 1 & 3 & 3 \\ 1 & 4 & 3 \\ 2 & 7 & 7 \end{bmatrix}.$$

Solution: We perform the forward and backward eliminations in the following

1. Forward elimination

$$\begin{bmatrix} 1 & 3 & 3 \\ 1 & 4 & 3 \\ 2 & 7 & 7 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{matrix} R2 - R1 \\ R3 - 2R1 \end{matrix} \Rightarrow \begin{bmatrix} 1 & 3 & 3 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \quad R3 - R2 \Rightarrow \begin{bmatrix} 1 & 3 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix}$$

2 Backward Elimination

$$\begin{bmatrix} 1 & 3 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad R1 - 3R3 \Rightarrow \begin{bmatrix} 1 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 4 & 3 & -3 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \quad R1 - 3R2 \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 7 & 0 & -3 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix}$$

Therefore,

$$A^{-1} = \begin{bmatrix} 7 & 0 & -3 \\ -1 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix}.$$

The algorithm is based on the extended augmented matrix. The following is an outline of the algorithm.

Let the following square matrix be invertible.

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}$$

The extended augmented matrix is

$$AM_{n,2n} = \left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{array}\right]_{n\times 2n}.$$

We can use Gauss-Jordan row operations to turn the above extended augmented matrix into the following form

$$IM_{n,2n} = \left[\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & w_{11} & w_{12} & \cdots & w_{1n} \\ 0 & 1 & \cdots & 0 & w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 1 & w_{n1} & w_{n2} & \cdots & w_{nn} \end{array}\right]_{n\times 2n}.$$

Then the inverse of A is given by

$$A^{-1} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & 0 \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix}_{n\times 2n}.$$

We can modify the `Gauss.Jordan()` in the previous note to get the following function to calculate the inverse matrix.

```
M.inv = function(A){
  # Input A: Must be a square matrix
  n = dim(A)[1]                  # number of rows
  A0 = cbind(A, diag(rep(1,n)))
  ##
  for(i in 1:(n-1)){   # iterator fo pivot element A0[i,i], i < n.
      #~~~~~~~~~~~~~~~~~~~~   Make the function robust  ~~~~~~~~~~~~~~~~~~~~~~~~~#
      if(A0[i,i] == 0){
          non.0 = which(A0[,i] != 0)
              #cat("\n\n i =",i, ", non.0 =", non.0,". (i+1):n =", (i+1):n, ".")
              id = intersect(non.0, (i+1):n)[1] # which() equiv to find() in MATLAB
              if(!is.na(id)){
              tempi = A0[i,]                    # put the i-th row in a place-holder
              A0[i,] = A0[id,]                  # row swapping
              A0[id,] = tempi
```

```
            }
        }
      #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
       if(A0[i,i] != 0){    # A0[i,i] is in the denominator of recursive equation
          for(j in (i+1):n){  # to make the augmented matrix in the row echelon form
             #cat("\n\n j =",j,".")
             A0[j,] = A0[j,] - (A0[j,i]/A0[i,i])*A0[i,]
           }
        }
    }
    A1=A0
    n = length(A1[,1])
    id.0 = rep(1,n)                      # indicator of zero A[i,i]
    for (i in 1:n){
          if(A1[i,i] == 0) id.0[i] = 0
        }
    if(sum(id.0) == n){                         # A[i,i] != 0
             for (i in 1:n){
                    A1[i,] = A1[i,] /A1[i,i]   # Divide each row by A1[i,i]
                   }                           # so that new A1[i,i] = 1.
             k = n
            A2 = A1[k:1, c(k:1,(n+1):(2*n))]     # re-arrange rows and columns
           #______________________________________________________________##
             for(i in 1:(n-1)){
                  if(A2[i,i] != 0){    # A0[i,i] is in the denominator of recursive eq
                  for(j in (i+1):n){  # to make the augmented matrix in the row echelo
                       A2[j,] = A2[j,] - (A2[j,i]/A2[i,i])*A2[i,]
                       }
                  }
               }
           #--------------------------------------------------------------##
             A4 = A2[k:1, c(k:1,(n+1):(2*n))]     # backward re-arrangement
             round(A4[, (n+1):(2*n)],7)
            } else if(dim(A)[1] != dim(A)[2]){
               cat("\n\nA square matrix is needed!")
            }else {# if exists A[i,i] =0
             cat("The matrix is singular")
           }
}
```

Example 6 (continuation of example 3) Let

$$A = \begin{bmatrix} 1 & 9 & 8 & 7 \\ 3 & 8 & 3 & 8 \\ 7 & 2 & 7 & 7 \\ 4 & 3 & 2 & 6 \end{bmatrix}.$$

The determinant of A can be calculated using R in the following

```
A = matrix(c(1,9,8,7,3,8,3,8,7,2,7,7,4,3,2,6), ncol = 4, byrow = TRUE)
M.inv(A)
```

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] -0.4893617  0.8723404  0.5319149 -1.2127660
## [2,] -0.2978723  0.7744681  0.3063830 -1.0425532
## [3,]  0.1489362 -0.1872340  0.0468085  0.0212766
## [4,]  0.4255319 -0.9063830 -0.5234043  1.4893617
```

```
solve(A)   # calling internal function solve() for comparison!
```

```
##            [,1]       [,2]        [,3]       [,4]
## [1,] -0.4893617  0.8723404  0.53191489 -1.2127660
## [2,] -0.2978723  0.7744681  0.30638298 -1.0425532
## [3,]  0.1489362 -0.1872340  0.04680851  0.0212766
## [4,]  0.4255319 -0.9063830 -0.52340426  1.4893617
```

Example 7 (continuation of example 4)

```
C=matrix(c(1,2,3,4,4,3,2,1,1,3,5,2), ncol = 4, byrow = TRUE)
M.inv(C)
```

```
##
##
## A square matrix is needed!
```

Example 8 (continuation of example 5) Find the inverse of the following matrix

$$A = \begin{bmatrix} 1 & 3 & 3 \\ 1 & 4 & 3 \\ 2 & 7 & 7 \end{bmatrix}.$$

```
A=matrix(c(1,3,3,1,4,3,2,7,7), ncol = 3, byrow = TRUE)
M.inv(A)
```

```
##      [,1] [,2] [,3]
## [1,]    7    0   -3
## [2,]   -1    1    0
## [3,]   -1   -1    1
```

Chapter 14

Matrix Factorization

In linear algebra, a matrix factorization is a factorization of a matrix into a product of matrices. There are differences in factorizing a matrix for different purposes. Among various ways of factorization, **lower–upper (LU)** factorization is the basic and is useful in solving linear systems. Although LU factorization works for any $m \times n$ matrix, it is common to factorize square matrix. In this note, we only discuss LU for square matrices.

Let A be the $n \times n$ matrix

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

A is said to have a LU-decomposition if there exists matrices L and U with the following properties:

1. L is a $n \times n$ lower triangular matrix with all diagonal entries being 1.

$$L_{n\times n} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}$$

2. U is a $n \times n$ matrix in some echelon form.

$$U_{n\times n} = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

3. $A = LU$.

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} = L_{n\times n} \times U_{n\times n}$$

14.1 Elementary and Permutation Matrices

Any **elementary matrix**, which we often denote by E, is obtained from applying **one** row operation to the **identity matrix** of the same size.

Examples Consider the resulting matrices after row operations to the identity matrix.

1. Adding one row to the other row: $R_1 + R_2 \to R_2$

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Multiplying a scalar to a row: $3 \times R2 \to R_2$

$$E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. Multiplying a scalar to a Row and then adding it to the other row: $5 \times R_1 + R3 \to R_3$

$$E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 0 & 1 \end{bmatrix}$$

4. Switching two rows: $R_1 \leftrightarrow R_2$:

$$E_3 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The elementary matrix obtained from switching two rows is commonly called **permutation matrix**.

To perform any of the first three row-operations on a matrix A, it suffices to take the product (left-multiplication) $E \times A$, where E is the elementary matrix obtained by using the desired row operation on the identity matrix.

Note that row operations (row switching) were used in matrix inversion and triangulation.

Example 1: Consider the triangulation problem:

$$A = \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \end{bmatrix}$$

Solution: The following three steps will complete the triangulation.

(1). $-2 \times R_1 + R_2 \to R_2$:

$$A = \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \end{bmatrix} \xrightarrow{-2\times R_1+R_2\to R_2} A_1 = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 3 & 2 & 5 \end{bmatrix}, \; E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that

$$E_1 \times A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 3 & 2 & 5 \end{bmatrix}$$

(2). $-3 \times R_1 + R_3 \to R_3$:

$$A_1 = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 3 & 2 & 5 \end{bmatrix} \xrightarrow{-3\times R_1+R_3\to R_3} A_2 = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & -10 & 2 \end{bmatrix}, \; E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix}$$

Note also that

$$E_2 \times E_1 \times A = E_2 \times A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 3 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & -10 & 2 \end{bmatrix} = A_2$$

(3). $-2 \times R_1 + R_3 \to R_3$:

$$A_2 = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & -10 & 2 \end{bmatrix} \xrightarrow{-3\times R_1+R_3\to R_3} A_3 = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \; E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix}$$

We note similarly that

$$E_3 \times E_2 \times E_1 \times A = E_3 \times A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & -10 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & 0 & 0 \end{bmatrix} = A_3$$

Next, we look at the product of the three (lower angular) elementary matrices.

$$E = E_3 \times E_2 \times E_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}$$

Several important observations help develop algorithms to perform the *LU* factorization.

1. All elementary matrices that are based on the forward row operations (NOT the row-swapping) are **lower triangular matrices** (and are obviously non-singular).
2. The product of all **lower triangular matrices** will also be a **lower triangular matrix**. This can easily be proved by the definition of matrix multiplication.
3. The product of elementary matrices is **non-singular** (i.e., the determinant is NOT equal to zero).
4. The inverse of a **lower triangular matrix** (if exists) is also a lower triangular. This is also obvious from the process of finding the inverse of a function

In the above example, A_3 is an upper triangular matrix (special echelon form). We now rewrite the row operations in the following

$$E \times A = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & 0 & 0 \end{bmatrix} = A_3$$

This implies that

$$A = \begin{bmatrix} 1 & 4 & 1 \\ 2 & 3 & 3 \\ 3 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} 1 & 4 & 1 \\ 0 & -5 & 1 \\ 0 & 0 & 0 \end{bmatrix} = E^{-1} \times A_3$$

Let $L = E^{-1}$ (lower triangular matrix) and $U = A_3$. Then $A = LU$! For a 3×3 matrix, we need to perform $3(3-1)/2 = 3$ row operations, which means, we need to take the product 3 elementary matrices to obtain the upper triangular matrix A_3. The calculation of E is tedious and time-consuming. However, we can use the R function `Gauss.Elimination()` on an arguments matrix to get E.

$$\left[\begin{array}{ccc:ccc} 1 & 4 & 1 & 1 & 0 & 0 \\ 2 & 3 & 3 & 0 & 1 & 0 \\ 3 & 2 & 5 & 0 & 0 & 1 \end{array}\right] \xrightarrow{\text{Gaussian Row Elimination}} \left[\begin{array}{ccc:ccc} 1 & 4 & 1 & 1 & 0 & 0 \\ 0 & -5 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \end{array}\right]$$

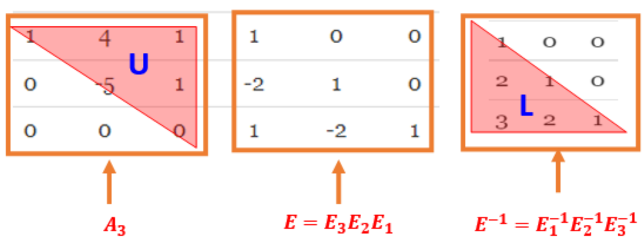
Using the code we developed in the earlier note to find the $E = E_3E_2E_1$ (order is important!)

```
Gauss.Elimination = function(A){
  # Input A: an n-by-m matrix
  n = dim(A)[1]          # number of rows
  A0 = cbind(A, diag(rep(1,n)))   # Extended augmented matrix
  #P = diag(rep(1,n))     # identity matrix for permutation
  for(i in 1:(n-1)){     # iterator fo pivot element A0[i,i]
                         # i = 1, 2, ..., n-1
      #~~~~~~~~~~~~~~~~~~~~   Make the function robust  ~~~~~~~~~~~~~~~~~~~~~~~~~#
      if(A0[i,i] == 0){
          non.0 = which(A0[,i] != 0)      # which() equiv to find() in MATLAB
              #cat("\n\n i =",i, ", non.0 =", non.0,". (i+1):n =", (i+1):n, ".")
              id = intersect(non.0, (i+1):n)[1] # vector of common row ID in the
                                                # lower triangular part of the matrix.
              if(!is.na(id)){     # if the common vector is not empty, do row-swapping!
              tempi = A0[i,]                    # put the i-th row in a place-holder
              A0[i,] = A0[id,]                  # row swapping
              A0[id,] = tempi
              ###                               # updating permutation matrix
              #tempP = P[i,]
              #P[i,] = P[id,]
              #P[id,] = tempP
             }
       }
```

```
      #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
       if(AO[i,i] != 0){    # AO[i,i] is in the denominator of recursive equation
          for(j in (i+1):n){  # to make the augmented matrix in the row echelon form
             #cat("\n\n j =",j,".")
             AO[j,] = AO[j,] - (AO[j,i]/AO[i,i])*AO[i,]
           }
        }
      }
   AO
 }
```

```
A=matrix(c(1,4,1,2,3,3,3,2,5), nrow=3, byrow = TRUE)
Gauss.Elimination(A=A)
```

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



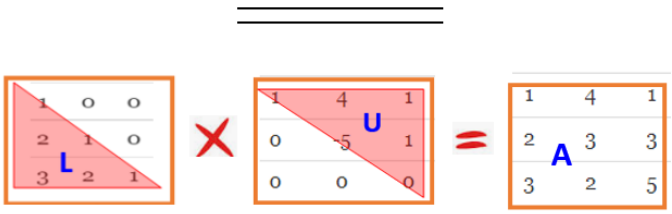

```
A=matrix(c(1,4,1,2,3,3,3,2,5), nrow=3, byrow = TRUE)
B=cbind(A, diag(rep(1,3)))
C = Gauss.Elimination(B)
U = C[,1:3]
L = solve(C[,4:6])
LU = L%*%U
```

```
pander(LU)
```

1	4	1
2	3	3
3	2	5

```
pander(A)
```

1	4	1
2	3	3
3	2	5



14.2 LU Factorization Algorithm

It took three elementary row operations to obtain the lower triangular matrix in the above special $3times3$ matrix. In general, for an $n \times n$ matrix, we need $n(n-1)/2$ forward row operations (excluding row-swapping) to get the upper triangular matrix.

14.2.1 Row Operations with Row Swapping

As explained in the illustrative example, we can apply the Gaussian elimination method to the extended augmented matrix to obtain the product of the elementary matrices induced from the corresponding row operations if no row-swapping is involved. To be more specific, let

$$A_{n\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix},$$

we define an extended augmented matrix in the following

$$A^0_{n\times n} = \left[\begin{array}{cccc|ccc} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & \cdots 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & 0 & 0 & \cdots 1 \end{array}\right],$$

After finishing the Gaussian elimination, we have upper triangular and lower triangular matrices that are similar to those in the example.

14.2.2 Row Operations with Row-swapping

When the row operation involves row-swapping (permutations), say, $P_1, P_2, \cdots, P_k$, we need to calculate $P = P_k \times P_{k-1} \times \cdots \times P_2 \times P_1$. Then we can apply the Gaussian elimination to PA to factorize it into the product of an upper triangular matrix and a lower triangular matrix.

Note that, the linear system $AX = B$ is equivalent to linear system $PAX = B$ since P involves only row-swapping.

Properties of Permutation Matrix. Let P be a permutation matrix, then

1. The inverse of P exists.
2. The inverse of P is equal to the transpose of P. $P^{-1} = P^T$.

14.2.3 LU ALgorithm

```
INPUT: The input n-by-m matrix A

OUTPUT:  L, U, P (if P = I, no permutation involved)

STEP 1: Initialization:
        n = #rows,
        A0 = [A,I]       (extended augmented matrix)
        P = I            (n-by-n)

STEP 2: FOR i = 1 TO n DO:     (running through pivot elements)
        STEP 3: IF A[i,i] = 0 for some i DO:
                row-swapping and updating P
                ENDIF
        STEP 4: FOR j = 1 TO n DO:
                A0[j,] = A0[j,] - (A0[j,i]/A0[i,i])*A0[i,]
                ENDFOR
        ENDFOR
STEP 5: Extract P, U = A0[,1:n], L = inverse of A0[, (n+1):2n]
STEP 6: RETURN A, P, PA, L, U.
```

R function for LU factorization

```
LU = function(A){
  # Input A: an n-by-m matrix
  n = dim(A)[1]           # number of rows
  A0 = cbind(A,diag(rep(1,n)))                # Extended augmented matrix
```

```
  P = diag(rep(1,n))      # identity matrix for permutation
  E = diag(rep(1,n))      # for the elementary matrix involving no row exchange.
  for(i in 1:(n-1)){      # iterator fo pivot element A0[i,i]
                          # i = 1, 2, ..., n-1
         #~~~~~~~~~~~~~~~~~~~~   Make the function robust  ~~~~~~~~~~~~~~~~~~~~~~~~#
         if(A0[i,i] == 0){
            non.0 = which(A0[,i] != 0)      # which() equiv to find() in MATLAB
              #cat("\n\n i =",i, ", non.0 =", non.0,". (i+1):n =", (i+1):n, ".")
              id = intersect(non.0, (i+1):n)[1] # vector of common row ID in the
                                                # lower triangular part of the matrix.
              if(!is.na(id)){      # if the common vector is not empty, do row-swapping!
              tempi = A0[i,]                    # put the i-th row in a place-holder
              A0[i,] = A0[id,]                  # row swapping
              A0[id,] = tempi
              ###                               # updating permutation matrix
              tempP = P[i,]
              P[i,] = P[id,]
              P[id,] = tempP
            }
           }
         #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
         if(A0[i,i] != 0){      # A0[i,i] is in the denominator of recursive equation
         for(j in (i+1):n){     # to make the augmented matrix in the row echelon form
              #cat("\n\n j =",j,".")
              A0[j,] = A0[j,] - (A0[j,i]/A0[i,i])*A0[i,]
              E[j,] = E[j,] - (A0[j,i]/A0[i,i])*E[i,]
            }
         }
        #print(E)
     }
   PA = P%*%A
   U = A0[,1:n]
   L = solve(A0[,(n+1):(2*n)])
   A = A
   list(A = A, P = P, PA = PA, L = L, U = U)
 }
```

Example 2

```
A=matrix(c(0,0,-1, 1, 1, 1, -1, 2, -1, -1, 2, 0, 1, 2, 0, 2), ncol  = 4, byrow = TRUE)
LU(A)
```

