How to learn Applied Mathematics through modern FORTRAN

Juan A. Hernández Ramos
Javier Escoto López

Department of Applied Mathematics
School of Aeronautical and Space Engineering
Technical University of Madrid (UPM)

Portrait:
Find your inspiration.
"Under the stars on a summer night" (August 2017).
Photography by Juan Ignacio Ruíz–Gopegui.
Cover design by Belén Moreno Santamaría.

ISBN 979-8604287071

Preface

During the process of learning mathematics, it is crucial to understand the theoretical basis associated with each concept. However, this necessary condition is not sufficient for a deep understanding of most mathematical problems. It is required to work the concepts with concrete examples and exercises in which numerical calculations are enlightening. For example, it is quite hard to get an intuitive vision of complex mathematical problems without working with their solutions. The solution of a mathematical problem involves designing an algorithm that is then implemented in a programming language. A natural question that can arise in this line of thought is: How closely could the programming language be of the mathematical language? It is not easy to provide an answer to this question. One main purpose of this book is to write programming codes with such a level of abstraction that the Fortran code resembles the mathematical language. To do that, profound knowledge of the Fortran language as well as the mathematical language is required.

Fortran is a strongly-typed language in which each data have a precise type, kind, and rank, and each subroutine or function states its communication requirements in terms of these types. Usually, when formulating a mathematical problem, a rigorous definition of the involved variables is stated. This is in consonance with a strongly-typed language. Besides, mathematical models are generally expressed in terms of functions and operators. Fortran language suits very elegantly with the functional paradigm. Even being possible to use the Fortran language to write object-oriented programming, the authors do not recommend this kind of programming with the Fortran language. The functional paradigm is much better suited. As in any other high-level programming language, the use of first-class functions allows easily the implementation of mathematical restrictions and functional operators. Fortran is a vector-based programming language dealing elegantly with mathematical operations between vectors and matrices of any finite vectorial space. All of the previously stated characteristics, along with the capabilities of a computer such as the graphical representation of data and the speed of computation, make the programming language a great help to understand many mathematical concepts. When treating with differential equations, which are very common in

physical problems, the use of programming is mandatory when there is no analytical solution. The effect of the different terms that are involved in an equation can be observed by changing the value of their coefficients and plotting the solution. Other phenomena, such as the dispersion and diffusion of waves when dealing with linear hyperbolic partial differential equations are much easier to be observed using plotting tools. From this list of examples, we can deduce that the use of programming languages serves as a great reinforcement in the development of intuition for mathematical problems.

The use of the functional paradigm of the Fortran language together with the construction of high-level abstractions are discussed in this book. Several mathematical problems of recurrent appearance in engineering are presented. Along with each type of problem, an algorithm for its resolution and its implementation in Fortran language are included. Moreover, an extended library of numerical methods accompanies this book. This library has two different abstraction layers: *(i)* the application layer in which subroutines are used as black boxes and *(ii)* the implementation layer that allows building different abstractions based on simpler concepts or functions.

The book is structured in three different parts: user manual, developer guidelines and Application Programming Interface (API). Part I, corresponding to the user manual, contains examples of mathematical problems of different natures and their resolution by means of the provided software. The second part describes each one of the problems and the algorithm that solves them. Besides, the deeper layer of the software is presented in order to help in the understanding of the functioning of the code. The final part, the Application Programming Interface, enlists for each type of mathematical problem the available subroutines and functions that compound the corresponding modules. The subprograms are defined by their interface in terms of their inputs and outputs. Every part is divided into chapters, one for each type of a mathematical problem.

Chapter 1 treats common topics from elementary operations with vectors and matrices from linear algebra and equations expressed in terms of elementary functions. The resolution of linear systems by LU decomposition, finding zeros of a nonlinear function by means of the Newton-Raphson method, eigenvalues, and eigenvectors by means of power method and SVD decomposition. These two latter methods are also applied to the computation of the condition number of a matrix, which gives information about its sensitivity.

Chapter 2 deals with the interpolation problem, focusing on polynomial interpolation by means of Lagrange polynomials. The Lagrange polynomials, their integrals, and derivatives are computed. The presented interpolation methods are used by the module that computes numerical derivatives of a function.

In Chapter 3, the computation of high order derivatives by means of Finite differences is carried out. This process is key to solving partial differential equations problems as boundary value problems or evolution problems in spatial domains. The computation of the derivatives is used to discretize the spatial domain by means of Lagrange interpolation. High order finite difference methods permit to transform the differential operator defined on a continuum domain to a differential

operator that is evaluated in a finite discrete set of points.

During Chapter 4 the initial value problem for ordinary differential equations, or also called Cauchy problem, is presented. This problem is of great applications in engineering problems. Starting by most problems of classical mechanics, such as orbital movements, or the attitude control of a satellite which are typical space applications. In other words, any problem that can be modeled as a first-order ordinary differential equation whose solution evolves from a certain initial value. This is achieved by means of temporal schemes that approximate the derivative of the differential equation.

In Chapter 5, the Boundary Value Problem is presented. This problem consists of determining a scalar or a vector field defined over a certain spatial domain governed by a differential equation with boundary conditions. Its applicability goes from structural static problems to thermal distributions or any physical problem that can be modeled as a partial differential equation with boundary conditions.

During Chapter 6, the Initial Value Boundary Problem is treated. This problem consists of an evolution problem for a vector field over a spatial domain, satisfying at each instant certain boundary conditions. Many classical problems such as the heat equation or the waves equation are governed by this kind of mathematical model. The method of resolution uses finite differences to discretize the spatial domain and transform the problem into a Cauchy problem, which is solved by the methods stated in Chapter 4.

Finally, in Chapter 7, mixed problems that include an initial value boundary problem coupled with a boundary value problem are presented. Complex physics such as the vibration of non-linear plates can be modeled using these types of problems.

We hope this book helps the student to understand profound concepts related to numerical mathematical problems and what is more important, from the point of view of authors, demystifying the chasm between programming and mathematics by making programming as beautiful and formal as the mathematical formulation of a problem.

Juan A. Hernández
Javier Escoto
Madrid, Septiembre 2019

Introduction

The following book is intended to serve as a guide for graduate students of engineering and scientific disciplines. Particularly, it has been developed thinking on the students of the Technical Superior School of Aeronautics and Space Engineering (ETSIAE) from Polytechnic University of Madrid (UPM). The topics presented cover many of the mathematical problems that appear on the different subjects of aerospace engineering problems. Far from being a classical textbook, with proofs and extended theoretical descriptions, the book is focused on the application and computation of the different problems. For this, for each type of mathematical problem, an implementation in Fortran language is presented. A complete library with different modules for each topic accompanies this book. The goal is to understand the different methods by directly by plotting numerical results and by changing parameters to evaluate the effect. Later, the student is advised to modify or to create his own code by studying the developer part of this book.

A complete set of libraries and the software code which is explained is this book can be downloaded from the repository:

`https://github.com/jahrWork/NumericalHUB.`

This repository is in continuous development by the authors. Once the compressed file is downloaded, Fortran sources files comprising different libraries as well as a Microsoft Visual Studio solution called `NumericalHUB.sln` can be extracted. If the reader is not familiar with the Microsoft Integrated Development Environment (IDE), it is highly recommended to read the book *Programming with Visual Studio: Fortran & Python & C++ & WEB projects. Amazon Kindle Direct Publishing 2019* . This book describes in detail how to manage big software projects by means of the Microsoft Visual Studio environment. Once the Microsoft Visual Studio is installed, the software solution `NumericalHUB.sln` allows running the book examples very easily.

The software solution `NumericalHUB.sln` comprises a set of extended examples of different simulations problems. Once the software solution `NumericalHUB.sln` is loaded and run, the following simple menu appears on the Command Prompt:

```
    write(*,*) "Welcome to NumericalHUB"

    write(*,*) " select an option "
    write(*,*) " 0. Exit/quit  "
    write(*,*) " 1. Systems of equations  "
    write(*,*) " 2. Lagrange interpolation  "
    write(*,*) " 3. Chebyshev interpolation  "
    write(*,*) " 4. ODE Cauchy problems   "
    write(*,*) " 5. Finite difference   "
    write(*,*) " 6. Boundary value problems  "
    write(*,*) " 7. Initial-boundary value problems  "
    write(*,*) " 8. Mixed problems: IBVP+BVP  "
    write(*,*) " 9. Advanced methods ODE methods "
```

Listing 1: `main_NumericalHUB.f90`

Each option is related to the different chapters of the book explained before. As was mentioned, the book is divided into three parts: Part I User, Part II Developer and Part III Application Program Interface (API) which share the same contents. From the user point of view, it is advised to focus on part I where easy examples are implemented and numerical results are explained. From the developer's point of view, part II explains in detail how different layers or levels of abstraction are implemented. This philosophy will allow the advanced user to implement his own codes. Part III of this book is intended to give a detailed API to use this software code by novel users or advanced users to create new codes for specific purposes.

Part I

User Manual

Chapter 1

Systems of equations

1.1 Overview

In this section, solutions of linear problems are obtained as well as the determination of zeroes of implicit functions. The first problem is the solution of a linear system of algebraic equations. LU factorization method (subroutine `LU_Solution`) and Gauss elimination method are proposed to obtain the solution. The natural step is to deal with solutions of a non linear system of equations. These systems are solved by the Newton method in the subroutine `Newton_Solution`. The eigenvalues problem of a given matrix is considered in the subroutine `Test_Power_Method` computed by means of the power method. Finally, to introduce the concept of conditioning of a matrix, the condition number of the Vandermonde matrix is computed in the subroutine `Vandermonde\_condition\_number`. All these subroutines are called from the subroutine `Systems_of_Equations_examples` which can be executed typing the first option of the main menu of the `NumericalHUB.sln` environment.

```
subroutine Systems_of_Equations_examples

    call LU_Solution
    call Newton_Solution
    call Implicit_explicit_equations
    call Test_Power_Method
    call Test_eigenvalues_PM
    call Vandermonde_condition_number
end subroutine
```

Listing 1.1: `API_Example_Systems_of_Equations.f90`

1.2 LU solution example

Let us consider the following system of linear equations:

$$Ax = b,$$

where $A \in \mathcal{M}_{4\times 4}$ and $b \in \mathbb{R}^4$ are:

$$A = \begin{pmatrix} 4 & 3 & 6 & 9 \\ 2 & 5 & 4 & 2 \\ 1 & 3 & 2 & 7 \\ 2 & 4 & 3 & 8 \end{pmatrix}, \qquad b = \begin{pmatrix} 3 \\ 1 \\ 5 \\ 2 \end{pmatrix}. \tag{1.1}$$

A common method to compute the solution to this problem is the LU factorization method. This method consists on the factorization of the matrix A into two simpler matrices L and U:

$$A = L\,U,$$

where L is a lower triangular matrix and U is an upper triangular matrix. This decomposition permits to calculate the solution x taking advantage of the factorization. Once the A matrix is factorized (subroutine `LU\_factorization`), the solution by means of a backward substitution can be obtained for any independent term b (subroutine `Solve\_LU`). The implementation of this problem is done as follows:

```
subroutine LU_Solution
    real :: A(4,4), b(4), x(4)
    integer :: i

    A(1,:) = [ 4, 3, 6, 9]
    A(2,:) = [ 2, 5,  4, 2]
    A(3,:) = [ 1, 3, 2, 7]
    A(4,:) = [ 2, 4, 3, 8]
    b = [ 3, 1, 5, 2]

    write (*,*) 'Linear system of equations  '
    write (*,*) 'Matrix of the system A= '
    do i=1, 4;   write(*,'(100f8.3)') A(i, :);  end do
    write (*,'(A20, 100f8.3)') 'Independent term b=  ', b
    write(*,*)

    call LU_factorization( A )
    x = Solve_LU( A , b )

    write (*,'(A20, 100f8.3)') 'The solution is = ', x
    write(*,*) "press enter " ; read(*,*)
end subroutine
```

Listing 1.2: `API_Example_Systems_of_Equations.f90`

Once the program is executed, the solution x results:

$$x = \begin{pmatrix} -7.811 \\ -0.962 \\ 4.943 \\ 0.830 \end{pmatrix}. \tag{1.2}$$

1.3 Newton solution example

Another common problem is to obtain the solution of a system of non linear equations. Iterative methods based on approximate solutions are always required. The rate of convergence and radius of convergence from an initial condition determine the election of the iterative method. The highest rate of convergence to the final solution is obtained with the Newton-Raphson method. However, the initial condition to iterate must be close to the solution to achieve convergence. Generally, this method is used when the initial approximation of the solution can be estimated approximately. To illustrate the use of this method, an example of a function $F : \mathbb{R}^3 \to \mathbb{R}^3$ is defined as follows:

$$\begin{aligned} F_1 &= x^2 - y^3 - 2, \\ F_2 &= 3\ x\ y - z, \\ F_3 &= z^2 - x. \end{aligned}$$

The implementation of the previous problem requires the definition of a vector function F as:

```
function F(xv)
    real, intent(in) :: xv(:)
    real:: F(size(xv))

    real :: x, y, z

    x = xv(1); y = xv(2); z = xv(3)

    F(1) = x**2 - y**3 - 2
    F(2) = 3 * x * y - z
    F(3) = z**2 - x

end function
```

Listing 1.3: `API_Example_Systems_of_Equations.f90`

The subroutine `Newton` gives the solution by means of an iterative method from the initial approximation `x0`. The solution is given in the same variable `x0`.

```
subroutine Newton_Solution

    real :: x0(3) = [1., 1., 1.  ]

    call Newton( F, x0 )

    write(*,*)  'Zeros of F(x) by Newton method '
    write(*,*)  'F(1) = x**2 - y**3 - 2'
    write(*,*)  'F(2) = 3 * x * y - z'
    write(*,*)  'F(3) = z**2 - x'
    write(*,'(A20, 100f8.3)')  'Zeroes of F(x) are x = ', x0
    write(*,*) "press enter "
    read(*,*)

end subroutine
```

Listing 1.4: `API_Example_Systems_of_Equations.f90`

Once the program is executed, the computed solution results:

$$(x, y, z) = (1.4219, 0.2795, 1.1924)$$

1.4 Implicit and explicit equations

Sometimes, the equations that governs a problem comprises explicit and implicit equations. For example, the following system of equations:

$$\begin{aligned} F_1 &= x^2 - y^3 - 2, \\ F_2 &= 3xy - z, \\ F_3 &= z^2 - x, \end{aligned}$$

can be expressed by maintaining two equations as implicit equations and one as an explicit equation:

$$\begin{aligned} x &= z^2, \\ F_1 &= x^2 - y^3 - 2, \\ F_2 &= 3xy - z. \end{aligned}$$

To solve this kind of problems, the subroutine `Newtonc` has been implemented. This subroutine takes into account that some equations are zero for all values of

the unknown x. In this case, the function $F(x)$ should provide explicit relationships for the components of x. An example of this kind of problem together with an initial approximation for the solution is shown in the following code:

```
subroutine Implicit_explicit_equations

    real :: x(3)
    write(*,*)  'Zeros of F(x) by Newton method '
    write(*,*)  'F(1) = x**2 - y**3 - 2'
    write(*,*)  'F(2) = 3 * x * y - z'
    write(*,*)  'F(3) = z**2 - x'

       x = 1
       call Newtonc( F = F1, x0 = x  )
       write(*,'(A35, 100f8.3)')  'Three implicit equations, x = ', x

       x = 1
       call Newtonc( F = F2, x0 = x  )
       write(*,'(A35, 100f8.3)')  'Two implicit + one explicit, x = ', x
       write(*,*) "press enter "; read(*,*)
contains

function F1(xv) result(F)
    real, target :: xv(:)
    real :: F(size(xv))

    real, pointer :: x, y, z

    x => xv(1); y => xv(2); z => xv(3)

    F(1) = x**2 - y**3 - 2
    F(2) = 3 * x * y - z
    F(3) = z**2 - x

end function

function F2(xv) result(F)
    real, target :: xv(:)
    real :: F(size(xv))

    real, pointer :: x, y, z

    x => xv(1); y => xv(2); z => xv(3)

    x =  z**2

    F(1) = 0 ! forall xv
    F(2) = x**2 - y**3 - 2
    F(3) = 3 * x * y - z

end function

end subroutine
```

Listing 1.5: API_Example_Systems_of_Equations.f90

1.5 Power method example

The determination of eigenvalues of square matrix are very valuable and also very challenging. If the matrix is symmetric, all eigenvalues are real and the determination of the eigenvalue with the largest module can be obtained easily by the power method. The power method is an iterative method that allows to determine the

Let us consider the symmetric matrix:

$$A = \begin{pmatrix} 7 & 4 & 1 \\ 4 & 4 & 4 \\ 1 & 4 & 7 \end{pmatrix}.$$

The determination of the largest eigenvalue is implemented in the following code by means of the power method:

```
subroutine Test_Power_Method

      integer, parameter :: N = 3
      real :: A(N, N), lambda, U(N)=1

      A(1,:) = [ 7, 4, 1 ]
      A(2,:) = [ 4, 4, 4 ]
      A(3,:) = [ 1, 4, 7 ]

      write (*,*) 'Power method  '
      call power_method(A, lambda, U)

      write(*,'(A10, f8.3, A10, 3f8.5)')  "lambda= ", lambda, "U= ", U
      write(*,*) "press enter "; read(*,*)

end subroutine
```

Listing 1.6: `API_Example_Systems_of_Equations.f90`

Once the program is executed, the largest eigenvalue yields:

$$\lambda = 12.00$$

and the associated eigenvector:

$$U = \begin{pmatrix} 0.5773 \\ 0.5773 \\ 0.5773 \end{pmatrix}.$$

Once the largest eigenvalue is obtained, it can be removed from the matrix A as well as its associated subspace. The new matrix A is obtained with the following expression

$$A_{ij} \rightarrow A_{ij} - \lambda_k \, U_i^k \, U_j^k .$$

Following this procedure, the next largest eigenvalue is obtained. This is done in the subroutine `eigenvalues_PM`.

An example of this procedure is shown in the following code where the eigenvalues of the same matrix A are calculated. The subroutine `eigenvalues_PM` calls the subroutine `power_method` and removes the calculated eigenvalues.

```
subroutine Test_eigenvalues_PM

 integer, parameter :: N = 3
 real :: A(N, N), lambda(N), U(N, N)
 integer :: i

 A(1,:) = [ 7, 4, 1 ]
 A(2,:) = [ 4, 4, 4 ]
 A(3,:) = [ 1, 4, 7 ]

 call Eigenvalues_PM(A, lambda, U)

 do i=1, N
    write(*,'(A8, f8.3, A15, 3f8.3)')  &
          "lambda = ", lambda(i),  "eigenvector = ", U(:,i)
 end do

end subroutine
```

Listing 1.7: `API_Example_Systems_of_Equations.f90`

Once the program is executed, the eigenvalues yield:

$$\lambda_1 = 12.00, \qquad \lambda_2 = 6.00, \qquad \lambda_3 = 1.33 \cdot 10^{-15},$$

and the associated eigenvectors:

$$U_1 = \begin{pmatrix} 0.5773 \\ 0.5773 \\ 0.5773 \end{pmatrix}, \qquad U_2 = \begin{pmatrix} -0.7071 \\ -8.5758 \cdot 10^{-13} \\ 0.7071 \end{pmatrix}, \qquad U_3 = \begin{pmatrix} 0.5773 \\ 0.5773 \\ 0.5773 \end{pmatrix}.$$

Notice that as λ_3 is null, the third eigenvector U_3 is the same as the first eigenvector U_1.

1.6 Condition number example

When solving a linear system of equations,

$$A\ x = b,$$

it is important to bound the error of the computed solution δx. Since the independent term b and the matrix A are entered in the computer as approximate values due to the round–off errors, the propagated error of the solution must be known.

The condition number $\kappa(A)$ of a matrix A is defined as:

$$\kappa(A) = \|A\|\|A^{-1}\|\,,$$

and allows to bound the error of the computed solution of a system of linear equations by the following expression:

$$\frac{\|\delta x\|}{\|x\|} \ \leq\ \kappa(A)\ \frac{\|\delta b\|}{\|b\|}.$$

The condition number of 6×6 Vandermonde matrix A is calculated in the following code:

```
subroutine Vandermonde_condition_number

    integer, parameter :: N = 10
    real :: A(N, N), kappa
    integer :: i, j

    do i=1, N; do j=1, N;
        A(i,j) = (i/real(N))**j
    end do; end do

    kappa = Condition_number(A)

    write(*,*)  'Condition number of Vandermonde matrix '
    write(*,'(A40, e10.3)') " Condition number (power method) =", kappa
    write(*,*) "press enter "; read(*,*)
end subroutine
```

Listing 1.8: `API_Example_Systems_of_Equations.f90`

Once this code is executed, the result given for the condition number is:

$$\kappa(A) = 0.109E + 09,$$

which indicates that the Vandermonde matrix is *ill-conditioned*. When solving a linear system of equations where the system matrix A is the Vandermonde matrix, a small error in the independent term b will be amplified by the condition number giving rise to large errors in the solution x.

Chapter 2

Lagrange interpolation

2.1 Overview

In this chapter, Lagrange and Chebyshev polynomial interpolations are discussed for equispaced and non uniform grids or nodal points. Given a set of nodal points x_i, their images through a given function $f(x)$ allow to build a polynomial interpolant. Examples are shown to alert of numerical problems associated to equispaced grid points. The subroutine `{Lagrange_Interpolation_examples` includes different examples to show the origin of these ill conditioning problems. To cure this problem, Chebyshev points are used to build interpolants.

```
subroutine Lagrange_Interpolation_examples

     call Interpolated_value_example
     call Interpolant_example
     call Integral_example

     call Lagrange_polynomial_example
     call Ill_posed_interpolation_example

     call Lebesgue_and_PI_functions

     call Chebyshev_polynomials
     call Interpolant_versus_Chebyshev

end subroutine
```

Listing 2.1: `API_Example_Lagrange_interpolation.f90`

All functions and subroutines used in this chapter are gathered in a Fortran module called: `Interpolation`. To make use of these functions the statement: `use Interpolation` should be included at the beginning of the program.

2.2 Interpolated value

In this first example, a set of six points is given:

$$x = [0, 0.1, 0.2, 0.5, 0.6, 0.7],$$

and the corresponding images of some unknown function $f(x)$:

$$f = [0.3, 0.5, 0.8, 0.2, 0.3, 0.6].$$

The idea is to interpolate or the predict with this information the value of $f(x)$ for $x_p = 0.15$. This is done in the following snippet of code:

```
subroutine Interpolated_value_example

    integer, parameter :: N = 6
    real :: x(N) = [ 0.0, 0.1, 0.2, 0.5, 0.6, 0.7 ]
    real :: f(N) = [ 0.3, 0.5, 0.8, 0.2, 0.3, 0.6 ]

    real :: xp = 0.15
    real :: yp(N-1)
    integer :: i

    do i=2, N-1
      yp(i) = Interpolated_value( x , f , xp, i )
      write (*,'(A10, i4, A40, f10.3)') 'Order = ', i, 'The interpolated
          value at xp is = ', yp(i)
    end do


    write (*,'(A20, 10f8.3)') 'xp = ', xp
    write (*,'(A20, 10f8.3)') 'nodes x_j = ', x
    write (*,'(A20, 10f8.3)') 'function f_j = = ', f
    write(*,*) "press enter "
    read(*,*)

end subroutine
```

Listing 2.2: `API_Example_Lagrange_interpolation.f90`

The first argument of the `Interpolated_value` function is the set of nodal points, the second argument is the set of images and the third argument is the order of the interpolant. The polynomials interpolation is built by piecewise interpolants of the desired order. Note that this third argument is optional. When it is not present, the function assumes that the interpolation order is two.

2.3 Interpolant and its derivatives

In this example an interpolant is evaluated for a complete set of points. Given a set of nodal or interpolation points:

$$\boldsymbol{x} = \{x_i \,|\, i = 0, \ldots, N\}, \qquad \boldsymbol{f} = \{f_i \,|\, i = 0, \ldots, N\}.$$

The interpolant and its derivatives are evaluated in the following set of equispaced points:

$$\{x_{pi} = a + (b-a)i/M, \quad i = 0, \ldots, M\}.$$

Note that in the following example that the number of nodal points is $N = 3$ and the number of points where this interpolant and its derivatives are evaluated is $M = 400$.

```
subroutine Interpolant_example

 integer, parameter :: N=3, M=400
 real ::  xp(0:M)
 real :: x(0:N) = [ 0.0, 0.1, 0.2, 0.5 ]
 real :: f(0:N) = [ 0.3, 0.5, 0.8, 0.2 ]
 real :: I_N(0:N, 0:M)  ! first index: derivative
                        ! second index: point where the interpolant is
                            evaluated
 real :: a, b
 integer :: i

  a = x(0); b = x(N)
  xp = [ (a + (b-a)*i/M, i=0, M) ]

  I_N = Interpolant(x, f, N, xp)

  write(*,*) "It plots an interpolant and its derivative"
  write(*,*) "press enter "; read(*,*)
  call plot_parametrics(xp, transpose(I_N(0:1,:)),["I", "dI/dx"],"x","y")


end subroutine
```

Listing 2.3: `API_Example_Lagrange_interpolation.f90`

The third argument of the function `Interpolant` is the order of the polynomial. It should be less or equal to N. The fourth argument is the set of points where the interpolant is evaluated. The function returns a matrix `I_N` containing the interpolation values and their derivatives in x_p. The first index holds for the order of the derivative and second index holds for the point x_{pi}. On figure 2.1, the interpolant and its first derivative are plotted.

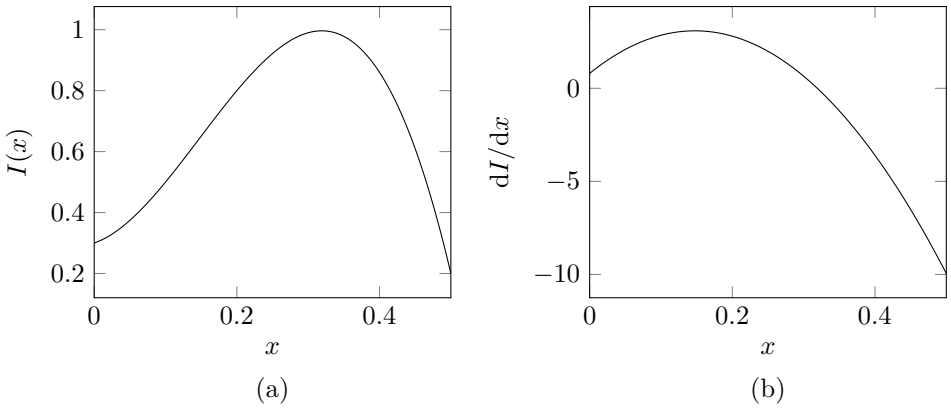

Figure 2.1: Lagrange interpolation with 4 nodal points. (a) Interpolant function. (b) First derivative of the interpolant.

2.4 Integral of a function

In this section, definite integrals are considered. Let's give the following example:

$$I_0 = \int_0^1 \sin(\pi x) dx.$$

To carry out the integral, an interpolant is built and later by integrating this interpolant the required value is obtained. The interpolant can be a piecewise polynomial interpolation of order $q < N$ or it can be a unique interpolant of order $q = N$. The function `Integral` has three arguments: the nodal points x, the images of the function f and the order of the interpolant q. In the following example, $N = 6$ equispaced nodal points are considered and integral is carried out with an interpolant of order $q = 4$. The result is compared with the exact value to know the error of this numerical integration.

```
subroutine Integral_example
 integer, parameter :: N=6
 real :: x(0:N), f(0:N), a = 0, b = 1, I0
 integer :: i
   x = [ (a + (b-a)*i/N, i=0, N) ]
   f = sin ( PI * x )

   I0 = Integral( x, f, 4 )
   write(*, *) "The integral [0,1] of sin( PI x ) is: ", I0
   write(*, *) "Error = ",  ( 1 -cos(PI) )/PI - I0
   write(*, *) "press enter ";  read(*,*)
end subroutine
```

Listing 2.4: `API_Example_Lagrange_interpolation.f90`

2.5 Lagrange polynomials

A polynomial interpolation $I_N(x)$ of degree N can be expressed in terms of the Lagrange polynomials $\ell_j(x)$ in the following way:

$$I_N(x) = \sum_{j=0}^{N} f_j \; \ell_j(x),$$

where f_j stands for the image of some function $f(x)$ in $N+1$ nodal points x_j and $\ell_j(x)$ is a Lagrange polynomial of degree N that is zero in all nodes except in x_j that is one. Besides, the sensitivity to round-off error is measured by the Lebesgue function and its derivatives defined by:

$$\lambda_N(x) = \sum_{j=0}^{N} |\ell_j(x)|, \qquad \lambda_N^{(k)}(x) = \sum_{j=0}^{N} |\ell_j^{(k)}(x)|.$$

In the following subroutine `Lagrange_polynomial_example`, the Lagrange polynomials and the Lebesgue function together with their derivatives are calculated for a equispaced grid of $N = 4$ nodal points. The first index of the resulting matrix `Lg` stands for the order of the derivative (k=-1 integral, k=0 function and k>0 derivative order). The second index identifies the Lagrange polynomial from $j = 0, \ldots, N$ and the third index stands for the point where the Lagrange polynomials or their derivatives are particularized. The same applies for the matrix `Lebesgue_N`. First index for the order of the derivative and second index for the point where the Lebesgue function or their derivatives are particularized.

```
subroutine Lagrange_polynomial_example
 integer, parameter :: N=4, M=400
 real :: x(0:N), xp(0:M), a=-1, b=1
 real :: Lg(-1:N, 0:N, 0:M)    ! Lagrange polynomials
                               ! -1:N (integral, lagrange, derivatives)
                               !  0:N ( L_0(x), L_1(x),... L_N(x)     )
                               !  0:M ( points where L_j(x) is evaluated  )
 real :: Lebesgue_N(-1:N, 0:M)
 character(len=2) :: legends(0:N) = [ "l0", "l1", "l2", "l3","l4" ]
 integer :: i

 x  = [ (a + (b-a)*i/N, i=0, N) ]
 xp = [ (a + (b-a)*i/M, i=0, M) ]

 do i=0, M;  Lg(:, :, i) = Lagrange_polynomials( x, xp(i) ); end do
 Lebesgue_N = Lebesgue_functions( x, xp )

 write (*, *) "It plots Lagrange functions"
 write(*,*) "press enter ";  read(*,*)
 call plot_parametrics(xp, transpose(Lg(0, 0:N, :)), legends,"x","y")
end subroutine
```

Listing 2.5: `API_Example_Lagrange_interpolation.f90`

On figure 2.2 the Lagrange polynomials and the Lebesgue function are shown. It is observed in figure 2.2a that ℓ_j values 1 at $x = x_j$ and 0 at $x = x_i$ with $j \neq i$. In figure 2.2b, the Lebesgue function together with its derivatives are presented.

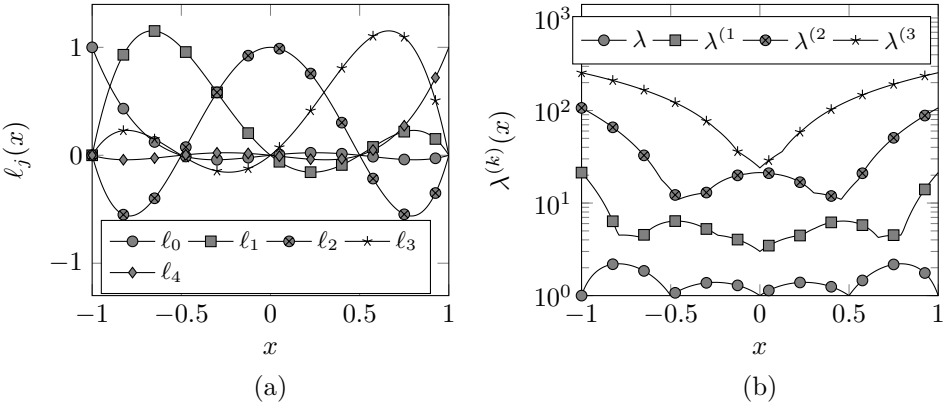

Figure 2.2: Lagrange polynomials $\ell_j(x)$ with $N = 4$ and Lebesgue function $\lambda(x)$ and its derivatives. (a) Lagrange polynomials $\ell_j(x)$ for $j = 0, 1, 2, 3, 4$. (b) Lebesgue function and its derivatives $\lambda^{(k)}(x)$ for $k = 0, 1, 2, 3$.

2.6 Ill–posed interpolants

When considering equispaced grid points, the Lagrange interpolation becomes ill–posed which means that a small perturbation like machine round-off error yields big errors in the interpolation result. In this section an interpolation example for the inoffensive $f(x) = \sin(\pi x)$ is analyzed to show the interpolant can have noticeable errors at boundaries.

The error is defined as the difference between the function and the polynomial interpolation

$$f(x) - I_N(x) = R_N(x) + R_L(x),$$

where $R_N(x)$ is the truncation error and $R_L(x)$ is the round–off error. Since the round–off error is present in the computer when any value is calculated, a polynomial interpolation $I_N(x)$ of degree N can be expressed in terms of the Lagrange polynomials $\ell_j(x)$ in the following way:

$$I_N(x) = \sum_{j=0}^{N} (f_j + \epsilon_j)\ \ell_j(x),$$

where ϵ_j can be considered as the round-off error of the image $f(x_j)$. Note that when working in double precision this ϵ_j is of order $\epsilon = 10^{-15}$. Hence, the error

of the interpolant has two components. The first one $R_N(x)$ associated to the truncation degree of the polynomial and the second one $r_L(x)$ associated to round-off errors. This error can be expressed by the following equation:

$$R_L(x) = \sum_{j=0}^{N} \epsilon_j \ \ell_j(x).$$

Although the exact values of the round–off errors ϵ_j are not not known, all values ϵ_j can be bounded by ϵ. It allows to bound the round–off error by the following expression:

$$|R_L(x)| \leq \epsilon \sum_{j=0}^{N} |\ell_j(x)|$$

which introduces naturally the Lebesgue function $\lambda_N(x)$. If the Lebesgue function reaches values of order 10^{15}, the round-off error becomes order unity.

In the following code, an interpolant for $f(x) = \sin(\pi x)$ with $N = 64$ nodal points is calculated together with the Lebesgue function. In figure 2.3a, the interpolant shows a considerable error at boundaries. It can be easily explained by means of figure 2.3b where the Lebesgue function is plotted. The Lebesgue function takes values of order 10^{15} close to the boundaries making the round-off error of order unity at the boundaries.

```
subroutine Ill_posed_interpolation_example

 integer, parameter :: N=64, M=300
 real :: x(0:N), f(0:N)
 real :: I_N(0:N, 0:M)
 real :: Lebesgue_N(-1:N, 0:M)
 real :: xp(0:M)
 real :: a=-1, b=1
 integer :: i

 x  = [ (a + (b-a)*i/N, i=0, N) ]
 xp = [ (a + (b-a)*i/M, i=0, M) ]
 f = sin ( PI * x )

 I_N = Interpolant(x, f, N, xp)
 Lebesgue_N = Lebesgue_functions( x, xp )

 write(*, *) "It plots an interpolant with errors at boundaries "
 write(*, *) "maxval Lebesgue =", maxval( Lebesgue_N(0,:) )

 write(*, *) "press enter "; read(*,*)
 call plot_parametrics(xp, transpose(I_N(0:0, :)), ["I"], "x", "y")

end subroutine
```

Listing 2.6: API_Example_Lagrange_interpolation.f90

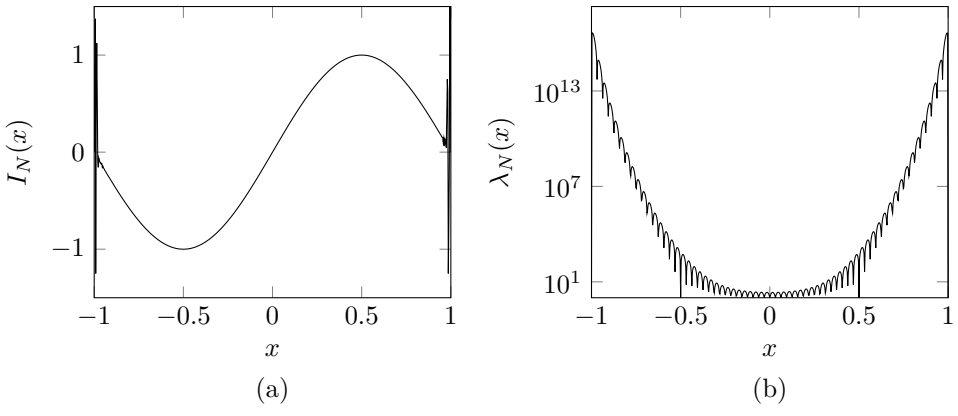

Figure 2.3: Interpolation for an equispaced grid with $N = 64$. (a) Ill posed interpolation $I_N(x)$, (b) Lebesgue function $\lambda_N(x)$.

2.7 Lebesgue function and error function

As it was mentioned in the preceding section, the interpolation error has two main contributions: the round-off error and the truncation error. In this section, a comparison of these two contributions is presented. It can be shown that the truncation error has the following expression:

$$R_N(x) = \pi_{N+1}(x)\frac{f^{(N+1)}(\xi)}{(N+1)!},$$

where π_{N+1} is a polynomial of degree $N+1$ and $f^{(N+1)}(\xi)$ represents the $N+1$–th derivative of the function $f(x)$ evaluate at some specific point $x = \xi$. The π_{N+1} polynomial vanishes in all nodal points and it is called the π error function.

In this section, the Lebesgue function $\lambda_N(x)$ and the error function $\pi_{N+1}(x)$ together with their derivatives are plotted to show the origin of the interpolation error.

In the following code, the π error function as well as the Lebesgue function $\lambda_N(x)$ are calculated for $N = 10$ interpolation points. A grid of $M = 700$ points is used to plot the results. In figure 2.4, the π error function is compared with the Lebesgue function $\lambda_N(x)$. Both the π error function and for the Lebesgue function show maximum values near the boundaries making clear that error will become more important near the boundaries. It is also observed that the Lebesgue values are greater than the π error function. However, it does not mean that the round–off error is greater than the truncation error because the truncation error depends on the regularity of $f(x)$ and the round–off error depends on the finite precision ϵ.

```
subroutine Lebesgue_and_PI_functions

 integer, parameter :: N=10, M=700
 real :: x(0:N), xp(0:M)
 real :: Lebesgue_N(-1:N, 0:M),  PI_N(0:N, 0:M)
 real :: a=-1, b=1
 integer :: i, k

   x  = [ (a + (b-a)*i/N, i=0, N) ]
   xp = [ (a + (b-a)*i/M, i=0, M) ]

   Lebesgue_N = Lebesgue_functions( x, xp )
   PI_N = PI_error_polynomial( x, xp )

   write(*, *) "It plots Lebesgue  functions"
   write(*, *) "function, first and second derivative"
   write(*, *) "press enter ";  read(*, *)
   call plot_parametrics(xp, transpose(Lebesgue_N(0:2, :)),  &
                         ["l", "dldx","d2ldx2"], "x","y")

   call plot_parametrics(xp, transpose(PI_N(0:2, :)),        &
                         ["pi", "dpidx","d2pidx2"], "x","y")


end subroutine
```

Listing 2.7: `API_Example_Lagrange_interpolation.f90`

What is also true is that the maximum value of the Lebesgue function grows with N and the maximum value of the π error function goes to zero with $N \to \infty$. Hence, with N great enough, the round–off error exceeds the truncation error.

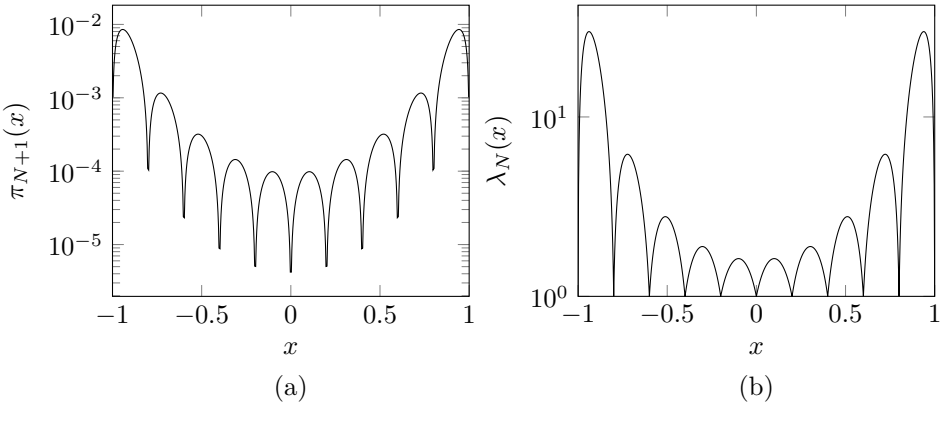

Figure 2.4: Error function $\pi_{N+1}(x)$ and Lebesgue function $\lambda_N(x)$ for $N = 10$. (a) Function $\pi_{N+1}(x)$. (b) Lebesgue function $\lambda_N(x)$

In figures 2.5 and 2.6, first and second derivatives of the π error function and the

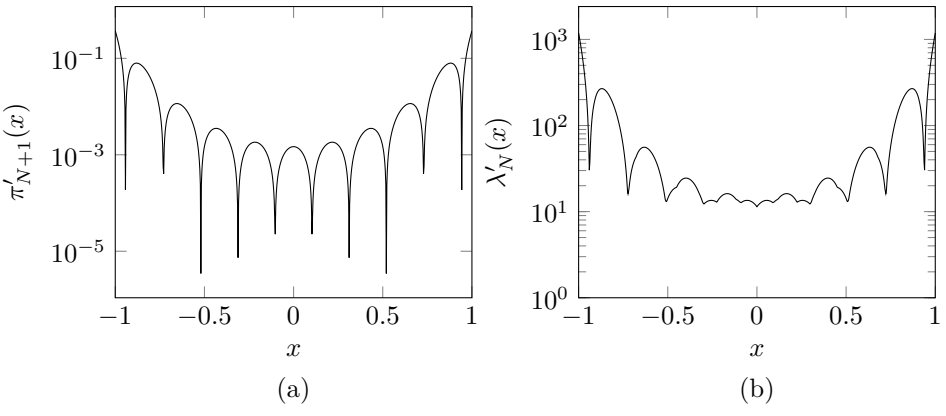

Figure 2.5: First derivative of the error function $\pi'_{N+1}(x)$ and Lebesgue function $\lambda'_N(x)$ for $N = 10$. (a) Function $\pi'_{N+1}(x)$. (b) Lebesgue function $\lambda'_N(x)$

Lebesgue function are shown and compared. It is observed that the first and second derivative of the Lebesgue function grow exponentially making more relevant the round–off error for the first and the second derivative of the interpolant. However, the derivatives of the π error function decreases with the order of the derivative.

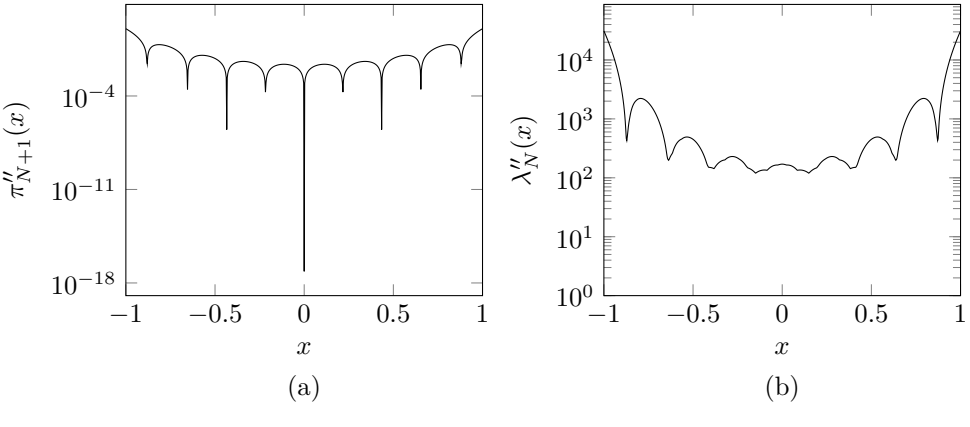

Figure 2.6: Second derivative of the error function $\pi''_{N+1}(x)$ and Lebesgue function $\lambda''_N(x)$ for $N = 10$. (a) Function $\pi''_{N+1}(x)$. (b) Lebesgue function $\lambda''_N(x)$

2.8 Chebyshev polynomials

Chebyshev polynomials have an important role in the polynomial interpolation theory. It will be shown in the next section that when some specific interpolation points are used, the polynomial interpolation results very close to a Chebyshev expansion. It makes important to revise the Chebyshev polynomials and their behavior. The approximation of $f(x)$ by means of Chebyshev polynomials is given by:

$$f(x) = \sum_{k=0}^{\infty} \hat{c}_k \; T_k(x),$$

where $T_k(x)$ are the Chebyshev polynomials and $\hat{c}_k$ are the projections of $f(x)$ in the Chebyshev basis. The are some special orthogonal basis very noticeable like Chebyshev polynomials. The first kind $T_k(x)$ and the second kind $U_k(x)$ of Chebyshev polynomials are defined by:

$$T_k(x) = \cos(k\theta), \qquad U_k(x) = \frac{\sin(k\theta)}{\sin\theta},$$

with $\cos\theta = x$.

