1 Numerical Techniques

"The general who wins a battle makes many calculations in the temple before an attack."
— Sun Tzu, *The Art of War*

“夫未战而庙算胜者，得算多也。”
— 孙子, *《孙子兵法始计篇》*

With more and more practical problems of applied mathematics appearing in different disciplines such as chemistry, biology, geology, management, economics and so on, the demands for numerical computation have increased considerably. Among these problems, frequently they do not have an analytical solution or their exact solution is time-consuming to arrive, therefore the solutions of these problems are dependent on the approximation based on the programmed computation, where the numerical techniques can be employed. In this chapter we start with a brief introduction of the matrix algebra and its computing. Then we move on to the practical problems including the root finding, the numerical integration and differentiation. In every section we illustrate the computing examples under the R context.

1.1 Matrix Algebra

Matrix algebra is a fundamental in numerical methods, therefore at the beginning of this section we introduce the the basic matrix operations and its computation in R. Thereafter we also introduce other operations, such as the inverse including the inverse for non-singular matrix and for singular matrix, the norms containing the vector norms and the matrix norms, the calculation for eigenvalues and eigenvectors and different types of matrix decompositions. We introduce theories and also the accompanied examples computed in R.

1.1.1 Matrix operations

There are four fundamental operations in arithmetic, addition, substraction, multiplication and division. Similar to these fundamental operations in arithmetic, in matrix algebra there exist also the analogous operations, the basic matrix operations such as matrix addition, substraction, multiplication and "division" which can be given with the matrix inverse operation.

```
# set matrices A and B.
> A = matrix(1:9, nrow=3, ncol=3, byrow=T)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> B = matrix(0:-8, nrow=3, ncol=3, byrow=T)
> B
     [,1] [,2] [,3]
[1,]    0   -1   -2
[2,]   -3   -4   -5
[3,]   -6   -7   -8
```

Matrix addition in R can be given with following example.

```
# perform the matrix addition.
> A + B
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Matrix subtraction in R can be given with following example.

```
> A - B # perform the matrix substraction.
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    7    9   11
[3,]   13   15   17
```

R reports an error if one tries to add or subtract matrices with different dimensions. Under R the elementary operations including addition, subtraction, multiplication, and division can also be used with a scalar and a matrix, and is applied on each entry of the matrix. For example the modulo operation:

```
> A %% 2 # perform the scalar multiplication for matrix entries.
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    0    1    0
[3,]    1    0    1
```

If one uses the elementary operations including addition +, subtraction -, multiplication *, and division / between two matrices, they are all interpreted as element operations. Under R one uses the operator \%*\% for the matrix multiplication, such that,

```
# perform the matrix multiplication.
> A %*% ( A + B )
     [,1] [,2] [,3]
[1,]    6    6    6
[2,]   15   15   15
[3,]   24   24   24
```

Different from the upper example for matrix multiplication, in the following example the element multiplication between two variables is given.

```
# perform the multiplication for matrix entries.
> A * (A + B)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

One computes the transposed matrix through the following command:

```
# perform the matrix transpose.
> t(A)
```

In R for computation of the rank of a matrix we can use the package Matrix. An example is given as follows,

```
# calculate the matrix rank.
> A = matrix(1:9, nrow=3, ncol=3, byrow=T)
> rankMatrix(A)
[1] 2
```

The upper example shows that the matrix A has the rank equal to 2 because A consists of only two linear independent column vectors. In R the matrix determinant computation can be given as follows,

```
# calculate the matrix determinant.
> det(A)
[1] 6.661338e-16
```

1.1.2 Inverse

$\mathcal{A}^{-1}$, the inverse of the matrix $\mathcal{A}$ exists if $|\mathcal{A}| \neq 0$ and $\mathcal{A}_{K\times K}$, then

$$\mathcal{A}^{-1}\mathcal{A} = \mathcal{A}\mathcal{A}^{-1} = \mathcal{I}_K. \tag{1.1}$$

Under the upper definition one can determine the inverse of a square matrix by solving the system of linear equations in equation (1.1) by employing the function `solve(A, b)`. In `R` this function can be used to solve a general system of linear equations with a square matrix `A` and a right side `b`. If one doesn't specify the right side `b` of the equation system, the `solve()` function computes the inverse for a square matrix `A`. For example the following codes computes the inverse of the above square matrix $\mathcal{A}$.

```
# set the matrix A as a diagonal matrix
# with each diagonal element equal to 2.
> A = 2*diag(x=1, 3, 3)
> A
     [,1] [,2] [,3]
[1,]    2    0    0
[2,]    0    2    0
[3,]    0    0    2
> solve(A)
     [,1] [,2] [,3]
[1,]  0.5  0.0  0.0
[2,]  0.0  0.5  0.0
[3,]  0.0  0.0  0.5
```

In practice we often confront with the matrix, whose determinant is equal to zero. In this situation the inverse can be given with a generalized form which is called the *Generalized Inverse*, which fulfills the following condition,

$$\mathcal{A}\mathcal{A}^{-1}\mathcal{A} = \mathcal{A}. \tag{1.2}$$

In `R` the generalized inverse of a matrix defined in equation (1.2) can be computed with the function `ginv` in the package `MASS`. Following we show an example for the generalized inverse of a matrix $\mathcal{A} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right]$ satisfying the equation (1.2), such that,

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}. \tag{1.3}$$

Following is an `R` computation example consistent with the upper example. Under the computation with the function `ginv` we can obtain the generalized inverse of the matrix $\mathcal{A} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right]$

equal to $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right]$, which is the same result with what we have shown in equation (1.3).

```
# set the matrix A as a non-inverse matrix
# with the first element equal to 1 and others equal to 0.
> A = matrix(0, 2, 2)
> A[1, 1] = 1
> A
     [,1] [,2]
[1,]    1    0
[2,]    0    0
> ginv(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    0
```

1.1.3 Norm

In this section we categorize the norm it into two parts for introduction, the vector norm and the matrix norm. In the first part we introduce the vector norm, which appears frequently in matrix algebra and numerical computation. In the second part we introduce the matrix norm, which is an extension of the vector norm.

DEFINITION 1.1 *Let V be a vector space over a real or complex field S, the scalar $b \in S$ and $x, y \in V$. Then a norm is a mapping, $f : V \to \mathbb{R}_0^+$, with the following 3 properties:*
(1) $f(bx) = |b|f(x)$,
(2) $f(x + y) \leq f(x) + f(y)$,
(3) $f(x) \geq 0$, where $f(x) = 0$ if and only if $x = 0$.

Let $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$, $p \geq 1$ and $p \in \mathbb{R}$. then a general norm is the L_p norm, which can be represented as follows,

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{1.4}$$

Therefore if $p = 1$ then it becomes the L_1 norm, which is also termed as the Manhattan norm, with the following representation,

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|. \tag{1.5}$$

Similarly if $p = 2$ then it is the Euclidean norm or L_2 norm defined as follows,

$$||x||_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}. \tag{1.6}$$

If $p \to \infty$ then the L_p norm becomes the L_∞, which is termed as the maximum norm such that,

$$||x||_\infty = \max\{|x_i|_{i=1}^n\}. \tag{1.7}$$

In R the functions for vector norm computation are rare as it is not difficult to program for a norm according to the Formula (1.5), (1.6) and (1.7). Therefore here we only give an R example for the Euclidean norm (L_2 norm), which can be realized in R with the function `norm(x, type)`.

```
# input a vector x.
> x = c(2, 1, 2)
# use norm() for computing and set type = c("2") for L2 norm.
> norm(x, type=c("2"))
[1] 3
```

As the L_2 norm for the vector $(2, 1, 2)^\top$ is equal to $\sqrt{2^2 + 1^2 + 2^2} = 3$, hence the both results are same.

DEFINITION 1.2 *Let U be a real or complex field and $U^{m\times n}$ be the set of the real or complex $(m \times n)$ matrices, then $U^{m\times n}$ employed with the matrix addition and the scalar multiplication is built as a vector space. Let $a \in U$ and $\mathcal{A}, \mathcal{B} \in U^{m\times n}$, then a matrix norm is a mapping, $g : U^{m\times n} \to \mathbb{R}_0^+$, with the following 3 properties:*
(1) $g(a\mathcal{A}) = |a|g(x)$,
(2) $g(\mathcal{A} + \mathcal{B}) \leq g(\mathcal{A}) + g(\mathcal{B})$,
(3) $g(\mathcal{A}) \geq 0$, where $g(\mathcal{A}) = 0$ if and only if $\mathcal{A} = 0$.

In R the function `norm()` is used for matrix norm computation under 5 different types, the one norm, the infinity norm, the Frobenius norm, the maximum norm and the spectral norm, whose

representations can be shown as follows,

$$\text{one norm}: \quad ||\mathcal{A}||_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|, \tag{1.8}$$

$$\text{infinity norm}: \quad ||\mathcal{A}||_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|, \tag{1.9}$$

$$\text{Frobenius norm}: \quad ||\mathcal{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}}, \tag{1.10}$$

$$\text{maximum norm}: \quad ||\mathcal{A}||_M = \max_{1\le i\le m, 1\le j\le n} \{|a_{ij}|\}, \tag{1.11}$$

$$\text{spectral norm}: \quad ||\mathcal{A}||_2 = \sqrt{\gamma_{max}(\mathcal{A}^C\mathcal{A})}, \tag{1.12}$$

where $\mathcal{A}^C$ is the conjugate matrix of $\mathcal{A}$. Following we illustrate examples with the `norm()` for a matrix $\mathcal{A} = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$ under the context of the five norms, the one norm, the infinity norm, the Frobenius norm, the maximum norm and the spectral norm.

```
> r1 = c(1, 2)
> r2 = c(3, 4)
# input a matrix A.
> A = cbind(r1, r2)
# compute the one norm.
> norm(A, type=c("O"))
[1] 7
# compute the infinity norm.
> norm(A, type=c("I"))
[1] 6
# compute the Frobenius norm.
> norm(A, type=c("F"))
[1] 5.477226
# compute the maximum norm.
> norm(A, type=c("M"))
[1] 4
# compute the spectral norm.
> norm(A, type=c("2"))
[1] 5.464986
```

1.1.4 Eigenvalues and eigenvectors

For a given basis of a vector space, a $(d \times d)$ matrix can represent a linear function of a d-dimensional vector space in itself. If this function is applied to a vector and maps the vector to a multiple of itself, the vector is called an eigenvector and the multiple is called eigenvalue. In Chapter 8 the theory around eigenvectors, eigenvalues and the later presented spectral decomposition becomes important in order to understand the *principal components analysis* and the *factor analysis*. The definition 1.3 provides a formal description of an eigenvalue and the corresponding eigenvector.

DEFINITION 1.3 *Let V be an any vector space over the real field K and $L : V \to V$ be a linear transformation mapping. Then a non-zero vector $x \in V$ is called an eigenvector, if and only if there exists a scalar $\lambda \in K$ such that $L(x) = \lambda x$.*

The following theorem provides a method of determining the eigenvalues and the corresponding eigenvectors of a matrix.

THEOREM 1.1 *Let V be an any vector space over the real field K and $L : V \to V$ be a linear transformation mapping. Here $L(x) = \mathcal{A}x$, then it holds such that*

$$\lambda \text{ is eigenvalue of } \mathcal{A} \Leftrightarrow det(\mathcal{A} - \lambda\mathcal{I}) = 0. \tag{1.13}$$

If there exists an eigenvector y corresponding to λ with $y \neq 0$, the proof of Theorem 1.1 simply uses the fact that the mapping $(\mathcal{A} - \lambda\mathcal{I})(x)$ is not injective because zero and the eigenvector $y \neq 0$ are all solutions of the function $(\mathcal{A} - \lambda\mathcal{I})(x) = 0$. Thus the associated matrix $\mathcal{A} - \lambda\mathcal{I}$ is not invertible. It follows that the determinant of the associated matrices has to be zero. The determinant $det(\mathcal{A} - \lambda\mathcal{I})$ is called the characteristic polynomial, which can be utilized to compute the eigenvalues of a matrix. In `R` the eigenvalues and eigenvectors of a matrix `A` can be calculated using the function `eigen()`.

```
# construct a matrix A.
> a = c(2, 0, 1)
> b = c(0, 3, 1)
> c = c(0, 6, 2)
> A = rbind(a, b, c)
> A
  [,1] [,2] [,3]
d    2    0    1
e    0    3    1
f    0    6    2
# compute eigenvalues and eigenvectors.
> eigen(A)
$values
[1] 5 2 0
$vectors
           [,1] [,2]       [,3]
[1,] 0.2857143    1 -0.4285714
[2,] 0.4285714    0 -0.2857143
[3,] 0.8571429    0  0.8571429
```

Let x_2 denote the second column of the `$vectors` output, $(1, 0, 0)^\top$. Then it can be seen that the following equation holds for the matrix `A`:

$$\mathtt{A}x_2 = 2x_2. \tag{1.14}$$

That means x_2 is an eigenvector corresponding to the eigenvalue 2 of the matrix `A`.