```
## $A
##      [,1] [,2] [,3] [,4]
```

```
## [1,]    0    0   -1    1
## [2,]    1    1   -1    2
## [3,]   -1   -1    2    0
## [4,]    1    2    0    2
##
## $P
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    0    0
## [2,]    0    0    0    1
## [3,]    0    0    1    0
## [4,]    1    0    0    0
##
## $PA
##      [,1] [,2] [,3] [,4]
## [1,]    1    1   -1    2
## [2,]    1    2    0    2
## [3,]   -1   -1    2    0
## [4,]    0    0   -1    1
##
## $L
##      [,1] [,2] [,3] [,4]
## [1,]    0    0   -1    1
## [2,]    1    0    0    0
## [3,]   -1    0    1    0
## [4,]    1    1    0    0
##
## $U
##      [,1] [,2] [,3] [,4]
## [1,]    1    1   -1    2
## [2,]    0    1    1    0
## [3,]    0    0    1    2
## [4,]    0    0    0    3
```

14.3 Benefit of LU Factorization

Suppose that A has been factored into the triangular form $A = LU$, where L is lower triangular and U is upper triangular. Then we can solve for x more easily by using a two-step process.

- Let $y = Ux$ and solve the lower triangular system $Ly = b$ for y. Since L is triangular, determining y from this equation requires only $O(n^2)$ operations.
- Once y is known, the upper triangular system $Ux = y$ requires only an additional $O(n^2)$ operations to determine the solution x.

Solving a linear system $Ax = b$ in factored form means that the number of operations needed to solve the system $Ax = b$ is reduced from $O(n^3/3)$ to $O(2n^2)$.

Chapter 15

Least Square Approximation

We have introduced the direct methods in matrix algebra and several interpolation methods to approximate a univariate function with either an explicitly given analytic expression or with an unknown expression. In this note, we use matrix algebra to approximate an unknown linear and some nonlinear functions with multiple variables.

15.1 Concepts of Least Square Approximation

To get a better idea of the least square approximation, we start with a linear approximation with only a single variable.

- **The Linear Function to Approximate**: $y = f(x) = \beta_0 + \beta_1 x$. β_0 and β_1 are unknown and will be estimated from the given information.
- **Given Information**: A set of data points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$.

The idea is to find $\hat{\beta}_0$ and $\hat{\beta}_1$ based on the given data points that **MINIMIZES** the mean square error (MSE) between the predicted second coordinate $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ and the observed y_i. To be more specific,

1. Assume that the hypothetical linear function has analytic expression $y = \beta_0 + \beta_1 x$.
2. Define the predicted to be $\hat{y}_i = \beta_0 + \beta_1 x_i$.
3. Define error $d_i = \hat{y}_i - y_i$.

4. Define MSE as

$$\text{MSE}(\beta_0, \beta_1) = \frac{\sum_{i=1}^n d_i^2}{n-2}$$

The LSE estimates the unknowns β_0 and β_1 by minimizing the MSE!

https://github.com/pengdsci/MAT325/raw/main/w14/img/w14-best-fit-line.gif

15.2 Approximating One-variable Linear Function

To approximate β_0 and β_1, we minimize MSE defined in the previous section in the following. Denote $Q(\beta_0, \beta_1) = \sum_{i=1}^n d_i^2 = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2$. The approximated β_0 and β_1, denoted by $\hat{\beta}_0$ and $\hat{\beta}_1$, is the solution to the following optimization problem

$$\arg\min_{\beta_0, \beta_1} Q(\beta_0, \beta_1)$$

To minimize $Q(\beta_0, \beta_1)$, we find the first-order derivatives and set them to zero.

$$\frac{\partial Q(\beta_0, \beta_1)}{\partial \beta_0} = -\sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i) = 0, \frac{\partial Q(\beta_0, \beta_1)}{\partial \beta_1} = -\sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i) x_i = 0.$$

The above system is equivalent to

$$\begin{array}{rcrcl} n\beta_0 & + & (\sum_{i=1}^n x_i)\beta_1 & = & \sum_{i=1}^n y_i, \\ (\sum_{i=1}^n x_i)\beta_0 & + & (\sum_{i=1}^n x_i^2)\beta_1 & = & \sum_{i=1}^n x_i y_i. \end{array}$$

The matrix representation of the above system is given by

$$\begin{bmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_i y_i \end{bmatrix},$$

which can be further re-expressed as

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ x_1 & x_2 & \cdots & x_{n-1} & x_n \end{bmatrix} \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{n-1} \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ x_1 & x_2 & \cdots & x_{n-1} & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}.$$

Denote

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{n-1} \\ 1 & x_n \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}, \quad \text{and} \quad = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}.$$

Then

$$\mathbf{X}^T\mathbf{X} \ = \mathbf{X}^T\mathbf{Y}.$$

Clearly, $\mathbf{X}^T\mathbf{X}$ is a square matrix. If it is invertible, we solve for from the above system

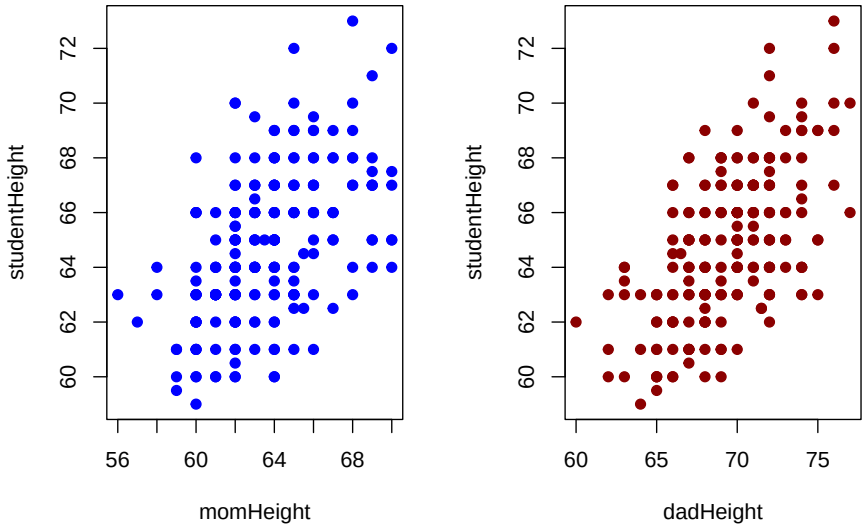
$$= [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{Y}.$$

The right-hand side of the above equation is only dependent on the given data points. Therefore,$\hat{} = [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{Y}$ is the solution to the minimization problem.

Therefore, $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$ is the least square approximation of $y = \beta_0 + \beta_1 x$.

Example 1: The data are from $n = 214$ females in statistics classes at the University of California at Davis (https://raw.githubusercontent.com/pengdsci/MAT325/main/w14/LS-ApproxHeights.txt). The variables are y = student's self-reported height, x_1 = student's guess at her mother's height, and x_2 = student's guess at her father's height. All heights are in inches. The scatter plots below are of each student's height versus the mother's height and the student's height against the father's height.

```
height = read.table("https://raw.githubusercontent.com/pengdsci/MAT325/main/w14/LS-ApproxHeights.
studentHeight = height$Height     # extract students' heights from the data set and store them in
momHeight = height$momheight      # vector of mothers' heights
dadHeight = height$dadheight      # vector of fathers' heights
## making scatter plots
par(mfrow = c(1,2))               # setting up the layout of the graphical page
plot(momHeight, studentHeight, pch = 19, col = "blue", main = "")
plot(dadHeight, studentHeight, pch = 19, col = "darkred", main = "")
```



Solution: We want to approximate the linear relationship between students' heights and their mothers' and fathers' heights respectively.

Part I: Assume that the linear function between students' heights and their mothers' heights is of the following form

$$y = \beta_0 + \beta_1 x_1$$

To approximate β_0 and β_1, we define $\mathbf{X}$ and $\mathbf{Y}$ and then find the solution of the LS approximation in the following R code.

```
n = length(studentHeight)
X = cbind(rep(1,n), momHeight)
Y = cbind(studentHeight)
beta = round(solve(t(X)%*%X)%*%(t(X)%*%Y),4)
kable(beta)
```

	studentHeight
	34.0223
momHeight	0.4834

Therefore, the linear function is approximated by $y = 34.0223 + 0.4834x_1$.

15.3 Approximating Multiple-variable Linear Function

15.3.1 An Extension of the Height Example

In the above example, we assume that a student's height is only dependent on her mother's height. However, the height of a person is dependent on both parents. We further assume that the relationship has form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$. Then we need to approximate unknowns $(\beta_0, \beta_1, \beta_2$ using the same logic that was used for approximating the linear function with only a single variable. To be more specific, we need only to modify the matrix of $\mathbf{X}$ by adding one additional column. The three matrices for

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{21} \\ 1 & x_{12} & x_{22} \\ \vdots & \vdots & \\ 1 & x_{1,n-1} & x_{2,n-1} \\ 1 & x_{1n} & x_{2n} \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}, \quad \text{and} \quad = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}.$$

The LS approximation of $(\beta_0, \beta_1, \beta_2)$ is given by

$$= [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{Y}.$$

Example 2 [Continuation of *Example 1*]. We now approximate linear function $y = \beta_0 + \beta_1 x_1 + \beta x_2$ using the data given in example 1. The following R code approximates the coefficients of the linear function.

```
height = read.table("https://raw.githubusercontent.com/pengdsci/MAT325/main/w14/LS-ApproxHeights.
studentHeight = height$Height     # extract students' heights from the data set and store them in
momHeight = height$momheight      # vector of mothers' heights
dadHeight = height$dadheight      # vector of fathers' heights
n = length(studentHeight)
X = cbind(rep(1,n), momHeight, dadHeight)
Y = cbind(studentHeight)
beta = round(solve(t(X)%*%X)%*%(t(X)%*%Y),4)
kable(beta)
```

	studentHeight
	18.5473
momHeight	0.3035
dadHeight	0.3879

Therefore, the approximated linear function is $y = 18.55 + 0.3035x_1 + 0.3879x_2$.

15.3.2 Approximating General Multi-variables Linear Functions

To approximate linear function $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots = \beta_k x_k$ based on a set of given data points $(y_i, x_{1i}, x_{2i}, \cdots, x_{ki})$ for $i = 1, 2, \cdots, n$, we extend the $\mathbf{X}$ matrix similarly by adding more columns to it based on the given data points. To be more specific, the three matrices used in the LS approximation are explicitly given by

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{21} & \cdots & x_{k1} \\ 1 & x_{12} & x_{22} & \cdots & x_{k2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{1,n-1} & x_{2,n-1} & \cdots & x_{k,n-1} \\ 1 & x_{1n} & x_{2n} & \cdots & x_{kn} \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}, \text{ and } = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{k-1} \\ \beta_k \end{bmatrix}.$$

The solution to the LS approximation has the same general form given below

$$= [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{Y}.$$

Example 3: For a sample of $n = 20$ individuals, we have measurements of $y =$ body fat, $x_1 =$ triceps skinfold thickness, $x_2 =$ thigh circumference, and $x_3 =$ midarm circumference. The Data can be found at https://raw.githubusercontent.com/pengdsci/MAT325/main/w14/LS-Approximation-Bodyfat.txt.

Solution We use the following code to find the approximated coefficients in the linear function $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$.

```
bodyFat = read.table("https://raw.githubusercontent.com/pengdsci/MAT325/main/w14/LS-Ap
n = dim(bodyFat)[1]    # number of data points
X = as.matrix(cbind(intercept=rep(1,n), bodyFat[, 1:3]))
Y = as.matrix(bodyFat[,4])
beta = round(solve(t(X)%*%X)%*%(t(X)%*%Y),4)
kable(beta)
```

intercept	117.0847
Triceps	4.3341
Thigh	-2.8568
Midarm	-2.1861

Therefore, the approximated linear function is $y = 117.1 + 4.334x_1 - 2.857x_2 - 2.186x_3$, or equivalently, body fat $=$ 117.1 + 4.334Tricps $-$ 2.857Thigh $-$ 2.186Midarm.

15.4 Approximating One-variable Polynomial Function

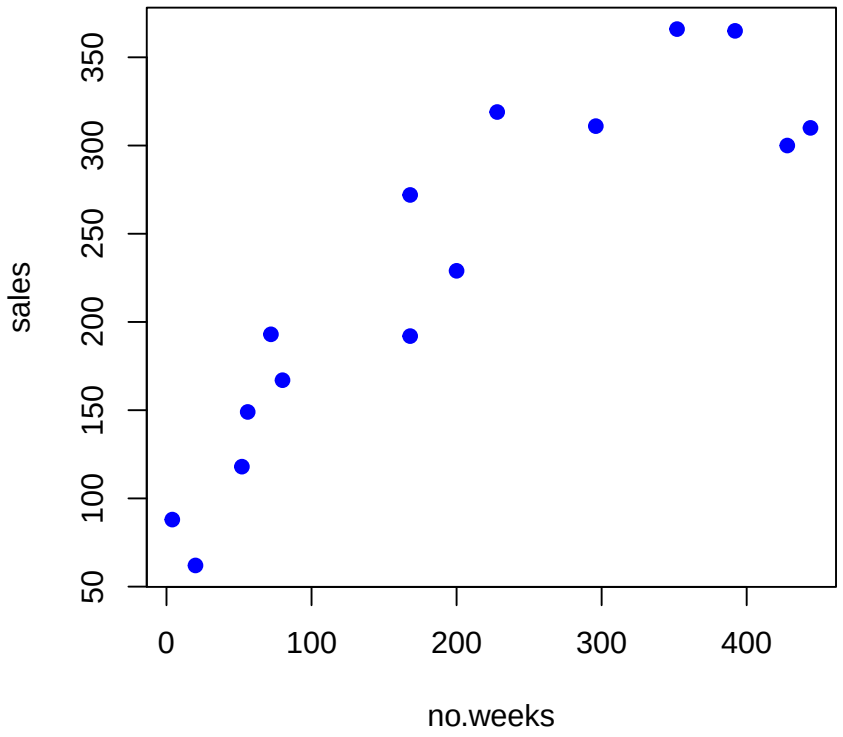
We have introduced several interpolation-based approximations to a function. In this section, we use the LS approach to approximate a single variable polynomial. $y = a_0 + a_1x + a_2x^2 + \cdots + a_kx^k$.

15.4.1 Approximating Quadratic Function

As an illustrative example, let's approximate the trend line of the sales of a particular brand of vehicles. sales = f(time), the variable **time** represents the number of weeks.

	No of weeks	Sold cars
0	168	272
1	428	300
2	296	311
3	392	365
4	80	167
5	56	149
6	352	366
7	444	310
8	168	192
9	200	229
10	4	88
11	52	118
12	20	62
13	228	319
14	72	193

The scatter plot of the two variables is given below.



The above plot indicates a non-linear trend. A quadratic trend seems to be more appropriate to characterize the relationship.

Assuming the quadratic relation between the time and the sales, we have the unknown quadratic function sales $= f(\text{time}) = a_0 + a_1\text{time} + a_2\text{time}^2$. If we denote $y =$ sales, $x_1 =$ time, and $x_2 = \text{time}^2$, the unknown quadratic function can written as $y = a_0 + a_1x_1 + a_2x_2$. This means we can still use the multi-variable LS to approximate the unknown quadratic function. To be more specific, matrices $\mathbf{X}$ and $\mathbf{Y}$ for the LS approximation are given by

$$\mathbf{X} = \begin{bmatrix} 1 & 168 & 28224 \\ 1 & 428 & 183184 \\ 1 & 296 & 87616 \\ 1 & 392 & 153664 \\ 1 & 80 & 6400 \\ 1 & 56 & 3136 \\ 1 & 352 & 123904 \\ 1 & 444 & 197136 \\ 1 & 168 & 28224 \\ 1 & 200 & 40000 \\ 1 & 4 & 16 \\ 1 & 52 & 2704 \\ 1 & 20 & 400 \\ 1 & 228 & 51984 \\ 1 & 72 & 5184 \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} 272 \\ 300 \\ 311 \\ 365 \\ 167 \\ 149 \\ 366 \\ 310 \\ 192 \\ 229 \\ 88 \\ 118 \\ 62 \\ 319 \\ 193 \end{bmatrix}$$

The coefficients of the quadratic function are estimated by

$$= [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{Y}.$$

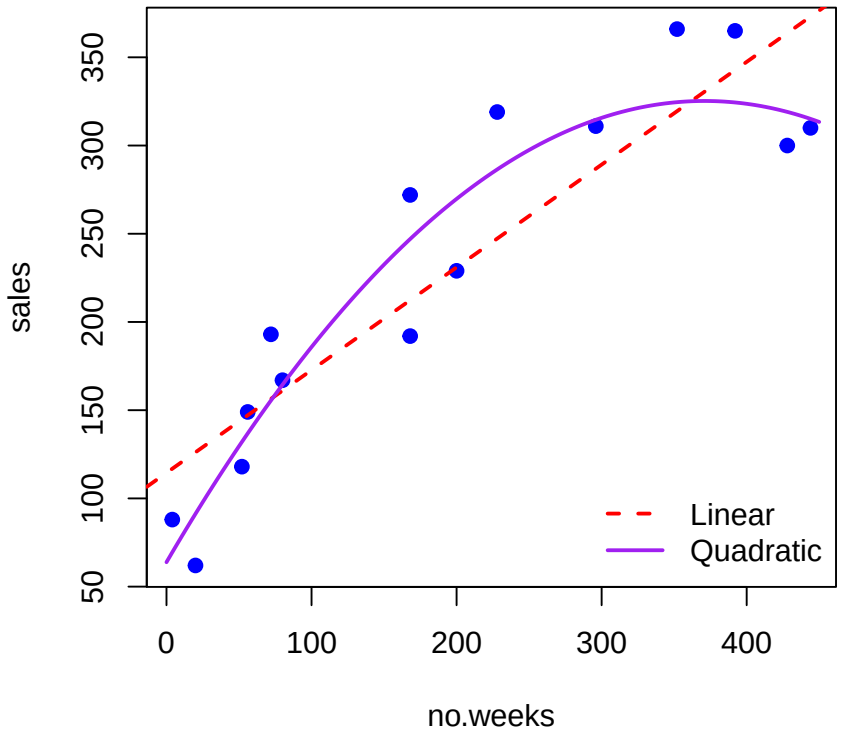
The following R code calculates the approximated coefficients.

```
Y = matrix(c(272, 300, 311, 365, 167, 149, 366, 310, 192, 229, 88, 118, 62, 319, 193), ncol=1)
no.weeks = c(168, 428, 296, 392, 80, 56, 352, 444, 168, 200, 4, 52, 20, 228, 72)
X = as.matrix(cbind(intercept = rep(1, length(Y)), weeks = no.weeks, weeks.sq = no.weeks^2))
beta = solve(t(X)%*%X)%*%(t(X)%*%Y)
kable(round(beta,4))
```

intercept	63.8510
weeks	1.4095
weeks.sq	-0.0019

Therefore, the unknown quadratic function is approximated by

$$\text{sales} = 63.8510 + 1.4095 \times \text{time} - 0.0019 \times \text{time}^2$$



15.4.2 Approximating General Polynomial Functions

In general, for a given set of data points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$, the unknown polynomial has form $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k$ can be approximated based on the following two matrices

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^k \\ 1 & x_2 & x_2^2 & \cdots & x_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^k \\ 1 & x_n & x_n^2 & \cdots & x_n^k \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}$$

The coefficients $= c(a_0, a_1, \cdots, a_k)^T$ is given by

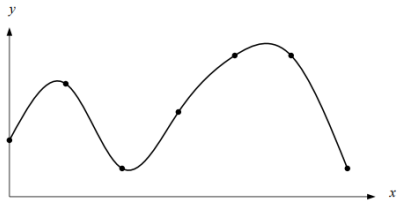
$$= (X^T X)^{-1} X^T Y.$$

Chapter 16

Concepts of Spline Interpolations

We will start with the concepts of various basic concepts of spline curves before introducing the cubic smoothing spline for estimating the functions with given points (knots).

Roughly speaking, splines are functions that are piece-wise polynomials. The coefficients of the polynomial differ from interval to interval, but the order of the polynomial is the same. An essential feature of splines is that function is continuous - i.e. has no breaks on the boundaries between two adjacent intervals. That is, they create smooth curves out of irregular data points.



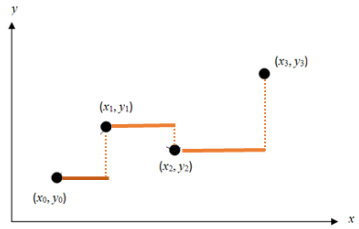
- Suppose that $n+1$ distinct points $x_0, x_1, \cdots, x_n$ have been specified and satisfy $x_0 < x_1 < \cdots < x_n$. *These points are called knots.*
- Suppose also that an integer $k \geq 0$ has been prescribed. A spline function of degree k having knots $x_0, x_1, \cdots, x_n$ is a function $S(\cdot)$ such that:
  - On each interval $[x_{i-1}, x_i]$, $S(\cdot)$ is a polynomial of degree $\leq k$.
  - $S(\cdot)$ has a continuous $(k-1)$-th derivative on $[x_0, x_n]$.

That is if $S(\cdot)$ is a piece-wise polynomial of degree at most 3 having continuous derivatives of all orders up to 2.

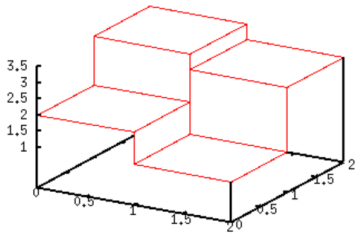
Example 1 Spline of degree 0 is piece-wise constant. A spline of degree 0 can be given explicitly in the form

$$S(x)=\begin{cases} S_0(x)=c_0, & x\in[x_0,x_1] \\ S_1(x)=c_1, & x\in[x_1,x_2] \\ \vdots & \vdots \\ S_{n-1}(x)=c_{n-1}, & x\in[x_{n-1},x_n] \end{cases}.$$

The intervals $[x_{i-1}, x_i]$ do not intersect each other and so no ambiguity arises in defining such a function at the knots. For example, in the following four-knot data, the zero-degree spline is graphically represented below.



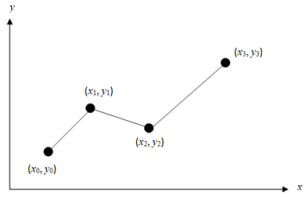
The three dimensional zero-degree spline can be similarly constructed.



We have also used this spline interpolation in Riemann sum in multivatiable integration in calculus. https://demonstrations.wolfram.com/ApproximatingADoubleIntegralWithCuboids/

16.1 Linear Splines

Linear spline interpolation is simply a line plot that connects all consecutive points $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1}), (x_n, y_n)\}$ $(x_0, x_1, \cdots, x_n)$. So if the above data is given in ascending order, the linear splines are given by $y_i = f(x_i)$.



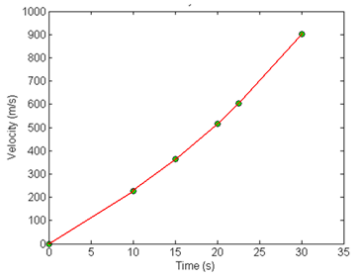
The function of the above curve (i.e., line plot) is given below.

$$f(x) = \begin{cases} f(x_0) + \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0), & x_0 \le x \le x_1 \\ f(x_1) + \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1), & x_1 \le x \le x_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ f(x_{n-1}) + \dfrac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_{n-1}), & x_{n-1} \le x \le x_n \end{cases}$$

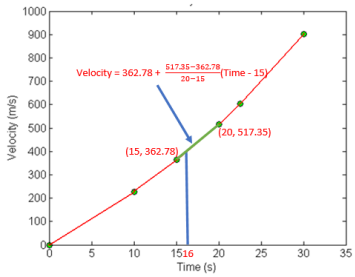
Example 2: The upward velocity of a rocket is given as a function of time in the table below. Using the linear spline to determine the value of the velocity at $t = 16$ seconds.

t (s)	$v(t)$ (m/s)
0	0
10	227.04
15	362.78
20	517.35
22.5	602.97
30	901.67

The linear spline is plotted in the following



We can use the data table to calculate the slope of each individual line segment in the above curve and express the spline function explicitly. The predicted velocity at $Time = 16$ can be found using the piece of the spline in the figure below.



That is,

$$\text{Velocity}_{pred} = 362.78 + \frac{517.35 - 362.78}{20 - 15}(16 - 15) = 393.694.$$

R/MATLAB Program to implement **Linear Spline**.

Assume n given points with coordinates $\{(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)\}$ $(x_0 < x_1 < \cdots < x_n)$. We use the vectorized function `which()` in R or `find()` in MATLAB to avoid using a loop in the algorithm.