In the following code, Chebyshev polynomials of order k of first kind and second kind are calculated for different values of x.

```
subroutine Chebyshev_polynomials

    integer, parameter :: N = 100, M = 5
    real :: x(0:N), theta(0:N), Tk(0:N, 0:M), Uk(0:N,0:M)
    real :: x0=-1, xf= 1
    integer :: i, k
    character(len=2) :: lTK(0:M) = ["T0","T1","T2","T3","T4","T5"]
    character(len=2) :: lUk(0:M) = ["U0","U1","U2","U3","U4","U5"]

    x = [ ( x0 + (xf-x0)*i/N, i=0, N ) ]
    do k=1, M
           theta = acos(x)
           Tk(:, k) = cos ( k * theta )
           Uk(:, k) = sin ( k * theta ) / sin (theta)
    end do

    write(*, *) "It plots Chebyshev polynomials"
    write(*, *) "press enter ";  read(*,*)
    call plot_parametrics(x, Tk(:, 0:M), lTk, "x","y")
    call plot_parametrics(x, Uk(:, 0:M), lUk, "x","y")
end subroutine
```

Listing 2.8: `API_Example_Lagrange_interpolation.f90`

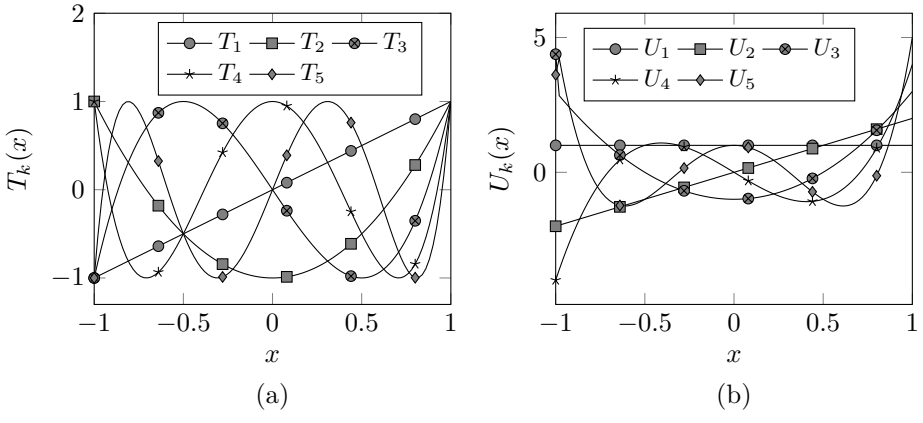

Figure 2.7: First kind and second kind Chebyshev polynomials. (a) First kind Chebyshev polynomials $T_k(x)$. (b) Second kind Chebyshev polynomials $U_k(x)$.

2.9 Chebyshev expansion and Lagrange interpolant

As it was shown in previous sections, when the interpolation points are equispaced, the error grows at boundaries no matter the regularity of the interpolated function $f(x)$. Hence, high order polynomial interpolation is prohibited with equispaced grid points. To cure this problem, concentration of grid points near the boundaries are usually proposed. One of the most important distribution of points that cure this bad behavior near the boundaries is the Chebyshev extrema

$$x_i = \cos\left(\frac{\pi i}{N}\right), \quad i = 0, \ldots, N.$$

In this section, a comparison between a Chebyshev expansion and a Lagrange interpolant is shown when the selected nodal points are the Chebyshev extrema. In the following code, a Chebyshev expansion `P_N` of $f(x) = \sin(\pi x)$ is calculated with 7 terms ($N = 6$). The coefficients $\hat{c}_k$ of the expansion are calculated by means of :

$$\hat{c}_k = \frac{1}{\gamma} \int_{-1}^{+1} \frac{f(x) T_k(x)}{\sqrt{1 - x^2}} dx.$$

In the same code a polynomial interpolation `I_N` based on the Chebyshev extrema is calculated. Errors for the expansion and for the interpolation are also obtained.

```
subroutine Interpolant_versus_Chebyshev

    integer, parameter :: N = 6  ! # of Chebyshev terms or poly order
    integer, parameter :: M = 500! # of points to plot
    real :: x(0:N), f(0:N)
    real :: I_N(0:N, 0:M)          ! Interpolant
    real :: P_N(0:M)               ! Truncated series
    real :: xp(0:M), theta(0:M)    ! domain to plot
    real :: Error(0:M, 2)          ! Error :interpolant and truncated
    real :: Integrand(0:M)
    character(len=8) :: legends(2) =  ["Error_I", "Error_P"]

    integer :: i, k
    real :: c_k, a=-1, b = 1, gamma

!** equispaced points to plot
    xp = [ (a + (b-a)*i/M, i=0, M) ]
    theta = acos( xp )

!** Chebyshev truncated series
    do k=0, N
        Integrand = sin( PI * xp) * cos ( k * theta )

        if (k==0) then;    gamma =  PI;
                  else;    gamma =  PI / 2;
        end if
        c_k = Integral( theta , Integrand ) / gamma
        P_N = P_N - c_k * cos( k * theta )
    end do

    x = [ (cos(PI*i/N), i=N, 0, -1) ]
    f = sin( PI * x )

!** Interpolant based on Chebyshev extrema
    I_N = Interpolant(x, f, N, xp)

    Error(:, 1) = sin( PI * xp ) - I_N(0, :)
    Error(:, 2) = sin( PI * xp ) - P_N
    call plot_parametrics(xp, Error(:, 1:2), legends, "x","y")

end subroutine
```

Listing 2.9: `API_Example_Lagrange_interpolation.f90`

In figure 2.8a, the truncated Chebyshev expansion `P_N` is plotted together with the polynomial interpolation `I_N` with no appreciable difference between them. This can be verified in figure 2.8b where the error for these two approximations are shown. It can be demonstrated that when choosing some specific nodal or grid points and if the function to be approximated is regular enough, the difference between the truncated Chebyshev expansion and the polynomial interpolation becomes very small. This difference is called aliasing error.

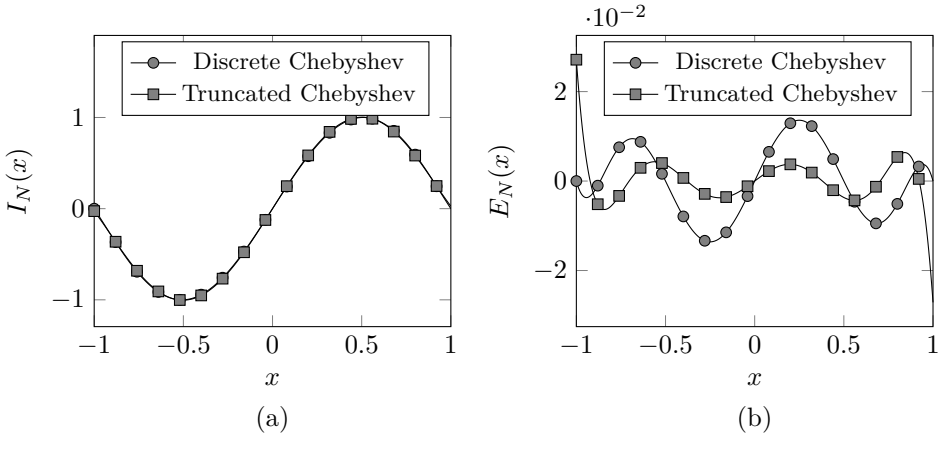

Figure 2.8: Chebyshev discrete expansion and truncated series for $N = 6$. (a) Chebyshev expansion. (b) Chebyshev expansion error.

Chapter 3

Finite Differences

3.1 Overview

From the numerical point of view, derivatives of some function $f(x)$ are always obtained by building an interpolant, deriving the interpolant analytically and later particularizing it at some point. When piecewise polynomial interpolation is considered, finite difference formulas are built to calculate derivatives.

In this chapter, several examples of derivatives are gathered in the subroutine `Finite_difference_examples`. The first example is run in the subroutine `Derivative_function_x` which calculates the derivatives of a function $f : \mathbb{R} \to \mathbb{R}$. The subroutine `Derivative_function_xy` performs the same for a function $f : \mathbb{R}^2 \to \mathbb{R}$. Finally, the influence of the truncation and round off errors with the spatial step between nodes are highlighted in the subroutine `Derivative_error`.

```
subroutine Finite_difference_examples


   call Derivative_function_x
   call Derivative_function_xy
   call Derivative_error


end subroutine
```

Listing 3.1: `API_Example_Finite_Differences.f90`

All functions and subroutines used in this chapter are gathered in a Fortran module called: `Finite_differences`. To make use of these functions the statement: `use Finite_differences` should be included at the beginning of the program.

3.2 Derivatives of a 1D function

As it was mentioned in the previous section, to calculate derivatives a polynomial interpolant should be built

$$I_N(x) = \sum_{j=0}^{N} f_j \ \ell_j(x).$$

To calculate the k–th derivative, the interpolant is derived analytically,

$$\frac{d^{(k)} I_N}{dx^k}(x) = \sum_{j=0}^{N} f_j \ \frac{d^{(k)} \ell_j}{dx^k}(x).$$

Finally, these derivatives are particularized at the nodal points,

$$\frac{d^{(k)} I_N}{dx^k}(x_i) = \sum_{j=0}^{N} f_j \ \frac{d^{(k)} \ell_j}{dx^k}(x_i).$$

In the following subroutine `Derivative_function_x`, the first and the second derivative of

$$u(x) = \sin(\pi x)$$

are calculated particularized at $N+1$ grid points x_i. First, a nonuniform grid $x_i \in [-1, +1]$ is created by the subroutine `Grid_initialization`. It builds internally a piecewise polynomial interpolant of order or degree 4, calculates the derivatives $k = 1, 2, 3$ of the Lagrange polynomials and particularizes their derivatives at x_i. The numbers or weights $\ell_j^{(k)}(x_i)$ are stored internally in a data structure to be used later by the subroutine `Derivative`. This subroutine multiplies the weights $\ell_j^{(k)}(x_i)$ by the function values $u(x_j)$ to yield the required derivative in the desired nodal point. The values $I_N^{(k)}(x_i)$ are stored in a matrix `uxk` of two indexes. The first index standing for the grid point x_i and the second one for the order of the derivative k.

In this case, a polynomial interpolation or degree 4 is used. It means that the polynomial valid around x_i is built with 5 surrounding points. If x_i is an interior

grid point, not close to the boundaries, these points are $\{x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}\}$ and the first and second derivatives give rise to:

$$\frac{du}{dx}(x_i) = \sum_{j=i-2}^{j=i+2} u(x_j)\ \ell_j^{(1)}(x_i), \qquad \frac{d^2u}{dx^2}(x_i) = \sum_{j=i-2}^{j=i+2} u(x_j)\ \ell_j^{(2)}(x_i).$$

The first argument of the subroutine `Derivative` is the direction `"x"` of the derivative, the second argument is the order of the derivative, the third one is the vector of images of $u(x)$ at the nodal points and the fourth argument is the derivative evaluated at the nodal points x_i ·

Additionally, the error of the first and second derivatives are calculated by subtracting the approximated value from the exact value of the derivative

$$E_1 = \left[\frac{\mathrm{d}I}{\mathrm{d}x}(x_i) - \pi\ \cos(\ \pi x_i\ )\right], \qquad E_2 = \left[\frac{\mathrm{d}^2I}{\mathrm{d}x^2}(x_i) + \pi^2\ \sin(\ \pi x_i\ )\right].$$

```
subroutine Derivative_function_x

    integer, parameter :: Nx = 20, Order = 4
    real :: x(0:Nx)
    real :: x0 = -1, xf = 1
    integer :: i
    real :: pi = 4 * atan(1.)
    real :: u(0:Nx), uxk(0:Nx, 2),  ErrorUxk(0:Nx, 2)

    x = [ (x0 + (xf-x0)*i/Nx, i=0, Nx) ]

    call Grid_Initialization( "nonuniform", "x", Order, x )

    u = sin(pi * x)

    call Derivative( 'x' , 1 , u , uxk(:,1) )
    call Derivative( 'x' , 2 , u , uxk(:,2) )


    ErrorUxk(:,1) = uxk(:,1) - pi* cos(pi * x)
    ErrorUxk(:,2) = uxk(:,2) + pi**2 * u

    write (*, *) 'Finite differences formulas: 4th order '
    write (*, *) 'First and second derivative of sin pi x  '
    write(*,*) "press enter ";   read(*,*)

    call plot_parametrics(x, Uxk, ["ux", "uxx"], "x", "y")


end subroutine
```

Listing 3.2: `API_Example_Finite_Differences.f90`

In figure 3.1, the first and second derivatives of $u(x) = \sin(\pi x)$ are plotted together with their numerical error.

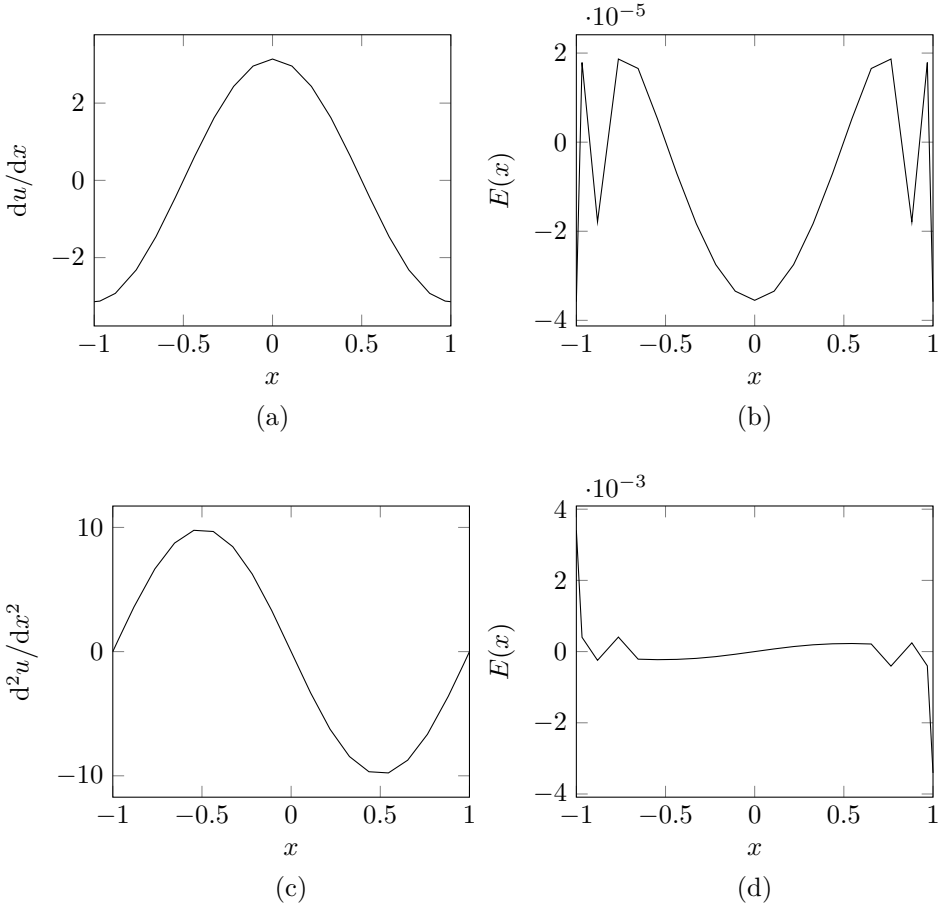

Figure 3.1: First and second derivatives of $u(x) = \sin(\pi x)$ by means of finite difference formulas and their associated error. The piecewise polynomial interpolation of degree 4 and 21 nodal points. (a) First numerical derivative $\mathrm{d}u/\mathrm{d}x$. (b) Error on the calculation of $\mathrm{d}u/\mathrm{d}x$. (c) Second numerical derivative $\mathrm{d}^2u/\mathrm{d}x^2$. (d) Error on the calculation of $\mathrm{d}^2u/\mathrm{d}x^2$.

3.3 Derivatives of a 2D function

In this section, a two dimensional space is considered and partial derivatives are calculated. It is considered the function:

$$u(x, y) = \sin(\pi x) \sin(\pi y).$$

The code implemented in subroutine `Derivative_function_xy` is similar to the preceding code of a 1D problem. Internally, the module `Finite_differences` interpolates in two orthogonal directions. This is done by calling twice the subroutine `Grid_Initiallization`. In this case, a piecewise polynomial interpolation of degree 6 is considered in both directions and 21 nodal points are used.

```
subroutine Derivative_function_xy

    integer, parameter :: Nx = 20, Ny = 20, Order = 6
    real :: x(0:Nx), y(0:Ny)
    real :: x0 = -1, xf = 1, y0 = -1, yf = 1
    integer :: i, j
    real :: pi = 4 * atan(1.0)
    real :: u(0:Nx,0:Ny), uxx(0:Nx,0:Ny), uy(0:Nx,0:Ny), uxy(0:Nx,0:Ny)
    real :: Erroruxx(0:Nx,0:Ny), Erroruxy(0:Nx,0:Ny)

    x = [ (x0 + (xf-x0)*i/Nx, i=0, Nx) ]
    y = [ (y0 + (yf-y0)*j/Ny, j=0, Ny) ]

    call Grid_Initialization( "nonuniform", "x", Order, x )
    call Grid_Initialization( "nonuniform", "y", Order, y )

    u = Tensor_product( sin(pi*x), sin(pi*y) )

    call Derivative( ["x", "y"], 1, 2, u, uxx )
    call Derivative( ["x", "y"], 2, 1, u, uy )
    call Derivative( ["x", "y"], 1, 1, uy, uxy )

    Erroruxx = uxx + pi**2 * u
    Erroruxy = uxy - pi**2 * Tensor_product( cos(pi*x), cos(pi*y) )

    write (*, *) '2D Finite differences formulas: 6th order '
    write (*, *) 'Second partial derivative with respect x'
    write (*, *) 'of  u(x,y) = sin pi x sin pi y  '
    write(*,*) "press enter ";  read(*,*)

    call plot_contour(x, y, uxx, "x", "y" )

end subroutine
```

Listing 3.3: `API_Example_Finite_Differences.f90`

The computed partial derivatives calculated by means of finite difference formulas together with their associated error are represented in figure 3.2

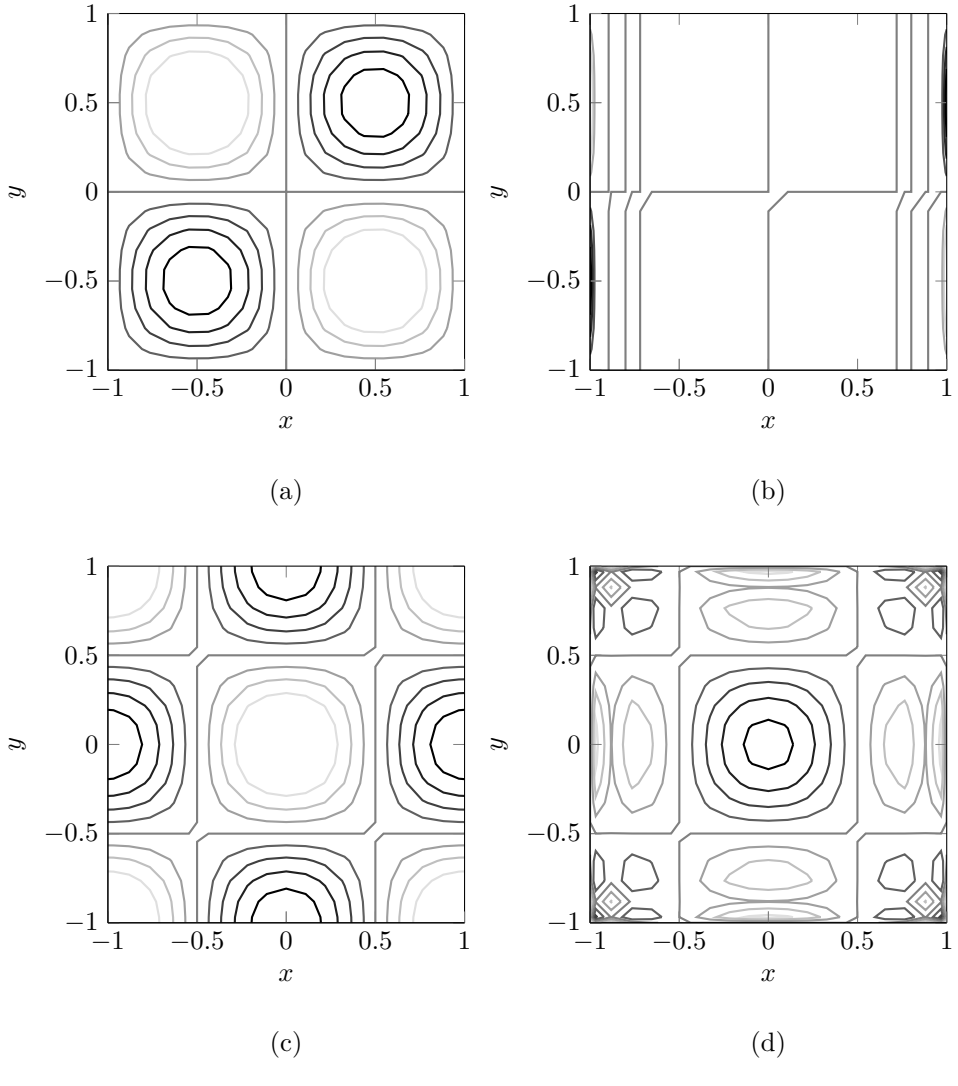

Figure 3.2: Numerical derivatives $\partial^2 u/\partial x^2$, $\partial^2 u/\partial x \partial y$ of $u(x, y) = \sin(\pi x)\ \sin(\pi y)$ and associated error for 21×21 nodal points and order $q = 6$. (a) Numerical derivative $\partial^2 u/\partial x^2$. (b) Error on the calculation of $\partial^2 u/\partial x^2$. (c) Numerical derivative $\partial^2 u/\partial x \partial y$. (d) Error on the calculation of $\partial^2 u/\partial x \partial y$.

3.4 Truncation and Round-off errors of derivatives

In order to analyze the effect of the truncation and round-off error produced in the approximation of a derivative by finite differences, the following example is considered. As it was shown in the Lagrange interpolation chapter, two contributions or errors are always present when interpolating some function: truncation error and round–off error. While the truncation error is reduced when decreasing the step size Δx between grid points, the round-off error is increased when reducing Δx. This is due to the growth of the Lebesgue function when reducing Δx.

In the following code, the second derivative for $f(x) = \cos(\pi x)$ is calculated with piecewise polynomial interpolation of degree $q = 2, 4, 6, 8$. The calculated value is compared with the exact value at some point x_p and the resulting is determined by:

$$E(x_p) = \frac{d^2 I}{dx^2}(x_p) - \pi^2 \ \cos(\pi x_p).$$

Since the grid spacing initialized in `Grid_initialization` is uniform, the step size $\Delta x = 2/N$. There are two loops in the code, the first one takes into account different degree q for the piecewise polynomial and the second one varies the number of grid points N and, consequently, the step size Δx. To measure the effect of the machine precision or errors associated to measurements, the function is perturbed by means of the subroutine `randon_number` with a module of order $\epsilon = 10^{-12}$ giving rise to the following perturbed values:

$$f(x) = \cos(\pi x) + \varepsilon(x).$$

The resulting error $E(x_p)$ is evaluated at the boundary $x_p = -1$ for each step size Δx.

In figure 3.3 and figure 3.4, the error E versus the step size Δx is plotted in logarithmic scale. As it is expected, when the step size Δx is reduced, the error decreases. However, when derivatives are calculated with smaller step sizes Δx, the round-off error becomes the same order of magnitude than the truncation error and the global error stops decreasing. When reducing, even more, the step size, the round-off error becomes dominant and the global error increases a lot. This behavior is observed in figure 3.3 and figure 3.4, which display the error at $x = -1$ and $x = 0$ respectively. As the order of interpolation grows, the minimum value of Δx grows too indicating that the round-off error starts being relevant for larger step sizes.

```
subroutine Derivative_error

 integer :: q                        ! interpolant order 2, 4, 6, 8
 integer :: Nq = 8                   ! max interpolant order
 integer :: N                        ! # of nodes (piecewise pol. interpol.)
 integer :: k = 2                    ! derivative order
 integer :: p = 0                    ! where error is evaluated p=0, 1,...N
 integer, parameter :: M = 100       ! number of grids ( N = 10,... N=10**4)
 real :: log_Error(M,4),log_dx(M)    ! Error versus Dx for q=2, 4, 6, 8
 real :: epsilon = 1d-12             ! order of the random perturbation

 real :: PI = 4 * atan(1d0), logN
 integer ::  j, l=0

 real, allocatable :: x(:), f(:), dfdx(:)  ! function to be interpolated
 real, allocatable :: dIdx(:)              ! derivative of the interpolant

 do q=2, Nq, 2
  l = l +1
  do j=1, M

   logN = 1 + 3.*(j-1)/(M-1)
   N = 2*int(10**logN)

   allocate( x(0:N), f(0:N),  dfdx(0:N), dIdx(0:N) )
   x(0) = -1; x(N) = 1

   call Grid_Initialization( "uniform", "x", q, x )

   call random_number(f)
   f = cos ( PI * x ) + epsilon * f
   dfdx = - PI**2 * cos ( PI * x )

   call Derivative( "x", k, f, dIdx )

   log_dx(j) = log( x(1) - x(0) )
   log_Error(j, l) = log( abs(dIdx(p) - dfdx(p)) )

   deallocate( x, f, dIdx, dfdx )
  end do
 end do

 call scrmod("reverse")
 write(*,*) "Second derivative error versus spatial step for q=2,4,6,8 "
 write(*,*) " Test function:  f(x) = cos pi x  "
 write(*,*) "press enter " ; read(*,*)

 call plot_parametrics( log_dx, log_Error, ["E2", "E4", "E6", "E8"], &
                       "log_dx","log_Error")

end subroutine
```

Listing 3.4: API_Example_Finite_Differences.f90

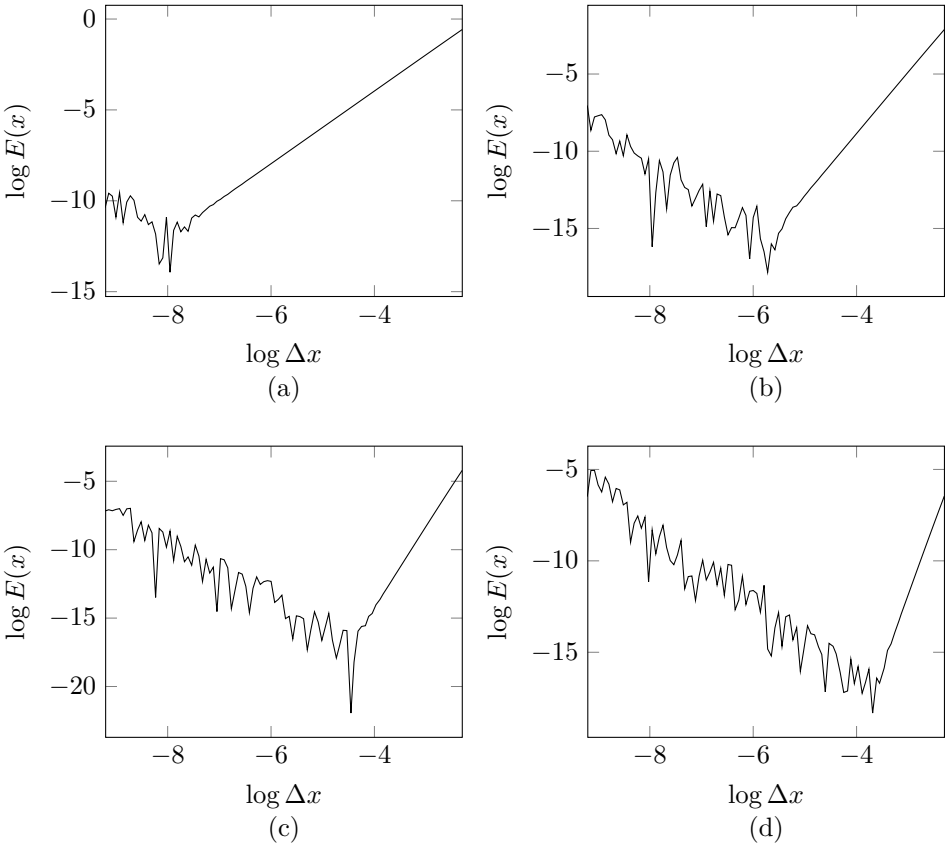

Figure 3.3: Numerical error of a second order derivative versus the step size Δx at $x = -1$. (a) Piecewise polynomials of degree $q = 2$. (b) $q = 4$. (c) $q = 6$. (d) $q = 8$.

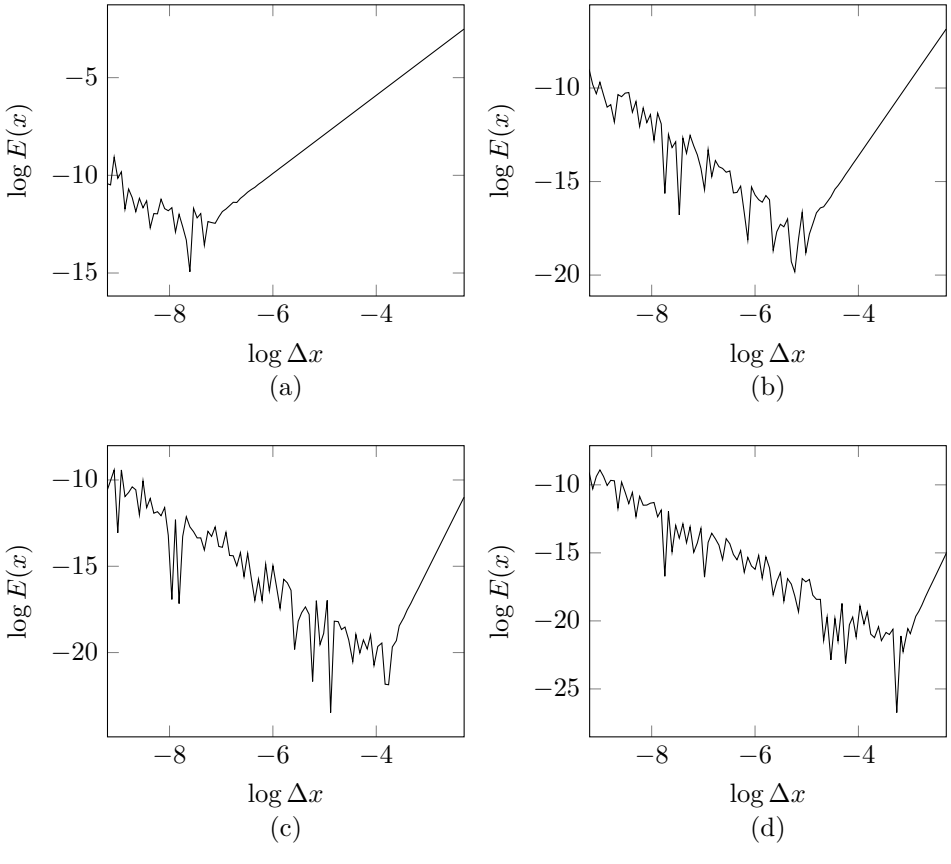

Figure 3.4: Numerical error of a second order derivative versus the step size Δx at $x = 0$. (a) Piecewise polynomials of degree $q = 2$. (b) $q = 4$. (c) $q = 6$. (d) $q = 8$.

Chapter 4

Cauchy Problem

4.1 Overview

In this chapter, some examples of the following Cauchy problem:

$$\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = \boldsymbol{f}\ (\boldsymbol{U},\ t), \quad \boldsymbol{U}(0) = \boldsymbol{U}^0, \qquad \boldsymbol{f} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$$

are presented.

It is started with a scalar first-order ordinary differential equation ($N = 1$) implemented in the `First_order_ODE`. The second example is devoted to the oscillations of a mass attached to a spring. The movement is governed by a second-order scalar equation implemented in `Linear_Spring`. The third example simulates the famous Lorenz attractor.

To alert of possible issues associated with numerical simulations, some other examples are shown. Absolute stability regions for a second and fourth-order Runge-Kutta method are obtained in `Stability_regions_RK2_RK4`. The absolute stability regions allow determining the stability of numerical simulations. In the subroutine `Error_solution`, the error associated to a numerical computation is discussed and finally, the convergence rate of the numerical solution to the exact solution is analyzed in the subroutine `Convergence_rate_RK2_RK4`.

All functions and subroutines used in this chapter are gathered in a Fortran module called: `Cauchy_problem`. To make use of these functions the statement: `use Cauchy_problem` should be included at the beginning of the program.

```
subroutine Cauchy_problem_examples


   call First_order_ODE
   call Linear_Spring
   call Lorenz_Attractor
   call Stability_regions_RK2_RK4
   call Error_solution
   call Convergence_rate_RK2_RK4

end subroutine
```

Listing 4.1: `API_Example_Cauchy_Problem.f90`

4.2 First order ODE

The following scalar first order ordinary differential equation is considered:

$$\frac{du}{dt} = -2u(t),$$

with the initial condition $u(0) = 1$. This Cauchy problem could describe the velocity along time of a punctual mass submitted to viscous damping. This problem has the following analytical solution:

$$u(t) = e^{-2t}.$$

The implementation of the problem requires the definition of the differential operator $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ as a function.

```
function F1( U, t ) result(F)
    real :: U(:), t
    real :: F(size(U))

    F(1) = -2*U(1)

end function
```

Listing 4.2: `API_Example_Cauchy_Problem.f90`

This function is used by the subroutine `Cauchy_ProblemS` to compute the numerical solution. Additionally, the time domain and the initial condition are required.

```
subroutine First_order_ODE

    real :: t0 = 0, tf = 4
    integer :: i
    integer, parameter :: N = 1000  !Time steps
    real :: Time(0:N), U(0:N,1)

    Time = [ (t0 + (tf -t0 ) * i / (1d0 * N), i=0, N ) ]
    U(0,1) =  1

    call Cauchy_ProblemS( Time_Domain = Time,                        &
                          Differential_operator = F1, Solution = U )

    write(*,*) "Solution of du/dt - 2 u"
    write(*,*)  "press enter "
    read(*,*)
    call plot_parametrics(Time, U, ["Solution of du/dt-2u"], "time", "u")

contains
```

Listing 4.3: `API_Example_Cauchy_Problem.f90`

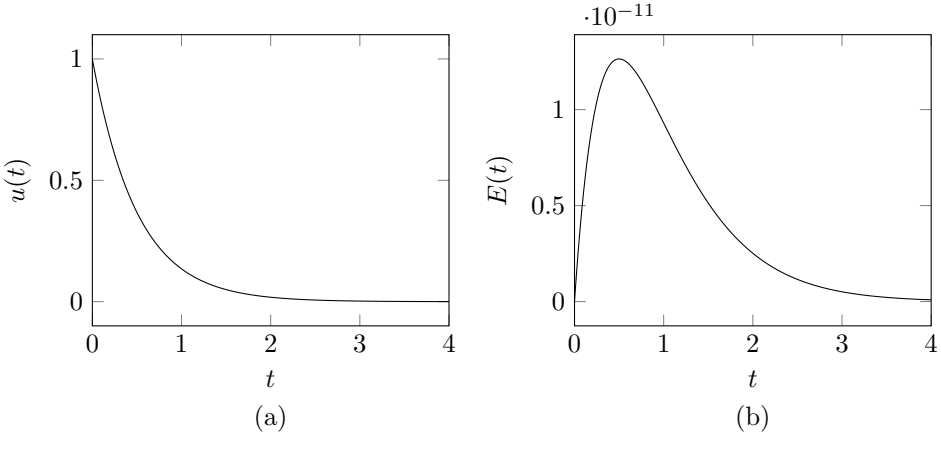

Figure 4.1: Numerical solution and error on the computation of the first order Cauchy problem. (a) Numerical solution of the first order Cauchy problem. (b) Error of the solution along time.

The numerical solution obtained using this code can be seen in figure 4.1. In it can be seen that the qualitative behavior of the solution $u(t)$ is the same as the described by the analytical solution. However, quantitative behavior is not exactly equal as it is an approximated solution. In figure 4.1(b) it can be seen that the solution tends to zero slower than the analytical one.

4.3 Linear spring

The second example is a second-order differential equation. It could represent the oscillations a punctual mass suspended by a linear spring whose stiffness increases along time. The problem is integrated in a temporal domain: $\Omega \subset \mathbb{R} : \{t \in [0, 4]\}$. The displacement $u(t)$ of the mass is governed by the equation:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} + \ a\ t\ u(t) = 0.$$

First of all, the problem must be formulated as a first order differential equation. This is done by means of the transformation:

$$u(t) = U_1(t), \qquad \frac{\mathrm{d}u}{\mathrm{d}t} = U_2(t),$$

which leads to the system:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ -a\ t & 0 \end{bmatrix} \begin{pmatrix} U_1 \\ U_2 \end{pmatrix}.$$

It is necessary to give an initial condition of position U_1 and velocity U_2. In this example, the movement starts with the mass with zero velocity and with the elongated spring.

$$\begin{pmatrix} U_1(0) \\ U_2(0) \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}.$$

The implementation of the problem requires the definition of the differential operator $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ as a vector function:

```
function F_spring( U, t )  result(F)

    real :: U(:), t
    real :: F(size(U))

    real, parameter :: a = 3.0

    F(1) = U(2)
    F(2) = -a * t * U(1)

end function
```

Listing 4.4: `API_Example_Cauchy_Problem.f90`

This function is used as an input argument for the subroutine `Cauchy_ProblemS`. In this example, the optional argument `Scheme` is used to select the `Crank_Nicolson` numerical scheme to integrate the problem. The solution `U` has two indexes. The first stands for the different time steps along the integration and the second one stands for the two variables of the system: position and velocity.

```
subroutine Linear_Spring
    integer :: i
    integer, parameter :: N = 100   !Time steps
    real :: t0 = 0, tf = 4, Time(0:N), U(0:N, 2)

    Time = [ (t0 + (tf -t0 ) * i / (1d0 * N), i=0, N ) ]
    U(0,:) = [ 5, 0]
    call Cauchy_ProblemS( Time_Domain = Time ,                        &
                          Differential_operator = F_spring,           &
                          Solution = U, Scheme = Crank_Nicolson       )

    write (*, *) 'Solution of the Cauchy problem:  '
    write (*, *) ' d2u/dt2 = -3 t u, u(0) = 5, du(0)/dt = 0'
    write(*,*) "press enter ";  read(*,*)
    call plot_parametrics(Time, U, ["Sd2u/dt2 = -3 t  u"], "time", "u")

contains
```

Listing 4.5: `API_Example_Cauchy_Problem.f90`

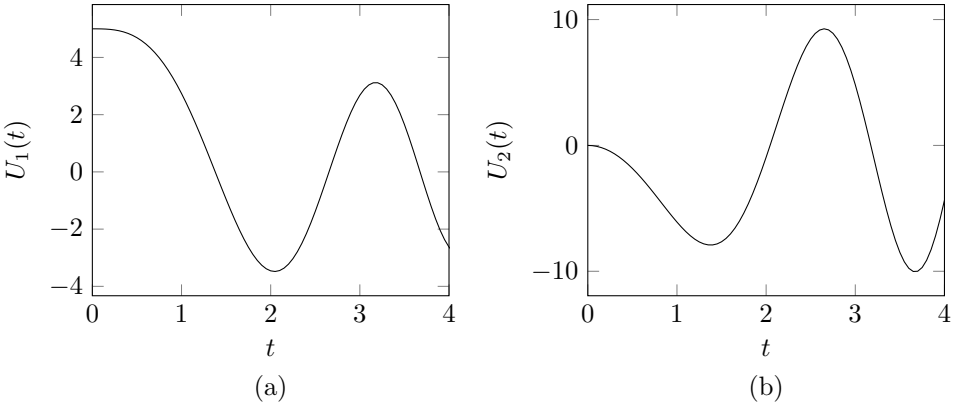

Figure 4.2: Numerical solution of the Linear spring movement. (a) Position along time. (b) Velocity along time.

The numerical solution of the problem is shown in figure 4.2. It can be seen how the initial condition for both U_1 and U_2 are satisfied and the oscillatory behavior of the solution.

4.4 Lorenz Attractor

Another interesting example is the differential equation system from which the strange Lorenz attractor was discovered. The Lorenz equations are a simplification of the Navier-Stokes fluid equations used to describe the weather behavior along time. The behavior of the solution is chaotic for certain values of the parameters involved in the equation. The equations are written:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a\ (y-z) \\ x\ (b-z) - y \\ x\ y - c\ z \end{pmatrix},$$

along with the initial conditions:

$$\begin{pmatrix} x(0) \\ y(0) \\ z(0) \end{pmatrix} = \begin{pmatrix} 12 \\ 15 \\ 30 \end{pmatrix}.$$

The implementation of the problem requires the definition of the differential operator $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ as a vector function:

```
function F_L(U, t) result(F)
    real :: U(:),t
    real :: F(size(U))

    real :: x, y , z

    x = U(1); y = U(2); z = U(3)

    F(1) =   a * ( y - x )
    F(2) =   x * ( b - z ) - y
    F(3) =   x * y - c * z

end function
```

Listing 4.6: `API_Example_Cauchy_Problem.f90`

The previous function will be used as an input argument for the subroutine that solves the Cauchy Problem. In this case, a fourth order Runge–Kutta scheme is used to integrate the problem.

```
subroutine Lorenz_Attractor
   integer, parameter :: N = 10000
   real :: Time(0:N), U(0:N,3)
   real :: a=10., b=28., c=2.6666666666
   real :: t0 =0, tf=25
   integer :: i

   Time = [ (t0 + (tf -t0 ) * i / (1d0 * N), i=0, N ) ]

   U(0,:) = [12, 15, 30]

    call Cauchy_ProblemS( Time_Domain=Time, Differential_operator=F_L, &
                          Solution = U, Scheme = Runge_Kutta4 )


    write (*, *) 'Solution of the Lorenz attractor  '
    write(*,*) "press enter " ; read(*,*)
    call plot_parametrics(U(:,1),U(:,2:2), ["Lorenz attractor"],"x","y")

contains
```

Listing 4.7: `API_Example_Cauchy_Problem.f90`

The chaotic behaviour appears for the values $a = 10$, $b = 28$ and $c = 8/3$. When solved for these values, the phase planes of $(x(t), y(t))$ and $(x(t), z(t))$ show the famous shape of the Lorenz attractor. Both phase planes can be observed on figure 4.3.

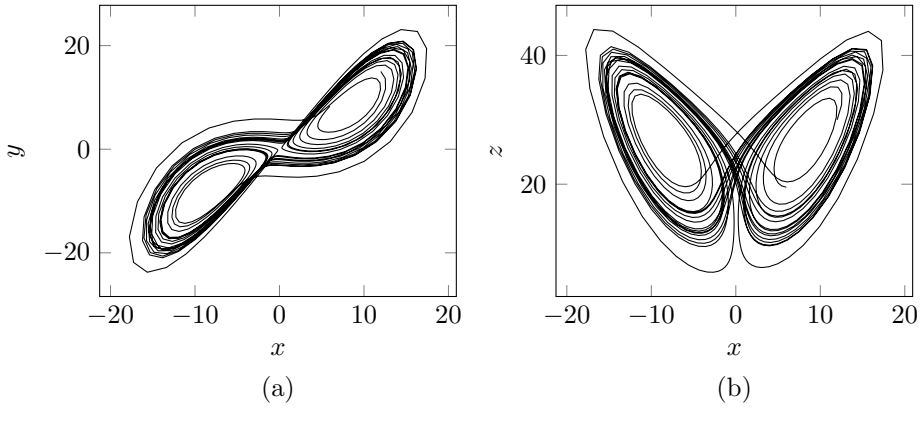

Figure 4.3: Solution of the Lorenz equations. (a) Phase plane (x, y) of the Lorenz attractor. (b) Phase plane (x, z) of the Lorenz attractor.

4.5 Stability regions

One of the capabilities of the library is to compute the region of absolute stability of a given temporal scheme. In the following example, the stability regions of second-order and fourth-order Runge-Kutta methods are determined.

```
do j=1, 2
  if (j==1)      then
      call Absolute_Stability_Region(Runge_Kutta2, x, y, Region)
  else if (j==2) then
      call Absolute_Stability_Region(Runge_Kutta4, x, y, Region)
  end if

  call plot_contour(x, y, Region, "$\Re(z)$","$\Im(z)$", levels, &
                    legends(j), path(j), "isolines")
end do
```

Listing 4.8: `API_Example_Cauchy_Problem.f90`

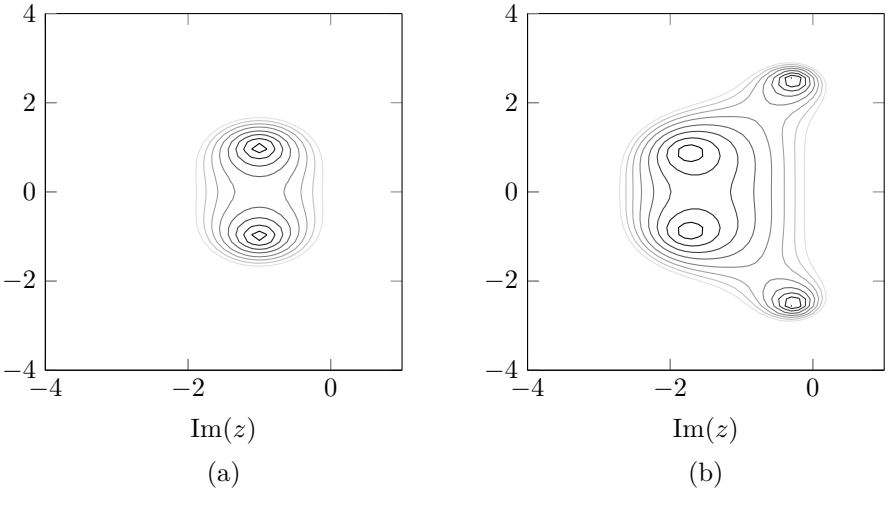

Figure 4.4: Absolute stability regions. (a) Stability region of second order Runge-Kutta. (b) Stability region of fourth order Runge-Kutta.