1.1.5 Spectral decomposition

The spectral decomposition is a method which can represent a matrix $\mathcal{A}$ in term of its eigenvalues and eigenvectors.

DEFINITION 1.4 *Let $\mathcal{A}_{(d\times d)}$ be a matrix defined over a real field. Then $\mathcal{A}$ is diagonalizable if there exists an invertible matrix $\mathcal{P}$ such that $\mathcal{P}\mathcal{A}\mathcal{P}^{-1} = \Lambda$, where Λ is a diagonal matrix, whose diagonal elements are eigenvalues of $\mathcal{A}$, and $\mathcal{P}$ is a matrix with columns from eigenvectors of $\mathcal{A}$.*

THEOREM 1.2 *Let $\mathcal{A}_{(d\times d)}$ be a matrix defined over a real field with eigenvalues $\lambda_1, \ldots, \lambda_d$ and the corresponding eigenvectors $\alpha_1, \ldots, \alpha_d$. Let $\mathcal{P}$ be the eigenvector matrix with $\alpha_1, \ldots, \alpha_d$*

as the corresponding columns and Λ *be the eigenvalue matrix with* $\lambda_1, \ldots, \lambda_d$ *as the corresponding diagonal entries. Then*

$$\mathcal{A} = \mathcal{P}\Lambda\mathcal{P}^{-1}, \tag{1.15}$$

$$\mathcal{A} = \Sigma_{i=1}^{d}\lambda_i\alpha_i\alpha_i^{\top}. \tag{1.16}$$

In R one can use the function `eigen()` for computation of eigenvalues and eigenvectors, where the eigenvalues are sorted in decreasing order within the vector named `values` (see the example above). Before computing the diagonal matrix one must check whether the eigenvectors are linearly independent. Using the output of the function `eigen()`, the linear independence of the eigenvectors can be checked for the above example by computing the rank of the matrix `vectors`.

```
# construct a matrix A.
> a = c(2, 0, 1)
> b = c(0, 3, 1)
> c = c(0, 6, 2)
> A = rbind(a, b, c)
> A
  [,1] [,2] [,3]
d    2    0    1
e    0    3    1
f    0    6    2
# compute eigenvalues and eigenvectors of matrix A.
> r = eigen(A)
# set P as eigenvector matrix.
> P = r$vectors
# check the independence of eigenvectors.
> qr(P)$rank
[1] 3
```

From the following computation it can be seen that the transformation matrix $\mathcal{P}$ has full rank in the above example. The diagonal matrix can be calculated by using the output of the function `eigen()`:

```
# set Lambda as the eigenvalue matrix.
> Lambda
     [,1] [,2] [,3]
[1,]    5    0    0
[2,]    0    2    0
[3,]    0    0    0
# check the numerical solution of A's spectral decomposition.
>  A
  [,1] [,2] [,3]
a    2    0    1
b    0    3    1
c    0    6    2
# reconstruct A with spectral decomposition.
> P %*% Lambda %*% solve(P)
     [,1]          [,2] [,3]
[1,]    2 4.440892e-16    1
[2,]    0 3.000000e+00    1
[3,]    0 6.000000e+00    2
```

From the upper computation in R we see that the reconstruction of $\mathcal{A}$, using the formula $\mathcal{P}\Lambda\mathcal{P}^{-1}$, with its numerical solution of the eigenvalue and eigenvector is different from the matrix $\mathcal{A}$.

1.1.6 Generalized inverse

A generalized inverse, also termed as a pseudo-inverse, A^{+} of a matrix A is a generalization of the inverse matrix. The Moore-Penrose generalized inverse (hereafter, just generalized inverse) is the most common type and was independently defined by [13] and [16]. It is used to compute a 'best fit' solution to a system of linear equations that lacks a unique solution. Another use is to find the minimum (Euclidean) norm solution to a system of linear equations with multiple solutions. The generalized inverse facilitates the statement and proof of results in linear algebra. The generalized inverse is defined and unique for all matrices whose entries are real or complex numbers. It can be computed using the singular value decomposition, see [17].

DEFINITION 1.5 *Let V be a real or complex field and M be the vector space of $(m \times n)$ matrix on V. Then for any $A \in M$ a generalized inverse of A can be defined as a matrix $A^{+} \in M$*

fulfilling the following 4 conditions,

$$AA^+A = A,$$
$$A^+AA^+ = A^+,$$
$$(AA^+)^* = AA^+,$$
$$(A^+A)^* = A^+A,$$

where * is the conjugate transpose.

In R the generalized inverse can be computed with the function `ginv()` in `MASS` package. Following we give an example for a matrix $\mathcal{A} = \left[\begin{smallmatrix} 1 & 2 & 3 \\ 11 & 12 & 13 \end{smallmatrix}\right]$ to compute the generalized inverse.

```
# define a non-square matrix.
> A = matrix(c(1, 2, 3, 11, 12, 13), nrow=2, ncol=3, byrow=TRUE)
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
# find the generalized inverse.
> ginv(A)
             [,1]         [,2]
[1,] -0.63333333  0.13333333
[2,] -0.03333333  0.03333333
[3,]  0.56666667 -0.06666667
```

Then we verify the four conditions listed in the definition of the generalized inverse of $A \in M$. We can see from the following computation that the four conditions are satisfied, although the numerical computation has rounding error.

```
# define a non-square matrix.
> A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
# find the generalized inverse.
> ginv(A)
             [,1]         [,2]
[1,] -0.63333333  0.13333333
[2,] -0.03333333  0.03333333
[3,]  0.56666667 -0.06666667
> Aplus = ginv(A)
# compute the first condition, A %*% Aplus %*% A = A.
```

```
> A %*% Aplus %*% A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
# compute the second condition, Aplus %*% A %*% Aplus = Aplus.
> Aplus %*% A %*% Aplus
            [,1]        [,2]
[1,] -0.63333333  0.13333333
[2,] -0.03333333  0.03333333
[3,]  0.56666667 -0.06666667
# compute the third condition, Conj(t(A %*% Aplus)) = A %*% Aplus.
> Conj(t(A %*% Aplus))
              [,1]          [,2]
[1,]  1.000000e+00 -1.221245e-15
[2,] -1.110223e-16  1.000000e+00
> A %*% Aplus
              [,1]          [,2]
[1,]  1.000000e+00 -1.110223e-16
[2,] -1.221245e-15  1.000000e+00
# compute the fourth condition, Conj(t(Aplus %*% A)) = Aplus %*% A.
> Conj(t(Aplus %*% A))
           [,1]      [,2]       [,3]
[1,]  0.8333333 0.3333333 -0.1666667
[2,]  0.3333333 0.3333333  0.3333333
[3,] -0.1666667 0.3333333  0.8333333
> Aplus %*% A
           [,1]      [,2]       [,3]
[1,]  0.8333333 0.3333333 -0.1666667
[2,]  0.3333333 0.3333333  0.3333333
[3,] -0.1666667 0.3333333  0.8333333
```

1.2 Root finding

1.2.1 Solving system of linear equations

Let K be a real or complex field and $a_{ij}, b_i \in K$ with $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Then the following system of equations is called a system of linear equations:

$$\begin{cases} a_{11}x_1 + \cdots + a_{1n}x_n = b_1, \\ \qquad\vdots \qquad\qquad\quad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n = b_m. \end{cases} \tag{1.17}$$

For a matrix $\mathcal{A} = (a_{ij}) \in K^{m\times n}$ and two vectors $x = (x_1, \ldots, x_n)^\top$ and $b = (b_1, \ldots, b_m)^\top$, the system of linear equations can be rewritten as the matrix representation:

$$\mathcal{A}x = b. \tag{1.18}$$

Let $\mathcal{A}_e$ be the extended matrix, where the $(n+1)$th column is the vector b. Then the system (1.18) can be solved if and only if the rank of $\mathcal{A}$ is the same as the rank of $\mathcal{A}_e$. Because then the vector b can be represented by a linear combination of the columns of $\mathcal{A}$. If the system (1.18) can be solved and the rank of $\mathcal{A}$ equals n, then there exists a unique solution. Otherwise (1.18) can have no solution or infinitely many solutions [8].

A frequently used approach is the Gaussian algorithm. The goal of the Gaussian algorithm is to transform the system of equations by elementary transformations in an upper triangular form. Then the solution can be computed by back substitution. The gaussian algorithm provides a matrix decomposition of $\mathcal{A}$, the so called $\mathcal{LU}$ decomposition (see Braun and Murdoch [3] for further details), where $\mathcal{L}$ is a lower triangular matrix and $\mathcal{U}$ is an upper triangular matrix with the following form:

$$\mathcal{L} = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ l_{21} & 1 & \ldots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n1} & l_{n2} & \ldots & 1 \end{pmatrix}, \qquad \mathcal{U} = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & u_{nn} \end{pmatrix}.$$

Then (1.18) can be rewritten as:

$$\mathcal{A}x = \mathcal{LU}x = b. \tag{1.19}$$

Now the system in (1.18) can be easily solved in two steps. First define $\mathcal{U}x = y$ and solve $\mathcal{L}y = b$ for y by forward substitution. Then solve $\mathcal{U}x = y$ for x by back substitution. In `R` the function `solve(A,b)` uses the $\mathcal{LU}$ decomposition to solve a system of linear equations with a matrix `A` and right side `b`. Another method that can be used in `R` to solve a system of linear equations is

the $\mathcal{QR}$ decomposition, where the matrix $\mathcal{A}$ is decomposed in an orthogonal matrix $\mathcal{Q}$ and an upper triangular matrix $\mathcal{R}$ (see Braun and Murdoch [3]). One uses the function `qr.solve()` to compute the solution of a system of linear equations using the $\mathcal{QR}$ decomposition. In contrast to the $\mathcal{LU}$ decomposition this method can be applied even if $\mathcal{A}$ is not square. Next we show an example of solving a system of linear equations in `R` with the function `solve()`.

EXAMPLE 1.1 *Solve the following system of linear equations in* `R` *with the Gaussian algorithm and the back-substitution,*

$$\begin{cases} 2x_1 - \frac{1}{2}x_2 - \frac{1}{2}x_3 = 0, \\ -\frac{1}{2}x_1 + 2x_3 - \frac{1}{2}x_4 = 3, \\ -\frac{1}{2}x_1 + 2x_2 - \frac{1}{2}x_4 = 3, \\ -\frac{1}{2}x_2 - \frac{1}{2}x_3 + 2x_4 = 0. \end{cases} \tag{1.20}$$

Here we firstly solve the upper system of linear equations by hand then we show the computation of this example in `R` for verification. This system of linear equations is not difficult to be solved with the Gaussian algorithm such that

$$\begin{cases} 2x_1 - \frac{1}{2}x_2 - \frac{1}{2}x_3 = 0, \\ \frac{15}{8}x_2 - \frac{1}{8}x_3 - \frac{1}{2}x_4 = 3, \\ \frac{28}{15}x_3 - \frac{8}{15}x_4 = \frac{16}{5}, \\ \frac{12}{7}x_4 = \frac{12}{7}. \end{cases} \tag{1.21}$$

Therefore we can use back substitution to obtain the final result that $(x_1, x_2, x_3, x_4)^\top = (1, 2, 2, 1)^\top$. Then we show the computation of solving this system of linear equation in `R`. In following computation, we need to input two parameters, the coefficient matrix `A` and the right hand side of the system of equations, `b`. Then we input `A` and `b` into the function `solve()` to obtain the final result.

```
# construct the coefficient matrix A.
> a1 = c(2, -1/2, -1/2, 0)
> a2 = c(-1/2, 0, 2, -1/2)
> a3 = c(-1/2, 2, 0, -1/2)
> a4 = c(0, -1/2, -1/2, 2)
> A = rbind(a1, a2, a3, a4)
> A
    [,1] [,2] [,3] [,4]
a1   2.0 -0.5 -0.5  0.0
a2  -0.5  0.0  2.0 -0.5
a3  -0.5  2.0  0.0 -0.5
a4   0.0 -0.5 -0.5  2.0
# specify the right hand side of the system.
> b = c(0, 3, 3, 0)
# calculate Ax=b with R solver.
> solve(A, b)
[1] 1 2 2 1
```

1.2.2 Solving system of nonlinear equations

A system of nonlinear equations is represented by a function $F : \mathbb{R}^n \to \mathbb{R}^n$ being $F = (f_1, \ldots, f_n)$. Therefore the system has the following form:

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0, \\ \quad \vdots \qquad\quad \vdots \\ f_n(x_1, \ldots, x_n) = 0. \end{cases} \tag{1.22}$$

There are many different numerical methods to solve a system of nonlinear equations. In general it can be differentiated between gradient and non-gradient methods. In the following the famous Newton method or also called Newton-Raphson method is presented. To get a better illustration of the idea behind the Newton-method, let us consider a continuous differentiable function $F : \mathbb{R} \to \mathbb{R}$, where one tries to find x_* with $F(x_*) = 0$ and $\frac{\partial F(x)}{\partial x}|_{x=x_*} \neq 0$. Start choosing a value $x_0 \in \mathbb{R}$ and define the following tangent:

$$p(x) = F(x_0) + \frac{\partial F(x)}{\partial x}|_{x=x_0}(x - x_0). \tag{1.23}$$

Then the tangent $p(x)$ is a good approximation for F in a sufficient small neighborhood of x_0. If $\frac{\partial F(x)}{\partial x}|_{x=x_0} \neq 0$, the root x_1 of the tangent in (1.23) can be computed as follows:

$$x_1 = x_0 - \frac{F(x_0)}{\frac{\partial F(x)}{\partial x}|_{x=x_0}}. \tag{1.24}$$

With the new value x_1 the rule can be applied again. This procedure can be applied iteratively and under certain theoretical conditions the solution should converge to the actual root. The Figure 1.1 demonstrates the Newton method for $f(x) = x^2 - 4$ with the starting value $x_0 = 10$.