```
INPUT: input nodes    (xi,yi) (i = 1, 2, ..., n)
       x.new          (a vector of x values for predicting y.new)

OUTPUT: y.new

STEP 1: Initializing:
        m = number of input x values
        y.new = NULL  (storing pred y)
STEP 2: FOR i = 1 TO m DO;
           IF x[k] <= x.new[i] < x[k+1] DO
               y.new[i] = ((y[k+1]-y[k])/(x[k+1] - x[k]))*(x.new-x[k]) + y[k]
           ENDIF
        ENDFOR
STEP 3: RETURN y.new
```

R Code

```
LSpline = function(x,          # x-coordinates of the input knots
                   y,          # y-coordinates of the input knots
                   x.new       # the new x values to be evaluated and returned
```

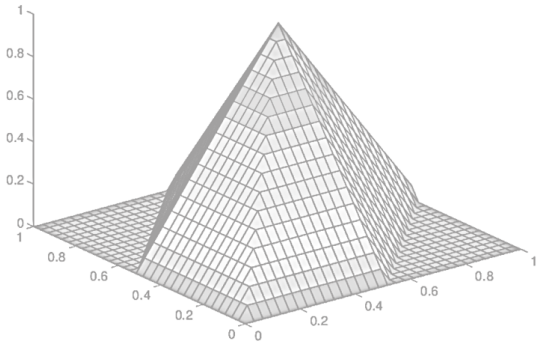
```
                    ){
    m = length(x.new)
    y.new = NULL
    for (i in 1:m){
       k = which(x >= x.new[i])[1]
       y.new[i] = ((y[k+1]-y[k])/(x[k+1] - x[k]))*(x.new[i]-x[k]) + y[k]
    }
    y.new
}
```

Example (reproduce the results of **Examples 2**) using the above function.

```
x = c(0, 10, 15, 20, 22.5, 30)
y = c(0, 227.04, 362.78, 517.35, 602.97, 901.67)
LSpline(x = x, y = y, x.new = c(10, 15, 16, 20))
```

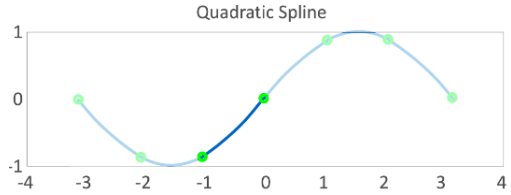
```
## [1] 227.040 362.780 380.358 517.350
```

As expected, linear spline idea can be used in the high dimensional space.



16.2 Quadratic Spline Interpolation

Unlike linear spline interpolation in which two consecutive knots are connected by a line segment, in quadratic spline interpolation, two consecutive knots are connected by a curve of quadratic function, and every adjacent quadratic curve is connected smoothly (i.e., the derivative of the resulting quadratic spline exists at all **inner knots**).



16.2.1 Formulation of Quadratic Spline

In these splines, a quadratic polynomial approximates the data between two consecutive data points. Given $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1}), (x_n, y_n), \}$, fit quadratic splines through the data. The splines are given by

$$f(x) = \begin{cases} a_1x^2 + b_1x + c_1, & x_0 \le x \le x_1 \\ a_2x^2 + b_2x + c_2, & x_1 \le x \le x_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_nx^2 + b_nx + c_n, & x_{n-1} \le x \le x_n \end{cases}$$

There are $3n$ coefficients (a_i, b_i, c_i) for $i = 1, 2, \cdots, n$ that must satisfy the following conditions so that the adjacent quadratic curves are connected smoothly.

- **Adjacent connection** implies that the quadratic curves pass through every knot. We have the following $2n$ equations.

$$a_1{x_0}^2 + b_1x_0 + c_1 = f(x_0) \quad \text{Knot 1}$$

$$\left.\begin{aligned} a_1{x_1}^2 + b_1x_1 + c_1 &= f(x_1) \\ a_2{x_1}^2 + b_2x_1 + c_2 &= f(x_1) \end{aligned}\right\} \quad \text{Knot 2}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$\left.\begin{aligned} a_{i-1}{x_{i-1}}^2 + b_{i-1}x_{i-1} + c_{i-1} &= f(x_{i-1}) \\ a_i{x_{i-1}}^2 + b_ix_{i-1} + c_i &= f(x_{i-1}) \end{aligned}\right\} \quad \text{Knot } i-1$$

$$\left.\begin{aligned} a_i{x_i}^2 + b_ix_i + c_i &= f(x_i) \\ a_{i+1}{x_i}^2 + b_{i+1}x_i + c_{i+1} &= f(x_i) \end{aligned}\right\} \quad \text{Knot } i$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$\left.\begin{aligned} a_{n-1}{x_{n-1}}^2 + b_{n-1}x_{n-1} + c_{n-1} &= f(x_{n-1}) \\ a_n{x_{n-1}}^2 + b_nx_{n-1} + c_n &= f(x_{n-1}) \end{aligned}\right\} \quad \text{Knot } n-1$$

$$a_n{x_n}^2 + b_nx_n + c_n = f(x_n) \quad \text{Knot n}$$

- **Smooth connection** requires the first-order derivatives of adjacent quadratic curves at each knot to be equal. This gives the following $n-1$ equations:

$$2a_k x_k + b_k = 2a_{k+1}x_k + b_{k+1}$$

for $k = 1, 2, \cdots, n-1$. that can be written as

$$(2x_k)a_k + b_k - (2x_k)a_{k+1} - b_{k+1} = 0$$

- **Default Assumption**. A common assumption we can take is to set $a_1 = 0$ or $a_1 = a_n$.

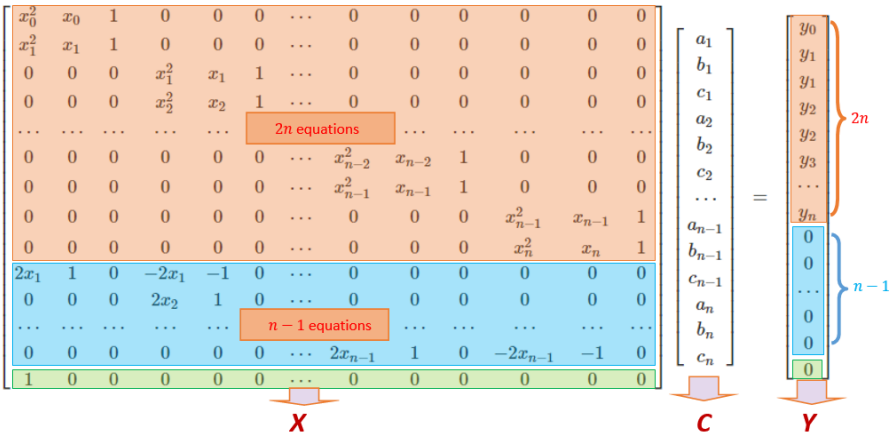
For given $n+1$ points ($n-1$ interior points), we can solve for the $3n$ coefficients for the n quadratic equations.

16.2.2 Matrix Representation of Quadratic Spline Problem

Since the knots will be given, the actual unknowns are the coefficients of the quadratic functions. We can write the spline problem in the following matrix form.

$$\begin{bmatrix}
x_0^2 & x_0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
x_1^2 & x_1 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & x_1^2 & x_1 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & x_2^2 & x_2 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & x_{n-2}^2 & x_{n-2} & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & x_{n-1}^2 & x_{n-1} & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & x_{n-1}^2 & x_{n-1} & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & x_n^2 & x_n & 1 \\
2x_1 & 1 & 0 & -2x_1 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 2x_2 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 2x_{n-1} & 1 & 0 & -2x_{n-1} & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ \cdots \\ a_{n-1} \\ b_{n-1} \\ c_{n-1} \\ a_n \\ b_n \\ c_n
\end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ y_1 \\ y_2 \\ y_2 \\ y_3 \\ \cdots \\ y_n \\ 0 \\ 0 \\ \cdots \\ 0 \\ 0 \\ 0
\end{bmatrix}$$

The annotated version of the above matrix equation is given below



16.2.3 Algorithm of Quadratic Spline

The above matrix representation of the quadratic spline problem is a linear system that has a unique solution. Using the algorithm we developed earlier to solve the linear system:

$$XC = Y$$

The solution is

$$C = X^{-1}Y$$

The following pseudo-code will be based on this solution based on given knots $\{(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)\}$.

Q-Spline Algorithm

```
INPUT:   (n+1) knots     (vertical and horizontal coordinates)
         x.new           (to be used to evaluate the spline function)
OUTPUT:  y.new

STEP 1:  Define X and Y
         m = number of values in x.new
         y.new = NULL     (to store values of Q-spline function at x.new)
STEP 2:  solve for C from XC = Y       (coefficients  of Q-spline polynomials)
STEP 3:  FOR i = 1 TO m DO:
            IF x[k] <= x.new[i] < x[k+1] DO:
               y.new[i] = C[3k]*x.new[i]^2 + C[3k+1]*x.new[i] + C[3k+2]
            ENDIF
         ENDFOR
```

```
STEP 4: RETURN y.new and coefficients if needed.
```

Next, we write an R function based on the above pseudo-code. We use vector operations in the code to reduce loops.

```
QSpline = function(x,         # x-coordinates of the input knots
                   y,         # y-coordinates of the input knots
                   x.new      # the new x values to be evaluated and returned
              ){
     m = length(x.new)
     n = length(x)-1
     y.new = NULL
     ##
     Y = rep(0, 3*n)
     Y[1] = y[1]
     ##
     A1 = matrix(0, ncol = 3*n, nrow = 2*n)      # continuity condition
     A2 = matrix(0, ncol = 3*n, nrow = n -1)     # smooth condition
     A3 = matrix(0, ncol = 3*n, nrow = 1)        # default initial condition
     for (i in 1:n){  # pay attention to the indexes
       A1[2*i-1,(3*(i-1)+1):(3*(i-1)+3)] = c(x[i]^2, x[i], 1)
       A1[2*i,(3*(i-1)+1):(3*(i-1)+3)] = c(x[i+1]^2, x[i+1], 1)
       if(i == n) break
       A2[i, (3*(i-1)+1):(3*(i-1)+6)] = c(2*x[i+1], 1, 0, -2*x[i+1], -1, 0)
       ##
      }
     A3[1,1] = 1
     X = rbind(A1, A2, A3)
     ##
     Y = rep(0, 3*n)
     Y[1] = y[1]
     for (i in 2:n){
         Y[2*i-2] = y[i]
         Y[2*i-1] = y[i]
        }
     Y[2*n] = y[n+1]
     C=solve(X)%*%Y
     a = C[seq(1,3*n, by = 3),]
     b = C[seq(2,3*n, by = 3),]
     c = C[seq(3,3*n, by = 3),]
     coef = round(cbind(a =a, b = b, c = c),3)
```

```
    ## Prediction
    for (j in 1:m){
      k = which(x >= x.new[j])[1]
      y.new[j] = a[k]*x.new[j]^2 + b[k]*x.new[j] + c[k]
    }
   list(y.new = y.new, QSpoly.coef = coef, QSeq.Matrix = X, Y = Y)
}
```

Example 3: (**Velocity example continued**) The upward velocity of a rocket is given as a function of time in the table below. Using the linear spline to determine the value of the velocity at $t = 16$ seconds.

t (s)	$v(t)$ (m/s)
0	0
10	227.04
15	362.78
20	517.35
22.5	602.97
30	901.67

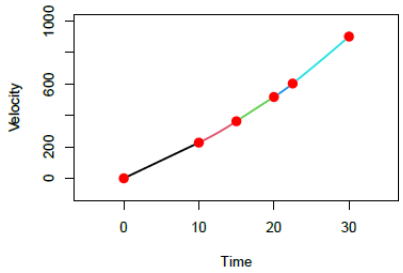
We fit the data with a quadratic spline with the assumption that $a_1 = 0$. The matrix form of the final system of $3 \times (6-1) = 15$ linear equations are given below.

$$\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
100 & 10 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 100 & 10 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 225 & 15 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 225 & 15 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 400 & 20 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 400 & 20 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 506.25 & 22.5 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 506.25 & 22.5 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 900 & 30 & 1 \\
20 & 1 & 0 & -20 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 30 & 1 & 0 & -30 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 40 & 1 & 0 & -40 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 45 & 1 & 0 & -45 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \\ a_3 \\ b_3 \\ c_3 \\ a_4 \\ b_4 \\ c_4 \\ a_5 \\ b_5 \\ c_5 \end{bmatrix}
=
\begin{bmatrix} 0 \\ 227.04 \\ 227.04 \\ 362.78 \\ 362.78 \\ 517.35 \\ 517.35 \\ 602.97 \\ 602.97 \\ 901.67 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The last row in the argument matrix reflects the default assumption $a_1 = 0$. Solve the above system of equations using a computer program, we have the solutions summarized in the following table.

i	a_i	b_i	c_i
1	0	22.704	0
2	0.8888	4.928	88.88
3	−0.1356	35.66	−141.61
4	1.6048	−33.956	554.55
5	0.20889	28.86	−152.13

Using the above results, we plot the quadratic spline with the following R code.



Using the above R function, we have the above results.

```
x = c(0, 10, 15, 20, 22.5, 30)
y = c(0, 227.04, 362.78, 517.35, 602.97, 901.67)
x.new = c(0, 10, 15, 16, 20)
QSpline(x = x, y = y, x.new = x.new)
```

```
## $y.new
## [1]   0.0000 227.0400 362.7800 422.0828 517.3500
##
## $QSpoly.coef
##           a       b       c
## [1,]  0.000  22.704    0.00
## [2,]  0.889   4.928   88.88
## [3,] -0.136  35.660 -141.61
## [4,]  1.605 -33.956  554.55
## [5,]  0.209  28.860 -152.13
##
## $QSeq.Matrix
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]   [,10] [,11] [,12]   [,13] [,14] [,15]
##  [1,]    0    0    1    0    0    0    0    0    0   0.00   0.0     0   0.00   0.0     0
##  [2,]  100   10    1    0    0    0    0    0    0   0.00   0.0     0   0.00   0.0     0
##  [3,]    0    0    0  100   10    1    0    0    0   0.00   0.0     0   0.00   0.0     0
##  [4,]    0    0    0  225   15    1    0    0    0   0.00   0.0     0   0.00   0.0     0
##  [5,]    0    0    0    0    0    0  225   15    1   0.00   0.0     0   0.00   0.0     0
##  [6,]    0    0    0    0    0    0  400   20    1   0.00   0.0     0   0.00   0.0     0
##  [7,]    0    0    0    0    0    0    0    0    0 400.00  20.0     1   0.00   0.0     0
##  [8,]    0    0    0    0    0    0    0    0    0 506.25  22.5     1   0.00   0.0     0
##  [9,]    0    0    0    0    0    0    0    0    0   0.00   0.0     0 506.25  22.5     1
## [10,]    0    0    0    0    0    0    0    0    0   0.00   0.0     0 900.00  30.0     1
## [11,]   20    1    0  -20   -1    0    0    0    0   0.00   0.0     0   0.00   0.0     0
## [12,]    0    0    0   30    1    0  -30   -1    0   0.00   0.0     0   0.00   0.0     0
## [13,]    0    0    0    0    0    0   40    1    0 -40.00  -1.0     0   0.00   0.0     0
## [14,]    0    0    0    0    0    0    0    0    0  45.00   1.0     0 -45.00  -1.0     0
## [15,]    1    0    0    0    0    0    0    0    0   0.00   0.0     0   0.00   0.0     0
##
## $Y
```

