Chapter 2
Numerical Techniques

The general who wins a battle makes many calculations in the temple before an attack.

— Sun Tzu, *The Art of War*

With more and more practical problems of applied mathematics appearing in different disciplines, such as chemistry, biology, geology, management and economics, to mention just a few, the demand for numerical computation has considerably increased. These problems frequently have no analytical solution or the exact result is time-consuming to derive. To solve these problems, numerical techniques are used to approximate the result. This chapter introduces matrix algebra, numerical integration, differentiation and root finding.

2.1 Matrix Algebra

Matrix algebra is a fundamental concept for applying and understanding numerical methods, therefore the beginning of this section introduces the basic characteristics of matrices, their operations and their implementation in R. Thereafter, other operations, such as the inverse, including the inverse both of non-singular and of singular matrices, the norms containing the vector norms and the matrix norms, the calculation of eigenvalues and eigenvectors and different types of matrix decompositions are presented. Some theorems and accompanying examples computed in R are also provided.

W.K. Härdle et al., *Basic Elements of Computational Statistics*,
Statistics and Computing, DOI 10.1007/978-3-319-55336-8_2

2.1.1 Characteristics of Matrices

A matrix $\mathcal{A}_{(n \times p)}$ is a system of numbers with n rows and p columns:

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{np} \end{pmatrix} = (a_{ij}).$$

Matrices with one column and n rows are column vectors, and matrices with one row and p columns are row vectors. The following `R` code produces (3×3) matrices `A` and `B` with the numbers from 1 to 9 and from 0 to -8, respectively. The matrices are filled by rows if `byrow = TRUE`.

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

There are several special matrices that are frequently encountered in practical and theoretical work. Diagonal matrices are special matrices where all off-diagonal elements are equal to 0, that is, $\mathcal{A}_{(n \times p)}$ is a diagonal matrix if $a_{ij} = 0$ for all $i \neq j$. The function `diag()` creates diagonal matrices (square or rectangular) or extracts the main diagonal of a matrix in `R`.

```
> A = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
> diag(x = A)                                  # extract diagonal
[1] 1 5 9
> diag(3)                                      # identity matrix
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> diag(2, 3)                                   # 2 on diag, 3x3
     [,1] [,2] [,3]
[1,]    2    0    0
[2,]    0    2    0
[3,]    0    0    2
> diag(c(1, 5, 9, 13), nrow = 3, ncol = 4) # 3x4
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    5    0    0
[3,]    0    0    9    0
> diag(2, 3, 4)                                # 3x4, 2 on diagonal
     [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    2    0    0
```

```
[3,]    0    0    2    0
```

As seen from the listing above, the argument `x` of `diag()` can be a matrix, a vector, or a scalar. In the first case, the function `diag()` extracts the diagonal elements of the existing matrix, and in the remaining two cases, it creates a diagonal matrix with a given diagonal or of given size.

Rank

The rank of $\mathcal{A}$ is denoted by $\text{rank}(\mathcal{A})$ and is the maximum number of linearly independent rows or columns. Linear independence of a set of h rows a_j means that $\sum_{j=1}^{h} c_j a_j = 0_p$ if and only if $c_j = 0$ for all j. If the rank is equal to the number of rows or columns, the matrix is called a full-rank matrix. In `R` the rank can be calculated using the function `qr()` (which does the so-called QR decomposition) with the object field `rank`.

```
> A = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
> qr(A)$rank # rank of matrix A
[1] 2
```

The matrix `A` is not of full rank, because the second column can be represented as a linear combination of the first and third columns:

$$\begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1+3 \\ 4+6 \\ 7+9 \end{pmatrix} \tag{2.1}$$

This shows that the general condition for linear independence is violated for the specific matrix `A`. The coefficients are $c_1 = c_3 = \frac{1}{2}$ and $c_2 = -1$, and are thus different from zero.

Trace

The trace of a matrix $\text{tr}(\mathcal{A})$ is the sum of its diagonal elements:

$$\text{tr}(A) = \sum_{i=1}^{\min(n,p)} a_{ii}.$$

The trace of a scalar just equals the scalar itself. One obtains the trace in `R` by combining the functions `diag()` and `sum()`:

```
> A = matrix(1:12, nrow = 4, ncol = 3); A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> sum(diag(A))      # trace
[1] 18
```

The function `diag()` extracts the diagonal elements of a matrix, which are then summed by the function `sum()`.

Determinant

The formal definition of the determinant of a square matrix $\mathcal{A}_{(p \times p)}$ is

$$\det(\mathcal{A}) = \sum (-1)^{\phi_p(\tau_1,\ldots,\tau_p)} a_{1\,\tau_1} \ldots a_{p\,\tau_p}, \tag{2.2}$$

where $\phi_p(\tau_1, \ldots, \tau_p) = n_1 + \cdots + n_p$ and n_k represents the number of integers in the subsequence $\tau_{k+1}, \ldots, \tau_p$ that are smaller than τ_k. For a square matrix $\mathcal{A}_{(2\times 2)}$ of dimension two, (2.2) reduces to

$$\det(\mathcal{A}_{(2\times 2)}) = a_{1\,1}a_{2\,2} - a_{1\,2}a_{2\,1}.$$

The determinant is often useful for checking whether matrices are singular or regular. If the determinant is equal to 0, then the matrix is singular. Singular matrices can not be inverted, which limits some computations. In `R` the determinant is computed by the function `det()`:

```
> A = matrix(1:9, nrow = 3, ncol = 3); A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> det(A) # determinant
[1] 0
```

Thus, `A` is singular.

Transpose

A matrix $\mathcal{A}_{(n\times p)}$ has a transpose $\mathcal{A}^\top_{(n\times p)}$, which is obtained by reordering the elements of the original matrix. Formally, the transpose of $\mathcal{A}_{(n\times p)}$ is

$$\mathcal{A}^\top_{(n\times p)} = (a_{i\,j})^\top = (a_{j\,i}).$$

The resulting matrix has p rows and n columns. One has that

$$(\mathcal{A}^\top)^\top = \mathcal{A},$$
$$(\mathcal{A}\mathcal{B})^\top = \mathcal{B}^\top\mathcal{A}^\top.$$

`R` provides the function `t()` which returns the transpose of a matrix:

```
> A = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> t(A)   # transpose
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
[2,]    2    5    8
[3,]    3    6    9
```

When creating a matrix with a constructor `matrix`, its transpose can be created by setting the argument `byrow` to `FALSE`.

A good overview of special matrices and vectors is provided by Table 2.1 in Härdle and Simar (2015), Chap. 2. The same notations are used in this book.

Conjugate transpose

Every matrix $\mathcal{A}_{(n \times p)}$ has a conjugate transpose $\mathcal{A}^C_{p \times n}$. The elements of $\mathcal{A}$ can be complex numbers. If a matrix entry $a_{ij} = \alpha + \beta i$ is a complex number with real numbers α, β and imaginary unit $i^2 = -1$, then its conjugate is $a^C_{ij} = \alpha - \beta i$. The same holds in the other direction: if $a_{ij} = \alpha - \beta i$, the conjugate is $a^C_{ij} = \alpha + \beta i$. Therefore the conjugate transpose is

$$\mathcal{A}^C = \begin{pmatrix} a^C_{11} & a^C_{21} & \dots & a^C_{n1} \\ a^C_{12} & a^C_{22} & \dots & a^C_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a^C_{p1} & \dots & \dots & a^C_{np} \end{pmatrix}. \tag{2.3}$$

The function `Conj()` yields the conjugates of the elements. One can combine the functions `Conj()` and `t()` to get the conjugate transpose of a matrix. For $\mathcal{A} = \begin{pmatrix} 1+0.5 \cdot i & 1 & 1 \\ 1 & 1 & 1-0.5 \cdot i \end{pmatrix}$, the conjugate transpose is computed in `R` as follows:

```
> a  = c(1 + 0.5i, 1, 1, 1, 1, 1 - 0.5i)           # matrix entries
> A  = matrix(a, nrow = 2, ncol = 3, byrow = TRUE) # complex matrix
> A
       [,1] [,2]   [,3]
[1,] 1+0.5i 1+0i 1+0.0i
[2,] 1+0.0i 1+0i 1-0.5i
> AC = Conj(t(A))                                  # conjugate
> AC                                               # transpose
       [,1]   [,2]
[1,] 1-0.5i 1+0.0i
[2,] 1+0.0i 1+0.0i
[3,] 1+0.0i 1+0.5i
```

For a matrix with only real values, the conjugate transpose $\mathcal{A}^C$ is equal to the normal transpose $\mathcal{A}^\top$.

2.1.2 Matrix Operations

There are four fundamental operations in arithmetic: addition, subtraction, multiplication and division. In matrix algebra, there exist analogous operations: matrix addition, matrix subtraction, matrix multiplication and 'division'.

Basic operations

For matrices $\mathcal{A}_{(n \times p)}$ and $\mathcal{B}_{(n \times p)}$ of the same dimensions, matrix addition and subtraction work elementwise as follows:

$$\mathcal{A} + \mathcal{B} = (a_{i\,j} + b_{i\,j}),$$
$$\mathcal{A} - \mathcal{B} = (a_{i\,j} - b_{i\,j}).$$

These operations can be applied in R as shown below.

```
> A = matrix(3:11, nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
> B = matrix(-3:-11, nrow = 3, ncol = 3, byrow = TRUE); B
     [,1] [,2] [,3]
[1,]   -3   -4   -5
[2,]   -6   -7   -8
[3,]   -9  -10  -11
> A + B
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

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

```
> A = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> A %% 2  # modulo operation
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    0    1    0
[3,]    1    0    1
```

If one uses in R the elementary operations, including addition +, subtraction -, multiplication *, or division/between two matrices, they are all interpreted in R as elementwise operations.

Matrix multiplication returns the matrix product of the matrices $\mathcal{A}_{(n \times p)}$ and $\mathcal{B}_{(p \times m)}$, which is

$$\mathcal{A}_{(n\times p)}\cdot\mathcal{B}_{(p\times m)}=C_{(n\times m)}=\begin{pmatrix}\sum_{i=1}^{p}a_{1i}b_{i1} & \sum_{i=1}^{p}a_{1i}b_{i2} & \dots & \sum_{i=1}^{p}a_{1i}b_{im}\\ \sum_{i=1}^{p}a_{2i}b_{i1} & \sum_{i=1}^{p}a_{2i}b_{i2} & \dots & \sum_{i=1}^{p}a_{2i}b_{im}\\ \vdots & \vdots & \ddots & \vdots\\ \sum_{i=1}^{p}a_{ni}b_{i1} & \dots & \dots & \sum_{i=1}^{p}a_{ni}b_{im}\end{pmatrix}.$$

In R, one uses the operator `%*%` between two objects for matrix multiplication. The objects have to be of class `vector` or `matrix`.

