Guido Buzzi-Ferraris and Flavio Manenti

Differential and Differential-Algebraic Systems for the Chemical Engineer

Solving Numerical Problems
Guido Buzzi-Ferraris
Flavio Manenti

Differential and
Differential-Algebraic
Systems for the
Chemical Engineer
Related Titles

Buzzi-Ferraris, G., Manenti, F.

Fundamentals and Linear Algebra for the Chemical Engineer
Solving Numerical Problems
2010
Print ISBN: 978-3-527-32552-8

Buzzi-Ferraris, G., Manenti, F.

Interpolation and Regression Models for the Chemical Engineer
Solving Numerical Problems
2010
Print ISBN: 978-3-527-32652-5

Buzzi-Ferraris, Guido/Manenti, Flavio

Nonlinear Systems and Optimization for the Chemical Engineer
Solving Numerical Problems
2013
Print ISBN: 978-3-527-33274-8; also available in digital formats

Velten, K.

Mathematical Modeling and Simulation
Introduction for Scientists and Engineers
2009
Print ISBN: 978-3-527-40758-3, also available in digital formats
Contents

Preface IX

1 Definite Integrals 1
1.1 Introduction 1
1.2 Calculation of Weights 2
1.3 Accuracy of Numerical Methods 3
1.4 Modification of the Integration Interval 4
1.5 Main Integration Methods 5
1.5.1 Newton–Cotes Formulae 5
1.5.2 Gauss Formulae 6
1.6 Algorithms Derived from the Trapezoid Method 9
1.6.1 Extended Newton–Cotes Formulae 10
1.6.2 Error in the Extended Formulae 11
1.6.3 Extrapolation of the Extended Formulae 12
1.7 Error Control 15
1.8 Improper Integrals 16
1.9 Gauss–Kronrod Algorithms 17
1.10 Adaptive Methods 19
1.10.1 Method Derived from the Gauss–Kronrod Algorithm 20
1.10.2 Method Derived from the Extended Trapezoid Algorithm 21
1.10.3 Method Derived from the Gauss–Lobatto Algorithm 22
1.11 Parallel Computations 23
1.12 Classes for Definite Integrals 23
1.13 Case Study: Optimal Adiabatic Bed Reactors for Sulfur Dioxide with Cold Shot Cooling 26

2 Ordinary Differential Equations Systems 31
2.1 Introduction 31
2.2 Algorithm Accuracy 35
2.3 Equation and System Conditioning 36
2.4 Algorithm Stability 40
2.5 Stiff Systems 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Multistep and Multivalue Algorithms for Stiff Systems</td>
</tr>
<tr>
<td>2.7</td>
<td>Control of the Integration Step</td>
</tr>
<tr>
<td>2.8</td>
<td>Runge–Kutta Methods</td>
</tr>
<tr>
<td>2.9</td>
<td>Explicit Runge–Kutta Methods</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Strategy to Automatically Control the Integration Step</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Estimation of the Local Error</td>
</tr>
<tr>
<td>2.9.2.1</td>
<td>Runge–Kutta–Merson Algorithm</td>
</tr>
<tr>
<td>2.9.2.2</td>
<td>Richardson Extrapolation</td>
</tr>
<tr>
<td>2.9.2.3</td>
<td>Embedded Algorithms</td>
</tr>
<tr>
<td>2.10</td>
<td>Classes Based on Runge–Kutta Algorithms in the BzzMath Library</td>
</tr>
<tr>
<td>2.11</td>
<td>Semi-Implicit Runge–Kutta Methods</td>
</tr>
<tr>
<td>2.12</td>
<td>Implicit and Diagonally Implicit Runge–Kutta Methods</td>
</tr>
<tr>
<td>2.13</td>
<td>Multistep Algorithms</td>
</tr>
<tr>
<td>2.13.1</td>
<td>Adams–Bashforth Algorithms</td>
</tr>
<tr>
<td>2.13.2</td>
<td>Adams–Moulton Algorithms</td>
</tr>
<tr>
<td>2.14</td>
<td>Multivalue Algorithms</td>
</tr>
<tr>
<td>2.14.1</td>
<td>Control of the Local Error</td>
</tr>
<tr>
<td>2.14.2</td>
<td>Change the Integration Step</td>
</tr>
<tr>
<td>2.14.3</td>
<td>Changing the Method Order</td>
</tr>
<tr>
<td>2.14.4</td>
<td>Strategy for Step and Order Selection</td>
</tr>
<tr>
<td>2.14.5</td>
<td>Initializing a Multivalue Method</td>
</tr>
<tr>
<td>2.14.6</td>
<td>Selecting the First Integration Step</td>
</tr>
<tr>
<td>2.14.7</td>
<td>Selecting the Multivalue Algorithms</td>
</tr>
<tr>
<td>2.14.7.1</td>
<td>Adams–Moulton Algorithms</td>
</tr>
<tr>
<td>2.14.7.2</td>
<td>Gear Algorithms</td>
</tr>
<tr>
<td>2.14.8</td>
<td>Nonlinear System Solution</td>
</tr>
<tr>
<td>2.15</td>
<td>Multivalue Algorithms for Nonstiff Problems</td>
</tr>
<tr>
<td>2.16</td>
<td>Multivalue Algorithms for Stiff Problems</td>
</tr>
<tr>
<td>2.16.1</td>
<td>Robustness in Stiff Problems</td>
</tr>
<tr>
<td>2.16.1.1</td>
<td>Eigenvalues with a Very Large Imaginary Part</td>
</tr>
<tr>
<td>2.16.1.2</td>
<td>Problems with Hard Discontinuities</td>
</tr>
<tr>
<td>2.16.1.3</td>
<td>Variable Constraints</td>
</tr>
<tr>
<td>2.16.2</td>
<td>Efficiency in Stiff Problems</td>
</tr>
<tr>
<td>2.16.2.1</td>
<td>When to Factorize the Matrix G</td>
</tr>
<tr>
<td>2.16.2.2</td>
<td>How to Factorize the Matrix G</td>
</tr>
<tr>
<td>2.16.2.3</td>
<td>When to Update the Jacobian J</td>
</tr>
<tr>
<td>2.16.2.4</td>
<td>How to Update the Jacobian J</td>
</tr>
<tr>
<td>2.17</td>
<td>Multivalue Classes in BzzMath Library</td>
</tr>
<tr>
<td>2.18</td>
<td>Extrapolation Methods</td>
</tr>
<tr>
<td>2.19</td>
<td>Some Caveats</td>
</tr>
<tr>
<td>3</td>
<td>ODE: Case Studies</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Nonstiff Problems</td>
</tr>
</tbody>
</table>
3.3 Volterra System 116
3.4 Simulation of Catalytic Effects 117
3.5 Ozone Decomposition 119
3.6 Robertson’s Kinetic 120
3.7 Belousov’s Reaction 121
3.8 Fluidized Bed 122
3.9 Problem with Discontinuities 123
3.10 Constrained Problem 124
3.11 Hires Problem 126
3.12 Van der Pol Oscillator 128
3.13 Regression Problems with an ODE Model 129
3.14 Zero-Crossing Problem 139
3.15 Optimization-Crossing Problem 142
3.15.1 Optimization of a Batch Reactor 142
3.15.2 Maximum Level in a Gravity-Flow Tank in Transient Conditions 145
3.15.3 Optimization of a Batch Reactor 148
3.16 Sparse Systems 150
3.17 Use of ODE Systems to Find Steady-State Conditions of Chemical Processes 155
3.18 Industrial Case: Spectrokinetic Modeling 157
3.18.1 CATalytic-Post-Processor 159
3.18.2 Nonreactive CFD Modeling 159
3.18.3 User-Defined Function 160
3.18.4 Reactor Modeling 160
3.18.5 Numerical Methods 162
3.18.6 Dynamic Simulation of an Operando FTIR Cell Used to Study NOx Storage on a LNT Catalyst 163
3.18.7 CAT-PP Simulation Results 166
3.18.8 Nomenclature 169