```
##  [1]   0.00 227.04 227.04 362.78 362.78 517.35 517.35 602.97 602.97 901.67   0.00
```

Chapter 17

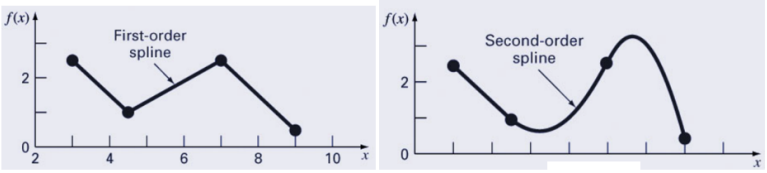
Cubic Spline Interpolation

As we have seen from the previous note on linear and quadratic spline interpolation, splines are functions that are piece-wise polynomials. The coefficients of the polynomial differ from interval to interval, but the order of the polynomial is the same. An essential feature of splines is that function is continuous - i.e. has no breaks on the boundaries between two adjacent intervals. That is, they create smooth curves out of irregular data points.

In this note, we use the same idea to discuss cubic spline interpolation.

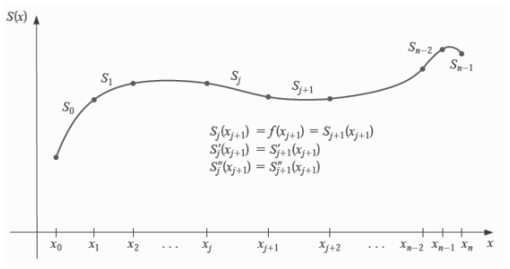
17.1 Cubic Splines

A cubic spline is a piece-wise cubic polynomial function. We will use the same logical process to develop the algorithm to find the cubic polynomials based on given knots.



17.1.1 Formulation of Cubic Spline

Suppose that we want to approximate $y = f(x)$ by cubic spline function $S(x)$. For a sampled set of points on the curve of $f(x)$: $\{(x_0, y_0).(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$.



Assume further that $x_0 < x_1 < x_2 < \cdots < x_n$. On each interval $[x_0, x_1), [x_1, x_2), \cdots, [x_{n-1}, x_n]$, $S(x)$ is given by a different cubic polynomial. Let $S_i(x)$ be the cubic polynomial that represents $S(x)$ on $[x_i, x_{i+1}]$. Thus

$$S(x) = \begin{cases} S_0(x), & x \in [x_0, x_1) \\ S_1(x), & x \in [x_1, x_2) \\ \vdots & \vdots \\ S_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases}$$

Where

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

for $j = 0, 1, \cdots, n-1$. There are $4n$ unknowns to be determined subject to the following conditions:

Condition 1: $S(x_j) = f(x_j)$ for each $j = 0, 1, \cdots, n$; *$(n + 1$ equations$)$*

Condition 2: $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$; *$(n-1$ equations$)$*

Condition 3: $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$; *$(n-1$ equations$)$*

Condition 4: $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for each $j = 0, 1, \cdots, n-2$; *$(n-1$ equations$)$*

Condition 5: One of the following boundary conditions:

a) $S''(x_0) = S''(x_n) = 0 \Rightarrow$ **Natural Spline**. *(2 equations)*

b) $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n) \Rightarrow$ **Clamped Spline**. *Remark: since $f(x)$ is unknown, we need to provide the two slopes according to the trend of the pattern of the scatter plot of the data. (2 equations)*

Example 1. Calculate **Cubic Natural Splines** based on given data $\{(1,5),(2,11),(4,8)\}$ are sampled from an unknown function $y = f(x)$. Find the approximated values of $f(1.5)$ and $f'(2)$ based on the cubic smoothing spline.

Solution: Let $S_0(x) = a_0 + b_0(x-1) + c_0(x-1)^2 + d_0(x-1)^3$ be the spline function on interval $[1,2)$ and $S_1(x) = a_1 + b_1(x-2) + c_1(x-2)^2 + d_1(x-2)^3$ be the spline function on interval $[2,4)$. We need the following 8 equations to determine the two cubic spline functions.

- The two splines pass individual points yielding
  1. $a_0 = 5$
  2. $a_0 + b_0 + c_0 + d_0 = 11$
  3. $a_1 = 11$
  4. $a_1 + 2b_1 + 4c_1 + 8d_1 = 8$
- First order smoothness: $b_0 + 2c_0(x-1) + 3d_0(x-1)^2 = b_1 + 2c_1(x-2) + 3d_1(x-2)^2$
  5. $b_0 + 2c_0 + 3d_0 = b_1$
- Second order smoothness: $2c_0 + 6d_0(x-1) = 2c_1 + 6d_1(x-2)$
  6. $2c_0 + 6d_0 = 2c_1$
- Boundary Conditions (natural spline):
  7. $d_0 = 0$
  8. $d_1 = 0$

We obtain the following system of equations

$$\begin{cases} 1a_0 + 0b_0 + 0c_0 + 0d_0 + 0a_1 + 0b_1 + 0c_1 + 0d_1 = 5 \\ 1a_0 + 1b_0 + 1c_0 + 1d_0 + 0a_1 + 0b_1 + 0c_1 + 0d_1 = 11 \\ 0a_0 + 0b_0 + 0c_0 + 0d_0 + 1a_1 + 0b_1 + 0c_1 + 0d_1 = 11 \\ 0a_0 + 0b_0 + 0c_0 + 0d_0 + 1a_1 + 2b_1 + 4c_1 + 8d_1 = 8 \\ 0a_0 + 1b_0 + 2c_0 + 3d_0 + 0a_1 - 1b_1 + 0c_1 + 0d_1 = 0 \\ 0a_0 + 0b_0 + 1c_0 + 3d_0 + 0a_1 + 0b_1 - 1c_1 + 0d_1 = 0 \\ 0a_0 + 0b_0 + 0c_0 + 1d_0 + 0a_1 + 0b_1 + 0c_1 + 0d_1 = 0 \\ 0a_0 + 0b_0 + 0c_0 + 0d_0 + 0a_1 + 0b_1 + 0c_1 + 1d_1 = 0 \end{cases}$$

The matrix representation of the above system is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 8 \\ 0 & 1 & 2 & 3 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 01 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \\ a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \\ 11 \\ 8 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The solution to the equation is

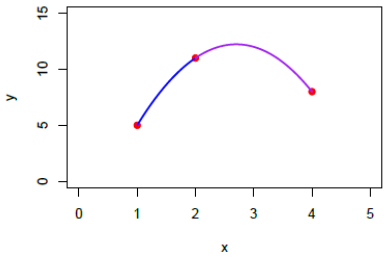
$$\begin{bmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \\ a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 8 \\ 0 & 1 & 2 & 3 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 01 & 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} 5 \\ 11 \\ 11 \\ 8 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 13.5 \\ -7.5 \\ 0 \\ 11 \\ 13.5 \\ -7.5 \\ 0 \end{bmatrix}.$$

We use R to solve the above equation.

Thais, $(a_0, b_0, c_0, d_0, a_1, b_1, c_1, d_1) = (5, 8.5, -2.5, 0, 11, 3.5, -2.5, 0)$. Therefore, the two resulting cubic spline functions (reduce to quadratic functions due to the choice of the natural spline) are

$$S(x) = \begin{cases} 5 + 8.5(x-1) - 2.5(x-1)^2 & \text{for } x \in [1, 2) \\ 11 + 3.5(x-2) - 2.5(x-2)^2 & \text{for } x \in [2, 4]. \end{cases}$$

Therefore, $f(1.5) \approx S_0(1.5) = 5 + 8.5 \times 0.5 - 2.5 \times 0.5^2 = 8.625$. $f'(2) = S_0'(2) = S_1'(2) = [11 + 3.5(x-2) - 2.5(x-2)^2]' = 3.5$.



Example 2: Calculate **Cubic Clamped Splines** based on given data points $\{(0, 1), (2, 2), (5, 0), (8, 0)\}$ are sampled from an unknown function $y = f(x)$.

Solution (sketch): We choose the slopes of the starting and ending points of $f(x)$ to be 2 and 1 respectively. *For convenience, we denote individual segments to have form* $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$. Using the same set of conditions, we list the resulting equations of the coefficients of the four cubic polynomials according to each spline function in the following.

Segment 0: $f_0(x)$	
Slope at 0:	$b_0 = 2$
Curve through (0, 1):	$a_0 = 1$
Curve through (2, 2):	$a_0 + 2b_0 + 4c_0 + 8d_0 = 2$
Slopes match at join with f_1:	$b_0 + 4c_0 + 12d_0 - b_1 - 4c_1 - 12d_1 = 0$
Curvatures match at join with f_1:	$2c_0 + 12d_0 - 2c_1 - 12d_1 = 0$

Segment 1: $f_1(x)$	
Curve through (2, 2):	$a_1 + 2b_1 + 4c_1 + 8d_1 = 2$
Curve through (5, 0):	$a_1 + 5b_1 + 25c_1 + 125d_1 = 0$
Slopes match at join with f_2:	$b_1 + 10c_1 + 75d_1 - b_2 - 10c_2 - 75d_2 = 0$
Curvatures match at join with f_2:	$2c_1 + 30d_1 - 2c_2 - 30d_2 = 0$

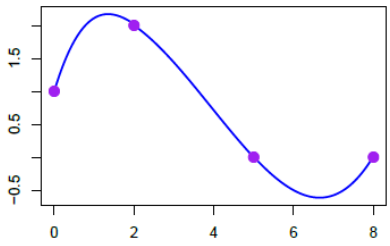
Segment 2: $f_2(x)$	
Curve through (5, 0):	$a_2 + 5b_2 + 25c_2 + 125d_2 = 0$
Curve through (8, 0):	$a_2 + 8b_2 + 64c_2 + 512d_2 = 0$
Slope at 8:	$b_2 + 16c_2 + 192d_2 = 1$

Rewriting the above system of equations in the matrix form, we have

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 4 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 12 & 0 & -1 & -4 & -12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 12 & 0 & 0 & -2 & -12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 5 & 25 & 125 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 10 & 75 & 0 & -1 & -10 & -75 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 30 & 0 & 0 & -2 & -30 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 & 25 & 125 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 8 & 64 & 512 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 192 \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \\ a_1 \\ b_1 \\ c_1 \\ d_1 \\ a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 2 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The solution of the above system equations is $(a_0, b_0, c_0, d_0, a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2,) =$ (1, 2, -1.0395, 0.1447, 1.9201, 0.6199, -0.3494, 0.0297, 0.7018, 1.3509, -0.4956, 0.0395). Therefore, the cubic spline function is given explicitly by

$$S(x) = \begin{cases} 1 + 2x - 1.0339x^2 + 0.1447x^3 & \text{for } x \in [0, 2); \\ 1.9201 + 0.6199x - 0.3494x^2 + 0.0297x^3 & \text{for } x \in [2, 5); \\ 0.7018 + 1.3509x - 0.4956x^2 + 0.0395x^3 & \text{for } x \in [5, 8]. \end{cases}$$



17.1.2 Construction of Cubic Splines

CUBIC SPLINE FUNCTION: $S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$

OBJECTIVE: Find Coefficients: (a_i, b_i, c_i, d_i)**!**

Let $h_j = x_{j+1} - x_j$. Since there are $4n$ unknowns that need to be determined from the data, we need to set up $4n$ equations as follows:

1. From **condition 1**: $S_j(x_j) = y_j$ gives n equations: $a_j = f(x_j) = y_j$ for $j = 0, 1, 2, \cdots, n$.
2. From **Condition 2**: $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ gives $n - 1$ equations: $a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3$ for $j = 0, 1, 2, \cdots, n - 2$.
3. From **condition 3**: Note that $S'_j(x) = b_j + 2c_j(x - x_j) + 3d_j(x - x_j)^2$. The condition $S'_{j+1}(x_{j+1}) = S'_j(x_j)$ gives $n - 1$ equations: $b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2$ for $j = 0, 1, 2, \cdots, n - 2$.
4. From **condition 4**, $S''_j(x) = 2c_j + 6d_j(x - x_j)$: The condition $S''_{j+1}(x_{j+1}) = S''_{j+1}(x_j)$ gives $n - 1$ equations: $c_{j+1} = c_j + 3d_j h_j$ for $j = 0, 1, 2, \cdots, n - 2$.
5. **Boundary conditions**:
   a) $c_0 = c_n = 0$
   b) $S'_n(b) = f'(b) = b_n$ gives $h_{n-1}(c_{n-1} + 2c_n) = 3f'(b) - 3(a_n - a_{n-1})/h_{n-1}$ while $S'_0(a) = f'(a) = b_0$ gives the following equatione $h_0(2c_0 + c_1) = 3(a_1 - a_0)/h_0 - 3f'(a)$.

Instead of solving the system of $3n$ equations based on the given $n+1$ knots $\{(x_0,y_0),(x_1,y_1),\cdots,(x_n,y_n)\}$, we next simplify the original system to reduce it to a relatively small system. to summarize what we obtained earlier, we list the following $3n-2$ equations

A. $a_{j+1}=a_j+b_jh_j+c_jh_j^2+d_jh_j^3$ for $j=0,1,2,\cdots,n.$

B. $b_{j+1}=b_j+2c_jh_j+3d_jh_j^2$ for $j=0,1,2,\cdots,n-1.$

C. $c_{j+1}=c_j+3d_jh_j$ for $j=0,1,2,\cdots,n-1.$

Note that a_j and h_j are known. Solve for d_j from (C), we have $d_j=(c_{j+1}-c_j)/(3h_j)$. Plug d_j into (A) and (B), and we have

D. $a_{j+1}=a_j+b_jh_j+h_j^2(2c_j+c_{j+1})/3.$

E. $b_{j+1}=b_j+h_j(c_j+c_{j+1})$

We can similarly solve b_j from (D) to get

F. $b_j=(a_{j+1}-a_j)/h_j-h_j(2c_j+c_{j+1})/3$ for $j=0,1,2,\cdots,n-1.$

which implies

G. $b_{j-1}=(a_j-a_{j-1})/h_{j-1}-h_{j-1}(2c_{j-1}+c_j)/3$ for $j=1,2,\cdots,n-1.$

We re-write (re-index) (E) as follows

H. $b_j=b_{j-1}+h_{j-1}(c_{j-1}+c_j)$ for $j=1,2,3,\cdots,n-1.$ (see the change of the index).

We substitute b_{j-1} and b_j in (H) with the ones in (F) and (G) and obtain

I. $(a_{j+1}-a_j)/h_j-h_j(2c_j+c_{j+1})/3=(a_j-a_{j-1})/h_{j-1}-h_{j-1}(2c_{j-1}+c_j)/3+h_{j-1}(c_{j-1}+c_j)$ for $j=1,2,\cdots,n-1$, where c_j $(j=1,2,...,n-1)$ are unknowns.

Finally, we end up with a linear system that involves unknowns $c_1,c_2,\cdots,c_{n-2}$ in the following standard form.

J.: $h_{j-1}c_{j-1}+2c_j(h_{j-1}+h_j)+h_jc_{j+1}=3(a_{j+1}-a_j)/h_j-3(a_j-a_{j-1})/h_{j-1}$ for $j=1,2,...,n-2$.

We next use the boundary condition **a)** to add two equations $c_0=c_{n-1}=0$ to (**J**). The resulting cubic spline is called the **NATURAL CUBIC SPLINE.**

The coefficient matrix of the system along with the two boundary conditions is given by

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}$$

Since every row of A has three non-zero elements (except for the first and the last rows), S is also called **tridiagonal matrix**.

Definition A **diagonally dominant matrix** is the square matrix whose absolute value of the diagonal element is **greater than or equal to** the sum of the absolute value of corresponding off-diagonal elements.

Definition A **Strictly diagonally dominant matrix** is the square matrix whose absolute value of the diagonal element is **strictly greater than** the sum of the absolute value of corresponding off-diagonal elements.

Theorem 1 A strictly diagonally dominant complex matrix is non-singular.

Proof: One can use the concept of eigen theory and proof-by-contradiction to prove this theorem. The proof will not be given in this note.

Theorem 2: If f is defined at $a = x_0 < x_1 < \cdots < x_n = b$, then f has a unique natural cubic spline interpolated function $S(x)$.

Note that A is strictly diagonally dominant. Therefore, A is invertible. Therefore, there is a unique solution to the system. Therefore, the natural cubic spline function is unique.

The resulting matrix of the equation based on **(J)** and the two boundary conditions of natural cubic spline is given by

$$A\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3(a_2-a_1)/h_1 - 3(a_1-a_0)/h_0 \\ 3(a_3-a_2)/h_2 - 3(a_2-a_1)/h_1 \\ \vdots \\ 3(a_n-a_{n-1})/h_{n-1} - 3(a_{n-1}-a_{n-2})/h_{n-2} \\ 0 \end{bmatrix}$$

Using **theorem 2**, there is a unique solution

$$\begin{bmatrix} c_0 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{bmatrix} = A^{-1}\begin{bmatrix} 0 \\ 3(a_2-a_1)/h_1 - 3(a_1-a_0)/h_0 \\ 3(a_3-a_2)/h_2 - 3(a_2-a_1)/h_1 \\ \vdots \\ 3(a_n-a_{n-1})/h_{n-1} - 3(a_{n-1}-a_{n-2})/h_{n-2} \\ 0 \end{bmatrix}$$

17.1.3 Cubic Spline Algorithm

Three key steps are needed in developing the algorithm.

1. Find the solution of equation (**J**) to find $c(c_0, c_1, \cdots, c_{n-1})$.
2. Using the backward substitution to find b_i and d_i $(i = 0, 1, \cdots, n-1)$

Because of multiple steps in simplification, the pseudo-code looks more complex than earlier algorithms.

Pseudo-code:

```
INPUT:   x         (x-coordinated of given knots)
         y         (y-coordinates of given knots)
         x.new     (new x-values for prediction)
OUTPUT:  coefficients
         y.new
STEP 1:  initialization
         A         (nxn zero matrix for solving c_i)
         Y         (nx1 zero matrix in the matrix equation J)
         n         (number of knots)
         m         (number of x.new)
         h = diff of adjacent input X
         aa        (coef of C-spline function)
         bb
         cc
         dd
STEP 2:  FOR i =2 TO (n-1) DO:
```

```
            A[i,(i-1):(i+1)] = (h[i-1], 2*(h[i-1]+h[i]), h[i])
            Y[i] = 3*(y[i+1]-y[i])/h[i] - 3*(y[i]-y[i-1])/h[i-1]
          ENDFOR
STEP 3:   Solve for cc  (coefficient of C-spline polynomials)
STEP 4:   aa = y
STEP 5:   FOR j = 1 TO (n-1) DO:
            dd[j] = (cc[i+1]-cc[i])/(3*h[i])
            bb[j] = (aa[i+1]-aa[i])/h[i] - h[i]*(2*cc[i] + cc[i+1])/3
          ENDFOR
STEP 6:   FOR k = 1 TO m DO:
            IF  (x[s] <= x.new[k] < x[s+1]) DO:
              y.new[k] = aa[i]+ bb[i]*(x.new[j] - x[i]) + cc[i]*(x.new[j] - x[i])^2 + c
            ENDIF
          ENDFOR
STEP 7: RETURN y.new and C-spline coefficients
```

R Code

```
###
Natural.CSpline = function(x,
                           y,
                           x.new){
  n = length(x)
  m = length(x.new)
  h = xvec[-1] - xvec[-n]
  A = matrix(rep(0,(n)^2), ncol=n)
  Y = rep(0,n)
  y.new = NULL
  Y[c(1,n)] = 0
  A[1,1] = 1
  A[n,n] = 1
  for (i in 2:(n-1)){
     A[i,(i-1):(i+1)] = c(h[i-1], 2*(h[i-1]+h[i]), h[i])
     Y[i] =3*(y[i+1]-y[i])/h[i]-3*(y[i]-y[i-1])/h[i-1]
  }
  cc = as.vector(solve(A)%*%Y)
  aa = y                  # input y values
  ID = 1:(n-1)
  bb = dd = NULL
  for (i in 1:(n-1)){
    dd[i] = (cc[i+1] - cc[i])/(3*h[i])
    bb[i] = (aa[i+1] - aa[i])/h[i] - h[i]*(2*cc[i] + cc[i+1])/3
  }
  cc = cc[-n]
```

```
  aa = aa[-n]
  coef = cbind(ID = ID, aa = aa, bb = bb, cc = cc, dd = dd)
  #predicted value
  dS = NULL                          # derivative of the C-spline.
  for (j in 1:length(x.new)){
  for (i in 1:n){
    if (x.new[j]<= x[i]){
       next
      }
     y.new[j] = aa[i] + bb[i]*(x.new[j] - x[i]) + cc[i]*(x.new[j] - x[i])^2 + dd[i]*(x.new[j] - x
     dS[j] = bb[i] + 2*cc[i]*(x.new[j] - x[i]) +3* dd[i]*(x.new[j] - x[i])^2
   }
  }
 list(coef = coef, y.new = y.new, d.S = dS)
}
```

Example 3: Approximate $f(x) = \log(e^x + 2)$ using nodes $x = -1. - 0.5, 0, 0.5$.

```
xvec=c(-1,-0.5,0,0.5)
yvec=log(exp(xvec)+2)
###
Natural.CSpline(x = xvec, y = yvec, x.new = 0.25)
```

```
## $coef
##      ID        aa        bb         cc          dd
## [1,]  1 0.8619948 0.1756378 0.00000000  0.06565087
## [2,]  2 0.9580201 0.2248760 0.09847631  0.02828097
## [3,]  3 1.0986123 0.3445630 0.14089776 -0.09393184
##
## $y.new
## [1] 1.192091
##
## $d.S
## [1] 0.3973997
```

Example 4 (Example 2 of the textbook, page 150): Use the data points $(0, 1), (1, e), (2, e^2)$, and $(3, e^3)$ to form a natural spline $S(x)$ that approximates $f(x) = e^x$.