4.6 Richardson extrapolation to calculate error

The library also computes the error of the obtained solution of a Cauchy problem using the Richardson extrapolation. The subroutine `Error_Cauchy_Problem` uses internally two different step sizes Δt and $\Delta t/2$, respectively, and estimates the error as:

$$E = \frac{\|\boldsymbol{u_1}^n - \boldsymbol{u_2}^n\|}{1 - 1/2^q},$$

where E is the estimated error, u_1^n is the solution at the final time calculated with the given time step, u_2^n is the solution at the final time calculated with $\Delta t/2$ and q is the order of the temporal scheme used for calculating both solutions.

This example estimates the error of a Van der Pol oscillator using a second-order Runge-Kutta.

```
call Error_Cauchy_Problem( Time, VanDerPol_equation,        &
                           Runge_Kutta2, 2, Solution, Error )
```

Listing 4.9: `API_Example_Cauchy_Problem.f90`

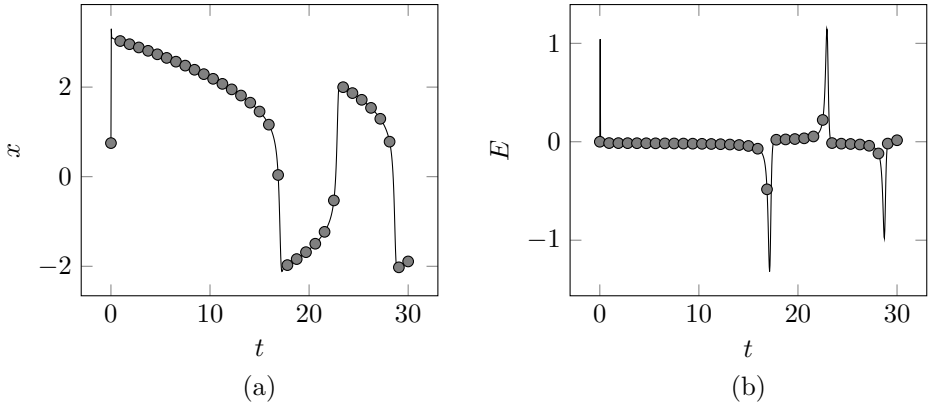

Figure 4.5: Integration of the Van der Pol oscillator. (a) Van der Pol solution, (b) Error of the solution.

In figure 4.5 the solution together with its error is plotted. Since the error varies significantly along time, the variable time step is required to maintain error under tolerance.

4.7 Convergence rate with time step

A temporal scheme is said to be of order q when its global error with $\Delta t \to 0$ goes to zero as $O(\Delta t^q)$. It means that high order numerical methods allow bigger time steps to reach a precise error tolerance. The subroutine `Temporal_convergence_rate` determines the error of the numerical solution as a function of the number of time steps N. This subroutine internally integrates a sequence of refined Δt_i and, by means of the Richardson extrapolation, determines the error.

In the following example, the error or convergence rate of a second and fourth-order Runge-Kutta for the Van der Pol oscillator are obtained.

```
call Temporal_convergence_rate( Time, VanDerPol_equation,  U0,      &
                                Runge_Kutta2, 2, log_E(:,1), log_N )

call Temporal_convergence_rate( Time, VanDerPol_equation,  U0,      &
                                Runge_Kutta4, 4, log_E(:,2), log_N )
```

Listing 4.10: `API_Example_Cauchy_Problem.f90`

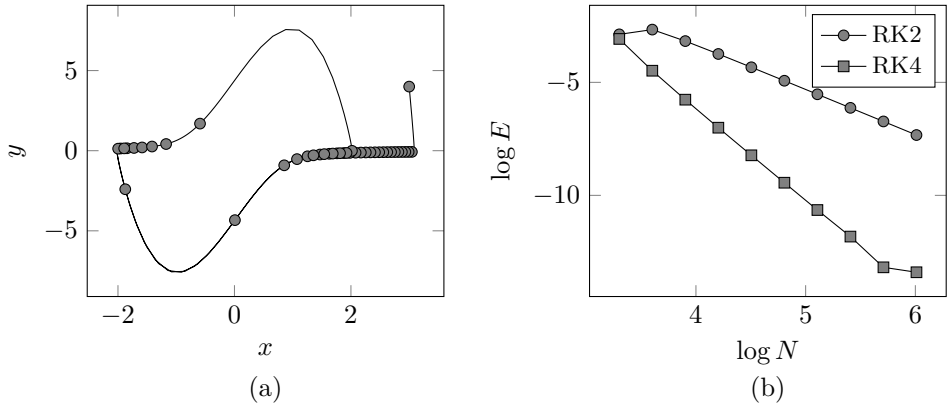

Figure 4.6: Convergence rate of a second and fourth order Runge–Kutta schemes with time step. (a) Van der Pol solution. (b) Error versus time steps.

In the figure 4.6a the Van der Pol solution is shown. In figure 4.6b the errors versus the number of time steps N are plotted in logarithmic scale. It can be observed that the fourth-order Runge-Kutta has an approximate slope of 4, whereas the slope of the second-order Runge–Kutta scheme is close to two.

4.8 Advanced high order numerical methods

When high precision requirements are necessary, high order temporal schemes must be used. This is the case of orbits or satellite missions. These simulations require very small errors during their temporal integration. Generally, it is said that a numerical method is of order q when its global error is $O(\Delta t^q)$. This means that high order numerical methods require greater time steps than low order schemes to accomplish the same error. Consequently, high order methods have lower computational effort than low order methods. The following subroutine shows the performance of some advanced high order methods when simulating orbit problems:

```
subroutine Advanced_Cauchy_problem_examples
  integer :: option = 1

  do while (option>0)

     write(*,*) "Advanced methods"
     write(*,*) " select an option "
     write(*,*) " 0. Exit/quit  "
     write(*,*) " 1. Van del Pol system "
     write(*,*) " 2. Henon Heiles system "
     write(*,*) " 3. Variable time step versus constant time step "
     write(*,*) " 4. Convergence rate of Runge Kutta wrappers  "
     write(*,*) " 5. Arenstorf orbit (embedded Runke-Kutta) "
     write(*,*) " 6. Arenstorf orbit (GBS methods, Wrapper ODEX)"
     write(*,*) " 7. Arenstorf orbit (ABM methods, Wrapper ODE113)"
     write(*,*) " 8. Computational effort Runge-Kutta methods"
     read(*,*) option

     select case(option)
      case(1)
             call Van_der_Pol_oscillator
      case(2)
             call Henon_Heiles_system
      case(3)
             call Variable_step_simulation
      case(4)
             call Convergence_rate_Runge_Kutta_wrappers
      case(5)
             call Runge_Kutta_wrappers_versus_original_codes
      case(6)
             call GBS_and_wrapper_ODEX
      case(7)
             call ABM_and_wrapper_ODE113
      case(8)
             call Temporal_effort_with_tolerance_eRK
      case default
     end select
  end do
end subroutine
```

Listing 4.11: `API_Example_Cauchy_Problem.f90`

4.9 Van der Pol oscillator

The van der Pol oscillator is a non-conservative stable oscillator which is applied to physical and biological sciences. Its second order differential equation is:

$$\ddot{x} - \mu\ (1 - x^2)\dot{x} + x = 0.$$

This equation can be expressed as the following first order system:

$$\begin{aligned} \dot{x} &= v, \\ \dot{v} &= -x + \mu\left(1 - x^2\right) v. \end{aligned}$$

To implement this problem, the the differential operator $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ is created.

```
function VanDerPol_equation( U, t ) result(F)
                    real :: U(:), t
                    real :: F(size(U))

    real :: mu = 5., x, v

    x = U(1); v = U(2)
    F = [ v, mu * (1 - x**2) * v  - x ]

end function
```

Listing 4.12: `API_Example_Cauchy_Problem.f90`

Again, the function is used as an input argument for the subroutine that computes the solution of the Cauchy Problem. In this case, advanced temporal methods for Cauchy problems are used, particularly embedded Runge-Kutta formulas. The methods used are `"RK87"` and `"Fehlberg87"` and require the use of an error tolerance, which is set as $\epsilon = 10^{-8}$. Both of them are selected by the subroutine `set_solver`.

Each method is given a different initial condition in order to illustrate the long time behavior of the solution. The asymptotic behavior of the solution tends to a limit cycle, that is, given sufficient time the solution becomes periodic. This can be observed from figure 4.7(a) where the solution is obtained with the embedded Runge Kutta scheme `"RK87"` and from figure 4.7(b) integrated with the `"Fehlberg87"` scheme. Although both solutions tend to the same cycle, a difference in their phases can be observed in figure 4.7(b).

```
subroutine Van_der_Pol_oscillator

    real :: t0 = 0, tf = 30
    integer, parameter :: N = 350, Nv = 2
    real :: Time (0:N), U(0:N, Nv, 2)
    integer :: i

    Time = [ (t0 + (tf -t0 ) * i / (1d0 * N), i=0, N ) ]

    U(0,:,1) =  [3, 4]
    call set_solver("eRK", "RK87")
    call set_tolerance(1d-8)
    call Cauchy_ProblemS( Time, VanDerPol_equation, U(:,:,1) )

    U(0,:,2) =  [0, 1]
    call set_solver("eRK", "Fehlberg87")
    call set_tolerance(1d-8)
    call Cauchy_ProblemS( Time, VanDerPol_equation, U(:,:,2) )

    write (*, *) "VanDerPol oscillator with RK87 and Fehlberg87 "
    write(*,*) "press enter "; read(*,*)
    call plot_parametrics(time,U(:,1, 1:2),["RK87","Fehlberg87"],"t","x")

end subroutine
```

Listing 4.13: `API_Example_Cauchy_Problem.f90`

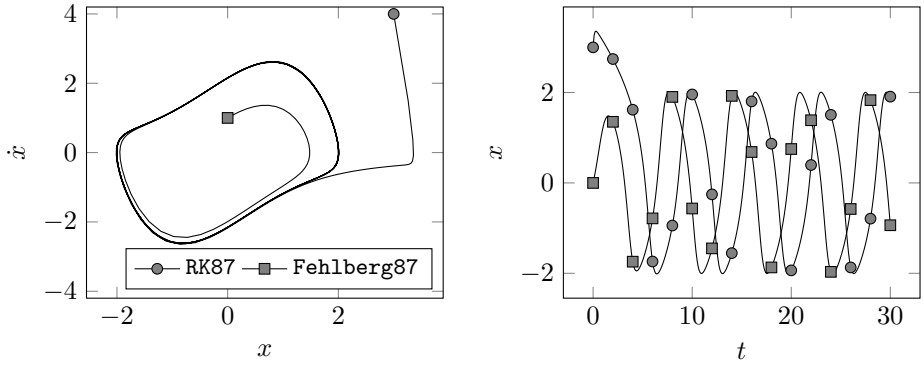

Figure 4.7: Solution of the Van der Pol oscillator. (a) Trajectory on the phase plane $(x, \dot{x})$. (b) Evolution along time of x.

4.10 Henon-Heiles system

The non-linear motion of a star around a galactic center, with the motion restricted to a plane, can be modeled through Henon-Heiles system:

$$\begin{aligned}\dot{x} &= p_x \\ \dot{y} &= p_y \\ \dot{p}_x &= -x - 2\lambda xy \\ \dot{p}_y &= -y - \lambda\left(x^2 - y^2\right).\end{aligned}$$

As usual, the differential operator is implemented as a function `Henon_equation` that is used as an input argument by the subroutine `Cauchy\_ProblemS`. The GBS temporal scheme is selected by the calling `set_solver` and its tolerance is set by `set_tolerance`.

```
function Henon_equation( U, t )  result(F)
                real :: U(:), t
                real :: F(size(U))

    real :: x, y, px, py, lambda = -1

    x = U(1) ; y = U(2); px = U(3); py = U(4)

    F = [ px, py, -x-2* lambda*x*y, -y-lambda*(x**2 - y**2) ]

end function
```

Listing 4.14: `API_Example_Cauchy_Problem.f90`

```
subroutine Henon_Heiles_system
    integer, parameter :: N = 1000, Nv = 4 , M = 1 !Time steps
    real, parameter :: dt = 0.1
    real :: t0 = 0, tf = dt * N
    real :: Time (0:N), U(0:N, Nv), H(0:N)
    integer :: i

    Time = [ (t0 + (tf -t0 ) * i / (1d0 * N), i=0, N ) ]

    U(0,:) = [0., 0., 0.6,0.]
    call set_solver("GBS")
    call set_tolerance(1d-2)
    call Cauchy_ProblemS( Time, Henon_equation, U )

    write (*, *) 'Henon Heiles system  '
    write(*,*) "press enter " ; read(*,*)
    call plot_parametrics(U(:,1),U(:,2:2), ["Henon Heiles"],"x","y")


end subroutine
```

Listing 4.15: `API_Example_Cauchy_Problem.f90`

Once the code is compiled and executed, the trajectories in the phase plane are shown in figure 4.8.

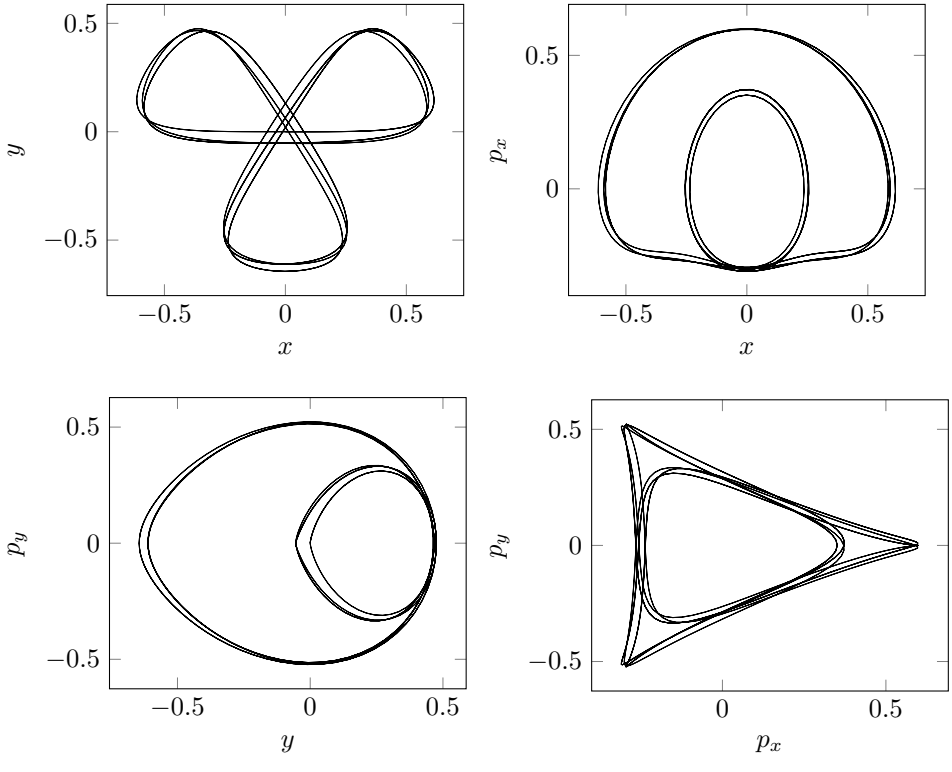

Figure 4.8: Heinon-Heiles system solution. (a) Trajectory of the star (x, y). (b) Projection $(x, \dot{x})$ of the solution in the phase plane. (c) Projection $(y, \dot{y})$ of the solution in the phase plane. (d) Projection $(\dot{x}, \dot{y})$ of the solution in the phase plane.

This simple Hamiltonian system can exhibit chaotic behavior for certain values of the initial conditions which represent different values of energy. For example, the initial conditions

$$(x(0), y(0), p_x(0), p_y(0)) = (0.5, 0.5, 0, 0),$$

give rise to chaotic behavior.

4.11 Constant time step and adaptive time step

Generally, time-dependent problems evolve with different growth rates during its time-span. This behavior motivates to use of variable time steps to march faster when small gradients are encountered and march slower reducing the time step when high gradients appear. To adapt automatically the time step, methods must estimate the error to reduce or to increase the time step to reach a specified tolerance.

In the following code, the Van der Pol problem is solved with a variable time step in an embedded Heun-Euler method. Since the imposed tolerance is set to 10^{10}, the embedded Heun–Euler method will not modify the time step because that tolerance is always reached.

The other simulation is carried out with a tolerance of 10^{-6}. In this case, the embedded Heun-Euler will adapt the time step to reach this specific tolerance.

```
call set_solver(family_name="eRK", scheme_name="HeunEuler21")
call set_tolerance(1e10)
call Cauchy_ProblemS( Time, VanDerPol_equation, U(:,:,1) )

call set_solver(family_name="eRK", scheme_name="HeunEuler21")
call set_tolerance(1e-6)
call Cauchy_ProblemS(Time, VanDerPol_equation, U(:,:,2) )
```

Listing 4.16: `API_Example_Cauchy_Problem.f90`

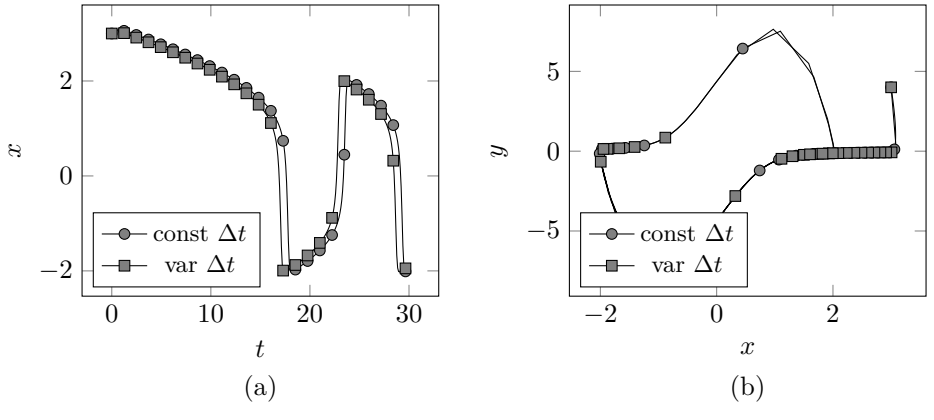

Figure 4.9: Comparison between constant and variable time step calculated by means of local estimation error. Integration of the Van der Pol oscillator with an embedded second order Runge–kutta HeunEuler21. (a) x position along time. (b) Phase diagram of the solutions.

4.12 Convergence rate of Runge–Kutta wrappers

A wrapper function is a subroutine whose main purpose is to call a second subroutine with little or no additional computation. Generally, wrapper functions are used to make writing computer programs easier to use by abstracting away the details of an old underlying implementation. In this way, old validated codes written in Fortran 77 can be used with a modern interface encapsulating the implementation details and making friendly interfaces.

In the following code, the classical `DOPRI5` and `DOP853` embedded Runge Kutta methods are used by means of a module that wraps the old codes.

```
call set_solver(family_name="weRK",scheme_name="WDOPRI5")
call set_tolerance(1e6)
call Temporal_convergence_rate(                                          &
     Time_domain = Time, Differential_operator = VanDerPol_equation, &
     U0 = U0,  order = 5, log_E = log_E(:,1), log_N = log_N )

call set_solver(family_name="weRK",scheme_name="WDOP853")
call set_tolerance(1e6)
call Temporal_convergence_rate(                                          &
     Time_domain = Time, Differential_operator = VanDerPol_equation, &
     U0 = U0,  order = 8, log_E = log_E(:,2), log_N = log_N )
```

Listing 4.17: `API_Example_Cauchy_Problem.f90`

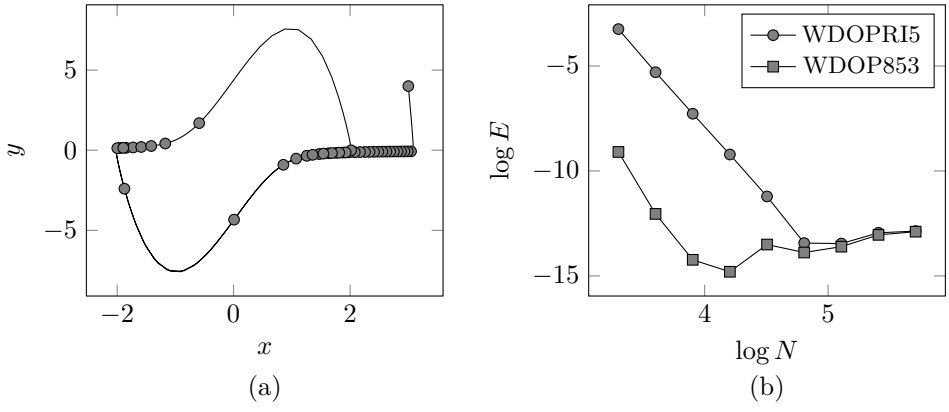

Figure 4.10: Convergence rate of Runge–Kutta wrappers based on DOPRI5 and DOP853 with number of steps. (a) Van der Pol solution. (b) Error versus time steps.

In figure 4.10b, the steeper slope of `DOP853` in comparison with the slope of `DOPRI5` shows its superiority in terms of its temporal error.

4.13 Arenstorf orbit. Embedded Runge–Kutta

The Arenstorf orbit is a stable periodic orbit between the Earth and the Moon which was used as the basis for the Apollo missions. They are closed trajectories of the restricted three-body problem, where two bodies of masses μ and $1-\mu$ are moving in a circular rotation, and the third body of negligible mass is moving in the same plane. The equations that govern the movement of the third body in axis rotating about the center of gravity of the Earth and the Moon are:

$$
\begin{aligned}
\dot{x} &= v_x, \\
\dot{y} &= v_y, \\
\dot{v_x} &= x + 2v_y - \frac{(1-\mu)(x+\mu)}{\sqrt{\left((x+\mu)^2+y^2\right)^3}} - \frac{\mu(x-(1-\mu))}{\sqrt{\left((x-(1-\mu))^2+y^2\right)^3}} \\
\dot{v_y} &= y - 2v_x - \frac{(1-\mu)y}{\sqrt{\left((x+\mu)^2+y^2\right)^3}} - \frac{\mu y}{\sqrt{\left((x-(1-\mu))^2+y^2\right)^3}}
\end{aligned}
$$

```
function Arenstorf_equations(U, t) result(F)
     real :: U(:),t
     real :: F(size(U))
     real :: mu = 0.012277471

     real :: x, y , vx, vy, dxdt, dydt, dvxdt, dvydt
     real :: D1, D2

     x = U(1); y = U(2); vx = U(3); vy = U(4)

     D1 = sqrt( (x+mu)**2     + y**2 )**3
     D2 = sqrt( (x-(1-mu))**2 + y**2 )**3

     dxdt = vx
     dydt = vy
     dvxdt = x + 2 * vy - (1-mu)*( x + mu )/D1 - mu*(x-(1-mu))/D2
     dvydt = y - 2 * vx - (1-mu) * y/D1 - mu * y/D2

     F = [ dxdt, dydt, dvxdt, dvydt ]

end function
```

Listing 4.18: `API_Example_Cauchy_Problem.f90`

The following code integrates the Arenstorf orbit by means of the classical wrapped `DOPRI54` and a new implementation written in modern Fortran. Different tolerances are selected to show the influence on the calculated orbit.

```
      U(0,:,j) = [0.994, 0., 0., -2.0015851063790825 ]
    end do
    Time = [ (t0 + (tf -t0 ) * i / real(N), i=0, N ) ]

    call set_solver(family_name="weRK", scheme_name="WDOPRI5")
    do j=1, Np
      call set_tolerance(tolerances(j))
      call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
    end do
    call plot_parametrics( U(:, 1, :), U(:, 2, :), names,           &
                           "$x$", "$y$", "(a)", path(1) )

    call set_solver(family_name="eRK", scheme_name="DOPRI54")
    do j=1, Np
      call set_tolerance(tolerances(j))
      call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
    end do
    call plot_parametrics( U(:, 1, :), U(:, 2, :) , names,          &
                           "$x$", "$y$", "(b)", path(2) )

end subroutine
```

Listing 4.19: `API_Example_Cauchy_Problem.f90`

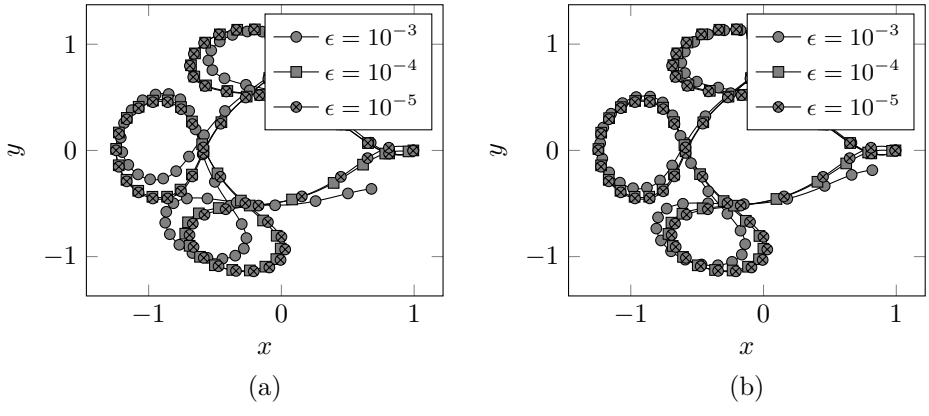

Figure 4.11: Integration of the Arenstorf orbit by means of embedded Runge–Kutta methods with a specific tolerance ϵ. (a) Wrapper of the embedded Runge-Kutta WDOPRI5. (b) New implementation of the embedded Runge-Kutta DOPRI54.

As expected, the wrapped code and the new implementation show similar results. When the tolerance error is decreased, the calculated orbit approaches to a closed trajectory.

4.14 Gragg-Bulirsch-Stoer Method

The Gragg-Bulirsch-Stoer Method is also a common high order method for solving ordinary equations. This method combines the Richardson extrapolation and the modified midpoint method. For this example, the new implementation of the GBS algorithm and the old wrapped `ODEX` have been used to simulate the Arenstorf orbit.

```
      call set_solver(family_name="wGBS")
      do j=1, Np
        call set_tolerance(tolerances(j))
        call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
      end do
      call plot_parametrics( U(:, 1, :), U(:, 2, :), names,            &
                             "$x$", "$y$", "(a)", path(1) )

      call set_solver(family_name="GBS")
      do j=1, Np
        call set_tolerance(tolerances(j))
        call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
      end do
      call plot_parametrics( U(:, 1, :), U(:, 2, :), names,            &
                             "$x$", "$y$", "(b)", path(2) )
end subroutine
```

Listing 4.20: `API_Example_Cauchy_Problem.f90`

Figure 4.12 show that GBS method is much less sensitive to the set tolerance and reach a trajectory closer to the solution than the eRK methods analyzed in the previous section.

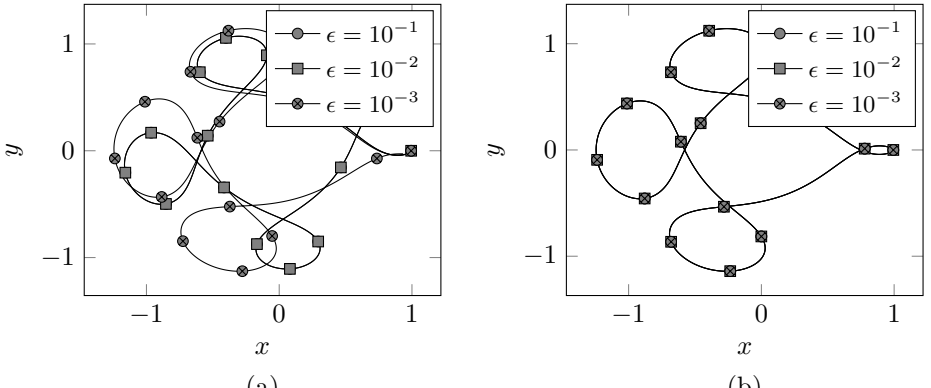

Figure 4.12: Integration of the Arenstorf orbit by means of the Gragg-Bulirsch-Stoer Method with a specific tolerance ϵ. (a) Wrapper of GMS method ODEX. (b) New implementation of the GBS method.

4.15 Adams-Bashforth-Moulton Methods

Adams–Bashforth–Moulton schemes are multi-step methods that require only two evaluations of the function of the Cauchy problem per time step. The local error estimation is based on a predictor-corrector scheme. The predictor is implemented as an Adams–Bashforth method and the corrector is an Adams–Moulton method. In the following code, the classical `ODE113` (wrapped by `wABM`) is used in comparison with the new implementation `ABM`.

```
     call set_solver(family_name="wABM")
     do j=1, Np
       call set_tolerance(tolerances(j))
       call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
     end do
     call plot_parametrics( U(:, 1, :), U(:, 2, :) , names,         &
                            "$x$", "$y$", "(a)", path(1) )

     call set_solver(family_name="ABM")
     do j=1, Np
       call set_tolerance(tolerances(j))
       call Cauchy_ProblemS( Time, Arenstorf_equations, U(:, :, j) )
     end do
     call plot_parametrics( U(:, 1, :), U(:, 2, :) , names,        &
                              "$x$", "$y$", "(b)", path(2) )

end subroutine
```

Listing 4.21: `API_Example_Cauchy_Problem.f90`

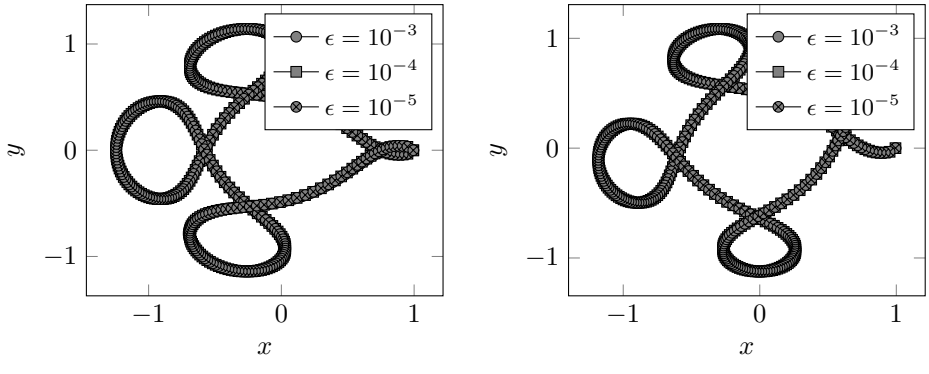

Figure 4.13: Integration of the Arenstorf orbit by means of the Adams-Bashforth-Moulton Methods with a specific tolerance ϵ. (a) Wrapper of ABM method ODE113. (b) New implementation of the ABM methods as a multi-value method.

4.16 Computational effort of Runge–Kutta schemes

When high order precision is required, it is important to select the best temporal scheme. The best scheme is the one that reaches the lowest error tolerance with the smallest CPU time. In the following code, a new subroutine called `Temporal_effort_with_tolerance` is used to measure the computational effort. Once the temporal scheme is selected, this subroutine runs the Cauchy problem with different error tolerance based on the input argument `log_mu`. It measures internally the number of evaluations of the function of the Cauchy problem for every simulation. In this way, the number of evaluations of the function of the Cauchy problem can be represented versus the error for different time schemes.

```
    log_mu = [( i, i=1, M ) ]

    do j=1, Np
      call set_solver(family_name = "eRK", scheme_name = names(j) )
      call Temporal_effort_with_tolerance( Time, VanDerPol_equation, U0,&
                                           log_mu, log_effort(:,j)  )
     end do

     call plot_parametrics( log_mu, log_effort(:,1:3), names(1:3),      &
                           "$-\log \epsilon$", "$\log M$ ", "(a)", path(1))
     call plot_parametrics( log_mu, log_effort(:,4:7), names(4:7),      &
                           "$-\log \epsilon$", "$\log M$ ", "(b)", path(2))
end subroutine
```

Listing 4.22: `API_Example_Cauchy_Problem.f90`

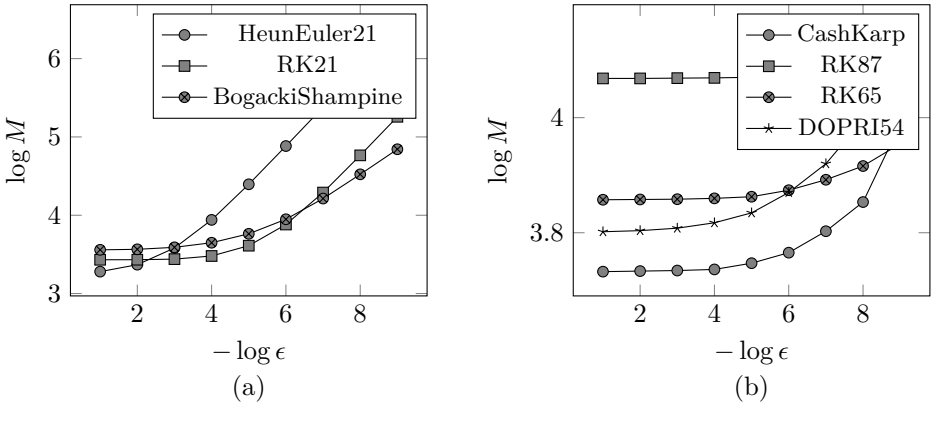

Figure 4.14: Computational effort of embedded Runge–Kutta. Number of time steps M as a function of the specified tolerance ϵ for the different member of the embedded Runge–Kutta family. (a) Embedded Runge-Kutta of second and third order. (b) Embedded Runge-Kutta from fourth to seventh order.

Chapter 5

Boundary Value Problems

5.1 Overview

Let $\Omega \subset \mathbb{R}^p$ be an open and connected set and $\partial\Omega$ its boundary set. The spatial domain D is defined as its closure, $D \equiv \{\Omega \cup \partial\Omega\}$. Each point of the spatial domain is written $\boldsymbol{x} \in D$. A Boundary Value Problem for a vectorial function $\boldsymbol{u} : D \to \mathbb{R}^N$ of N variables is defined as:

$$\begin{aligned} &\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x})) = 0, && \forall \ \ \boldsymbol{x} \in \Omega, \\ &\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))\big|_{\partial\Omega} = 0, && \forall \ \ \boldsymbol{x} \in \partial\Omega, \end{aligned}$$

where $\boldsymbol{\mathcal{L}}$ is the spatial differential operator and $\boldsymbol{h}$ is the boundary conditions operator that must satisfy the solution at the boundary $\partial\Omega$.

In the subroutine `BVP_examples`, examples of boundary value problems are presented. The first example is the classical Legendre equation for a 1D space. The second solves a Poisson problem in a 2D. The third studies the deflection of an elastic plate subjected to external loads in a 2D space. Finally, the fourth example analyzes the deflection on an nonlinear plate subjected to external loads.

```
subroutine BVP_examples
   call Legendre_1D
   call Poisson_2D
   call Elastic_Plate_2D
   call Elastic_Nonlinear_Plate_2D
end subroutine
```

Listing 5.1: `API_example_Boundary_Value_Problem.f90`

To use all functions of this module, the statement: `use Boundary_value_problems` should be included at the beginning of the program.

5.2 Legendre equation

Legendre polynomials are a system of complete and orthogonal polynomials with numerous applications. Legendre polynomials can be defined as the solutions of the Legendre's differential equation on a domain $\Omega \subset \mathbb{R} : \{x \in [-1, 1]\}$:

$$(1 - x^2)\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} - 2x\frac{\mathrm{d}y}{\mathrm{d}x} + n(n + 1)y = 0,$$

where n stands for the degree of the Legendre polynomial. For $n = 6$, the boundary conditions are: $y(-1) = -1,\ y(1) = 1$ and the exact solution is:

$$y(x) = \frac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5).$$

This problem is solved by means of piecewise polynomial interpolation of degree q or finite differences of order q. The implementation of the problem requires the definition of the differential operator $\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))$:

```
real function Legendre(x, y, yx, yxx) result(L)
   real, intent(in) :: x, y, yx, yxx

    integer :: n = 6

    L = (1 - x**2) * yxx - 2 * x * yx + n * (n + 1) * y

end function
```

Listing 5.2: `API_example_Boundary_Value_Problem.f90`

And the boundary conditions $\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))$ are implemented as a function:

```
real function Legendre_BCs(x, y, yx) result(BCs)
           real, intent(in) :: x, y, yx

        if (x==x0 .or. x==xf ) then
                          BCs = y - 1
        else
            write(*,*) " Error BCs x=", x; stop
        endif

end function
```

Listing 5.3: `API_example_Boundary_Value_Problem.f90`

These two functions are input arguments of the subroutine `Boundary_Value_Problem`:

```
!   Legendre solution
    call Boundary_Value_Problem( x_nodes = x,                          &
                                 Differential_operator = Legendre,     &
                                 Boundary_conditions   = Legendre_BCs, &
                                 Solution = U(:,1) )

    Error(:,1) = U(:,1) - ( 231 * x**6 - 315 * x**4 + 105 * x**2 - 5 )/16.
```

Listing 5.4: `API_example_Boundary_Value_Problem.f90`

In this example, the piecewise polynomial interpolation is of degree $q = 6$ and the problem is discretized with $N = 20$ grid nodes.

```
    integer, parameter :: N = 20, q = 6
```

Listing 5.5: `API_example_Boundary_Value_Problem.f90`

Since the degree of the piecewise polynomial interpolation coincides with the degree of the solution or the Legendre polynomial, no error is expected to obtain. It can be observed in figure 5.1b that error is of the order of the round–off value. The solution or the Legendre polynomial is shown in 5.1a.

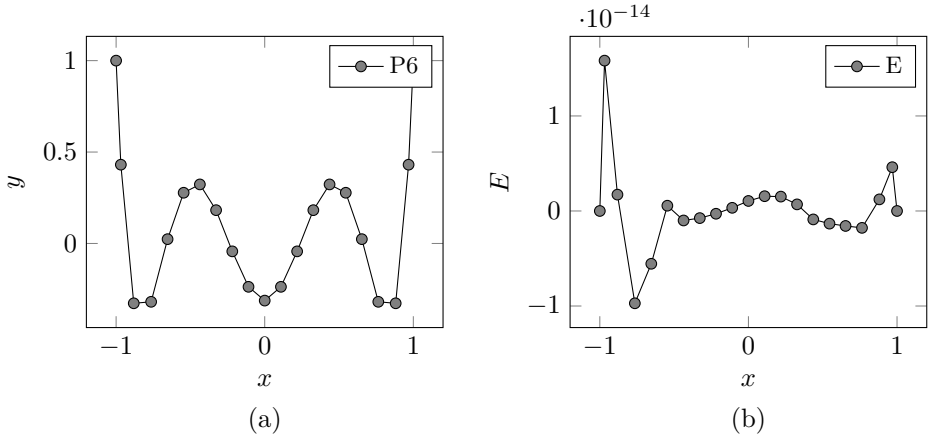

Figure 5.1: Solution of the Legendre equation with $N = 20$ grid points. (a) Legrendre polynomial of degree $n = 6$. (b) Error of the solution.

5.3 Poisson equation

Poisson's equation is a partial differential equation of elliptic type with broad utility in mechanical engineering and theoretical physics. This equation arises to describe the potential field caused by a given charge or mass density distribution. In the case of fluid mechanics, it is used to determine potential flows, streamlines and pressure distributions for incompressible flows. It is an in-homogeneous differential equation with a source term representing the volume charge density, the mass density or the vorticity function in the case of a fluid. It is written in the following form:

$$\nabla^2 u = s(x, y),$$

where $\nabla^2 u = \partial^2 u/\partial x^2 + \partial^2 u/\partial y^2$ and $s(x, y)$ is the source term. This Poisson equation is implemented by the following code:

```
real function Poisson(x, y, u, ux, uy, uxx, uyy, uxy) result(L)
  real, intent(in) :: x, y, u, ux, uy, uxx, uyy, uxy

         L =  uxx + uyy - source(x,y)

end function
```

Listing 5.6: API_example_Boundary_Value_Problem.f90

It is considered two punctual sources given by the expression:

$$S(x, y) = ae^{-ar_1^2} + ae^{-ar_2^2}, \quad r_i^2 = (x - x_i)^2 + (y - y_i)^2,$$

where a is an attenuation parameter and (x_1, y_1) and (x_2, y_2) are the positions of the sources. The source term is implemented in the following code:

```
real function source(x, y)
    real, intent(in) :: x, y

        real :: r1, r2, a=100

         r1 = norm2( [x, y] - [ 0.2, 0.5] )
         r2 = norm2( [x, y] - [ 0.8, 0.5] )

         source   = a * exp(-a*r1**2) + a * exp(-a*r2**2)

end function
```

Listing 5.7: API_example_Boundary_Value_Problem.f90

In this example, homogeneous boundary conditions are considered and they implemented by the function `PBCs`:

```
real function PBCs(x, y, u, ux, uy) result(BCs)
real, intent(in) :: x, y, u, ux, uy

        if ( x==a .or. x==b .or. y==a .or. y==b ) then
                                                  BCs = u
        else
            write(*,*) " Error BCs x=", x;stop
        endif

end function
```

Listing 5.8: `API_example_Boundary_Value_Problem.f90`

The differential operator `Poisson` with its boundary conditions `PBCs` are used as input arguments for the subroutine `Boundary_value_problem`

```
!   Poisson equation
    call Boundary_Value_Problem( x_nodes = x, y_nodes = y,                &
                                 Differential_operator = Poisson,         &
                                 Boundary_conditions = PBCs, Solution = U)
```

Listing 5.9: `API_example_Boundary_Value_Problem.f90`

In figure 5.2 the solution of this Poisson equation is shown.

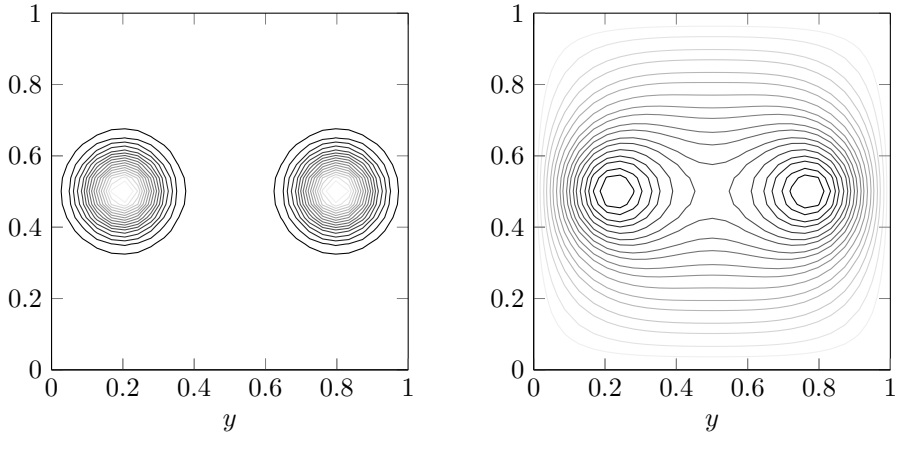

Figure 5.2: Solution of the Poisson equation with $Nx = 30$, $Ny = 30$ and piecewise interpolation of degree $q = 11$. (a) Source term s(x,y).(b) Solution u(x,y).

5.4 Deflection of an elastic linear plate

In this section, an elastic plate submitted to a distributed load is implemented. It is Considered a plate with simply supported edges with a distributed load $p(x, y)$ in a domain $\Omega \subset \mathbb{R}^2$:$\{(x, y) \in [-1, 1] \times [-1, 1]\}$.The deflection $w(x, y)$ of a the plate is governed by the following bi-harmonic equation:

$$\nabla^4 w(x, y) = p(x, y),$$

where $\nabla^4 = \nabla^2(\nabla^2)$ is the bi-harmonic operator. The simply supported condition is set by imposed that the displacement is zero and bending moments are zero at the boundaries. It can be proven that the zero bending moment condition is equivalent that the Laplacian of the displacement is zero $\nabla^2 w = 0$.

Since the module `Boundary_value_problems` only takes into account second-order derivatives, the problem must be transformed into the second-order problem by means of the transformation:

$$\boldsymbol{u}(x, y) = [\ w(x, y),\ v(x, y)\ ],$$

which leads to the system:

$$\nabla^2 w = v,$$
$$\nabla^2 v = p(x, y).$$

The above equations are implemented in the function `Elastic_Plate`

```
function Elastic_Plate(x, y, u, ux, uy, uxx, uyy, uxy) result(L)
   real, intent(in) :: x, y, u(:), ux(:), uy(:), uxx(:), uyy(:), uxy(:)
   real :: L(size(u))

   real :: w, wxx, wyy, v, vxx, vyy

        w = u(1); wxx = uxx(1); wyy = uyy(1)
        v = u(2); vxx = uxx(2); vyy = uyy(2)

        L(1) =  wxx + wyy - v
        L(2) =  vxx + vyy - load(x,y)

end function
```

Listing 5.10: `API_example_Boundary_Value_Problem.f90`

In this example, it is considered a vertical plate in y direction submitted to ambient pressure to one side and a hydro-static pressure to the other side. Besides,

the fluid at $y = 0$ has ambient pressure. With these considerations, the plate is submitted to the following non-dimensional net force:

$$p(x, y) = ay,$$

where a is a non-dimensional parameter. This external load is implemented in the function `load`

```
real  function load(x, y)
   real, intent(in) :: x, y

   load =  100*y

end function
```

Listing 5.11: `API_example_Boundary_Value_Problem.f90`

The boundary conditions are:

$$w\big|_{\partial\Omega} = 0,$$
$$\nabla^2 w\big|_{\partial\Omega} = 0.$$

Since $v = \nabla^2 w$, these conditions leads to $w = 0, v = 0$ at the boundaries and they are implemented in the following function `Plate_BCs`:

```
function Plate_BCs(x, y, u, ux, uy) result(BCs)
  real, intent(in) :: x, y, u(:), ux(:), uy(:)
  real :: BCs(size(u))

        real :: w, v

        w = u(1)
        v = u(2)

        if (x==x0 .or. x==xf .or. y==y0 .or. y==yf ) then

            BCs(1) = w
            BCs(2) = v

        else
            write(*,*) " Error BCs x=", x; stop
        endif

end function
```

Listing 5.12: `API_example_Boundary_Value_Problem.f90`

In this example, piecewise polynomial interpolation of degree $q = 4$ is chosen. The non-uniform grid points are selected by the subroutine `Grid_initialization` by imposing constant truncation error.