4 Differential and Algebraic Equation Systems 171
4.1 Introduction 171
4.2 Multivalue Method 174
4.3 DAE Classes in the BzzMath Library 175

5 DAE: Case Studies 187
5.1 Introduction 187
5.2 Van der Pol Oscillator 187
5.3 Regression Problems with the DAE Model 189
5.4 Sparse Structured Matrices 193
5.5 Industrial Case: Distillation Unit 199
5.5.1 Management of System Sparsity and Unstructured Elements 200
5.5.2 DAE Solver for Partially Structured Systems 201
5.5.3 Case-Study for Solver Validation: Nonequilibrium Distillation Column Model 202
5.5.4 Numerical Results 205
Notations for Table 5.1 208
Subscripts 208
Symbols 208

6 Boundary Value Problems 209
6.1 Introduction 209
6.1.1 Integral Relationships 211
6.1.2 Continuation Methods 212
6.1.3 Problems with an Unknown Constant Parameter 214
6.1.4 Problem with Unknown Boundary 214
6.2 Shooting Methods 215
6.3 Special Boundary Value Problems 217
6.3.1 Runge–Kutta Implicit Methods 218
6.4 More General BVP Methods 221
6.4.1 Collocation Method 222
6.4.2 Galerkin Method 222
6.4.3 Momentum Method 223
6.4.4 Least-Squares Method 223
6.5 Selection of the Approximating Function 224
6.6 Which and How Many Support Points Have to Be Considered? 225
6.7 Which Variables Should Be Selected as Adaptive Parameters? 231
6.8 The BVP Solution Classes in the BzzMath Library 237
6.9 Adaptive Mesh Selection 251
6.10 Case studies 253

Reference 265

Appendix A: Linking the BzzMath Library to Matlab 269
A.1 Introduction 269
A.2 BzzSum Function 269
A.2.1 Header File 270
A.2.2 MEX Function 270
A.2.3 C++ Part 271
A.2.4 Compiling 272
A.3 Chemical Engineering Example 272
A.3.1 Definition of a New Class 274
A.3.2 Main Program in C++ 275
A.3.3 Main Program in Matlab 277

Appendix B: Copyrights 279

Index 281
Preface

This book is aimed at students and professionals needing to numerically solve scientific problems involving differential and algebraic–differential systems.

We assume our readers have the basic familiarity with numerical methods that any undergraduate student in scientific or engineering disciplines should have. We also recommend at least a basic knowledge of C++ programming.

Readers who do not have any of the above should first refer to the companion books in this series:


These books explain and apply the fundamentals of numerical methods in C++.

Although many books on differential and algebraic–differential systems approach these topics from a theoretical viewpoint only, we wanted to explain the theoretical aspects in an informal way, by offering an applied approach to this scientific discipline. In fact, this volume focuses on the solution of concrete problems and includes many examples, applications, code samples, programming, and overall programs, to give readers not only the methodology to tackle their specific problems but also the structure to implement an appropriate program and ad hoc algorithms to solve it.

The book describes numerical methods, high-performance algorithms, specific devices, and innovative techniques and strategies, all of which are implemented in a well-established numerical library: the BzzMath library, developed by Prof. Guido Buzzi-Ferraris at the Politecnico di Milano and downloadable from http://www.chem.polimi.it/homes/gbuzzi.

This gives readers the invaluable opportunity to use and implement their code in a numerical library that involves some of the most appealing algorithms in the solution of differential equations, algebraic systems, optimal problems, data
regressions for linear and nonlinear cases, boundary value problems, linear pro-
gramming, and so on.