```
xvec=c(0, 1, 2, 3)
yvec=exp(xvec)
x.new = c(1, 1.5, 2)
```

```
###
Natural.CSpline(x = xvec, y = yvec, x.new = x.new)
```

```
## $coef
##      ID       aa       bb        cc         dd
## [1,]  1 1.000000 1.465998 0.0000000  0.2522842
## [2,]  2 2.718282 2.222850 0.7568526  1.6910714
## [3,]  3 7.389056 8.809770 5.8300668 -1.9433556
##
## $y.new
## [1] 2.718282 4.230304 7.389056
##
## $d.S
## [1] 2.222850 4.248006 8.809770
```

17.2 Clamped Splines

The only difference between natural splines and clamped splines is the use of boundary conditions. In the natural cubic spline, the boundary conditions are $c_0 - c_{n-1} = 0$.

The clamped splines use the (clamped) boundary conditions: $S'(x_0) = f'(x_0$ and $S'(x_n) = f'(x_n)$.

We will not develop an R function to implement the clamped cubic spline interpolation. Both R and MATLAB have built-in functions to perform clamped cubic spline interpolation.

17.3 Error Analysis

Theorem: Let $f \in C^4[a,b]$ with $\max_{a\le x\le b}|f^{(4)}(x)| = M$. If S is the unique clamped cubic spline interpolant to f with respect to nodes $a = x_0 < x_1 < \cdots < x_n = b$, then for all $x \in [a,b]$,

$$|f(x) - S(x)| \le \frac{5M \cdot \max_{0\le x\le n-1}|x_{j+1} - x_j|^4}{384}$$

Using the above theorem, we can find the error bound of the clamped cubic spline.

The error bound of the natural spline is dependent on the 4-th order derivative of $f(x)$. It is more complicated than that of the clamped cubic spline. We will not introduce this in this course.

Chapter 18

Newton Method for Nonlinear System and Optimization

We have introduced the direct methods in matrix algebra and several interpolation methods to approximate a univariate function with either an explicitly given analytic expression or with an unknown expression. We use matrix algebra to solve problems of least-square approximation of multiple-variable functions. In this note, we will introduce Newton's methods to solve nonlinear system equations with applications in optimization.

18.1 Multivariate Taylor Expansion

The recursive algorithm of Newton's method for single variable nonlinear equations is derived based on the Taylor expansion. The method can be generalized to solve the system of nonlinear functions. We consider Taylor expansion of two-variable function $f(x, y)$ at (x_0, y_0). For convenience

$$f(x,y) = f(x_0,y_0) + f_x(x_0,y_0)(x-x_0) + f_y(x_0,y_0)(y-y_0)$$

$$+\frac{f_{x^2}(x_0,y_0)}{2!}(x-x_0)^2 + \frac{f_{y^2}(x_0,y_0)}{2!}(y-y_0)^2 + \frac{f_{xy}(x_0,y_0)}{1!1!}(x-x_0)(y-y_0)$$

$$+\cdots+\sum_{k+m=n 0\le k,\ m\le n} \frac{f_{x^k y^m}(x_0,y_0)}{k!\ m!}(x-x_0)^k(y-y_0)^m + E_n$$

where

$$f_{x^k}(x_0,y_0)=\frac{\partial^k f(x,y)}{\partial x^k}\Big|_{x=x_0 y=y_0},\quad f_{y^k}(x_0,y_0)=\frac{\partial^k f(x,y)}{\partial y^k}\Big|_{x=x_0 y=y_0},\quad f_{x^p y^q}(x_0,y_0)=\frac{\partial^{p+q} f(x,y)}{\partial x^p \partial y^q}\Big|_{x=x_0,y=}$$

and E_n is the remainder term.

Example 1: Find the third order Taylor expansion of $f(x,y)=e^{2x}\sin(3y)$ about $(x_0,y_0)=(0,0)$ using the above formula. We first compute all partial derivatives up to order 3 at (x_0,y_0).

$$\begin{array}{ll}
f(x,y)=e^{2x}\sin(3y) & f(x_0,y_0)=0\\
f_x(x,y)=2e^{2x}\sin(3y) & f_x(x_0,y_0)=0\\
f_y(x,y)=3e^{2x}\cos(3y) & f_y(x_0,y_0)=3\\
f_{x^2}(x,y)=4e^{2x}\sin(3y) & f_{x^2}(x_0,y_0)=0\\
f_{xy}(x,y)=6e^{2x}\cos(3y) & f_{xy}(x_0,y_0)=6\\
f_{y^2}(x,y)=-9e^{2x}\sin(3y) & f_{y^2}(x_0,y_0)=0\\
f_{x^3}(x,y)=8e^{2x}\sin(3y) & f_{x^3}(x_0,y_0)=0\\
f_{x^2y}(x,y)=12e^{2x}\cos(3y) & f_{x^2y}(x_0,y_0)=12\\
f_{xy^2}(x,y)=-18e^{2x}\sin(3y) & f_{xy^2}(x_0,y_0)=0\\
f_{y^3}(x,y)=-27e^{2x}\cos(3y) & f_{y^3}(x_0,y_0)=-27
\end{array}$$

Using the above formula, we have

$$e^{2x}\sin(3y)=\frac{3}{1}y+\frac{6}{1!1!}xy+\frac{12}{2!1!}x^2y-\frac{27}{3!}y^3+E_3(x,y)\quad=3y+6xy+6x^2y-4.5y^3+E_3(x,y).$$

18.2 Newton Method for Nonlinear Systems

To find the root of nonlinear equation $f(x)=0$, we assume that $f(x)$ is differentiable. Using Taylor's expansion, we have

$$f(x)\approx f(x_0)+f'(x_0)(x-x_0)\quad\Rightarrow\quad x=x_0-\frac{f(x_0)}{f'(x_0)}$$

18.2.1 System of Two Nonlinear Equations

Consider system

$$\left\{\begin{array}{rcl} f_1(x,y) &=& 0\\ f_2(x,y) &=& 0\end{array}\right.$$

As we did in single function equation, we take the first order derivative of both $f_1(x,y)$ and $f_2(x,y)$, the do linear approximations to both functions

$$\begin{cases} f_1(x,y) &= f_1(x_0,y_0) + \frac{\partial f_1(x_0,y_0)}{\partial x}(x-x_0) + \frac{\partial f_1(x_0,y_0)}{\partial y}(y-y_0) \\ f_2(x,y) &= f_2(x_0,y_0) + \frac{\partial f_2(x_0,y_0)}{\partial x}(x-x_0) + \frac{\partial f_2(x_0,y_0)}{\partial y}(y-y_0) \end{cases}$$

The above system can be written in the following matrix equation

$$\begin{bmatrix} \frac{\partial f_1(x_0,y_0)}{\partial x} & \frac{\partial f_1(x_0,y_0)}{\partial y} \\ \frac{\partial f_2(x_0,y_0)}{\partial x} & \frac{\partial f_2(x_0,y_0)}{\partial y} \end{bmatrix} \begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} = -\begin{bmatrix} f_1(x_0,y_0) \\ f_2(x_0,y_0) \end{bmatrix}$$

The coefficient matrix in the above matrix equation is called **Jacobian matrix** and is denoted by $\mathbf{J}$. If the inverse of

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1(x_0,y_0)}{\partial x} & \frac{\partial f_1(x_0,y_0)}{\partial y} \\ \frac{\partial f_2(x_0,y_0)}{\partial x} & \frac{\partial f_2(x_0,y_0)}{\partial y} \end{bmatrix}$$

exists, then

$$\begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} = -\begin{bmatrix} \frac{\partial f_1(x_0,y_0)}{\partial x} & \frac{\partial f_1(x_0,y_0)}{\partial y} \\ \frac{\partial f_2(x_0,y_0)}{\partial x} & \frac{\partial f_2(x_0,y_0)}{\partial y} \end{bmatrix}^{-1} \begin{bmatrix} f_1(x_0,y_0) \\ f_2(x_0,y_0) \end{bmatrix}$$

which can be further re-expressed as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} - \begin{bmatrix} \frac{\partial f_1(x_0,y_0)}{\partial x} & \frac{\partial f_1(x_0,y_0)}{\partial y} \\ \frac{\partial f_2(x_0,y_0)}{\partial x} & \frac{\partial f_2(x_0,y_0)}{\partial y} \end{bmatrix}^{-1} \begin{bmatrix} f_1(x_0,y_0) \\ f_2(x_0,y_0) \end{bmatrix}.$$

The Newton method of the above system of two nonlinear equations is based on the following recursive relationship

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \begin{bmatrix} \frac{\partial f_1(x_k,y_k)}{\partial x} & \frac{\partial f_1(x_k,y_k)}{\partial y} \\ \frac{\partial f_2(x_k,y_k)}{\partial x} & \frac{\partial f_2(x_k,y_k)}{\partial y} \end{bmatrix}^{-1} \begin{bmatrix} f_1(x_k,y_k) \\ f_2(x_k,y_k) \end{bmatrix}.$$

Denote

$$\mathbf{X}_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix}, \quad \mathbf{X}_k = \begin{bmatrix} x_k \\ y_k \end{bmatrix}, \quad \mathbf{h} = -\begin{bmatrix} \frac{\partial f_1(x_k,y_k)}{\partial x} & \frac{\partial f_1(x_k,y_k)}{\partial y} \\ \frac{\partial f_2(x_k,y_k)}{\partial x} & \frac{\partial f_2(x_k,y_k)}{\partial y} \end{bmatrix}^{-1} \begin{bmatrix} f_1(x_k,y_k) \\ f_2(x_k,y_k) \end{bmatrix}.$$

Then the recursive relationship is given by

$$\mathbf{X}_{k+1} = \mathbf{X}_k + \mathbf{h}.$$

18.2.2 Newton Algorithm

With the above recursive relation, we develop the following *brief* pseudo-code

```
 INPUT: fn,       (vector of the system of nonlinear equations)
        J,        (Jacobian matrix based on fn)
        ini.val,
        TOL,
        maxit
OUTPUT: sol, etc.

STEP 1: initialization
         iterator: i = 1
         err = 1  (any number that is bigger than TOL)
         sol      (initialize the matrix to store output information)
STEP 2: WHILE err > TOL AND i < maxit DO
        h = inverse(J)xY
        new.x = ini.x + h  (updating x vector)
        ENDWHILE
STEP 3: RETURN sol
```

Example 2: Solve the following system of nonlinear equations

$$\begin{cases} x^2 + y^2 &= 4 \\ xy &= 1 \end{cases}$$

which corresponds to finding the intersection points of a circle and a hyperbola in the plane.

Solution: In order to use the Newton method, we need to find the Jacobian matrix. Denote $f_1(x, y) = x^2 + y^2 - 4$ and $f_2(x, y) = xy - 1$. Then

$$J(x, y) = \begin{bmatrix} \frac{\partial f_1(x_k, y_k)}{\partial x} & \frac{\partial f_1(x_k, y_k)}{\partial y} \\ \frac{\partial f_2(x_k, y_k)}{\partial x} & \frac{\partial f_2(x_k, y_k)}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

```r
##
fn.vec=function(ini.val){
 x=ini.val[1]
 y=ini.val[2]
 f1 = x^2+y^2 -4
 f2 = x*y-1
 c(f1,f2)
}

Jacobian=function(ini.val){
 x=ini.val[1]
 y=ini.val[2]
 f1.x = 2*x
```

```
 f1.y = 2*y
 f2.x = y
 f2.y = x
 m=matrix(c(f1.x, f1.y, f2.x, f2.y), ncol=2, byrow=T)
 m
}

Newton=function(fn.vec, Jacobian, ini.val, tol, maxit=100){
 x=ini.val[1]
 y=ini.val[2]
 ### initialization
 err=1
 i = 1
 sol.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 err.vec = rep(0, maxit)
 fn.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 while(err > tol  && i < maxit){
   h = - solve(Jacobian(ini.val))%*%fn.vec(ini.val)
   new.val = ini.val + h
   err=max(abs(h))
   ## store intermediate outputs
   err.vec[i] = err
   sol.mtx[i,] = as.vector(new.val)
   fn.mtx[i,] = fn.vec(new.val)
   ## updating the root and the iteration ID
   ini.val=new.val
   i = i + 1
 }
 id = which(err.vec==0)[1]-1   # locate the starting rows with all zero cells
 list(solution = sol.mtx[1:id,], error = err.vec[1:id], fn.values = fn.mtx[1:id,])
}
# function call

Newton(fn.vec, Jacobian, ini.val=c(1,1.5), tol=10^(-4))
```

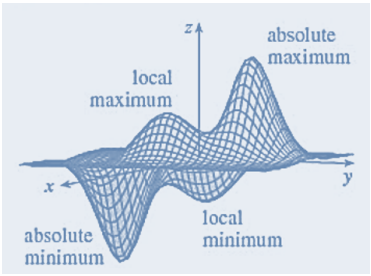
```
## $solution
##           [,1]     [,2]
## [1,] 0.1000000 2.350000
## [2,] 0.4400227 2.009467
## [3,] 0.5137997 1.935690
## [4,] 0.5176277 1.931862
## [5,] 0.5176381 1.931852
##
## $error
## [1] 9.000000e-01 3.405329e-01 7.377707e-02 3.828039e-03 1.036171e-05
```

```
##
## $fn.values
##               [,1]          [,2]
## [1,] 1.532500e+00 -7.650000e-01
## [2,] 2.315781e-01 -1.157889e-01
## [3,] 1.088610e-02 -5.443052e-03
## [4,] 2.930776e-05 -1.465388e-05
## [5,] 2.147305e-10 -1.073650e-10
```

18.3 Optimization

For ease of presentation, we focus on bivariate functions. Multivariate functions can be treated similarly.

```
knitr::include_graphics("img14/w14-optimization.png")
```



First, we recall some of the results in Calculus. Let $f(x, y$ be a two variable real function. $f(x, y$ has a local maximum at (a, b) if $f(x, y) \leq f(a, b)$ when (x, y) is in the neighborhood of (a, b). $f(a, b)$ is the local maximum value. $f(x, y$ has a local maximum at (a, b) if $f(x, y) \geq f(a, b)$ when (x, y) is in the neighborhood of (a, b). $f(a, b)$ is the local minimum value.

Theorem 1 If $f(x, y)$ has a local maximum or minimum at (a, b) and the first order partial derivatives of $f(x, y)$ exist, the $f_x(a, b) = 0$ and $f_y(a, b) = 0$.

Theorem 2 Suppose the second order partial derivative of $f(x, y)$ are continuous on a disk with center (a, b) (i.e., the neighborhood of (a, b)), and assume that $f_x(a, b) = 0$ and $f_y(a, b) = 0$ [that is, (a, b) is a critical point of $f(x, y)$]. Let

$$D(a, b) = f_{x^2}(a, b)f_{y^2}(a, b) - [f_{xy}(a, b)]^2.$$

(a). If $D > 0$ and $f_{x^2}(a, b) > 0$, then $f(a, b)$ is a local minimum.

(b). If $D > 0$ and $f_{x^2}(a, b) < 0$, then $f(a, b)$ is a local maximum.

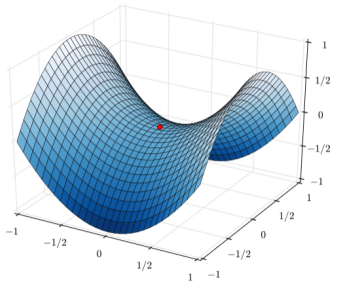
(c). If $D < 0$, then $f(a, b)$ is not a local minimum or maximum.

Notations $f_{x^2}(a, b)$, $f_{y^2}(a, b)$, and $f_{xy}(a, b)$ are specified earlier.

Some Remarks:

- In case (c) the point (a, b) is called a **saddle point** of $f(x, y)$ and the graph of $f(x, y)$ crosses its tangent plane at (a, b). The red point on the surface (hyperbolic paraboloid) in the following figure is a saddle point.

```
knitr::include_graphics("img14/w14-saddlePoint.png")
```



- If $D = 0$, the test gives no information. $f(x, y)$ could have a local maximum or local minimum at (a, b), or (a,b) could be a saddle point of $f(x, y)$.
- To remember the formula for $D(a, b)$, it's helpful to write it as a determinant:

$$D(a,b) = \begin{bmatrix} f_{x^2}(a,b) & f_{xy}(a,b) \\ f_{xy}(a,b) & f_{y^2}(a,b) \end{bmatrix} = f_{x^2}(a,b) f_{y^2}(a,b) - [f_{xy}(a,b)]^2$$

- The matrix D in the above expression is called **Hessian Matrix** in the optimization problem.

Example 3: Find the critical value of $f(x, y) = 2x^3/3 + 2xy^2 - 8x - 4y + 6$ and justify whether the critical point is a local maximum or local minimum or a saddle point.

Solution We first find the critical point(s) of by solving the following system of nonlinear equations

$$\begin{cases} f_x(x,y) = 2x^2 + 2y^2 - 8 &= 0 \\ f_y(x,y) = 4xy - 4 &= 0 \end{cases}$$

The above system is identical to the nonlinear system in **Example 2**. We know the solution to the above system is $(a, b) = (0.5176381, 1.931852)$. To see whether $(0.5176381, 1.931852)$ is a local extreme value or a saddle point, we need to know the **Hessian matrix**

$$D(x, y) = \begin{bmatrix} 4x & 4y \\ 4y & 4x \end{bmatrix}$$

The determinant of the Hessian matrix at $(0.5176381, 1.931852)$ is

$$D(0.5176381, 1.931852) = \det \begin{bmatrix} 4 \times 0.5176381 & 4 \times 1.931852 \\ 4 \times 1.931852 & 4 \times 0.5176381 \end{bmatrix} = 14(0.5176381^2 - 1.931852^2) < 0.$$

Example 4: Find and classify the critical points of the function

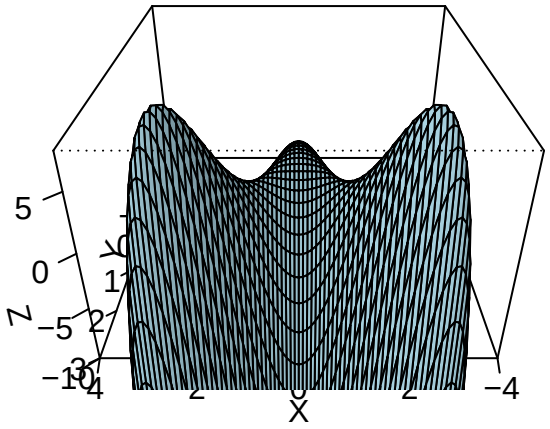
$$f(x, y) = 10x^2y - 5x^2 - 4y^2 - x^4 - 2y^4$$

Also find the highest point on the graph of $f(x, y)$.

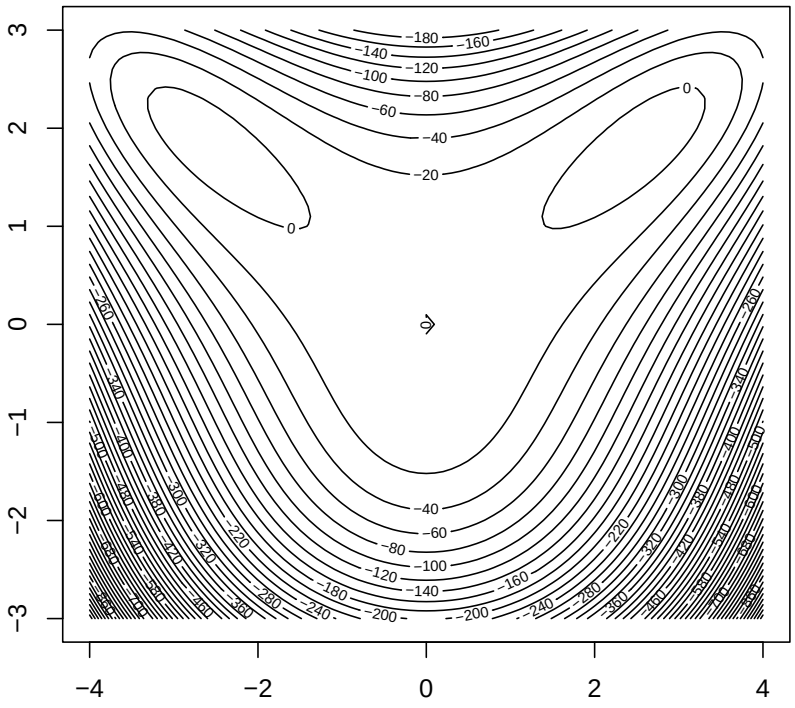
Solution: We first make a 3D

```
FN=function(x,y) 10*y*x^2-5*x^2-4*y^2-x^4-2*y^4
x=seq(-4,4,0.1)
y=seq(-3,3,0.1)
z=outer(x,y,FN)
persp(x, y, z, theta = 180, phi = 30, expand = 0.6, col = "lightblue", ltheta = 90,
shade = 0.15, ticktype = "detailed", xlab = "X", ylab = "Y", zlab = "Z", zlim =c(-10,8
)
```

```
## Warning in persp.default(x, y, z, theta = 180, phi = 30, expand = 0.6, col = "lightb
```




```
contour(x,y,z, nlevels = 40)
```



The 3D surface indicates that there are three local extrema and two saddle points on the surface. We need to choose appropriate initial values to locate the critical points and saddle points. We first take partial derivatives of the given function $f(x, y)$ and set them to zero.

$$\begin{cases} f_x(x,y) &=& 20xy - 10x - 4x^3 &=& 0 \\ f_y(x,y) &=& 10x^2 - 8y - 8y^3 &=& 0 \end{cases}$$

The Jacobian matrix of the above nonlinear system is given by

$$\mathbf{J} = \begin{bmatrix} f_{x^2}(x,y) & f_{xy}(x,y) \\ f_{xy}(x,y) & f_{y^2}(x,y) \end{bmatrix} = \begin{bmatrix} 20y - 12x^2 - 10 & 20x \\ 20x & -24y^2 - 8 \end{bmatrix}$$

We now use the Newton method with different initial values to find all critical values (guided by the surface in the figure).