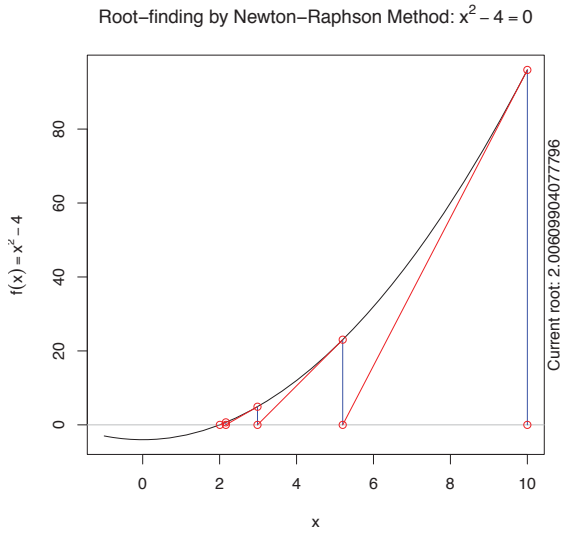


Figure 1.1: Newton's method. QBSCnewton.R

The picture was computed using the function `newton.method(f, init,...)` from the package `animation`, where `f` is a function and `init` a starting value. The function provides an illustration of the iterations in Newton's method (see `help(newton.method)` for further details).

The function `uniroot()` searches an interval for a root, i.e. value of zero, of a function and returns only one root, even if multiple roots exist within in the interval.

```
# set the function for which we want to find the roots:
> f = function(x){-x^4-cos(x)+9*x^2-x-5}
# plot the function to see the roots:
> curve(f(x), -3, 3, lwd=2,
+ main=expression(f(x)==-x^4-cos(x)+9*x^2-x-5))
> abline(h=0, lty=2, lwd=1)
> points(c(2.855595, .8913973, -.7673259, -2.980561),
```

```
+ c(0,0,0,0), pch=19, cex=1.5, col="blue")
# find the root between 0 and 2.
> str(uniroot(f, c(0, 2)))
List of 4
 $ root      : num 0.891
 $ f.root    : num -0.000311
 $ iter      : int 6
 $ estim.prec: num 6.1e-05
# Only returns one root, even if there are multiple roots in the
# interval:
> str(uniroot(f, c(-3, 2)))
List of 4
 $ root      : num -2.98
 $ f.root    : num -1.03e-06
 $ iter      : int 4
 $ estim.prec: num 0.000117
# The values of f at the end points must be of opposite sign:
# i.e. for x in [a, b], then f(a)*f(b) < 0
# so that the algorithm can work.
> str(uniroot(f, c(0, 3)))
Error in uniroot(f, c(0, 3)) :
f() values at end points not of opposite sign
```

For a real or complex polynomial of the form $p(x) = z_1 + z_2 \cdot x + \ldots + z_n \cdot x^{n-1}$ the function `polyroot(z)` with z being the vector of coefficients computes the root. The algorithm does not guarantee to find all the roots of the polynomial.

```
> z = c(0.2567, 0.1570, 0.0821, -0.3357, 1)
# compute the 4 complex roots of the polynomial.
> polyroot(z)
[1]  0.5871611+0.5963709i -0.4193111+0.4366629i
-0.4193111-0.4366629i 0.5871611-0.5963709i
```

1.2.3 Maximization and Minimization of Functions

Maximization and minimization of functions are also termed as the optimization problem, which contains 2 components such as an objective function $f(m)$ and constrains. A optimization problem can be classified into 2 categories according to the existence of constraints that if there are constraints affiliated to the objective function then it is termed as a constrained optimization otherwise a non-constrained optimization. In this section we introduce first 4

different non-constrained optimization techniques including the golden ratio search method, the Nelder-Mead method, the BFGS method and the conjugate gradient method. Afterwards 2 constrained optimization approaches, linear programming (LP) and non-linear programming (NLP) will be introduced.

At the beginning let us define the local extrema and the global extrema, which will be frequently utilized later.

DEFINITION 1.6 *A real function f defined on a domain M has a global maximum point at m_{opt} if $f(m_{opt}) \geq f(m)$ for all m in M, then $f(m_{opt})$, the evaluation of the function at the maximum point m_{opt} is called the maximum value of the function. Analogously, the function has a global minimum point at m_{opt} if $f(m_{opt}) \leq f(m)$ for all m in M, then $f(m_{opt})$, the evaluation of the function at the minimum point m_{opt}, is called the minimum value of the function.*

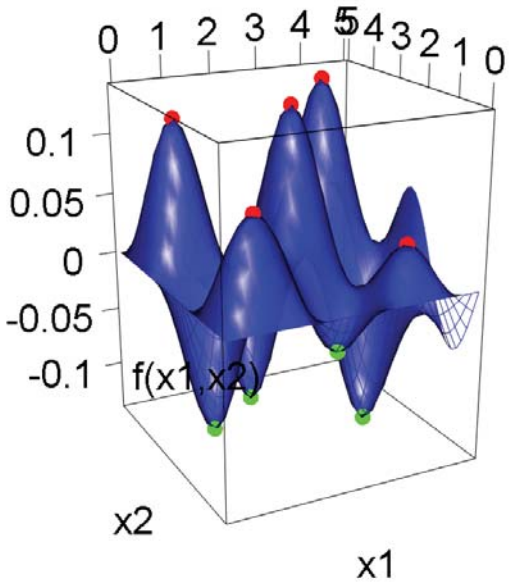
DEFINITION 1.7 *If the domain M is a metric space then f is said to have a local maximum point at the point m_{opt} if there exists some neighborhood $\epsilon > 0$ such that $f(m_{opt}) \geq f(m)$ for all m in M within distance ϵ centering on m_{opt}. Analogously, the function has a local minimum point at m_{opt} if $f(m_{opt}) \leq f(m)$ for all m in M within distance ϵ centering on m_{opt}.*

EXAMPLE 1.2 *The following function possesses local maxima, local minima, global maxima and global minima. All global maxima and local maxima are pointed with the red color, and all global minima and local minima are pointed with the green color. It can be seen in Figure 1.2 (a) that the Formula (1.25) has local maxima at points $(0.77, 3.98)$, $(0.87, 0.73)$, $(3.83, 0.61)$, global maxima at points $(0.77, 2.31)$, $(2.45, 0.61)$, local minima at points $(2.34, 2.22)$, $(3.95, 4.06)$ and global minima at points $(2.34, 4.06)$, $(3.95, 2.22)$.*

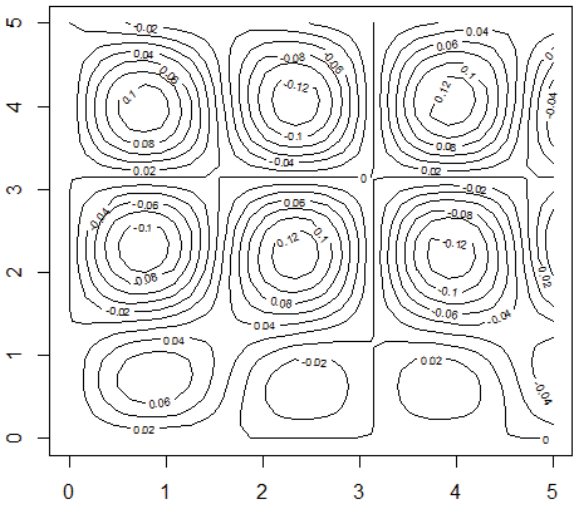
$$\begin{aligned} f(x_1, x_2) &= 0.03\sin(x_1)\sin(x_2) - 0.05\sin(2x_1)\sin(x_2) \\ &\quad +0.01\sin(x_1)\sin(2x_2) + 0.09\sin(2x_1)\sin(2x_2). \end{aligned} \tag{1.25}$$

Golden Ratio Search Method

The golden ratio search method was proposed by Jack Carl Kiefer *cf.* Kiefer [12]. This method is frequently employed for solving optimization with 1-dimensional uni-modal objective function, which belongs to a non-gradient method. The main algorithm for minimization version of this method is given as follows.



(a)



(b)

Figure 1.2: Plots for the multi-modal function in Formula (1.25). (a) A 3d plot for Formula (1.25). (b) A contour plot for Formula (1.25).
BSCmultimodal.R BSCcontour.R

Algorithm of the Golden Ratio Search Method

SET $r = (\sqrt{5}-1)/2$
WHILE $|f(x_U) - f(x_L)| \geq Tolerance$
  $d = (x_U - x_L)r$
  $x_1 = x_L + d$
  $x_2 = x_U - d$
  IF $f(x_1) > f(x_2)$ **THEN**
    $x_U = x_2$
  ELSE
    $x_L = x_1$
  END IF
END WHILE
IF $f(x_1) < f(x_2)$ **THEN**
  $x_{minimum} = x_1$
ELSE IF $f(x_1) \geq f(x_2)$ **THEN**
  $x_{minimum} = x_2$
END IF
RETURN $x_{minimum}$

Following is the explanation of the above algorithm,

1. Initialization: set the so called "Golden Ratio" $r = (\sqrt{5}-1)/2$ and the initial step-width $d = (x_U - x_L)r$, and then choose 2 intermediate points x_1 and x_2 from the interval $[x_L, x_U]$, and set $x_1 = x_L + d$ and $x_2 = x_U - d$.

2. Termination: set the termination conditions for algorithm that if either $|f(x_U) - f(x_L)|$ or $|x_U - x_L|$ is sufficiently smaller than the accuracy tolerance setting at the beginning then the algorithm stops and produces output that if $f(x_1) < f(x_2)$ then $x_{minimum} = x_1$, otherwise if $f(x_1) \geq f(x_2)$ then $x_{minimum} = x_2$.

3. Iteration: if the algorithm is not stopped in the second step by the accuracy tolerance then if $f(x_1) > f(x_2)$ we update the upper bound $x_U = x_2$, otherwise update the lower bound $x_L = x_1$.

4. Loop: iterate from the second step.

EXAMPLE 1.3 *Use R function* `optimize` *from the package* `stat` *to perform the golden ratio search for the following optimization.*

$$\max f(x) = -(x-3)^2 + 10. \tag{1.26}$$

```
# specify the function.
> f = function(x){-(x-3)^2+10}
# use optimize() as an R solver.
> optimize(f, c(-10,10), tol=0.0001, maximum=T)
$maximum
[1] 3

$objective
[1] 10
```

In the R code part we have set the tolerance parameter of the stop, `tol = 0.0001`, of the algorithm as $1/10000$ and set the parameter `maximum = T` for a maximization operation. It is trivial that the function (1.26) can be analytically solved with the first order derivative condition that the function can arrive at the global maximum point $x = 3$ and the maximum value of the function such that $f(3) = 10$. And the output from the upper R computation shows that for optimization problem (1.26) it has the maximum point 3 and maximum value of function equals 10, what is identical to the analytical solution.

Nelder-Mead Method

This method was proposed by John Nelder and Roger Mead *cf.* Nelder and Mead [15] and is applied frequently in multivariate non-constrained optimization problems, which is also a direct method without integrating gradient into the computation. The main idea in minimization version of the Nelder-Mead method is briefly concluded as follows and a graph for this method in 2-dimensional input case is shown in (Figure

Algorithm of the Nelder-Mead Method

SET $f(x_1) < f(x_2) < f(x_3),\ x_1, x_2, x_3 \in \mathbb{R}^n$
WHILE $\|f(x_i) - f(x_j)\|_{L2} \geq Tolerance,\ for\ i \neq j,\ i,\ j \in \{1,\ 2,\ 3\}$
  $z = 1/2(x_1 + x_2)$
  $d = 2z - x_3$
  IF $f(x_1) < f(d) < f(x_2)$ **THEN**
    $x_3 = d$
  ELSE IF $f(d) \leq f(x_1)$ **THEN**
    $k = 2d - z$
    IF $f(k) < x_1$ **THEN**
      $x_3 = k$
    ELSE

$x_3 = d$
END IF
ELSE IF $f(x_3) > f(d) \geq f(x_2)$ **THEN**
$x_3 = d$
ELSE IF $f(x_3) \leq f(d)$ **THEN**
$t = \{t | f(t) = min(f(t_1), f(t_2))\}$
IF $f(t) < f(x_3)$ **THEN**
$x_3 = t$
ELSE
$x_3 = s = (x_1 + x_3)/2$
$d = 2z - x_3$
END IF
END IF
END WHILE
RETURN $x_{minimum} = x_1$

Following we give the explanation of this algorithm,

1. Initialization: choose 3 initial guesses, such that x_1, x_2, x_3, and sort the evaluation of the function such that $f(x_1) < f(x_2) < f(x_3)$, where x_1, x_2, x_3 can be vectors.