The differential operator `Elastic_Plate` and its boundary conditions `Plate_PBCs` are used as input arguments for the subroutine `Boundary_value_problem`.

```
!   Elastic linear plate
    call Grid_Initialization( "nonuniform", "x", q, x )
    call Grid_Initialization( "nonuniform", "y", q, y )

    call Boundary_Value_Problem( x_nodes = x, y_nodes = y,             &
                                 Differential_operator = Elastic_Plate,&
                                 Boundary_conditions = Plate_BCs,      &
                                 Solution = U )
```

Listing 5.13: `API_example_Boundary_Value_Problem.f90`

In figure 5.3a, the external load is shown. As it was mentioned, the net force between the hydro-static pressure and the ambient pressure takes zero value at the vertical position $y = 0$. For values $y > 0$, the external load is positive and for values $y < 0$ the load is negative. This external load divides the plate vertically into two parts. A depressed lower part and a bulged upper part is shown in figure 5.3b.

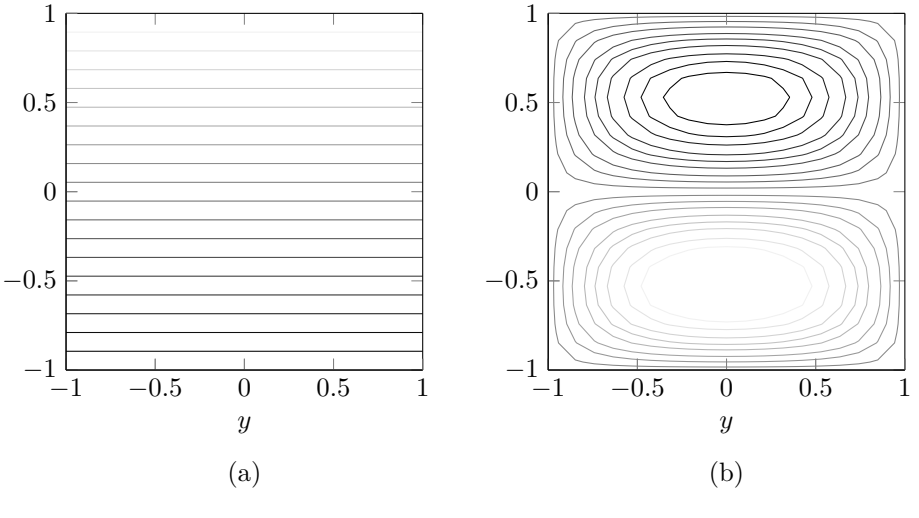

Figure 5.3: Linear plate solution with 21×21 nodal points and $q = 4$. (a) External load $p(x, y)$. (b) Displacement $w(x, y)$.

5.5 Deflection of an elastic non linear plate

A more complex example of a 2D boundary value problem is shown in this section. A nonlinear elastic plate submitted to a distributed load simply supported on its four edges is considered. The deflection w of the nonlinear plate is ruled by the bi-harmonic equation plus a non linear term which depends on the stress Airy function ϕ. As in the section before, the simply supported edges are considered by imposing zero displacement and zero Laplacian of the displacement. The problem in a domain $\Omega \subset \mathbb{R}^2 : \{(x, y) \in [-1, 1] \times [-1, 1]\}$ is formulated as:

$$\begin{aligned}\nabla^4 w &= p(x, y) + \mu\ \mathcal{L}(w, \phi),\\ \nabla^4 \phi &+ \mathcal{L}(w, w) = 0,\end{aligned}$$

where μ is a non-dimensional parameter and $\mathcal{L}$ is the bi-linear operator:

$$\mathcal{L}(w, \phi) = \frac{\partial^2 w}{\partial x^2}\frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 w}{\partial y^2}\frac{\partial^2 \phi}{\partial x^2} - 2\frac{\partial^2 w}{\partial x \partial y}\frac{\partial^2 \phi}{\partial x \partial y}.$$

Since the module `Boundary_value_problems` only takes into account second-order derivatives, the problem must be transformed into the second-order problem by means of the transformation:

$$\boldsymbol{u}(x, y) = [\ w,\ v,\ \phi,\ F],$$

which leads to the system:

$$\begin{aligned}\nabla^2 w &= v,\\ \nabla^2 v &= p(x, y) + \mu\ \mathcal{L}(w, \phi),\\ \nabla^2 \phi &= F,\\ \nabla^2 F &= -\ \mathcal{L}(w, w).\end{aligned}$$

The external load applied to the nonlinear plate is the same that it was used in the deflections of the linear plate

$$p(x, y) = ay.$$

It allows comparing a linear solution with a nonlinear solution. The plate behaves non-linearly when deflections are of the order of the plate thickness. Since the deflections are caused by the external load, the non-dimensional parameter a can be used to take the plate to a nonlinear regime.

The nonlinear plate equations expressed as a second order derivatives system of equations is implemented in the following function `NL_Plate`:

```
function NL_Plate(x, y, u, ux, uy, uxx, uyy, uxy) result(L)
  real, intent(in) :: x, y, u(:), ux(:), uy(:), uxx(:), uyy(:), uxy(:)
  real :: L(size(u))

  real ::   w, wxx, wyy, wxy, v, vxx, vyy
  real :: phi, phixx, phiyy, phixy, F, Fxx, Fyy


  w   = u(1);   wxx = uxx(1);   wyy = uyy(1) ;   wxy = uxy(1)
  v   = u(2);   vxx = uxx(2);   vyy = uyy(2)
  phi = u(3); phixx = uxx(3); phiyy = uyy(3) ; phixy = uxy(3)
  F   = u(4);   Fxx = uxx(4);   Fyy = uyy(4)

  L(1) = wxx + wyy - v
  L(2) = vxx + vyy - load(x,y)                                &
         - mu * Lb(wxx, wyy, wxy, phixx, phiyy, phixy)
  L(3) = phixx + phiyy - F
  L(4) = Fxx + Fyy  +  Lb(wxx, wyy, wxy, wxx, wyy, wxy)

end function
```

Listing 5.14: API_example_Boundary_Value_Problem.f90

In this function the bi-linear operator $\mathcal{L}$ is implemented in the function Lb

```
real function Lb( wxx, wyy, wxy, pxx, pyy, pxy)
  real, intent(in) ::  wxx, wyy, wxy, pxx, pyy, pxy

  Lb = wxx * pyy + wyy * pxx - 2 * wxy * pxy

end function
```

Listing 5.15: API_example_Boundary_Value_Problem.f90

The boundary conditions are implemented in the function NL_Plate_BCs

```
function NL_Plate_BCs(x, y, u, ux, uy) result(BCs)
  real, intent(in) :: x, y, u(:), ux(:), uy(:)
  real :: BCs(size(u))

       if (x==x0 .or. x==xf .or. y==y0 .or. y==yf) then
                          BCs = u
       else
           write(*,*) " Error BCs x, y=", x, y; stop
       endif

    end function
```

Listing 5.16: API_example_Boundary_Value_Problem.f90

The differential operator `NL_Plate` and its boundary conditions `NL_Plate_BCs` are used as input arguments for the subroutine `Boundary_value_problem`.

```
!   Elastic nonlinear plate
    call Boundary_Value_Problem( x_nodes = x, y_nodes = y,             &
                                 Differential_operator = NL_Plate,     &
                                 Boundary_conditions   = NL_Plate_BCs, &
                                 Solution = U )
```

Listing 5.17: `API_example_Boundary_Value_Problem.f90`

In figure 5.4, the solution of the nonlinear plate model is shown.

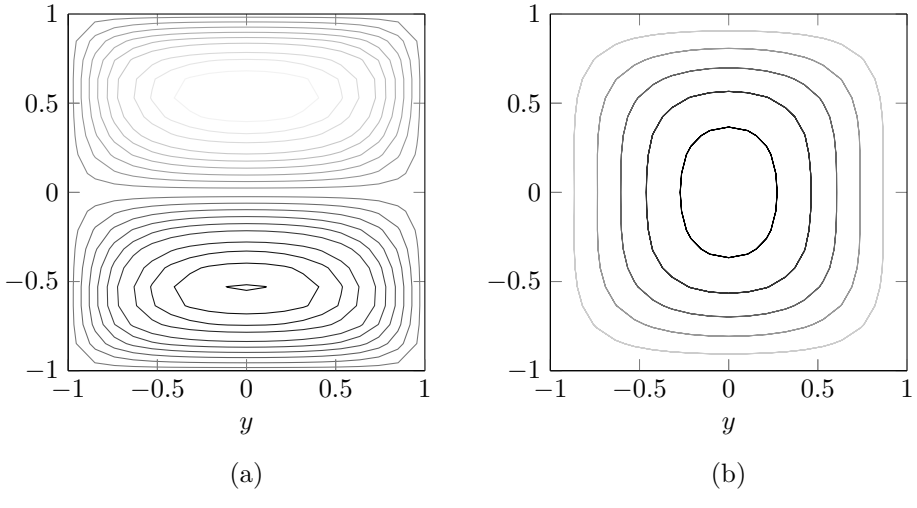

Figure 5.4: Non linear elastic plate solution with $N_x = 20$, $N_y = 20$, $q = 4$ and $\mu = 100$. (a) Displacement $w(x, y)$. (b) Solution $\phi(x, y)$

Chapter 6

Initial Boundary Value Problems

6.1 Overview

In this chapter, several Initial Boundary Value problems will be presented. These problems can be divided into those purely diffusive are the heat equation and those purely convective as the wave equation. In between, there are convective and diffusive problems that are represented by the convective-diffusive equation. In the subroutine `IBVP_examples`, six examples of these problems are implemented. The first problem is to obtain the solution of the one-dimensional heat equation. The second problem presents a two-dimensional solution of the heat equation with non-homogeneous boundary conditions. The third problem and fourth problem are devoted to the advection-diffusion equation in 1D and 2D spaces. The fifth problem and sixth problem integrate movement of reflecting waves in a 1D closed tube and in a 2D quadrangular box.

```
subroutine IBVP_examples

              call Heat_equation_1D
              call Heat_equation_2D
              call Advection_Diffusion_1D
              call Advection_Diffusion_2D
              call Wave_equation_1D
              call Wave_equation_2D
end subroutine
```

Listing 6.1: `API_Example_Initial_Boundary_Value_Problem.f90`

The statement `use Initial_Boundary_Value_Problems` should be included at the beginning of the program to make use of the functions and the subroutines of this module.

6.2 Heat equation 1D

The heat equation is a partial differential equation that describes how the temperature evolves in a solid medium. The physical mechanism is the thermal conduction is associated with microscopic transfers of momentum within a body. The Fourier's law states the heat flux depends on temperature gradient and thermal conductivity. By imposing the energy balance of a control volume and taking into account the Fourier's law, the heat equation is derived:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$

This spatial domain is $\Omega \subset \mathbb{R} : \{x \in [-1, 1]\}$ and the temporal domain is $t \in [0, 1]$. The boundary conditions are set by imposing a given temperature or heat flux at boundaries. In this example, a homogeneous temperature is imposed at boundaries:

$$u(-1, t) = 0,$$
$$u(1, t) = 0,$$

and the initial temperature profile is:

$$u(x, 0) = \exp\left(-25x^2\right).$$

The implementation of the differential operator is done by means of the function `Heat_equation1D`

```
real function Heat_equation1D( x, t, u, ux, uxx) result(F)
          real, intent(in) ::  x, t, u, ux, uxx

             F =   uxx
end function
```

Listing 6.2: `API_Example_Initial_Boundary_Value_Problem.f90`

and the boundary conditions by means of the function `Heat_BC1D`

```
real function Heat_BC1D(x, t, u, ux) result(BC)
    real, intent(in) :: x, t, u, ux

        if (x==x0 .or. x==xf) then
                              BC = u
        else
             write(*,*)  "Error in Heat_BC1D"; stop
        endif
end function
```

Listing 6.3: `API_Example_Initial_Boundary_Value_Problem.f90`

The problem is integrated with piecewise polynomials of degree six or finite differences of sixth-order $q = 6$. Once the grid or the mesh points are chosen by the subroutine `Grid_initialization` and the initial condition is set, the problem is integrated by the subroutine `Initial_Boundary_Value_Problem` by making use of the definition of the differential operator and the boundary conditions previously defined.

```
!    Heat equation 1D
     call Grid_Initialization( "nonuniform", "x", q, x )

     U(0, :)  =  exp(-25*x**2 )
     call Initial_Boundary_Value_Problem(                                  &
                      Time_Domain = Time, x_nodes = x,                     &
                      Differential_operator =  Heat_equation1D,            &
                      Boundary_conditions   =  Heat_BC1D,                  &
                      Solution = U )
```

Listing 6.4: `API_Example_Initial_Boundary_Value_Problem.f90`

In figure 6.1, the temperature $u(x,t)$ is shown during time integration by different parametric curves. From the initial condition, the temperature diffuses to both sides of the spatial domain verifying zero temperature at boundaries.

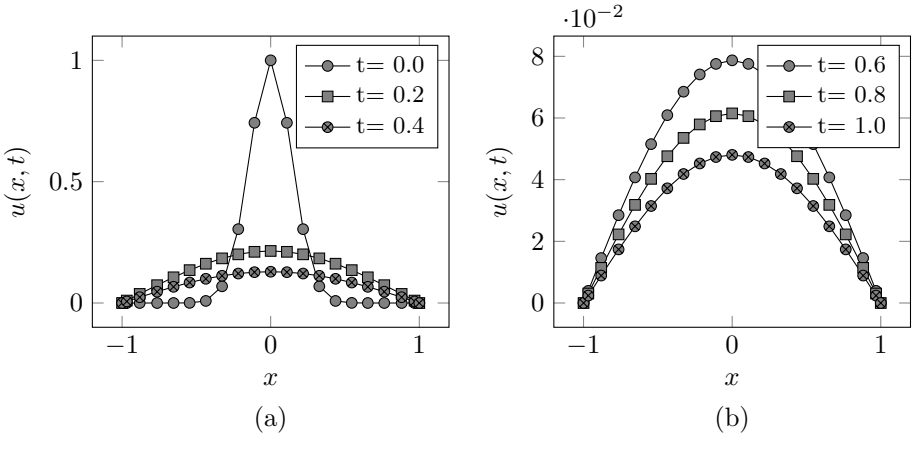

Figure 6.1: Time evolution of the heat equation with $Nx = 20$ and $q = 6$. (a) Temperature profile $u(x,t)$ at $t = 0, 0.2, 0.4$. (b) Temperature profile $u(x,t)$ at $t = 0.6, 0.8, 1$.

6.3 Heat equation 2D

In this section, the heat equation is integrated in a two-dimensional quadrangular box $\Omega \subset \mathbb{R}^2 : \{(x, y) \in [-1, 1] \times [-1, 1]\}$ allowing heat fluxes in both in vertical and horizontal directions. The heat equation expressed in a Cartesian two dimensional space is:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

This differential operator is implemented in `Heat_equation2D`

```
real function Heat_equation2D(x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy) result(F)
           real,intent(in) :: x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy

              F = Uxx + Uyy
end function
```

Listing 6.5: `API_Example_Initial_Boundary_Value_Problem.f90`

In this example, the influence of the non-homogeneous boundary conditions is taken into account by imposing the following temperatures at boundaries:

$$u(-1, y, t) = 1, \quad u(+1, y, t) = 0, \quad u(x, -1, t) = 0, \quad u(x, +1, t) = 0,$$

and zero temperature as an initial condition $u(x, y, 0) = 0$.

```
real function Heat_BC2D( x, y, t, U, Ux, Uy ) result (BC)
     real, intent(in) :: x, y, t, U, Ux, Uy

        if (x==x0) then
                                                    BC = U - 1
        else if (x==xf .or. y==y0 .or. y==yf ) then
                                                    BC = U
        else
            write(*,*)  "Error in Heat_BC2D"; stop
        end if
end function
```

Listing 6.6: `API_Example_Initial_Boundary_Value_Problem.f90`

The subroutine `InitialValue_Boundary_Problem` uses these definitions to integrate the solution

```
!    Heat equation 2D
     call Initial_Boundary_Value_Problem(                                  &
           Time_Domain = Time, x_nodes = x, y_nodes = y,                   &
           Differential_operator =  Heat_equation2D,                       &
           Boundary_conditions   =  Heat_BC2D,  Solution = U )
```

Listing 6.7: `API_Example_Initial_Boundary_Value_Problem.f90`

In figure 6.2, the two dimensional distribution of temperature is shown from the early stages of time to its final steady state. The temperature evolves from the zero initial condition to a steady state assuring the imposed boundary conditions.

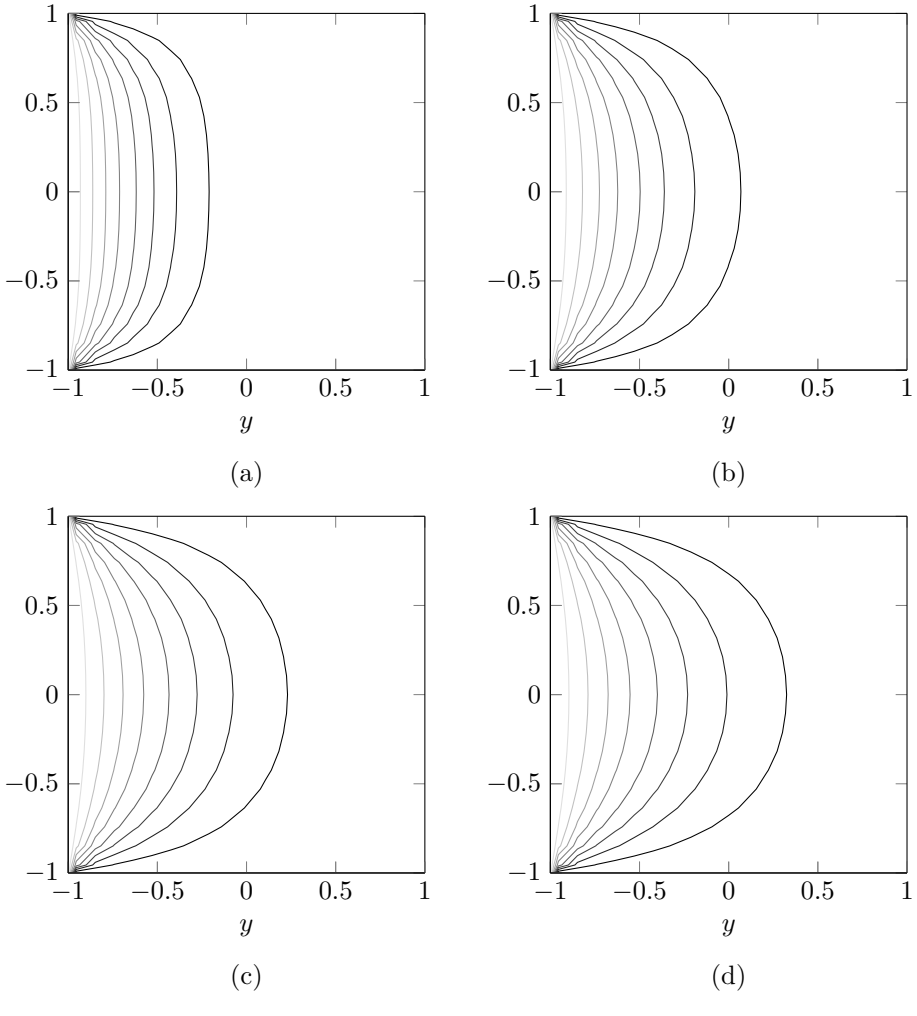

Figure 6.2: Solution of the 2D heat equation solution with $N_x = 20, N_y = 20$ and order $q = 4$, (a) temperature at $t = 0.125$, (b) temperature at $t = 0.250$,(c) temperature at $t = 0.375$, (d) temperature at $t = 0.5$.

6.4 Advection Diffusion equation 1D

When convection together with diffusion is present in the physical energy transfer mechanism, boundary conditions become tricky. For example, let us consider a fluid inside a pipe moving to the right at constant velocity transferring by conductivity heat to right and to the left. At the same time and due to its convective velocity, the energy is transported downstream. It is clear that the inlet temperature can be imposed but nothing can be said of the outlet temperature. In this section, the influence of extra boundary conditions is analyzed. The one-dimensional energy transfer mechanism associated to advection and diffusion is governed by the following equation:

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2},$$

where ν is a non dimensional parameter which measures the importance of the diffusion versus the convection. This spatial domain is $\Omega \subset \mathbb{R} : \{x \in [-1, 1]\}$. As it was mentioned, extra boundary conditions are imposed to analyze their effect.

$$u(-1, t) = 0, \quad u(1, t) = 0.$$

The convective and diffusive eveloution is studied from the following initial condition:

$$u(x, 0) = \exp\big(-25x^2\big).$$

The differential operator and the boundary equations are implemented in the following two subroutines:

```
real function Advection_equation1D( x, t, u, ux, uxx) result(F)
               real, intent(in) ::  x, t, u, ux, uxx

            real :: nu = 0.02

            F = - ux + nu * uxx
end function
```

Listing 6.8: API_Example_Initial_Boundary_Value_Problem.f90

```
real function Advection_BC1D(x, t, u, ux) result(BC)
         real, intent(in) :: x, t, u, ux

        if (x==x0 .or. x==xf) then
                            BC = u
        else
             write(*,*)  "Error in Advection_BC1D"; stop
        endif
end function
```

Listing 6.9: API_Example_Initial_Boundary_Value_Problem.f90

The subroutine `InitialValue_Boundary_Problem` uses these definitions to integrate the solution

```
!    Advection diffusion 1D
     call Initial_Boundary_Value_Problem(                                &
                  Time_Domain = Time, x_nodes = x,                       &
                  Differential_operator =  Advection_equation1D,         &
                  Boundary_conditions   =  Advection_BC1D,  Solution = U )
```

Listing 6.10: `API_Example_Initial_Boundary_Value_Problem.f90`

In this example, fourth-order finite differences $q =$ have been used. In figure 6.3, the evolution of the temperature is shown for the early stage of the simulation. The initial temperature profile moves to the right due to its constant velocity 1. At the same time, it diffuses to the right and to the left due to its conductivity. The problem arises when the temperature profile reaches the boundary $x = +1$ where the extra boundary condition $u(+1, t) = 0$ is imposed. The result of the simulation is observed in figure 6.3b where the presence of the extra boundary condition introduces oscillations of disturbances in the temperature profile which are not desirable. the solution evaluated at four instants of time.

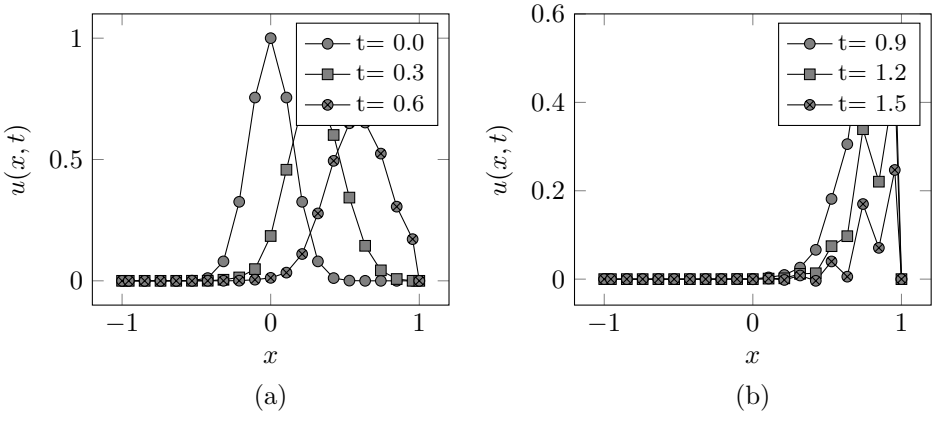

Figure 6.3: Solution of the advection-diffusion equation subjected to extra boundary conditions with $N_x = 20$ and order $q = 4$. (a) Early stages of the temperature profile for $t = 0, 0.3, 0.6$, (b) the temperature profile for $t = 0.9, 1.2, 1.5$.

6.5 Advection-Diffusion equation 2D

The purpose of this section is to show how the elimination of the extra boundary conditions imposed in the 1D advection-diffusion problem allows obtaining the desired result.

Let us consider a fluid moving with a given constant velocity $\boldsymbol{v}$. While the convective energy transfer mechanism is determined by $\boldsymbol{v} \cdot \nabla u$, the energy transferred by thermal conductivity is ∇u. With these considerations, the temperature evolution of the fluid is governed by the following equation:

$$\frac{\partial u}{\partial t} + \boldsymbol{v} \cdot \nabla u \;=\; \nu \, \nabla u,$$

here ν is a non dimensional parameter which measures the importance of the diffusion versus the convection.

In this example, $\boldsymbol{v} = (1, 0)$ and the energy transfer occurs in a two dimensional domain $\Omega \subset \mathbb{R}^2 : \{(x, y) \in [-1, 1] \times [-1, 1]\}$. The above equation yields,

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right).$$

This differential operator is implemented in the function `Advection_equation2D`

```
function Advection_equation2D(x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy) result(F)
           real,intent(in) :: x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy
           real :: F

        real :: nu = 0.02

        F = - Ux + nu * ( Uxx + Uyy )

end function
```

Listing 6.11: `API_Example_Initial_Boundary_Value_Problem.f90`

The constant velocity $\boldsymbol{v}$ of the flow allows deciding inflow or outflow boundaries by projecting the velocity on the normal direction to the boundary. In our case, only the boundary $x = +1$ is an outflow. It is considered the flow enter at zero temperature but no boundary condition is imposed at the outflow.

$$u(-1, y, t) = 0, \quad u(x, -1, t) = 0, \quad u(x, +1, t) = 0.$$

The question that arises is: if no boundary condition is imposed, how these boundaries conditions are modified or evolved ? The answer is to consider the boundary points as interior points. In this way, the evolution of these points is governed by

the advection-diffusion equation. To take into account that there are points with this requirement, the keyword `FREE_BOUNDARY_CONDITION` is used. In the following function `Advection_BC2D` these special boundary points are implemented:

```
real function Advection_BC2D( x, y, t, U, Ux, Uy ) result (BC)
          real, intent(in) :: x, y, t, U, Ux, Uy

        if (x==x0 .or. y==y0 .or. y==yf ) then
                                BC = U
        elseif (x==xf) then

                                BC =  FREE_BOUNDARY_CONDITION
        else
            Write(*,*)  "Error in Advection_BC2D"; stop
        end if

end function
```

Listing 6.12: `API_Example_Initial_Boundary_Value_Problem.f90`

The subroutine `InitialValue_Boundary_Problem` uses the function of the differential operator as well as the function that imposes the boundary conditions to integrate the solution

```
!    Advection diffusion 2D
     call Initial_Boundary_Value_Problem(                              &
         Time_Domain = Time, x_nodes = x, y_nodes = y,                 &
         Differential_operator =  Advection_equation2D,                &
         Boundary_conditions   =  Advection_BC2D,  Solution = U )
```

Listing 6.13: `API_Example_Initial_Boundary_Value_Problem.f90`

In figure 6.4, the temperature distribution is shown. At the early stages of the simulation 6.4a and 6.4b, the energy is transported to the right and at the same time the thermal conductivity diffuses its initial distribution. In figure 6.4c and 6.4d, the flow has reached the outflow boundary. Since no boundary conditions are imposed at the outflow boundary $x = +1$, the simulation predicts what is supposed to happen. The energy abandons the spatial domain with no reflections or perturbation in the temperature distribution.

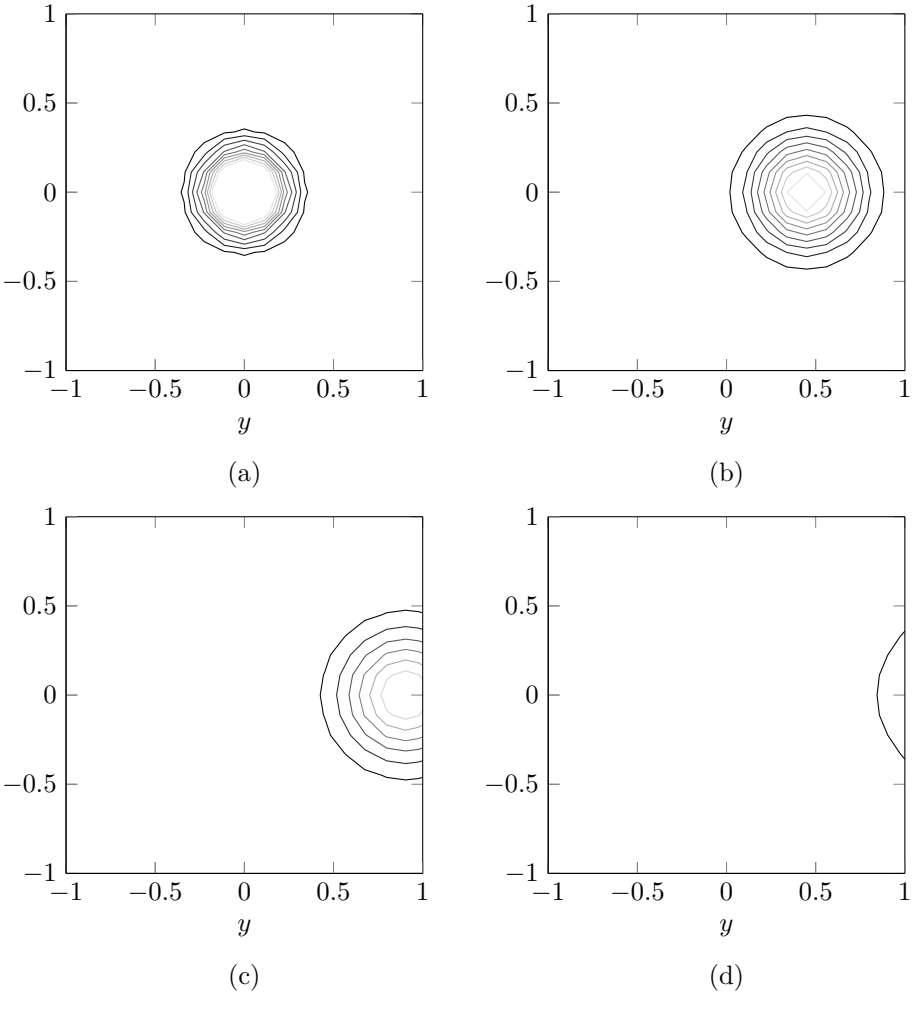

Figure 6.4: Solution of the advection-diffusion equation with outflow boundary conditions with $N_x = 20$,$N_y = 20$ and order $q = 8$. (a) Initial condition $u(x, y, 0)$, (b) solution at $t = 0.45$, (c) solution at $t = 0.9$, (d) solution at $t = 1.35$

6.6 Wave equation 1D

The wave equation is a conservative equation that describes waves such as pressure waves or sound waves, water waves, solid waves or light waves. It is a partial differential equation that predicts the evolution of a function $u(x,t)$ where x represents the spatial variable and t stands for time variable. The equation that governs the quantity $u(x,t)$ such as the pressure in a liquid or gas, or the displacement of some media is:

$$\frac{\partial^2 v}{\partial t^2} - \frac{\partial^2 v}{\partial x^2} = 0.$$

Since the module `Initial_Boundary_Value_Problems` is written for systems of second order derivatives in space and first order in time, the problem must be rewritten by means of the following transformation:

$$\boldsymbol{u}(x,t) = [\ v(x,t),\ w(x,t)\ ].$$

The wave equation is transformed in a system of equations of first order in time and second order in space

$$\frac{\partial v}{\partial t} = w,$$
$$\frac{\partial w}{\partial t} = \frac{\partial^2 v}{\partial x^2}.$$

This set of two differential equations is implemented in `Wave_equation1D`

```
function Wave_equation1D( x, t, u, ux, uxx) result(F)
     real, intent(in) ::  x, t, u(:), ux(:), uxx(:)
     real :: F(size(u))

        real :: v, vxx, w
        v = u(1); vxx = uxx(1);
        w = u(2);

        F  = [w,  vxx]

end function
```

Listing 6.14: `API_Example_Initial_Boundary_Value_Problem.f90`

These equations must be completed with initial and boundary conditions. In this example, a one-dimensional tube with closed ends is considered. This spatial domain is $\Omega \subset \mathbb{R} : \{x \in [-1,1]\}$ and the temporal domain is $t \in [0,4]$. It means that waves reflects at the boundaries conserving their energy with $v(\pm 1,t) = 0$ and $w(\pm 1,t) = 0$. The initial condition is $v(x,0) = \exp\left(-15x^2\right)$, and $w(x,0) = 0$. The boundary conditions are implemented in the following function `Wave_BC1D`:

```
function Wave_BC1D(x, t, u, ux) result(BC)
  real, intent(in) :: x, t, u(:), ux(:)
  real :: BC( size(u) )

        if (x==x0 .or. x==xf) then
                                  BC = u
        else
             write(*,*)  "Error in Waves_BC1D"; stop
        endif
end function
```

Listing 6.15: `API_Example_Initial_Boundary_Value_Problem.f90`

The differential operator and the boundary conditions function are used as input arguments of the subroutine `Initial_Boundary_Value_Problem`

```
!    Wave equation 1D
     call Initial_Boundary_Value_Problem(                              &
            Time_Domain = Time, x_nodes = x,                           &
            Differential_operator =  Wave_equation1D,                  &
            Boundary_conditions   =  Wave_BC1D,  Solution = U )
```

Listing 6.16: `API_Example_Initial_Boundary_Value_Problem.f90`

In figure 6.5, time evolution of $u(x,t)$ is shown. Since the initial condition is symmetric with respect to $x = 0$ and the system is conservative, the solution is periodic of periodicity $T = 4$. It is shown in 6.5b that the displacement profile $u(x,t)$ at $t = T$ coincides with the initial condition.

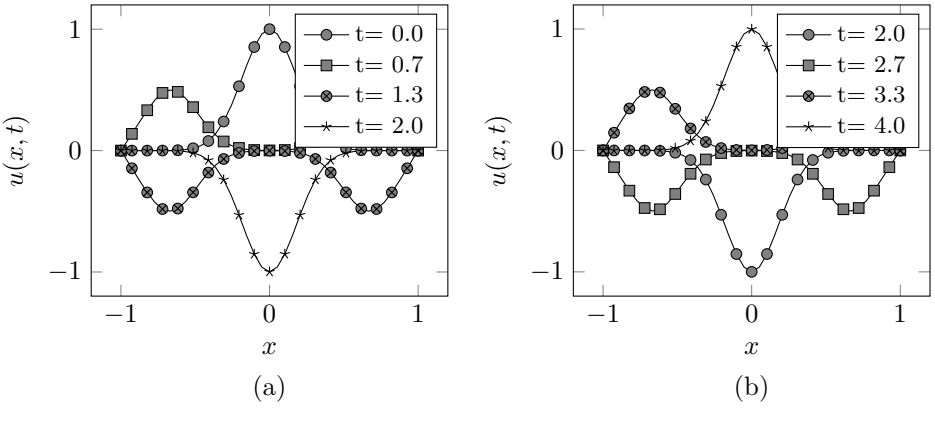

Figure 6.5: Wave equation solution with $N_x = 41$ and order $q = 6$. (a) Time evolution of $u(x,t)$ from $t = 0$ to $t = 2$. (b) Time evolution of $u(x,t)$ from $t = 2$ to $t = 4$.

6.7 Wave equation 2D

In two space dimensions, the wave equation is

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}$$

As it was done with the wave equation in 1D, the problem must transformed to a system of first order in time by means of the following change of variables:

$$\boldsymbol{u}(x,t) = [\ v(x,t),\ w(x,t)\ ]$$

giving rise to the system

$$\begin{aligned}\frac{\partial v}{\partial t} &= w,\\ \frac{\partial w}{\partial t} &= \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}.\end{aligned}$$

This system is implemented in the function `Wave_equation2D`

```
function Wave_equation2D( x, y, t, u, ux, uy, uxx, uyy, uxy ) result(L)
      real, intent(in) ::  x,y,t,u(:),ux(:),uy(:),uxx(:),uyy(:),uxy(:)
      real :: L(size(u))

            real :: v, vxx, vyy, w

            v = u(1);  vxx = uxx(1); vyy = uyy(1)
            w = u(2);

            L(1)  = w
            L(2)  = vxx +vyy

end function
```

Listing 6.17: `API_Example_Initial_Boundary_Value_Problem.f90`

Regarding boundary conditions, reflexive or non absorbing walls are considered in the spatial domain $\Omega \equiv \{(x,y) \in [-1,1] \times [-1,1]\}$. The time interval is $t \in [0,2]$. Hence, the boundary conditions are:

$$\begin{aligned} &v(+1,y,t) = 0, \quad v(-1,y,t) = 0, \quad v(x,-1,t) = 0, \quad v(x,+1,t) = 0,\\ &w(+1,y,t) = 0, \quad w(-1,y,t) = 0, \quad w(x,-1,t) = 0, \quad w(x,+1,t) = 0. \end{aligned}$$

And the initial values:

$$\begin{aligned} v(x,y,0) &= \exp\bigl(-10(x^2+y^2)\bigr),\\ w(x,y,0) &= 0. \end{aligned}$$

The boundary conditions are implemented in `Wave_BC2D`

```
function Wave_BC2D(x,y, t, u, ux, uy) result(BC)
  real, intent(in) :: x, y,  t, u(:), ux(:), uy(:)
  real :: BC( size(u) )

        real :: v, w
        v = u(1)
        w = u(2)

        if (x==x0 .or. x==xf .or. y==y0 .or. y==yf ) then
                            BC = [v, w]
        else
             write(*,*)  "Error in BC2D_waves"; stop
        endif

end function
```

Listing 6.18: `API_Example_Initial_Boundary_Value_Problem.f90`

The differential operator, its boundary conditions and initial condition are used in the following code snippet:

```
!    Wave equation 2D
     call Grid_Initialization( "nonuniform", "x", Order, x )
     call Grid_Initialization( "nonuniform", "y", Order, y )

     U(0, :, :, 1) = Tensor_product( exp(-10*x**2) , exp(-10*y**2) )
     U(0, :, :, 2) = 0

     call Initial_Boundary_Value_Problem(                              &
                       Time_Domain = Time, x_nodes = x, y_nodes = y,  &
                       Differential_operator = Wave_equation2D,       &
                       Boundary_conditions = Wave_BC2D, Solution = U )
```

Listing 6.19: `API_Example_Initial_Boundary_Value_Problem.f90`

In figure 6.6 time evolution of $u(x, y, t)$ is shown from the initial condition to time $t = 2$. Since waves reflect from different walls with different directions and the round trip time depends on the direction, the problem becomes much more complicated to analyze than the pure one-dimensional problem.

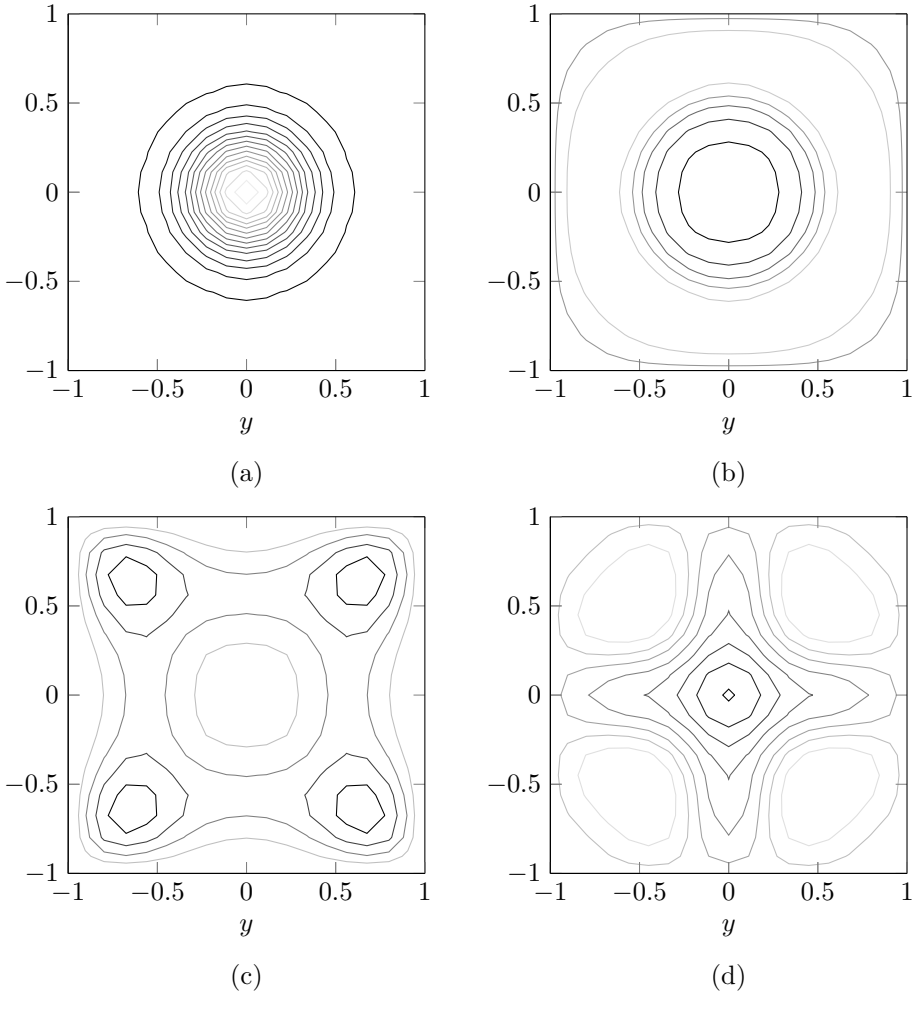

Figure 6.6: Wave equation solution with $N_x = 20$, $N_y = 20$ and order $q = 8$. (a) Initial value $u(x, y, 0)$. (b) Numerical solution at $t = 0.66$, (c) numerical solution at $t = 1.33$, (d) numerical solution at $t = 2$.

Chapter 7

Mixed Boundary and Initial Value Problems

7.1 Overview

In this chapter, a mixed problem coupled with elliptic and parabolic equations is solved making use of the module `IBVP_and_BVP`. Briefly and not rigorously, these problems are governed by a parabolic time dependent problem for $\boldsymbol{u}(\boldsymbol{x},t)$ and an elliptic problem or a boundary value problem for $\boldsymbol{v}(\boldsymbol{x},t)$. Let Ω be an open and connected set and $\partial\Omega$ its boundary. These problems are formulated with the following set of equations:

$$
\begin{aligned}
&\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x},t) = \boldsymbol{\mathcal{L}}_u(\boldsymbol{x},t,\boldsymbol{u}(\boldsymbol{x},t),\boldsymbol{v}(\boldsymbol{x},t)), && \forall\ \boldsymbol{x}\in\Omega,\\
&\boldsymbol{h}_u(\boldsymbol{x},t,\boldsymbol{u}(\boldsymbol{x},t))\big|_{\partial\Omega} = 0, && \forall\ \boldsymbol{x}\in\partial\Omega,\\
&\boldsymbol{u}(\boldsymbol{x},t_0) = \boldsymbol{u}_0(\boldsymbol{x}), && \forall\ \boldsymbol{x}\in D,\\
&\\
&\boldsymbol{\mathcal{L}}_v(\boldsymbol{x},t,\boldsymbol{v}(\boldsymbol{x},t),\boldsymbol{u}(\boldsymbol{x},t)) = 0, && \forall\ \boldsymbol{x}\in\Omega,\\
&\boldsymbol{h}_v(\boldsymbol{x},t,\boldsymbol{v}(\boldsymbol{x},t))\big|_{\partial\Omega} = 0, && \forall\ \boldsymbol{x}\in\partial\Omega,
\end{aligned}
$$

where $\boldsymbol{\mathcal{L}}_u$ is the spatial differential operator of the initial value problem of N_u equations, $\boldsymbol{u}_0(\boldsymbol{x})$ is the initial value, $\boldsymbol{h}_u$ represents the boundary conditions operator for the solution at the boundary points $\boldsymbol{u}\big|_{\partial\Omega}$, $\boldsymbol{\mathcal{L}}_v$ is the spatial differential operator of the boundary value problem of N_v equations and $\boldsymbol{h}_v$ represents the boundary conditions operator for $\boldsymbol{v}$ at the boundary points $\boldsymbol{v}\big|_{\partial\Omega}$.