```
fn.vec=function(ini.val){
 x=ini.val[1]
 y=ini.val[2]
 fx = 20*x*y-10*x-4*x^3
 fy = 10*x^2-8*y-8*y^3
 c(fx,fy)
}

Jacobian=function(ini.val){
 x=ini.val[1]
 y=ini.val[2]
 fxx = 20*y-10-12*x^2
 fxy = 20*x
 fyx = 20*x
 fyy = -8-24*y^2
 m=matrix(c(fxx, fxy, fyx, fyy), ncol=2, byrow=T)
 m
}

Newton.Raphson=function(fn.vec, Jacobian, ini.val, tol, maxit=100, trace = TRUE){
 x=ini.val[1]
 y=ini.val[2]
 ### initialization
 err = 1
 i = 1
 sol.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 err.vec = rep(0, maxit)
 fn.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 while(err > tol  && i < maxit){
   h = - solve(Jacobian(ini.val))%*%fn.vec(ini.val)
   new.val = ini.val + h
   err=max(abs(h))
   ## store intermediate outputs
   err.vec[i] = err
   sol.mtx[i,] = as.vector(new.val)
   fn.mtx[i,] = fn.vec(new.val)
   ## updating the root and the iteration ID
   ini.val=new.val
   i = i + 1
  }
  id = which(err.vec==0)[1]-1   # locate the starting rows with all zero cells
  ##  Determinant of Hessian
  D = det(Jacobian(sol.mtx[id,]))
  fxx = Jacobian(sol.mtx[id,])[1,1]
  ##
```

```
  if(trace ==TRUE){
      list(solution = sol.mtx[1:id,],  error = err.vec[1:id], fn.values = fn.mtx[1:id,]
  } else {
   if(D > 0 & fxx > 0) extreme = "local minimum"
   if(D > 0 & fxx < 0) extreme = "local maximum"
   if(D < 0 ) extreme = "saddle point"
   list(iterations = id, solution = sol.mtx[id,], D = D, fxx = fxx, extreme = extreme,
  }
 }
```

```
Newton.Raphson(fn.vec, Jacobian, ini.val=c(0.8,0.6), tol=10^(-4), trace = FALSE)
```

```
## $iterations
## [1] 3
##
## $solution
## [1] 0.8566569 0.6467722
##
## $D
## [1] -187.6363
##
## $fxx
## [1] -5.870888
##
## $extreme
## [1] "saddle point"
##
## $error
## [1] 3.268639e-06
##
## $fn.values
## [1]  9.349321e-11 -1.186673e-10
```

```
Newton.Raphson(fn.vec, Jacobian, ini.val=c(-0.8,0.6), tol=10^(-4), trace = FALSE)
```

```
## $iterations
## [1] 3
##
## $solution
## [1] -0.8566569  0.6467722
##
## $D
## [1] -187.6363
##
## $fxx
## [1] -5.870888
```

```
##
## $extreme
## [1] "saddle point"
##
## $error
## [1] 3.268639e-06
##
## $fn.values
## [1] -9.349321e-11 -1.186673e-10
```

```
Newton.Raphson(fn.vec, Jacobian, ini.val=c(0.1,0.2), tol=10^(-4), trace = FALSE)
```

```
## $iterations
## [1] 4
##
## $solution
## [1] -4.419185e-11 -1.840406e-10
##
## $D
## [1] 80
##
## $fxx
## [1] -10
##
## $extreme
## [1] "local maximum"
##
## $error
## [1] 1.213394e-05
##
## $fn.values
## [1] 4.419185e-10 1.472325e-09
```

```
Newton.Raphson(fn.vec, Jacobian, ini.val=c(-2.8,1.6), tol=10^(-4), trace = FALSE)
```

```
## $iterations
## [1] 5
##
## $solution
## [1] -2.644224  1.898384
##
## $D
## [1] 2488.717
##
## $fxx
## [1] -55.93538
##
```

```
## $extreme
## [1] "local maximum"
##
## $error
## [1] 7.298599e-07
##
## $fn.values
## [1]  8.540724e-12 -9.613643e-12
```

```
Newton.Raphson(fn.vec, Jacobian, ini.val=c(-2.8,1.6), tol=10^(-4), trace = FALSE)
```

```
## $iterations
## [1] 5
##
## $solution
## [1] -2.644224  1.898384
##
## $D
## [1] 2488.717
##
## $fxx
## [1] -55.93538
##
## $extreme
## [1] "local maximum"
##
## $error
## [1] 7.298599e-07
##
## $fn.values
## [1]  8.540724e-12 -9.613643e-12
```

18.4 A Case Study

Many studies indicated that body mass index (BMI) is a more powerful risk factor for diabetes than genetics. The objective of this case study is to explore the association between BMI and diabetes. Studies show that the probability of diabetes and MBI have the following relationship.

$$\text{Prob(diabetes)} = \frac{e^{\alpha_0+\alpha_1 BMI}}{1+e^{\alpha_0+\alpha_1 BMI}}$$

Let x be the measurement BMI and y be the status of diabetes that is numerically encoded as

$$y = \begin{cases} 1 & \text{if the diabetes test result is positive,} \\ 0 & \text{if the diabetes test result is negative.} \end{cases}$$

For a given set of historical observations (i.e., nodes) $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$. Using the theory of likelihood, we can approximate unknowns α_0 and α_1 by **maximizing** the following logarithmic likelihood function defined based on the given data points

$$\mathbf{LL}(\alpha_0, \alpha_1) = -\sum_{i=1}^{n} \ln[1 + e^{\alpha_0 + \alpha_1 x_i}] + \sum_{i=1}^{n} y_i(\alpha_0 + \alpha_1 x_i)$$

To find the critical values, we set up the following nonlinear system of equations

$$\frac{\partial LL}{\partial \alpha_0} = -\sum_{i=1}^{n} \frac{e^{\alpha_0 + \alpha_1 x_i}}{1 + e^{\alpha_0 + \alpha_1 x_i}} + \sum_{i=1}^{n} y_i = 0$$

$$\frac{\partial LL}{\partial \alpha_1} = -\sum_{i=1}^{n} \frac{e^{\alpha_0 + \alpha_1 x_i}}{1 + e^{\alpha_0 + \alpha_1 x_i}} x_i + \sum_{i=1}^{n} x_i y_i = 0$$

The Jacobian matrix of the above system is

$$\mathbf{J} = \begin{bmatrix} \frac{\partial^2 LL}{\partial \alpha_0^2} & \frac{\partial^2 LL}{\partial \alpha_0 \alpha_1} \\ \frac{\partial^2 LL}{\partial \alpha_0 \alpha_1} & \frac{\partial^2 LL}{\partial \alpha_1^2} \end{bmatrix}$$

where

$$\frac{\partial^2 LL}{\partial \alpha_0^2} = -\sum_{i=1}^{n} \frac{e^{\alpha_0 + \alpha_1 x_i}}{[1 + e^{\alpha_0 + \alpha_1 x_i}]^2}, \quad \frac{\partial^2 LL}{\partial \alpha_0 \alpha_1} = -\sum_{i=1}^{n} \frac{x_i e^{\alpha_0 + \alpha_1 x_i}}{[1 + e^{\alpha_0 + \alpha_1 x_i}]^2}, \quad \frac{\partial^2 LL}{\partial \alpha_1^2} = -\sum_{i=1}^{n} \frac{x_i^2 e^{\alpha_0 + \alpha_1 x_i}}{[1 + e^{\alpha_0 + \alpha_1 x_i}]^2},$$

The Newton recursive formula is given by

$$\begin{bmatrix} \alpha_0^{(k+1)} \\ \alpha_1^{(k+1)} \end{bmatrix} = \begin{bmatrix} \alpha_0^{(k)} \\ \alpha_1^{(k)} \end{bmatrix} - \begin{bmatrix} \frac{\partial^2 LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_0^2} & \frac{\partial^2 LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_0 \alpha_1} \\ \frac{\partial^2 LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_0 \alpha_1} & \frac{\partial^2 LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_1^2} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_0} \\ \frac{\partial LL(\alpha_0^{(k)}, \alpha_1^{(k)})}{\partial \alpha_1} \end{bmatrix}$$

Note $\mathbf{J}$ is the Jacobian matrix of (the vector of) the two nonlinear functions in the derived systems of nonlinear equations. It is also the Hessian matrix of the log-likelihood function.

Next, we write an R function to return the Jacobian matrix and the vector of the values of the two gradients of $\mathbf{LL}(\alpha_0, \alpha_1)$.

```
JH = function(x, y, a0, a1){
  # x = x-coordinates
  # y = y-coordinates
  # a0 = alpha_0, a1 = alpha_1
  E = exp(a0+a1*x)
  L0 = -sum(E/(1+E)) + sum(y)
  L1 = -sum(x*E/(1+E)) + sum(x*y)
  L00 = -sum(E/(1+E)^2)
  L01 = -sum(x*E/(1+E)^2)
  L10 = L01
  L11 =  -sum(x^2*E/(1+E)^2)
  LV = c(L0, L1)
  J = matrix(c(L00, L01, L10, L11), ncol =2, byrow = TRUE)
  list(LV = LV, Jacob = J)
}
```

```
Newton.Alg=function(x, y, ini.val, tol, maxit=100, trace = TRUE){
 ### initialization
 err = 1
 i = 1
 sol.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 err.vec = rep(0, maxit)
 fn.mtx = matrix(0, nrow=maxit, ncol=length(ini.val))
 while(err > tol  && i < maxit){
 ### initialization
   JnFun = JH(x, y, a0 = ini.val[1],a1 = ini.val[2])
   JMatrix = JnFun$Jacob
   fn.vec = JnFun$LV
   if(det(JMatrix) == 0){
     cat("\n\n The Hessian matrix is singular!")
     break
   }
   h = - solve(JMatrix)%*%fn.vec
   new.val = ini.val + h
   err=max(abs(h))
   ## store intermediate outputs
   err.vec[i] = err
   sol.mtx[i,] = as.vector(new.val)
   fn.mtx[i,] = JH(x, y, a0 = ini.val[1],a1 = ini.val[2])$LV
   ## updating the root and the iteration ID
   ini.val=new.val
   i = i + 1
  }
  id = which(err.vec==0)[1]-1   # locate the starting rows with all zero cells
  ##  Determinant of Hessian
```

```
  SOL = sol.mtx[id,]
  D = det(JH(x, y, a0 = SOL[1], a1 = SOL[2])$Jacob)
  #D = det(Jacobian(sol.mtx[id,]))
  fxx = (JH(x, y, a0 = SOL[1], a1 = SOL[2])$Jacob)[1,1]
  ##
  if(trace ==TRUE){
      list(solution = sol.mtx[1:id,],  error = err.vec[1:id], fn.values = fn.mtx[1:id,])
  } else {
   if(D > 0 & fxx > 0) extreme = "local minimum"
   if(D > 0 & fxx < 0) extreme = "local maximum"
   if(D < 0 ) extreme = "saddle point"
   list(iterations = id, solution = sol.mtx[id,],
        D = D,
        fxx = fxx,
        extreme = extreme,
        error = err.vec[id],
        fn.values = fn.mtx[id,])
  }
 }
```

Next, we use diabetes data to illustrate the above predictive algorithm. The data is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of collecting the data is to diagnostically **predict whether or not a patient has diabetes**, based on certain diagnostic measurements included in the data. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of *Pima Indian heritage*.

A portion of the above data with missing components removed is stored in the GitHub repository of this course: https://raw.githubusercontent.com/pengdsci/MAT325/main/w12/diabetes.csv

We only use one of the potential risk factors, BMI, to make a prediction. We use

```
diabetes = read.csv("https://raw.githubusercontent.com/pengdsci/MAT325/main/w12/diabetes.csv")
diabeticStatus = diabetes$Outcome      # y-variable: 1 = diabetes, 0 = no-diabetes
BMI = diabetes$BMI                     # x-variable (risk factor)
#glm(diabeticStatus ~ BMI, family = binomial(link = logit))
Newton.Alg(x=BMI, y=diabeticStatus, ini.val=c(-6, .2), tol = 10^(-8), maxit=100, trace = FALSE)
```

```
## $iterations
## [1] 7
##
## $solution
```

```
## [1] -3.60614324  0.08632952
##
## $D
## [1] 276736.4
##
## $fxx
## [1] -80.481
##
## $extreme
## [1] "local maximum"
##
## $error
## [1] 1.009945e-10
##
## $fn.values
## [1] -5.07498e-10 -7.73980e-09
```

The potential challenge of using Newton's method is to find the initial value so that the algorithm converges to the solution to the system of the nonlinear equations or the optimization problem.

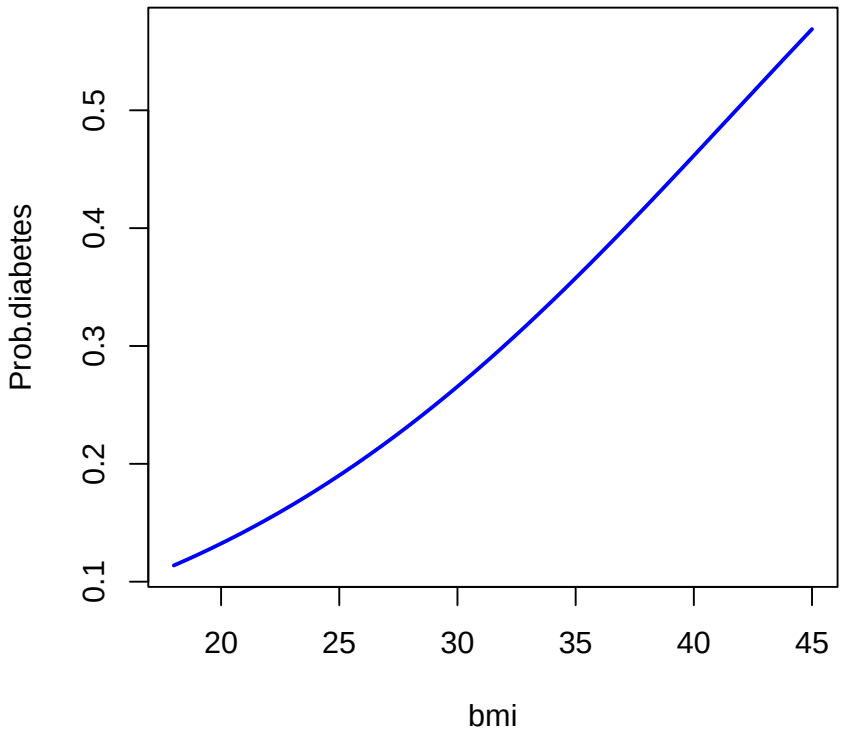
Recall that the practical question is to predict diabetes diagnostically with given risk factors. This case study involves only one factor - BMI. With the above result of the Newton method, we have the predictive model in the following form

$$\text{P[Y = diabetes]} = \frac{e^{-3.6061+0.0863\times BMI}}{1+e^{-3.6061+0.0863\times BMI}}.$$

The above function predicts the probability of being diabetic with a given BMI measurement. For example, if someone's BMI is 30, the probability of getting diabetes is

$$\text{P[Y = diabetes]} = \frac{e^{-3.6061+0.0863\times 30}}{1+e^{-3.6061+0.0863\times 30}} \approx 0.2656$$

```
bmi=seq(18,45, length = 50)
E = exp(-3.6061 + 0.0863*bmi)
Prob.diabetes = E/(1 + E)
plot(bmi, Prob.diabetes, type = "l", lwd=2, col = "blue")
```



The above figure shows that the chance of getting diabetes increases as BMI increases.

Chapter 19

Numerical Integration I

This note introduces several methods for numerical integration. Before discussing the specific method, we introduce numerical approximation for derivatives based on the Taylor series and then use some definitions and ideas to develop numerical methods for approximating integrals.

19.1 Numerical Approximation of Derivatives

The function $f(x)$ can be expanded over a small interval t using the Taylor series from a start or reference point

$$y(x+t) = y(x) + y'(x)t + \frac{y''(x)}{2!}t^2 + \frac{y'''(x)}{3!}t^3 + \frac{y^{(4)}(x)}{4!}t^4 + \cdots$$

This forms an approximation to $f(x)$ in t which runs, say, from x_i to x_j, and can be used to integrate $y(t)$. For example, if we keep the first four terms on the right-hand side of the above series in integration, we get

$$I = \int_0^h y dt = \int_0^h \left[y + y't + \frac{y''}{2!}t^2 + \frac{y'''}{4!}t^4 + O(t^4 y^{(4)}) \right] dt$$

$$= hy + \frac{h^2}{2!}y' + \frac{h^3}{3!}y'' + \frac{h^4}{4!}y''' + O[h^5 y^{(4)}]$$

where $h_i = x_{i+1} - x_i = h$ is a constant.

Alternatively, we can rewrite the Taylor series as

$$y(x+t) = y(x) + y'(x)t + O(y''t^2)$$

Solving y' from the above equation

$$y'(x) = \frac{y(x+t) - y(x)}{t} - O(y''t)$$

by setting t to h in the Taylor series, we may also approximate the first derivative

$$y'(x) = \frac{y(x+h) - y(x)}{h} - O(hy'').$$

That is,

$$y'_i = \frac{y_{i+1} - y_i}{h} - O(hy'')$$

The last expression, **a first-forward finite difference**, will be used to complete the trapezoid rule below. Note that this is only the **first order accurate**.

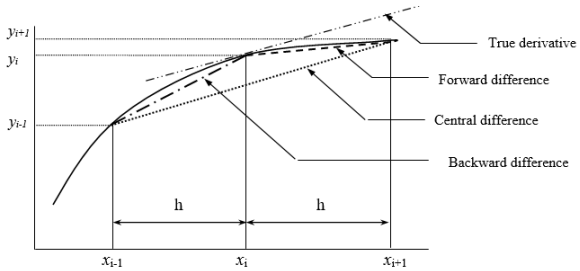
The well-known **second-order accurate central difference** approximations for derivatives are obtained by writing a backward Taylor expansion $y(x-t)$ and letting $t = h$. Then subtracting it from $y(x+t)$ yields

$$y(x+t) = y(x)+y'(x)t+\frac{y''(x)}{2!}t^2+\frac{y'''(x)}{3!}t^3+O(y^{(4)}t^4) y(x-t) = y(x)+y'(x)(-t)+\frac{y''(x)}{2!}(-t)^2+\frac{y'''(x)}{3!}(-t$$

$$y(x+t) + y(x-t) = 2y(x) + y''(x)(t^2) + O(y^{(4)}t^4).$$

Solve for $y''(x)$, we have

$$y''(x) = \frac{y(x+t) + y(x-t) - 2y(x)}{t^2} - O(y^{(4)}t^2).$$



19.2 Trapezoid Method

The concept of the trapezoidal rule comes from approximating $y(t)$ at any point t in an interval by a linear polynomial using Taylor's series and then integrating the approximation.

https://github.com/pengdsci/MAT325/raw/main/w13/img/w12-trapezoid-sum.gif

19.2.1 Formulation of Trapezoid Method

If the approximation is a linear function, the resulting integration is called the trapezoidal rule. For the data interval $[x_i, x_{i+1}]$

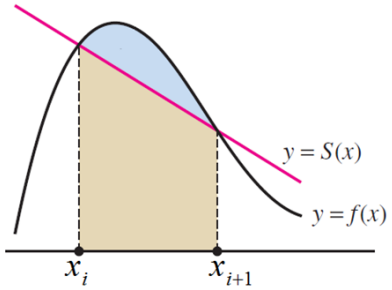
$$I_{x_i}^{x_{i+1}} = \int_0^h y dt = \int_0^h \left[y_i + y't + O(y''t^2)\right] dt = \int_0^h y_i dt + \int_0^h y' t dt + \int_0^h O(y''h^2) dt = hy_i + \frac{y'h^2}{2} + O(y''h^3) = hy_i + \frac{h^2}{2}$$

This is the trapezoidal rule for one interval.

19.2.2 Error Analysis

The **Error bound** of the trapezoid rule for one interval $[x_i, x_{i+1}]$ is easy to see that $S(x)$ in the following figure has an equation

$$S(x) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} x + \frac{x_{i+1} f(x_i) - x_i f(x_{i+1})}{x_{i+1} - x_i}$$



The error bound we will estimate is based on the following inequality

$$E_i = \left| \int_{x_i}^{x_{i+1}} [f(x) - S(x)]\, dx \right|$$

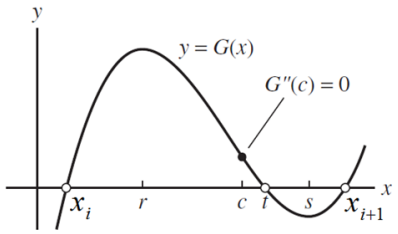
Define the auxiliary function

$$G(x) = f(x) - S(x) - q(x - x_i)(x - x_{i+1}),$$

where q is a constant. For a given $t \in (x_i, x_{i+1})$, we can choose q such that $G(t) = 0$. That is,

$$q = \frac{f(t) - S(t)}{(t - x_i)(t - x_{i+1})}.$$

Therefore, $G(x) = 0$ has three distinct roots $x = x_i, t, x_{i+1}$. If $f(x)$ is twice differentiable, there exists $c \in (x_i, x_{i+1})$ such that $G''(c) = 0$.



On the other hand, $G''(x) = f''(x)$–$2q$ implies that $q = f''(c)/2$. We rewrite the expression q and obtain the following

$$f(t) - S(t) = \frac{f''(c)}{2}(t - x_i)(t - x_{i+1}).$$

A direct calculation gives

$$\int_{x_i}^{x_{i+1}} [f(t) - S(t)]dt = \frac{f''(c)}{2} \int_{x_i}^{x_{i+1}} (t - x_i)(t - x_{i+1})dt = \frac{f''(c)}{2} \left[-\frac{(x_{i+1} - x_i)^3}{6} \right]$$

Therefore, if $f''(c) \leq K_i$ on (x_i, x_{i+1}), we have

$$E_i = \left| \int_{x_i}^{x_{i+1}} [f(x) - S(x)]\, dx \right| \leq \frac{h^3 K_i}{12}$$

where $h = x_{i+1} - x_i$, $\{a = x_0, x_1, \cdots, x_{n-1}, x_n = b\}$ be equally spaced partition of $[a, b]$.

19.2.3 Composite Trapezoid Method and Error Analysis

Using the same idea, we estimate the overall error over interval $[a, b]$ as follows

$$I_a^b = \sum_{i=1}^{n} \frac{h}{2}(y_i + y_{i-1}) + \sum_{i=1}^{n} O(h^3 y'') = \frac{h}{2}\left(y_0 + 2\sum_{i=2}^{n-1} y_i + y_n\right) + \sum_{i=1}^{n} O(h^3 y'').$$

This is called the **composite trapezoidal rule**.

The overall **error bound** for the **composite trapezoidal rule** can be similarly estimated as

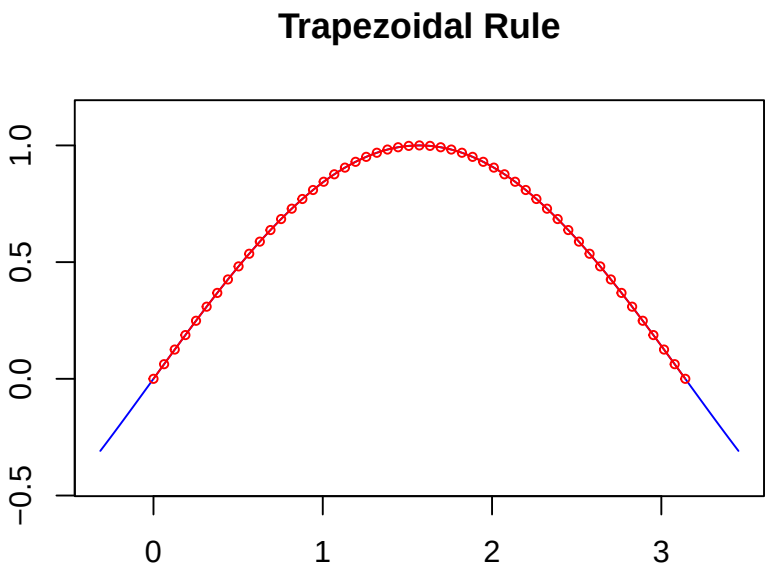
$$E = \sum_{i=1}^{n} E_i \leq \sum_{i=1}^{n} \frac{h^3 K_i}{12} = \frac{1}{12}\left(\frac{b-a}{n}\right)^3 \sum_{i=1}^{n} K_i$$

19.2.4 Trapezoid Algorithm

The algorithm of the trapezoid method is relatively simple. We assume the partition of the given interval $[a, b]$ is equally spaced. We will not develop the pseudo-code. The following is the R function of the trapezoid method.