2. Termination: set the stop conditions for the algorithm that if $\|x_i - x_j\| < \epsilon_x,\ for\ i \neq j,\ i,\ j \in \{1,\ 2,\ 3\}$ or $\|f(x_i) - f(x_j)\| < \epsilon_f,\ for\ i \neq j,\ i,\ j \in \{1,\ 2,\ 3\}$ holds, where $\epsilon_x,\ \epsilon_f$ are the accuracy tolerance of variables and function values respectively and $\|\cdot\|$ is $L2$ measure, then iteration stops.

3. Iteration: find the midpoint at the opposite site of point x_3, on the segment of x_1x_2, and set it as point $z = 1/2(x_1 + x_2)$, then linearize point x_3 though the point z and extend the segment x_3z to the point $d = 2z - x_3$.

   a) If $f(x_1) < f(d) < f(x_2)$ then we replace $x_3 = d$, otherwise, if $f(d) \leq f(x_1)$ then extend the segment x_3d to $k = 2d - z$, and if $f(k) < x_1$ then $x_3 = k$ else $x_3 = d$.

   b) If $f(x_3) > f(d) \geq f(x_2)$ then $x_3 = d$, otherwise, if $f(x_3) \leq f(d)$ then find $t = \{t | f(t) = min(f(t_1), f(t_2))\}$, where if $f(t) < f(x_3)$ then $x_3 = t$ else do shrinking that $x_3 = s = 1/2(x_1 + x_3)$ and $x_2 = z$.

4. Loop: iterate from the second step.

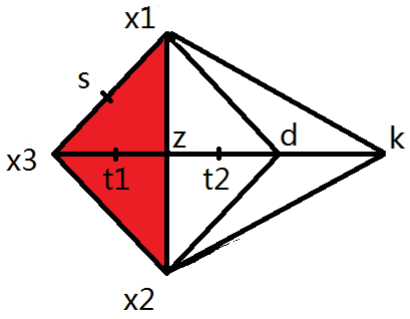
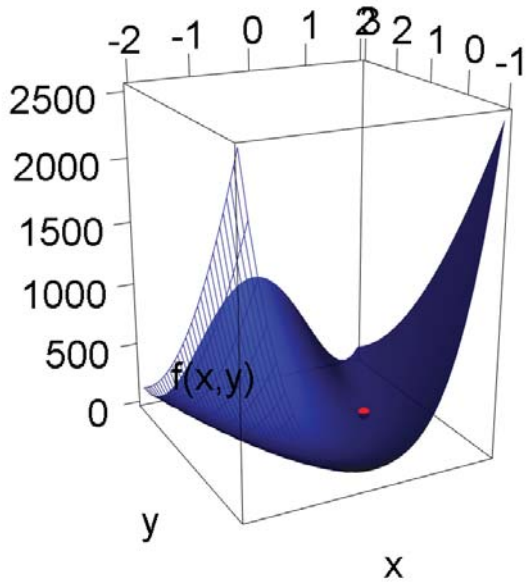


Figure 1.3: Algorithm Graph for Nelder-Mead Method

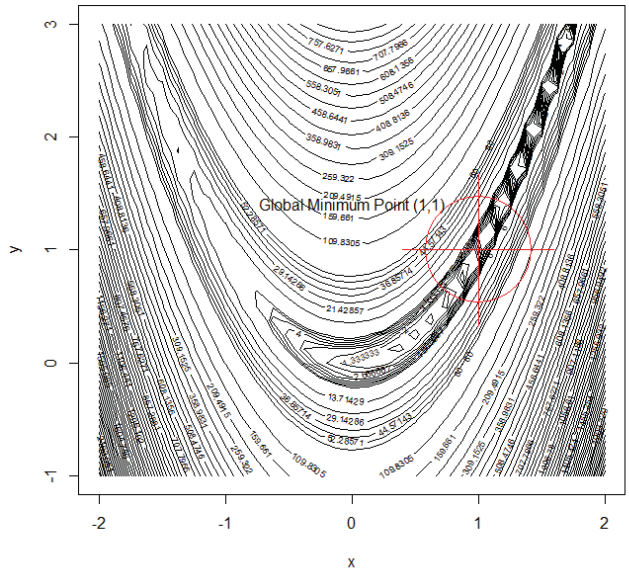
EXAMPLE 1.4 *For Nelder-Mead method we employ the function* `fminsearch` *from the* `neldermead` *package developed by Sebastien Bihorel cf. Bihorel [2]. The function to be minimized is Rosenbrock function, which has an analytical solution with global minimum point* $(1, 1)$ *and global minimum value of the function* $f(1, 1) = 0$*, expressed as follows,*

$$\min f(x, y) = 100(y - x^2)^2 + (1 - x)^2. \tag{1.27}$$

```
# specify the function.
> fct = function(x){
+ y=100*(x[2]-x[1]^2)^2+(1-x[1])^2
+ }
# do optimization with an R solver fminsearch().
> answer = fminsearch(fct, c(-1.2, 1))
> answer
$x
              [,1]           [,2]
[1,] 1.00002202178 1.00004221975

$fval
[1] 8.17766110646e-10

$exitflag
[1] 1

$output
$output$algorithm
[1] "Nelder-Mead simplex direct search"
```



(a)



(b)

Figure 1.4: Plots for the Rosenbrock function in Formula (1.27). (a) A 3d plot for Formula (1.27). (b) A contour plot for Formula (1.27).
BSCrosenbrock.R BSCrbcontour.R

```
$output$funcCount
[1] 159

$output$iterations
[1] 85

$output$message
[1] "Optimization terminated:\n the current x satisfies the
termination criteria using OPTIONS.TolX of 1.000000e-04\n
and F(X) satisfies the convergence criteria using
OPTIONS.TolFun of 1.000000e-04\n"
```

The upper computation illustrates that the optimization problem (1.27) has a local minimum point at $x = 1.000022, y = 1.000042$ and the minimum value of the function is $f(x, y) = 8.1777e - 10$, which can be compared with the exact analytical global minimum point $(1, 1)$ and the exact analytical global minimum value of the function $f(1, 1) = 0$.

BFGS Method

This method is frequently used in the multivariate optimization problem without constraints. The name of BFGS is an abbreviation of four authors, who independently constructed this method in 1970, standing for Broyden, Fletcher, Goldfarb and Shanno, *cf.* Broyden [4], Fletcher [6], Goldfarb [7] and Shanno [18]. The main idea of this method originated from the Newton's method, where the second-order Taylor expansion for a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ at $x = x_j \in \mathbb{R}^n$ is employed, such that

$$f(x) = f(x_i) + \nabla f^\top(x_i)p + \frac{1}{2}p^\top \mathcal{B}_i p,$$

where $p = x - x_i$, and $\nabla f(x_i)$ is the value of partial derivative of function f at point x_i, and $\mathcal{B}_i$ is the Hessian matrix. According to the first-order condition we obtain,

$$\nabla f = \nabla f(x_i) + \mathcal{B}_i p = 0,$$

hence if $\mathcal{B}_i$ is invertible then,

$$\begin{aligned} p &= -\mathcal{B}_i^{-1}\nabla f(x_i), \\ x - x_i &= -\mathcal{B}_i^{-1}\nabla f(x_i), \\ x_{i+1} &= x_i - \mathcal{B}_i^{-1}\nabla f(x_i). \end{aligned}$$

And the last function will converge quadratically to the optimum. But the problem is that Newton's method requires the computation of the exact Hessian at each iteration, which is

computationally expensive. Therefore the BFGS method overcomes this disadvantage with an approximation of Hessian's inverse by solving an optimization as follows,

$$\arg\min_{\mathcal{B}} \|\mathcal{B}^{-1} - \mathcal{B}_i^{-1}\|_{\mathcal{W}},$$

subject to:

$$\begin{aligned} \mathcal{B}^{-1} &= (\mathcal{B}^{-1})^\top, \\ \mathcal{B}^{-1}(\nabla f_i - \nabla f_{i-1}) &= x_i - x_{i-1}. \end{aligned}$$

where $\|\cdot\|_{\mathcal{W}}$ is called the weighted Frobenius norm and $\mathcal{W}$ is a matrix and both are expressed respectively as follows,

$$\begin{aligned} \|\mathcal{B}^{-1} - \mathcal{B}_i^{-1}\|_{\mathcal{W}} &= \|\mathcal{W}^{\frac{1}{2}}(\mathcal{B}^{-1} - \mathcal{B}_i^{-1})\mathcal{W}^{\frac{1}{2}}\|, \\ \mathcal{W}(\nabla f_i - \nabla f_{i-1}) &= x_i - x_{i-1}. \end{aligned}$$

And (1.28) has a unique solution such that

$$\begin{aligned} \mathcal{B}_i^{-1} &= \mathcal{M}_1\mathcal{B}_{i-1}\mathcal{M}_2 + (x_i - x_{i-1})\gamma_{i-1}(x_i - x_{i-1})^\top, \\ \mathcal{M}_1 &= \mathcal{I} - \gamma_{i-1}(x_i - x_{i-1})(\nabla f_i - \nabla f_{i-1})^\top, \\ \mathcal{M}_2 &= \mathcal{I} - \gamma_{i-1}(\nabla f_i - \nabla f_{i-1})(x_i - x_{i-1})^\top, \\ \gamma_i &= ((\nabla f_i - \nabla f_{i-1})^\top(x_i - x_{i-1}))^{-1}. \end{aligned}$$

EXAMPLE 1.5 *For the BFGS method we still use the Rosenbrock function (1.27) and we employ package `optimx` developed by John C. Nash in 2011 (see Nash and Varadhan [14]).*

$$\min f(x, y) = 100(y - x^2)^2 + (1 - x)^2.$$

```
# specify the function.
> fct = function(x){
+ y=100*(x[2]-x[1]^2)^2+(1-x[1])^2
+ }
# perform the optimization with an R solver optimx().
> answer = optimx(fn=fct, par=c(-1.2, 1), method=c("BFGS"))
> answer
                  p1             p2             value fevals gevals
BFGS 0.999804433231 0.999608380623 3.82738275896e-08    127     38
```

The upper computation illustrates that under the method of BFGS the minimum value of the function (1.27) is $3.8274e-08$ at the minimum point $(0.9998, 0.9996)$ and the outputs `fevals = 127, gevals = 38` show correspondingly the calls of the objective function and the calls of the gradients, whose outputs are close to the exact solution $(x, y, f(x, y)) = (0, 0, 1)$.

Conjugate Gradient Method

The conjugate gradient method is proposed by Hestenes and Stiefel in 1952 *cf.* Hestenes and Stiefel [10], which is widely used for solving symmetric positive definite linear systems, which often appear in a multivariate unconstrained optimization problem, that is

$$\mathcal{A}x = b,\ \mathcal{A} \in \mathbb{R}^{n\times n},\ \mathcal{A} = \mathcal{A}^{\top},\ \text{and } \mathcal{A} \text{ positive definite.}$$

The main idea behind CG is to utilize iterations stepwise to approach the optimum of the linear system and the algorithm is as follows.

Algorithm of the Conjugate Gradient Method

SET

$r_0 = b - \mathcal{A}x_0$

$p_0 = r_0$

$p_i,\ i \in \{1, \ldots, n\}$ the basis vectors in $\mathbb{R}^n$

WHILE $r_i - \alpha_i \mathcal{A} p_i \geq Tolerance$

$\alpha_i = \frac{r_i^{\top} r_i}{p_i^{\top} \mathcal{A} p_i}$

$r_{i+1} = r_i - \alpha_i \mathcal{A} p_i$

$\beta_i = \frac{r_{i+1}^{\top} r_{i+1}}{r_i^{\top} r_i}$

$p_{i+1} = r_{i+1} + \beta_i p_i$

$x_{i+1} = x_i + \alpha_i p_i$

END WHILE

RETURN x$_{minimum} = x_{i+1}$

Following we give the explanation of this algorithm,

1. Initialization: set an initial guess,

$$\begin{aligned} r_0 &= b - \mathcal{A}x_0, \\ p_0 &= r_0, \end{aligned}$$

where r_0 is the initial residual and $p_i,\ i \in \{1, \ldots, n\}$ is the basis vectors in $\mathbb{R}^n$.