```
> A = matrix(3:11, nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
> B = matrix(-3:-11, nrow = 3, ncol = 3, byrow = TRUE); B
     [,1] [,2] [,3]
[1,]   -3   -4   -5
[2,]   -6   -7   -8
[3,]   -9  -10  -11
> A %*% B # matrix multiplication
     [,1] [,2] [,3]
[1,]  -78  -90 -102
[2,] -132 -153 -174
[3,] -186 -216 -246
```

The number of columns of $\mathcal{A}$ has to equal the number of rows of $\mathcal{B}$.

Inverse

The division operation for square matrices is done by inverting a matrix. The inverse $\mathcal{A}^{-1}$ of a square matrix $\mathcal{A}_{(p\times p)}$ exists if $\det(\mathcal{A})\neq 0$:

$$\mathcal{A}^{-1}\mathcal{A}=\mathcal{A}\mathcal{A}^{-1}=\mathcal{I}_p. \tag{2.4}$$

The inverse of $\mathcal{A}=(a_{ij})$ can be calculated by

$$\mathcal{A}^{-1}=\frac{\mathcal{W}}{\det(\mathcal{A})},$$

where $\mathcal{W}=(w_{ij})$ is the adjoint matrix of $\mathcal{A}$. The elements of $\mathcal{W}$ are

$$(w_{ji})=(-1)^{i+j}\det\begin{pmatrix}a_{11} & \dots & a_{1(j-1)} & a_{1(j+1)} & \dots & a_{1p}\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ a_{(i-1)1} & \dots & a_{(i-1)(j-1)} & a_{(i-1)(j+1)} & \dots & a_{(i-1)p}\\ a_{(i+1)1} & \dots & a_{(i+1)(j-1)} & a_{(i+1)(j+1)} & \dots & a_{(i+1)p}\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ a_{p1} & \dots & a_{p(j-1)} & a_{p(j+1)} & \dots & a_{pp}\end{pmatrix},$$

which are the cofactors of $\mathcal{A}$. To compute the cofactors w_{ji}, one deletes column j and row i of $\mathcal{A}$, then computes the determinant for that reduced matrix, and then multiplies by 1 if $j+i$ is even or by -1 if it is odd. This computation is only feasible for small matrices.

Using the above definition, one can determine the inverse of a square matrix by solving the system of linear equations (see Sect. 2.4.1) in (2.4) by employing the function `solve(A, b)`. In `R` this function can be used to solve a general system of linear equations $\mathcal{A}x = b$. If one does not specify the right side `b` of the system of equations, the `solve()` function computes the inverse of the square matrix $\mathcal{A}$. The following code computes the inverse of the square matrix $\mathcal{A} = \begin{pmatrix} 1 & 2 & 5 \\ 3 & 9 & 2 \\ 2 & 2 & 2 \end{pmatrix}$.

```
> A = matrix(c(1, 2, 5, 3, 9, 2, 2, 2, 2),  # all elements
+   nrow = 3, ncol = 3, byrow = TRUE); A    # matrix dimensions
     [,1] [,2] [,3]
[1,]    1    2    5
[2,]    3    9    2
[3,]    2    2    2
> solve(A)                                  # inverse of A
      [,1]  [,2]  [,3]
[1,] -0.28 -0.12  0.82
[2,]  0.04  0.16 -0.26
[3,]  0.24 -0.04 -0.06
> A %*% solve(A)                            # check (2.4)
              [,1]          [,2]          [,3]
[1,]  1.000000e+00 5.551115e-17 -1.110223e-16
[2,] -5.551115e-17 1.000000e+00  8.326673e-17
[3,] -5.551115e-17 1.804112e-16  1.000000e+00
```

For diagonal matrices, the inverse $\mathcal{A}^{-1} = (a_{ii}^{-1})$ if $a_{ii} \neq 0$ for all i.

Generalised inverse

In practice, we are often confronted with singular matrices, whose determinant is equal to zero. In this situation, the inverse can be given by a generalised inverse $\mathcal{A}^-$ satisfying

$$\mathcal{A}\mathcal{A}^-\mathcal{A} = \mathcal{A}. \tag{2.5}$$

Consider the singular matrix $\mathcal{A} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$. Its inverse $\mathcal{A}^-$ must satisfy (2.5),

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \mathcal{A}^- \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}. \tag{2.6}$$

There are sometimes several $\mathcal{A}^-$ which satisfy (2.5). The Moore–Penrose generalised inverse (hereafter, just 'generalised inverse') is the most common type and was developed by Moore (1920) and Penrose (1955). It is used to compute the 'best fit' solution to a system of linear equations that does not have a unique solution. Another approach is to find the minimum (Euclidean) norm (see Sect. 2.1.5) solution to a system of linear equations with several solutions. The Moore–Penrose generalised

inverse is defined and unique for all matrices with real or complex entries. It can be computed using the singular value decomposition, see Press (1992).

In R the generalised inverse of a matrix defined in (2.5) can be computed with the function `ginv()` from the `MASS` package. With `ginv()` one obtains the generalised inverse of the matrix $\mathcal{A} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$, which is equal to $\mathcal{A}^- = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$.

```
> require(MASS)
> A = matrix(c(1, 0, 0, 0),
+   ncol = 2, nrow = 2); A        # matrix from (2.6)
     [,1] [,2]
[1,]    1    0
[2,]    0    0
> ginv(A)                         # generalised inverse
     [,1] [,2]
[1,]    1    0
[2,]    0    0
```

The `ginv()` function can also be used for non-square matrices, like $\mathcal{A} = \left(\begin{smallmatrix} 1 & 2 & 3 \\ 11 & 12 & 13 \end{smallmatrix}\right)$.

```
> require(MASS)
> A = matrix(c(1, 2, 3, 11, 12, 13),
+   nrow = 2, ncol = 3, byrow = TRUE); A # non-square matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
> A.ginv = ginv(A)                      # generalised inverse
            [,1]        [,2]
[1,] -0.63333333  0.13333333
[2,] -0.03333333  0.03333333
[3,]  0.56666667 -0.06666667
```

The condition (2.5) can be verified in R by the following code:

```
> A %*% A.ginv %*% A              # first condition
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
```

This code shows that the solution for A^- fulfills the condition $\mathcal{A}\mathcal{A}^-\mathcal{A} = \mathcal{A}$.

2.1.3 *Eigenvalues and Eigenvectors*

For a given basis of a vector space, a matrix $\mathcal{A}_{(p\times p)}$ can represent a linear function of a p-dimensional vector space to itself. If this function is applied to a nonzero vector and maps that vector to a multiple of itself, that vector is called an eigenvector γ and the multiple is called the corresponding eigenvalue λ. Formally this can be written as

$$\mathcal{A}\gamma = \lambda\gamma.$$

In Chap. 8, the theory of eigenvectors, eigenvalues and spectral decomposition, presented in Sect. 2.1.4, becomes important in order to understand the *principal components analysis*, the *factor analysis* and other methods of dimension reduction. Definition 2.1 provides a formal description of an eigenvalue and the corresponding eigenvector.

Definition 2.1 Let V be any real vector space and $L : V \to V$ be a linear transformation. Then a nonzero vector $\gamma \in V$ is called an eigenvector if, and only if, there exists a scalar $\lambda \in \mathbb{R}$ such that $L(\gamma) = \mathcal{A}\gamma = \lambda\gamma$. Therefore

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

If we consider λ as an unknown variable, then $\det(\mathcal{A} - \lambda\mathcal{I})$ is a polynomial of degree n in λ, and is called the characteristic polynomial of $\mathcal{A}$. Its coefficients can be computed from the matrix entries of $\mathcal{A}$. Its roots $\lambda_1, \ldots, \lambda_p$, which might be complex, are the eigenvalues of $\mathcal{A}$. The eigenvalue matrix Λ is a diagonal matrix with elements $\lambda_1, \ldots, \lambda_p$. Vectors $\gamma_1, \ldots, \gamma_p$ are the eigenvectors, that correspond to eigenvalues $\lambda_1, \ldots, \lambda_p$. The eigenvector matrix $\mathcal{P}$ has columns $\gamma_1, \ldots, \gamma_p$. In the following example, the computation of the eigenvalues of a matrix of dimension 3 is shown.

Example 2.1 Consider the matrix $\mathcal{A} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 3 & 1 \\ 0 & 6 & 2 \end{pmatrix}$ and the matrix $\mathcal{D} = \mathcal{A} - \lambda\mathcal{I}$.

In order to obtain the eigenvalues of $\mathcal{A}$ one has to solve for the roots λ of the polynomial $\det(\mathcal{D}) = 0$. For a three-dimensional matrix this looks like

$$\begin{aligned}
\det(\mathcal{D}) &= c_0 + c_1\lambda + c_2\lambda^2 + c_3\lambda^3, \\
c_0 &= \det(\mathcal{A}), \\
c_1 &= \sum_{1 \le i \ne j \le 3} a_{ii}a_{jj} - \sum_{1 \le i \ne j \le 3} a_{ij}a_{ji}, \\
c_2 &= \sum_{1 \le i \le 3} a_{ii}, \\
c_3 &= -1.
\end{aligned}$$

Thus, the characteristic polynomial of $\mathcal{A}$ is $-\lambda^3 + 7\lambda^2 + 10\lambda$. In this case, $\mathcal{A}$ is singular: the intercept of the polynomial is equal to zero. Therefore, one eigenvalue is 0 and the other two are 2 and 5.

In R the eigenvalues and eigenvectors of a matrix $\mathcal{A}$ can be calculated using the function `eigen()`.

```
> A = matrix(c(2, 0, 1, 0, 3, 1, 0, 6, 2), # matrix A
+   nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    2    0    1
[2,]    0    3    1
[3,]    0    6    2
> Eigen = eigen(A)                          # eigenvectors and -values
```

```
> Eigen$values                              # eigenvalues
[1] 5 2 0
> Eigen$vectors                             # eigenvector matrix
          [,1] [,2]       [,3]
[1,] 0.2857143    1 -0.4285714
[2,] 0.4285714    0 -0.2857143
[3,] 0.8571429    0  0.8571429
```

Let $\gamma_2 = (1, 0, 0)^\top$ be the second column of the eigenvector matrix $\mathcal{P} = (\gamma_1, \gamma_2, \gamma_3)$. Then it can be seen that

$$\mathcal{A}\gamma_2 = 2\gamma_2.$$

This means that γ_2 is the eigenvector corresponding to the eigenvalue $\lambda_2 = 2$ of $\mathcal{A}$.

After the eigenvalues of a matrix are found, it is easy to compute its trace or determinant. The sum of all its eigenvalues is its trace: $\text{tr}(\mathcal{A}) = \sum_{i=1}^{p} \lambda_i$. The product of its eigenvalues is its determinant: $\det(\mathcal{A}) = \prod_{i=1}^{p} \lambda_i$.

2.1.4 Spectral Decomposition

The spectral decomposition of a matrix is a representation of that matrix in terms of its eigenvalues and eigenvectors.

Theorem 2.1 *Let $A_{(p\times p)}$ be a matrix with real entries and let Λ be its eigenvalue matrix and $\mathcal{P}$ the corresponding eigenvector matrix. Then*

$$\mathcal{A} = \mathcal{P}\Lambda\mathcal{P}^{-1}. \tag{2.7}$$

In R, one can use the function `eigen()` to compute eigenvalues and eigenvectors. The eigenvalues are in the field named `values` and are sorted in decreasing order (see the example above). 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 $\mathcal{P}$:

```
> A = matrix(c(2, 0, 1, 0, 3, 1, 0, 6, 2),  # matrix A
+   nrow = 3, ncol = 3, byrow = TRUE); A
     [,1] [,2] [,3]
[1,]    2    0    1
[2,]    0    3    1
[3,]    0    6    2
> Eigen = eigen(A)                          # eigenvectors and -values
> P     = eigen(A)$vectors                  # eigenvector matrix
> L     = diag(eigen(A)$values)             # eigenvalue matrix
> qr(P)$rank                                # rank of P
[1] 3
> P %*% L %*% solve(P)                      # spectral decomposition
     [,1]         [,2] [,3]
[1,]    2 4.440892e-16    1
[2,]    0 3.000000e+00    1
[3,]    0 6.000000e+00    2
```

From this computation, it can be seen that $\mathcal{P}$ has full rank. The diagonal matrix can be obtained by extracting the eigenvalues from the output of the function `eigen()`. It is possible to decompose the matrix $\mathcal{A}$ by (2.7) in R. The difference between $\mathcal{A}$ and the result from the spectral decomposition in R is negligibly small.

2.1.5 Norm

There are two types of frequently used norms: the vector norm and the matrix norm. The vector norm, which appears frequently in matrix algebra and numerical computation, will be introduced first. An extension of the vector norm is the matrix norm.

Definition 2.2 Let V be a vector space and b be a scalar, both lying either in $\mathbb{R}^n$ or $\mathbb{C}^n$. Consider the vectors $x, y \in V$. Then a norm is a mapping $\|\cdot\| : V \to \mathbb{R}_0^+$ with the following properties:

1. $\|bx\| = |b|\|x\|$,
2. $\|x + y\| \leq \|x\| + \|y\|$,
3. $\|x\| \geq 0$, where $\|x\| = 0$ if and only if $x = 0$.

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

$$\|x\|_k = \left(\sum_{i=1}^{n} |x_i|^k \right)^{1/k}.$$

There are several special norms, depending on the value of k, some are listed below.

$$\text{Manhattan norm} : \; ||x||_1 = \sum_{i=1}^{n} |x_i|, \tag{2.8}$$

$$\text{Euclidean norm} : \; ||x||_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} = \sqrt{x^\top x}, \tag{2.9}$$

$$\text{infinity norm} : \; ||x||_\infty = \max\{|x_i|_{i=1}^n\}, \tag{2.10}$$

$$\text{Frobenius norm} : \; ||x||_F = \sqrt{x^\top x}. \tag{2.11}$$

The most frequently used norms are the Manhattan and Euclidean norms. For vector norms, the Euclidean and Frobenius norm coincide. The infinity norm selects the maximum absolute value of the elements of x and the maximum norm just the maximum value.

In R the function `norm()` can return the norms from (2.8) to (2.11). The argument `type` specifies which norm is returned.