7.2 Non Linear Plate Vibration

The vibrations $w(x, y, t)$ of an nonlinear plate subjected to a transversal load $p(x, y, t)$ are governed by the following set of equations:

$$\frac{\partial^2 w}{\partial t^2} + \nabla^4 w = p(x, y, t) + \mu\ \mathcal{B}(w, \phi),$$
$$\nabla^4 \phi + \mathcal{B}(w, w) = 0,$$

where $\mathcal{B}$ is the bilinear operator:

$$\mathcal{B}(w, \phi) = \frac{\partial^2 w}{\partial x^2}\frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial x^2}\frac{\partial^2 w}{\partial y^2} - 2\frac{\partial^2 \phi}{\partial x \partial y}\frac{\partial^2 w}{\partial x \partial y}.$$

These equations together with boundary and initial conditions allow predicting the oscillations of the plate. Since second order derivatives of the displacement $w(x, y, t)$ are involved, the initial position and the initial velocity are given. In this example,

$$w(x, y, 0) = e^{-10(x^2+y^2)}.$$
$$\frac{\partial w}{\partial t}(x, y, 0) = 0.$$

Besides, simple supported edges are considered which means that the displacement $w(x, y, t)$ and the bending moments $\nabla^2 w$ are zero at boundaries. In this example, the spatial domain $\Omega \equiv \{(x, y) \in [-1, 1] \times [-1, 1]\}$ and the time domain $t \in [0, 1]$. Since the module `IBVP_and_BVP` is written for systems of second order derivatives in space and first order in time, the problem is rewritten by means of the following transformation:

$$\boldsymbol{u} = [\ w,\ w_2,\ w_3\ ],$$

which leads to the evolution system of equations for $\boldsymbol{u}(x, y, t)$ with:

$$\boldsymbol{\mathcal{L}}_u = [\ w_2,\ -\nabla^2 w_3 + p + \mu\ \mathcal{B}(w, \phi),\ \nabla^2 w_2\ ],$$

To implement the elliptic boundary problem in terms of second order derivatives, the following transformation is used:

$$\boldsymbol{v} = [\ \phi,\ F\ ],$$

which leads to the system $\boldsymbol{\mathcal{L}}_v(\boldsymbol{x}, t, \boldsymbol{v}, \boldsymbol{u}) = 0$ with:

$$\boldsymbol{\mathcal{L}}_v = [\ \nabla^2 \phi - F,\ \nabla^2 F + \mathcal{B}(w, w)\ ].$$

The evolution differential operator $\boldsymbol{\mathcal{L}}_u$ together with the elliptic differential $\boldsymbol{\mathcal{L}}_v$ are implemented in the following vector functions:

```
function Lu(x, y, t, u, ux, uy, uxx, uyy, uxy , v, vx, vy, vxx, vyy, vxy)
 real, intent(in) ::  x, y, t, u(:), ux(:), uy(:), uxx(:), uyy(:), uxy(:)
 real, intent(in) ::           v(:), vx(:), vy(:), vxx(:), vyy(:), vxy(:)
 real :: Lu(size(u))

     real :: wxx, wyy, wxy, pxx, pyy, pxy
     real :: w2, w2xx, w2yy,  w3xx, w3yy

     wxx  = uxx(1); wyy  = uyy(1); wxy = uxy(1);
     w2xx = uxx(2); w2yy = uyy(2); w2  = u(2);
     w3xx = uxx(3); w3yy = uyy(3);
     pxx  = vxx(1); pyy  = vyy(1); pxy = vxy(1);


     Lu(1)  =    w2
     Lu(2)  =  - w3xx - w3yy + load(x, y, t)              &
               + mu * B( wxx, wyy, wxy, pxx, pyy, pxy)
     Lu(3)  =    w2xx + w2yy

end function
```

Listing 7.1: `API_Example_IBVP_and_BVP.f90`

```
function Lv( x, y, t, v, vx, vy, vxx, vyy, vxy , u, ux, uy, uxx, uyy, uxy)
 real, intent(in) ::  x, y, t, u(:), ux(:), uy(:), uxx(:), uyy(:), uxy(:)
 real, intent(in) ::           v(:), vx(:), vy(:), vxx(:), vyy(:), vxy(:)
 real :: Lv(size(v))

     real :: wxx, wyy, wxy, pxx, pyy, pxy, Fxx, Fyy, Fxy, F

     pxx = vxx(1); pyy = vyy(1); pxy = vxy(1);
     Fxx = vxx(2); Fyy = vyy(2); Fxy = vxy(2); F = v(1);
     wxx = uxx(1); wyy = uyy(1); wxy = uxy(1);

     Lv(1) = pxx + pyy - F
     Lv(2) = Fxx + Fyy + B(wxx, wyy, wxy, wxx, wyy, wxy)

end function
```

Listing 7.2: `API_Example_IBVP_and_BVP.f90`

To impose simple supported edges, the values of all components of $\boldsymbol{u}(x, y, t)$ and $\boldsymbol{v}(x, y, t)$ must be determined analytically at boundaries. Since $w(x, y, t)$ is zero at boundaries for all time and $w_2 = \partial w/\partial t$ then, w_2 is zero at boundaries. Since $\nabla^2 w$ is zero at boundaries for all time and

$$\frac{\partial w_3}{\partial t} = \frac{\partial \nabla^2 w}{\partial t}$$

then, w_3 is zero at boundaries. The same reasoning is applied to determine $\boldsymbol{v}(x, y, t)$ components at boundaries.With these considerations, the boundary conditions $\boldsymbol{h}_u$ and $\boldsymbol{h}_v$ are implemented by:

```
function BCu(x, y, t, u, ux, uy)
  real, intent(in) :: x, y,  t, u(:), ux(:), uy(:)
  real :: BCu( size(u) )

        if (x==x0 .or. x==xf .or. y==y0 .or. y==yf) then
                           BCu = u
        else
             write(*,*)  "Error in BC1 "; stop
             stop
        endif

end function
```

Listing 7.3: API_Example_IBVP_and_BVP.f90

```
function BCv(x, y, t, v, vx, vy)
  real, intent(in) :: x, y,  t, v(:), vx(:), vy(:)
  real :: BCv( size(v) )

        if (x==x0 .or. x==xf .or. y==y0 .or. y==yf) then
                           BCv = v
        else
             write(*,*)  "Error in BC2 "; stop
             stop
        endif

end function
```

Listing 7.4: API_Example_IBVP_and_BVP.f90

These differential operators Lu and Lv together with their boundary conditions BCu and BCv are used as input arguments for the subroutine IBVP_and_BVP

```
!  Nonlinear Plate Vibration
   call IBVP_and_BVP( Time, x, y, Lu, Lv, BCu, BCv, U, V )
```

Listing 7.5: API_Example_IBVP_and_BVP.f90

In figure 7.1, the oscillations of a plate with zero external loads starting from an elongated position with zero velocity are shown.

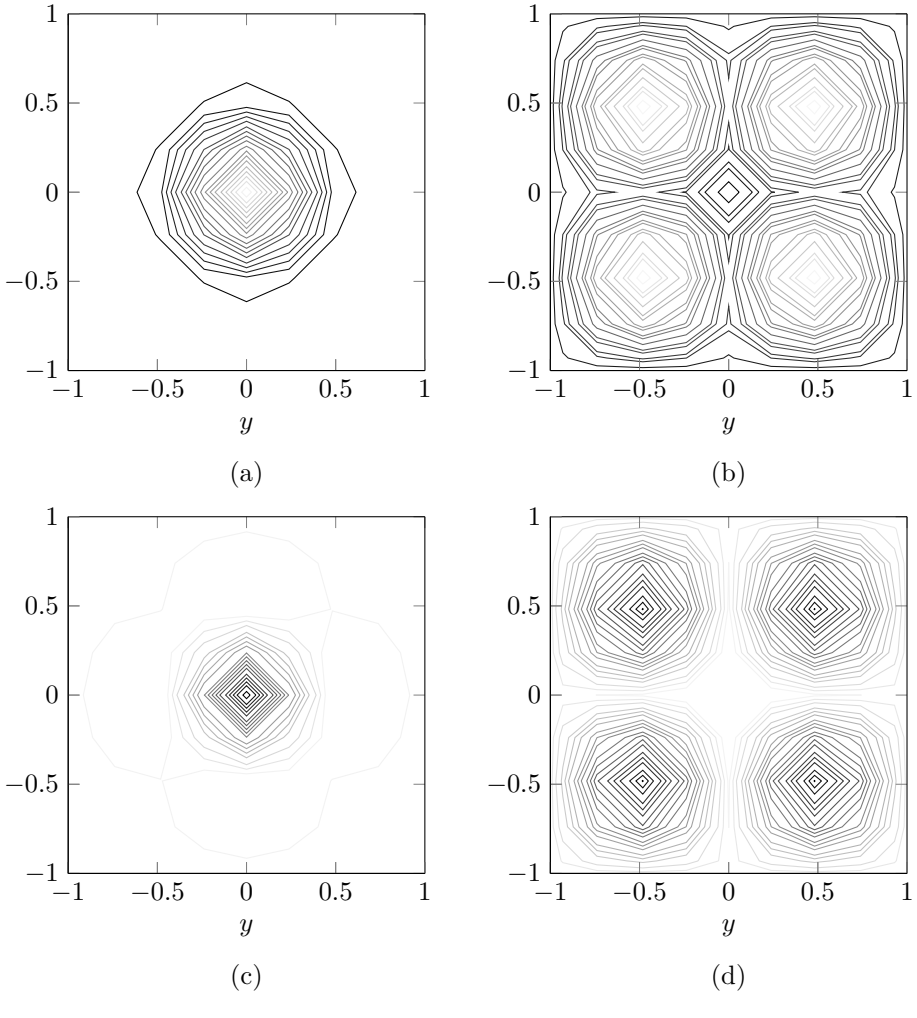

Figure 7.1: Time evolution of nonlinear vibrations $w(x, y, t)$ with 11×11 nodal points and order $q = 6$. (a) $w(x, y, 0.25)$. (b) $w(x, y, 0.5)$. (c) Numerical solution at $w(x, y, 0.75)$. (d)Numerical solution at $w(x, y, 1)$.

Part II

Developer guidelines

Chapter 1

Systems of equations

1.1 Overview

In this chapter, it is intended to cover the implementation of some classic issues that might appear in algebraic problems from applied mathematics. In particular, the operations related to linear and non linear systems and operations with matrices such as: LU factorization, real eigenvalues and eigenvectors computation and SVD decomposition will be presented.

1.2 Linear systems and LU factorization

In all first courses in linear algebra, the resolution of linear systems is treated as the fundamental problem to be solved. This problem consists on solving for an unknown $\boldsymbol{x} \in \mathbb{R}^N$ the problem:

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1.1}$$

where $A \in \mathcal{M}_{N\times N}$ verifies $\det(A) \neq 0$ and $\boldsymbol{b} \in \mathbb{R}^N$.

Concepts such as linear combination, pivots and elemental operations matrices are used, leading to the well-known Gauss elimination method. This method operates by rows on an extended matrix which contains all the columns of A and $\boldsymbol{b}$ as its last column until the rows of A form an upper diagonal matrix. Once we have the upper diagonal matrix, the resolution of the problem is straightforward. However, even though Gauss elimination is a successful algorithm to deal with linear systems, its straightforward implementation has an inconvenient: it depends on $\boldsymbol{b}$. This means that every time we change the independent term we have to apply again the algorithm and perform around N^3 operations. This is undesirable as in many situations, we need to compute the solution of $A\boldsymbol{x} = \boldsymbol{b}$ for different source terms. A more efficient manner to think of Gaussian elimination is through LU factorization. The latter method is based on the fact that to reach the upper diagonal matrix of Gauss method, which we will denote U, a bunch of elemental row operations have to be performed over A. This means that there exists a $N \times N$ invertible matrix L^{-1} containing these operations such that when A is premultiplied by it we get U. In other words this means that we can express A as:

$$A = LU, \tag{1.2}$$

where L and U are lower and upper triangular matrices respectively. Note that as U is obtained through a Gaussian elimination process, the number of operations to compute LU factorization is the same. However, relation (1.2) gives a recursion to obtain both L and U operating only over elements of A. The factorization of A is equivalent to the relation between their components:

$$A_{ij} = \sum_m L_{im}U_{mj}, \quad \text{for} \quad m \in [1, \min\{i, j\}], \tag{1.3}$$

from which we want to obtain L_{ij} and U_{ij}. By definition, the number of non null terms in L and U are $N(N+1)/2$, which leads to $N^2 + N$ unknown variables. However, the number of equations supplied by (1.2) is N^2. This makes necessary to fix the value of N unknown variables. To solve this, we force $L_{kk} = 1$. Once this is done, we can obtain the k-th row of U from the equation for the components A_{kj} with $j \geq k$ if the previous k rows are known. Taking into account that $U_{11} = A_{11}$ we can compute the recursion

$$U_{kj} = A_{kj} - \sum_m L_{km}U_{mj}, \quad \text{for} \quad m \in [1, k-1]. \tag{1.4}$$

Note that the first row of U is just the first row of A. Hence, we can calculate each row of U recursively by a direct implementation.

```
   do j=k, N
        A(k,j) =  A(k,j) - dot_product( A(k, 1:k-1), A(1:k-1, j) )
   end do
```

Listing 1.1: `Linear_systems.f90`

Note that the upper diagonal matrix U is stored on the upper diagonal elements of A. Once we haved calculated U, a recursion to obtain the i-th row of, if all the previous $i-1$ rows of L are known. Note that for $i > j = 1$, $A_{i1} = L_{i1}U_{11}$ and the first column of L can be given as an initial condition. Therefore, we can compute the recursion as:

$$L_{ik} = \frac{A_{ik} - \sum_m L_{mk}U_{im}}{U_{kk}}, \quad \text{for} \quad m \in [1, k-1]. \tag{1.5}$$

Again, this recursion can be computed through a direct implementation:

```
   do i=k+1, N
        A(i,k) = (A(i,k) - dot_product( A(1:k-1, k), A(i, 1:k-1) ) )/A(k,k)
   end do
```

Listing 1.2: `Linear_systems.f90`

The factorization of A is implemented in the subroutine `LU\_factorization`

```
subroutine LU_factorization( A )
      real, intent(inout) :: A(:, :)

  integer :: N
  integer :: k, i, j
  N =size(A, dim =1)

  A(1, :) = A(1,:)
  A(2:N,1) = A(2:N,1)/A(1,1)

  do k=2, N
   do j=k, N
        A(k,j) =  A(k,j) - dot_product( A(k, 1:k-1), A(1:k-1, j) )
   end do

   do i=k+1, N
        A(i,k) = (A(i,k) - dot_product( A(1:k-1, k), A(i, 1:k-1) ) )/A(k,k)
   end do
  end do
end subroutine
```

Listing 1.3: `Linear_systems.f90`

Once the matrix A is factorized it is possible to solve the system (1.1). In first place it is defined $\boldsymbol{y} = U\boldsymbol{x}$, and thus:

$$\sum_j L_{ij} y_j = b_i, \quad \text{for} \quad j \in [1, i]. \tag{1.6}$$

As $L_{ij} = 0$ for $i < j$, and $L_{ii} = 1$ the first row of (1.6) gives $y_1 = b_1$ and the value of each y_i can be written on terms of the previous y_j, that is:

$$y_i = b_i - \sum_j L_{ij} y_j, \quad \text{for} \quad 1 < j < i, \tag{1.7}$$

thus, sweeping through $i = 2, \ldots N$, over (1.7) $\boldsymbol{y}$ is obtained. This is computed through a direct implementation as:

```
do i=2,N
       y(i) = b(i) - dot_product( A(i, 1:i-1), y(1:i-1) )
enddo
```

Listing 1.4: `Linear_systems.f90`

To obtain $\boldsymbol{x}$ it is used the definition of $\boldsymbol{y}$, which is written:

$$y_i = \sum_j U_{ij} x_j, \quad \text{for} \quad j \in [i, N]. \tag{1.8}$$

In a similar manner than before, as $u_{ij} = 0$ for $i > j$, the last row of (1.8) gives $x_N = y_N / u_{NN}$ and each x_i can be written in terms of the next x_j with $i < j \leq N$ as expresses the equation (1.9):

$$x_i = \frac{y_i - \sum_j U_{ij} x_j}{u_{ii}}, \quad \text{for} \quad j \in [i, N]. \tag{1.9}$$

Therefore, evaluating recursively $i = N - 1, \ldots 1$ (1.9), the solution $\boldsymbol{x}$ is obtained. The implementation of this recursion is straightforward:

```
do i=N-1, 1, -1
 x(i) = (y(i) - dot_product( A(i, i+1:N), x(i+1:N) ) )/ A(i,i)
end do
```

Listing 1.5: `Linear_systems.f90`

Hence, we can contain the whole process of obtaining the solution of $LU\boldsymbol{x} = \boldsymbol{b}$ in a subroutine named `Solve_LU` which contains the presented pieces of code along with the initialization $y_1 = b_1$.

```
function Solve_LU( A, b )
  real, intent(in) :: A(:, :), b(:)
  real :: Solve_LU( size(b) )

   real :: y (size(b)), x(size(b))
   integer :: i, N

   N = size(b)

   y(1) = b(1)
   do i=2,N
          y(i) = b(i) - dot_product( A(i, 1:i-1), y(1:i-1) )
   enddo

    x(N) = y(N) / A(N,N)
    do i=N-1, 1, -1
     x(i) = (y(i) - dot_product( A(i, i+1:N), x(i+1:N) ) )/ A(i,i)
    end do

    Solve_LU = x

end function
```

Listing 1.6: `Linear_systems.f90`

1.3 Non linear systems of equations

A nonlinear system of equations is a set of simultaneous equations in which the unknowns appear non-linearly. In other words, the equations to be solved cannot be written as a linear combination of the unknown variables. As nonlinear equations are difficult to solve, nonlinear systems are commonly approximated by linear equations or linearized.

Let $\boldsymbol{f} : \mathbb{R}^N \to \mathbb{R}^N$ be a real mapping and $\boldsymbol{x} \in \mathbb{R}^N$ the independent variable. The roots of system can be calculated by solving:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}. \tag{1.10}$$

As in general there is no analytical way of solving (1.10) for $\boldsymbol{x}$, methods which give approximate solutions have been developed historically. There are many ways to approximate the solution of (1.10) but the most famous method for differentiable functions was developed by sir Isaac Newton from whom receives its name. The goal of Newton method is to construct a sequence which converges to the solution $\boldsymbol{x}$ by linearizing $\boldsymbol{f}$ around a point $\boldsymbol{x}_i$, called initial guess. That is, the sequence must provide a point $\boldsymbol{x}_{i+1}$ which is closer to $\boldsymbol{x}$ than $\boldsymbol{x}_i$. To do this, the method takes into account that for every differentiable mapping there is a neighborhood of $\boldsymbol{x}_i$ in which we can approximate the function as:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_i) + \nabla \boldsymbol{f}(\boldsymbol{x}_i) \cdot (\boldsymbol{x} - \boldsymbol{x}_i) + O\left(\|\boldsymbol{x} - \boldsymbol{x}_i\|^2\right), \tag{1.11}$$

where $\nabla \boldsymbol{f}$ is the gradient or Jacobian matrix of $\boldsymbol{f}$. Hence, the sequence is constructed by evaluating (1.11) in the next iteration initial guess $\boldsymbol{x}_i$ and imposing that $\boldsymbol{f}(\boldsymbol{x}_i) = \boldsymbol{0}$, leading to the system of equations:

$$\nabla \boldsymbol{f}(\boldsymbol{x}_i) \cdot (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) = -\boldsymbol{f}(\boldsymbol{x}_i), \tag{1.12}$$

and if $\boldsymbol{f}$ is invertible in a neighborhood of $\boldsymbol{x}_i$ we can write[1]:

$$\boldsymbol{x}_{i+1} - \boldsymbol{x}_i = -\left(\nabla \boldsymbol{f}(\boldsymbol{x}_i)\right)^{-1} \cdot \boldsymbol{f}(\boldsymbol{x}_i), \tag{1.13}$$

where $(\nabla \boldsymbol{f}(\boldsymbol{x}_i))^{-1}$ is the inverse of the Jacobian matrix. Equation (1.13) provides an explicit sequence which converges to the solution of (1.10) $\boldsymbol{x}$ if the initial condition is sufficiently close to it. Hence, a recursive iteration on (1.13) will give an approximate solution of the non linear problem. The recursion is stopped defining a convergence criteria for $\boldsymbol{x}$. That is, the recursion will stop when $\|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\| \leq \varepsilon$, where ε is a sufficiently small positive number for the desired accuracy. The implementation of an algorithm which computes the Newton method for any function is presented in the following pages.

[1]Local invertibility is equivalent to the invertibility of the Jacobian matrix as the inverse function theorem states.

1. **Jacobian matrix calculation**

In order to implement the Newton method, first, we have to calculate the Jacobian matrix of the function. To avoid an excessive analytical effort, the columns of $\nabla \boldsymbol{f}(\boldsymbol{x}_i)$ are calculated using order 2 centered finite differences:

$$\frac{\partial \boldsymbol{f}(\boldsymbol{x}_i)}{\partial x_j} \simeq \frac{\boldsymbol{f}(\boldsymbol{x}_i + \Delta x \boldsymbol{e}_j) - \boldsymbol{f}(\boldsymbol{x}_i - \Delta x \boldsymbol{e}_j)}{2\Delta x}, \tag{1.14}$$

where $\boldsymbol{e}_j = (0, \ldots, 1, \ldots, 0)$ is the canonical basis vector whose only non zero entry is the j-th. Thus, the implementation of the computation of each column $\partial \boldsymbol{f}(\boldsymbol{x}_i)/\partial x_j$ is straightforward:

```
      Jacobian(:,j) =  ( F(xp + xj) - F(xp - xj) )/norm2(2*xj);
```

Listing 1.7: `Jacobian_module.f90`

where `xj` is a small perturbation along the coordinate x_j. The calculation of all the Jacobian columns is implemented in a function called `Jacobian`, which computes the gradient at the point x_i sweeping through $j \in [1, N]$, that is introducing the piece of code exposed in a `do` loop:

```
function Jacobian( F, xp )
  procedure (FunctionRN_RN) :: F
  real, intent(in) :: xp(:)
  real :: Jacobian( size(xp), size(xp) )

   integer ::  j, N
   real :: xj( size(xp) )

    N = size(Xp)

    do j = 1, N
       xj = 0
       xj(j) = 1d-3
       Jacobian(:,j) =  ( F(xp + xj) - F(xp - xj) )/norm2(2*xj);
    enddo

end function
```

Listing 1.8: `Jacobian_module.f90`

Hence, the Jacobian can be calculated by a simple call.

```
      J = Jacobian( F, x0 )
```

Listing 1.9: `Non_linear_systems.f90`

2. **Linear system solution**

Once the Jacobian is calculated, it is used to compute the next iteration initial guess $\boldsymbol{x}_{i+1}$. Instead of computing the inverse of the Jacobian, we solve the system:

$$\nabla \boldsymbol{f}(\boldsymbol{x}_i) \cdot \Delta \boldsymbol{x}_i = \boldsymbol{f}(\boldsymbol{x}_i), \tag{1.15}$$

whose solution is $\Delta \boldsymbol{x}_i = \boldsymbol{x}_i - \boldsymbol{x}_{i+1}$. This is implemented by performing a LU factorization as explained in the previous section:

```
call LU_factorization( J )
b = F(x0);
Dx = Solve_LU( J,  b )
```

Listing 1.10: `Non_linear_systems.f90`

and thus, the calculation of $\Delta \boldsymbol{x}_i$ allows to compute $\boldsymbol{x}_{i+1}$ as:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \Delta \boldsymbol{x}_i,$$

```
x0 = x0 - Dx;
```

Listing 1.11: `Non_linear_systems.f90`

3. **Next iteration**

Once we have calculated the next iteration initial guess $\boldsymbol{x}_{i+1}$ we just have to make the assignation:

$$i \to i+1, \qquad \boldsymbol{x}_i \to \boldsymbol{x}_{i+1}. \tag{1.16}$$

The assignation of the value $\boldsymbol{x}_{i+1}$ to $\boldsymbol{x}_i$ is done immediately as the value of the former is stored over the latter in the vector `x0`. The iteration evolution is implemented as:

```
iteration = iteration + 1
```

Listing 1.12: `Non_linear_systems.f90`

This process is carried out until $\|\Delta \boldsymbol{x}_i\| \leq \varepsilon = 10^{-8}$ or a maximum number of iterations `itmax` is achieved. This means that the pieces of code presented above have to be contained in a conditional loop whose mask takes into account the convergence criteria. Thus, the iterative process is embedded in a subroutine named `Newton` which takes the initial guess through its input `x0` and the function to be solved as a module procedure. To avoid overflows, the mask for the conditional loop is not only the convergence criteria but also that the number of iterations does not overcome `itmax`. In addition, a warning message is displayed on command line if this latter condition is not satisfied. In case that happens it means that the solution may not be as accurate as specified and the subroutine also displays the final value of $\|\Delta \boldsymbol{x}_i\|$ which is stored on the scalar `eps`:

```
subroutine Newton (F, x0)

 procedure (FunctionRN_RN) :: F
 real, intent(inout) :: x0(:)

   real ::   Dx( size(x0) ), b(size(x0)), eps
   real :: J( size(x0), size(x0) )
   integer :: iteration, itmax = 1000

   integer :: N

   N = size(x0)

   Dx = 2 * x0
   iteration = 0
   eps = 1

   do while ( eps > 1d-8 .and. iteration <= itmax )

      iteration = iteration + 1
      J = Jacobian( F, x0 )

      call LU_factorization( J )
      b = F(x0);
      Dx = Solve_LU( J,  b )

      x0 = x0 - Dx;

      eps = norm2( DX )

   end do

   if (iteration == itmax) then
      write(*,*) " morm2(J) =", maxval(J),  minval(J)
      write(*,*) " Norm2(Dx) = ", eps, iteration
   endif


end subroutine
```

Listing 1.13: `Non_linear_systems.f90`

1.4 Eigenvalues and eigenvectors

Calculation of eigenvalues and eigenvectors is a fundamental problem from linear algebra with a wide variety of applications: structural analysis, image processing, stability of orbits and even the most famous search engine requires of computing eigenvectors. In this section, an introduction to eigenvalues and eigenvectors computation and a piece of their foundings will be presented. Besides, we shall see how to implement algorithms to obtain eigenvalues and eigenvectors for a certain set of normal matrices. The eigenvalues and eigenvectors problem for a real square matrix A consists on finding all scalars λ_i and non zero vectors $\boldsymbol{v}_i$ such that:

$$(A - \lambda_i I)\boldsymbol{v}_i = 0. \tag{1.17}$$

In figure 1.1 is given a classification from the spectral point of view, of the possible situations for a real square matrix. The main characteristic which will classify a matrix is whether is normal or not. A normal matrix commutes with its transpose, that is, it verifies $AA^T = A^T A$ and these matrices can be diagonalized by orthonormal vectors. The fact that normal matrices can be diagonalized by a set of orthonormal vectors is a consequence of Schur decomposition theorem[2] and the fact that all normal upper-triangular matrices are diagonal. A practical manner to check (not to prove) this fact is by taking two eigenvectors $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ of A and noticing that:

$$\boldsymbol{v}_i \cdot A\boldsymbol{v}_j = \lambda_j \boldsymbol{v}_i \cdot \boldsymbol{v}_j, \tag{1.18}$$

and if $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ are orthogonal and unitary, then the matrix whose components are given by

$$D_{ij} = \boldsymbol{v}_i \cdot A\boldsymbol{v}_j = \lambda_j \delta_{ij},$$

is diagonal and its non zero entries are the eigenvalues of A. This means that defining a matrix V whose columns are the eigenvectors of A, we can factorize A as:

$$A = VDV^*, \tag{1.19}$$

where V^* stands for the conjugate transpose of V. Until now, we have not specified to which field ($\mathbb{R}$ or $\mathbb{C}$) belong the eigenvalues of λ and over which field is defined the vector space containing its eigenvectors. A sufficient condition for a real matrix to have real eigenvalues and eigenvectors is given by the spectral theorem: all symmetric matrices (which are normal) have real eigenvalues and eigenvectors and are diagonalizable. If a matrix is normal but not symmetric then in general its

[2]This theorem asserts that for any square matrix A we can find a unitary matrix U (that is $U^* = U^{-1}$) such that $A = UTU^{-1}$, where T is an upper-triangular matrix and where U^* stands for the conjugate transpose of U.

eigenvalues and eigenvectors are complex but they can have zero imaginary part (not only symmetric matrices have real eigenvalues). Non normal matrices are not diagonalizable in the sense we have defined but can be diagonalized by blocks through the Jordan canonical form. In this book we will restrict ourselves to the case of normal matrices with real eigenvalues, that is, to the case in which the eigenvectors of A spans the real vector space $\mathbb{R}^n$.

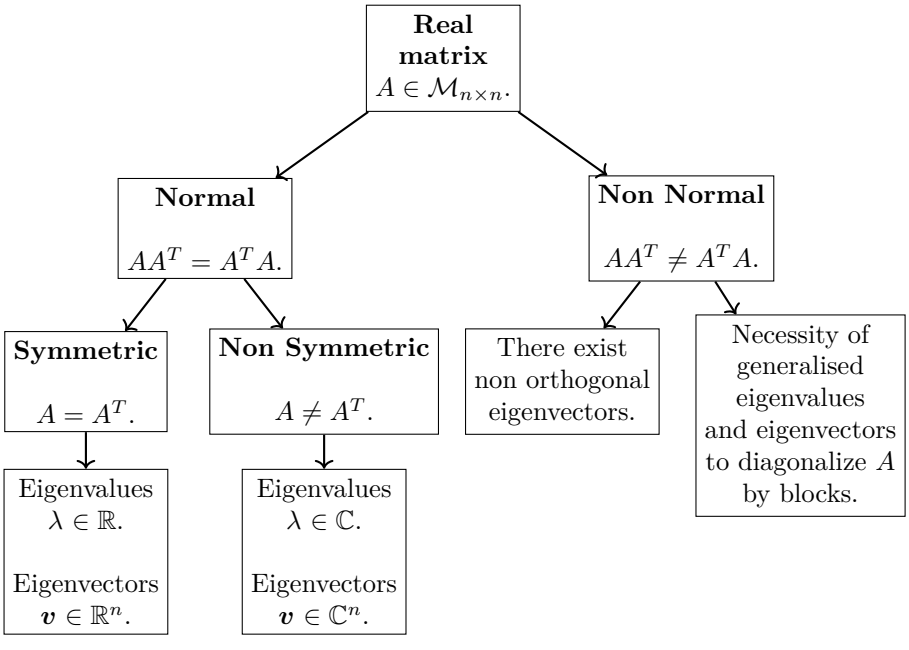

Figure 1.1: Spectral classification of real matrices.

1.5 Power method and deflation method

In this section we will present an iterative method to compute the eigenvalues and eigenvectors of normal matrices whose spectrum spans the whole real vector space $\mathbb{R}^n$. The power method is an iterative method which gives back the module of the maximum eigenvalue. Let $A \in \mathcal{M}_{n\times n}$ be a square real normal matrix with real eigenvalues $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$, and their orthonormal associated eigenvectors $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. The method is based on the fact that as the eigenvectors of A form a basis of $\mathbb{R}^n$ we can write for any vector $\boldsymbol{x}_0 \in \mathbb{R}^n$:

$$\boldsymbol{x}_0 = \sum_i a^i \boldsymbol{v}_i. \tag{1.20}$$

From (1.20) and the definition of eigenvectors we can compute:

$$A^k \boldsymbol{x}_0 = \sum_i a^i \lambda_i^k \boldsymbol{v}_i = \lambda_1^k \left(a^1 \boldsymbol{v}_1 + \sum_{i \neq 1} a^i \frac{\lambda_i^k}{\lambda_1^k} \boldsymbol{v}_i \right),$$

therefore we have that:

$$\frac{A^k \boldsymbol{x}_0}{\|A^k \boldsymbol{x}_0\|} = \left(a^1 \boldsymbol{v}_1 + \sum_{i \neq 1} a^i \frac{\lambda_i^k}{\lambda_1^k} \boldsymbol{v}_i \right) \left\| a^1 \boldsymbol{v}_1 + \sum_{i \neq 1} a^i \frac{\lambda_i^k}{\lambda_1^k} \boldsymbol{v}_i \right\|^{-1}. \tag{1.21}$$

Thus, if we define $\boldsymbol{x}_k = A^k \boldsymbol{x}_0 / \|\boldsymbol{x}_0\|$ we get a recursion

$$\boldsymbol{x}_{k+1} = \frac{A \boldsymbol{x}_k}{\|\boldsymbol{x}_k\|} = \frac{A^k \boldsymbol{x}_0}{\|A^k \boldsymbol{x}_0\|}, \tag{1.22}$$

which as $|\lambda_1| > |\lambda_i|$ for all $i > 1$ and taking in account (1.21) verifies:

$$\lim_{k \to \infty} \boldsymbol{x}_k = \boldsymbol{v}_1. \tag{1.23}$$

Hence, by iterating over (1.22) we can obtain an approximation of the eigenvector associated to the maximum module eigenvalue $\boldsymbol{v}_1$. The process of approximating the eigenvector is stopped using a convergence criteria: when $\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| \leq \epsilon$, where ϵ is a sufficiently small positive number, the iteration process stops.

Once we have computed this eigenvector we can obtain the associated eigenvalue λ_1 from the Rayleigh quotient as[3]:

$$\lim_{k \to \infty} \boldsymbol{x}_k \cdot A \boldsymbol{x}_k = \boldsymbol{v}_1 \cdot A \boldsymbol{v}_1 = \lambda_1. \tag{1.24}$$

The algorithm that carries out the power method can be summarized in three steps

1. **Initial condition**: We can set $\boldsymbol{x}_0$ to be any vector, for example its components can be the natural numbers $1, 2, \ldots, n$:

```
U = [ (k, k=1, N) ]
```

Listing 1.14: `Linear_systems.f90`

[3]Note that we have used that $\|\boldsymbol{x}_k\| \to 1$ when $k \to \infty$.

2. **Eigenvector calculation**: The eigenvector is calculated using recursion (1.22) as:

$$\boldsymbol{v} = A\boldsymbol{x}_k/\|\boldsymbol{x}_k\|, \tag{1.25}$$

$$\boldsymbol{u} = \boldsymbol{v}/\|\boldsymbol{v}\|, \tag{1.26}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{u} \tag{1.27}$$

which has an straightforward implementation in a conditional loop. For each iteration `V` stores $\boldsymbol{v} = A\boldsymbol{x}_k/\|\boldsymbol{x}_k\|$ and the approximation $\boldsymbol{u} = \boldsymbol{x}_{k+1}/\|\boldsymbol{x}_{k+1}\|$ is stored in `U` which is finally assigned to. The vector `U0` is used to define the convergence criteria for $\epsilon = 10^{-12}$. To avoid overflows in case that the process does not converge a number maximum of 10000 iterations is set.

```
do while( norm2(U-U0) > 1d-12 .and. k < k_max )
     U0 = U
     V = matmul( A, U )
     U = V / norm2(V)
     k = k + 1
end do
```

Listing 1.15: `Linear_systems.f90`

3. **Eigenvalue calculation**: Once we have computed the approximate eigenvector $\boldsymbol{x}_k \simeq \boldsymbol{v}_1$ and is stored on `U`, the eigenvalue is calculated taking in account equation (1.23) as:

$$\lambda_1 \simeq \boldsymbol{x}_k \cdot A\boldsymbol{x}_k. \tag{1.28}$$

The implementation of this calculation is immediate and the result is stored on `lambda`:

```
lambda = dot_product( U, matmul(A, U) )
```

Listing 1.16: `Linear_systems.f90`

All the previous steps are implemented in the subroutine `Power_method` which takes the matrix `A` and gives back the eigenvalue `lambda` and the eigenvector `U`.

```
subroutine Power_method(A, lambda, U)
      real, intent(in) :: A(:,:)
      real, intent(out) :: lambda, U(:)

  integer :: N, k, k_max = 10000
  real, allocatable :: U0(:), V(:)

    N = size( A, dim=1)
    allocate( U0(N), V(N) )
    U = [ (k, k=1, N) ]

    k = 1
    do while( norm2(U-U0) > 1d-12 .and. k < k_max )
         U0 = U
         V = matmul( A, U )
         U = V / norm2(V)
         k = k + 1
    end do

    lambda = dot_product( U, matmul(A, U) )


end subroutine
```

Listing 1.17: `Linear_systems.f90`

This subroutine, given a normal matrix A gives back its maximum module eigenvalue λ_1 and its associated eigenvector $\boldsymbol{v}_1$. The eigenvalue is yielded on the real `lambda` and the eigenvector in the vector `U`.

Once we have presented the power method iteration to compute the dominant eigenvalue λ_1 and its associated eigenvector $\boldsymbol{v}_1$, a method to compute all the eigenvalues and eigenvectors of a matrix using power method is presented. This iterative method is call deflation method. Note that for the power method to work properly we need $|\lambda_1|$ to be strictly greater than the rest of eigenvalues, but if $|\lambda_1| = |\lambda_2|$ the method works fine. Deflation method requires a stronger condition which is that the eigenvalues satisfy $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. The method is based on the fact that as the matrix $B_2 = A - \lambda_1 \boldsymbol{v}_1 \otimes \boldsymbol{v}_1$ replaces the eigenvalue λ_1 for an eigenvalue of zero value. The symbol $\otimes$ stands for the tensor product in $\mathbb{R}^n$ which is defined from the contraction $\boldsymbol{a} \otimes \boldsymbol{b} \cdot \boldsymbol{c} = \boldsymbol{a}(\boldsymbol{b} \cdot \boldsymbol{c})$. When this is done λ_1 is replaced, but the rest of the eigenvalues and eigenvectors remain invariant and therefore the dominant eigenvector of B_2 is λ_2. This is a consequence of the spectral theorem, which asserts that A can be written as:

$$A = \sum_i \lambda_i \boldsymbol{v}_i \otimes \boldsymbol{v}_i, \tag{1.29}$$

and therefore we can write B_2 as

$$B_2 = \boldsymbol{v}_1 \otimes \boldsymbol{v}_1 + \sum_{i \neq 1} \lambda_i \boldsymbol{v}_i \otimes \boldsymbol{v}_i, \tag{1.30}$$

where we see explicitly how the eigenvalue is replaced and the rest remain unaltered. Hence, if we define a succession of matrices

$$B_{k+1} = B_k - \lambda_k \boldsymbol{v}_k \otimes \boldsymbol{v}_k, \qquad \text{for} \quad k = 1, \ldots, n-1, \tag{1.31}$$

which starts at the value $B_1 = A$, for each B_k, the dominant eigenvalue is λ_k. Therefore, if we use the power method for each B_k we can compute both λ_k and $\boldsymbol{v}_k$. Thus, the algorithm can be summarized in two steps for which we consider B_k as the initial matrix:

1. **Power method over the initial matrix**: First we apply the power method to B_k and compute λ_k and $\boldsymbol{v}_k$. This is implemented by a simple call to the subroutine `Power_method` where the array `A` stores the entries of B_k.

```
call Power_method(A, lambda(k), U(:, k) )
```

Listing 1.18: `Linear_systems.f90`

2. **Next step matrix:** Once we have λ_k and $\boldsymbol{v}_k$ stored over `lambda(k)` and `U(:,k)` respectively, we can obtain B_{k+1} by simply applying formula (1.31) and storing its result on `A`:

```
A = A - lambda(k) * Tensor_product( U(:, k), U(:, k) )
```

Listing 1.19: `Linear_systems.f90`

To sweep k through the values $1, \ldots, n$, we contain both steps of the algorithm in a loop. Thus, a subroutine named `Eigenvalues` is implemented to carry out the deflation method. Given a square matrix `A` it gives back the scalar `lambda` and the square matrix `U`, whose columns are the eigenvectors of `A`:

```
subroutine Eigenvalues_PM(A, lambda, U)
      real, intent(inout) :: A(:,:)
      real, intent(out) :: lambda(:), U(:,:)

      integer :: i, j, k, N

      N = size(A, dim=1)

      do k=1, N

          call Power_method(A, lambda(k), U(:, k) )

          A = A - lambda(k) * Tensor_product( U(:, k), U(:, k) )

      end do

end subroutine
```

Listing 1.20: `Linear_systems.f90`

1.6 Inverse power method

For non singular matrices, a method which gives the eigenvalue of lesser module $|\lambda_n|$ and its associated eigenvector $\boldsymbol{v}_n$ is presented. If A is non singular, we can premultiply (1.17) by its inverse A^{-1} obtaining:

$$A^{-1}\boldsymbol{v} = \lambda^{-1}\boldsymbol{v}. \tag{1.32}$$

Therefore, we can extract two conclusions, the first is that A and A^{-1} have the same eigenvectors and the second is that their eigenvalues are inversely proportional to each other. This means that if μ is an eigenvalue of A^{-1} with eigenvector $\boldsymbol{v}$, it satisfies $\mu = \lambda^{-1}$. Therefore if the eigenvalues of A^{-1} satisfy $|\mu_n| > |\mu_{n-1}| \geq \cdots \geq |\mu_1|$, we have that the dominant eigenvalue of A^{-1} is related to the eigenvalue of A of minimum module λ_n. Hence, if we apply the power method to A^{-1} we get λ_n^{-1} and $\boldsymbol{v}_n$. This method is known as inverse power method for obvious reasons and the recursion for it is obtained substituting A by A^{-1} in (1.22), leading to:

$$\boldsymbol{x}_{k+1} = \frac{A^{-1}\boldsymbol{x}_k}{\|\boldsymbol{x}_k\|},$$

or equivalently:

$$A\boldsymbol{x}_{k+1} = \frac{\boldsymbol{x}_k}{\|\boldsymbol{x}_k\|}, \tag{1.33}$$

and for each iteration we solve the system (1.33). The algorithm that carries out the inverse power method is summarized in four steps:

1. **LU factorization of A:** Prior to solve the system of the recursion, we factorize A by a simple call for the array that will store the lower and upper matrices of the LU factorization, which is named `Ac`:

```
call LU_factorization(Ac)
```

Listing 1.21: `Linear_systems.f90`

2. **Initial condition**: We can set $\boldsymbol{x}_0$ to be any vector, for example its components can be the natural numbers $1, 2, \ldots, n$:

```
U = [ (k, k=1, N) ]
```

Listing 1.22: `Linear_systems.f90`

3. **Eigenvector calculation**: The eigenvector is calculated solving recursion (1.33) by LU factorization. First, the matrix `Ac` is factorized

$$\boldsymbol{v} = A^{-1}\boldsymbol{x}_k/\|\boldsymbol{x}_k\|, \tag{1.34}$$

$$\boldsymbol{u} = \boldsymbol{v}/\|\boldsymbol{v}\|, \tag{1.35}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{u} \tag{1.36}$$

which has an straightforward implementation in a conditional loop. For each iteration `V` stores $\boldsymbol{v} = A^{-1}\boldsymbol{x}_k/\|\boldsymbol{x}_k\|$ and the approximation $\boldsymbol{u} = \boldsymbol{x}_{k+1}/\|\boldsymbol{x}_{k+1}\|$ is stored in `U` which is finally assigned to. The vector `U0` is used to define the convergence criteria for $\epsilon = 10^{-12}$. To avoid overflows in case that the process does not converge a number maximum of 10000 iterations is set. The only change of this step with respect to the algorithm for power method is that now `V` stores the value that comes out of solving a LU system:

```
        V = solve_LU(Ac, U)
```

Listing 1.23: `Linear_systems.f90`

4. **Eigenvalue calculation**: Once we have computed the approximate eigenvector $\boldsymbol{x}_k \simeq \boldsymbol{v}_n$ and is stored on `U`, the eigenvalue is calculated taking in account equation (1.23) as:

$$\lambda_n \simeq \boldsymbol{x}_k \cdot A\boldsymbol{x}_k, \tag{1.37}$$

where we have used that A and A^{-1} share eigenvectors. The implementation of this calculation is immediate and the result is stored on `lambda`:

```
    lambda = dot_product( U, matmul(A, U) )
```

Listing 1.24: `Linear_systems.f90`

The algorithm of the inverse power method is implemented in the subroutine `Inverse_power_method`. This subroutine, given a normal matrix `A` whose minimum module eigenvalue λ_n is strictly lesser than the rest of eigenvalues, gives λ_n and its associated eigenvector $\boldsymbol{v}_n$. The eigenvalue is stored on the real `lambda` and the eigenvector in the vector `U`.

```
subroutine Inverse_power_method(A, lambda, U)
      real, intent(inout) :: A(:,:)
      real, intent(out) :: lambda, U(:)

  integer :: N, k, k_max = 10000
  real, allocatable :: U0(:), V(:), Ac(:, :)


    N = size(U)
    allocate ( Ac(N,N), U0(N), V(N) )

    Ac = A
    call LU_factorization(Ac)
    U = [ (k, k=1, N) ]

    k = 1
    do while( norm2(U-U0) > 1d-12 .and. k < k_max )

         U0 = U
         V = solve_LU(Ac, U)
         U = V / norm2(V)
         k = k + 1
    end do

    lambda = norm2(matmul(A, U))


end subroutine
```

Listing 1.25: `Linear_systems.f90`

1.7 SVD decomposition

Once we have seen how we can compute real eigenvalues and eigenvectors of normal matrices whose eigenvalues are strictly ordered, we can speak of probably the most important matrix factorization. Singular Value Decomposition or SVD is a factorization applicable to any real matrix $A \in \mathcal{M}_{m\times n}$ and that provides all the information about the fundamental sub-spaces of the matrix. Let's recall that the four fundamental sub-spaces of A are its image $\operatorname{Im} A \subset \mathbb{R}^m$ and kernel $\ker A \subset \mathbb{R}^n$ and the image and kernel of its transpose $\operatorname{Im} A^T \subset \mathbb{R}^n$ and $\ker A^T \subset \mathbb{R}^m$. The singular value decomposition is a factorization such that:

$$A = U^T \Sigma V, \tag{1.38}$$

where $U \in \mathcal{M}_{m\times m}$ and $V \in \mathcal{M}_{m\times n}$ are orthogonal matrices and $\Sigma \in \mathcal{M}_{m\times n}$ is a matrix containing the singular values of A in its principal diagonal. For the moment, that is all we will say about the matrices involved, and what is a singular value will be explained in the following lines. We will see that in U is contained the information about $\operatorname{Im} A$ and $\ker A^T$. On other hand, we will see how is contained all the information about $\operatorname{Im} A^T$ and $\ker A$ in V. For the moment, let's begin with some previous definitions which will help the reader to understand the importance of SVD.

Let $A \in \mathcal{M}_{m\times n}$ a real matrix, it is easy to check that the matrix $A^T A \in \mathcal{M}_{n\times n}$ is symmetric (and therefore normal) and semi-definite positive. The symmetry is immediate taking into account the rule of the transpose of a product, to prove that is semi-definite positive we have to check that $\boldsymbol{x} \cdot A^T A\boldsymbol{x} \geq$ for any $\boldsymbol{x} \in \mathbb{R}^n$. This is done as follows:

$$\|A\boldsymbol{x}\|^2 = A\boldsymbol{x} \cdot A\boldsymbol{x} = \boldsymbol{x} \cdot A^T A\boldsymbol{x} \geq 0, \tag{1.39}$$

where the equality holds if $\boldsymbol{x} \in \ker A$ or $\boldsymbol{x} \in \ker A^T A$.