2. Termination: set the stop condition that if the residual r smaller than the accuracy tolerance setting then the loop terminates.

3. Iteration: iterate employing the following equations,

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i p_i, \\ \text{where} & \\ \alpha_i &= \frac{r_i^\top r_i}{p_i^\top \mathcal{A} p_i}, \\ r_{i+1} &= r_i - \alpha_i \mathcal{A} p_i, \\ p_{i+1} &= r_{i+1} + \beta_i p_i, \\ \beta_i &= \frac{r_{i+1}^\top r_{i+1}}{r_i^\top r_i}. \end{aligned}$$

4. Loop: iterate from the second step.

EXAMPLE 1.6 *For the CG method we also use the Rosenbrock function (1.27) with the package* `optimx`*, for which we use the point* `c(1.2 ,1)` *as an initial guess inputting in the function* `optimx` *and* `c("CG")` *as the chosen optimization method.*

```
# specify the function.
fct = function (x){
+ y=100*(x[2]-x[1]^2)^2+(1-x[1])^2
+ }
# perform the optimization with an R solver optimx().
> answer = optimx(fn=fct, par=c(1.2, 1), method=c("CG"))
> answer
          p1        p2         value fevals gevals
CG 1.030077 1.061209 0.0009036108    403    101
```

The upper computation illustrates that under the method of CG the minimum value of the function (1.27) is 0.0009 at the minimum point of $(1.0301, 1.0612)$.

Constrained Optimization

The constrained optimization can be categorized into 2 parts with respect to the linearity of the objective function and the constraints, that is if both objective function and constraints are linear function then it is termed as a linear programming problem, otherwise a non-linear programming problem.

Formally, linear programming (LP) is an approach for the optimization problem with a linear objective function, under constraints of linear equality and linear in-equality. It has feasible region identical to a convex polyhedron, which is composed of a set made by the intersection of finitely many half spaces, each of which comes from a linear inequality. The objective of linear

programming is to find a point in the polyhedron, where the objective function can reach the minimum (or maximum). A representative LP can be expressed as follows:

$$\begin{aligned} &\arg\max_x a^\top x, \\ \text{subject to: } &\mathcal{C}x \leq b, \\ &x \geq 0, \end{aligned}$$

where $x \in \mathbb{R}^n$ is a vector of variables to be identified, a and b are vectors of known coefficients, and $\mathcal{C}$ is a known matrix of coefficients in constraints. The expression $a^\top x$ to be optimized is the objective function. The inequalities $\mathcal{C}x \leq b$ and $x \geq 0$ are the constraints, under which the objective function will be optimized.

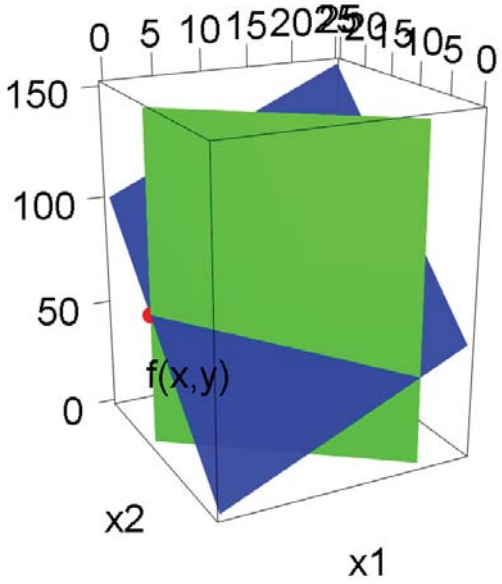
Non-linear programming (NLP) has an analogous definition compared to LP problem, and the differences between NLP an LP are that both objective function and constraints in NLP can be non-linear functions. For LP problem here we give an example as follows,

EXAMPLE 1.7 *Solve the following linear programming optimization problem with R.*

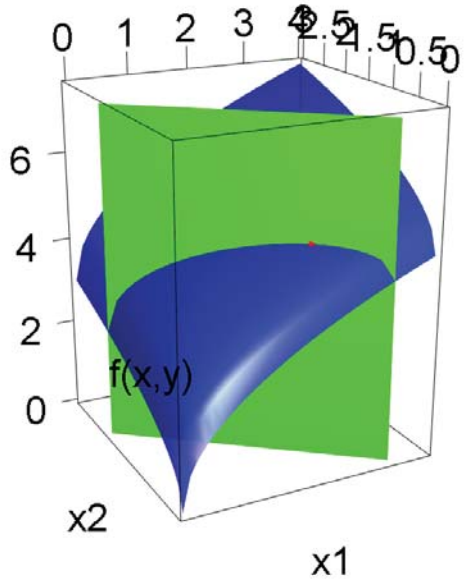
$$\begin{aligned} \max y &= 2x_1 + 4x_2, \\ \textit{subject to:} & \\ 3x_1 + 4x_2 &\leq 60, \\ x_1 &\geq 0, \\ x_2 &\geq 0. \end{aligned} \tag{1.28}$$

And for the example of (1.28) we use the function from the package `Rglpk` *cf.* Theussl [20] for computation and R code is shown as follows.

```
# specify the objective function.
> fct = c(2, 4)
> st = matrix(c(3, 4), nrow=1)
> leq = c("<=")
> rhs = c(60)
# perform the optimization with an R solver Rglpk_solve_LP().
> Rglpk_solve_LP(fct, st, leq, rhs, max=TRUE)
$optimum
[1] 60

$solution
[1]  0 15

$status
[1] 0
```



(a)



(b)

Figure 1.5: Plots for linear programming and nonlinear programming problems, where the blue plane is the illustration of the objective function, the green plane is the illustration of the linear constraint. The red point is the maximum.. (a) Linear programming problem. (b) Nonlinear programming problem.
BSClp.R BSCnlp.R

The output of the upper computation illustrates that under the method of LP the maximum value of the function (1.28) arrives at 60 at the maximum point of $(0, 15)$.

For NLP problem here we give an example as follows,

EXAMPLE 1.8 *Solve the following non-linear optimization problem with R,*

$$\begin{aligned} \max y &= \sqrt{5x_1} + \sqrt{3x_2}, \\ \textit{subject to:} & \\ 3x_1 + 5x_2 &\leq 10. \end{aligned} \tag{1.29}$$

And for this example we use package `stat` for computation and the output is shown as follows.

```
# specify the objective function.
> fct = function(x){sqrt(5*x[1])+sqrt(3*x[2])}
> A = matrix(c(-3, -5), nrow=1, ncol=2, byrow=TRUE)
> b = c(-10)
# perform the optimization with an R solver constrOptim().
> constrOptim(c(1, 1), fct, NULL, ui=A, ci=b,
+ control=list(fnscale=-1))

$par
[1] 2.4510595 0.5293643

$value
[1] 4.760952

$counts
function gradient
     170       NA

$convergence
[1] 0

$message
NULL

$outer.iterations
[1] 3

$barrier.value
[1] 0.0009999994
```

The upper computation illustrates that under the method of NLP the maximum value of the function (1.29) arrives at `value = 4.7610` at maximum point of `par = (2.4511, 0.5294)`. And `function = 170` means that the objective function has been called 170 times.

1.3 Numerical Integration

In calculus it is not possible for every function $f \in C[a,b]$ to obtain the analytical representation of the corresponding indefinite integral. Therefore it couldn't be possible to compute the area under a curve analytically. An example would be the following integral:

$$\int_0^1 \exp(-x^2)\,dx. \tag{1.30}$$

There exists no closed representation of (1.30). Therefore the integral has to be computed numerically using methods of numerical integration. Sometimes also called quadrature. The basic idea behind the numerical integration lies in an approximation of the function by a polynomial and a later integration using Newton-Cotes-rule. If a function $f \in C[a,b]$ and nodes $a = x_0 < x_1 < \cdots < x_n = b$ are given, one searches a polynomial $p_n \in P_n$ with the condition:

$$p_n(x_k) = f(x_k),\ k \in \{0,\ldots,n\}. \tag{1.31}$$

To construct a polynomial that satisfies the above condition, the lagrange polynomial is considered. The Lagrange polynomial uses the following polynomials as a basis of P_n:

$$L_k(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}. \tag{1.32}$$

Therefore the following sum is consistent with the condition in (1.31),

$$p_n(x) = \sum_{k=0}^{n} f(x_k) L_k(x). \tag{1.33}$$

The polynomial in (1.33) is called Lagrange polynomial and is the unique polynomial that satisfies (1.31). Let $I(f) = \int_a^b f(x)\,dx$ be the exact integration operator for a function $f \in C[a,b]$. Then $I_n(f)$ is defined as the approximation of $I(f)$ using the above polynomial approximation for f:

$$I_n(f) = \int_a^b p_n(x)\,dx, \tag{1.34}$$

The equation in (1.34) can be restated using weights for the different function values of f:

$$I_n(f) = (b-a)\sum_{k=0}^{n} f(x_k)\alpha_k, \tag{1.35}$$
$$\text{with weights} \quad \alpha_k = \frac{1}{b-a}\int_a^b L_k(x)\,dx,$$

Because of its construction the formula in (1.35) is exact for every $f \in P_n$. Let the nodes x_k be chosen equidistant on $[a,b]$ with $x_k = a + kh$, where $h = \frac{b-a}{n}$. Then (1.35) is called (closed) Newton-Cotes-rule. The weights α_k can be computed explicitly up to $n = 7$ because starting from $n = 8$ negative weights occur and the Newton-Cotes-rule cannot be applied anymore. As an example for the Newton-Cotes-rule the *trapezoidal rule* is considered:

EXAMPLE 1.9 *For $n = 1$ and $I(f) = \int_a^b f(x)\,dx$ the nodes are given as follows: $x_0 = a$, $x_1 = b$. The weights can be computed explicitly by transforming the integral using two substitutions:*

$$\alpha_k = \frac{1}{b-a}\int_a^b L_k(x)\,dx = \int_0^1 \prod_{i=0, i\neq k}^{n} \frac{t-t_i}{t_k-t_i}\,dt = \frac{1}{n}\int_0^n \prod_{i=0, i\neq k}^{n} \frac{s-i}{k-i}\,ds. \tag{1.36}$$

Then the weights for $n = 1$ are $\alpha_0 = \frac{1}{2}$ and $\alpha_1 = \frac{1}{2}$. So the Newton-Cotes-rule $I_1(f)$ is given as the formulae to compute the surface area of a trapezoid:

$$I_1(f) = (b-a)\left\{\frac{f(a)+f(b)}{2}\right\}. \tag{1.37}$$

In R the *trapezoidal rule* is implemented within the package `caTools`. There the function `trapz(x,y)` can be used with a sorted vector `x` that contains the x-axis values and a vector `y` with the corresponding y-axis values. The function `trapz()` uses a summed version of the *trapezoidal rule*, where $[a,b]$ is separated in equidistant intervals and on every interval the *trapezoidal rule* is applied (called *extended trapezoidal rule*). For example consider the integral of the cosine function on $[-\frac{\pi}{2}, \frac{\pi}{2}]$ and separate the interval in 10 subintervals, where the *trapezoidal rule* is applied:

```
> x = (-5:5)*(pi/2)/5
> trapz(x, cos(x))
[1] 1.983524
```

The integral of the cosine function on $[-\frac{\pi}{2}, \frac{\pi}{2}]$ supposed to be exactly 2. So the absolute error is almost 0.02:

```
> c = trapz(x, cos(x))
> abs(c-2)
[1] 0.01647646
```

It can be shown that the error of the *trapezoidal rule* lies in $\mathcal{O}(h^3 f'')$ with an unknown value of the function's second derivative. If one uses a Newton-Cotes-rule with more nodes, integrand will be approximated with a polynomial of higher order. Therefore the error could diminish, if the integrand is very smooth, so that it can be approximated well by a polynomial.
In `R` the function `integrate()` uses an integration method that is based on the Gaussian quadrature (the exact method is called the Gauss-Kronrod quadrature, see [17] for further details). The Gaussian method approximates the integral so that polynomials of higher order can be integrated more exactly than using the Newton-Cotes-rule. It can produce accurate results for $n+1$ nodes if the integrand is a polynomial with order $2n+1$. In contrast to the Newton-Cotes-rule the Gaussian quadrature doesn't use equidistant nodes. Not only the weights α_k can be chosen, but also the location of the nodes. Therefore with the Gaussian quadrature one has twice as much degrees of freedom as with the Newton-Cotes-rule.

DEFINITION 1.8 *A method of numerical integration for a function $f : [a, b] \to \mathbb{R}$ with the formula*

$$I_n^G(f) = \sum_{k=0}^{n} f(x_k)\alpha_k,$$

and $n+1$ nodes is called Gaussian quadrature, if all $p \in P_{2n+1}$ are approximated exactly.