```
> x = matrix(c(2, 1, 2), nrow = 3, ncol = 1) # vector x
> norm(x, type = c("O"))                     # Manhattan norm
[1] 5
> norm(x, type = c("2"))                     # Euclidean norm
[1] 3
> norm(x, type = c("I"))                     # infinity norm
[1] 2
> norm(x, type = c("F"))                     # Frobenius norm
[1] 3
```

The object `x` has to be of class `matrix` in `R` to compute all norms.

Definition 2.3 Let $U^{n\times p}$ be a set of $(n \times p)$ matrices and a be a scalar, which are either real or complex. $U^{n\times p}$ is a vector space equipped with matrix addition and scalar multiplication. Let $\mathcal{A}, \mathcal{B} \in U^{n\times p}$. Then a matrix norm is a mapping $\|\cdot\| : U^{n\times p} \to \mathbb{R}_0^+$ with the following properties:

1. $\|a\mathcal{A}\| = |a|\|\mathcal{A}\|$,
2. $\|\mathcal{A} + \mathcal{B}\| \leq \|\mathcal{A}\| + \|\mathcal{B}\|$,
3. $\|\mathcal{A}\| \geq 0$, where $\|\mathcal{A}\| = 0$ if and only if $\mathcal{A} = 0$.

In `R`, the function `norm()` can be applied to vectors and matrices in the same fashion. The one norm, the infinity norm, the Frobenius norm, the maximum norm and the spectral norm for matrices are represented by

$$\begin{aligned}
\text{one norm}: \quad ||\mathcal{A}||_1 &= \max_{1\leq j\leq p} \sum_{i=1}^{n} |a_{ij}|,\\
\text{spectral/Euclidean norm}: \quad ||\mathcal{A}||_2 &= \sqrt{\lambda_{max}(\mathcal{A}^C\mathcal{A})},\\
\text{infinity norm}: \quad ||\mathcal{A}||_\infty &= \max_{1\leq i\leq n} \sum_{j=1}^{p} |a_{ij}|,\\
\text{Frobenius norm}: \quad ||\mathcal{A}||_F &= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{p} |a_{ij}|^2},
\end{aligned}$$

where $\mathcal{A}^C$ is the conjugate matrix of $\mathcal{A}$. The next code shows how to compute these five norms with the function `norm()` for the matrix $\mathcal{A} = \left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$.

```
> A = matrix(c(1, 2, 3, 4),
+   ncol = 2, nrow = 2, byrow = TRUE)     # matrix A
> norm(A, type = c("O"))                  # one norm
[1] 6                                     # maximum of column sums
> norm(A, type = c("2"))                  # Euclidean norm
[1] 5.464986
> norm(A, type = c("I"))                  # infinity norm
[1] 7                                     # maximum of row sums
> norm(A, type = c("F"))                  # Frobenius norm
[1] 5.477226
```

Note that the Frobenius norm returns the square root of the trace of the matrix product of the matrix with its conjugate transpose, $\sqrt{\text{tr}(\mathcal{A}^C\mathcal{A})}$. The spectral norm or Euclidean norm returns the square root of the maximum eigenvalue of $\mathcal{A}^C\mathcal{A}$.

2.2 Numerical Integration

This section discusses numerical methods in R for integrating a function. Some integrals cannot be computed analytically and numerical methods should be used.

2.2.1 Integration of Functions of One Variable

Not every function $f \in C[a, b]$ has an indefinite integral with an analytical representation. Therefore, it is not always possible to analytically compute the area under a curve. An important example is

$$\int \exp(-x^2)dx. \tag{2.12}$$

There exists no analytical, closed-form representation of (2.12). Therefore, the corresponding definite integral has to be computed numerically using numerical integration, also called 'quadrature'. The basic idea behind numerical integration lies in approximating the function by a polynomial and subsequently integrating it using the Newton–Cotes rule.

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 looks for a polynomial $p_n(x) = \sum_{j=1}^{n} c_j x^{j-1} \in P_n$, where P_n are basis polynomials, satisfying the condition

$$p_n(x_k) = f(x_k), \text{ for all } k \in \{0, \ldots, n\}. \tag{2.13}$$

To construct a polynomial that satisfies this condition, the following basis polynomials are used:

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

This leads to the so-called *Lagrange polynomial*, which satisfies the condition in (2.13) (assuming $\frac{0}{0} = 1$).

$$p_n(x) = \sum_{k=0}^{n} f(x_k)L_k(x). \tag{2.14}$$

Let $I(f) = \int_a^b f(x)dx$ be the exact integration operator applied to a function $f \in C[a, b]$. Then define $I_n(f)$ as the approximation of $I(f)$ using (2.14) as an approximation for f:

$$I_n(f) = \int_a^b p_n(x)\,dx,$$

which can be restated using weights for the different values of f:

$$I_n(f) = (b-a)\sum_{k=0}^{n} f(x_k)\alpha_k, \tag{2.15}$$

$$\text{with weights} \quad \alpha_k = \frac{1}{b-a}\int_a^b L_k(x)\,dx.$$

By construction, (2.15) is exact for every $f \in P_n$. Suppose the nodes x_k are equidistant in $[a, b]$, i.e., $x_k = a + kh$, where $h = (b-a)n^{-1}$. Then (2.15) is the (closed) *Newton–Cotes rule*. The weights α_k can be explicitly computed up to $n = 7$. Starting from $n = 8$, negative weights occur and the Newton–Cotes rule can no longer be applied. The *trapezoidal rule* is an example of the Newton–Cotes rule.

Example 2.2 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:

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

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 by the formula for the area of a trapezoid:

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

In `R`, the trapezoidal rule is implemented within the package `caTools`. There the function `trapz(x, y)` is used with a sorted vector `x` that contains the x-axis values and a vector `y` with the corresponding y-axis values. This function uses a summed version of the trapezoidal rule, where $[a, b]$ is split into n equidistant intervals. For all $k = \{1, \ldots, n\}$, the integral $I_k(f)$ is computed according to the trapezoidal rule: this is the so-called *extended trapezoidal rule*.

$$I_k(f) = \frac{b-a}{2n}\left[f\{a + (k-1)n^{-1}(b-a)\} + f\{a + kn^{-1}(b-a)\}\right].$$

Therefore, the whole integral $I(f)$ is approximated by

$$I(f) \approx \sum_{k=1}^{n} I_k(f) = \sum_{k=1}^{n} \frac{b-a}{2n}\left[f\{a + (k-1)n^{-1}(b-a)\} + f\{a + kn^{-1}(b-a)\}\right].$$

For example, consider the integral of the cosine function on $[-\frac{\pi}{2}, \frac{\pi}{2}]$ and split the interval into 10 subintervals, where the trapezoidal rule is applied:

```
> require(caTools)
> x      = (-5:5) * (pi / 2) / 5     # set subintervals
> intcos = trapz(x, cos(x)); intcos  # integration
[1] 1.983524
> abs(intcos - 2)                    # absolute error
[1] 0.01647646
```

The integral of the cosine function on $[-\frac{\pi}{2}, \frac{\pi}{2}]$ is supposed to be exactly 2, so the absolute error is almost 0.02. It can be shown that the error of the trapezoidal rule is of order $\mathcal{O}(h^3 \| f'' \|^2)$ with an unknown second derivative of the function. If a Newton–Cotes rule with more nodes is used, the integrand will be approximated by 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.

Gaussian quadrature

In `R`, the function `integrate()` uses an integration method that is based on Gaussian quadrature (the exact method is called the *Gauss–Kronrod quadrature*, see Press (1992) for further details). The Gaussian method uses non-predetermined nodes $x_1, \ldots, x_n$ to approximate the integral, so that polynomials of higher order can be integrated more precisely than using the Newton–Cotes rule. For n nodes, it uses a polynomial $p(x) = \sum_{j=1}^{2n} c_j x^{j-1}$ of order $2n - 1$.

Definition 2.4 A method of numerical integration for a function $f : [a, b] \to \mathbb{R}$ with the formula

$$I(f) = \int_a^b f(x)dx \approx \int_a^b w(x)p(x)dx \approx I_n^G(f) = \sum_{k=1}^{n} f(x_k)\alpha_k,$$

and n nodes $x_1, \ldots, x_n$ is called Gaussian quadrature if an arbitrary weighting function $w(x)$ and a polynomial $p(x)$ of degree $2n - 1$ exactly approximate $f(x)$, such that $f(x) = w(x)p(x)$.

Consider the simplest case, where $w(x) = 1$. Then the method of undetermined coefficients leads to the following nonlinear system of equations:

$$\frac{b^j - a^j}{j!} = \sum_{k=1}^{n} \alpha_k x_k^{j-1}, \text{ for } j = 1, \ldots, 2n.$$

A total of $2n$ equations are used to find the nodes $x_1, \ldots, x_n$ and the coefficients $\alpha_1, \ldots, \alpha_n$.

Consider the special case with two nodes (x_1, x_2) and two weights (α_1, α_2). The particular polynomial $p(x)$ is of order $2 \cdot n - 1 = 3$, where the number of nodes is n. The integral is approximated by $\alpha_1 f(x_1) + \alpha_2 f(x_2)$ and it is assumed that $f(x) = p(x)$. Therefore the following two equations can be derived:

$$I_2^G(f) = c_1(\alpha_1 + \alpha_2) + c_2(\alpha_1 x_1 + \alpha_2 x_2) + c_3(\alpha_1 x_1^2 + \alpha_2 x_2^2) + c_4(\alpha_1 x_1^3 + \alpha_2 x_2^3),$$
$$I_2^G(f) = c_1(b-a) + c_2\Big(\frac{b^2-a^2}{2}\Big) + c_3\Big(\frac{b^3-a^3}{3}\Big) + c_4\Big(\frac{b^4-a^4}{4}\Big).$$

All coefficients c_j of the polynomial are set equal to each other, because the coefficients are arbitrary. The system of four nonlinear equations is

$$\begin{aligned} b-a &= \alpha_1 + \alpha_2; \\ 1/2 \cdot (b^2-a^2) &= \alpha_1 x_1 + \alpha_2 x_2; \\ 1/3 \cdot (b^3-a^3) &= \alpha_1 x_1^2 + \alpha_2 x_2^3; \\ 1/4 \cdot (b^4-a^4) &= \alpha_1 x_1^3 + \alpha_2 x_2^3. \end{aligned}$$

For simplicity, in most cases the interval $[-1, 1]$ is considered. It is possible to extend these results to the more general interval $[a, b]$. To apply the results for $[-1, 1]$ to the interval $[a, b]$, one uses

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f\Big(\frac{b-a}{2}x + \frac{a+b}{2}\Big)dx.$$

For the special case $w(x) = 1$ and the interval $[-1, 1]$, the procedure is called *Gauss–Legendre quadrature*. The nodes are the roots of the *Legendre polynomials* $P_n(x) = \frac{1}{2n \cdot n!}\frac{d^n}{dx^n}\{(x^2-1)^n\}$. The weights α_k can be calculated by

$$\alpha_k = \frac{2}{(1-x_k^2)\{P_n'(x_k)\}^2}.$$

In the following example, we illustrate the process of numerical integration using the function `integrate()`. One can specify the following arguments: `f` (integrand), `a` (lower limit) and `b` (upper limit), `subdivisions` (number of subintervals) and arguments `rel.tol`, as well as `abs.tol` for the relative and absolute accuracy requested. Consider again the cosine function on the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

```
> require(stats)
> integrate(cos,      # integrand
+   lower = -pi / 2, # lower integration limit
+   upper = pi / 2)  # upper integration limit
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 this example, the absolute error is smaller than $2.2 \cdot 10^{-14}$. Therefore, the `integrate()` function is much more accurate for the cosine function than the `trapz()` function used in a previous example.