That $A^T A$ is semi-definite positive makes that all of its eigenvalues are not only real but positive or zero. If $\{\lambda_1, \ldots, \lambda_n\}$ and $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ are the eigenvalues and associated orthonormal eigenvectors of $A^T A$ respectively, we can write:

$$\|A\boldsymbol{v}_i\|^2 = \boldsymbol{v}_i \cdot A^T A\boldsymbol{v}_i = \lambda_i,$$

and therefore we conclude that $\lambda_i \geq 0$ and we define the singular values of A as:

$$\sigma_i = \sqrt{\lambda_i}, \qquad \text{for} \quad i = 1, \ldots, n. \tag{1.40}$$

And now we know that the entries of the main diagonal of Σ are just the square roots of the eigenvalues of $A^T A$. The answer to why (1.38) is a correct factorization of A and what are the explicit expressions of U and V requires to state some useful

facts. Let's suppose without loss of generality that only r of the eigenvalues of A^TA are non zero and that they are ordered as:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r > \lambda_{r+1} = \lambda_{r+2} = \cdots = \lambda_n = 0. \tag{1.41}$$

In first place we must note that the images of the eigenvectors of A^TA by A are an orthogonal set of vectors. That is, the set $\{A\boldsymbol{v}_1, \ldots, A\boldsymbol{v}_r\} \subset \mathbb{R}^m$ is an orthogonal set. To check this we take into account that the eigenvectors of A^TA form an orthonormal set of $\mathbb{R}^n$ and we can write:

$$A\boldsymbol{v}_i \cdot A\boldsymbol{v}_j = \boldsymbol{v}_i \cdot A^TA\boldsymbol{v}_j = \lambda_j \boldsymbol{v}_i \cdot \boldsymbol{v}_j = \lambda_j \delta_{ij}, \tag{1.42}$$

and is clear that for $i = j$ we get $\|A\boldsymbol{v}_i\| = \sigma_i$. Hence, if we name:

$$\boldsymbol{u}_i = \frac{A\boldsymbol{v}_i}{\sigma_i}, \qquad \text{for} \quad i = 1, \ldots, r, \tag{1.43}$$

is immediate that $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\} \subset \mathbb{R}^m$is an orthonormal set. To prove that (1.38) is correct, we just have to write (1.43) in a different manner, extending it to $i > r$ as:

$$\begin{cases} A\boldsymbol{v}_1 = \boldsymbol{u}_1\sigma_1, \\ \quad \vdots \\ A\boldsymbol{v}_r = \boldsymbol{u}_r\sigma_r \\ A\boldsymbol{v}_{r+1} = \boldsymbol{0} \\ \quad \vdots \\ A\boldsymbol{v}_n = \boldsymbol{0} \end{cases}, \tag{1.44}$$

and defining the orthogonal matrices:

$$V = \left[\begin{array}{c|c|c} \boldsymbol{v}_1 & \cdots & \boldsymbol{v}_n \end{array} \right] \in \mathcal{M}_{n\times n}, , \tag{1.45}$$

$$U = \left[\begin{array}{c|c|c} \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_m \end{array} \right] \in \mathcal{M}_{m\times m}, \tag{1.46}$$

where if $r < m$, we can always compute the vectors $\boldsymbol{u}_i$ with $i > m$ as orthogonal to each other and to the set $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$, therefore both V and U are orthogonal. With the definition above, we can rewrite (1.44) in matrix form as:

$$AV = U\Sigma,$$

and as V is orthogonal $VV^T = I$, the equation above is equivalent to (1.38).

Now we have seen that such a factorization is possible let's see what information of A is provided by V and U. In first place we have to notice that $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$ is a basis for $\operatorname{Im} A$, to see this let's pick a generic element $\boldsymbol{y} \in \operatorname{Im} A$, that is $\boldsymbol{y} = A\boldsymbol{x}$

for some $\boldsymbol{x} \in \mathbb{R}^n$ and taking into account that the eigenvectors of A^TA span $\mathbb{R}^n$ we can project $\boldsymbol{y}$ over any $A\boldsymbol{v}_i$ for $i = 1, \ldots, n$ as:

$$\boldsymbol{y} \cdot A\boldsymbol{v}_i = A\boldsymbol{x} \cdot A\boldsymbol{v}_i = \boldsymbol{x} \cdot A^TA\boldsymbol{v}_i = \sum_j x_j\boldsymbol{v}_j \cdot A^TA\boldsymbol{v}_i = x_i\lambda_i,$$

Note that for $i > r$ we have $\boldsymbol{y} \cdot A\boldsymbol{v}_i = 0$, which means that $A\boldsymbol{v}_i$ is perpendicular to any element of the image. This perpendicularity implies that the subspace spanned by $\{A\boldsymbol{v}_{r+1}, \ldots, A\boldsymbol{v}_n\}$ is orthogonal to $\operatorname{Im} A$. This implies that if we project $\boldsymbol{y}$ onto the set $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m\}$ (which is a basis of $\mathbb{R}^m$) we have:

$$\boldsymbol{y} = \sum_i \boldsymbol{y} \cdot \boldsymbol{u}_i\boldsymbol{y} = \sum_{i=1}^{r} \boldsymbol{y} \cdot \boldsymbol{u}_i\boldsymbol{y},$$

and therefore $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$ an orthonormal basis of $\operatorname{Im} A$.

The second realization to do is that the set $\{\boldsymbol{v}_{r+1}, \ldots, \boldsymbol{v}_n\}$ forms an orthonormal basis for $\ker A$. For this we first check that $\left\|A^TA\boldsymbol{x}\right\| = 0$ if and only if

$$\boldsymbol{x} = \sum_{i=r+1}^{n} x_i\boldsymbol{v}_i, \tag{1.47}$$

which means that $\{\boldsymbol{v}_{r+1}, \ldots, \boldsymbol{v}_n\}$ forms a basis for $\ker A^TA$. From (1.39) we deduce that if $\boldsymbol{x} \in \ker A^TA$ then $\boldsymbol{x} \in \ker A$ ($\ker A^TA \subset \ker A$). Conversely, we have that if $\boldsymbol{x} \in \ker A$

$$0 \leq \left\|A^TA\boldsymbol{x}\right\| \leq \left\|A^T\right\|\|A\boldsymbol{x}\| = 0, \tag{1.48}$$

where $\left\|A^T\right\|$ stands for the induced norm for matrices from the norm in the vector-space:

$$\left\|A^T\right\| = \sup \frac{\left\|A^T\boldsymbol{y}\right\|}{\|\boldsymbol{y}\|}, \qquad \forall\ \boldsymbol{y} \neq \boldsymbol{0} \in \mathbb{R}^m,$$

and from (1.48) we have that $\ker A^TA = \ker A$. Hence, the set $\{\boldsymbol{v}_{r+1}, \ldots, \boldsymbol{v}_n\}$ is a basis for $\ker A$. This implies that these sets of vectors can be used to define projection matrices (matrices whose image is always in the subspace onto which they project). A dual argument will serve to prove that the remaining vectors of the two sets of orthonormal vectors also serve as basis for the remaining fundamental sub-spaces of A. Hence, if we define the reduced matrices:

$$V_{n-r} = \left[\ \boldsymbol{v}_{r+1} \ \middle|\ \cdots \ \middle|\ \boldsymbol{v}_n\ \right] \in \mathcal{M}_{n\times r}, , \tag{1.49}$$

$$U_r = \left[\ \boldsymbol{u}_1 \ \middle|\ \cdots \ \middle|\ \boldsymbol{u}_r\ \right] \in \mathcal{M}_{m\times r}, \tag{1.50}$$

We have that the projection matrices onto $\ker A$ and $\operatorname{Im} A$ are respectively.

$$P_{\ker A} = V_{n-r}V_{n-r}^T, \tag{1.51}$$

$$P_{\operatorname{Im} A} = U_rU_r^T. \tag{1.52}$$

As $\ker A \perp \operatorname{Im} A^T$ and $\operatorname{Im} A \perp \ker A^T$ we have the projection matrices.

$$P_{\ker A^T} = I - U_r U_r^T = U_{m-r} U_{m-r}^T, \tag{1.53}$$

$$P_{\operatorname{Im} A^T} = V_r V_r^T = I - V_{n-r} V_{n-r}^T, \tag{1.54}$$

where

$$U_{m-r} = \left[\begin{array}{c|c|c} \boldsymbol{u}_{r+1} & \cdots & \boldsymbol{u}_m \end{array} \right] \in \mathcal{M}_{m\times r},, \tag{1.55}$$

$$V_r = \left[\begin{array}{c|c|c} \boldsymbol{v}_1 & \cdots & \boldsymbol{v}_r \end{array} \right] \in \mathcal{M}_{n\times r}. \tag{1.56}$$

Thus, the reader can have an idea of the importance of SVD factorization as once is done, it provides all the information about the four fundamental sub-spaces condensed on U and V. Besides, the rank of Σ is the rank of A.

For square matrices the SVD is computed in two steps:

1. **Eigenvalues and eigenvectors of** $A^T A$: In first place we compute the eigenvalues of $A^T A$ and their associated singular values. As is symmetric we can use the `Eigenvalues` subroutine previously presented.

```
B = matmul( transpose(A), A )
call Eigenvalues_PM( B, sigma, V )
sigma = sqrt(sigma)
```

Listing 1.26: `Linear_systems.f90`

2. **Calculation of** U: With the eigenvectors of $A^T A$ and the singular values of A we can compute the i-th column of U as given by (1.43):

```
U(:,i) = matmul( A, V(:, i) ) / sigma(i)
```

Listing 1.27: `Linear_systems.f90`

The whole precess is embedded in the subroutine `SVD` which takes `A` as input and gives back the singular values, U and V respectively on the arrays `sigma`, `U` and `V`.

```
subroutine SVD(A, sigma, U, V)
      real, intent(in) :: A(:,:)
      real, intent(out) :: sigma(:), U(:,:), V(:,:)

      integer :: i, N
      real, allocatable :: B(:,:)

      N = size(A, dim=1)

      B = matmul( transpose(A), A )
      call Eigenvalues_PM( B, sigma, V )
      sigma = sqrt(sigma)

      do i=1, N

          if ( abs(sigma(i)) > 1d-10 ) then
                U(:,i) = matmul( A, V(:, i) ) / sigma(i)
          else
                write(*,*) " Singular value is zero"
                stop
          end if

      end do


end subroutine
```

Listing 1.28: `Linear_systems.f90`

1.8 Condition number

When solving a linear system of equations, the round–off error of the solution is associated to the condition number of the matrix system. In order to understand the motivation of this concept, let us be a linear system of equations such as:

$$A\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{x}, \boldsymbol{b}$ are vectors from a vector space V, equipped with the norm $\|\cdot\|$ and A is a square matrix.

If an induced norm is defined for matrices the previous equation can give us a measurable relation from the system of equations. In these conditions, the following order relation is satisfied:

$$\|\boldsymbol{b}\| \leq \|A\|\|\boldsymbol{x}\|.$$

Given the linearity of the system, if the vector $\boldsymbol{b}$ is perturbed with a perturbation $\delta\boldsymbol{b}$, the solution will be as well with the perturbation $\delta\boldsymbol{x}$ and if A is non singular, the following order relation is satisfied:

$$\|\delta\boldsymbol{x}\| \leq \|A^{-1}\| \, \|\delta\boldsymbol{b}\|.$$

Combining both order relations it is obtained an upper bound for the relative perturbation of the solution, that is:

$$\frac{\|\delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \|A\|\|A^{-1}\| \, \frac{\|\delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|},$$

where $\|A\|\|A^{-1}\|$ determines the upper bound of the perturbation in the solution. The condition number $\kappa(A)$ for this linear system can be written:

$$\kappa(A) = \|A\|\|A^{-1}\| \, .$$

Whenever the norm defined for V is the quadratic norm $\|\cdot\|_2$, the condition number can be written in terms of the square roots of the maximum and minimum module eigenvalues of AA^T, $\sigma_{\max}$ and $\sigma_{\min}$:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}},$$

as $\|A\| = \sigma_{\max}$ and $\|A^{-1}\| = 1/\sigma_{\min}$.

Hence, the condition number is intrinsically related to the disturbance of the solution and determines if a matrix A is *well-conditioned* if $\kappa(A)$ is small or *ill-conditioned* if $\kappa(A)$ is large.

To implement the condition number computation for a matrix we follow three steps.

1. **Maximum singular value of A^TA:** This is done calling `Power_method` using as input a matrix `B` that stores A^TA and computing the square root of the resulting eigenvalue storing it in `sigma_max`

2. **Minimum eigenvalue of A^TA:** This is done calling `Power_method` using as input `B` and computing the square root of the resulting eigenvalue storing it in `sigma_min`

3. **Condition number of A:** The condition number is calculated for the output of the function `Condition_number` doing the ratio `sigma_max/sigma_min`

```
real function Condition_number(A)
      real, intent(in) :: A(:,:)


      integer :: i, j, k, N
      real, allocatable :: B(:,:), U(:)
      real :: sigma_max, sigma_min, lambda

      N = size(A, dim=1)
      allocate( U(N), B(N,N) )
      B = matmul( transpose(A), A )


     call Power_method( B,  lambda, U )
     sigma_max = sqrt(lambda)

     call Inverse_power_method( B,  lambda, U )
     sigma_min = sqrt(lambda)

     Condition_number = sigma_max / sigma_min


end function
```

Listing 1.29: `Linear_systems.f90`

Chapter 2

Lagrange Interpolation

2.1 Overview

One of the core topics in theory of approximation is the interpolation of functions. The main idea of interpolation is to approximate a function $f(x)$ in an interval $[x_0, x_f]$ by means of a set of known functions $\{g_j(x)\}$ which are intended to be simpler than $f(x)$. The set $\{g_j(x)\}$ can be polynomials, monomials or trigonometric functions. In this chapter we will restrict ourselves to the case of polynomial interpolation, and in particular when Lagrange polynomials are used.

2.2 Lagrange interpolation

In this section, polynomial interpolation using Lagrange polynomials will be presented. For this a general view on how to use Lagrange polynomials is explained, considering different scenarios. First how to calculate recursively Lagrange polynomials, their derivatives and integral will be explained. After that, once we have available the derivatives and integral of these polynomials, how to use them to approximate this quantities for a function $f(x)$. At the same time, an implementation of the discussed procedures shall be presented. To end the chapter, we present briefly how to extend the notion of Lagrange interpolation to functions of more than one variable considering the case of a function of two variables.

2.2.1 Lagrange polynomials

The Lagrange polynomials $\ell_j(x)$ of grade N for a set of points $\{x_j\}$ for $j = 0, 1, 2 \ldots, N$ are defined as:

$$\ell_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x - x_i}{x_j - x_i}, \tag{2.1}$$

which satisfy:

$$\ell_j(x_i) = \delta_{ij}, \tag{2.2}$$

where δ_{ij} is the delta Kronecker function. This property is fundamental because as we will see it permits to obtain the Lagrange interpolant very easily once the Lagrange polynomials are determined.

Another interesting property specially for recursively determining Lagrange polynomials appears when considering ℓ_{jk} as the Lagrange polynomial of grade k at x_j for the set of nodes $\{x_0, x_1 \ldots, x_k\}$ and $0 \leq j \leq k$, that is:

$$\ell_{jk}(x) = \prod_{\substack{i=0 \\ i \neq j}}^{k} \frac{x - x_i}{x_j - x_i} = \left(\frac{x - x_{k-1}}{x_j - x_{k-1}} \right) \prod_{\substack{i=0 \\ i \neq j}}^{k-1} \frac{x - x_i}{x_j - x_i}, \tag{2.3}$$

which results in the property:

$$\ell_{jk}(x) = \ell_{jk-1}(x) \left(\frac{x - x_{k-1}}{x_j - x_{k-1}} \right), \tag{2.4}$$

where $1 \leq k \leq N$. This property permits to obtain the Lagrange interpolant of grade k for the set of points $\{x_0, x_1 \ldots, x_k\}$ if its known the interpolant of grade $k-1$ for the set of points $\{x_0, x_1 \ldots, x_{k-1}\}$. Besides, for $k = 0$ is satisfied:

$$\ell_{j0}(x) = 1. \tag{2.5}$$

Hence, it can be obtained recursively the interpolant at x_j for each grade as:

$$
\begin{aligned}
\ell_{j1}(x) &= \left(\frac{x-x_0}{x_j-x_0}\right),\\
\ell_{j2}(x) &= \left(\frac{x-x_0}{x_j-x_0}\right)\left(\frac{x-x_1}{x_j-x_1}\right),\\
&\vdots\\
\ell_{jk}(x) &= \underbrace{\left(\frac{x-x_0}{x_j-x_0}\right)\ldots\left(\frac{x-x_{j-1}}{x_j-x_{j-1}}\right)\left(\frac{x-x_{j+1}}{x_j-x_{j+1}}\right)\ldots\left(\frac{x-x_{k-2}}{x_j-x_{k-2}}\right)}_{\ell_{jk-1}(x)}\left(\frac{x-x_{k-1}}{x_j-x_{k-1}}\right).
\end{aligned}
$$

From equation (2.4). a recursion to calculate the k first derivatives of $l_{jk}(x)$ is obtained.

$$
\begin{aligned}
\ell'_{jk}(x) &= \ell'_{jk-1}(x)\left(\frac{x-x_{k-1}}{x_j-x_{k-1}}\right)+\left(\frac{\ell_{jk-1}(x)}{x_j-x_{k-1}}\right),\\
\ell''_{jk}(x) &= \ell''_{jk-1}(x)\left(\frac{x-x_{k-1}}{x_j-x_{k-1}}\right)+\left(\frac{2\ell'_{jk-1}(x)}{x_j-x_{k-1}}\right),\\
&\vdots\\
\ell^{(m)}_{jk}(x) &= \ell^{(m)}_{jk-1}(x)\left(\frac{x-x_{k-1}}{x_j-x_{k-1}}\right)+\left(\frac{m\ell^{(m-1)}_{jk-1}(x)}{x_j-x_{k-1}}\right), \qquad (2.6)\\
&\vdots\\
\ell^{(k)}_{jk}(x) &= \cancelto{0}{\ell^{(k)}_{jk-1}(x)}\left(\frac{x-x_{k-1}}{x_j-x_{k-1}}\right)+\left(\frac{k\ell^{(k-1)}_{jk-1}(x)}{x_j-x_{k-1}}\right).
\end{aligned}
$$

Note that equation (2.6) is also valid for $m = 0$, value for which it reduces to (2.4). Hence, starting from the value $\ell_{j0} = 1$ we can compute the polynomial ℓ_{jk} and its first k derivatives using recursion (2.6). The idea is known the first $k-1$ derivatives of ℓ_{jk-1} we start computing $\ell^{k)}_{jk}$, then $\ell^{k-1)}_{jk}$ until we calculate $\ell^{0)}_{jk} = \ell_{jk}$. For example, supposing $j \neq 0$ we calculate ℓ'_{j1} and ℓ_{j1} from $\ell_{j0} = 1$ as

$$
\begin{aligned}
\ell'_{j1} &= \frac{\ell_{j0}}{x_j-x_0} = \frac{1}{x_j-x_0},\\
\ell_{j1} &= \ell_{j0}\frac{x-x_0}{x_j-x_0} = \frac{x-x_0}{x_j-x_0}.
\end{aligned}
$$

The reason to sweep m in descending order through the interval $[0, k]$ is because computing the recursion in this manner permits to implement the calculation of the derivatives storing the values of $\ell^{(m)}_{jk}$ over the value of $\ell^{(m)}_{jk-1}$.

Once ℓ_{jk} and its k first derivatives are calculated we can compute the integral of ℓ_{jk} in the interval $[x_0, x]$ from its truncated Taylor series of grade k. Hence, we can express the integral as:

$$\int_{x_0}^{x} \ell_{jk}(x)\mathrm{d}x = \ell_{jk}(x_0)(x-x_0) + \ell'_{jk}(x_0)\frac{(x-x_0)^2}{2} + \cdots + \ell^{(k)}_{jk}(x_0)\frac{(x-x_0)^{k+1}}{(k+1)!}. \tag{2.7}$$

The computation of the first k derivatives and the integral for a grid of $k+1$ nodes, is carried out by the function `Lagrange_polynomials`. The derivatives and integrals at a point `xp` are stored on a vector `d` whose dimension is the number of set nodes. For a fixed point of the grid (that is, for fixed j) the following loop computes the derivatives and the image of the Lagrange polynomial ℓ_j, evaluated at `xp`.

```
 ! ** k derivative of lagrange(x) at xp
      do  r = 0, N

         if (r/=j) then
          do k = Nk, 0, -1
            d(k) = ( d(k) *( xp - x(r) ) + k * d(k-1) ) /( x(j) - x(r) )
          end do
        endif
```

Listing 2.1: `Lagrange_interpolation.f90`

The integral is computed in a different loop once the derivatives are calculated

```
!  ** integral of lagrange(x) form x(jp) to xp
      f = 1
      j1 = minloc( abs(x - xp) ) - 2
      jp = max(0, j1(1))

      do k=0, Nk
         f = f * ( k + 1 )
         d(-1) = d(-1) -  d(k) * ( x(jp) - xp )**(k+1) / f
      enddo
```

Listing 2.2: `Lagrange_interpolation.f90`

Both processes are carried out in the pure function `Lagrange_polynomials`, once both derivatives and integral are computed at a nodal point labeled by j the values stored at `d` are assigned to the output of the function. Then the same procedure is carried out for the next grid point $j+1$.

```
pure function Lagrange_polynomials( x, xp )
   real, intent(in) :: x(0:), xp
   real Lagrange_polynomials(-1:size(x)-1,0:size(x)-1)
```

```
   integer :: j   ! node
   integer :: r   ! recursive index
   integer :: k   ! derivative
   integer :: Nk  ! maximum order of the derivative
   integer :: N, j1(1), jp

   real :: d(-1:size(x)-1)
   real :: f

   Nk = size(x) - 1
   N  = size(x) - 1

   do j = 0, N
      d(-1:Nk) = 0
      d(0) = 1

 ! ** k derivative of lagrange(x) at xp
      do  r = 0, N

         if (r/=j) then
          do k = Nk, 0, -1
            d(k) = ( d(k) *( xp - x(r) ) + k * d(k-1) ) /( x(j) - x(r) )
          end do
         endif

      enddo

!  ** integral of lagrange(x) form x(jp) to xp
      f = 1
      j1 = minloc( abs(x - xp) ) - 2
      jp = max(0, j1(1))

      do k=0, Nk
         f = f * ( k + 1 )
         d(-1) = d(-1) -  d(k) * ( x(jp) - xp )**(k+1) / f
      enddo

      Lagrange_polynomials(-1:Nk, j ) = d(-1:Nk)

  end do

  end function
```

Listing 2.3: `Lagrange_interpolation.f90`

2.2.2 Single variable functions

Whenever is considered a single variable scalar function $f : \mathbb{R} \to \mathbb{R}$, whose value $f(x_j)$ at the nodes x_j for $j = 0, 1, 2 \ldots, N$ is known, the Lagrange interpolant $I(x)$ that approximates the function in the interval $[x_0, x_N]$ takes the form:

$$I(x) = \sum_{j=0}^{N} b_j \ell_j(x). \tag{2.8}$$

This interpolant is used to approximate the function $f(x)$ within the interval $[x_0, x_N]$. For this, the constants of the linear combination b_j must be determined. The interpolant must satisfy to intersect the exact function $f(x)$ on the nodal points x_i for $i = 0, 1, 2 \ldots, N$, that is:

$$I(x_i) = f(x_i). \tag{2.9}$$

Taking into account the property (2.2) leads to:

$$I(x_i) = f(x_i) = \sum_{j=0}^{N} b_j \ell_j(x_i) = \sum_{j=0}^{N} b_j \delta_{ij}, \quad \Rightarrow \quad f(x_j) = b_j. \tag{2.10}$$

Hence, the interpolant for $f(x)$ on the nodal points x_j for $j = 0, 1, 2 \ldots, N$ is written:

$$I(x) = \sum_{j=0}^{N} f(x_j) \ell_j(x). \tag{2.11}$$

Note that in the equation above the degree of ℓ_j does not necessarily need to be N and in general its degree q satisfies $q \leq N$.

A very common use of interpolation is to compute an approximation of f in a non-nodal point x_p. The implementation of the interpolation of f evaluated at x_p is carried out by the function `Interpolated_value` which given a set of nodes `x` and the image of the function at those points `y`, computes the interpolated value of f in `xp` using Lagrange interpolation of a certain degree. First, it checks whether the degree of the polynomial `degree` is even or odd as depending on this, the starting point of the set of nodes used by the Lagrange polynomials varies. Once the stencil is determined is necessary to compute the coefficients that multiply the images $f(x_j)$ which are the Lagrange polynomials evaluated at x_p. For this task, it calls the function `Lagrange_polynomials` and stores its output in the array `Weights`. Finally, we just need to sum the values of the Lagrange polynomials stored on `Weights` at each point, scaled by the nodal images `y`.

```
real pure function Interpolated_value(x, y, xp, degree)
    real, intent(in) :: x(0:), y(0:), xp
    integer, optional, intent(in) :: degree
    integer :: N, s, j
```

```
!   maximum order of derivative and width of the stencil
    integer :: Nk !

!   Lagrange coefficients and their derivatives at xp
    real, allocatable :: Weights(:,:)

    N = size(x) - 1


    if(present(degree))then
        Nk = degree
    else
        Nk = 2
    end if

    allocate( Weights(-1:Nk, 0:Nk))

    j = max(0, maxloc(x, 1, x < xp ) - 1)

    if( (j+1) <= N ) then !  the (j+1) cannot be accessed
        if( xp > (x(j) + x(j + 1))/2 ) then
            j = j + 1
        end if
    end if



    if (mod(Nk,2)==0) then
        s = max( 0, min(j-Nk/2, N-Nk) )   ! For Nk=2
    else
        s = max( 0, min(j-(Nk-1)/2, N-Nk) )
    endif


    Weights(-1:Nk, 0:Nk)  = Lagrange_polynomials( x = x(s:s+Nk), xp = xp )
    interpolated_value = sum ( Weights(0, 0:Nk) * y(s:s+Nk) )

    deallocate(Weights)

end function
```

Listing 2.4: `Interpolation.f90`

A different application of interpolation is to estimate the derivatives of the function f by calculating the derivatives of the interpolant I. Mathematically, the solution to the problem is straightforward once the interpolant has been constructed. The k-th derivative of the interpolant is written as:

$$I^{(k)}(x) = \sum_{j=0}^{N} f(x_j)\ell_j^{(k)}(x). \tag{2.12}$$

In many situations is required to compute the first k derivatives and image of the interpolant in a set of points $x_{p,i}$, $i = 0, \dots M$ contained in the interval $[x_0, x_N]$. Given the equation above, we just have to evaluate it in the set of nodes $x_{p,i}$, that is, we calculate them as:

$$I^{(k)}(x_{p,i}) = \sum_{j=0}^{N} f(x_j)\ell_j^{(k)}(x_{p,i}), \qquad i = 0, \dots M. \tag{2.13}$$

The implementation of this calculation is obtained in a similar manner as it was done to obtain the interpolated value in a single point. In fact, the function `Interpolant` is an extension of the function `Interpolated_value`. Note that in this new function, the degree of the interpolant used is also checked as the stencil used to compute the derivatives of the interpolant varies whenever the degree is odd or even. For each point $x_{p,i}$ we have to calculate the derivatives of the Lagrange polynomials. In other words, we have to summon the function `Lagrange_polynomials` to compute these derivatives at each point $x_{p,i}$. From the implementation point of view this requires embedding the process in a loop that goes from $i = 0$ to $i = M$. The derivatives $\ell_j^{(k)}(x_{p,i})$ are stored in the array `Weights` for a posterior usage. Once the Lagrange polynomials derivatives are computed, we just have to linearly combine the images $f(x_j)$ using the elements of the array `Weights` as coefficients.

The implementation of the function `Interpolant` is shown in the following listing:

```
function Interpolant(x, y, degree, xp )
    real, intent(in) :: x(0:), y(0:), xp(0:)
    integer, intent(in) :: degree
    real :: Interpolant(0:degree, 0:size(xp)-1)

    integer :: N, M, s, i, j, k

!   maximum order of derivative and width of the stencil
    integer :: Nk

!   Lagrange coefficients and their derivatives at xp
    real, allocatable :: Weights(:,:)

    N = size(x) - 1
    M = size(xp) - 1
    Nk = degree
    allocate( Weights(-1:Nk, 0:Nk))

do i=0, M

    j = max(0, maxloc(x, 1, x < xp(i) ) - 1)

    if( (j+1) <= N ) then ! the (j+1) cannot be accessed
        if( xp(i) > (x(j) + x(j + 1))/2 ) then
            j = j + 1
        end if
    end if


    if (mod(Nk,2)==0) then
        s = max( 0, min(j-Nk/2, N-Nk) )   ! For Nk=2
    else
        s = max( 0, min(j-(Nk-1)/2, N-Nk) )
    endif


    Weights(-1:Nk, 0:Nk)=Lagrange_polynomials( x = x(s:s+Nk), xp = xp(i) )

    do k=0, Nk
        Interpolant(k, i) = sum ( Weights(k, 0:Nk) * y(s:s+Nk) )
    end do

end do

    deallocate(Weights)

end function
```

Listing 2.5: `Interpolation.f90`

Interpolation serves also to approximate integrals in an interval $[x_0, x_N]$. Mathematically, the integral is computed from the interpolant as:

$$\int_{x_0}^{x_N} I(x)\mathrm{d}x = \sum_{j=0}^{N} f(x_j) \int_{x_0}^{x_N} \ell_j(x)\mathrm{d}x. \tag{2.14}$$

The implementation is done in a function called `Integral` given the set of nodes x_i, the images y_i, $i = 0, \dots N$ and optionally the degree of the polynomials used.

```
real function Integral(x, y, degree)
    real, intent(in) :: x(0:), y(0:)
    integer, optional, intent(in) :: degree

    integer :: N, j, s
    real :: summation, Int, xp

!   maximum order of derivative and width of the stencil
    integer :: Nk

 !  Lagrange coefficients and their derivatives at xp
    real, allocatable :: Weights(:,:,:)

    N = size(x) - 1

    if(present(degree))then
                          Nk = degree
    else
                          Nk = 2
    end if


    allocate(Weights(-1:Nk, 0:Nk, 0:N))

    summation = 0

    do j=0, N
     if (mod(Nk,2)==0) then
                          s = max( 0, min(j-Nk/2, N-Nk) )
     else
                          s = max( 0, min(j-(Nk-1)/2, N-Nk) )
     endif
     xp = x(j)
     Weights(-1:Nk, 0:Nk, j)=Lagrange_polynomials(x = x(s:s+Nk), xp = xp )

     Int = sum ( Weights(-1, 0:Nk, j) * y(s:s+Nk) )

     summation   = summation  + Int

   enddo

   Integral = summation

   deallocate(Weights)

end function
```

Listing 2.6: `Interpolation.f90`

Finally, an additional function which is important for the next chapter is defined. This function determines the stencil or information that a q order interpolation requires.

```
function Stencilv(Order, N) result(S)

 integer, intent(in) :: Order, N
 integer :: S(0:N)

   integer :: i, N1, N2;

   if (mod(Order,2)==0) then
                     N1 = Order/2;
                     N2 = Order/2;
   else
                  N1 = (Order-1)/2;
                  N2 = (Order+1)/2;
   endif

   S(0:N1-1) = 0;
   S(N1:N-N2) = [ ( i, i=0, N-N1-N2 ) ];
   S(N-N2+1:N) = N - Order;


end function
```

Listing 2.7: `Lagrange_interpolation.f90`

2.2.3 Two variables functions

Whenever the approximated function for the set of nodes $\{(x_i, y_j)\}$ for $i = 0, 1 \ldots, N_x$ and $j = 0, 1 \ldots, N_y$ is $f : \mathbb{R}^2 \to \mathbb{R}$, the interpolant $I(x, y)$ can be calculated as a two dimensional extension of the interpolant for the single variable function. In such case, a way in which the interpolant $I(x, y)$ can be expressed is:

$$I(x, y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} b_{ij} \ell_i(x) \ell_j(y). \tag{2.15}$$

Again, using the property of Lagrange polynomials (2.2), the coefficients b_{ij} are determined as:

$$b_{ij} = f(x_i, y_j), \tag{2.16}$$

leading to the final expression for the interpolant:

$$I(x,y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} f(x_i, y_j)\ell_i(x)\ell_j(y). \tag{2.17}$$

Notice that when the interpolant is evaluated at a particular coordinate line $x = x_m$ or alternatively at $y = y_n$, it is obtained:

$$I(x_m, y) = \sum_{j=0}^{N_y} f(x_m, y_j)\ell_j(y), \qquad I(x, y_n) = \sum_{i=0}^{N_x} f(x_i, y_n)\ell_i(x), \tag{2.18}$$

which permits writing the interpolant as

$$\begin{aligned} I(x,y) &= \sum_{i=0}^{N_x} I(x_i, y)\ell_i(x) \\ &= \sum_{j=0}^{N_y} I(x, y_j)\ell_j(y). \end{aligned} \tag{2.19}$$

The form in which the interpolant is written in (2.19) suggests a procedure to obtain the interpolant recursively.

Another manner to interpret the equation (2.18) is as a bi-linear form. If the vectors $\boldsymbol{\ell}_x = \ell_i(x)\boldsymbol{e}_i$, $\boldsymbol{\ell}_y = \ell_j(y)\boldsymbol{e}_j$ and the second order tensor $\mathcal{F} = f(x_i, y_j)\boldsymbol{e}_i \otimes \boldsymbol{e}_j$ are defined, where the index (i, j) go through $[0, N_x] \times [0, N_y]$. This manner, the equation (2.18) can be written:

$$I(x,y) = \boldsymbol{\ell}_x \cdot \mathcal{F} \cdot \boldsymbol{\ell}_y. \tag{2.20}$$

Another perspective to interpret the interpolation is obtained by considering the process geometrically. In first place, it is calculated a single variable Lagrange interpolant for the function restricted at $y = s$:

$$f(x,y)\Big|_{y=s} = \tilde{f}(x;s) \simeq \tilde{I}(x;s) = \sum_{i=0}^{N_x} b_i(s)\ell_i(x), \tag{2.21}$$

where $\tilde{f}(x;s)$ is the restricted function, $\tilde{I}(x;s)$ its interpolant, $b_i(s) = \tilde{f}(x_i;s)$ are the coefficients of the interpolation and $\ell_i(x)$ are Lagrange polynomials.

The coefficients $b_i(s)$ can also be interpolated as:

$$b_i(s) = \sum_{j=0}^{N_y} b_{ij}\ell_j(s). \tag{2.22}$$

Hence, the restricted interpolant can be written:

$$\tilde{I}(x;s) = I(x,y)\Big|_{y=s} = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(s), \tag{2.23}$$

and therefore the interpolant $I(x,y)$ can be expressed:

$$I(x,y) = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(y). \tag{2.24}$$

In the same manner, the interpolated value $I(x,y)$ can be achieved restricting the value in $x = s$:

$$f(x,y)\Big|_{x=s} = \tilde{f}(y;s) \simeq \tilde{I}(y;s) = \sum_{j=0}^{N_y} b_j(s)\ell_j(y), \tag{2.25}$$

in which the coefficients $b_j(s)$ can be interpolated as well:

$$b_j(s) = \sum_{i=0}^{N_x} b_{ij}\ell_i(s). \tag{2.26}$$

This time, the restricted interpolant is expressed as:

$$\tilde{I}(y;s) = I(x,y)\Big|_{x=s} = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(s)\ell_j(y), \tag{2.27}$$

which leads to the interpolated value:

$$I(x,y) = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(y). \tag{2.28}$$

The interpolation procedure and its geometric interpretation can be observed on figure 2.1. In blue are represented the values $b_{ij} = f(x_i, y_j)$, in red the desired value $f(x, y) \simeq I(x, y)$ and in black the restricted interpolants.

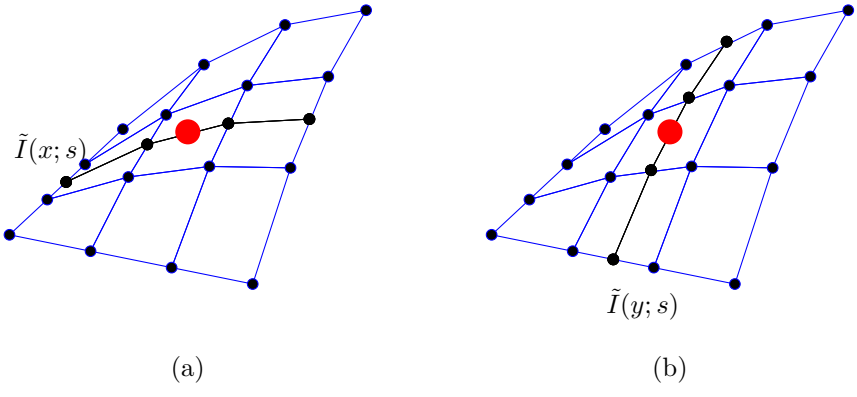

Figure 2.1: Geometric interpretation of the interpolation of a 2D function. (a) Geometric interpretation when restricted along y. (b) Geometric interpretation when restricted along x

Chapter 3

Finite Differences

3.1 Finite differences

On chapter 2, the interpolation using Lagrange polynomials has been presented. By means of this interpolation we have seen how it is possible to compute an approximation of the derivatives of a function. This is very advantageous not only to calculate derivatives of a known function but also to obtain approximate solutions of differential equations. Given a set of nodes $\{x_i \in \mathbb{R} \mid \quad i = 0, \ldots, q\}$ a *finite difference* formula is an expression that permits to approximate the derivative of a function $f(x)$ at these nodal points from its image at the set of nodes $\{f_i = f(x_i) \mid \quad i = 0, \ldots, q\}$. Let's suppose that we approximate f in a domain $[x_0, x_q]$ using Lagrange interpolation, that is we consider f to follow the expression:

$$f(x) = \sum_{i=0}^{q} f_i \ell_i(x), \tag{3.1}$$

therefore its k-th order derivative is written as

$$\frac{\mathrm{d}^k f(x)}{\mathrm{d}x^k} = \sum_{i=0}^{q} f_i \frac{\mathrm{d}^k \ell_i(x)}{\mathrm{d}x^k}. \tag{3.2}$$

If we want to calculate the derivative at a nodal point x_j we just have to evaluate (3.2) at that point, that is:

$$\frac{\mathrm{d}^k f(x_j)}{\mathrm{d}x^k} = \sum_{i=0}^{q} f_i \frac{\mathrm{d}^k \ell_i(x_j)}{\mathrm{d}x^k}. \tag{3.3}$$

The expression (3.3) is the finite difference formula of order q which approximates the derivative of order k at the point x_j. To illustrate the procedure let's consider the computation of the two first derivatives for order $q = 2$ and the set of equispaced nodes $\{x_0, x_1, x_2\}$, that is $x_2 - x_1 = x_1 - x_0 = \Delta x$. For this problem the interpolant and its derivatives are:

$$\begin{aligned} f(x) &= f_0 \frac{(x-x_1)(x-x_2)}{2\Delta x^2} - f_1 \frac{(x-x_0)(x-x_2)}{\Delta x^2} \\ &\quad + f_2 \frac{(x-x_0)(x-x_1)}{2\Delta x^2}, \\ \frac{\mathrm{d}f(x)}{\mathrm{d}x} &= f_0 \frac{(x-x_1)+(x-x_2)}{2\Delta x^2} - f_1 \frac{(x-x_0)+(x-x_2)}{\Delta x^2} \\ &\quad + f_2 \frac{(x-x_0)+(x-x_1)}{2\Delta x^2}, \\ \frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} &= \frac{f_0}{\Delta x^2} - \frac{2f_1}{\Delta x^2} + \frac{f_2}{\Delta x^2}. \end{aligned}$$

Note that the second derivative is the famous finite difference formula for centered second order derivatives. Evaluating the first derivative in the nodal points we obtain the well-known forward, centered and backward finite differences approximations of order 2:

$$\begin{aligned} \text{Forward:} \quad & \frac{\mathrm{d}f(x_0)}{\mathrm{d}x} = \frac{-3f_0 + 4f_1 - f_2}{2\Delta x}. \\ \text{Centered:} \quad & \frac{\mathrm{d}f(x_1)}{\mathrm{d}x} = \frac{f_2 - f_0}{2\Delta x}. \\ \text{Backward:} \quad & \frac{\mathrm{d}f(x_2)}{\mathrm{d}x} = \frac{f_0 - 4f_1 + 3f_2}{2\Delta x}. \end{aligned}$$

The main application of finite differences is the numerical resolution of differential equations. They serve to approximate the value of the unknown function in a set of nodes taking a finite number of points of the domain. For example, if we want to solve the 1D Boundary value problem:

$$\begin{aligned} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + 2\frac{\mathrm{d}u}{\mathrm{d}x} + u(x) = 0, \qquad & x \in (0,1), \\ \mathrm{d}u/\mathrm{d}x\,(0) = -2, \qquad & u(1) = 0. \end{aligned}$$

We can select a set of equispaced nodes $\{x_j \in [0,1] \,|\, j = 0, 1, \dots N\}$ which satisfy $0 = x_0 < x_1 < \cdots < x_N = 1$ and approximate the derivatives at those points by means of finite differences. If we use the previously derived second order formulas

we get the following system of $N+1$ equations:

$$\begin{aligned} \frac{-3u_0+4u_1-u_2}{2\Delta x} &= -2, \\ \frac{u_{j-1}-u_j+u_{j+1}}{\Delta x^2}+2\frac{u_{j+1}-u_{j-1}}{2\Delta x}+u_j &= 0, \qquad j=1,2,\dots N-1, \\ u_N &= 0, \end{aligned}$$

whose solution is an approximation of $u(x)$ in the nodal values. Note that for every point $j=0,1,\dots,N-1$ the formula used to approximate the first derivative is different. This is so as the set of Lagrange polynomials used to approximate the derivative at each point is different. For $j=0$ we use $\{\ell_0,\ell_1,\ell_2\}$ for the stencil $\{0,1,2\}$, while for $j=1,\dots,N-1$ we use $\{\ell_{j-1},\ell_j,\ell_{j+1}\}$ for the stencil $\{j-1,j,j+1\}$. The selection of the stencil must be done taking into account the order q of interpolation. In this example, we just had to differentiate between the inner points $0<j<N$ and the boundary points $j=0,N$ (note that if we needed to compute derivatives at x_N the formula would be the backward finite difference) but for generic order q the situation is slightly different. First of all, the stencil for even values of q consists of an odd number of nodal points and therefore the formulas can be centered. On the contrary, for odd values of q as the stencil contains an even number of nodal points the formulas are not centered. Nevertheless, in both cases the stencil is composed of $q+1$ nodal points which will be the ones used by the corresponding Lagrange interpolants. In the following lines we give a classification for both even and odd generic order q.

1. **Even order**: When q is even we have three possible scenarios for the stencil depending on the nodal point x_j. We classify the stencil in terms of its first element which corresponds to the index $j-q/2$.

   - For $j-q/2<0$ we use the stencil $\{x_0,\dots,x_q\}$ and its associated Lagrange polynomials $\{\ell_0(x),\dots,\ell_q(x)\}$ evaluated at x_j.
   - For $0\leq j-q/2\leq N-q$ we use the stencil $\{x_{j-q/2},\dots,x_{j+q/2}\}$ and its associated Lagrange polynomials $\{\ell_{j-q/2}(x),\dots,\ell_{j+q/2}(x)\}$ evaluated at x_j.
   - For $j-q/2>N-q$ we use the stencil $\{x_{N-q},\dots,x_N\}$ and its associated Lagrange polynomials $\{\ell_{N-q}(x),\dots,\ell_N(x)\}$ evaluated at x_j.

   On figure 3.1 is represented a sketch of the three different stencils for even order and the conditions under which are used.

2. **Odd order**: When q is odd we have three possible scenarios for the stencil depending on the nodal point x_j. We classify the stencil in terms of its first element which corresponds to the index $j-(q-1)/2$.

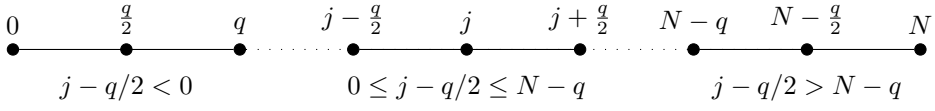

Figure 3.1: Sketch of the possible stencils for finite differences of even order q. Each set of three nodes represent the $q+1$ nodes that constitute the grid for the Lagrange polynomial $\ell_j(x)$.

- For $j-(q-1)/2<0$ we use the stencil $\{x_0,\dots,x_q\}$ and its associated Lagrange polynomials $\{\ell_0(x),\dots,\ell_q(x)\}$ evaluated at x_j.
- For $0\le j-(q-1)/2\le N-q$ we use the stencil $\{x_{j-(q-1)/2},\dots,x_{j+(q+1)/2}\}$ and its associated polynomials $\{\ell_{j-(q-1)/2}(x),\dots,\ell_{j+(q+1)/2}(x)\}$ evaluated at x_j.
- For $j-(q-1)/2>N-q$ we use the stencil $\{x_{N-q},\dots,x_N\}$ and its associated Lagrange polynomials $\{\ell_{N-q}(x),\dots,\ell_N(x)\}$ evaluated at x_j.

On figure 3.2 is represented a sketch of the three different stencils for odd order and the conditions under which are used.