Thus one has twice as much degrees of freedom to minimize the expected error obtained in performing the approximation in (1.35). Without loss of generality, the interval $[-1, 1]$ is used instead of the interval $[a, b]$. It can be shown that the nodes of the Gaussian quadrature can be derived by finding the roots of a polynomial $p \in P_{n+1}$ with $p = \prod_{j=0}^{n}(x - x_j)$ that satisfies the following condition:

$$\int_{-1}^{1} pq(x)\,dx = 0, \quad \text{for all } q \in P_n. \tag{1.38}$$

Because $\int_{-1}^{1} f(x)g(x)\,dx$ is a scalar product for two functions f, g on $[-1, 1]$, the condition in (1.38) states that the polynomial p has to be orthogonal on every polynomial $q(x) \in P_n$. After computing the nodes, the weights of the Gaussian quadrature can be computed using (1.36) on the interval $[-1, 1]$. The polynomials that satisfy the condition (1.38) are called Gauss-Legendre polynomials. A more general approach of Gaussian quadrature splits the integrand into the product of a weighting function $w(x)$ and a rest $f(x)$ before the nodes and weights are determined. The function $w(x)$ can be chosen to remove integrable singularities. Then the Gaussian quadrature is exact for a more general class of functions. In the following an example the function `integrate(f,a,b,`
`subdivisions, rel.tol, abs.tol,...)` is used, where `f` is the integrand, `a` and `b` the lower and upper limit, `subdivisions` the number of subintervals on which the *Gauss-Kronrod quadrature* should be applied (standard: 100) and `rel.tol` as well as `abs.tol` for the relative

and absolute accuracy requested (standard: `.Machine$double.eps^0.25`). Consider as above the cosine function on the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$:

```
> integrate(cos, -pi/2, pi/2)
2 with absolute error < 2.2e-14
```

The output of the `integrate()` function delivers the computed value of the definite integral and an upper bound on the absolute error. In the example the absolute error is smaller than $2.2 \cdot 10^{-14}$. Therefore, the `integrate()` function is more accurate for the cosine function than the above-presented `trapz()` function.

1.3.1 Multivariate Integration

In this section we introduce three numerical methods for multiple integral including the repeated quadrature method, the adaptive quadrature method and the Monte-Carlo method.

Firstly, we introduce the repeated quadrature method. Similar to the numerical integration in the one dimensional context, in the case of high-dimension, a multivariate integration can be expressed as follows,

$$\int_{D_1} \dots \int_{D_n} f(x_1, \dots, x_n) dy_1 \dots dx_n \approx \sum_{i_n=1}^{N} \cdots \sum_{i_1=1}^{N} W_{i_1} \cdots W_{i_n} f(x_{i_1}, \dots, x_{i_n}), \tag{1.39}$$

where $D_i,\ i \in \{1, \dots, n\}$ is a integration region in $\mathbb{R}$ and $(x_{i_1}, \dots, x_{i_n})$ is the n-dimensional point at i-th dimension, where $i \in \{1, \dots, n\}$, and W_i is the coefficient used as the weight. The problem for the repeated quadrature is that when we compute the approximation (1.39) we need to evaluate N^n terms which will confront with the "curse of dimensionality" and computation-intensive.

Secondly, we introduce the adaptive method. The adaptive method in context of multiple-integration is proposed by Paul van Dooren and Luc de Ridder in 1976 *cf.* van Dooren and de Ridder [21]. The main algorithm behind this method is described briefly as follows.

1. Define the integration,

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} f(x_1, \dots, x_n) dx_1 \dots dx_n. \tag{1.40}$$

And two integration rules are applied in the computation,

$$R_5 = A_5 f(0, 0, \dots, 0) + B_5 \sum_{FS} f(s, 0, \dots, 0) + D_5 \sum_{FS} f(s, s, \dots, 0), \tag{1.41}$$

$$\begin{aligned} R_7 = {} & A_7 f(0, 0, \dots, 0) + B_7 \sum_{FS} f(\lambda_1, 0, \dots, 0) + C_7 \sum_{FS} f(\lambda_2, 0, \dots, 0) \\ & + D_7 \sum_{FS} f(\mu, \mu, \dots, 0) + E_7 \sum_{FS} f(\nu, \nu, \nu, 0, \dots, 0), \end{aligned} \tag{1.42}$$

where $\sum_{FS}$ stands for "fully symmetric", that is the summation of all permutations of the coordinates, $A_7, B_7, \ldots, D_5, \lambda_1, \lambda_2, \mu, \nu, s$ are parameters given in Stroud (see Stroud [19]).

2. Subdivide the hyper-rectangle into 2 subregions V_1, V_2 and for each subregion R_5 and R_7 will be employed to obtain the estimate of integral I_i and error E_i, and the total sum of them, $\hat{I} = \sum_i I_i$ and $\hat{E} = \sum_i E_i$, which can be seen as the approximation of integral and error over the whole integration region. In this step if $\hat{E} = \sum_i E_i$ is smaller than the tolerance set at the beginning then the algorithm stops, otherwise it continues to the next step.

3. The subregion V_j with the largest error E_j will be selected out and divided into 2 another subregions and again the step 2 will be iterated for computing the integral estimates and error estimates.

4. Iterate from the second step.

Berntsen *et al.* (1991a) *cf.* Jarle Berntsen and Genz [11] improved the reliability of Paul van Dooren and Luc de Ridder (1976) *cf.* van Dooren and de Ridder [21] from two aspects including the strategy of selection of subregions, error estimation and parallelization of computation.

Thirdly, we introduce the Monte-Carlo method. For a multiple integral

$$I = \int_{a_1}^{b_1} \ldots \int_{a_n}^{b_n} f(x_1, \ldots, x_n) dx_1 \ldots dx_n = \int \cdots \int_D f(x) dx, \tag{1.43}$$

where x stands for a vector $(x_1, \ldots, x_n)$ and D the integration region, X is now set as a random vector, and each component X_i is in X uniformly distributed in each dimension D_i. Then the algorithm of the Monte-Carlo multiple-integration can be described as follows. In the 1st step, N points are randomly and uniformly dropped out from region D, such that

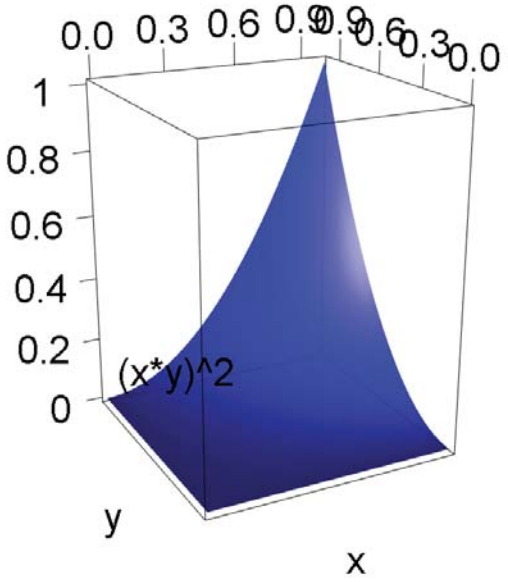
$$(x_{1,1}, \ldots, x_{1,n}), \ldots, (x_{N,1}, \ldots, x_{N,n}),$$

where N represents the number of random points dropped out and n represents the dimensionality. In the 2nd step, n-dimensional volume V is estimated and evaluation of integrand f at each of N points is performed. In the 3rd step, the integral I can be estimated using a sample moment function,

$$I(f) \approx \hat{I}(f) = V \frac{1}{N} \sum_{i=1}^{N} f(x_{i,1}, \ldots, x_{i,n}). \tag{1.44}$$

And the absolute error ϵ can be approximated as follows,

$$\epsilon = |I - \hat{I}| \approx \frac{V I(f^2) - I^2(f)}{\sqrt{N}}. \tag{1.45}$$



(a)

Figure 1.6: Integrand Graph for (1.46).
BSCintegrand.R

Next we give 2 numerical examples using adaptive method and Monte-Carlo method. For the computation for multiple-integral we use R package `R2Cuba` developed by Thomas Hahn *cf.* Hahn [9], in which 4 different algorithms are included for multivariate integration, where the function `cuhre` utilizes adaptive method and `vegas` function utilizes Monte-Carlo method.

EXAMPLE 1.10 *Evaluate the bivariate function,*

$$\int \int_D x^2 y^2 dxdy, \tag{1.46}$$

where D is the integration region with $x \in [0,1],\ y \in [0,1]$.

For this example, the integration $\int \int_D x^2 y^2 dxdy$ is trivial and has analytical solution $1/12$. The graph of integration can be shown in Figure 1.6. Here we use both adaptive method and Monte-Carlo method to compute.

```
# set number of the dimension of the integration.
> NoDim = 2
# set number of the integrand.
```

```
> NoInt = 1
# construct the integrand.
> integrand = function(arg, phase) {
+ x=arg[1]
+ y=arg[2]
+ f=x^{2}*y^{3};
+ return(f)
+ }
# do integration computation.
> divonne(NoDim, NoInt, integrand,
+ lower=rep(0, 2), upper=rep(1, 2),
+ rel.tol=1e-3, abs.tol=1e-12,
+ flags=list(verbose=2), key1=47)
Divonne input parameters:
  ndim 2
  ncomp 1
  rel.tol 0.001
  abs.tol 1e-012
  pseudo.random  0
  final 0
  verbose 2
  min.eval 0
  max.eval 50000
  key1 47
  key2 1
  key3 1
  max.pass 5
  border 0
  max.chisq 10
  min.deviation 0.25
  ngiven 0
  nextra 0
Partitioning phase:
Iteration 1 (pass 0):  7 regions
    577 integrand evaluations so far,
    243 in optimizing regions,
     22 in finding cuts
[1] 0.0835638 +- 0.00197331
Iteration 2 (pass 0):  8 regions
    660 integrand evaluations so far,
```

```
    271 in optimizing regions,
     29 in finding cuts
...
...
...
Iteration 15 (pass 5):  21 regions
   1774 integrand evaluations so far,
    686 in optimizing regions,
    104 in finding cuts
[1] 0.0834081 +- 0.000257177

Main integration on 21 regions with 149 samples per region.
integral: 0.0833479364765 (+-7.7e-05)
nregions: 21; number of evaluations:  3349;
probability:  4.82144415846e-17
```

The output shows that the adaptive algorithm subdivided the D into 21 subregions and 149 points for every subregion are employed, and the approximation of integral I is 0.08335, which can be compared with the analytical solution 0.08. For more detail output introduction reader can refer to Hahn [9].

```
# set the number of the dimension of the integration.
> NoDim = 2
# set the number of the integrand.
> NoInt = 1
# construct the integrand.
> integrand = function(arg, phase) {
+ x = arg[1]
+ y = arg[2]
+ f = x^{2}*y^{3};
+ return(f)
+ }
# do the integration.
> vegas(NoDim, NoInt, integrand, rel.tol = 1e-3,
+ abs.tol = 1e-12, flags = list(verbose=2))
Vegas input parameters:
  ndim 3
  ncomp 1
  rel.tol 0.001
  abs.tol 1e-012
  smooth 0
```

```
  pseudo.random  0
  final 0
  verbose 2
  min.eval 0
  max.eval 50000
  nstart 1000
  nincrease 500
  vegas.gridno 0
  vegas.state ""
Iteration 1:  1000 integrand evaluations so far
[1] 0.0829732 +- 0.00463074     chisq 0 (0 df)
Iteration 2:  2500 integrand evaluations so far
[1] 0.0834441 +- 0.00121646     chisq 0.011107 (1 df)
...
...
...
Iteration 7:  17500 integrand evaluations so far
[1] 0.0832916 +- 7.90009e-005   chisq 2.17857 (6 df)
integral: 0.0832915860332 (+-7.9e-05)
number of evaluations:  17500; probability:  0.0974358205562
```

The output shows that the Monte-Carlo algorithm executed 7 iteration and 17500 evaluation of integrand, and the approximation of integral I is 0.0832, which can be compared with the analytical solution 0.08 and adaptive solution 0.0833.

EXAMPLE 1.11 *Evaluate the tri-variate function,*

$$\int\int\int_D \sin(x)\cos(2y)\exp(3z)dxdydz, \tag{1.47}$$

where D is the integration region with $x \in [0,1],\ y \in [0,1],\ z \in [0,1]$.

R code for both adaptive and Monte-Carlo method are given as follows,

```
# using the adaptive method.
# construct the integrand.
integrand = function(arg, weight) {
x = arg[1]
y = arg[2]
z = arg[3]
f = sin(x)*cos(2*y)*exp(3*z);
return(f)
```

```
}
# do the integration computation.
divonne(3, 1, integrand, rel.tol=1e-3, abs.tol=1e-12,
flags=list(verbose=2))

# using the Monte-Carlo method.
# construct the integrand.
integrand = function(arg, weight) {
x = arg[1]
y = arg[2]
z = arg[3]
f = sin(x)*cos(2*y)*exp(3*z);
return(f)
}
# do the integration computation.
vegas(3, 1, integrand, rel.tol=1e-3, abs.tol=1e-12,
flags = list(verbose=2))
```

The output of the adaptive method gives the integral 1.3296 and the Monte-Carlo method gives 1.3297.