Unfortunately, unlike many other books that cover only theory, all these
numerical contents cannot be explained in a single volume because of their
application to real problems and the need for specific code examples. We there-
fore decided to split the numerical analysis topics into several distinct areas, each
one covered by an *ad hoc* book by the same authors and adopting the same
philosophy:

- Vol. I: Buzzi-Ferraris and Manenti (2010), *Fundamentals and Linear Alge-
  bra for the Chemical Engineer: Solving Numerical Problems*, Wiley-VCH
  Verlag GmbH, Weinheim, Germany.
- Vol. II: Buzzi-Ferraris and Manenti (2010), *Interpolation and Regression
  Models for the Chemical Engineer: Solving Numerical Problems*, Wiley-VCH
  Verlag GmbH, Weinheim, Germany.
- Vol. III: Buzzi-Ferraris and Manenti (2014) *Nonlinear Systems and Optimi-
  zation for the Chemical Engineer: Solving Numerical Problems*, Wiley-VCH
  Verlag GmbH, Weinheim, Germany.
  Algebraic Systems for the Chemical Engineer: Solving Numerical Problems*,
  Wiley-VCH Verlag GmbH, Weinheim, Germany.
  Engineer: Solving Numerical Problems*, Wiley-VCH Verlag GmbH, Wein-
  heim, Germany, in progress.

This book proposes algorithms and methods to solve differential and
differential–algebraic systems, whereas the companion books cover linear alge-
bra and linear systems, data analysis and regressions, and nonlinear systems and
optimization, respectively. After having introduced the theoretical content, all
explain their application in detail and provide optimized C++ code samples to
solve general problems. This allows readers to use the proposed programs to
tackle their specific numerical issues more easily by using the *BzzMath* library.

The *BzzMath* library can be used in any scientific field in which there is a need
to solve numerical problems. Its primary use is in engineering, but it can also be
used in statistics, medicine, economics, physics, management, environmental sci-
ences, biosciences, and so on.

**Outline of This Book**

This book deals with the solution of differential and differential–algebraic sys-
tems. Analogously to the aforementioned companion books, it proposes a series
of robust and high-performance algorithms implemented in the *BzzMath* library
to tackle these multifaceted and notoriously difficult issues.
Definite integrals are solved in Chapter 1. Existing methods and novel alternatives are proposed, implemented in the BzzMath library, and adopted to solve some well-established literature-based tests. Parallel computations are also introduced.

Ordinary differential equation systems are broached in Chapter 2. Conditioning, stability, and stiffness are described in detail by giving specific information on how to handle them whenever they arise. The BzzMath library also implements a wide set of algorithms to solve classical problems and chemical/process engineering problems.

Chapter 3 reports a collection of literature and industrial problems based on ordinary differential equation systems. The basics of the physical problem are described and the model behind it is given together as the initial conditions. Implementation tricks, special functions of the classes, and suggestions to improve the solution’s accuracy and efficiency are provided through various examples.

Differential–algebraic systems are explored in greater depth in Chapter 4. Special algorithms to handle this family of problems are described and implemented in the BzzMath library. Classes to handle the sparsity and structure of such systems typical of chemical engineering are also described.

Literature-based examples and industrial case studies are collected in Chapter 5. Implementation tricks and useful functions to handle very large and sparse systems with/without parallel computing are introduced.

Chapter 6 also introduces a novel general class to solve boundary value problems. Very stiff problems, such as shock waves and peaks, are automatically identified and the solution strategy self-adapt to such a situation.

**Notation**

These books contain icons not only to highlight some important features and concepts but also to underscore that there is potential for serious errors in programming or in selecting the appropriate numerical methods.

New concepts or new ideas. As they may be difficult to understand, it is necessary to change the point of view.

Description and remarks on important concepts and smart and interesting ideas.

Positive aspects, benefits, and advantages of algorithms, methods, and techniques in solving a specific problem.

Negative aspects and disadvantages of algorithms, methods, and techniques in solving a specific problem.

Some aspects are intentionally neglected.
Caveat, risk of making sneaky mistakes, and spread errors.

Description of some BzzMath library classes or functions.

Definitions and properties.

Conditioning status of the mathematical formulation.

Algorithm stability.

The algorithm efficiency assessment.

The problem, method, . . . is obsolete.


BzzMath Library Style

In order to facilitate both implementation and program reading, it was necessary to diversify the style of the identifiers.

C++ is a case-sensitive language and thus distinguishes between capital letters and small ones. Moreover, C++ identifiers are unlimited in the number of chars for their name, unlike FORTRAN77 identifiers. It is thus possible, and we feel indispensable, to use these prerogatives by giving every variable, object, constant, function, and so on, an identifier that allows us to immediately recognize what we are looking at.

Programmers typically use two different styles to characterize an identifier that consists of two words. One possibility is to separate the word by means of an underscore, that is, dynamic_viscosity. The other possibility is to begin the second word with a capital letter, that is, dynamicViscosity.

The style adopted in the BzzMath library is described hereinafter:

• **Constants**: The identifier should have more than two capital letters. If several words are to be used, they must be separated by an underscore.

  Some good examples are MACH_EPS, PI, BZZ_BIG_FLOAT, and TOLERANCE.

  Bad examples are A, Tolerance, tolerance, tol, and MachEps.

• **Variables** (standard type, derived type, class object): When the identifier consists of a single word, it may consist either of different chars starting with a small letter or of a single char either capitalized or small. On the
other hand, when the identifier consists of more than a single word, each word should start with a capital letter except for the first one, whereas all the remaining letters have to be small.

Some good examples are `machEpsilon`, `tol`, `x`, `A`, `G`, `dynamicViscosity`, and `yDoubleValue`.

Bad examples are `Aa`, `AA`, `A_A`, `Tolerance`, `tOLerance`, `MachEps`, and `mach_ellipse`.

• **Functions**: The identifier should have at least two chars: the first is capital, whereas the others are not. When the identifier consists of more words, each of them has to start with a capital letter.

Some good examples are `MachEpsilon`, `Tolerance`, `Aa`, `Abcde`, `DynamicViscosity`, and `MyBestFunction`.

Bad examples are `A`, `F`, `AA`, `A_A`, `tolerance`, `TOL`, and `machEps`.

• **New Types of Object**: This is similar to the function identifier, but in order to distinguish it from functions, it is useful to add a prefix. All the classes belonging to the `BzzMath` library are characterized by the prefix `Bzz`.

Some good examples are `BzzMatrix`, `BzzVector`, `BzzMinimum`, and `BzzOdeStiff`.

Bad examples are `A`, `matrix`, and `Matrix`.

Another style-based decision was to standardize the bracket positions at the beginning and at the end of a block to make C++ programs easier to read.