2.2.2 Integration of Functions of Several Variables

Repeated quadrature method

Similar to numerical integration in the context of one variable, in the case of more variables, an integration can be expressed as follows:

$$\int_{a_1}^{b_1} \dots \int_{a_p}^{b_p} f(x_1, \dots, x_n) dx_1 \dots dx_p \approx \sum_{i_1=1}^{n} \cdots \sum_{i_p=1}^{n} W_{i_1} \cdots W_{i_p} f(x_{i_1}, \dots, x_{i_p}), \tag{2.16}$$

where $D_j = [a_j,\ b_j],\ j \in \{1, \dots, p\}$ is an integration region in $\mathbb{R}$ and $(x_{i_1}, \dots, x_{i_p})$ is the p-dimensional point at the i-th dimension, where $i_j \in \{1, \dots, n\}$, and W_{i_j} is the coefficient used as the weight. The problem with the repeated quadrature is that for (2.16) one needs to evaluate p^n terms, which may lead to computational difficulties.

Adaptive method

The adaptive method in the context of multiple integrals divides the integration region $D \in \mathbb{R}^p$ into subregions $S_j \in \mathbb{R}^p$. For each subregion S_j, specific rules are applied to approximate the integral. To improve the approximation, consider the error E_j for each subregion. If the overall error $\sum_j E_j$ is smaller than a predefined tolerance level, the algorithm stops. But if this condition is not met, the highest error is selected and the corresponding region is split into two subregions. Then the rules are applied again to each subregion until the tolerance level is met. For a more detailed description of this algorithm, see van Dooren and de Ridder (1976) and Stroud (1971).

Jarle Berntsen and Genz (1991) improved the reliability of the algorithm, including a strategy for the selection of subregions, error estimation and parallelisation of the computation.

Example 2.3 Integrate the function of two variables

$$\int_0^1 \int_0^1 x^2 y^3 dx dy. \tag{2.17}$$

Analytical computation of this integral yields $1/12$. The surface $z = x^2 y^3$ for the interval $[0, 1]^2$ is depicted in Fig. 2.1. For the computation of multiple integrals, the `R` package `R2Cuba` is used, which is introduced in Hahn (2013). It includes four different algorithms for multivariate integration, where the function `cuhre` uses the adaptive method.

```
> require(R2Cuba)
> integrand = function(arg){          # construct the integrand
+   x = arg[1]                        # function argument x
+   y = arg[2]                        # function argument y
+   (x^2) * (y^3)                     # function
+ }
> cuhre(integrand,                    # adaptive method
+   ncomp   = 1,                      # number of components
+   ndim    = 2,                      # dimension of the integral
```

```
+    lower    = rep(0, 2),              # lower integration bound
+    upper    = rep(1, 2),              # upper integration bound
+    rel.tol = 1e-3,                    # relative tolerance level
+    abs.tol = 1e-12,                   # absolute tolerance level
+    flags    = list(verbose = 1))      # controls output
Iteration 1:  65 integrand evaluations so far
[1] 0.0833333 +- 1.15907e-015   chisq 0 (0 df)
Iteration 2:  195 integrand evaluations so far
[1] 0.0833333 +- 1.04612e-015   chisq 0.10352e-05 (1 df)
integral: 0.08333333 (+-1.3e-15)
nregions: 2; number of evaluations:  195; probability: 0.00623341
```

The output shows that the adaptive algorithm carried out two iteration steps. Only two subregions have been used for the computation, which is stated by the output value `nregions`. The output value `neval` states that the number of evaluations is 195. To make a statement about the reliability of the process, consider the `probability` value. A probability of 0 for the χ^2 distribution (see Sect. 4.4.1) means that the null

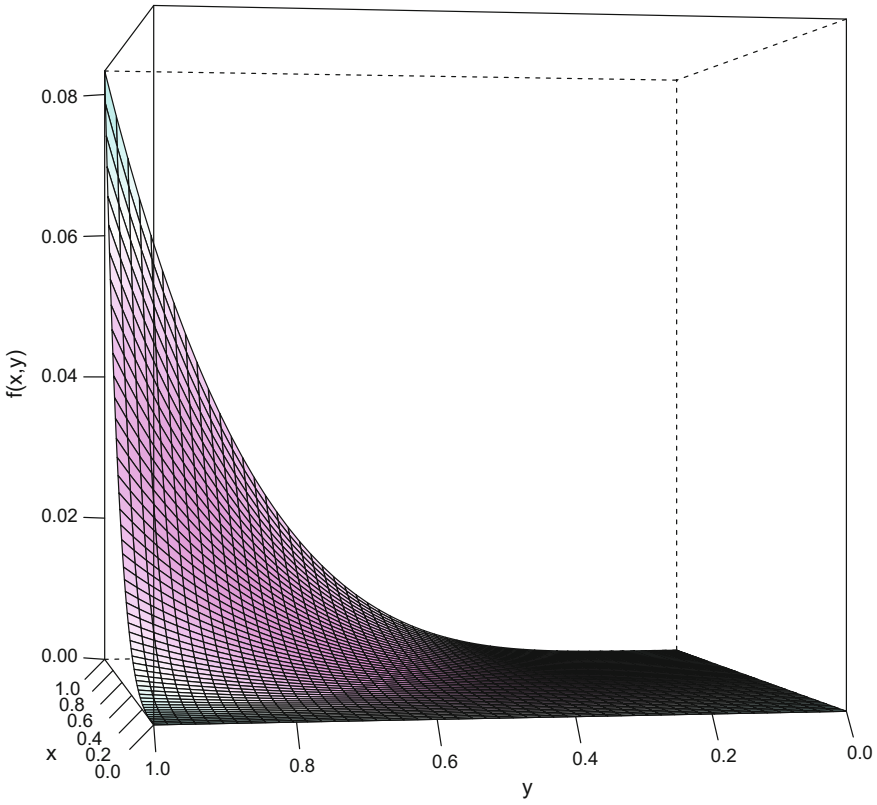

Fig. 2.1 Plot of the multivariate function (2.17). BCS_Integrand

hypothesis can be rejected. The null hypothesis states that the absolute error estimate is not a reliable estimate of the true integration error. The approximation of integral I is 0.08333, which is close to the result of the analytical computation, $\frac{1}{12}$. For a more detailed discussion of the output, refer to Hahn (2013).

Example 2.4 Evaluate the integral with three variables

$$\int_0^1 \int_0^1 \int_0^1 \sin(x) \log(1+2y) \exp(3z) dx dy dz, \tag{2.18}$$

```
> require(R2Cuba)
> integrand = function(arg){              # construct the integrand
+   x = arg[1]                            # function argument x
+   y = arg[2]                            # function argument y
+   z = arg[3]                            # function argument z
+   sin(x) * log(1 + 2 * y) * exp(3 * z)  # function
+ }
> cuhre(integrand,                        # adaptive method
+   ncomp   = 1,                          # number of components
+   ndim    = 3,                          # dimension of the integral
+   lower   = rep(0, 3),                  # lower bound of interval
+   upper   = rep(1, 3),                  # upper bound of interval
+   rel.tol = 1e-3,                       # relative tolerance level
+   abs.tol = 1e-12,                      # absolute tolerance level
+   flags   = list(verbose = 0))          # controls output
integral: 1.894854 (+-4.1e-07)
nregions: 2; number of evaluations:  381; probability:  0.04784836
```

For the function of three variables (2.18), an analytical computation yields the value:

$$\int_0^1 \sin(x) dx \int_0^1 \log(1+2y) dy \int_0^1 \exp(3z) dz$$
$$= \{1 - \cos(1)\} \frac{1}{2} [3\{\log(3) - 1\} + 1] \frac{1}{3} \{\exp(3) - 1\} = 1.89485.$$

The value provided by the adaptive method is very close to the exact value.

Monte Carlo method

For a multiple integral I of the function of p variables $f(x_1, \ldots, x_p)$ with lower bounds $a_1, \ldots, a_p$ and upper bounds $b_1, \ldots, b_p$, the integral is given by

$$I(f) = \int_{a_1}^{b_1} \ldots \int_{a_p}^{b_p} f(x_1, \ldots, x_p) dx_1 \ldots dx_p = \int \cdots \int_D f(x) dx,$$

where x stands for a vector $(x_1, \ldots, x_p)$ and D for the integration region. Let X be a random vector (see Chap. 6), with each component X_j of X uniformly distributed (Sect. 4.2) in $[a_j, b_j]$. Then the algorithm of Monte Carlo multiple integration can be described as follows. In the first step, n points of dimension p are randomly drawn from the region D, such that

$$(x_{1_1}, \ldots, x_{1_p}), \ldots, (x_{n_1}, \ldots, x_{n_p}).$$

In the second step, the p-dimensional volume is estimated by $V = \prod_{j=1}^{p}(b_j - a_j)$ and the integrand f is evaluated for all n points. In the third step, the integral I can be estimated using a sample moment function,

$$I(f) \approx \hat{I}(f) = n^{-1} V \sum_{i=1}^{n} f(x_{i_1}, \ldots, x_{i_p}).$$

The absolute error ϵ can be approximated as follows:

$$\epsilon = |(I(f) - \hat{I}(f)| \approx n^{-1/2}\{V I(f^2) - I^2(f)\}.$$

The Monte Carlo method is applied to example (2.17) via the function `vegas`.

```
> require(R2Cuba)
> integrand = function(arg){              # construct the integrand
+   x = arg[1]                            # function argument x
+   y = arg[2]                            # function argument y
+   (x^2) * (y^3)                         # function
+ }
> vegas(integrand,                        # Monte Carlo method
+   ncomp   = 1,                          # number of components
+   ndim    = 2,                          # dimension of the integral
+   lower   = rep(0, 2),                  # lower integration bound
+   upper   = rep(1, 2),                  # upper integration bound
+   rel.tol = 1e-3,                       # relative tolerance level
+   abs.tol = 1e-12,                      # absolute tolerance level
+   flags   = list(verbose = 0))          # controls output
integral: 0.08329357 (+-7.5e-05)
number of evaluations:  17500; probability:  0.1201993
```

The outputs of the functions `vegas` and `cuhre` are almost identical. Additional output information can be obtained by setting the argument `verbose` to one. Then the output shows that the Monte Carlo algorithm executed 7 iterations and 17 500 evaluations of the integrand. The approximation of integral I is 0.0832, which is close to the exact value $\frac{1}{12}$. For the function (2.18) the Monte Carlo algorithm looks as follows:

```
> require(R2Cuba)
> integrand = function(arg){              # construct the integrand
+   x = arg[1]                            # function argument x
+   y = arg[2]                            # function argument y
+   z = arg[3]                            # function argument z
+   sin(x) * log(1 + 2 * y) * exp(3 * z)  # function
+ }
> vegas(integrand,                        # Monte Carlo method
+   ncomp   = 1,                          # number of components
+   ndim    = 3,                          # dimension of the integral
+   lower   = rep(0, 3),                  # lower integration bound
+   upper   = rep(1, 3),                  # upper integration bound
+   rel.tol = 1e-3,                       # relative tolerance level
+   abs.tol = 1e-12,                      # absolute tolerance level
+   flags   = list(verbose = 0))          # controls output
```

```
integral: 1.894488 (+-0.0016)
number of evaluations:  13500; probability:  0.1099108
```

The performance of the adaptive method is again superior to that of the Monte Carlo method, which gives 1.894488 as the value of the integral.