1.4 Numerical Differentiation

The analytic computation of the derivative may be impossible if the function to be differentiated is given only indirectly (for example by an algorithm) and can be evaluated only point-wise. Therefore it is necessary to use numerical methods in order to compute the derivative of a function. Before presenting some numerical methods for Differentiation, it is demonstrated how one can compute the derivative analytically in R.

1.4.1 Analytical Differentiation in R

To calculate the derivative of a one-dimensional function in R, the function `D(expr,name)` is used. For `expr` the function is inserted (as an object of mode `expression`) and `name` identifies the variable with respect to which a derivative will be computed. Consider the following example:

```
> f = expression(3*x^3+x^2)
> D(f, "x")
3 * (3 * x^2) + 2 * x
```

Because the function `D()` returns an argument of type `call` (see `help(call)` for further information), `D()` can be used recursive to compute higher-order derivatives. For example consider the second derivative of $3x^3 + x^2$:

```
> D(D(f, "x"), "x")
3 * (3 * (2 * x)) + 2
```

To compute higher-order derivatives, it can be useful to define a recursive function:

```
DD = function(expr,name, order = 1) {
    if(order < 1) stop("'order' must be >= 1")
    if(order == 1) D(expr,name)
    else DD(D(expr, name), name, order - 1)
}
```

Then the third derivative for $3x^3 + x^2$ can be computed as follows:

```
> DD(f, "x", order=3)
3 * (3 * 2)
```

If one wants to compute the gradient of a function, it can also be done using the function `D()`. Before it is shown how to compute the gradient using `D()`, a short definition of the gradient is given.

DEFINITION 1.9 *Let f be a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ and $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$. Then the vector*

$$gradf(x) = \left\{ \frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right\}^\top,$$

is called the gradient of f at point x.

Now consider a function $f : \mathbb{R}^2 \to \mathbb{R}$, which maps the x and y coordinates to the square of their Euclidean norm:

```
> f = expression(x^2+y^2)
> expr = expression(D(f, "x"),D(f, "y"))
> grad = c(eval(expr[1]), eval(expr[2]))
> grad
[[1]]
2 * x

[[2]]
2 * y
```

The variable `grad` is then a vector filled with `expressions`. Therefore one could compute the

second derivative with respect to x by using `grad[[1]]`:

```
> D(grad[[1]], "x")
[1] 2
```

If it is necessary to have the gradient as a function that can be evaluated easily, the function `deriv(f, name, function.arg = NULL, hessian = FALSE)` should be used, where `f` is the function (as an object of mode `expression`), `name` identifies the vector with respect to which a derivative will be computed, `function.arg` specifies the parameters of the returned function and `hessian` indicates whether the second derivatives should be calculated. When `function.arg` is not specified, the return value of `deriv()` is an `expression` and not a function. As an application of the function `deriv()`, consider the above function `f`:

```
> eucld2 = deriv(f, c("x", "y"), function.arg=c("x", "y"))
> eucld2(2, 2)
[1] 8
attr(, "gradient")
     x y
[1,] 4 4
```

The function `eucld2(x,y)` delivers the function value of `f` at `(x,y)` and as an attribute (see `help(attr)`) the gradient of `f` evaluated at `(x,y)`. If only the evaluated gradient at `(x,y)` should be returned, the function `attr(x, which)` should be used, where `x` is an object and `which` a non-empty character string specifying which attribute is to be accessed:

```
> attr(eucld2(2, 2), "gradient")
     x y
[1,] 4 4
```

If the option `hessian` is set to `TRUE`, the hessian matrix at a point `(x,y)` is also given by using the function `attr(eucld(2,2),"hessian")`.

1.4.2 Numerical Methods

To develop numerical methods for determining derivatives of a function at a point x we use Taylor expansion given as follows,

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x) + \mathcal{O}(h^3). \tag{1.48}$$

Only if the fourth derivative of f exists and f is bounded on $[x, x+h]$, the representation in (1.48) is valid. If the Taylor expansion is ended after the linear term, then (1.48) can be solved

for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h^2). \tag{1.49}$$

Therefore an approximation for the derivative at point x could be the following expression:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{1.50}$$

Another more accurate method uses the Richardson extrapolation. Let's define the expression in (1.50) with $g(h) = \frac{f(x+h)-f(x)}{h}$. Then the Formula (1.49) can be written as follows:

$$f'(x) = g(h) + k_1 h + k_2 h^2 + k_3 h^3 + \cdots . \tag{1.51}$$

where $k_1, k_2, k_3, \ldots$ represent the constant terms involving the derivatives at point x. Because the Taylor theorem holds for all positive h, one can replace h by $\frac{h}{2}$:

$$f'(x) = g\left(\frac{h}{2}\right) + k_1 \frac{h}{2} + k_2 \frac{h^2}{4} + k_3 \frac{h^3}{8} + \cdots . \tag{1.52}$$

Now (1.51) can be subtracted from twice equation (1.52). Then the term involving k_1 is eliminated:

$$f'(x) = 2g\left(\frac{h}{2}\right) - g(h) + k_2\left(\frac{h^2}{2} - h^2\right) + k_3\left(\frac{h^3}{4} - h^3\right) + \cdots . \tag{1.53}$$

Therefore $f'(x)$ can be rewritten as follows:

$$f'(x) = 2g\left(\frac{h}{2}\right) - g(h) + \mathcal{O}(h^2). \tag{1.54}$$

This process can be continued to achieve formulae of higher order. In `R` the package `numDeriv` provides some functions that use the above-presented methods to differentiate a function numerically. For example the function `grad(func, x, method="Richardson", method.args=list(eps=1e-4, d=0.0001, zero.tol,...)` calculates a numerical approximation of the gradient of `func` at the point `x`. As a `method` one can choose `"simple''` or `''Richardson''`. If the method `simple` is used, a formula as in (1.50) is applied. Then only the element `eps` of `methods.args` is used (equivalent to the above h in (1.50)). The method `''Richardson''` uses the Richardson extrapolation. In the following an example for computing the gradient at a point `x` is given. Consider the function f and $(x_1, x_2, x_3) = \sqrt{x_1^2 + x_2^2 + x_3^2}$ that has the following gradient

$$gradf(x) = \left(\frac{x_1}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_2}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_3}{\sqrt{x_1^2 + x_2^2 + x_3^2}}\right). \tag{1.55}$$

The gradient of f represents the normalized coordinates of a vector with respect to the Euclidean norm. The evaluation of the gradient using `grad()` e.g. at the point `(1,0,0)` would be as follows:

```
> func = function(x){sqrt(sum(x^2))}
> library(numDeriv)
> grad(func, c(1,0,0))
[1] 1 0 0
```

It could be also of interest to compute the jacobian or the hessian matrix of a function $F : \mathbb{R}^n \to \mathbb{R}^m$ numerically. In the following a short definition of the jacobian and the hessian matrix will be given.

DEFINITION 1.10 *Let F be a differentiable function $F : \mathbb{R}^n \to \mathbb{R}^m$ with $F = (f_1, \ldots, f_m)$ and the coordinates $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, then the Jacobian matrix in a point $b \in \mathbb{R}^n$ is defined as follows:*

$$J_F(b) := \left\{ \frac{\partial f_i(b)}{\partial x_j} \right\}_{i=1,\ldots,m;\, j=1,\ldots,n}.$$

In `R` the function `jacobian(func, x, method="Richardson", method.args=list(), ...)` can be used to compute the Jacobian matrix for a function `func` at a point `x`. As with the function `grad()`, the function `jacobian()` uses the Richardson extrapolation by default. Consider the following example, where the Jacobian matrix of the function $F = \{sin(x_1 + x_2),\ cos(x_1 + x_2)\}$ at the point $(0, 2\pi)$ is computed:

```
> func1 = function(x){c(sin(sum(x)), cos(sum(x)))}
> x = (0:1)*2*pi
> library(numDeriv)
> jacobian(func1, x)
     [,1] [,2]
[1,]    1    1
[2,]    0    0
```

DEFINITION 1.11 *Let F be a function $f : \mathbb{R}^n \to \mathbb{R}$ that is twice continuously differentiable. Then the hessian matrix of f at a point b is defined as follows:*

$$(Hess f)(b) := \left\{ \frac{\partial^2 f}{\partial x_i \partial x_j}(b) \right\}_{i,j=1,\ldots,n}. \tag{1.56}$$

The Hessian matrix is symmetric and can be computed in `R` with `hessian(func, x, method="Richardson", method.args=list(), ...)` a twice continuously differentiable function

`func` at a point `x`. For example consider the above-presented function f, that maps the coordinates of a vector to their Euclidean norm (e.g. $x \in \mathbb{R}^3$ with $x \mapsto \sqrt{x_1^2 + x_2^2 + x_3^2}$). The following computation provides the hessian matrix at the point $(0, 0, 0)$:

```
> hessian(func, c(0,0,0))
          [,1]       [,2]       [,3]
[1,] 194419.75 -56944.23 -56944.23
[2,] -56944.23 194419.75 -56944.23
[3,] -56944.23 -56944.23 194419.75
```

From the definition of the Euclidean norm, it would make sense, if f has a minimum at $(0, 0, 0)$. The above information can be used to check whether the function f has a local minimum at point $(0, 0, 0)$. In order to check that, two conditions have to be fulfilled. The gradient at $(0, 0, 0)$ has to be the zero vector and the hessian matrix should be positive-definite (see [5] for further information on the calculation of local extreme values using the hessian matrix). The second condition can be restated by using the fact that a positive-definite matrix has only positive eigenvalues. Therefore the second condition can be checked by computing the eigenvalues of the above hessian matrix and the first condition can be checked easily using the above-presented `grad()` function:

```
> grad(func, c(0,0,0))
[1] 0 0 0
> hessm<-hessian(func,c(0,0,0))
> eigen(hessm)$values
[1] 251364.0 251364.0  80531.3
```

The above output shows that the gradient at $(0, 0, 0)$ is the zero vector and the eigenvalues are all positive. Therefore as expected, the point $(0, 0, 0)$ is a local minimum of f.

1.4.3 Automatic Differentiation

For function $f : \mathbb{R}^n \to \mathbb{R}^m$ automatic differentiation (AD), which is also referred using the terminology algorithmic differentiation or computational differentiation, is a technique employed for evaluation of derivatives based on incorporation of chain-rule. As derivatives of elementary functions, such as exp, log, sin, cos, *etc.*, are already known, therefore the whole derivative of function f can be an automatic assemble of all these known elementary function partial derivative according to the chain-rule.

Automatic differentiation is different from two other methods of differentiation, symbolic differentiation and numerical differentiation.

The main difference between the automatic differentiation and the symbolic differentiation

is that the later is focused on the symbolic expression of formulae and the automatic differentiation is concentrated on the evaluation. The disadvantages of symbolic differentiation lie in both taking up too much memory in computation and generation of unnecessary expression associated to derivative computation. For instance do a symbolic differentiation for function

$$f(\mathbf{x}) = \prod_{i=1}^{10} x_i = x_1 * x_2 * \cdots * x_{10}, \tag{1.57}$$

The corresponding gradient in symbolic style is shown as follows,

$$\begin{aligned} \nabla f(\mathbf{x}) &= (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_{10}}), \\ &= (x_2 * x_3 * \cdots * x_{10}, \\ &\quad x_1 * x_3 * \cdots * x_{10}, \\ &\quad \ldots \\ &\quad x_1 * x_2 * \cdots * x_9). \end{aligned} \tag{1.58}$$

If we set $n = 10^{10}$ then the expression will be a memory-tremendous and tedious representation.

The main difference between automatic differentiation and numerical differentiation (or divided difference) is that for derivative such as

$$f'(x) \approx \frac{f(x+k) - f(x)}{k}, \tag{1.59}$$

or

$$f'(x) \approx \frac{f(x+k) - f(x-k)}{2k}, \tag{1.60}$$

it is obvious that the accuracy of this type differentiation is related with the choice of k, that is if k is small then divided difference has errors introduced by rounding-off the floating point numbers and if k is large then the formula disobeys the essential of this method, which assumes k leading to zero. Also divided difference method introduces truncation error as neglecting the $\mathcal{O}(k^2)$, which will not appear in automatic differentiation.