In this case also, programmers adopt two alternatives: some put the first bracket on the same row where the block starts, while some others put it on the following line with the same indenting of the bracket that closes the block.

The former case takes to the following style:

```cpp
for(i =1; i <=n; i ++){
    ... } if(x>1.){
        ... }
```

whereas the latter case takes to the following style:

```cpp
for(i =1; i <=n; i ++){
    ... }
} if(x>1.){
    ... }
```

This latter alternative is adopted in the `BzzMath` library.
A third important style-based decision concerned the criterion to pass variables of a function either by value or by reference. In the BzzMath library, we adopt the following criteria:

- If the variable is standard and the function keeps it unchanged, it is passed by value.
- If the variable is an object and the function keeps it unchanged, it is passed by reference and, if possible, as const type.
- If the variable (either standard or object) is to be modified by the function, its pointer must be provided.

The object C only is modified in the following statements:

\[
\text{Product}(3., A, &C);
\text{Product}(A, B, &C);
\]

Basic Requirements for Using BzzMath Library

BzzMath library, release 7.0, was designed for a Microsoft Windows environment.

Thanks to the synergistic collaboration with Professor Wozny’s research group at Technische Universitat Berlin, it will also be available in the Linux environment. The library is released for the following compilers:

- Visual C++ 2010 and Visual C++ 2012 for Windows environment and Linux gcc for Linux environments, the library is available for 64-bit machines.

openMP directives for parallel computing are available for all the above compilers except for Visual C++ 6.

Moreover, FORTRAN users can either adopt all the classes belonging to the BzzMath library using opportune interfaces or directly use pieces of C++ codes in FORTRAN, by means of the so-called mixed language (see Appendix A of Vol. 2, Buzzi-Ferraris and Manenti, 2010b).

The previous version of the BzzMath library (release 6.0) is updated until May 20, 2011 and will not undergo any further development. Moreover, the new release 7.0 has been extended quite significantly, particularly for classes dedicated to optimization and exploitation of openMP directives and to differential, differential–algebraic, and boundary value problems.

Also, MATLAB users can either adopt all the classes belonging to BzzMath library through opportune interfaces or directly use pieces of C++ codes in MATLAB by means of the so-called mixed language (see Appendix A of the present volume).
How to Install Examples Collected in This Book

Download and unzip WileyVol4.zip from Buzzi-Ferraris’s homepage (http://www.chem.polimi.it/homes/gbuzzi). Login is required, but download is free for non-profit uses.

A Few Steps to Install BzzMath Library

Windows users must follow these general tasks to use the BzzMath library on a computer:

- Unzip the file BzzMath7.zip in a convenient directory (for example in C:\NumericalLibraries\). This directory will be called DIRECTORY in the following. This unzip creates the subdirectory BzzMath, including other five subdirectories:
  - Lib, hpp, exe, Examples, and BzzMathTutorial are created into DIRECTORY\BzzMath.
  - The BzzMath.lib library is copied into DIRECTORY\BzzMath\Lib subdirectories, according to the compiler one would use (VCPP6, VCPP9, VCPP10, VCPP12, and INTEL11);
  - hpp files are copied into directory DIRECTORY\BzzMath\hpp.
  - exe files are copied into the directory DIRECTORY\BzzMath\exe.
  - The overall tutorial, .ppt files, is copied into the directory DIRECTORY\BzzMath\BzzMathTutorial.
  - Example files are copied into the directory DIRECTORY\BzzMath\Examples.
- In Microsoft Developer Studio 6 or later, open Options in the Tools menu option, then choose the tab Directories, and add the directory specification DIRECTORY\BzzMath\hpp to include files.
- Add DIRECTORY\BzzMath\exe and DIRECTORY\BzzMath\BzzMathTutorial in the PATH option of your operating system (Windows): Click with the right mouse button on System Resources. Choose the option Properties. Choose the option Advanced. Choose Ambient Variables. Choose the option PATH. Add the voice: DIRECTORY\BzzMath\exe; DIRECTORY\BzzMath\BzzMathTutorial;.

Please note that when a new directory is added to the PATH environment variable, the semicolon must be included before specifying the new directory.

After having changed the PATH environment variable, you must restart the computer. At the next machine start, you can use BzzMath.exe programs and/or the BzzMathTutorial.pps file, placed into the directory DIRECTORY\BzzMath\BzzMathTutorial.
Linux users will find gcc library file into DIRECTORY\BzzMath\Lib\Linux subdirectory.

Include the BzzMath Library in a Calculation Program

Whereas the previous paragraph describes an operation that should be performed only once, the following operations are needed whenever a new project is open:

1) BzzMath.lib must be added to the project (see also the following paragraph).
2) When at least an object of BzzMath library is used, it is necessary to select the appropriate compiler by choosing one of the following alternatives:
   o //default: Visual C++ 6.0 Windows without openMP
     #define BZZ_COMPILER 0//32 bit
   o //Visual C++ 9.0 (Visual 2008) Windows with openMP
     #define BZZ_COMPILER 1//32 bit
   o //Visual C++ 2010 Windows with openMP
     #define BZZ_COMPILER 2//64 bit
   o //Visual C++ 2012 Windows with openMP
     #define BZZ_COMPILER 3//64 bit
   o //INTEL 2013 with openMP
     #define BZZ_COMPILER 11//32 bit
   o //LINUX GCC with openMP
     #define BZZ_COMPILER 101//64 bit

   • Moreover, whenever even one BzzMath library object is used, it is always necessary to introduce the statement

     #include "BzzMath.hpp"

   at the beginning of the program, just below the BZZ_COMPILER selection. For example, using the INTEL 2013 with openMP in the Windows environment, you must enter the following statements:

     #define BZZ_COMPILER 11
     #include "BzzMath.hpp"
1 Definite Integrals

Examples from this chapter can be found in the directory Vol4_Chapter1 in the WileyVol4.zip file available at the following web site: http://www.chem.polimi.it/homes/gbuzzi.