2.3 Differentiation

The analytical computation of derivatives may be impossible if the function is only given indirectly (for example by an algorithm) and can be evaluated only point-wise. Therefore, it is necessary to use numerical methods. Before presenting some numerical methods for differentiation, first it will be shown how to compute analytically the derivative in R.

2.3.1 Analytical Differentiation

To calculate the derivative of a one variable 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 the derivative will be computed. Consider the following example:

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

The function `D()` returns an argument of type `call` (see `help(call)` for further information) and one can therefore recursively 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")# warning message
+   if(order == 1) D(expr, name)              # first derivative
+   else DD(D(expr, name), name, order - 1)   # 1st derivative of DD
+ }
```

This function replaces the initial function with its first derivative until the argument order is reduced to one. Then the third derivative for $3x^3 + x^2$ can be computed with this function.

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

The gradient of a function can also be computed using the function `D()`.

Definition 2.5 Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function and $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$. Then the vector

$$\nabla f(x) \stackrel{\text{def}}{=} \left\{ \frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right\}^\top$$

is called the *gradient* of f at point x.

If f maps its arguments to a multidimensional space, then we consider the Jacobian matrix as a generalisation of the gradient.

Definition 2.6 Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable function with $F = (f_1, \ldots, f_m)$ and the coordinates $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$. Then the *Jacobian matrix* at a point $x \in \mathbb{R}^n$ is defined as follows:

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

If one is interested in the second derivatives of a function, the Hessian matrix is important.

Definition 2.7 Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice continuously differentiable. Then the *Hessian matrix* of f at a point x is defined as follows:

$$H_f(x) \stackrel{\text{def}}{=} \left\{ \frac{\partial^2 f}{\partial x_i \partial x_j}(x) \right\}_{i,j=1,\ldots,n}. \tag{2.19}$$

Now consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ that maps x_1 and x_2 coordinates to the square of their Euclidean norm.

```
> f    = expression(x^2 + y^2)            # function
> grad = c(D(f,"x"), D(f,"y"))            # gradient vector
> grad
[[1]]
2 * x

[[2]]
2 * y
```

If it is necessary to have the gradient as a function that can be evaluated, the function `deriv(f, name, function.arg = NULL, hessian = FALSE)` should be used. The function argument `f` is the function (as an object of mode `expression`) and the argument `name` identifies the vector with respect to which the derivative will be computed. Furthermore the arguments `function.arg` specify 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 example of the use of the function `deriv()`, consider the above function `f`.

```
> eucld2 = deriv(f,
+    name = c("x","y"),        # variable names +
function.arg = c("x","y"))    # arguments for a function return
> eucld2(2, 2)
[1] 8
attr(,"gradient")
     x y
[1,] 4 4
```

The function `eucld2(x, y)` delivers the 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)` can be retrieved through the call `attr(eucld(2, 2),"hessian")`.

2.3.2 Numerical Differentiation

To develop numerical methods for determining the derivatives of a function at a point x, one uses the Taylor expansion

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(x) + \mathcal{O}(h^3). \tag{2.20}$$

Only if the fourth derivative of f exists and f is bounded on $[x, x+h]$ the representation in (2.20) is valid. If the Taylor expansion is truncated after the linear term, then (2.20) can be solved for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{2.21}$$

Therefore an approximation for the derivative at point x could be

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2.22}$$

Another more accurate method uses the Richardson (1911) extrapolation. Redefine the expression in (2.22) with $g(h) = \frac{f(x+h)-f(x)}{h}$. Then (2.21) can be written as

$$f'(x) = g(h) + k_1 h + k_2 h^2 + k_3 h^3 + \cdots, \tag{2.23}$$

where $k_1, k_2, k_3, \ldots$ represent constant terms involving the derivatives at point x. Taylor's theorem holds for all positive h, one therefore can replace h by $h/2$:

$$f'(x) = g\left(\frac{h}{2}\right) + k_1 \frac{h}{2} + k_2 \frac{h^2}{4} + k_3 \frac{h^3}{8} + \cdots. \tag{2.24}$$

Now (2.23) can be subtracted from (2.24) times two. Then the term involving k_1 is eliminated:

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

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

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

This process can be continued to obtain formulae of higher order. In `R`, the package `numDeriv` provides some functions that use these methods to differentiate a function numerically. For example, the function `grad()` calculates a numerical approximation to the gradient of `func` at the point `x`. The argument `method` can be `"simple"` or `"Richardson"`. If the `method` argument is `simple`, a formula as in (2.22) is applied. Then only the element `eps` of `methods.args` is used (equivalent to the above h in (2.22)). The method `"Richardson"` uses the Richardson extrapolation. Consider the function $f(x_1, x_2, x_3) = \sqrt{x_1^2 + x_2^2 + x_3^2}$, which has the gradient

$$\nabla f(x) = \left(\frac{x_1}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_2}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_3}{\sqrt{x_1^2 + x_2^2 + x_3^2}}\right)^\top.$$

The gradient of f represents the normalised 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 calculated by

```
> require(numDeriv)
> f = function(x){sqrt(sum(x^2))}
> grad(f,
+   x      = c(1,0,0),    # point at which to compute the gradient
+   method ="Richardson") # method to use for the approximation
[1] 1 0 0
> grad(f,
+   x      = c(1,0,0),    # point at which to compute the gradient
+   method ="simple")     # method to use for the approximation
[1] 1e+00 5e-05 5e-05
```

It could also be interesting to compute numerically the Jacobian or the Hessian matrix of a function $F : \mathbb{R}^n \to \mathbb{R}^m$.

In `R`, the function `jacobian(func,x,...)` can be used to compute the Jacobian matrix of 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 $f(x) = \{\sin(x_1 + x_2),\ \cos(x_1 + x_2)\}$ at the point $(0, 2\pi)$ is computed:

```
> require(numDeriv)
> f1 = function(x){c(sin(sum(x)), cos(sum(x)))}
> jacobian(f1, x = c(0, 2 * pi))
     [,1] [,2]
[1,]    1    1
[2,]    0    0
```

The Hessian matrix is symmetric and can be computed in `R` with `hessian(func,x,...)`. For example, consider $f(x_1, x_2, x_3) = \sqrt{x_1^2 + x_2^2 + x_3^2}$ as above, which maps the coordinates of a vector to their Euclidean norm. The following computation provides the Hessian matrix at the point $(0, 0, 0)$.

```
> f = function(x){sqrt(sum(x^2))}
> 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 for f to have a minimum at $(0, 0, 0)$. The above information can be used to check whether f has a local minimum at $(0, 0, 0)$. In order to check this, 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 Canuto and Tabacco 2010 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 using the `grad()` function.

```
> f = function(x){sqrt(sum(x^2))}
> grad(f, x = c(0, 0, 0))               # gradient at the
[1] 0 0 0                               # optimum point
> hessm = hessian(func, x = c(0, 0, 0)) # Hessian matrix
> eigen(hessm)$values                   # eigenvalues
[1] 251364.0 251364.0  80531.3
```

This 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.

2.3.3 Automatic Differentiation

For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, automatic differentiation, which is also called algorithmic differentiation or computational differentiation, is a technique employed to evaluate derivatives based on the chain rule. As the derivatives of the elementary functions, such as exp, log, sin, cos, etc., are already known, the derivative of f can be an automatically assembled from these known elementary partial derivatives by employing the chain rule.

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

The main difference between automatic differentiation and symbolic differentiation is that the latter focuses on the symbolic expression of formulae and the former concentrates on evaluation. The disadvantages of symbolic differentiation is in both taking up too much memory for the computation, and generating unnecessary expressions associated with the computation. Consider, for example, the symbolic differentiation of

$$f(x) = \prod_{i=1}^{10} x_i = x_1 \cdot x_2 \cdot \cdots \cdot x_{10}.$$

The corresponding gradient in symbolic style is

$$\begin{aligned} \nabla f(x) &= \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_{10}} \right) \\ &= (x_2 \cdot x_3 \cdot \cdots \cdot x_{10}, x_1 \cdot x_3 \cdot \cdots \cdot x_{10}, \ldots, x_1 \cdot x_2 \cdot \cdots \cdot x_9). \end{aligned}$$

If the number of variables becomes large, then the expression will use a tremendous amount of memory and have a very tedious representation.

In automatic differentiation, all arguments of the function are redefined as dual numbers, $x_i + x_i'\varepsilon$, where ε has the property that $\varepsilon^2 \approx 0$. The change in x_i is $x_i'\varepsilon$, for all i. Therefore, automatic differentiation for this function looks like

$$\nabla f(x) = \prod_{i=1}^{10} x_i + \varepsilon \left(x_1' \prod_{i=2}^{10} x_i + \cdots + x_j' \prod_{i \neq j}^{10} x_i + \cdots + x_{10}' \prod_{i=1}^{9} x_i \right).$$

Automatic differentiation is more accurate than numerical differentiation. Numerical differentiation (or divided differences) uses

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

or

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

It is obvious that the accuracy of this type of differentiation is related to the choice of h. If h is small, then the method of divided differences has errors introduced by rounding off the floating point numbers. If h is large, then the formula disobeys the essence of this method, which assumes that h tends to zero. Also, the method of divided differences introduces truncation errors by neglecting the terms of order $\mathcal{O}(h^2)$, something which does not happen in automatic differentiation.

Automatic differentiation has two operation modes: forward and reverse. For forward mode, the algorithm starts by evaluating the derivatives of every elementary function, the function arguments itself, of f at the given points. In each intermediate step, the derivatives are combined to reproduce the derivatives of more complicated functions. The last step merely assembles the evaluations from the results of the computations already performed, employing the chain rule. For example, we use the forward mode to evaluate the derivative of $f(x) = (x + x^2)^3$: the pseudocode can be summarised as

```
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 f' represents the derivative, i.e. $\partial f/\partial x$. Therefore, let us evaluate the derivative of $f(x) = (x + x^2)^3$ at the point $x = 2$ with the forward mode.

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

For reverse mode, the programme performs the computation in the reverse direction. We need to set $\bar{v} = dy/dv$, then $\bar{y} = dy/dy = 1$. For the same example as before, where the derivative at $x = 2$ is evaluated, it looks as

$$\begin{aligned} \bar{s}_2 &= 3 \cdot s_2^2 = 3 \cdot 36 = 108, \\ \bar{s}_1 &= 3 \cdot s_2^2 = 3 \cdot 36 = 108, \\ \bar{x} &= \bar{s}_2 + 4\bar{s}_1 = 108 + 4 \cdot 108 = 540. \end{aligned}$$

Two examples are implemented in `R` using the package `radx` developed by Annamalai (2010). This package is not available on CRAN, therefore is installed via

function `install_github` (provided by package `devtools`) from GitHub from repository `radx` by `quantumelixir`.

Example 2.5 Evaluate the first-order derivative of

$$f(x) = \left(x + x^2\right)^3, \quad for\ x = 2. \tag{2.25}$$

```
> require(devtools)
> # install_github("radx","quantumelixir")  # installs from GitHub
> require(radx)                             # not provided by CRAN
> f = function(x) {(x^2 + x)^3}             # function
> radxeval(f,                               # automatic differ.
+   point = 2,                              # point at which to eval.
+   d     = 1)                              # order of differ.
     [,1]
[1,]  540
```

The upper computation illustrates that the value of the first derivative of the function (2.25) at $x = 2$ is equal to 540.

Example 2.6 Evaluate the first and second derivatives of the vector function

$$f_1(x, y) = 1 - 3y + \sin(3\pi y) - x,$$
$$f_2(x, y) = y - \sin(3\pi x)/2,$$

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