```
Trapezoid.int = function(fun,           # fun = user-defined function
                         xvec,          # interval [a, b]
                         n = 1,         # number of partitions
                         graph=TRUE){   # request graphical representation
  a = min(xvec)
  b = max(xvec)
  m = length(xvec)
  if (n >= m) xvec = seq(a,b,(b-a)/n)
  yvec = fun(xvec)
  nn = length(yvec)
  h = xvec[-1]- xvec[-nn]
  y.adjacent.sum =  yvec[-1] + yvec[-nn]
  Iab = sum(h*y.adjacent.sum)/2
  if(graph == TRUE){
      x.margin = 0.1*abs(b-a)
      tt = seq(a-x.margin, b+x.margin, (b-a+2*x.margin)/10^4)
      lim.x = c(a-x.margin, b+x.margin)
      y.max = max(fun(tt))
      y.min = min(fun(tt))
      y.margin = 0.1*abs(y.max-y.min)
```

```
      lim.y = c(y.min - y.margin, y.max + y.margin)
      plot(tt, fun(tt), type="l", col="blue", xlim=lim.x, ylim=lim.y, xlab=" ",ylab="")
      title("Trapezoidal Rule")
      lines(xvec, yvec, type="l", col="red")
      points(xvec, yvec, pch=21, col="red", cex = 0.6)
    }
  Iab
}
fun=function(x) sin(x)
Trapezoid.int(fun=fun, xvec=c(0,pi), n = 50, graph=TRUE)
```




```
## [1] 1.999342
```

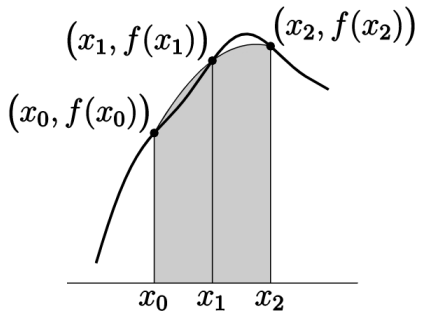
19.3 Simpson's One-third Rule

Simpson's One-third rule is an extension of the Trapezoidal rule where the integrand is approximated by a second-order polynomial.

Simpson's rule approximates the integral over two neighboring sub-intervals by the area between a parabola and the x-axis. To describe this parabola we need 3 distinct points (which is why we approximate two sub-integrals at a time). That is, we approximate

$$\int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx = \int_{x_0}^{x_2} f(x)dx$$

by the area bounded by the parabola that passes through the three points $(x_0, f(x_0)), (x_1, f(x_1))$ and $(x_2, f(x_2))$. The x-axis and the vertical lines $x = x_0$ and $x = x_2$



The next animated graph shows the process of the approximation over an interval with different partitions.

https://github.com/pengdsci/MAT325/raw/main/w12/img/w12-Simpsons_One-Third_Rule.gif

19.3.1 Derivation of Simpson's Rule

To derive Simpson's rule formula, we first find the equation of the parabola that passes through the three points $(x_0, f(x_0)), (x_1, f(x_1))$ and $(x_2, f(x_2))$. Then we find the area between the x-axis and the part of that parabola with $x_0 \leq x \leq x_2$. Assuming an equally spaced partition, we consider the three points $(-h, y_{-h}), (0, y_0)$ and (h, y_h) first (without loss of generality) to develop the formula of the approximated formula that is only dependent on the y-coordinates and the width of the sub-intervals $h = x_{i+1} - x_i$, for $i = 0, 1, \cdots, n-1$.

Using the quadratic function in the following form,

$$y = ax^2 + bx + c$$

we write the area between it and the x-axis with x running from $-h$ to h as

$$\int_{-h}^{h}[ax^2+bx+c]dx=\frac{h}{3}(2ah^2+6c)$$

Because the quadratic function passes through $(-h,y_{-h}),(0,y_0)$ and (h,y_h), so we have

$$\begin{array}{cccc} ah^2 & -bh & +c & = & y_{-h} \\ & & c & = & y_0 \\ ah^2 & +bh & +c & = & y_h \end{array}$$

Therefore, we can express $2ah^2+6c=y_{-h}+4y_0+y_h$, consequently

$$\text{Area}=\int_{-h}^{h}(ax^2+bx+c)dx=\frac{h}{3}(y_{-h}+4y_0+y_h)$$

Remarks

- h is one-half of the length of the x-interval under consideration.
- y_{-h} is the height of the parabola at the left-hand end of the interval under consideration.
- y_0 is the height of the parabola at the middle point of the interval under consideration.
- y_h is the height of the parabola at the right-hand end of the interval under consideration.

19.3.2 Explicit Function on $[x_{i-1},x_{i+1}]$

Assume that $y=A_ix^2+B_ix+C_i$ passes through $(x_{i-1},f(x_{i-1})),(x_i,f(x_i))$ and $(x_{i+1},f(x_{i+1}))$. This implies that

$$\begin{array}{cccc} A_ix_{i-1}^2 & -B_ix_{i-1} & +C_i & = & y_{i-1} \\ A_ix_i^2 & -B_ix_i & +C_i & = & y_i \\ A_ix_{i+1}^2 & -B_ix_{i+1} & +C_i & = & y_{i+1} \end{array}$$

The solution to the above system is given by

$$\begin{array}{rcl} C_i & = & \frac{x_{i-1}^2f(x_{i+1})+x_{i-1}x_{i+1}f(x_{i+1})-4x_{i-1}x_{i+1}f(x_i)+x_{i-1}x_{i+1}f(x_{i-1})+x_{i+1}^2f(x_{i-1})}{(x_{i+1}-x_{i-1})^2} \\ B_i & = & \frac{x_{i-1}f(x_{i-1})-4x_{i-1}f(x_i)+3x_{i-1}f(x_{i+1})+3x_{i+1}f(x_{i-1})-4x_{i+1}f(x_i)+x_{i+1}f(x_{i+1})}{(x_{i+1}-x_{i-1})^2} \\ A_i & = & \frac{2[f(x_{i-1})-2f(x_i)+f(x_{i+1})]}{(x_{i+1}-x_{i-1})^2} \end{array}$$

19.3.3 The Composite Simpsons' Rule

To approximate the integral over interval $[a, b]$, we calculate the approximate integrals $[x_0, x_2], [x_2, x_4], \cdots, [x_{n-2}, x_n]$ separately and then take the sum of all individual integral.

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{\Delta x}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

and

$$\int_{x_2}^{x_4} f(x)dx \approx \frac{\Delta x}{3}[f(x_2) + 4f(x_3) + f(x_4)],$$

and so on. Summing these all together gives

$$I_a^b = \int_a^b f(x)dx \approx \frac{\Delta x}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+2f(x_{n-2})+4f(x_{n-1})+f(x_n)].$$

$$= \frac{\Delta x}{3}\left[f(x_0) + 4\sum_{i=1,\ i=\text{odd}}^{n-1} f(x_i) + 2\sum_{i=2,\ i=\text{even}}^{n-2} f(x_i) + f(x_n)\right]$$

19.3.4 Error Analysis

Error bound of Simpson's rule: Without loss of generality, we consider interval $[-h, +h]$ (i.e., $a = -h, b = +h, (a + b)/2 = 0$). Let $f^{(n)}$ be the n-th derivative of $f(x)$ evaluated at $x = 0$. On one hand,

$$I = \int_{-h}^{h} f(x)dx = \int_{-h}^{h}\left[f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + \frac{f'''(0)}{3!}x^3 + \frac{f^{(4)}(0)}{4!}x^4 + O(x^5)\right]dx$$

$$= 2hf(0) + \frac{2h^3}{6}f''(x_0) + \frac{2h^5}{5!}f^{(4)} + O(h^7)$$

On the other hand,

$$S = \frac{h}{3}[f(-h) + 4f(0) + f(h)] = \frac{h}{3}\left[6f(0) + \frac{2h^2}{2}f''(0) + \frac{2h^4}{4!}f^{(4)} + O(h^6)\right]$$

$$= 2hf(0) + \frac{2h^3}{6}f''(0) + \frac{2h^5}{3 \times 4!}f^{(4)} + O(h^7)$$

Therefore,

$$E = |I - S| = \frac{h^5}{90} f^{(5)}(0) + O(h^6) = \frac{h^5}{90} f^{(4)}(\xi) \quad \text{for} \quad \xi \in (-h, h).$$

Let's consider three points x_{i-1}, x_i, x_{i+1} and $h = x_{i+1} - x_i = x_i - x_{i-1} = (b-a)/n$. Then

$$E_i = |I_i - S_i| = \frac{h^5}{90} f^{(4)}(\xi) = \frac{(x_{i+1} - x_{i-1})^5}{2880} f^{(4)}(\xi)$$

In general, for a given interval $[a, b]$, let equally spaced partition $a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$. The error

$$E_1 = \frac{(x_2 - x_0)^5}{2880} f^{(4)}(\xi_1) = \frac{h^5}{90} f^{(4)}(\xi_1), \quad x_0 < \xi_1 < x_2$$

$$E_2 = \frac{(x_4 - x_2)^5}{2880} f^{(4)}(\xi_2) = \frac{h^5}{90} f^{(4)}(\xi_2), \quad x_2 < \xi_2 < x_4$$

$$E_3 = \frac{(x_6 - x_4)^5}{2880} f^{(4)}(\xi_3) = \frac{h^5}{90} f^{(4)}(\xi_3), \quad x_4 < \xi_3 < x_6$$

Assuming is an even number, then

$$E_{n/2} = \frac{(x_n - x_{n-2})^5}{2880} f^{(4)}(\xi_{n/2}) = \frac{h^5}{90} f^{(4)}(\xi_{n/2}), \quad x_{n-2} < \xi_{n/2} < x_n$$

Therefore, the total approximation error is given by

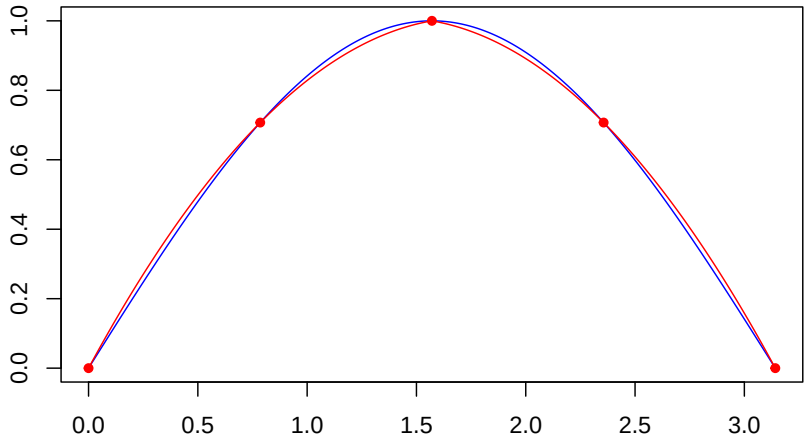
$$E = \sum_{i=1}^{n/2} E_i = \frac{h^5}{90} \sum_{i=1}^{n/2} f^{(4)}(\xi_i) = \frac{1}{90} \left(\frac{b-a}{n} \right)^5 \sum_{i=1}^{n/2} f^{(4)}(\xi_i)$$

19.3.5 Algorithm of Simpson's Rule

Simpson's one-third rule is defined based on intervals $[x_0, x_2], []x_2, x_4], \cdots, [x_{n-2}, x_n]$. The function is not necessarily to be known if both x-coordinates and y-coordinates of the $n + 1$ points are given. Because the explicit formula of the approximation is explicitly given. We can simply write a function in either R or MATLAB to implement the Simpson Rule.

R code

```r
SimpsonGraph = function(fun = NULL,      # fun = user-defined function
                        xvec,            # vec = interval [a,b]
                        n                # number of partitions: MUST be an even number.
                        ){
  a = min(xvec)
  b = max(xvec)
  m = length(xvec)
  x0= seq(a,b,(b-a)/n)
  y0 = fun(x0)
  ##
  xx = seq(a, b, length = 500)
  yy = fun(xx)
  ##
  id1 = (1:length(x0))%%2 ==1
  x = x0[id1]
  nn = length(x)
  plot(xx, yy, type = "l", col = "blue", xlab = "", ylab = "", main = "")
  points(x0, y0, pch = 19, col = "red", cex = 0.8)
  fn = fun
  A = matrix(0, ncol = 3, nrow = nn-1)
  for(i in 1:(nn-1)){
    xi = seq(x[i], x[i+1], length=100)
    #cat("\n\n i =",i,".")
    A2 = 2*(fn(x[i]) -2*fn((x[i]+x[i+1])/2)+fn(x[i+1]))/(x[i+1]-x[i])^2
    A1 = -(x[i]*fn(x[i])-4*x[i]*fn((x[i]+x[i+1])/2)+3*x[i]*fn(x[i+1])+3*x[i+1]*fn(x[i])-4*x[i+1]*
    A0 = (x[i]^2*fn(x[i+1])+x[i]*x[i+1]*fn(x[i+1])-4*x[i]*x[i+1]*fn((x[i+1]+x[i])/2)+x[i]*x[i+1]*
    yi = A2*xi^2 + A1*xi + A0
    lines(xi, yi, col = "red")
    A[i,] = c(A2, A1, A0)
  }
  list(coef = A)
}
fun=function(x) sin(x)
SimpsonGraph(fun = fun, xvec=c(0,pi), n = 4)
```




```
## $coef
##             [,1]      [,2]      [,3]
## [1,] -0.3357489 1.1640129 0.0000000
## [2,] -0.3357489 0.9455595 0.3431458
```

```
fun=function(x) sin(x)
##
Simpson.int=function(fun = NULL,      # fun = user-defined function
                     xvec,            # vec = interval [a,b]
                     n           # number of partitions: MUST be an even number.
                     ){
  a = min(xvec)
  b = max(xvec)
  m = length(xvec)
  if (n > m) x = seq(a,b,(b-a)/n)
  y = fun(x)
  nn = length(y)
  h = x[-1] - x[-nn]
  ### caution: the formula starts from i = 0!
  fx0 = y[1]
  y.0 = y[-1]
  N = length(y.0)
  id02 = seq(2,N-2,2)
  id01 = seq(1,N-1,2)
  Iab = ((b-a)/(3*N))*(fx0 + 4*sum(y.0[id01])+2*sum(y.0[id02])+y.0[N])
```

```
  list(Estimated.Area = Iab)
}

Simpson.int(fun=fun, xvec=c(0,pi), n = 18)
```

```
## $Estimated.Area
## [1] 2.00001
```

Chapter 20

Romberg Integration and Gaussian Quadratures

Interpolation is to estimate a value between a given set of known values. Extrapolation is to use of known values to project a value outside of the intended range of the previous values.

In trapezoid and Simpson's methods, the accuracy of the approximation increases as the step size $h = (b-a)/n$ decreases (i.e., n gets bigger). From the computational perspective, reducing h requires increasing the number of points n, and so increases the "cost" (time and other resources needed) of the calculation.

https://github.com/pengdsci/MAT325/raw/main/w13/img/w13-trapezoid-sum.gif

In this note, we will introduce methods that use the same step size h but archives a more accurate approximation.

20.1 Richardson Extrapolation

Richardson's extrapolation process is a well known method to improve the order of several approximation processes. It can be applied not only to improve the order of a numerical differentiation formula but also to find in fact the original formula.

In this section, we use the concept of Richardson Extrapolation to demonstrate how a higher-order integration can be achieved using only a series of values

from the Trapezoidal Rule. Similarly, accurate values of derivatives could be obtained using low-order central difference derivatives.

20.1.1 The Logic of the Richardson Method

Assume that $\phi(h)$ is infinitely continuously differentiable as a function of h, thus allowing us to expand $\phi(h)$ in the Taylor series

$$\phi(h) = \phi(0) + h\phi'(0) + \frac{\phi''(0)}{2!}h^2 + \frac{\phi'''(0)}{3!}h^3 + \frac{\phi^{(4)}(0)}{4!}h^4 + O(h^5)$$

Let $c_i = \frac{\phi^{(i)}(0)}{i!}$. We rewrite the above Taylor expansion to get

$$\phi(h) = \phi(0) + c_1 h + c_2 h^2 + c_3 h^3 + c_4 h^4 + O(h^5)$$

Apparently,

$$\phi(h/2) = \phi(0) + \frac{c_1}{2}h + \frac{c_2}{4}h^2 + \frac{c_3}{8}h^3 + \frac{c_4}{16}h^4 + O(h^5)$$

Define $\psi(h) = 2\phi(h/2) - \phi(h)$. We have

$$\psi(h) = \phi(0) - \frac{c_2}{2}h^2 - \frac{3c_3}{4}h^3 - \frac{7c_4}{8}h^4 - O(h^5)$$

Note that $\psi(h)$ also approximates $\phi(0)$, but with an $O(h^2)$ error, rather than the $O(h)$ error. For small h, **this $O(h^2)$ approximation will be considerably more accurate**.

If we repeat what we did on $\phi(x)$ to $\psi(x)$, that is

$$\psi(h/2) = \phi(0) - \frac{c_2}{8}h^2 - \frac{3c_3}{32}h^3 - \frac{7c_4}{128}h^4 - O(h^5)$$

To cancel the two terms that contain h^2, we define $\psi_1(h) = [4\psi(h/2) - \psi(h)]/3$ in the following

$$\psi_1(h) = \phi(0) + \frac{4\psi(h/2) - \psi(h)}{3} = \frac{c_3}{8}h^3 + \frac{7c_4}{32}h^4 + O(h^5)$$

Similarly, $\psi_1(0) = \psi(0) = \phi(0)$ but the approximation $\psi_1(0)$ has an $O(h^3)$ error. This means $\psi_1(0)$ more accurate than $\psi(0)$.

We can continue this procedure repeatedly, each time improving the accuracy by one order, at the cost of one additional computation with a smaller h.

20.1.2 Algorithm of Richardson Method

To facilitate generalization and to avoid a further tangle of notations for developing an algorithm, we use two indices and define

$$\begin{aligned} R(j,0) &:= \phi(h/2^j) && j \geq 0; \\ R(j,k) &:= \frac{2^k R(j,k-1) - R(j-1,k-1)}{2^k - 1} && j \geq k > 0. \end{aligned}$$

This procedure is called Richardson extrapolation after the British applied mathematician Lewis Fry Richardson, a pioneer of the numerical solution of partial differential equations, weather modeling, and mathematical models in political science.

With the above notations, we summarize the previous derivation of various approximations of $\phi(0)$ in the following table.

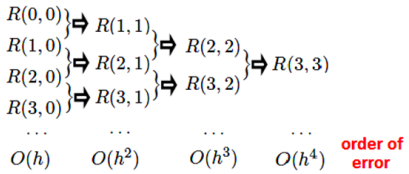
R(0; 0) = **ϕ(h),**
R(1; 0) = ϕ(h/2), R(1; 1) = **ψ(h);**
R(2; 0) = ϕ(h/2²), R(2; 1) = ψ(h/2), R(2; 2) = **ψ₁(h)**

In general, the recursive algorithm can be represented in the following triangular extrapolation table.

$R(0,0)$				
$R(1,0)$	$R(1,1)$			
$R(2,0)$	$R(2,1)$	$R(2,2)$		
$R(3,0)$	$R(3,1)$	$R(3,2)$	$R(3,3)$	
$\cdots$	$\cdots$	$\cdots$	$\cdots$	
$O(h)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	order of error

Remarks:

1. We expect the bottom-right element in the table to be the most accurate approximation to $\phi(0)$.
2. The recursive process is built on the approximations in the first column (see the following flow chart).



3. Each of the cells in the triangular table is a valid approximation of $\phi(0)$.

20.1.3 Higher Order Extrapolation

If the initial algorithm $\phi(h)$ is better than $O(h)$ accurate and in this case, the formula for $R(j;k)$ should be adjusted to take advantage. That is, if

$$\phi(h) = \phi(0) + c_1 h^r + c_2 h^{2r} + c_3 h^{3r} + c_4 h^{4r} + O(h^{5r})$$

for some integer $x \geq 1$. For example,

$$\cos(h) = 1 - \frac{1}{2!}h^2 + \frac{1}{4!}h^{2\times 2} - \frac{1}{3!}h^{2\times 3}h^6 + O(h^{2\times 4})$$

where $r = 2$.

We then can define the following recursive relationship below.

$$\begin{array}{lcll} R(j,0) & := & \phi(h/2^j) & j \geq 0; \\ R(j,k) & := & \frac{2^{rk}R(j,k-1) - R(j-1,k-1)}{2^{rk}-1} & j \geq k > 0. \end{array}$$

The corresponding extrapolation table is given by

$$\begin{array}{cccc} \boxed{R(0,0)} & & & \\ R(1,0) & \boxed{R(1,1)} & & \\ R(2,0) & R(2,1) & \boxed{R(2,2)} & \\ R(3,0) & R(3,1) & R(3,2) & \boxed{R(3,3)} \\ \cdots & \cdots & \cdots & \cdots \quad \ddots \\ \uparrow & \uparrow & \uparrow & \uparrow \\ O(h^r) & O(h^{2r}) & O(h^{3r}) & O(h^{4r}) \quad \boxed{\text{Error Order}} \end{array}$$

20.1.4 Algorithm and Code

The recursive extrapolation procedure itself is simple arithmetic. It is easy to make a R/MATLAB function to implement the algorithm.