1.1 Introduction

This chapter deals with the numerical integration of a function:

\[ I = \int_{a}^{b} f(x) \, dx \]  \hfill (1.1)

In the first part of the chapter, we suppose that the function \( f(x) \) leads to no numerical issues within the selected interval \([a; b]\) and that \( a \) and \( b \) can be represented as floating points without any overflow and underflow problems.

We consider the algorithms that approximate the integral \( I \) as follows:

\[ I = \sum_{i=1}^{n} w_i f(x_i), \quad n \geq 1 \]  \hfill (1.2)

These algorithms are different for the position \( x_i \), where the function is to be evaluated, as well as for the weights \( w_i \). In the following, we will assume we have all the points distinctly and sequentially placed:

\[ x_1 < x_2 < \cdots < x_n \]  \hfill (1.3)

The values of the function \( f(x_i) \) evaluated at the points \( x_i \) shall be denoted as \( f_i \) and the distance between \( x_i \) and \( x_{i+1} \) as \( h_i \). Moreover, if the points are evenly spaced, their distance is denoted by the generic \( h \).

If \( x_1 = a \) and \( x_n = b \), the rule is close; if only an external point corresponds to an extreme of the integration interval, the rule is semiopen; if neither of the external points coincide with the integration interval extremes, the rule is open.
For example, the trapezoid rule (also known as the trapezoidal rule or trapezium rule) is close:

\[ I = \frac{b-a}{2} \left[ f(a) + f(b) \right] = \frac{h}{2} \left[ f_1 + f_2 \right] \]  

whereas the midpoint rule is open:

\[ I = (b-a)f \left( \frac{a+b}{2} \right) \]  

### 1.2 Calculation of Weights

The numerical integration formulae use the following strategy: the function is approximate to a model that is easy to integrate analytically and that interpolates exactly \( n \) support points \( (x_i, f_i) \). In practice, all the proposed formulae use a polynomial with an adequate degree.

Of all the possible representations of this polynomial, the Lagrange representation is particularly suitable, since it allows us to easily evaluate the weights \( w_i \) of (1.2).

In fact, the interpolating polynomial is

\[ P_{n-1}(x) = \sum_{i=1}^{n} f_i L_i(x) \]  

where the Lagrange polynomials do not depend on the specific function \( f(x) \) considered:

\[ L_i(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_1)(x_i - x_2) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)} \]  

Given \( x_i \), the \( w_i \) values are easily calculated as follows:

\[ w_i = \int_a^b L_i(x)dx \]  

For example, selecting the points \( x_1 = a, x_2 = (a + b)/2, x_3 = b \), the weights are as follows:

\[ w_1 = \int_a^b \frac{(x - x_2)(x - b)}{(a - x_2)(a - b)} \]  
\[ w_2 = \int_a^b \frac{(x - a)(x - b)}{(x_2 - a)(x_2 - b)} \]  
\[ w_3 = \int_a^b \frac{(x - a)(x - x_2)}{(b - a)(b - x_2)} \]
and the integration formula is the *Cavalieri–Simpson* rule. Denoting $h$ as the distance between two successive points:

$$h = x_3 - x_2 = x_2 - x_1 = \frac{b - a}{2}$$

it results in

$$I = \frac{h}{3} (f_1 + 4f_2 + f_3)$$

### 1.3 Accuracy of Numerical Methods

For many rules like (1.2), it is possible to obtain an explicit expression of the local error. For other algorithms, it is only possible to know the order of magnitude of this error.

It is opportune to remark that the local error of an algorithm is evaluated by assuming that no numerical errors are present in both calculations and data (see Vol. 1, Buzzi-Ferraris and Manenti, 2010a). Since all the formulae use a polynomial that exactly interpolates the support points $(x_i, f_i)$, the local error depends on both the values of selected support abscissas $x_i$ and the problem itself.

When the points are evenly spaced, it is usual to express the *order m of local error of the algorithm as a function of the integration step* $h$: $O(h^m)$.

For example, the local error for the *trapezoid* rule is $-h^3 f^{(2)}(\xi)/12$ with $a \leq \xi \leq b$. Thus, the local error is on the order of $O(h^3)$.

When the algorithm has several not evenly spaced points inside the interval $[a; b]$, it is common to express the *order m of local error of the algorithm as a function of the interval* $[a; b]$: $O((b - a)^m)$.

An algorithm is *of p order* if it exactly integrates a polynomial of $(p - 1)$-degree, but it is inexact for $p$-degree polynomials. Many authors indicate this order as the *precision of the algorithm*.

For instance, the local error for the *trapezoid* rule is $-h^3 f^{(2)}(\xi)/12$ and it is exact for 1-degree polynomials; therefore, the order of the *trapezoid* rule is $p = 2$.

It is worth remarking that the local error expression is valid only if the hypotheses, which we assumed to calculate it, are verified. Specifically, it is essential to have no discontinuities in the function and derivatives up to a certain order according to the algorithm.

An increase in the *order of the algorithm*, $p$, or of its *local error order* $m$ does not necessarily lead to an increase of accuracy.
1.4
Modification of the Integration Interval

Integration formulae are usually given with particular values of \( a \) and \( b \), such as an interval \([0; 1]\) or \([-1; 1]\). In these cases, it is necessary to adapt them to the particular problem we are solving.

Let us denote with \([\alpha; \beta]\) the interval in which a specific formula is valid (where \( x_i \) and \( w_i \) are known):

\[
\int_{\alpha}^{\beta} f(x) \, dx = \sum_{i=1}^{n} w_i f_i, \quad n \geq 1
\] (1.9)

To calculate the integral

\[
I = \int_{a}^{b} g(t) \, dt
\] (1.10)

when \( a \) and \( b \) are both definite, it is possible to perform the variable transformation:

\[
t = \frac{(b - a)x + a\beta - ab}{\beta - \alpha}
\] (1.11)

Hence, it results in

\[
I = \frac{b - a}{\beta - \alpha} \int_{\alpha}^{\beta} g \left( \frac{(b - a)x + a\beta - ab}{\beta - \alpha} \right) \, dx
\]

\[
\approx \frac{b - a}{\beta - \alpha} \sum_{i=1}^{n} w_i g \left( \frac{(b - a)x_i + a\beta - ab}{\beta - \alpha} \right)
\] (1.12)