```
> f = function(x, y){                 # multidimensional function
+   c(1 - 3 * y + sin(3 * pi * y) - x,
+     y - sin(3 * pi * x) / 2)
+ }
> radxeval(f,
+   point = c(3, 5),                  # point at which to evaluate
+   d     = 1)                        # 1st order of differentiation
          [,1]     [,2]
[1,]  -1.00000 4.712389
[2,] -12.42478 1.000000
> radxeval(f,
+   point = c(3, 5),                  # point at which to evaluate
+   d     = 2)                        # 2nd order of differentiation
             [,1]         [,2]
[1,]  0.00000e+00 4.894984e-14
[2,]  0.00000e+00 0.000000e+00
[3,] -4.78741e-13 0.000000e+00
```

2.4 Root Finding

A root is the solution to a system of equations or an optimisation problem. In both cases, one tries to find values for the arguments of the function such that the value of the function is zero. In the case of an optimisation problem, this is done for the first derivative of the objective function.

2.4.1 Solving Systems of Linear Equations

Let K denote either the set of real numbers, or the set of complex numbers. Suppose $a_{ij}, b_i \in K$ with $i = 1, \ldots, n$ and $j = 1, \ldots, p$. Then the following system of equations is called a system of linear equations:

$$\begin{cases} a_{11}x_1 + \ldots + a_{1p}x_p = b_1 \\ \vdots \qquad\qquad\qquad \vdots \\ a_{n1}x_1 + \ldots + a_{np}x_p = b_n \end{cases}$$

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

$$\mathcal{A}x = b. \tag{2.26}$$

Let $\mathcal{A}^e_{n\times(p+1)}$ be the extended matrix, i.e. the matrix whose last column is the vector of constants b, and otherwise is the same as $\mathcal{A}$. Then (2.26) can be solved if and only if the rank of $\mathcal{A}$ is the same as the rank of $\mathcal{A}^e$. In this case b can be represented by a linear combination of the columns of $\mathcal{A}$. If (2.26) can be solved and the rank of $\mathcal{A}$ equals $n = p$, then there exists a unique solution. Otherwise (2.26) might have no solution or infinitely many solutions, see Greub (1975).

The Gaussian algorithm, which transforms the system of equations by elementary transformations to upper triangular form, is frequently applied. The solution can be computed by back-substitution. The Gaussian algorithm decomposes $\mathcal{A}$ into the matrices $\mathcal{L}$ and $\mathcal{U}$, the so-called $\mathcal{LU}$ decomposition (see Braun and Murdoch (2007) for further details). $\mathcal{L}$ 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 (2.26) can be rewritten as

$$\mathcal{A}x = \mathcal{L}\mathcal{U}x = b. \tag{2.27}$$

Now the system in (2.26) can be 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 the matrix `A` and the 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 into the product of an orthogonal matrix $\mathcal{Q}$ and an upper triangular matrix $\mathcal{R}$. 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 a square matrix. The next example shows how to solve a system of linear equations in `R` using `solve()`.

Example 2.7 Solve the following system of linear equations in `R` with the Gaussian algorithm and back-substitution,

$$\begin{aligned}
&\mathcal{A}x = b,\\
&\mathcal{A} = \begin{pmatrix} 2 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & 2 & -\frac{1}{2} \\ -\frac{1}{2} & 2 & 0 & -\frac{1}{2} \\ 0 & -\frac{1}{2} & -\frac{1}{2} & 2 \end{pmatrix},\\
&b = (0, 3, 3, 0)^\top,\\
&\mathcal{A}^e = \begin{pmatrix} 2 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ -\frac{1}{2} & 0 & 2 & -\frac{1}{2} & 3 \\ -\frac{1}{2} & 2 & 0 & -\frac{1}{2} & 3 \\ 0 & -\frac{1}{2} & -\frac{1}{2} & 2 & 0 \end{pmatrix}.
\end{aligned}$$

The upper system of linear equations is solved first by hand and then the example is computed in `R` for verification. This system of linear equations is not difficult to solve with the Gaussian algorithm. First, one finds the upper triangular matrix

$$\mathcal{U}^e = \begin{pmatrix} 2 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & \frac{15}{8} & -\frac{1}{8} & -\frac{1}{2} & 3 \\ 0 & 0 & \frac{28}{15} & -\frac{8}{15} & \frac{16}{5} \\ 0 & 0 & 0 & \frac{12}{7} & \frac{12}{7} \end{pmatrix}.$$

Second, one uses back-substitution to obtain the final result, that $(x_1, x_2, x_3, x_4)^\top = (1, 2, 2, 1)^\top$. Then the solution of this system of linear equations in `R` is presented. Two parameters are required: the coefficient matrix `A` and the vector of constraints `b`.

```
> A = matrix(                  # coefficient matrix
+   c(   2, -1/2, -1/2,    0,
+     -1/2,    0,    2, -1/2,
+     -1/2,    2,    0, -1/2,
```

```
+         0, -1/2, -1/2,    2),
+   ncol = 4, nrow = 4, byrow = TRUE)
> b = c(0, 3, 3, 0)            # vector of constants
> solve(A, b)
[1] 1 2 2 1                    # x1, x2, x3, x4
```

The manually found solution for the system coincides with the solution found in R.

2.4.2 Solving Systems of Nonlinear Equations

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

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

There are many different numerical methods for solving systems of nonlinear equations. In general, one distinguishes between gradient and non-gradient methods. In the following, the *Newton method*, or the *Newton–Raphson method*, is presented. To get a better illustration of the idea behind the Newton method, consider a continuous differentiable function $F : \mathbb{R} \to \mathbb{R}$, where one tries to find x^* with $F(x^*) = 0$ and $\frac{\partial F(x)}{\partial x}\Big|_{x=x^*} \neq 0$. Start by choosing a starting value $x_0 \in \mathbb{R}$ and define the tangent line

$$p(x) = F(x_0) + \frac{\partial F(x)}{\partial x}\Big|_{x=x_0} (x - x_0). \tag{2.28}$$

Then the tangent line $p(x)$ is a good approximation to F in a sufficiently small neighbourhood of x_0. If $\frac{\partial F(x)}{\partial x}\Big|_{x=x_0} \neq 0$, the root x_1 of p in (2.28) can be computed as follows:

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

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. Figure 2.2 demonstrates the Newton method for $f(x) = x^2 - 4$ with the starting value $x_0 = 6$.

The Fig. 2.2 was computed using the function `newton.method(f, init, ...)` from the package `animation`, where `f` is the function of interest and `init` is the starting value for the iteration process. The function provides an illustration of the iterations in Newton's method (see `help(newton.method)` for further details). The function `uniroot()` searches in an interval for a root of a function and returns

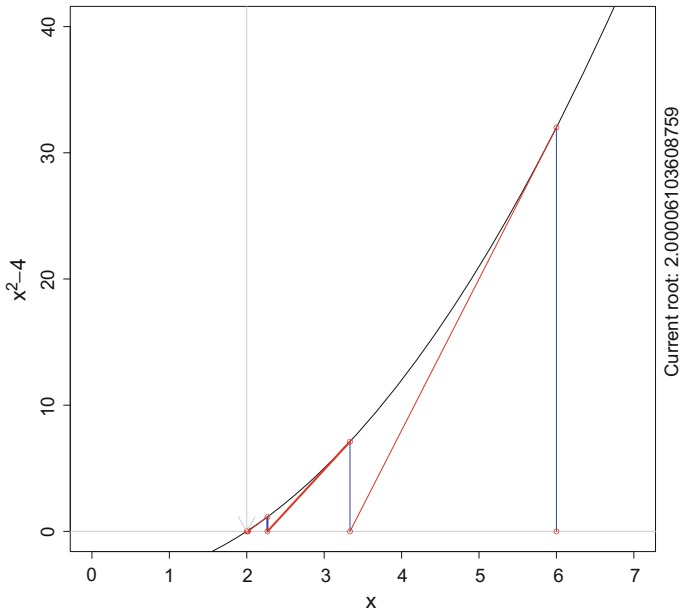

Fig. 2.2 Illustration of the iteration steps of Newton's method to find the root of $f(x) = x^2 - 4$ with $x_0 = 6$. BCS_Newton

only one root, even if several roots exist within the interval. At the boundaries of the interval, the sign of the value of the function must change.

```
> f = function(x){                  # objective function
+   -x^4 - cos(x) + 9 * x^2 - x - 5
+ }
> uniroot(f,
+   interval = c(0, 2))$root        # root in [0, 2]
[1] 0.8913574
> uniroot(f,
+   interval = c(-3, 2))$root       # root in [-3, 2]
[1] -2.980569
> uniroot(f,
+   interval = c(0, 3))$root        # f(0) and f(3) negative
Error in uniroot(f, c(0, 3)) :
  Values of f() at the boundaries have same 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 in the increasing order, computes a root. The algorithm does not guarantee it will find all the roots of the polynomial.

```
> z = c(0.2567, 0.1570, 0.0821, -0.3357, 1) # coefficients
> round(polyroot(z), digits = 2)            # complex roots
[1]  0.59+0.60i -0.42+0.44i -0.42-0.44i  0.59-0.60i
```

2.4.3 Maximisation and Minimisation of Functions

The maximisation and minimisation of functions, or optimisation problems, contain two components, such as an objective function $f(x)$ and constraints $g(x)$. Optimisation problems can be classified into two categories, according to the existence of constraints. If there are constraints affiliated with the objective function, then it is a constrained optimisation problem; otherwise, it is a unconstrained optimisation problem. This section introduces six different optimisation techniques. The first four are the golden ratio search method, the Nelder–Mead method, the BFGS method, and the conjugate gradient method for unconstrained optimisation. The two other optimisation techniques, linear programming (LP) and nonlinear programming (NLP), are used for constrained optimisation problems.

First, one needs to define the concepts of local and global extrema, which will be frequently used later on.

Definition 2.8 A real function f defined on a domain M has a global maximum at x_{opt} if $f(x_{opt}) \geq f(x)$ for all x in M. Then $f(x_{opt})$ is called the maximum value of the function. Analogously, the function has a global minimum at x_{opt} if $f(x_{opt}) \leq f(x)$ for all x in M. Then $f(x_{opt})$ is called the minimum value of the function.

Definition 2.9 If the domain M is a metric space, then f is said to have a local maximum at x_{opt} if there exists some $\epsilon > 0$ such that $f(x_{opt}) \geq f(x)$ for all x in M within a distance of ϵ from x_{opt}. Analogously, the function has a local minimum at x_{opt} if $f(x_{opt}) \leq f(x)$ for all x in M within ϵ of x_{opt}.

Maxima and minima are not always unique. Consider the function $\sin(x)$, which has global maxima $f(x_{max}) = 1$ and global minima $f(x_{min}) = -1$ for every $x_{max} = (0.5 + 2k)\pi$ and $x_{min} = (-0.5 + 2k)\pi$ for $k \in \mathbb{Z}$.

Example 2.8 The following function possesses several local maxima, local minima, global maxima and global minima.

$$\begin{aligned} f(x, y) = {} & 0.03 \sin(x) \sin(y) - 0.05 \sin(2x) \sin(y) \\ & + 0.01 \sin(x) \sin(2y) + 0.09 \sin(2x) \sin(2y). \end{aligned} \tag{2.29}$$

The function is plotted in Fig. 2.3 with highlighted extrema.

Golden ratio search method

The golden ratio section search method was proposed in Kiefer (1953). This method is frequently employed for solving optimisation problems with one-dimensional uni-modal objective functions, and belongs to the group of non-gradient methods. A very common algorithm for this method is the following one:

1. Define upper bound x_U, lower bound x_L, $x_U > x_L$ and ε.
2. Set $r = (\sqrt{5} + 1)/2$, $d = (x_U - x_L)r$.
3. Choose x_1 and $x_2 \in [x_L, x_U]$. Set $x_1 = x_L + d$ and $x_2 = x_U - d$.