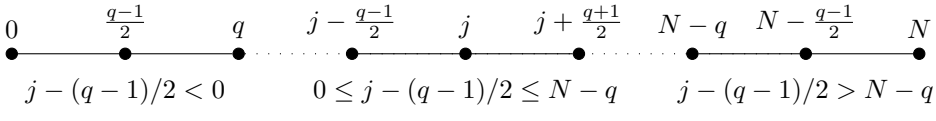

Figure 3.2: Sketch of the possible stencils for finite differences of odd order q. Each set of three nodes represent the $q+1$ nodes that constitute the grid for the Lagrange polynomial $\ell_j(x)$.

Hence, given a set of nodal points $\{x_0,\dots,x_N\}$ and an order of interpolation q, we can compute the coefficients of the finite difference formulae for the k-th derivative selecting the stencil as exposed using equation (3.3). This procedure is used to discretize the spatial domains of differential equations and transform them into systems of N equations. Although the procedure presented is for 1D domains it can be extended to higher dimensions in which the coefficients will involve Lagrange polynomials and stencils along the different dimensions. The main purpose of the module `Finite_differences` is: given a spatial grid (set of nodes) and an order q to compute the coefficients of the finite difference formulas for each point of the grid. In the following pages we present a brief explanation of how a library that carries out this procedure is implemented.

3.1.1 Algorithm implementation

In order to store the information and properties of the grid, a derived data type called `Grid` is defined and its properties declared as globals. This will permit to perform the computation of the coefficients of the high order derivatives just once.

```
  type Grid

    character(len=30) :: name
    real, allocatable :: Derivatives( :, :, :)
    integer :: N
    real, allocatable :: nodes(:)

  end type

  integer, save :: Order
  integer, parameter :: Nmax = 20
  type (Grid), save :: Grids(1:Nmax)
  integer, save :: ind = 0
```

Listing 3.1: `Finite_differences.f90`

The computation of the derivatives is carried out by the subroutine: `High_order_derivatives` which calls the function `Lagrange_polynomials`.

```
subroutine High_order_derivatives( z_nodes, Order, Derivatives)

  real, intent(in) ::  z_nodes(0:)
  integer, intent(in) :: Order
  real, intent(out) :: Derivatives(-1:Order, 0:Order, 0:size(z_nodes)-1)

   integer :: N, j, s
   real :: xp

   N = size(z_nodes) - 1;

   do j=0, N

    if (mod(Order,2)==0) then
                            s = max( 0, min(j-Order/2, N-Order) )
    else
                            s = max( 0, min(j-(Order-1)/2, N-Order) )
    endif

    xp = z_nodes(j)
    Derivatives(-1:Order, 0:Order, j) =  &
    Lagrange_polynomials( x = z_nodes(s:s+Order), xp = xp )

   enddo
end subroutine
```

Listing 3.2: `Finite_differences.f90`

The coefficients are computed once by the subroutine `Grid_initialization`.

```
subroutine Grid_Initialization( grid_spacing , direction, q, nodes )

          character(len=*),  intent(in) :: grid_spacing, direction
          integer, intent(in) ::  q
          real, intent(inout) :: nodes(:)

          integer d, df

     Order = q

     if (grid_spacing == "uniform") then

                call  Uniform_grid( nodes )
     else
                call  Non_uniform_grid( nodes, Order )
     endif

     d = 0
     d = findloc( Grids(:) % name, direction, dim=1   )

     if (d == 0) then

       ind = ind + 1
       Grids(ind) % N = size(nodes) - 1
       Grids(ind) % name = direction

       allocate(Grids(ind)%nodes(0:Grids(ind) % N ))
       allocate(Grids(ind)%Derivatives(-1:Order, 0:Order, 0:Grids(ind)%N))

       Grids(ind) % nodes = nodes

       call High_order_derivatives( Grids(ind) % nodes, Order,  &
                                    Grids(ind) % Derivatives    )
       write(*,*) " Grid name = ", Grids(ind) % name

     elseif (d > 0) then

       Grids(d) % N = size(nodes) - 1
       Grids(d) % name = direction

       deallocate(Grids(d) % nodes, Grids(d) % Derivatives)

       allocate(Grids(d) % nodes(0:Grids(d) % N ))
       allocate(Grids(d) % Derivatives(-1:Order, 0:Order, 0:Grids(d)%N))

       Grids(d) % nodes = nodes

       call High_order_derivatives( Grids(d) % nodes, Order,  &
                                    Grids(d) % Derivatives    )
     endif
end subroutine
```

Listing 3.3: `Finite_differences.f90`

Hence, after a single call to `Grid_initialization`, the uniform or non uniform grid of order q is defined and the coefficients of the derivatives settled down. In these conditions, defining a derivative (by finite differences) requires only an aditional slice of information, which is the stencil. It is clear that the amount of nodes required to compute a finite difference increases as the interpolation order does. For this, the subroutine that computes the derivatives must know how is defined the computational cell, that is, it must call the function `Stencilv`.

Taking this into account, the subroutine `Derivative1D` which calculates derivatives of single variable functions is implemented as follows.

```
subroutine Derivative1D( direction, derivative_order, W, Wxi )

   character(len=*), intent(in) :: direction
   integer, intent(in) :: derivative_order
   real, intent(in) ::   W(0:)
   real, intent(out)::   Wxi(0:)

    integer :: i, d, N
    integer, allocatable :: sx(:)
    integer :: k

    d = 0
    d = findloc( Grids(:) % name, direction, dim=1   )
    k = derivative_order

    if (d > 0) then

        N = Grids(d) % N
        allocate ( sx(0:N) )
        sx = Stencilv( Order, N )

        do i= 0,  N

            Wxi(i)=dot_product( Grids(d) % Derivatives(k, 0:Order, i), &
                                W(sx(i):sx(i)+Order) )
        enddo

       deallocate( sx )

    else

        write(*,*) " Error Derivative1D"
        stop

    endif

end subroutine
```

Listing 3.4: `Finite_differences.f90`

In an analogous manner, the computation of derivatives of functions of two

variables is carried out by the subroutine `Derivative2D`.

```
subroutine Derivative2D( direction, coordinate, derivative_order, W, Wxi )

   character(len=*), intent(in) :: direction(1:2)
   integer, intent(in) :: coordinate, derivative_order
   real, intent(in) ::   W(0:, 0:)
   real, intent(out)::   Wxi(0:, 0:)

    integer :: i, j, d1, d2, Nx, Ny
    integer, allocatable :: sx(:), sy(:)
    integer :: k

    d1 = 0  ;   d1 = findloc( Grids(:) % name, direction(1), dim=1 )
    d2 = 0  ;   d2 = findloc( Grids(:) % name, direction(2), dim=1 )

    k = derivative_order

 if (d1 > 0 .and. d2 > 0) then
 Nx = Grids(d1) % N
 Ny = Grids(d2) % N
 allocate( sx(0:Nx), sy(0:Ny) )
 sx = Stencilv( Order, Nx )
 sy = Stencilv( Order, Ny )

   do i=0, Nx
      do j=0, Ny

        if     (coordinate == 1) then

         Wxi(i,j) = dot_product( Grids(d1) % Derivatives(k, 0:Order, i), &
                                 W(sx(i):sx(i)+Order, j) );

        elseif (coordinate == 2) then

         Wxi(i,j) = dot_product( Grids(d2) % Derivatives(k, 0:Order, j), &
                                 W(i, sy(j):sy(j)+Order) );
        else
                write(*,*) " Error Derivative"
                stop
        endif

      enddo
   enddo
   deallocate( sx, sy )

   else
        write(*,*) " Error Derivative2D"
        write(*,*) "Grids =", Grids(:)% name, "direction =", direction
        write(*,*) "d1 =", d1, "d2 =", d2
        stop
   end if
end subroutine
```

Listing 3.5: `Finite_differences.f90`

Chapter 4

Cauchy Problem

4.1 Overview

In this chapter, a mathematical description of the Cauchy Problem is presented. Different temporal schemes are discussed as different algorithms to obtain the solution of a Cauchy problem. These algorithms are implemented by using vector operations that allows the Fortran language.

From the physical point of view, a Cauchy problem represents the evolution of any physical system with different degrees of freedom. From the movement of a material point in a three-dimensional space to the movement of satellites or stars, the movement is governed by a system of ordinary differential equations. If the initial condition of all degrees of freedom of this system is know, the movement can be predicted and the problem is named a Cauchy problem. Generally, this system involves first and second order derivatives of functions that depend on time. In order to design and to use the different temporal schemes, the problem is always formulated a system of first order equations.

From the mathematical point of view, a Cauchy problem is composed by a system of first order ordinary differential equations for $U : \mathbb{R} \to \mathbb{R}^N$ together with an initial condition $U(t_0) \in \mathbb{R}^N$.

$$\frac{dU}{dt} = F(U;\ t), \qquad F : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N, \tag{4.1}$$

$$U(t_0) = U^0, \quad \forall\ t\ \in\ [t_0,\ +\infty). \tag{4.2}$$

4.2 Algorithms or temporal schemes

To obtain temporal schemes, equation (4.1) is integrated between t_n and t_{n+1}

$$U(t_{n+1}) \;=\; U(t_n) \;+\; \int_{t_n}^{t_{n+1}} F(U;\; t)\; dt. \tag{4.3}$$

The idea of any temporal scheme is to approximate the integral appearing in (4.3) with an approximate value. Once the integral is approximated, U^n is used to denote the approximate value to differentiate it from the exact value $U(t_n)$. In figure 4.1, an scheme with the nomenclature of this chapter is shown. Superscript n stands for the approximated value at the temporal instant t_n. The approximated value of $F(U(t_n), t_n)$ is denoted by F^n.

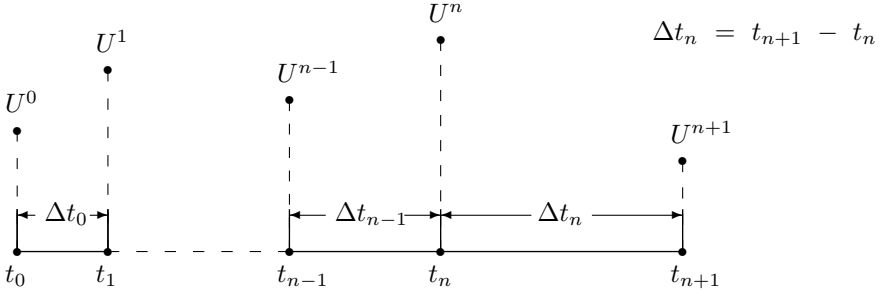

Figure 4.1: Partition of the temporal domain

Depending on how the integral appearing in equation (4.3) is carried out, different schemes are divided in the following groups:

1. Adams-Bashforth Moulton methods. If a polynomial interpolant to calculate the integral of equation (4.3) based on s time steps

   $$F^{n+1-j}, \quad = 0, \ldots, s$$

   is used, the resulting schemes are called Adams-Bashforth-Moulton methods.

2. Runge-Kutta methods. In this case, the integral of equation (4.3) is approximated by internal evaluations or temporal stages of $F(U, t\;)$ between t_n and t_{n+1}.

3. Gragg-borslish-Stoer methods. An algorithm based on successive refined grids inside the interval $[t_n, t_{n+1}]$ and using the Richardson extrapolation technique yields this schemes.

From the implementation point of view, two main main subroutines are designed. Given a temporal domain partition $[t_i, \quad i = 0, \dots M]$, a subroutine called `Cauchy_ProblemS` is responsible to call different temporal schemes to approximate (4.3). In the following code the implementation of this subroutine is shown:

```
subroutine Cauchy_ProblemS( Time_Domain, Differential_operator,  &
                            Solution, Scheme )
     real, intent(in) :: Time_Domain(:)
     procedure (ODES) :: Differential_operator
     real, intent(out) :: Solution(:,:)
     procedure (Temporal_Scheme), optional :: Scheme

!  *** Initial and final time
       real :: start, finish, t1, t2
       integer ::  i, N_steps, ierr

!  *** loop for temporal integration
       call cpu_time(start)
       N_steps = size(Time_Domain) - 1;
       do i=1, N_steps
           t1 = Time_Domain(i) ; t2 = Time_Domain(i+1);

           if (present(Scheme))  then
               call Scheme( Differential_operator, t1, t2,            &
                            Solution(i,:), Solution(i+1,:), ierr )

           else if (family/=" ") then
               call Adavanced_Scheme

           else
               call Runge_Kutta4( Differential_operator, t1, t2,      &
                                  Solution(i,:), Solution(i+1,:), ierr )
           endif
           if (ierr>0) exit
       enddo
      call cpu_time(finish)
 write(*, '("Cauchy_Problem, CPU Time=",f6.3," seconds.")') finish-start
 write(*, *)
contains
```

Listing 4.1: `Cauchy_problem.f90`

The arguments of this subroutine are: the `Time_Domain` represented in figure (4.1), the `Differential_operator` or the vector function $F(u,t)$, the `Solution` or the vector U and the selected temporal scheme to carry out the integral of equation (4.3). Note that `Solution` is a two-dimensional array that stores the value of every variable of system (second index) at every time step (first index).

It can be seen that the temporal scheme is an optional argument, if it is not present, this subroutine uses a classical fourth order Runge-Kutta scheme. Besides, advanced high order methods, belonging to different families or groups, can be used.

Once the arguments of `Cauchy_ProblemS` are associated, this subroutine calls the selected temporal scheme to integrate from t_i to t_{i+1}. In this way, the `Scheme` subroutine calculates `Solution(i+1,:)` from the input value `Solution(i,:)`. Hence, the intelligence of the particular details of any specific algorithm is hidden in `Scheme`. In the following code the implementation of this subroutine `Runge_Kutta4` is shown:

```
subroutine Runge_Kutta4( F, t1, t2, U1, U2, ierr )
     procedure (ODES) :: F
     real, intent(in) :: t1, t2, U1(:)
     real, intent(out) ::  U2(:)
     integer, intent(out) :: ierr

       real :: t, dt
       real :: k1(size(U1)), k2(size(U1)), k3(size(U1)), k4(size(U1))

       dt = t2-t1;   t = t1

       k1 = F( U1, t)
       k2 = F( U1 + dt * k1/2, t + dt/2 )
       k3 = F( U1 + dt * k2/2, t + dt/2 )
       k4 = F( U1 + dt * k3,   t + dt   )

       U2 = U1 + dt * ( k1 + 2*k2 + 2*k3 + k4 )/6

       ierr = 0

    end subroutine
```

Listing 4.2: `Temporal_Schemes.f90`

This is the classical fourth order Runge-Kutta. Given the input value `U1`, and the vector function `F`,the scheme calculates the value `U2`. In the following code the interface of the vector function `F` is shown:

```
  function ODES( U, t)

         real :: U(:), t
         real :: ODES( size(U) )

  end function
```

Listing 4.3: `ODE_Interface.f90`

4.3 Implicit temporal schemes

When the integral of equation (4.3) done by any the the different temporal schemes involves the value U^{n+1}, the resulting scheme becomes implicit and a nonlinear system of N equations must be solved at each time step. Since the complexity and the computational cost of implicit methods is much greater than the explicit methods, the only reason to implement these methods relies on the stability behavior. Generally, implicit methods do nor require time steps limitations or constraints to be numerically stable. The simplest implicit method is the inverse Euler method,

$$U^{n+1} = U^n + \Delta t_n \; F^{n+1}. \tag{4.4}$$

It is obtained when interpolating in equation (4.3) with a constant value F^{n+1}. If U^n is known from the last time step, equation (4.4) can be formulated as the determination of roots of the following equation:

$$G(X) = X - U^n - \Delta t \; F(X, \; t_{n+1}). \tag{4.5}$$

From the implementation point of view, the scheme can be implemented with the methodology presented above. In the following code, the subroutine `Inverse_Euler` uses a Newton method to solve the equation (4.5) each time step.

```
subroutine Inverse_Euler(F, t1, t2, U1, U2, ierr )
     procedure (ODES) :: F
     real, intent(in) :: t1, t2, U1(:)
     real, intent(out) ::  U2(:)
     integer, intent(out) :: ierr

      real :: dt

      dt = t2-t1
      U2 = U1

    ! Try to find a zero of the residual of the inverse Euler
      call Newtonc( F = Residual_IE, x0 = U2 )

      ierr = 0
contains

function Residual_IE(X) result(G)
         real, target :: X(:), G(size(X))

      G = X - U1 - dt * F(X, t2)

      where (F(X, t2)==ZERO) G = 0

end function
end subroutine
```

Listing 4.4: `Temporal_Schemes.f90`

4.4 Richardson's extrapolation to determine error

Since the error of a numerical solution is defined as the difference between the exact solution $\boldsymbol{u}(t_n)$ minus the the approximate solution U^n at the same instant t_n

$$E^n = \boldsymbol{u}(t_n) - U^n, \tag{4.6}$$

the determination the error requires knowing the exact solution. This situation is unusual and makes necessary to find some technique out.

If the global error could be expanded in power series of Δt like

$$E^n = k(t_n)\Delta t^q + O(\Delta t^{q+1}), \tag{4.7}$$

with $K(t_n)$ independent of Δt, then an estimation based on Richardson's extrapolation could be done.

For one-step methods this expansion can be found. However, for multi-step methods, the presence of spurious solutions do not allow this expansion. To cure this problem and to eliminate the oscillatory behavior of the error, averaged values $\overline{U}^n$ can be defined as:

$$\overline{U}^n = \frac{1}{4}\left(U^n + 2U^{n-1} + U^{n-2}\right), \tag{4.8}$$

allowing expansions like (4.7).

If the error can be expanded like in (4.7) and by integrating two grids one with time step Δt_n and other with $\Delta t_n/2$, an estimation of the error based on Richardson's extrapolation can be found. Let U_1 be the solution integrated with Δt_n and U_2 the solution integrated with $\Delta t_n/2$. The expression (4.7) for the two solutions is written:

$$\boldsymbol{u}(t_n) - U_1^n = k(t_n)\Delta t^q + O(\Delta t^{q+1}), \tag{4.9}$$

$$\boldsymbol{u}(t_n) - U_2^{2n} = k(t_n)\left(\frac{\Delta t}{2}\right)^q + O(\Delta t^{q+1}). \tag{4.10}$$

Substracting equation (4.9) and equation (4.10),

$$U_2^{2n} - U_1^n = k(t_n)\Delta t^q\left(1 - \frac{1}{2^q}\right) + O(\Delta t^{q+1}), \tag{4.11}$$

allowing the following error estimation:

$$E^n = \frac{U_2^{2n} - U_1^n}{1 - \frac{1}{2^q}}. \tag{4.12}$$

In the following code, the error estimation based on Richardson's extrapolation is implemented:

```
subroutine Error_Cauchy_Problem( Time_Domain, Differential_operator, &
                                 Scheme, order, Solution, Error )
            real, intent(in) :: Time_Domain(0:)
            procedure (ODES) :: Differential_operator
            procedure (Temporal_Scheme) :: Scheme
            integer, intent(in) :: order
            real, intent(out) ::  Solution(0:,:), Error(0:,:)

   integer :: i, N, Nv
   real, allocatable :: t1(:), U1(:,:), t2(:), U2(:,:)

       N = size(Time_Domain)-1;  Nv = size(Solution, dim=2)
       allocate ( t1(0:N),  U1(0:N, Nv), t2(0:2*N), U2(0:2*N, Nv) )
       t1 = Time_Domain

       do i=0, N-1
           t2(2*i)   = t1(i)
           t2(2*i+1) = ( t1(i) + t1(i+1) )/2
       end do
       t2(2*N) = t1(N)

       U1(0,:) = Solution(0,:); U2(0,:) = Solution(0,:)

       call  Cauchy_ProblemS(t1, Differential_operator, U1, Scheme)
       call  Cauchy_ProblemS(t2, Differential_operator, U2, Scheme)

       do i=N, 0, -1
                 U1(i,:) = ( U1(i, :) + 2 * U1(i-1, :) + U1(i-2, :) )/4
       end do
       do i=2*N, 0, -1
                 U2(i,:) = ( U2(i, :) + 2 * U2(i-1, :) + U2(i-2, :) )/4
       end do

       do i=0, N
            Error(i,:) = ( U2(2*i, :)- U1(i, :) )/( 1 - 1./2**order )
       end do
       Solution = U1 + Error

       deallocate ( t1,  U1, t2, U2 )

end subroutine
```

Listing 4.5: `Temporal_error.f90`

Given a `Time_Domain`, two temporal grids are defined `t1` and `t2`. While `t1` is the original temporal grid, `t2` has double of points than `t1` and it is obtained by halving the time steps of `t1`. Then, two independent simulations `U1` and `U2` are carried out starting from the same initial condition. They are averaged with expression (4.8) to eliminate oscillations and `Error` is calculated with expresion (4.12). Finally, the `Error` is used to correct the `U1` solution to give the `Solution`.

4.5 Convergence rate of temporal schemes

A numerical scheme is said to be of order q if its numerical error is $O(\Delta t^q)$. It means that if Δt is small enough, error tends to zero with the same velocity than Δt^q. Taking norms and logarithms in the error expression (4.7) and taking into account that $\Delta t \propto N^{-1}$,

$$\log \|E^n\| = C - q \ \log N. \tag{4.13}$$

When plotting this expression in log scale, it appears a line with a negative slope q which is the order of the method. When dealing with complex temporal schemes or when developing new methods, it is importance to know the convergence rate of the scheme or its real order. To do that, the error must be known. As it was shown in the last section, error can be determined based on Richardson's extrapolation. In the following code, a sequence of Cauchy problems with $\Delta t_n/2^k$ is integrated. This subroutine allows obtaining the dependency of logarithm of the error `log_E` with the logarithm of number of time steps `log_N`.

```
subroutine Temporal_convergence_rate( Time_Domain, Differential_operator,&
                                      U0, Scheme, order, log_E, log_N)
                  real, intent(in) :: Time_Domain(:), U0(:)
                  procedure (ODES) :: Differential_operator
                  procedure (Temporal_Scheme), optional :: Scheme
                  integer, intent(in) :: order
                  real, intent(out) :: log_E(:), log_N(:)

    real :: error
    real, allocatable :: t1(:), t2(:),  U1(:, :), U2(:, :)
    integer :: i, m, N, Nv

    N = size( Time_Domain ) - 1; Nv = size( U0 );  m = size(log_N)
    allocate ( t1(0:N), U1(0:N, Nv) )

    U1(0,:) = U0(:)
    t1 = Time_Domain
    call Cauchy_ProblemS( t1, Differential_operator, U1, Scheme )

    do i = 1, m ! simulations in different grids
       N =  2 * N
       allocate ( t2(0:N), U2(0:N, Nv) )
       t2(0:N:2) =  t1; t2(1:N-1:2) = ( t1(1:N/2)  + t1(0:N/2-1) )/ 2
       U2(0,:) = U0(:)

       call Cauchy_ProblemS( t2, Differential_operator, U2, Scheme )

       error = norm2( U2(N, :) - U1(N/2, :) ) / (1 - 1./2**order)
       log_E(i) = log10( error );  log_N(i) = log10( real(N) )
       deallocate( t1, U1 );  allocate ( t1(0:N), U1(0:N, Nv) )
       t1 = t2;  U1 = U2;     deallocate( t2, U2)
    end do
end subroutine
```

Listing 4.6: `Temporal_error.f90`

4.6 Embedded Runge-Kutta methods

Adaptive Runge-Kutta methods are designed to produce an estimate of the local truncation error. If that error is below the required tolerance, the time step is accepted and the ne time step is increased. If not, the time step is reduced based on the local truncation error. This is done by having two Runge-Kutta methods at the same time, one with order q and one with order $q+1$. These methods are interwoven sharing intermediate steps or stages. Thanks to this, the error estimation has negligible computational. Moreover, the time adapts automatically depending on the gradients of the solution reducing the computational cost.

The two Runge-Kutta formulas calculate the approximation $\boldsymbol{u}^{n+1}$ of order q and another approximation $\hat{\boldsymbol{u}}^{n+1}$ of order $q+1$. The subtraction $\boldsymbol{u}^{n+1} - \hat{\boldsymbol{u}}^{n+1}$ gives an estimation of the local truncation error. Hence, the local error can be controlled by changing the step size for each temporal step.

A Runge-Kutta method of e stages predicts the approximation $\boldsymbol{u}^{n+1}$ from the previous value $\boldsymbol{u}^{n+1}$ by the following expression:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + h \sum_{i=1}^{e} \boldsymbol{k}_i \, b_i, \tag{4.14}$$

where $h = t_{n+1} - t_n$, matrix a_{ij} is the Butcher's array, b_i and c_i are constant of the scheme and

$$\boldsymbol{k}_i = \boldsymbol{F}\left(t_n + c_i h, \boldsymbol{u}^n + \sum_{j=1}^{e} a_{ij} \, \boldsymbol{k}_j\right).$$

The Butcher array for a generic Runge-Kutta is written as follows:

$$\begin{array}{c|ccc} & a_{11} & \dots & a_{1e} \\ c_2 & a_{21} & \dots & a_{2e} \\ \vdots & \vdots & & \vdots \\ c_e & a_{e1} & \dots & a_{ee} \\ \hline \boldsymbol{u}^{n+1} & b_1 & \dots & b_e \end{array} \tag{4.15}$$

Note that as $c_1 = 0$, it does not appear on the Butcher array. In the special case in which $a_{ij} = 0, \forall\, i \leq j$, then the Runge-Kutta is explicit, that is, $\boldsymbol{k}_i$ can be obtained from $\{\boldsymbol{k}_1, \dots, \boldsymbol{k}_{i-1}\}$.

The embedded Runge-Kutta method uses two explicit schemes sharing c_i and

a_{ij} for all $i, j \leq e$. Therefore, this method has the extended Butcher's array:

$$\begin{array}{c|ccc} c_2 & a_{21} & & \\ \vdots & \vdots & & \\ c_e & a_{e1} & \dots & a_{ee-1} \\ \hline \boldsymbol{u}^{n+1} & b_1 & \dots & b_e \\ \hline \hat{\boldsymbol{u}}^{n+1} & \hat{b}_1 & \dots & \hat{b}_e \end{array} \tag{4.16}$$

In which b_i and $\hat{b}_i$ are respectively the coefficients of the approximated solutions $\boldsymbol{u}^{n+1}$ and $\hat{\boldsymbol{u}}^{n+1}$. Since the local truncation error of $\boldsymbol{u}^{n+1}$ is $C\ h^{q+1}$ and the error of $\hat{\boldsymbol{u}}^{n+1}$ is $\hat{C}\ h^{q+2}$, the estimation of the local truncation error $\boldsymbol{T}^{n+1}$ of order $q+1$ is obtained by substantiating the two approximations:

$$\boldsymbol{T}^{n+1} = \boldsymbol{u}^{n+1} - \hat{\boldsymbol{u}}^{n+1} = \boldsymbol{C}\ h^{q+1}. \tag{4.17}$$

If the norm of the local truncation error should less than a prescribed tolerance ϵ, then the optimal time step $\hat{h}$ can be obtained from equation (4.17) to yield:

$$\epsilon = \|\boldsymbol{C}\|\ \hat{h}^{q+1}. \tag{4.18}$$

Taking norms into equation (4.17) and dividing the result by equation (4.18)

$$\frac{\epsilon}{\|\boldsymbol{T}^{n+1}\|} = \left(\frac{\hat{h}}{h}\right)^{q+1} \tag{4.19}$$

and the optimum time step can be obtained from the previous time step

$$\hat{h} = h\left(\frac{\epsilon}{\|\boldsymbol{T}^{n+1}\|}\right)^{\frac{1}{q+1}}. \tag{4.20}$$

This step size selection is implemented in the following code:

```
real function Step_size( dU, tolerance , q , h  )
     real, intent(in) :: dU(:), tolerance, h
     integer, intent(in) :: q

    real :: normT
    normT = norm2(dU)

    if (normT > tolerance) then

           Step_size = h * (tolerance/normT)**(1./(q+1))
    else
           Step_size = h
    end if
end function
```

Listing 4.7: `Embedded_RKs.f90`

With this time step selection, an embedded Runge-Kutta scheme is implemented:

```
subroutine ERK_scheme( F, t1, t2, U1, U2, ierr )
       procedure (ODES) :: F
       real, intent(in) :: t1, t2, U1(:)
       real, intent(out) ::  U2(:)
       integer, intent(out) :: ierr

       real :: V1(size(U1)) , V2(size(U1)), h, t
       integer ::  i, N

 ! *** Check if a method has been selected
       if (.not.Method_selection) then
           RK_Method = "DOPRI54"
           RK_Tolerance = 1d-4
       end if

 ! *** First order q solution and Second order q+1
 !     for initial step size
       call RK_scheme( RK_Method,  "First", F, t1, t2, U1, V1 )
       call RK_scheme( RK_Method, "Second", F, t1, t2, U1, V2 )

 ! *** Local error estimation and step size calculation
 !     to satisfy tolerance condition
       h = t2 - t1
       h =  min( h, Step_size(V1 - V2, RK_Tolerance, minval(q), h) )

 ! *** Grid for the new step
       N = int( (t2 - t1) / h  ) +1
       h = (t2 - t1) / N

 ! *** Solution for embedded grid
       V1 = U1  ; V2 = U1
       do i = 0, N - 1
          t = t1 + i * (t2 - t1) / N
          V1 = V2
          call RK_scheme( RK_Method, "First", F, t, t + h, V1, V2 )
       end do

       U2 = V2

       ierr = 0
end subroutine
```

Listing 4.8: `Embedded_RKs.f90`

The subroutine `RK_scheme` is called twice to calculate the approximate solution `V1` and `V2` form the previous solution `U1`. Once the subroutine `set_tolerance` assign a specific value to the required tolerance `RK_tolerance`, the subroutine `Step_size` validates or reduces the time step `h=t2-t1`. Then, with the resulting time step `h` and by means of the `"First"` Runge-Kutta scheme, the approximate solution `U2` is obtained.

In the following code, the subroutine `RK_schme is implemented`

```
subroutine RK_scheme( name, tag , F, t1, t2, U1, U2 )
       character(len=*), intent(in) :: name , tag
       procedure (ODES) :: F
       real, intent(in) :: t1, t2, U1(:)
       real, intent(out) ::   U2(:)

       real :: Up( size(U1) ), h
       integer ::  i, j, Ne

       h = t2 - t1

 ! *** Solution for the first RK
       if ( tag == "First" ) then

          call Butcher_array( name, Ne )
          if (.not.allocated(k)) allocate ( k( Ne, size(U1) ) )
          do i = 1, Ne

            Up = U1
            do j=1, i-1
               Up = Up + h * a(i,j) * k(j, :)
            end do

            k(i,:) = F( Up, t1 + c(i) * h )

          end do
          N_eRK_effort = N_eRK_effort + Ne

          U2 = U1 + h * matmul( b, k )

 ! *** Solution for the second RK
       elseif ( tag == "Second") then

          U2 = U1 + h * matmul( bs, k )
          deallocate(k)

       end if

end subroutine
```

Listing 4.9: `Embedded_RKs.f90`

A pair of Runge-Kutta schemes are identified by its `name` which must be previously selected by the subroutine `set_solver`. If the subroutine is called with `tag="First"`, the butcher's array is created and the values of different stages $\boldsymbol{k}_i$ are calculated and stored in `k(i,:)` where the first index stands for the stage index and the second index represents the index of the variable. Later, an approximate value for `U2` is calculated by using `b` coefficients previously defined. If the subroutine is called with `tag="Second"` and since the butcher's array and the `k(i,:)` are saved, an approximate value for `U2` is calculated by using `bs` coefficients previously defined.

4.7 Gragg-Bulirsch-Stoer method

In the GBS algorithm, the error is improved by halving consecutively the interval $[t_n, t_{n+1}]$ and by using the Richardson's extrapolation technique.

The method divides consecutively the interval in $2n_i$ pieces where $i = 1, 2 \dots l$ where n_i is the sequence of grid levels. The number of grids l is also called the number of levels that the GBS algorithm descends.

For each level, a solution at the next step $\boldsymbol{u}_i^{n+1}$ is obtained applying the Modified midpoint scheme. Note that this solution is the solution for a certain grid at t_{n+1}. Hence, l solutions $\boldsymbol{u}_i^{n+1}$ will allow by using the Richardson's extrapolation to obtain an estimation of the global error of order $2l$. This estimation is used to correct the approximated solution. The algorithm for Gragg-Bulirsch-Stoer method can be summed up as follows:

1. Divide the interval in $2n_i$ pieces.

   For each level, the time step h is divided into $2n_i$ segments:

   $$t_j = t_n + j\ h_i, \qquad j = 0, 1 \dots, 2n_i. \tag{4.21}$$

   where $h_i = \ h/(2n_i)$.

2. Modified midpoint scheme.

   The solution $\boldsymbol{u}_i^{n+1}$ is obtained in each level by applying the modified midpoint scheme:

   $$\tilde{\boldsymbol{u}}^1 = \boldsymbol{u}^0 + h_i \boldsymbol{f}(t_0, \boldsymbol{u}^0), \tag{4.22}$$
   $$\tilde{\boldsymbol{u}}^{j+1} = \tilde{\boldsymbol{u}}^{j-1} + 2h_i \boldsymbol{f}(t_j, \tilde{\boldsymbol{u}}^j), \qquad j = 1, 2 \dots, 2n_i - 1, \tag{4.23}$$
   $$\boldsymbol{u}_i^{n+1} = \left(\tilde{\boldsymbol{u}}^{2n_i - 2} + 2\tilde{\boldsymbol{u}}^{2n_i - 1} + \tilde{\boldsymbol{u}}^{2n_i}\right)/4 \tag{4.24}$$

   The midpoint scheme or the Leap-Frog method is used to determine the solution at the inner points of any level. Since the the Leap-Frog method is a two-step scheme, an extra initial condition is required. This is given an explicit Euler scheme (4.22). Once the Leap-Frog reaches the end of the interval, the solution is smoothed by equation (4.24).

3. Richardson extrapolation for GBS algorithm.

   Due to the symmetry of the GBS scheme, it was proven by Gragg (1963) that an estimation of its global error based on the $\boldsymbol{u}_i^{n+1}$ solutions is $O(h^{2l})$. This error is used to improve the precision of the solution:

   $$\boldsymbol{u}^{n+1} = \boldsymbol{u}_1^{n+1} + \boldsymbol{E}^{n+1}.$$

In the following discussion, a formula for the error $\boldsymbol{E}^{n+1}$ based on the solutions $\boldsymbol{u}_i^{n+1}$ will be given. It was proven that the global error

$$\boldsymbol{E}(t_{n+1}) = \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}^{n+1} \tag{4.25}$$

posses an asymptotic expansion in terms of even powers of the step h:

$$\boldsymbol{E}(t_{n+1}) = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h^{2j}. \tag{4.26}$$

For each level or grid, the temporal step is written $h_i = h/n_i$ which leads to l expressions for the error:

$$\begin{aligned}
\boldsymbol{E}_1(t_{n+1}) &= \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}_1^{n+1} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h_1^{2j} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left(\frac{h}{n_1}\right)^{2j}, \\
\boldsymbol{E}_2(t_{n+1}) &= \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}_2^{n+1} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h_2^{2j} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left(\frac{h}{n_2}\right)^{2j}, \\
&\vdots \\
\boldsymbol{E}_i(t_{n+1}) &= \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}_i^{n+1} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h_i^{2j} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left(\frac{h}{n_i}\right)^{2j}, \\
&\vdots \\
\boldsymbol{E}_l(t_{n+1}) &= \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}_l^{n+1} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h_l^{2j} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left(\frac{h}{n_l}\right)^{2j}.
\end{aligned}$$

The system of equations can be expressed for $i = 1, 2 \ldots, l$ as:

$$\boldsymbol{E}_i(t_{n+1}) = \boldsymbol{u}(t_{n+1}) - \boldsymbol{u}_i^{n+1} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h_i^{2j} = \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left(\frac{h}{n_i}\right)^{2j}. \tag{4.27}$$

Substracting equation of level $i+1$ and level i yields:

$$\begin{aligned}
\boldsymbol{u}_{i+1}^{n+1} - \boldsymbol{u}_i^{n+1} &= \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) \left[\left(\frac{h}{n_i}\right)^{2j} - \left(\frac{h}{n_{i+1}}\right)^{2j} \right], \\
&= \sum_{j=1}^{\infty} \boldsymbol{k}_{2j}(t_n) h^{2j} \left[\left(\frac{1}{n_i}\right)^{2j} - \left(\frac{1}{n_{i+1}}\right)^{2j} \right], \\
&= \sum_{j=1}^{\infty} A_{ij}\, \boldsymbol{k}_{2j}(t_n) h^{2j},
\end{aligned} \tag{4.28}$$

The above expression yields the exact global error but requires an infinite number of terms. If the number of levels is high enough, the rate of convergence of terms of series h^{2j} allow to truncate the series to give a good approximation. That is, an estimation of the error can be obtained with l levels

$$\boldsymbol{u}_{i+1}^{n+1} - \boldsymbol{u}_{i}^{n+1} \simeq \sum_{j=1}^{l-1} A_{ij}\, \boldsymbol{k}_{2j}(t_n) h^{2j}, \tag{4.29}$$

or in vector form:

$$\begin{bmatrix} \boldsymbol{u}_{2}^{n+1} - \boldsymbol{u}_{1}^{n+1} \\ \vdots \\ \boldsymbol{u}_{i+1}^{n+1} - \boldsymbol{u}_{i}^{n+1} \\ \vdots \\ \boldsymbol{u}_{l}^{n+1} - \boldsymbol{u}_{l-1}^{n+1} \end{bmatrix} = \begin{bmatrix} A_{ij} \end{bmatrix} \begin{bmatrix} \boldsymbol{k}_2(t_n) h^2 \\ \vdots \\ \boldsymbol{k}_{2j}(t_n) h^{2j} \\ \vdots \\ \boldsymbol{k}_{2(l-1)}(t_n) h^{2(l-1)} \end{bmatrix}. \tag{4.30}$$

This system is invertible and can be solved as:

$$\begin{bmatrix} \boldsymbol{k}_2(t_n) h^2 \\ \vdots \\ \boldsymbol{k}_{2j}(t_n) h^{2j} \\ \vdots \\ \boldsymbol{k}_{2(l-1)}(t_n) h^{2(l-1)} \end{bmatrix} = \begin{bmatrix} A_{ij}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_{2}^{n+1} - \boldsymbol{u}_{1}^{n+1} \\ \vdots \\ \boldsymbol{u}_{i+1}^{n+1} - \boldsymbol{u}_{i}^{n+1} \\ \vdots \\ \boldsymbol{u}_{l}^{n+1} - \boldsymbol{u}_{l-1}^{n+1} \end{bmatrix} \tag{4.31}$$

Since an estimation of the error is:

$$\boldsymbol{E}(t_{n+1}) = \sum_{j=1}^{l-1} \boldsymbol{k}_{2j}(t_n) h^{2j}, \tag{4.32}$$

$$\boldsymbol{E}^{n+1} = \begin{bmatrix} 1, \ldots, 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{k}_2(t_n) h^2 \\ \vdots \\ \boldsymbol{k}_{2j}(t_n) h^{2j} \\ \vdots \\ \boldsymbol{k}_{2(l-1)}(t_n) h^{2(l-1)} \end{bmatrix} = \begin{bmatrix} 1, \ldots, 1 \end{bmatrix} \begin{bmatrix} A_{ij}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_{2}^{n+1} - \boldsymbol{u}_{1}^{n+1} \\ \vdots \\ \boldsymbol{u}_{i+1}^{n+1} - \boldsymbol{u}_{i}^{n+1} \\ \vdots \\ \boldsymbol{u}_{l}^{n+1} - \boldsymbol{u}_{l-1}^{n+1} \end{bmatrix}. \tag{4.33}$$

By substracting pairs of solutions of levels $i+1$ and i, the system of equations (4.33) allows to estimate the global error $\boldsymbol{E}^{n+1}$. This error is used to improve the precision of the solution:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}_1^{n+1} + \boldsymbol{E}^{n+1}.$$

The GBS algorithm described in this section is implemented in the following code:

```
subroutine GBS_Scheme( F, t1, t2, U1, U2, ierr)
     procedure (ODES) ::   F
     real, intent(in) ::   t1, t2, U1(:)
     real, intent(out) ::  U2(:)
     integer, intent(out) :: ierr

    real, allocatable :: U(:, :), dU(:, :), b(:)
    integer, allocatable :: n(:)
    real :: Error( size( U1) )
    integer :: i, Nv, N_levels, Nmax = 9

    Nv = size(U1)
    if (Tolerance==0) Tolerance = 0.1

    N_levels = 1; Error = 10
    do while (norm2(Error) > Tolerance)

        if (N_levels > Nmax) then
            write(*,*) "ERROR GBS Tolerance not reached", norm2(Error);
                exit
        end if
        N_levels = N_levels + 1
        allocate( U( Nv, N_Levels ), dU( Nv, N_Levels-1 ) )
        allocate( n(N_Levels), b(N_Levels-1) )

  ! *** Partition sequence definition
        n = [(i, i=1, N_levels)]

  ! *** Richardson extrapolation coeffcicients
        call GBS_Richardson_coefficients(n, b)

  ! *** Modified midpoint scheme for each level
        do i = 1, N_Levels
           call Modified_midpoint_scheme( F, t1, t2, U1, U(:, i), n(i) )
        end do

  ! *** Error by means of the difference between solutions j and j+1
        do i = 1, N_Levels - 1
             dU(:, i)  = U(:, i+1) - U(:, i)
        end do
        Error(:) = matmul( dU(:,:) , b )

  ! *** Solution correction j=1
        U(:, 1) = U(:, 1) + Error(:)

  ! *** Final solution
        U2 = U(:, 1); ierr = 0
        deallocate( U, dU, n, b)

    end do
end subroutine
```

Listing 4.10: `Gragg_Burlisch_Stoer.f90`

The required operations described in expression (4.33) to obtain the error estimation is implemented in the following code:

```
subroutine GBS_Richardson_coefficients(n, b)
      integer, intent(in) :: n(:)
      real, intent(out) :: b(:)

      real :: ones( size(n)-1 ), A( size(n)-1, size(n)-1)
      integer :: i, j, q

    ! *** A^T computation
          q = size(n) - 1

          do i = 1, q
             do j = 1, q
              A(j,i) = ( ( 1./n(i))**(2*j) - (1./n(i+1))**(2*j)  )
            end do
          end do

    ! *** Vector b computation
          ones = 1.
          call LU_factorization( A )
          b = Solve_LU( A , ones )

end subroutine
```

Listing 4.11: Gragg_Burlisch_Stoer.f90

The modified midpoint method is implemented in the following code:

```
subroutine Modified_midpoint_scheme( F, t0, t, U0, Un, n )
       procedure (ODES) ::    F
       real, intent(in) ::    t0, t , U0(:)
       real, intent(out) ::   Un(:)
       integer, intent(in) :: n

       real :: ti, h, U( size(U0), 0:2*n+1 )
       integer ::  i

        h = (t - t0) / ( 2*n )

        U(:,0) = U0
        U(:,1) = U(:,0) + h * F( U(:,0), t0 )

        do i=1, 2*n
            ti = t0 + i*h
            U(:, i+1)  = U(:, i-1) + 2*h* F( U(:,i), ti )
        end do

        Un = ( U(:, 2*n-1) + 2 *  U(:, 2*n) + U(:, 2*n+1)  )/4.

        N_GBS_effort = N_GBS_effort + 2*n + 1

end subroutine
```

Listing 4.12: Gragg_Burlisch_Stoer.f90

4.8 ABM or multi-value methods

In this chapter, the last of the classical high order temporal schemes will be presented, the Adams-Bashforth-Moulton methods (ABM). These methods are based on linear multi-step methods (Adams) which can be explicit (Adams-Bashforth) or implicit (Adams-Moulton). Given an interval $[t_n, t_{n+1}]$ of length Δt, this family of methods gives the solution at the end of the interval as:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t \sum_{j=0}^{s} \beta_j \boldsymbol{F}^{n+1-j}, \tag{4.34}$$

where $\boldsymbol{F}^{n+1-j} = \boldsymbol{F}(t_{n+1-j}, \boldsymbol{u}^{n+1-j})$, and β_j are the coefficients of the scheme. Note, that an $s-$step explicit method satisfies $\beta_0 = 0$ and an implicit method of the same number of steps satisfies $\beta_s = 0$. The resolution for explicit methods is straightforward meanwhile for implicit methods the solution must be obtained by an iterative process.