For example, the Gauss–Legendre formula with three points is valid for the interval \([-1; 1]\) and uses the points

\[
x_1 = -\sqrt{\frac{3}{5}}; \quad x_2 = 0; \quad x_3 = \sqrt{\frac{3}{5}}
\] (1.13)

with the weights

\[
w_1 = \frac{5}{9}; \quad w_2 = \frac{8}{9}; \quad w_3 = \frac{5}{9}
\] (1.14)

If it is applied to the integral

\[
\int_{0}^{1} \exp(-t) \, dt
\] (1.15)
it results in $t = (x + 1)/2$ and hence

$$I = \frac{1}{2} \int_{-1}^{1} \exp \left( -\frac{x+1}{2} \right) dx$$

$$\approx \frac{1}{18} \left[ 5 \exp \left( \frac{\sqrt{3/5} - 1}{2} \right) + 8 \exp(-0.5) + 5 \exp \left( -\frac{\sqrt{3/5} - 1}{2} \right) \right]$$

$$= 0.6321203$$

(1.16)

1.5 Main Integration Methods

Many algorithms have been proposed to perform the numerical integration of functions. We consider only two families of algorithms, which are the basis for the development and implementation of an even number of general programs for the numerical integration: the Newton–Cotes and the Gauss formulae.

1.5.1 Newton–Cotes Formulae

The Newton–Cotes formulae use a constant distance between the points within the integration interval. They can be close, open, or semiopen and they allow us to obtain the expression of the local error depending on $h$.

It results in the following:

- The trapezoid rule (also known as the trapezoidal rule or trapezium rule):

$$I = \frac{h}{2} \left[ f_1 + f_2 \right] - \frac{h^3}{12} f^{(3)}(\xi)$$

and analogously

- the Cavalieri–Simpson rule:

$$I = \frac{h}{3} \left[ f_1 + 4f_2 + f_3 \right] - \frac{h^5}{90} f^{(4)}(\xi)$$

(1.18)

- 3/8 rule:

$$I = \frac{3h}{8} \left[ f_1 + 3f_2 + 3f_3 + f_4 \right] - \frac{3h^5}{80} f^{(4)}(\xi)$$

(1.19)

- Boole’s rule:

$$I = \frac{2h}{45} \left[ 7f_1 + 32f_2 + 12f_3 + 32f_4 + 7f_5 \right] - \frac{8h^7}{945} f^{(6)}(\xi)$$

(1.20)
The order of the rules increases with the number of points while \( h \) decreases simultaneously. This could lead to the assumption that it is always suitable to increase the number of points to have an easier convergence to the problem solution.

For many practical problems, the close forms of Newton–Cotes formulae diverge from the solution, while the points are increased. In other words, it is not suitable to use high orders of Newton–Cotes formulae.

The reason for this divergence is that higher-degree interpolating polynomials do not perform as well in representing a function with evenly spaced points (see Vol. 2, Buzzi–Ferraris and Manenti, 2010b). Moreover, in the case of high-order Newton–Cotes formulae, certain coefficients become negative, consequently worsening numerical precision due to the difference between numbers of the same order of magnitude (see Vol. 1, Buzzi–Ferraris and Manenti, 2010a).

The only open Newton–Cotes of practical interest is the midpoint rule:

\[
I = (b - a) f\left(\frac{a + b}{2}\right) + \frac{(b - a)^2}{24} f^{(2)}(\xi) \tag{1.21}
\]

The other open and semiopen Newton–Cotes formulae are of purely historical interest: the open formulae are less effective than the Gauss formulae, and both the open and semiopen formulae are harder than the close formulae in their extensions (see Section 1.6.1).

1.5.2 Gauss Formulae

Gauss formulae exploit the positions of \( x_i \) as degrees of freedom to increase precision by preserving the number of points where the function is evaluated.

If the points are assigned a priori (i.e., evenly spaced), a formula with \( n \) points is exact for a \((n - 1)\)-degree polynomial; if we also exploit the \( n \) degrees of freedom related to the positions of \( x_i \), it is possible to have an exact formula for \((2n - 1)\)-degree polynomials.

Suppose we build a formula:

\[
I = \int_a^b r(x)f(x)dx = \sum_{i=1}^{n} w_i f(x_i), \quad n \geq 1 \tag{1.22}
\]

which is exact when \( f(x) \) is a polynomial smaller than or equal to \( 2n - 1 \).

In (1.22), \( r(x) \) is a particular function weight and its scope will be clarified later. If \( x_i \) for \( i = 1, \ldots, n \) were known, the \((n - 1)\)-degree polynomial passing through the \( n \) points \((x_i, f_i)\) would have the property (see Vol. 2, Buzzi–Ferraris and Manenti, 2010b):

\[
f(x) = P_{n-1}(x) + \frac{(x - x_1)(x - x_2)\cdots(x - x_n)f^{(n)}(\xi)}{n!} \tag{1.23}
\]
If \( f(x) \) is a \((2n - 1)\)-degree polynomial, the \( n \)th derivative \( f^{(n)}(\xi) \) must be a \((n - 1)\)-degree polynomial:

\[
f^{(n)}(\xi) = Q_{n-1}(x)
\]

(1.24)
since it is the only way to have a \((2n - 1)\)-degree polynomial after the product with the polynomial \((x - x_1)(x - x_2) \cdots (x - x_n)\).

In this case, the integral (1.22) results in

\[
I = \int_a^b r(x)f(x)dx = \int_a^b r(x)P_{n-1}(x)dx
\]

(1.25)
\[
+ \int_a^b r(x) \frac{(x - x_1)(x - x_2) \cdots (x - x_n)Q_{n-1}(x)}{n!} dx
\]

Suppose that the values of \( a \) and \( b \) and the function weight \( r(x) \) are such that a family of orthogonal polynomials \( Z_k(x) \) exists:

\[
\int_a^b r(x)Z_k(x)Z_j(x)dx = 0, \quad k \neq j
\]

(1.26)
For example, if \( a = -\infty, b = \infty, r(x) = 1 \), Legendre polynomials are orthogonal.