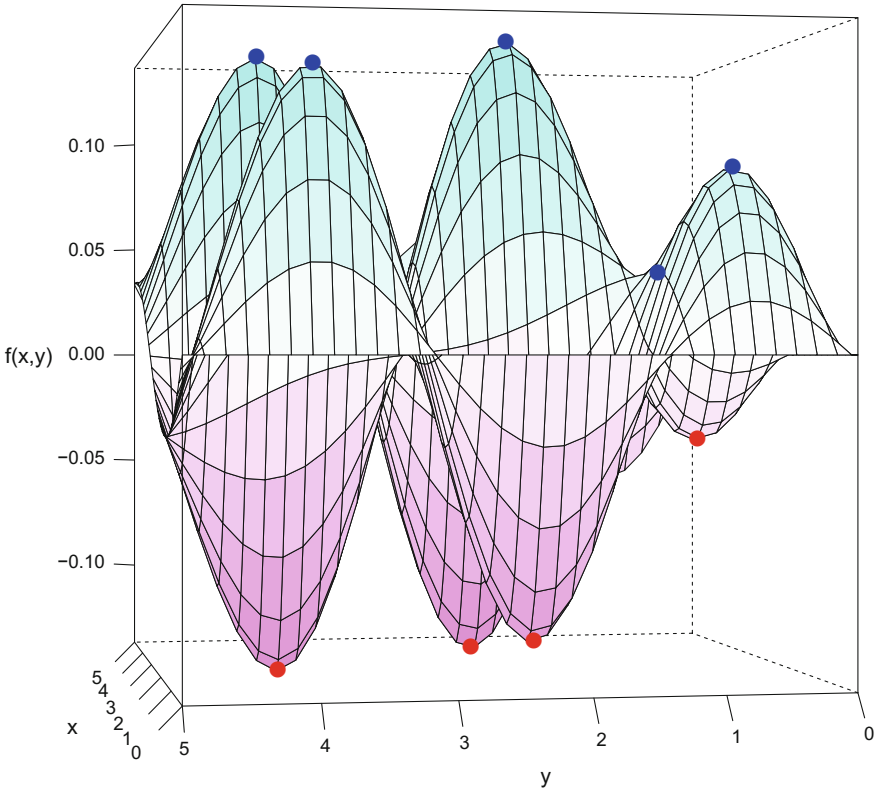

Fig. 2.3 3D plot of the function (2.29) with maxima and minima depicted by points. BCS_Multimodal

4. Stop if $|f(x_U) - f(x_L)| < \varepsilon$ or $|x_U - x_L| < \varepsilon$. If $f(x_1) < f(x_2) \Rightarrow x_{min} = x_1$, otherwise $x_{min} = x_2$.
5. Update x_U or x_L if $|f(x_U) - f(x_L)| \geq \varepsilon$ or $|x_U - x_L| \geq \varepsilon$. If $f(x_1) > f(x_2) \Rightarrow x_U = x_2$, otherwise $x_L = x_1$.
6. Return to step 2.

The algorithm first defines an initial search interval d. The length of the interval depends on the difference between the upper and lower bounds $x_U - x_L$ and the 'Golden Ratio' $r = \frac{1+\sqrt{5}}{2}$. The points x_1 and x_2 will decrease the length of the search interval. If the tolerance level ε for the search criteria ($|f(x_U) - f(x_L)|$ or $|x_U - x_L|$) is satisfied, the process stops. But if the search criteria is still greater than or equal to the tolerance level, the bounds are updated and the algorithm starts again.

Example 2.9 Apply the golden ratio search method to find the maximum of

$$f(x) = -(x-3)^2 + 10. \tag{2.30}$$

```
> require(stats)
> f = function(x){-(x - 3)^2 + 10}   # function
> optimize(f,                        # objective function
+   interval = c(-10, 10),           # interval
+   tol      = 0.0001,               # level of the tolerance
+   maximum  = TRUE)                 # to find maximum
$maximum
[1] 3

$objective
[1] 10
```

The argument `tol` defines the convergence criterion for the results. The function reaches its global maximum at $x_{opt} = 3$, which is easily derived by solving the first-order condition $-2x_{opt} + 6 = 0$ for x_{opt} and computing the value $f(x_{opt})$. For a maximum at x_{opt}, one should have $\frac{\partial^2 f(x)}{\partial x^2} < 0$ and $\frac{\partial^2 f(x)}{\partial x^2}\Big|_{x=x_{opt}} = -2$. Therefore $x_{opt} = 3$, which is verified in R with the code from above.

Nelder–Mead method

This method was proposed in Nelder and Mead (1965) and is applied frequently in multivariate unconstrained optimisation problems. It is a direct method, where the computation does not use gradients. The main idea of the Nelder–Mead method is briefly explained below and a graph for a two-dimensional input case is shown in Fig. 2.4.

1. Choose x_1, x_2, x_3 such that $f(x_1) < f(x_2) < f(x_3)$ and set ϵ_x and/or ϵ_f.
2. Stop if $\|x_i - x_j\| < \epsilon_x$ and/or $\|f(x_i) - f(x_j)\| < \epsilon_f$, $for\ i \neq j,\ i,\ j \in \{1, 2, 3\}$ and set $x_{min} = x_1$.
3. Else, compute $z = \frac{1}{2}(x_1 + x_2)$ and $d = 2z - x_3$.

   If $f(x_1) < f(d) < f(x_2) \Rightarrow x_3 = d$.
   If $f(d) \leq f(x_1)$, compute $k = 2d - z$.
   If $f(k) < f(x_1) \Rightarrow x_3 = k$.
   Else, $x_3 = d$.
   If $f(x_3) > f(d) \geq f(x_2) \Rightarrow x_3 = d$.
   Else, compute $t = [t|f(t) = \min\{f(t_1), f(t_2)\}]$, where $t_1 = \frac{1}{2}(x_3 + z)$ and $t_2 = \frac{1}{2}(d + z)$.
   If $f(t) < f(x_3) \Rightarrow x_3 = t$.
   Else, $x_3 = s = 1/2(x_1 + x_3)$ and $x_2 = z$.

4. Return to step 2.

In general, the Nelder–Mead algorithm works with more than three initial guesses. The starting values x_i are allowed to be vectors. In the iteration procedure one tries to improve the initial guesses step by step. The worst guess x_3 will be replaced by better values until the convergence criterion for the values ϵ_f of the function or the arguments ϵ_x of the function is met. Next we will give an example of how to use the Nelder–Mead method to find extrema of a function in R (Fig. 2.5).

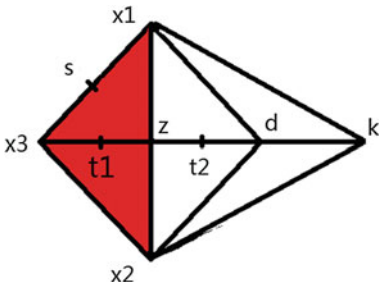

Fig. 2.4 Algorithm graph for the Nelder–Mead method. The variables x_1, x_2 and x_3 are the search region at the specific iteration step. All other variables, d, k, s, t_1 and t_2, are possible updates for one x_i

Example 2.10 The function to be minimized is the Rosenbrock function, which has an analytic solution with global minimum at $(1, 1)$ and a global minimum value $f(1, 1) = 0$.

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{2.31}$$

```
> require(neldermead)
> f = function(x){
+   100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2 # Rosenbrock function
+ }
> fNM = fminsearch(fun = f,
+   x0      = c(-1.2, 1),                  # starting point
+   verbose = FALSE)
> neldermead.get(fNM, key ="xopt")         # optimal x-values
          [,1]
[1,] 1.000022
[2,] 1.000042
> neldermead.get(fNM, key ="fopt")         # optimal function value
[1] 8.177661e-10
```

The upper computation illustrates that the numerical solution by the Nelder–Mead method is close to the analytical solution for the Rosenbrock function (2.31). The errors of the numerical solution are negligibly small.

BFGS method

This frequently used method for multivariate optimisation problems was proposed independently in Broyden (1970), Fletcher (1970), Goldfarb (1970) and Shanno (1970). BFGS stands for the first letters of each author, in alphabetical order. The main idea of this method originated from Newton's method, where the second-order Taylor expansion for a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ at $x = x_i \in \mathbb{R}^n$ is employed, such that

$$f(x) = f(x_i) + \nabla f^\top(x_i)q + \frac{1}{2}q^\top H(x_i)q,$$

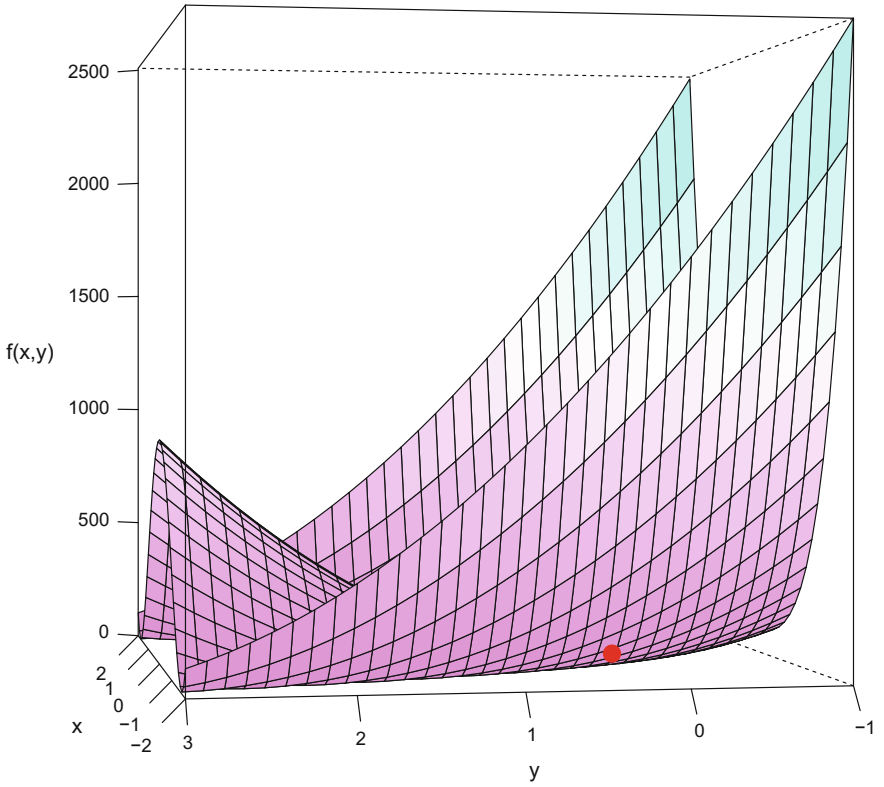

Fig. 2.5 Plot for the Rosenbrock function with its minimum depicted by a point. BCS_Rosenbrock

where $q = x - x_i$, and $\nabla f(x_i)$ is the value of the partial derivative of f at the point x_i, and $H(x_i)$ is the Hessian matrix. Employing the first-order condition, one obtains

$$\nabla f(x) = \nabla f(x_i) + H(x_i)q = 0,$$

hence, if $H(x_i)$ is invertible, then

$$\begin{aligned} q &= x - x_i = -H^{-1}(x_i)\nabla f(x_i), \\ x_{i+1} &= x_i - H^{-1}(x_i)\nabla f(x_i). \end{aligned}$$

The recursion will converge quadratically to the optimum. 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 the Hessian's inverse obtained from the following optimisation problem,

$$H(x_i) = \arg\min_{H} \|H^{-1} - H^{-1}(x_{i-1})\|_{\mathcal{W}}, \tag{2.32}$$
$$\text{subject to:} \quad \mathcal{B}^{-1} = (H^{-1})^{\top},$$
$$\mathcal{B}^{-1}(\nabla f_i - \nabla f_{i-1}) = x_i - x_{i-1}.$$

The weighted Frobenius norm, denoted by $\|\cdot\|_{\mathcal{W}}$, and the matrix $\mathcal{W}$ are, respectively,

$$\|H^{-1} - H^{-1}(x_{i-1})\|_{\mathcal{W}} = \|\mathcal{W}^{\frac{1}{2}}\{H^{-1} - H^{-1}(x_{i-1})\}\mathcal{W}^{\frac{1}{2}}\|,$$
$$\mathcal{W}(\nabla f_i - \nabla f_{i-1}) = x_i - x_{i-1}.$$

Equation (2.32) has a unique solution such that

$$H^{-1}(x_i) = \mathcal{M}_1 H(x_{i-1})\mathcal{M}_2 + (x_i - x_{i-1})\gamma_{i-1}(x_i - x_{i-1})^{\top},$$
$$\mathcal{M}_1 = \mathcal{I} - \gamma_{i-1}(x_i - x_{i-1})(\nabla f_i - \nabla f_{i-1})^{\top},$$
$$\mathcal{M}_2 = \mathcal{I} - \gamma_{i-1}(\nabla f_i - \nabla f_{i-1})(x_i - x_{i-1})^{\top},$$
$$\gamma_i = \{(\nabla f_i - \nabla f_{i-1})^{\top}(x_i - x_{i-1})\}^{-1}.$$

Example 2.11 Here, the BFGS method is used to minimise the Rosenbrock function (2.31) using `optimx` package (see Nash and Varadhan 2011).

```
> require(optimx)
> f     = function(x){100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2}
> fBFGS = optimx(fn = f,                # objective function
+   par    = c(-1.2, 1),                # starting point
+   method ="BFGS")                     # optimisation method
> print(data.frame(fBFGS$p1, fBFGS$p2, fBFGS$value))
  fBFGS.p1  fBFGS.p2  fBFGS.value
1 0.9998044 0.9996084 3.827383e-08     # minimum
```

The BFGS method computes the minimum value of the function (2.31) to be $3.83e-08$ at the minimum point $(0.99, 0.99)$. The outputs `fevals = 127, gevals = 38` show the calls of the objective function and the calls of the gradients, respectively. These outputs are close to the exact solution $x_{opt} = (1, 1)$ and $f(x_{opt}) = 0$.