```
Richardson.Method = function(fn,           # input function
                             h,            # step size: h
                             r = 1,        # input r-th order of error
                             digit = 7,    # number of digits to keep
                             J             # number of extrapolations
                            ){
  RR = matrix(rep(NA,J^2), ncol = J)      # Output table of Richardson
```

```
  for (i in 1:J) RR[i,1] = fn(h/2^(i-1)) # defining R(j,0)
  for (k in 2:J){
      for (j in k:J){                    #
          RR[j,k] = (2^(r*(k-1))*RR[j,k-1] - RR[j-1, k-1])/(2^(r*(k-1))-1)
       }
  }

  options(digits = digit)
  print(RR, na.print = "")
}
```

Example 1: Let $f(x) = \exp(x)$. Find $f'(1)$ with $h = 1$.

Solution Define

$$\phi(h) = \frac{f(1+h) - f(1)}{h}$$

Then $f'(1) = \lim_{h\to 0} \phi(h) = f'(1)$. We use Richardson extrapolation to approximate $f'(1)$.

```
fn = function(h) (exp(1+h)-exp(1))/h
Richardson.Method(fn, h=1, r=1, digit=8, J=7)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 4.6707743
## [2,] 3.5268145 2.3828547
## [3,] 3.0882445 2.6496745 2.7386145
## [4,] 2.8954802 2.7027158 2.7203962 2.7177936
## [5,] 2.8050259 2.7145715 2.7185234 2.7182559 2.7182867
## [6,] 2.7612009 2.7173759 2.7183107 2.7182803 2.7182820 2.7182818
## [7,] 2.7396294 2.7180580 2.7182854 2.7182817 2.7182818 2.7182818 2.7182818
```

Example 2. Find the **first order derivative** of $f(x) = x * \exp(x)$ at $x = 2$ with $h = 1/2$. The true value is $f'(2) = 2 + 2\exp(2)$.

Solution: To use the Richardson method, we define

$$\phi(h) = \frac{f(2+h) - f(2)}{h}$$

```
fn = function(h) ((2+h)*exp(2+h)-(2)*exp(2))/(h)
Richardson.Method(fn, h=1/2, r=1, digit=8, J=5)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 31.356245
## [2,] 26.277174 21.198102
## [3,] 24.114360 21.951546 22.202694
## [4,] 23.115311 22.116262 22.171167 22.166664
## [5,] 22.635054 22.154798 22.167643 22.167140 22.167171
```

20.2 Romberg Integration

The **Romberg integration** uses the Richardson extrapolation on the composite trapezoidal rule.

20.2.1 Approximation formula and Error Analysis

Recall that, if $f(x)$ is in $C^\infty[a,b]$ (i.e., any given order of derivative of $f(x)$ exists and is continuous), the composite trapezoid rule approximates the integral $\int_a^b f(x)dx$ by

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\right]$$

where $h = (b-a)/n$. n is the number of sub-intervals. The error term given in the previous note can be written as

$$E = \sum_{i=1}^{n} E_i = \sum_{i=1}^{n} \int_{x_i}^{x_{i+1}} [f(x) - S(x)]\, dx = \sum_{i=1}^{n} \frac{f''(c_i)}{2} \times \left[-\frac{(x_{i+1}-x_i)^3}{6}\right] \quad \text{(see the last note)}$$

$$= \sum_{i=1}^{n} \frac{f''(c_i)}{n} \times \left[-\frac{n(x_{i+1}-x_i)^3}{12}\right] = -\frac{nh^3}{12}\sum_{i=1}^{n}\frac{f''(c_i)}{n}.$$

where c_i is a number in subinterval $[x_i, x_{i+1}]$. Clearly for $i = 1, ..., n$,

$$\min_{x\in[a,b]} f''(x) \le f''(c_i) \le \max_{x\in[a,b]} f''(x).$$

Therefore,

$$\min_{x\in[a,b]} f''(x) \le \frac{\sum_{i=1}^{n} f''(c_i)}{n} \le \max_{x\in[a,b]} f''(x).$$

By the **intermediate mean value theorem**, there exists a value in $[a,b]$, say ξ, such that

$$f''(\xi) = \sum_{i=1}^{n} \frac{f''(c_i)}{n}$$

Therefore,

$$E = \sum_{i=1}^{n} E_i = -\frac{nh^3}{12} \sum_{i=1}^{n} \frac{f''(c_i)}{n} = -\frac{nh^3}{12} f''(\xi) = -\frac{(b-a)f''(\xi)}{12} h^2.$$

The complete expression of the numerical integration using the trapezoid rule with n-subintervals is given by

$$T(h) = \int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\right] - \frac{(b-a)f''(\xi)}{12} h^2$$

where $a = x_0 < x_1 < \cdots < x_n = b$. $\{x_1, \cdots, x_n\}$ are equal spaced and $h = (b-a)/n$. c_i is in $[a, b]$.

20.2.2 Romberg Algorithm

From the last formula of the previous sub-section, we see that the trapezoidal rule has an order of approximation $O(h^2)$.

The higher order Richardson's algorithm has the following recursive relationship.

$$\begin{array}{lcll} R(j,0) & := & \phi(h/2^j) & j \geq 0; \\ R(j,k) & := & \frac{2^{rk}R(j,k-1) - R(j-1,k-1)}{2^{rk}-1} & j \geq k > 0. \end{array}$$

Therefore, following the Richardson extrapolation with $r = 2$, we have

$$R(j,0) = T(h/2^j) \quad \text{for } j \geq 0$$

$$R(j,k) = \frac{4^k R(j,k-1) - R(j-1,k-1)}{4^k - 1} \quad \text{for } j \geq k > 0.$$

The corresponding Richardson extrapolation table is given by

$O(h^2)$	$O(h^4)$	$O(h^6)$	$O(h^8)$ Error Order
$R_h(0,0)$			
$R_{\frac{h}{2}}(1,0)$	$R_h(1,1)$		
$R_{\frac{h}{4}}(2,0)$	$R_{\frac{h}{2}}(2,1)$	$R_h(2,2)$	
$R_{\frac{h}{8}}(3,0)$	$R_{\frac{h}{4}}(3,1)$	$R_{\frac{h}{2}}(3,2)$	$R_h(3,3)$
……	……	……	……

20.2.3 R Code

Since this is a direct application of the Richardson extrapolation. We will simply implement it in R.

Goal: approximate the definite integral

$$\int_a^b f(x)dx$$

Partition: partition interval $[a, b]$ with equal width h that gives explicit partition

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n$$

where $ h = x_{i+1} - x_i$ for $i = 0, 1, \cdots, n-1$.

Approximated area formula based on composite trapzoid rule

$$T(h) = \int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_i) + f(b)\right] + O(h^2)$$

$$= \sum_{j=1}^{n-1} f(x_i) + \frac{h[f(a)+f(b)]}{2} + O(h^2).$$

The approximated area is a function of h

$$\phi(h) = \sum_{j=1}^{n-1} f(x_i) + \frac{h[f(a)+f(b)]}{2} \approx \int_a^b f(x)dx.$$

with order $O(h^2)$.

Therefore, in the high order Richardson extrapolation, $r = 2$ should be used. The following code implements the approximation.

```
Romberg.Trapz = function(fn,       # The original function f(x)
                         a,        # lower limit of the given interval
                         b,        # upper limit of the given interval
                         r = 2,    # 2nd order approximation error for the trapezoid
                         dec = 7, # number of digits in the output
                         J         # order of Richardson extrapolation
                         ){
   ## Trapezoid rule
   h = b - a       # initial h
   Trapz.fn = function(a,b,h){
                  X = seq(a,b,h)  # partition of interval [a, b]
                  TP = h*sum(fn(X)) + h*(fn(a)+fn(b))/2
                  TP
```

```
                }
  ## Romberg Int. starts:
  RR = matrix(rep(NA,J^2), ncol=J)
  for (i in 1:J)  RR[i,1] = Trapz.fn(a,b, h/(2^(i-1)))   ## defining R(j,0)
  for (k in 2:J){
      for (j in k:J){
          RR[j,k]=(2^(r*(k-1))*RR[j,k-1]-RR[j-1, k-1])/(2^(r*(k-1))-1)
       }
    }
  options(digits = dec)
  print(RR, na.print = "")
}
```

Example 4: Find the approximation of the integral

$$\int_0^{\pi} \sin(x)dx$$

Solution: We first use the Romberg method.

```
fn = function(x) sin(x)
Romberg.Trapz(fn, a = 0, b = pi, r = 2, dec = 8, J = 5)
```

```
##               [,1]      [,2]      [,3]      [,4] [,5]
## [1,] 5.7708215e-16
## [2,] 1.5707963e+00 2.0943951
## [3,] 1.8961189e+00 2.0045598 1.9985707
## [4,] 1.9742316e+00 2.0002692 1.9999831 2.0000055
## [5,] 1.9935703e+00 2.0000166 1.9999998 2.0000000    2
```

We can also print out the error of approximation in the following (not that the value of the integral is 2)

```
print(2-Romberg.Trapz(fn, a = 0, b = pi, r = 2, dec = 8, J = 6), na.print = "")
```

```
##               [,1]      [,2]      [,3]      [,4] [,5] [,6]
## [1,] 5.7708215e-16
## [2,] 1.5707963e+00 2.0943951
## [3,] 1.8961189e+00 2.0045598 1.9985707
## [4,] 1.9742316e+00 2.0002692 1.9999831 2.0000055
## [5,] 1.9935703e+00 2.0000166 1.9999998 2.0000000    2
## [6,] 1.9983934e+00 2.0000010 2.0000000 2.0000000    2    2
##               [,1]           [,2]          [,3]           [,4]          [,5]           [,6]
## [1,] 2.0000000000
## [2,] 0.4292036732 -9.4395102e-02
## [3,] 0.1038811021 -4.5597550e-03 1.4292682e-03
## [4,] 0.0257683981 -2.6916995e-04 1.6869054e-05 -5.5499797e-06
## [5,] 0.0064296562 -1.6591048e-05 2.4754543e-07 -1.6288042e-08 5.4127094e-09
```

```
## [6,] 0.0016066390 -1.0333694e-06 3.8091554e-09 -5.9674488e-11 3.9661607e-12 -1.3207
```

In the above example, the error for **Romberg approximation** using the composite trapezoidal rule with 16 ($= 2^4$) intervals is about 10^{-9} (level 5).

Next, we use the trapezoid method to approximate the integral and calculate absolute error in the following (the R function is copied from the previous note).

```
Trapezoid.int = function(fun,           # fun = user-defined function
                         xvec,          # interval [a, b]
                         n = 1,         # number of partitions
                         graph=TRUE){   # request graphical representation
  a = min(xvec)
  b = max(xvec)
  m = length(xvec)
  if (n >= m) xvec = seq(a,b,(b-a)/n)
  yvec = fun(xvec)
  nn = length(yvec)
  h = xvec[-1]- xvec[-nn]
  y.adjacent.sum =  yvec[-1] + yvec[-nn]
  Iab = sum(h*y.adjacent.sum)/2
  if(graph == TRUE){
      x.margin = 0.1*abs(b-a)
      tt = seq(a-x.margin, b+x.margin, (b-a+2*x.margin)/10^4)
      lim.x = c(a-x.margin, b+x.margin)
      y.max = max(fun(tt))
      y.min = min(fun(tt))
      y.margin = 0.1*abs(y.max-y.min)
      lim.y = c(y.min - y.margin, y.max + y.margin)
      plot(tt, fun(tt), type="l", col="blue", xlim=lim.x, ylim=lim.y, xlab=" ",ylab="")
      title("Trapezoidal Rule")
      lines(xvec, yvec, type="l", col="red")
      points(xvec, yvec, pch=21, col="red", cex = 0.6)
    }
  Iab
}
```

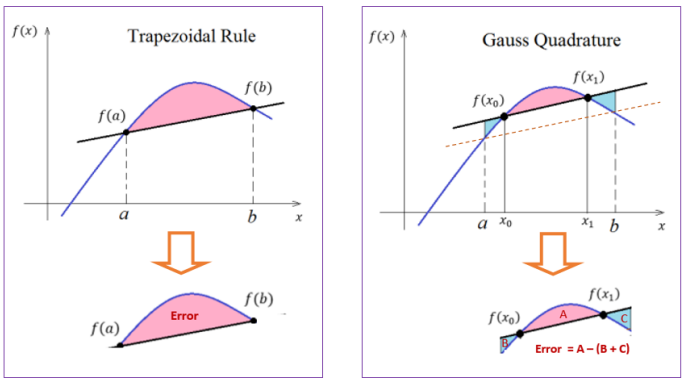
```
fun = function(x) sin(x)
2 - Trapezoid.int(fun=fun, xvec=c(0,pi), n = 15000, graph=FALSE)
```

```
## [1] 7.3108182e-09
```

To obtain the same approximation error, the trapezoid method requires about 15000 intervals. That is, the step size $h = \pi/15000 \approx 0.00021$. In the Romberg approach, the step size is $h = \pi/16 \approx 0.196$. The Romberg approach certainly saves a significant amount of computational time.

20.3 Gaussian Quadrature

The accuracy of numerical integration can be improved by choosing the sampling points wisely (not necessarily to be equally spaced). For example, consider the approximation of $\int_a^b f(x)dx$ in the following figure, the left panel represents the trapezoid approximation with an approximation error being represented as the area of the pink region. We can search two points x_0 and x_1 in the neighborhood a and b respectively such that $A \approx C + B$ (see the right panel in the following figure), we will then get a slightly bigger trapezoid (defined on the same interval $[a, b]$ as that in the regular trapezoid approximation). The resulting approximation error is approximately equal to $A - (C + B)$ which is significantly more accurate than the trapezoid approximation.



Recall that the area under the straight line, generated by the trapezoidal rule, can be expressed as

$$I_T = \frac{b-a}{2}[f(a) + f(b)] = \frac{b-a}{2}f(a) + \frac{b-a}{2}f(b) = \alpha_0 f(a) + \beta_0 f(b)$$

where $\alpha_0 = \beta_0 = (b-a)/2$. The objective of the Gauss quadrature is to fit the straight line, through two points $(x_0, f(x_0))$ and $(x_1, f(x_1))$, such that the area

$$I_G = \alpha f(x_0) + \beta f(x_1)$$

is exactly when the function $f(x)$ being integrated is linear or constant. Next, we derive the Gauss quadrature approximation formula.

20.3.1 Two-point Gauss Legendre Formula

Since there are four parameters $(\alpha, \beta, x_0, x_1)$ in I_G to determine, we need to have four equations based on the assumption that I_G is approximated by its

true integral based on the four basis polynomials $\{1 = x^0, x_1, x^2, x^3\}$. To be more specific,

1. $f(x) = K$ (constant function), $I_G = \alpha K + \beta K = \int_a^b K dx = K(b-a) \Rightarrow \alpha + \beta = b - a$.
2. $f(x) = Kx$: $I_G = \alpha K x_0 + \beta K x_1 = \int_a^b K x dx = K(b^2 - a^2)/2 \Rightarrow \alpha x_0 + \beta x_1 = (b^2 - a^2)/2$.
3. $f(x) = Kx^2$: $I_G = \alpha K x_0^2 + \beta K x_1^2 = \int_a^b K x^2 dx = K(b^3 - a^3)/3 \Rightarrow \alpha x_0^2 + \beta x_1^2 = (b^3 - a^3)/3$.
4. $f(x) = Kx^3$: $I_G = \alpha K x_0^3 + \beta K x_1^3 = \int_a^b K x^3 dx = K(b^4 - a^4)/4 \Rightarrow \alpha x_0^3 + \beta x_1^3 = (b^4 - a^4)/4$.

After some algebra, we can find the solution to the following non-linear equation

$$\begin{array}{lcll} \alpha & + & \beta & = & a - b \\ \alpha x_0 & + & \beta x_1 & = & (a^2 - b^2)/2 \\ \alpha x_0^2 & + & \beta x_1^2 & = & (a^3 - b^3)/3 \\ \alpha x_0^3 & + & \beta x_1^3 & = & (a^4 - b^4)/4 \end{array}$$

that satisfies $x_0 < x_1$ and $a < b$ has the following form

$$\begin{array}{lcl} x_0 & = & (a+b)/2 - \sqrt{3}(b-a)/6 \\ x_1 & = & (a+b)/2 + \sqrt{3}(b-a)/6 \\ \alpha & = & (b-a)/2 \\ \beta & = & (b-a)/2 \end{array}$$

Therefore, the **two-points Gauss-Legendre formula** is given by

$$I_G = \frac{b-a}{2}\left\{ f\left[\frac{a+b}{2} - \frac{\sqrt{3}}{6}(b-a)\right] + f\left[\frac{a+b}{2} + \frac{\sqrt{3}}{6}(b-a)\right]\right\}$$

20.3.2 General Gauss Legendre Formula

Consider the general case of n segments with step size $h = (b-a)/n$. For $I_G^{(k)}$ on $[x_k, x_{k+1}]$, $h = x_{k+1} - x_k$ and $x_k + x_{k+1} = [a + (k-1)h] + [a + kh] = 2a + (2k-1)h$. that is,

$$I_G^{(k)} = \frac{x_{k+1} - x_k}{2}\left\{ f\left[\frac{x_k + x_{k+1}}{2} - \frac{\sqrt{3}}{6}(x_{k+1} - x_k)\right] + f\left[\frac{x_k + x_{k+1}}{2} + \frac{\sqrt{3}}{6}(x_{k+1} - x_k)\right]\right\}$$

$$= \frac{h}{2}\left\{ f\left[\frac{2a+(2k-1)h}{2} - \frac{\sqrt{3}}{6}h\right] + f\left[\frac{2a+(2k-1)h}{2} + \frac{\sqrt{3}}{6}h\right]\right\}$$

$$= \frac{h}{2}\left\{ f\left[a + \left(k - \frac{3+\sqrt{3}}{6}\right)h\right] + f\left[a + \left(k - \frac{3-\sqrt{3}}{6}\right)h\right]\right\}$$

Therefore,

$$I_G = \sum_{k=1}^{n} I_G^{(k)} = \frac{h}{2}\sum_{k=1}^{n}\left\{ f\left[a + \left(k - \frac{3+\sqrt{3}}{6}\right)h\right] + f\left[a + \left(k - \frac{3-\sqrt{3}}{6}\right)h\right]\right\}.$$

20.3.3 R Code

We convert the above formula to an R function in the following.

```
Gauss.Quadrature = function( fn,    # input function
                             a,     # lower limit of the input interval
                             b,     # upper limit of the input interval
                             n      # sub-intervals in the partition
                           ){
    h = (b - a)/n                     # step size
    k = 1:n
    X0 = a + (k -(3+sqrt(3))/6)*h
    X1 = a + (k -(3-sqrt(3))/6)*h
    Ig = 0.5*h*sum(fn(X0)+fn(X1))
    Ig
}
```

```
fn = function(x) sin(sin(x))
Gauss.Quadrature(fn=fn, a = 1, b = 2, n = 10)
```

```
## [1] 0.81644998
```

Example: Consider integral

$$\int_1^3 [x^6 - x^2\sin(2x)]dx \approx 317.20236$$

The Gauss Quadrature approximation with n = 3 gives the following approximated value

```
fn = function(x) x^6 - x^2*sin(2*x)
Gauss.Quadrature(fn=fn, a = 1, b = 3, n = 3)
```

```
## [1] 317.20203
```

20.4 Remarks

We make several remarks to conclude this note.

1. The analysis of the error of Gauss Legendre is out of the scope of this course. We will not discuss error bound or the order of approximation.
2. There are built-in functions in both R and MATLAB that implement Gauss quadrature.