Automatic differentiation has two modes of operations, that are forward mode and reverse mode. For forward mode, the algorithm starts from evaluation of given points at derivatives of every elementary function of f and in every intermediate step evaluations are continuously executed. The last step is only to assemble the evaluations from results of ahead performed computations according to chain-rule. A vanilla forward example is given as follows. For example we use the forward mode to evaluate a derivative of function $f(x) = (x + x^2)^3$, then the pseudo code can be shown,

```
function(y,y')=f'(x,x')
  s1=x*x;
    s1'=2*x*x';
  s2=x+s1;
    s2'=x'+s1';
  y=s2*s2*s2;
    y'=3*s2*s2*s2'
end
```

where v' represents the derivative, *i.e.* dv/dx. Therefore let us evaluate the derivative of $f(x) = (x + x^2)^3$ at point $x = 2$ with the forward mode, then

$$\begin{aligned} s_1 = x * x = 2 * 2 &= 4, \\ s_1^{'} = 2 * x * x = 2 * 2 * 1 &= 4, \\ s_2 = x + s_1 = 2 + 4 &= 6, \\ s_2^{'} = x' + s_1^{'} = 1 + 4 &= 5, \\ y = s_2 * s_2 * s_2 = 6 * 6 * 6 &= 216, \\ y^{'} = 3 * s_2 * s_2 * s_2^{'} = 3 * 6 * 6 * 5 &= 540. \end{aligned}$$

For the reverse mode, the programme performs the computation in a reverse version. We need to set $\bar{v} = dy/dv$, then $\bar{y} = dy/dy = 1$. We use the same example as before, where we evaluate the derivative at $x = 2$, then we derive the following computation,

$$\begin{aligned} \bar{s}_2 = 3 * s_2 = 3 * 6 &= 108, \\ \bar{s}_1 = 3 * s_2 = 3 * 6 &= 108, \\ \bar{x} = \bar{s}_2 + 4\bar{s}_1 = 108 + 4 * 108 &= 540. \end{aligned} \tag{1.61}$$

At last, we give 2 examples employing R computation, where we use the package `radx` developed by Chidambaram Annamalai *cf.* Annamalai [1].

EXAMPLE 1.12 *Evaluate the first order derivative of a univariate function,*

$$f(x) = (x + x^2)^3, \tag{1.62}$$

at point $x = 2$

```
> # define a one input, one output scalar function.
> f = function(x) {
+ (x^2+x)^3
```

```
+ }
> # find its derivatives up to the first order
> # at the point (x=2).
> radxeval(f, c(2), 1)
     [,1]
[1,]  540
```

The upper computation illustrates that the value of 1st derivative of the function (1.62) at the point $x = 2$ is equal to 540.

EXAMPLE 1.13 *Evaluate the first and second order derivatives of a vector function,*

$$f_1(x, y) = 1 - 3y + sin(3\pi y) - x, \tag{1.63}$$
$$f_2(x, y) = y - sin(3\pi x)/2, \tag{1.64}$$

at point $(x = 3, y = 5)$

```
> # define a two input, two output vector function.
> f = function(x, y) {
+ c(1-3*y+sin(3*pi*y)-x, y-sin(3*pi*x)/2)
+ }
>
> # find its derivatives up to the first order
  # at the point (x=3,y=5).
> radxeval(f, c(3,5), 1)
          [,1]     [,2]
[1,]  -1.00000 4.712389
[2,] -12.42478 1.000000
>
> # find its derivatives up to the second order
  # at the point (x=3,y=5).
> radxeval(f, c(3,5), 2)
             [,1]          [,2]
[1,]  0.00000e+00 4.894984e-14
[2,]  0.00000e+00 0.000000e+00
[3,] -4.78741e-13 0.000000e+00
```

Symbols and Notations

Basics

X, Y	random variables or vectors
$X_1, X_2, \ldots, X_p$	random variables
$X = (X_1, \ldots, X_p)^\top$	random vector
$X \sim \cdot$	X has distribution $\cdot$
$\mathcal{A}, \mathcal{B}$	matrices
Γ, Δ	matrices
$\mathcal{X}, \mathcal{Y}$	data matrices
Σ	covariance matrix
1_n	vector of ones $(\underbrace{1, \ldots, 1}_{n\text{-times}})^\top$
0_n	vector of zeros $(\underbrace{0, \ldots, 0}_{n\text{-times}})^\top$
$\mathrm{I}(.)$	indicator function, i.e. for a set M is $\mathrm{I} = 1$ on M, $\mathrm{I} = 0$ otherwise
$\lceil \ldots \rceil$	Ceiling function
$\lfloor \ldots \rfloor$	Floor function
$\mathbf{i}$	imaginary unit, $\mathbf{i}^2 = -1$
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence
$\approx$	approximately equal
$\otimes$	Kronecker product
$\xrightarrow{a.s}$	almost sure convergence
iff	if and only if, equivalence
i.i.d.	independent and identically distributed
rv	random variable
$\mathbb{R}^n$	n dimensional space of real numbers
δ_{ik}	The kronecker delta, that is 1 if $i = k$ and 0 otherwise
P_n	$P_n = \{v \in C[a, b] \vert v(x) = \sum_{i=0}^{n} a_i x^i, a_i \in \mathbb{R}\}$
$f(x) \in \mathcal{O}(g(x))$	There is $k > 0$ such that for all sufficiently large values of x, $f(x)$ is at most $kg(x)$ in absolute value
$med(x)$	the median value of rv X

Samples

x, y	observations of X and Y
$x_1, \ldots, x_n = \{x_i\}_{i=1}^n$	sample of n observations of X
$\mathcal{X} = \{x_{ij}\}_{i=1,\ldots,n;j=1,\ldots,p}$	$(n \times p)$ data matrix of observations of $X_1, \ldots, X_p$ or of $X = (X_1, \ldots, X_p)^T$
$x_{(1)}, \ldots, x_{(n)}$	the order statistic of $x_1, \ldots, x_n$
$\mathcal{H}$	centering matrix, $\mathcal{H} = \mathcal{I}_n - n^{-1} 1_n 1_n^\top$
$\bar{x}$	the sample mean

Densities and Distribution Functions

$f(x)$	density of X
$f(x, y)$	joint density of X and Y
$f_X(x), f_Y(y)$	marginal densities of X and Y
$f_{X_1}(x_1), \ldots, f_{X_p}(x_2)$	marginal densities of $X_1, \ldots, X_p$
$\hat{f}_h(x)$	histogram or kernel estimator of $f(x)$
$F(x)$	distribution function of X
$F(x, y)$	joint distribution function of X and Y
$F_X(x), F_Y(y)$	marginal distribution functions of X and Y
$F_{X_1}(x_1), \ldots, f_{X_p}(x_p)$	marginal distribution functions of $X_1, \ldots, X_p$
$\varphi_X(t)$	characteristic function of X
m_k	k-th moment of X
κ_j	cumulants or semi-invariants of X
edf	Empirical cumulative distribution function
pdf	Probability density function
qf	quantile function

Moments

$\mathsf{E}\, X, \mathsf{E}\, Y$	mean values of random variables or vectors X and Y
$\sigma_{XY} = \mathsf{Cov}(X, Y)$	covariance between random variables X and Y
$\sigma_{XX} = \mathsf{Var}(X)$	variance of random variable X
$\rho_{XY} = \dfrac{\mathsf{Cov}(X, Y)}{\sqrt{\mathsf{Var}(X)\,\mathsf{Var}(Y)}}$	correlation between random variables X and Y
$\Sigma_{XY} = \mathsf{Cov}(X, Y)$	covariance between random vectors X and Y, i.e., $\mathsf{Cov}(X, Y) = E(X - EX)(Y - EY)^\top$
$\Sigma_{XX} = \mathsf{Var}(X)$	covariance matrix of the random vector X

Empirical Moments

$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$	average of X sampled by $\{x_i\}_{i=1,\dots,n}$
$s_{XY} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$	empirical covariance of random variables X and Y sampled by $\{x_i\}_{i=1,\dots,n}$ and $\{y_i\}_{i=1,\dots,n}$
$s_{XX} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$	empirical variance of random variable X sampled by $\{x_i\}_{i=1,\dots,n}$
$r_{XY} = \frac{s_{XY}}{\sqrt{s_{XX}s_{YY}}}$	empirical correlation of X and Y
$\mathcal{S} = \{s_{X_iX_j}\} = x^\top \mathcal{H} x$	empirical covariance matrix of $X_1, \dots, X_p$ or of the random vector $X = (X_1, \dots, X_p)^\top$
$\mathcal{R} = \{r_{X_iX_j}\} = \mathcal{D}^{-1/2}\mathcal{S}\mathcal{D}^{-1/2}$	empirical correlation matrix of $X_1, \dots, X_p$ or of the random vector $X = (X_1, \dots, X_p)^\top$

Distributions

$\varphi(x)$	density of the standard normal distribution
$\Phi(x)$	distribution function of the standard normal distribution
$N(0,1)$	standard normal or Gaussian distribution
$N(\mu, \sigma^2)$	normal distribution with mean μ and variance σ^2
$N_p(\mu, \Sigma)$	p-dimensional normal distribution with mean μ and covariance matrix Σ
$\xrightarrow{\mathcal{L}}$	convergence in distribution
$\overset{a}{\sim}$	asymptotic distribution
$U(a,b)$	uniform distribution on (a,b)
CLT	Central Limit Theorem
χ^2_p	χ^2 distribution with p degrees of freedom
$\chi^2_{1-\alpha;p}$	$1-\alpha$ quantile of the χ^2 distribution with p degrees of freedom
t_n	t-distribution with n degrees of freedom
$t_{1-\alpha/2;n}$	$1-\alpha/2$ quantile of the t-distribution with n d.f.
$F_{n,m}$	F-distribution with n and m degrees of freedom
$F_{1-\alpha;n,m}$	$1-\alpha$ quantile of the F-distribution with n and m degrees of freedom
$B(n,p)$	Binomial distribution
$H(x;n,M,N)$	Hypergeometric distribution
Pois(λ_i)	Poisson distribution with parameter λ_i

Mathematical abbreviations

$\mathrm{tr}(\mathcal{A})$	trace of matrix $\mathcal{A}$
$\mathrm{hull}(x_1, \ldots, x_k)$	convex hull of points $\{x_1, \ldots, x_k\}$
$\mathrm{diag}(\mathcal{A})$	diagonal of matrix $\mathcal{A}$
$\mathrm{rank}(\mathcal{A})$	rank of matrix $\mathcal{A}$
$\det(\mathcal{A})$	determinant of matrix $\mathcal{A}$
$\mathcal{I}$	Identity matrix
id	identity function on a vector space V
$C[a, b]$	The set of all continuous differentiable functions on the intervall $[a, b]$

Bibliography

[1] Annamalai, C. [2010]. Package "radx".
URL: *https://github.com/quantumelixir/radx*

[2] Bihorel, S. [2012]. The neldermead package.
URL: *http://cran.r-project.org/web/packages/neldermead/vignettes/neldermead.pdf*

[3] Braun, W. and Murdoch, D. [2007]. *A First Course in Statistical Programming with R*, Cambridge University Press.

[4] Broyden, C. G. [1970]. The convergence of a class of double-rank minimization algorithms, *Journal of the Institute of Mathematics and Its Applications* **6**: 76–90.

[5] Canuto, C. and Tabacco, A. [2010]. *Mathematical Analysis II*, Universitext Series, Springer.

[6] Fletcher, R. [1970]. A new approach to variable metric algorithms, *Computer Journal* **13**: 317–322.

[7] Goldfarb, D. [1970]. A family of variable metric updates derived by variational means, *Mathematics of Computation* **24**: 23–26.

[8] Greub, W. [1975]. *Linear Algebra*, Graduate Texts in Mathematics, Springer.

[9] Hahn, T. [2013]. Package "r2cuba".
URL: *http://cran.r-project.org/web/packages/R2Cuba/R2Cuba.pdf*

[10] Hestenes, M. R. and Stiefel, E. [1952]. Methods of conjugate gradients for solving linear systems, *Journal of Research of the National Bureau of Standards* **49**.

[11] Jarle Berntsen, T. E. and Genz, A. [1991]. An adaptive algorithm for the approximate calculation of multiple integrals, *ACM Transactions on Mathematical Software* **17**: 437–451.

[12] Kiefer, J. [1953]. Sequential minimax search for a maximum, *Proceedings of the American Mathematical Society* **4 (3)**: 502–506.

[13] Moore, E. [1920]. On the reciprocal of the general algebraic matrix, *Bull. Amer. Math. Soc* **26**: 394–395.

[14] Nash, J. C. N. and Varadhan, R. [2011]. Unifying optimization algorithms to aid software system users: optimx for r, *Journal of Statistical Software* **43**.

[15] Nelder, J. A. and Mead, R. [1965]. A simplex method for function minimization, *Computer Journal* **7**: 308–313.

[16] Penrose, R. [1955]. A generalized inverse for matrices, *Proc. Cambridge Philos. Soc*, Vol. 51, Cambridge Univ Press, pp. 406–413.

[17] Press, W. [1992]. *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press.

[18] Shanno, D. F. [1970]. Conditioning of quasi-newton methods for function minimization, *Mathematics of Computation* **24**: 647–656.

[19] Stroud, A. H. [1971]. *Approximation Caculation of Multiple Integrals*, New Jersey: Prentice Hall.

[20] Theussl, S. [2013]. Package "rglpk".
URL: *http://cran.r-project.org/web/packages/Rglpk/Rglpk.pdf*

[21] van Dooren, P. and de Ridder, L. [1976]. An adaptive algorithm for numerical integration over an n-dimensional cube, *Journal of Computational and Applied Mathematics* **2**: 207–217.