Conjugate gradient method

The conjugate gradient method was proposed in Hestenes and Stiefel (1952) and is widely used for solving symmetric positive definite linear systems. A multivariate unconstrained optimisation problem, like

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

can be solved with the Conjugate Gradient Method. The main idea behind this method is to use iterations to approach the optimum of the linear system.

1. Set x_0 and ε, then compute $p_0 = r_0 = b - \mathcal{A}x_0$.
2. Stop if $r_i < \varepsilon$ and set $x_{opt} = x_{i+1}$.
3. Else, compute:

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

4. Update x_i, r_i and p_i. Increment i.
5. Return to step 2.

At first, the initial guess x_0 determines the residual r_0 and the initially used basis vector p_0. The algorithm tries to reduce the residual r_i at each step to get to the optimal solution. At the optimum, $0 = b - Ax_i$. The tolerance level t and the final residual should be close to zero. The parameters α_i and β_i are improvement parameters for the next iteration. The parameter α_i directly determines the size of the improvement for the residual and indirectly influences the conjugate vector p_{i+1} used in the next iteration. For β_i, the opposite is true. The final result depends on both parameters.

Example 2.12 To illustrate the Conjugate Gradient method, let us again consider the Rosenbrock function (2.31).

```
> require(optimx)
> f   = function(x){100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2}
> fCG = optimx(fn = f,                          # objective function
+   par     = c(1.2, 1),                        # initial guess (x_0)
+   control = list(reltol = 10^-7),             # relative tolerance
+   method  ="CG")                              # method of optimisation
>
> print(data.frame(fCG$p1, fCG$p2, fCG$value))  # minimum
   fCG.p1    fCG.p2    fCG.value
1  1.030077  1.061209 0.0009036108
```

For the Rosenbrock function, the Conjugate Gradient method delivers the biggest errors, compared to the Nelder–Mead and BFGS methods. All numerical methods which are applied to optimize a function will only approximately find the true solution. The examples above show how the choice of method might influence the accuracy of the result. Worth mentioning is, that in the latter case we changed the initial guess, as the function failed with the same starting value as we took for BFGS method.

Constrained optimisation

Constrained optimisation problems can be categorised into two classes in terms of to the linearity of the objective function and the constraints. A linear programming

(LP) problem has a linear objective function and linear constraints, otherwise it is a nonlinear programming problem (NLP).

LP is a method to find the solution to an optimisation problem with a linear objective function, under constraints in the form of linear equalities and linear inequalities. It has a feasible region defined by a convex polyhedron, which is a set made by the intersection of finitely many half-spaces. These represent linear inequalities. The objective of linear programming is to find a point in the polyhedron where the objective function reaches a minimum or maximum value. A representative LP can be expressed as follows:

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

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 the coefficients in the constraints. The expression $a^\top x$ 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.

NLP has an analogous definition as that of the LP problem. The differences between NLP and LP are that the objective function or the constraints in an NLP can be nonlinear functions. The following example is an LP problem (Fig. 2.6).

Example 2.13 Solve the following linear programming optimisation problem with R.

$$\begin{aligned} &\arg\max_{x_1,\, x_2} 2x_1 + 4x_2, \\ \text{subject to:} \quad &3x_1 + 4x_2 \leq 60, \\ &x_1 \geq 0, \\ &x_2 \geq 0. \end{aligned} \tag{2.33}$$

For the example in (2.33), the function from the package `Rglpk` (see Theussl 2013) is used to compute the solution in `R`.

```
> require(Rglpk)
> Rglpk_solve_LP(obj = c(2, 4),       # objective function
+   mat = matrix(c(3, 4), nrow = 1), # constrains coefficients
+   dir ="<=",                        # type of constrains
+   rhs = 60,                         # constrains vector
+   max = TRUE)                       # to maximise
$optimum                             # maximum
[1] 60

$solution                            # point of maximum
[1]  0 15

$status                              # no errors
[1] 0
```

The maximum value of the function (2.33) is 60 and occurs at the point (0, 15).

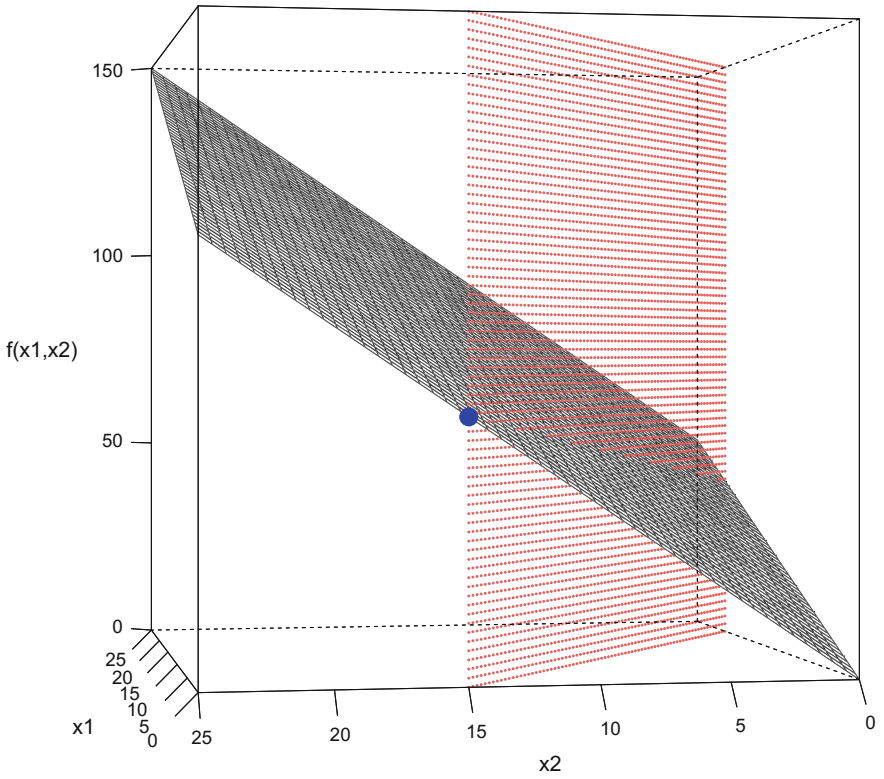

Fig. 2.6 Plot for the linear programming problem with the constraint hyperplane depicted by the grid and the optimum by a point. BCS_LP

Example 2.14 Next, consider a constrained nonlinear optimisation problem, which can be solved in R using `constrOptim()`. Solve the following nonlinear optimisation problem with R (Fig. 2.7).

$$\begin{aligned} \arg\max_{x_1,\, x_2} \sqrt{5x_1} + \sqrt{3x_2}, \qquad (2.34) \\ \text{subject to: } 3x_1 + 5x_2 \leq 10, \\ x_1 \geq 0, \\ x_2 \geq 0. \end{aligned}$$

```
> require(stats)
> f      = function(x){
+   sqrt(5 * x[1]) + sqrt(3 * x[2])   # objective function
+ }
> A      = matrix(c(-3, -5), nrow = 1,
+   ncol = 2, byrow = TRUE)           # coefficients matrix
> b      = c(-10)                     # vector of constraints
```

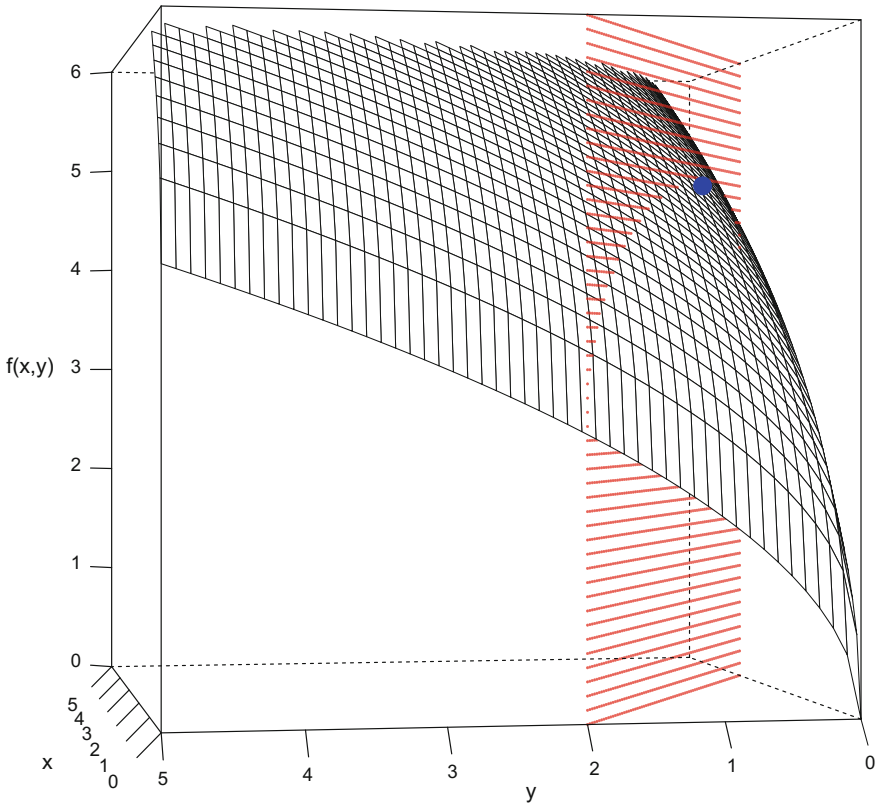

Fig. 2.7 Plot for the objective function with its constraint from (2.34) and the optimum depicted by the point. BCS_NLP

```
> answer = constrOptim(f = f,          # objective function
+    theta   = c(1, 1),                # initial guess
+    grad    = NULL,                   # no gradient provided
+    ui      = A,
+    ci      = b,                      # vector of constrains
+    control = list(fnscale = -1))     # to maximise
> c(answer$par, answer$value)          # optimum
[1] 2.4510595 0.5293643 4.7609523
```

The upper computation illustrates that the maximum value of the function (2.34) is 4.7610, and occurs at the point (2.4511, 0.5294). `answer$function` equal to 170 means that the objective function has been called 170 times.

http://www.springer.com/978-3-319-55335-1

Basic Elements of Computational Statistics

Härdle, W.; Okhrin, O.; Okhrin, Y.

2017, XXI, 305 p. 97 illus., 66 illus. in color., Hardcover
ISBN: 978-3-319-55335-1