The aim of \( r(x) \) is to facilitate the selection of the appropriate family of orthogonal polynomials for different \( a \) and \( b \), by remarking that the procedure is efficient if \( f(x) \) is a polynomial (or, at least, well representable by a polynomial).

For example, if \( a = -\infty, b = \infty \), the family of Hermite orthogonal polynomials can be used; therefore, it results in \( r(x) = \exp(-x^2) \).

Since the \( n \) zeroes of an orthogonal \( n \)-degree polynomial are all real and distinct, it is possible to use the roots of the \( n \)-degree polynomial \( Z_n \), which belongs to the family of polynomials that make (1.26) valid, as points \( x_i \).

\( Z_n(x) \) can be written in the power form:

\[
Z_n(x) = a_n(x - x_1)(x - x_2) \cdots (x - x_n)
\]

(1.27)

If we develop the polynomial \( Q_{n-1}(x) \) as a series of polynomials coming from the family of \( Z_n \):

\[
Q_{n-1}(x) = \sum_{k=0}^{n-1} b_k Z_k(x)
\]

(1.28)
and consider the polynomial orthogonality, the latter integral of (1.25) is equal to zero and then

\[
I = \int_a^b r(x)f(x)dx = \int_a^b r(x)P_{n-1}(x)dx
\]

(1.29)
\[
= \sum_{i=1}^{n} \int_a^b r(x)L_i(x)dx = \sum_{i=1}^{n} w_i f_i
\]
when \( f(x) \) is a \((2n - 1)\)-degree polynomial.
To use the Gauss formulae, it is necessary to know the zeroes of the polynomial \( Z_n \) and the weights \( w_i \). They are tabulated for many families of orthogonal polynomials and for many combinations of \( a \) and \( b \) and \( r(x) \).

For example, the Gauss–Legendre method \((a = -1, \ b = 1, \ r(x) = 1)\) with four points and of order 8 requires the following values:

\[
\begin{align*}
    x_1 &= -0.8611363115940526 & w_1 &= 0.3478548451374538 \\
    x_2 &= -0.3399810435848563 & w_2 &= 0.6521451548625461 \\
    x_3 &= +0.3399810435848563 & w_3 &= 0.6521451548625461 \\
    x_4 &= +0.8611363115940526 & w_4 &= 0.3478548451374538
\end{align*}
\] (1.30)

To see other examples of Gauss’s methods of different orders, we remind to web databases.

An estimation of the local error is also provided for each Gauss rule family (that depends on the function \( r(x) \) and the values of \( a \) and \( b \)). For instance, in the important Gauss–Legendre method, we have the following error estimation (Kahaner, Moler, and Nash, 1989):

\[
\frac{(b - a)^{2n+1}(n!)^4}{(2n + 1)(2n)!} f^{2n}(\xi), \quad a < \xi < b \] (1.31)

The order of Gauss formulae that uses \( n \) internal points is \( p = 2 \cdot n \).

Another important family of methods is the set of Gauss–Radau formulae. In this family, just one of the two extremes of interval is a support point. They are therefore semiopen algorithms.

The order of Gauss–Radau formulae that use \( n - 1 \) internal points and an extreme of the interval is \( p = 2 \cdot n - 1 \).

Support points and weights for the Radau algorithms with one and two internal points are given in Tables 1.1 and 1.2, respectively. We refer readers to web databases for other algorithms from this family.

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5000000000000000</td>
<td>0.5000000000000000</td>
</tr>
<tr>
<td>1.5000000000000000</td>
<td>1.5000000000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.222222222222222</td>
<td>1.0249716523768433</td>
</tr>
<tr>
<td>0.75280612540093450</td>
<td>0.75280612540093450</td>
</tr>
</tbody>
</table>
If the extreme point of the interval is the upper point, 1, the formulae are symmetric with respect to the previous ones.

A third family of methods of relevant interest is the Gauss–Lobatto. In this family, the two extremes of the interval are support points and, thus, the formulae are close.

The order of Gauss–Lobatto formulae that uses \( n \) internal points beyond the interval extremes is \( p = 2 \cdot n - 2 \).

Note that the formula of Gauss–Lobatto with three points is equal to the Cavalieri–Simpson rule.

In the BzzMath library, the BzzIntegralGaussLobatto class uses the algorithms of Gauss–Lobatto with 5 and 7 internal points. Points and weights are reported in Tables 1.3 and 1.4.

For other examples of Gauss–Lobatto methods of different orders, we refer readers to web databases. In Section 1.10.3, we will demonstrate how to implement these algorithms to obtain a very efficient algorithm.

### 1.6 Algorithms Derived from the Trapezoid Method

The trapezoid rule is of little practical interest, but it lays the foundation for efficient calculation programs since it has certain very appealing features.

To understand the structure of these programs, it is essential to analyze the following points:

- Extended Newton–Cotes formulae
- Error of the extended trapezoid rule and the extended middle point rule
- Extrapolation of the formulae

#### Table 1.3 Values of \( x_i \) and \( w_i \) for the Gauss–Lobatto formula with five points.

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\pm 1.0</td>
<td>0.1 000 000 000 000 000</td>
</tr>
<tr>
<td>\pm 0.6546536707079777</td>
<td>0.5 444 444 444 444 444</td>
</tr>
<tr>
<td>0.0</td>
<td>0.7 111 111 111 111 111</td>
</tr>
</tbody>
</table>

#### Table 1.4 Values of \( x_i \) and \( w_i \) for the Gauss–Lobatto formula with five points.

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\pm 1.0</td>
<td>0.04 761 904 761 994 762</td>
</tr>
<tr>
<td>\pm 0.8302238962785669</td>
<td>0.27 682 604 736 156 594</td>
</tr>
<tr>
<td>\pm 0.4688487934707142</td>
<td>0.43 174 538 120 986 262</td>
</tr>
<tr>
<td>0.0</td>
<td>0.48 761 904 761 904 762</td>
</tr>
</tbody>
</table>
1.6.1

**Extended Newton–Cotes Formulae**

If a close Newton–Cotes formula is iteratively applied to adjacent intervals, the *extended Newton–Cotes formulae* are obtained and they exploit the points shared by the adjacent intervals.

In the case of the trapezoid rule, we have

\[
\int_{x_i}^{x_n} f(x) \, dx = T_h + O\left((b - a)h^2f^{(2)}\right)
\]

with

\[
T_h = h \left[ \frac{f_1}{2} + f_2 + f_3 + \cdots + \frac{f_n}{2} \right]
\]

(1.32)

It is worth remarking that the error of the extended formula is \(O(h^2)\) since it is equal to the sum of local errors (\(O(h^3)\) each) within the single intervals.

The extended *Cavalieri–Simpson* formula is obtained in an analogous way:

\[
\int_{x_i}^{x_n} f(x) \, dx = S_h + O\left((b - a)h^4f^{(5)}\right)
\]

with

\[
S_h = \frac{h}{3} \left[ f_1 + 4f_2 + 2f_3 + 4f_4 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n \right]
\]

(1.34)

Finally, the extended central point:

\[
\int_{x_i}^{x_n} f(x) \, dx = M_h + O\left((b - a)h^2f^{(2)}\right)
\]

with

\[
M_h = h \left[ f_{1+1/2} + f_{2+1/2} + f_{3+1/2} + \cdots + f_{n+1/2} \right]
\]

(1.36)

and

\[
x_{i+1/2} = a + \left( i - \frac{1}{2} \right) h
\]

(1.37)

(1.38)

is the central point of the interval between \(x_i\) and \(x_{i+1}\) with width \(h\).

There are certain features that make the extended trapezoid rule particularly interesting.

The first important feature of the trapezoid formula is that if we double the integration points (i.e., if we change from an integration step \(h\) to \(h/2\)), the previous points can be used without any recalculation.

For example, let us consider the interval \([a = 0; b = 4]\). If we adopt the extended trapezoid method for integration with step \(h = 2\), the points we use...
are \( x_1 = 0, x_2 = 2, \) and \( x_3 = 4 \). If we halve the integration step \( (h = 1) \), the function must be evaluated in \( x_1 = 0, x_2 = 1, x_3 = 2, x_4 = 3, \) and \( x_5 = 4 \). Thus, the previous function calculations in correspondence with \( x_1, x_3, x_5 \) can be exploited.

Note that the points needed with the extended trapezoid formula when the integration step is halved correspond to the points needed by the central point formula and the previous integration step.

In the example above, the new points \( x_2 = 1 \) and \( x_4 = 3 \) are needed to integrate the function through the extended central point and an integration step \( h = 2 \).

Certain programs, which implement the extended trapezoid formula, exploit the property:

\[
T_{h/2} = \frac{T_h + M_h}{2}
\]  

(1.39)

For example, the integral

\[
\int_1^2 \left( x^2 + \frac{1}{x} \right) dx = 3.026481
\]

We obtain

\begin{align*}
  h &= 1 & T &= 3.25 & M &= 2.916667 \\
  h &= 0.5 & T &= 3.08333 & M &= 2.998214 \\
  h &= 0.25 & T &= 3.040774 & M &= 3.019345 \\
  h &= 0.125 & T &= 3.03059 & M &= 3.024692 \\
  h &= 0.0625 & T &= 3.027375 & M &= 3.026033 \\
  h &= 0.03125 & T &= 3.026704 & M &= 3.026368 \\
  h &= 0.015625 & T &= 3.026536 & M &= 3.026453 \\
  h &= 0.0078125 & T &= 3.026495 & M &= 3.026473 \\
  h &= 0.00390625 & T &= 3.026484
\end{align*}

Note that the convergence speed of the extended trapezoid rule is quite slow. Thus, the method has to be used in this form only when the integral does not require a massive computation effort.

1.6.2 Error in the Extended Formulae

A second useful feature in the extended trapezoid rule is in the special form of its local error.

In fact, the error is given by the Euler–MacLaurin relation:

\[
T_h = \int_a^b f(x)dx + \sum_{k=1}^{\infty} \frac{B_{2k}h^{2k}}{(2k)!} \left[ f^{(2k-1)}b - f^{(2k-1)}a \right]
\]  

(1.40)

where the coefficients \( B_{2k} \) are the Bernoulli numbers.
For many problems, the relation (1.40) can be written in the form

\[ T_h = \int_a^b f(x)dx + \sum_{k=1}^{N} C_k h^{2k} + R_{N+1}(h) \]  
(1.41)

in which coefficients \( C_k \) are independent from \( h \) and \( R_{N+1} \to 0 \) with \( O(h^{2N+2}) \).

This relation means that in many practical cases, the extended trapezoid formula has an error that depends on the even powers of \( h \) only.

An analogous relation is also valid for the extended central point method and it is possible in many practical cases to write

\[ M_h = \int_a^b f(x)dx + \sum_{k=1}^{N} D_k h^{2k} + Q_{N+1}(h) \]  
(1.42)

It is therefore possible to combine the two methods to have an estimation of the error. This point will be demonstrated later in the chapter.

1.6.3 Extrapolation of the Extended Formulae

The relations (1.41) and (1.42) allow extrapolation techniques to be used. Only the relevant steps are reported hereinafter for the extrapolation technique; we refer readers to Vol. 2 (Buzzi-Ferraris and Manenti, 2010b) for detailed description:

1) Error estimation. By indicating with \( T_h \) and \( T_{2h} \) two applications of the extended trapezoid rule with integration steps \( h \) and \( 2h \), respectively, the error with step \( h \) is estimated:

\[ Ch^2 = \frac{T_h - T_{2h}}{3} \]  
(1.43)

Such an estimation is good if

\[ T_h - T_{2h} \approx 4T_{h/2} - T_h \]  
(1.44)

2) Improving the integral calculation. Once the error is estimated, it can be used to improve the integral estimation:

\[ T_h^* = T_h + \frac{T_h - T_{2h}}{3} \]  
(1.45)

\[ = \frac{h}{3} (f_1 + 4f_2 + 2f_3 + 4f_4 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n) = S_h \]

Note that (1.45) coincides with (1.35) of the extended Cavalieri–Simpson method.