Adams-Bashforth-Moulton methods consist on a pair of Bashforth and Moulton methods which are used in predictor-corrector configuration. A predictor-corrector scheme consists on, first a prediction of the solution at next step $\boldsymbol{u}_*^{n+1}$ obtained applying an explicit Adams-Bashforth which is used to evaluate $\boldsymbol{F}$ obtaining $\boldsymbol{F}_*^{n+1}$, to finally obtain $\boldsymbol{u}^{n+1}$ using the implicit scheme but avoiding the iterative resolution. These schemes can be written as:

$$\text{Prediction:} \quad \boldsymbol{u}_*^{n+1} = \boldsymbol{u}^n + \Delta t \sum_{j=1}^{s} \beta_j \boldsymbol{F}^{n+1-j}, \tag{4.35}$$

$$\text{Correction:} \quad \boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t \beta_0 \boldsymbol{F}_*^{n+1} + \Delta t \sum_{j=1}^{s-1} \beta_j \boldsymbol{F}^{n+1-j}. \tag{4.36}$$

The origin of the coefficients, and therefore of the methods, has its basis on approximating the quadrature

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \int_{t_n}^{t_{n+1}} \boldsymbol{F}(t, \boldsymbol{u}) \mathrm{d}t, \tag{4.37}$$

by means of an interpolant of grade $s-1$ (using s points) for $\boldsymbol{F}$. The interpolant $\boldsymbol{I}$ takes a different form depending if the scheme is explicit or implicit:

$$\text{Explicit:} \quad \boldsymbol{I}(t) = \sum_{1}^{s} \boldsymbol{F}^{n+1-j} \ell_{n+1-j}(t), \tag{4.38}$$

$$\text{Implicit:} \quad \boldsymbol{I}(t) = \sum_{0}^{s-1} \boldsymbol{F}^{n+1-j} \ell_{n+1-j}(t), \tag{4.39}$$

where $\ell_{n+1-j} : \mathbb{R} \longrightarrow \mathbb{R}$ are the Lagrange polynomials given by:

$$\text{Explicit:} \quad \ell_{n+1-j} = \prod_{\substack{k=n+1-s \\ k\neq n+1-j}}^{n} \frac{t - t_k}{t_{n+1-j} - t_k}, \tag{4.40}$$

$$\text{Implicit:} \quad \ell_{n+1-j} = \prod_{\substack{k=n+2-s \\ k\neq n+1-j}}^{n+1} \frac{t - t_k}{t_{n+1-j} - t_k}. \tag{4.41}$$

Hence, for a fixed step size Δt the coefficients are obtained by the approximation:

$$\begin{aligned} \text{Explicit:} \quad \boldsymbol{u}^{n+1} &\simeq \boldsymbol{u}^n + \int_{t_n}^{t_{n+1}} \boldsymbol{I}(t)\mathrm{d}t \\ &= \boldsymbol{u}^n + \sum_{1}^{s} \boldsymbol{F}^{n+1-j} \int_{t_n}^{t_{n+1}} \ell_{n+1-j}\mathrm{d}t, \end{aligned} \tag{4.42}$$

$$\begin{aligned} \text{Implicit:} \quad \boldsymbol{u}^{n+1} &\simeq \boldsymbol{u}^n + \int_{t_n}^{t_{n+1}} \boldsymbol{I}(t)\mathrm{d}t \\ &= \boldsymbol{u}^n + \sum_{0}^{s-1} \boldsymbol{F}^{n+1-j} \int_{t_n}^{t_{n+1}} \ell_{n+1-j}\mathrm{d}t, \end{aligned} \tag{4.43}$$

therefore the coefficients for both explicit and implicit schemes (choosing the appropriate interpolant) are written as:

$$\beta_j = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \ell_{n+1-j}\mathrm{d}t. \tag{4.44}$$

One interesting remark is that the coefficients are dependant on the step size distribution of the temporal grid. For this reason, it becomes very expensive to compute variable step size Adams methods.

Example 1. Two steps Adams-Bashforth. Let's consider the case $s = 2$ and constant Δt for an explicit method. In these conditions, the interpolant for the differential operator can be written:

$$\begin{aligned} \boldsymbol{I}(t) &= \boldsymbol{F}^n \ell_n(t) + \boldsymbol{F}^{n-1} \ell_{n-1}(t) \\ &= \boldsymbol{F}^n \frac{t - t_{n-1}}{\Delta t} - \boldsymbol{F}^{n-1} \frac{t - t_n}{\Delta t}, \end{aligned}$$

and the coefficients are calculated as:

$$\beta_1 = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \ell_n \mathrm{d}t = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \frac{t - t_{n-1}}{\Delta t} \mathrm{d}t = \frac{3}{2},$$

$$\beta_2 = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \ell_{n-1} \mathrm{d}t = -\frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \frac{t - t_n}{\Delta t} \mathrm{d}t = -\frac{1}{2},$$

leading to the scheme:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \frac{\Delta t}{2}\left(3\boldsymbol{F}^n - \boldsymbol{F}^{n-1}\right). \tag{4.45}$$

Example 2. Two steps Adams-Moulton. Let's consider the case $s = 2$ and constant Δt for an implicit method. In these conditions, the interpolant for the differential operator can be written:

$$\begin{aligned}\boldsymbol{I}(t) &= \boldsymbol{F}^{n+1}\ell_{n+1}(t) + \boldsymbol{F}^n\ell_n(t)\\ &= \boldsymbol{F}^{n+1}\frac{t-t_n}{\Delta t} - \boldsymbol{F}^n\frac{t-t_{n+1}}{\Delta t},\end{aligned}$$

and the coefficients are calculated as:

$$\begin{aligned}\beta_0 &= \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \ell_{n+1}\mathrm{d}t = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \frac{t-t_n}{\Delta t}\mathrm{d}t = \frac{1}{2},\\ \beta_1 &= \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \ell_n\mathrm{d}t = -\frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \frac{t-t_{n+1}}{\Delta t}\mathrm{d}t = \frac{1}{2},\end{aligned}$$

leading to the scheme:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \frac{\Delta t}{2}\left(\boldsymbol{F}^{n+1} + \boldsymbol{F}^n\right). \tag{4.46}$$

Example 3. Three steps Adams-Bashforth. Let's consider the case $s = 3$ and constant Δt for an explicit method. In these conditions, the interpolant for the differential operator can be written:

$$\boldsymbol{I}(t) = \boldsymbol{F}^n\ell_n(t) + \boldsymbol{F}^{n-1}\ell_{n-1}(t) + \boldsymbol{F}^{n-2}\ell_{n-2}(t),$$

where

$$\begin{aligned}\ell_n(t) &= \frac{(t-t_{n-1})(t-t_{n-2})}{2\Delta t^2},\\ \ell_{n-1}(t) &= -\frac{(t-t_n)(t-t_{n-2})}{\Delta t^2},\\ \ell_{n-2}(t) &= \frac{(t-t_n)(t-t_{n-1})}{2\Delta t^2},\end{aligned}$$

and the coefficients are calculated as:

$$\begin{aligned}\beta_1 &= \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \ell_n\mathrm{d}t = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \frac{(t-t_{n-1})(t-t_{n-2})}{2\Delta t^2}\mathrm{d}t = \frac{23}{12},\\ \beta_2 &= \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \ell_{n-1}\mathrm{d}t = -\frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \frac{(t-t_n)(t-t_{n-2})}{\Delta t^2}\mathrm{d}t = -\frac{16}{12},\\ \beta_3 &= \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \ell_{n-2}\mathrm{d}t = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \frac{(t-t_n)(t-t_{n-1})}{2\Delta t^2}\mathrm{d}t = \frac{5}{12},\end{aligned}$$

leading to the scheme:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \frac{\Delta t}{12}\left(23\boldsymbol{F}^n - 16\boldsymbol{F}^{n-1} + 5\boldsymbol{F}^{n-2}\right). \tag{4.47}$$

Note that if for each step Δt changed its value, that is $\Delta t_1 = t_n - t_{n-1}$, $\Delta t_2 = t_{n-1} - t_{n-2}$, the Lagrange polynomials would depend on these step sizes:

$$\begin{aligned}
\ell_n(t) &= \frac{(t-t_{n-1})(t-t_{n-2})}{\Delta t_1(\Delta t_1 + \Delta t_2)},\\
\ell_{n-1}(t) &= -\frac{(t-t_n)(t-t_{n-2})}{\Delta t_1 \Delta t_2},\\
\ell_{n-2}(t) &= \frac{(t-t_n)(t-t_{n-1})}{2\Delta t_2(\Delta t_1 + \Delta t_2)},
\end{aligned}$$

and therefore, for each step it would be necessary to calculate the coefficients which will depend on $(\Delta t_1, \Delta t_2)$.

In sight of the previous examples it is possible to obtain the coefficients for any desired value of s in a similar manner. However, in order to implement an algorithm that controls the step size, it would require to calculate the coefficents at each step of the simulation. This means a high computational cost for the algorithm in terms of computation time, which is undesirable.

The multi-value formulation will permit to reduce the computational and implementation cost associated to the obtention of the coefficients β_j of Adams methods.

For multivalue methods, instead of interpolating the differential operator, a truncated Taylor expansion is performed on the solution:

$$\tilde{\boldsymbol{u}}(t) = \sum_{j=0}^{s} \frac{\boldsymbol{u}_n^{j)}}{j!}(t-t_n)^j, \tag{4.48}$$

where is used the notation:

$$\boldsymbol{u}_n^{j)} = \left.\frac{\mathrm{d}^j \boldsymbol{u}}{\mathrm{d}t^j}\right|_{t_n} = \boldsymbol{u}_n^{\overbrace{\prime\prime\cdots\prime}^{j}}, \qquad \boldsymbol{u}_n^{0)} = \boldsymbol{u}_n.$$

From the expansion (4.48) it can be obtained the values of the s first derivatives of $\tilde{\boldsymbol{u}}$ on t_{n+1}. In general, the $i-$th derivative can be written:

$$\tilde{\boldsymbol{u}}_{n+1}^{i)} = \sum_{j=i}^{s} \frac{\boldsymbol{u}_n^{j)}}{(j-i)!}\Delta t^{j-i}, \tag{4.49}$$

where the same notation holds for $\tilde{\boldsymbol{u}}^{i)}_{n+1}$. For Nordsieck methods, instead of saving the values of the differential operator at the s steps, the values of the s derivatives are stored. In particular, the addends $\boldsymbol{y}^i_n = \Delta t^i \boldsymbol{u}^{i)}_n / i!$ of the expansion are considered in a new state vector in the following manner:

$$\tilde{\boldsymbol{y}}^i_{n+1} = \frac{\Delta t^i}{i!}\tilde{\boldsymbol{u}}^{i)}_{n+1} = \sum_{j=i}^{s} \frac{\boldsymbol{u}^{j)}_n}{i!(j-i)!}\Delta t^j = \sum_{j=i}^{s} \frac{j!}{i!(j-i)!}\boldsymbol{y}^j_n$$

hence, defining a matrix $B \in \mathcal{M}_{s+1\times s+1}$ as:

$$B_{ij} = \begin{cases} 0, & \text{if } j < i \\ \dfrac{j!}{i!(j-i)!}, & \text{if } j \geq i \end{cases}, \qquad i,j = 0,1,2\ldots,s,$$

we can write:

$$\tilde{\boldsymbol{y}}^i_{n+1} = \sum_{j=i}^{s} B_{ij}\boldsymbol{y}^j_n, \qquad i = 0,1,2\ldots,s. \tag{4.50}$$

The extrapolation $\tilde{\boldsymbol{y}}^i_{n+1}$ must be corrected by two parameters $\boldsymbol{\alpha} \in \mathbb{R}^{N_v}$ and r_i as:

$$\boldsymbol{y}^i_{n+1} = \tilde{\boldsymbol{y}}^i_{n+1} + r_i\boldsymbol{\alpha}, \qquad i = 0,1,2\ldots,s. \tag{4.51}$$

Note that (4.51) is equivalent to:

$$\boldsymbol{u}^{i)}_{n+1} = \tilde{\boldsymbol{u}}^{i)}_{n+1} + \frac{r_i i!}{\Delta t^i}\boldsymbol{\alpha}.$$

It has not yet defined the value for the coefficients r_i and $\boldsymbol{\alpha}$. This quantities will be obtained so the multi-value methods become equivalent to Adams methods as we will see in the next pages.

To obtain the coefficients for the multi-value method, let's consider the case $N_v = 1$, as once obtained for it, the case for $N_v > 1$ is straightforward. For this case we can write:

$$y^i_{n+1} = \tilde{y}^i_{n+1} + r_i\alpha, \qquad i = 0,1,2\ldots,s, \tag{4.52}$$

which is equivalent to:

$$u^{i)}_{n+1} = \tilde{u}^{i)}_{n+1} + \frac{i!}{\Delta t^i}r_i\alpha, \qquad i = 0,1,2\ldots,s. \tag{4.53}$$

The value of α is obtained by forcing that the solution for $i = 1$, that is, the derivative satisfies the differential equation. In other words, $u^{1)}_{n+1} = u'_{n+1} = F_{n+1} = F(t_{n+1}, u_{n+1})$. Also, as there are $s+2$ unknowns for $s+1$ equations we

shall fix $r_1 = 1$ for convenience. With this restriction α is determined from the second equation of (4.51) as:

$$\alpha = \Delta t F_{n+1} - \Delta t \tilde{u}'_{n+1} = \Delta t F_{n+1} - \Delta t \sum_{j=1}^{s} \frac{u_n^{j)}}{(j-1)!} \Delta t^{j-1}.$$

Note that imposing this value for α introduces the information of the differential equation to the multivalue method. The rest of the coefficients r_i can be obtained as:

$$\begin{aligned} r_i &= \frac{\Delta t^i}{i!} \frac{u_{n+1}^{i)} - \tilde{u}_{n+1}^{i)}}{\alpha} \\ &= \frac{\Delta t^{i-1}}{i!} \frac{u_{n+1}^{i)} - \tilde{u}_{n+1}^{i)}}{F_{n+1} - \tilde{u}_{n+1}^{1)}} \\ &= \frac{\Delta t^{i-1}}{i!} \frac{u_{n+1}^{i)} - \tilde{u}_{n+1}^{i)}}{F_{n+1} - \sum_{j=1}^{s} \frac{u_n^{j)}}{(j-1)!} \Delta t^{j-1}}. \end{aligned}$$

Notice that in this equation, for each r_i there is associated a value of $u_{n+1}^{i)}$. Therefore, the value of the coefficients can be fixed to satisfy that $u_{n+1}^{i)}$ comes from the $i-1$-th derivative of an interpolant $I(t)$, which is called $I^{i-1)}$. That is, imposing:

$$u_{n+1}^{i)} = \delta_{i0} u_n + I_{n+1}^{i-1)}, \qquad I_{n+1}^{i-1)} := I^{i-1)}(t_{n+1}), \qquad i = 0, 2, 3 \ldots, s, \tag{4.54}$$

where δ_{i0} is the delta Kronecker, and $I^{-1)}$ is defined as the integral:

$$I^{-1)}(t_{n+1}) = \int_{t_n}^{t_{n+1}} I \mathrm{d}t. \tag{4.55}$$

It has not been yet specified the stencil of the interpolant, which as we have seen. For $i > 1$ the interpolant must include the differential operator evaluated at the next step, that is $\boldsymbol{F}_{n+1}$. However, in the case $i = 0$ it depends on if the Adams method associated to the coefficient values is explicit or implicit. As has been stated previously, the interpolant takes a different form depending on this characteristic of the scheme. However, for Bashforth methods, the value obtained for u_{n+1} is exactly the value of the extrapolation, when the derivatives of u_n are computed as the derivatives (of one order less) of I in t_n, that is:

$$u_{n+1} = u_n + \int_{t_n}^{t_{n+1}} I \mathrm{d}t = u_n + \sum_{j=1}^{s} F_{n+1-j} \, \ell_{n+1-j}^{-1)}(t_{n+1}),$$

$$\begin{aligned}\tilde{u}_{n+1} &= u_n + \sum_{i=1}^{s} \frac{\Delta t^i}{i!} u_n^{i)} \\ &= u_n + \sum_{i=1}^{s} \frac{\Delta t^i}{i!} \left(\sum_{j=1}^{s} F_{n+1-j}\, \ell_{n+1-j}^{i-1)}(t_n) \right) \\ &= u_n + \sum_{j=1}^{s} F_{n+1-j} \left(\sum_{i=1}^{s} \frac{\Delta t^i}{i!} \ell_{n+1-j}^{i-1)}(t_n) \right).\end{aligned}$$

As for Adams Bashforth methods, it is satisfied that:

$$\ell_{n+1-j}^{-1)}(t_{n+1}) = \sum_{i=1}^{s} \frac{\Delta t^i}{i!} \ell_{n+1-j}^{i-1)}(t_n),$$

we have that the value at the next step u_{n+1} is the same as the extrapolation $\tilde{u}_{n+1}$, and $r_0 = 0$ for explicit Adams. Note that this can also be intuited from the equation:

$$r_i = \frac{u_{n+1}^{i)} - \tilde{u}_{n+1}^{i)}}{F_{n+1} - \tilde{u}'_{n+1}} \frac{\Delta t^{i-1}}{i!}.$$

For the case $i = 0$, r_0 represents the difference between the extrapolation $\tilde{u}_{n+1}$ and the solution given by the scheme u_{n+1}. This means that for implicit methods, it shall be obtained by determining u_{n+1} with the interpolant for implicit methods which we will call I and $\tilde{u}_{n+1}$ with the interpolant for explicit methods which we will call $\tilde{I}$. Their Lagrange polynomials respectively will be called $\tilde{\ell}_k$ and ℓ_k. Therefore, we can write r_0 for implicit methods as:

$$\begin{aligned}r_0 &= \frac{u_{n+1} - \tilde{u}_{n+1}}{F^{n+1} - \tilde{u}'_{n+1}} \frac{1}{\Delta t} \\ &= \frac{\sum_{j=0}^{s-1} F^{n+1-j} \ell_{n+1-j}^{-1)} - \sum_{j=1}^{s} F^{n+1-j} \tilde{\ell}_{n+1-j}^{-1)}}{F^{n+1} - \sum_{j=1}^{s} F^{n+1-j} \tilde{\ell}'_{n+1-j}} \frac{1}{\Delta t} \\ &= \beta_0 \frac{F^{n+1-j} + \sum_{j=1}^{s-1} F^{n+1-j} \beta_j/\beta_0 - \sum_{j=1}^{s} F^{n+1-j} \tilde{\beta}_j/\beta_0}{F^{n+1} - \sum_{j=1}^{s} F^{n+1-j} \tilde{\ell}'_{n+1-j}} \\ &= \beta_0. \end{aligned} \tag{4.56}$$

Hence, for Adams Moulton, the first coefficient r_0 for the multivalue expression is the same as the coefficient β_0 of the classical approach. To check the veracity of this claims, let's consider some examples.

Example 4. Two steps Adams Bashforth: For this method we have that

$$\ell_n^{-1)}(t_{n+1}) = \frac{3\Delta t}{2} = \beta_1 \Delta t, \qquad \ell'_n(t_n) = \frac{1}{\Delta t},$$
$$\ell_{n-1}^{-1)}(t_{n+1}) = -\frac{\Delta t}{2} = \beta_2 \Delta t, \qquad \ell'_{n-1}(t_n) = -\frac{1}{\Delta t},$$

and is satisfied that

$$\ell_n^{-1)}(t_{n+1}) = \sum_{i=1}^{2} \frac{\Delta t^i}{i!} \ell_n^{i-1)}(t_n) = \Delta t + \frac{\Delta t}{2} = \frac{3\Delta t}{2},$$
$$\ell_{n-1}^{-1)}(t_{n+1}) = \sum_{i=1}^{2} \frac{\Delta t^i}{i!} \ell_{n-1}^{i-1)}(t_n) = 0 - \frac{\Delta t}{2} = -\frac{\Delta t}{2},$$

hence, for two steps Adams Bashforth $r_0 = 0$ and $\tilde{u}_{n+1} = u_{n+1}$.

Example 5. Two steps Adams Moulton: For this method we have that

$$u_{n+1} = u_n + \frac{\Delta t}{2}\left(F^{n+1} + F^n\right),$$
$$\tilde{u}_{n+1} = u_n + \frac{\Delta t}{2}\left(3F^n - F^n\right),$$
$$\tilde{u}'_{n+1} = u'_n + u''_n \Delta t = F_n + \underbrace{F_n - F_{n-1}}_{\tilde{I}'_{n+1}} = 2F_n - F_{n-1},$$

And r_0 can be computed as:

$$r_0 = \frac{1}{2}\frac{F^{n+1} + F^n - 3F^n + F^{n-1}}{F^{n+1} - 2F_n + F_{n-1}} = \frac{1}{2} = \beta_0.$$

Whenever we want to determine the coefficients for $i > 1$ (for both explicit and implicit methods), the obtention of r_i must be done with an interpolant which includes the value of F at t_{n+1}. As the interpolant is of grade $s-1$, if we do not inclued the values at t_{n+1} the value of $u_{n+1}^{s)}$ would be the same as $u_n^{s)}$ which is incorrect. For this case we have:

$$u_{n+1}^{i)} = \delta_{i0} u_n + I_{n+1}^{i-1)} = \delta_{i0} u_n + \sum_{j=0}^{s-1} F_{n+1-j}\, \ell_{n+1-j}^{i-1)}(t_{n+1}),$$
$$u_{n+1}^{i)} = \sum_{j=i}^{s} \frac{u_n^{j)}}{(j-i)!} \Delta t^{j-i} + r_i \Delta t \left(F_{n+1} - \sum_{j=1}^{s} \frac{u_n^{j)}}{(j-1)!} \Delta t^{j-1} \right)$$

$$\begin{aligned} r_i &= \frac{\Delta t^{i-1}}{i!} \frac{u_{n+1}^{i)} - \tilde{u}_{n+1}^{i)}}{F_{n+1} - \tilde{u}'_{n+1}}, \\ &= \frac{\Delta t^{i-1}}{i!} \frac{\delta_{i0} u_n + \sum_{j=0}^{s-1} F_{n+1-j}\, \ell_{n+1-j}^{i-1)}(t_{n+1}) - \tilde{u}_{n+1}^{i)}}{F_{n+1} - \tilde{u}'_{n+1}} \\ &= \frac{\Delta t^{i-1}}{i!} \ell_{n+1}^{i-1)}(t_{n+1})\, \eta_i, \end{aligned}$$

where:

$$\eta_i = \frac{\dfrac{\delta_{i0}u_n}{\ell_{n+1}^{i-1)}(t_{n+1})} + \sum_{j=0}^{s-1} F_{n+1-j}\,\dfrac{\ell_{n+1-j}^{i-1)}(t_{n+1})}{\ell_{n+1}^{i-1)}(t_{n+1})} - \dfrac{\tilde{u}_{n+1}^{i)}}{\ell_{n+1}^{i-1)}(t_{n+1})}}{F_{n+1} - \tilde{u}'_{n+1}}$$
$$= 1. \tag{4.57}$$

And therefore, the coefficients can be written as:

$$r_0 = \begin{cases} 0, & \text{for explicit methods.} \\ \dfrac{\ell_{n+1}^{-1)}(t_{n+1})}{\Delta t} = \beta_0, & \text{for implicit methods.} \end{cases},$$

$$r_i = \frac{\Delta t^{i-1}}{i!}\ell_{n+1}^{i-1)}(t_{n+1}), \qquad \text{for } i > 1. \tag{4.58}$$

To see that indeed, (4.58) holds, let's consider an example.

Example 6. Two steps Adams For this method we have seen that for the explicit $r_0 = 0$ and to satisfy the differential equation $r_1 = 1$ for both explicit and implicit methods ($u'_{n+1} = F_{n+1}$). The only remaining coefficients are r_0 and r_2, we will see that both can be computed as given by (4.58).

$$\begin{aligned} \tilde{u}_{n+1} &= u_n + u'_n\Delta t + u''_n\Delta t^2/2, \\ \tilde{u}'_{n+1} &= u'_n + u''_n\Delta t, \\ \tilde{u}''_{n+1} &= u''_n, \end{aligned}$$

and we saw that $\ell'_{n+1}(t_{n+1}) = 1/\Delta t$, $\ell_{n+1}^{-1)}(t_{n+1}) = \Delta t/2$. As was explained the second derivative is computed as the first derivative of the interpolant I.

Therefore, we have that:

$$\begin{aligned}
\eta_0 &= \frac{\dfrac{u_n}{\ell^{-1)}_{n+1}(t_{n+1})} + \sum_{j=0}^{1} F_{n+1-j}\,\dfrac{\ell^{-1)}_{n+1-j}(t_{n+1})}{\ell^{-1)}_{n+1}(t_{n+1})} - \dfrac{\tilde{u}_{n+1}}{\ell^{-1)}_{n+1}(t_{n+1})}}{F_{n+1} - \tilde{u}'_{n+1}} \\
&= \frac{\sum_{j=0}^{1} F_{n+1-j}\,\dfrac{\ell^{-1)}_{n+1-j}(t_{n+1})}{\ell^{-1)}_{n+1}(t_{n+1})} - \dfrac{u'_n\Delta t + u''_n\Delta t^2/2}{\ell^{-1)}_{n+1}(t_{n+1})}}{F_{n+1} - \tilde{u}'_{n+1}} \\
&= 1. \\
\eta_2 &= \frac{\sum_{j=0}^{1} F_{n+1-j}\,\dfrac{\ell^{1)}_{n+1-j}(t_{n+1})}{\ell^{1)}_{n+1}(t_{n+1})} - \dfrac{\tilde{u}^{1)}_{n+1}}{\ell^{1)}_{n+1}(t_{n+1})}}{F_{n+1} - \tilde{u}'_{n+1}} \\
&= \frac{\sum_{j=0}^{1} F_{n+1-j}\,\dfrac{\ell^{1)}_{n+1-j}(t_{n+1})}{\ell^{1)}_{n+1}(t_{n+1})} - \dfrac{u'_n + u''_n\Delta t}{\ell^{1)}_{n+1}(t_{n+1})}}{F_{n+1} - \tilde{u}'_{n+1}} \\
&= 1.
\end{aligned}$$

And finally the coefficients can be calculated as:

$$r_0 = \ell^{-1)}_{n+1}(t_{n+1})/\Delta t = 1/2, \qquad r_2 = \ell^{1)}_{n+1}(t_{n+1})\Delta t/2 = 1/2.$$

Hence, we have seen that multi-value methods are equivalent to Adams methods. When formulating the multi-step methods in the form given by (4.51), the change of step size is very simple as if we calculate $\boldsymbol{y}^i_{n+1}$ for a given Δt_1, the solution for another step size Δt_2 is given by $\Delta t^i_2 \boldsymbol{y}^i_{n+1}/\Delta t^i_2$. The change of the method from Adams family is done by simply changing r_i. Besides, this formulation permits to write the multi-step methods on a more compact form defining two new state vectors $Y^n, U^n \in \mathcal{M}_{s+1\times N_v}$ given by

$$Y^n = \begin{bmatrix} \boldsymbol{y}^0_n \\ \vdots \\ \boldsymbol{y}^i_n \\ \vdots \\ \boldsymbol{y}^s_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_n \\ \vdots \\ \Delta t^i \boldsymbol{u}^{i)}_n/i! \\ \vdots \\ \Delta t^s \boldsymbol{u}^{s)}_n/s! \end{bmatrix} = AU^n, \tag{4.59}$$

where $A \in \mathcal{M}_{s+1\times s+1}$ is given by $A_{ij} = \delta_{ij}\Delta t^i/i!$. Thus, we can write:

$$\begin{aligned}
Y^{n+1} &= BY^n + \boldsymbol{r}\otimes\boldsymbol{\alpha}, \\
U^{n+1} &= A^{-1}BAU^n + A^{-1}\boldsymbol{r}\otimes\boldsymbol{\alpha},
\end{aligned} \tag{4.60}$$

where $\otimes : \mathbb{R}^n \times \mathbb{R}^m \longrightarrow \mathcal{M}_{n\times m}$ denotes the tensor product between real vector spaces. Note that for explicit methods, the computation of Y^{n+1} (or equivalently U^{n+1}) is straightforward as the computation of $\boldsymbol{y}^0_{n+1}$ is exactly $\tilde{\boldsymbol{y}}^0_{n+1}$ and can be decoupled from the rest. This means that $\boldsymbol{\alpha}$ is known from the initial conditions.

The previous circumstance is taken in advantage by *predictor-corrector multi-value* methods, which are also equivalent to predictor-corrector methods based on pairs of Adams-Bashforth and Adams-Moulton. For multi-value methods, they can be written as:

$$Y^{n+1} = Y^n + \boldsymbol{r} \otimes \tilde{\boldsymbol{\alpha}},$$

where $\tilde{\boldsymbol{\alpha}} = \Delta t F(\tilde{\boldsymbol{y}}^0_{n+1}, t_{n+1}) - \tilde{\boldsymbol{y}}^0_{n+1}$, and $\boldsymbol{r}$ is the coefficients vector. Note that the multi-value formulation permits to compute the Adams-Bashforth-Moulton methods and to easily change the method changing $\boldsymbol{r}$ and the step size scaling the solution U^{n+1} by a proper A^{-1}.

Chapter 5

Boundary Value Problems

5.1 Overview

In this chapter, the mathematical foundations of the boundary value problems are presented. Generally, these problems are devoted to find a solution of some scalar or vector function in a spatial domain. This solution is forced to comply some specific boundary conditions. The elliptic character of the solution of a boundary value problem means that the every point of the spatial domain is influenced by the whole points of the domain. From the numerical point of view, it means that the discretized solution is obtained by solving an algebraic system of equations. The algorithm and the implementation to obtain and solve the system of equations is presented.

Let $\Omega \subset \mathbb{R}^p$ be an open and connected set and $\partial\Omega$ its boundary set. The spatial domain D is defined as its closure, $D \equiv \{\Omega \cup \partial\Omega\}$. Each point of the spatial domain is written $\boldsymbol{x} \in D$. A Boundary Value Problem for a vector function $\boldsymbol{u} : D \to \mathbb{R}^N$ of N variables is defined as:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x})) = 0, \qquad \forall \ \ \boldsymbol{x} \in \Omega, \tag{5.1}$$

$$\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))\big|_{\partial\Omega} = 0, \qquad \forall \ \ \boldsymbol{x} \in \partial\Omega, \tag{5.2}$$

where $\boldsymbol{\mathcal{L}}$ is the spatial differential operator and $\boldsymbol{h}$ is the boundary conditions operator that must satisfy the solution at the boundary $\partial\Omega$.

5.2 Algorithm to solve Boundary Value Problems

If the spatial domain D is discretized in N_D points, the problem extends from vector to tensor, as a tensor system of equations of order p appears for each variable of $\boldsymbol{u}(\boldsymbol{x})$. The order of the tensor system merging from the complete system is $p+1$ and its number of elements is $N = N_v \times N_D$ where N_v is the number of variables of $\boldsymbol{u}(\boldsymbol{x})$. The number of points in the spatial domain N_D can be divided on inner points N_Ω and on boundary points $N_{\partial\Omega}$, satisfying: $N_D = N_\Omega + N_{\partial\Omega}$. Thus, the number of elements of the tensor system evaluated on the boundary points is $N_C = N_v \times N_{\partial\Omega}$. Once the spatial discretization is done, the system emerges as a tensor difference equation that can be rearranged into a vector system of N equations. Particularly two systems appear: one of $N - N_C$ equations from the differential operator on inner grid points and another of N_C equations from the boundary conditions on boundary points:

$$\begin{aligned} L(U) &= 0, \\ H(U)\big|_{\partial\Omega} &= 0 \end{aligned}$$

where $U \in \mathbb{R}^N$ comprises the discretized solution at inner points and boundary points. Notice that

$$L : \mathbb{R}^N \to \mathbb{R}^{N-N_C}$$

is the difference operator associated to the differential operator $\boldsymbol{\mathcal{L}}$ and

$$H : \mathbb{R}^N \to \mathbb{R}^{N_C}$$

is the difference operator associated to the boundary conditions operator $\boldsymbol{h}$. To solve the systems, both set of equations are packed in the vector function

$$F : \mathbb{R}^N \to \mathbb{R}^N,$$

with $F = [L, H]$ satisfying the differential equation and the boundary conditions

$$F(U) = 0.$$

The algorithm to solve this boundary value problem is explained in two steps:

1. Obtention of the vector function F.

   From a discretized solution U, derivatives are calculated and the differential equation is forced at inner grid points yielding N_Ω equations. Imposing the boundary conditions constraints, additional $N_{\partial\Omega}$ equations are obtained.

2. Solution of an algebraic system of equations.

   Once F is built, any available solver to obtain the solution of a system of equations is used.

The algorithm is represented schematically on figure 5.1. If the differential operator $\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u})$ and the boundary conditions $\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u})$ depend linearly with the dependent variable $\boldsymbol{u}(\boldsymbol{x})$, the problem is linear and the function F can be expressed by means of sytem matrix A and independent term b in the following form:

$$F = A\,U - b.$$

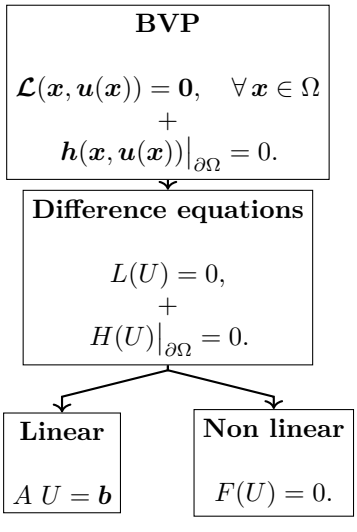

Figure 5.1: Linear and non linear boundary value problems.

5.3 From classical to modern approaches

This section is intended to consolidate the understanding of the procedure to implement the boundary value problem based on lower abstraction layers such as the finite differences layer. A nonlinear one-dimensional boundary value problem is considered to explain the algorithm. The following differential equation in the domain $x \in [-1, 1]$ is chosen:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + \sin u = 0,$$

along with boundary conditions:

$$u(-1) = 1, \qquad u(1) = 0.$$

The algorithm to solve this problem, based on second order finite differences formulas, consists on defining a equispaced mesh with Δx spatial size

$$\{x_i, \quad i = 0, \ldots, N\},$$

impose the discretized differential equations in these points

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + \sin u_i, \quad i = 1, \ldots N - 1,$$

and impose the boundary conditions

$$u_0 = 1, \quad u_N = 0.$$

Finally, these nonlinear $N + 1$ equations are solved.

However, this approach is tedious and requires extra analytical work if we change the equation or the order of the finite differences formulas. One of the main objectives of our `NumericalHUB` is to allow such a level of abstraction that these numerical details are hidden, focusing on more important issues related to the physical behavior or the numerical scheme error.

This abstraction level allows integrating this boundary value problem with different finite-difference orders or with different mathematical models by making a very low effort from the algorithm and implementation point of view. As it was mentioned, the solution of a boundary value problem requires three steps: select the grid distribution, impose the difference equations and solution of the resulting system. Let us try to do it from a general point of view.

1. **Grid points**.

   Define an equispaced or a nonuniform grid distribution of points x_i. This can be done by the following subroutine which determines the optimum distribution of points to minimize the truncation error depending on the degree or

order of the interpolation. If second order is considered, this distribution is uniform.

```
!  Grid points
   call Grid_Initialization("nonuniform", "x", Order, x)
```

Listing 5.1: API_Example_Finite_Differences.f90

This grid defines the approximate values u_i of the unknown $u(x)$ at the grid points x_i. Once the order is set and the grid points are given, Lagrange polynomials and their derivatives are built and particularized at the grid points x_i. These numbers are stored to be used as the coefficients of the finite differences formulas when calculating derivatives.

2. **Difference equations**.

   Once the grid is initialized and the derivative coefficients are calculated, the subroutine `Derivative` allows calculating the second derivative in every grid point x_i and their values are stores in `uxx`. Once all derivatives are expressed in terms of the nodal points u_i, the difference equations are built by means of the following equation:

```
! Difference equations
 function Equations(u) result(F)
    real, intent (in) :: u(0:)
    real :: F(0:size(u)-1)

    real :: uxx(0:Nx)

    call Derivative( "x", 2, u, uxx )

    F =  uxx + sin(6*u)   !  inner points

    F(0)  = u(0) - 1      ! B.C. at x = -1
    F(Nx) = u(Nx)         ! B.C. at x =  1

 end function
```

Listing 5.2: API_Example_Finite_Differences.f90

3. **Solution of a nonlinear system of equations**.

   The last step is to solve the resulting system of difference equations by means of a Newton-Raphson method:

```
    call Newton(Equations, u)
```

Listing 5.3: API_Example_Finite_Differences.f90

To have a complete image of the procedure presented before, the following subroutine BVP_FD implements the algorithm:

```
subroutine BVP_FD

    integer, parameter :: Nx = 40 ! grid points
    integer ::  Order = 6         ! finite differences order
    real :: x(0:Nx)               ! Grid distribution
    real :: u(0:Nx)               ! Solution u(x)

!   Spatial domain
    x(0) = -1; x(Nx) = +1

 !  Grid points
    call Grid_Initialization("nonuniform", "x", Order, x)

!   Initial guess
    u = 1

!   Newton solution
    call Newton(Equations, u)

!   Graph
    call qplot(x, u, Nx+1)
contains

! Difference equations
 function Equations(u) result(F)
    real, intent (in) :: u(0:)
    real :: F(0:size(u)-1)

    real :: uxx(0:Nx)

    call Derivative( "x", 2, u, uxx )

    F =  uxx + sin(6*u)   !  inner points

    F(0)  = u(0) - 1      ! B.C. at x = -1
    F(Nx) = u(Nx)         ! B.C. at x =  1

 end function

end subroutine
```

Listing 5.4: API_Example_Finite_Differences.f90

5.4 Overloading the Boundary Value Problem

Boundary value problems can be expressed in spatial domains $\Omega \subset \mathbb{R}^p$ with $d = 1, 2, 3$. From the conceptual point of view, there is no difference in the algorithm explained before. However, from the implementation point of view, tensor variables are of order $p + 1$ which makes the implementation slightly different. To make a user friendly interface for the user, the boundary value problem has been overloaded. It means that the subroutine to solve the boundary value problem is named `Boundary_Value_Problem` for all values of d and for different number of variables of $\boldsymbol{u}$. The overloading is done in the following code,

```
module Boundary_value_problems
 use Boundary_value_problems1D
 use Boundary_value_problems2D
 use Boundary_value_problems3D

 implicit none
 private
 public :: Boundary_Value_Problem ! It solves a boundary value problem

 interface Boundary_Value_Problem
      module procedure Boundary_Value_Problem1D,         &
                       Boundary_Value_Problem2D,         &
                       Boundary_Value_Problem2D_system,  &
                       Boundary_Value_Problem3D_system
 end interface
end module
```

Listing 5.5: `Boundary_value_problems.f90`

For example, if a scalar 2D problem is solved, the software recognizes automatically associated to the interface of $\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))$ and $\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))$ using the subroutine `Boundary_Value_Problem2D`. If the given interface of $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{h}$ does not match the implemented existing interfaces of the boundary value problem, the compiler will complain saying that this problem is not implemented. As an example, the following code shows the interface of 1D and 2D differential operators $\boldsymbol{\mathcal{L}}$

```
      real function DifferentialOperator1D(x, u, ux, uxx)
                       real, intent(in) :: x, u, ux, uxx
      end function
```

Listing 5.6: `Boundary_value_problems1D.f90`

```
    real function DifferentialOperator2D(x, y, u, ux, uy, uxx, uyy, uxy)
                     real, intent(in) :: x, y, u, ux, uy, uxx, uyy, uxy
    end function
```

Listing 5.7: `Boundary_value_problems2D.f90`

5.5 Linear and nonlinear BVP in 1D

For the sake of simplicity, the implementation of the algorithm provided below is only shown 1D problems. Once the program matches the interface of a 1D boundary value problem, the code will use the following subroutine:

```
subroutine Boundary_Value_Problem1D(                    &
                x_nodes, Differential_operator,         &
                Boundary_conditions, Solution, Solver )

    real, intent(in) :: x_nodes(0:)
    procedure (DifferentialOperator1D) :: Differential_operator
    procedure (BC1D) ::  Boundary_conditions
    real, intent(inout) :: Solution(0:)
    procedure (NonLinearSolver), optional:: Solver

    dU = Dependencies_BVP_1D( Differential_operator )

    linear1D = Linearity_BVP_1D( Differential_operator )

    if (linear1D) then

      call Linear_Boundary_Value_Problem1D( x_nodes,                  &
           Differential_operator, Boundary_conditions, Solution)
    else

      call Non_Linear_Boundary_Value_Problem1D( x_nodes,               &
           Differential_operator, Boundary_conditions, Solution, Solver)
    end if

end subroutine
```

Listing 5.8: `Boundary_value_problems1D.f90`

Depending on the linearity of the problem, the implementation differs. For this reason and in order to classify between linear and nonlinear problem, the subroutine `Linearity_BVP_1D` is used. Besides and in order to speed up the calculation, the subroutine `Dependencies_BVP_1D` checks if the differential operator $\mathcal{L}$ depends on first or second derivative of $u(x)$. If the differential operator does not depend on the first derivative, only second derivative will be calculated to build the difference operator. The same applies if no dependency on the second derivative is encountered.

Once the problem is classified into a linear or a nonlinear problem and dependencies of derivatives are determined, the linear or nonlinear subroutine is called in accordance.

5.6 Non Linear Boundary Value Problems in 1D

The following subroutine `Nonlinear_boundary_Problem1d` solves the problem

```
subroutine Non_Linear_Boundary_Value_Problem1D( x_nodes,            &
                        Differential_operator,  Boundary_conditions, &
                        Solution, Solver)

     real, intent(in) :: x_nodes(0:)
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     real, intent(inout) :: Solution(0:)
     procedure (NonLinearSolver), optional:: Solver

!  *** Nummber of grid points
       integer ::  N
       N = size(x_nodes) - 1

!  *** Non linear solver
       if (present(Solver))  then
                    call Solver(BVP_discretization, Solution)
       else
                    call Newton(BVP_discretization, Solution)
       end if
contains
```

Listing 5.9: `Boundary_value_problems1D.f90`

As it was mentioned, the algorithm to solve a BVP comprises two steps:

1. Construction of the difference operator or system of nonlinear equations.

   The system of nonlinear equations is built in `BVP_discretization`. Notice that the interface of the vector function that uses the Newton-Raphson method must be $F : \mathbb{R}^N \to \mathbb{R}^N$. This requirement must be taken into account when implementing the function `BVP_discretization`.

2. Resolution of the nonlinear system of equations by a specific method.

   This subroutine checks if the `Solver` is present. If not, the code uses a classical Newton-Raphson method to obtain the solution. Since any available and validated `Solver` can be used, no further explanations to the resolution method will be made.

The function `BVP_discretization` is implemented by the following code:

```
function BVP_discretization(U) result(F)
           real, intent(in) ::  U(0:)
           real :: F(0:size(U)-1)

    integer :: i
    real :: D, C
    real :: Ux(0:N), Uxx(0:N)


        if (dU(1)) call Derivative( "x", 1, U, Ux)
        if (dU(2)) call Derivative( "x", 2, U, Uxx)

!  ***  boundary conditions
        do i=0, N, N
             D = Differential_operator( x_nodes(i), U(i), Ux(i), Uxx(i) )
             C = Boundary_conditions( x_nodes(i), U(i), Ux(i) )
             if (C == FREE_BOUNDARY_CONDITION) then
                 F(i) = D
             else
                 F(i) = C
             end if
         end do

!  *** inner grid points
        do i=1, N-1
            F(i) = Differential_operator( x_nodes(i), U(i), Ux(i), Uxx(i)
                )
        enddo

end function
```

Listing 5.10: `Boundary_value_problems1D.f90`

First, the subroutine calculates only the derivatives appearing in the problem. Second, the boundary conditions at x_0 and x_N are analyzed. If there is no imposed boundary condition (`C == FREE_BOUNDARY_CONDITION`), the corresponding equation is taken from the differential operator. On the contrary, the equation represents the discretized boundary condition.

Once the boundary conditions are discretized, the differential operator is discretized for the inner grid points $x_1, \ldots, x_{n-1}$.

Once the complete systems of N equations is built, the Newton-Raphson method or the optional `Solver` is used to obtain the solution of the boundary value problem.

5.7 Linear Boundary Value Problems in 1D

The implementation of a linear BVP is more complex than the implementation of a nonlinear BVP. However, the computational cost of the linear BVP can be lower. The idea behind the implementation of the linear BVP relies on the expression of the resulting system of equations. If the system is linear,

$$F(U) = A\ U - b, \tag{5.3}$$

where A is the matrix of the linear system of equations and b is the independent term.

The algorithm proposed to obtain A is based on N successive evaluations of $F(U)$ to obtain the matrix A and the vector b. To determine the vector b, the function in equation 5.3 is evaluated with $U = 0$

$$F(0) = -b.$$

To determine the first column of matrix A, the function in equation 5.3 is evaluated with $U^1 = [1, 0, \ldots, 0]^T$

$$F(U^1) = A\ U^1 - b$$

The components of $F(U^1)$ are $A_{i1} - b_i$. Since the independent term b is known, the first column of the matrix A is

$$C_1 = F(U^1) + b$$

Proceeding similarly with another evaluation of F with $U^2 = [0, 1, 0, \ldots, 0]^T$, the second column of A is obtained

$$C_2 = F(U^2) + b.$$

In this way, the columns of the matrix A are determined by means of $N + 1$ evaluations of the function F.

Once the matrix A and the independent term b are obtained, any validated subroutine to solve linear systems can be used. This algorithm to solve the BVP based on the determination of A and b is implemented in the following subroutine called `Linear_Boundary_Valu_Problem1D`:

```
subroutine Linear_Boundary_Value_Problem1D( x_nodes,                    &
                                            Differential_operator,      &
                                            Boundary_conditions, Solution)
     real, intent(in) :: x_nodes(0:)
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     real, intent(out) :: Solution(0:)

!  *** auxiliary variables
       integer ::  i, N
       real, allocatable ::  b(:), U(:), A(:,:)

!  *** Integration domain
       N = size(x_nodes) - 1
       allocate( b(0:N), U(0:N), A(0:N, 0:N) )

!  *** independent term  A U = b  ( U = inverse(A) b )
       U = 0
       b = BVP_discretization(U)

!  *** Kronecker delta to calculate the difference operator
       do i=0, N
          U(0:N) = 0
          U(i) = 1.0
          A(0:N, i) = BVP_discretization(U) - b
       enddo

!  *** solve the linear system of equations
       call LU_Factorization(A)

       Solution = Solve_LU( A, -b )

       deallocate( U, A, b )

contains
```

Listing 5.11: `Boundary_value_problems1D.f90`

The function `BVP_discretization` gives the components of the function F and it is the same function that is described in the nonlinear problem.

At the beginning of this subroutine, the independent term `b` and the matrix `A` are calculated. Then, the linear system is solved by means of a $L\,U$ factorization.

Chapter 6

Initial Boundary Value Problems

6.1 Overview

In this chapter, an algorithm and the implementation of initial boundary value problems will be presented. From the physical point of view, an initial boundary value problem represents an evolution problem in a spatial domain in which some constraints associated to the boundaries of the domain must be verified.

From the mathematical point of view, it can be defined as follows. Let $\Omega \subset \mathbb{R}^p$ be an open and connected set, and $\partial\Omega$ its boundary set. The spatial domain D is defined as its closure, $D \equiv \{\Omega \cup \partial\Omega\}$. Each element of the spatial domain is called $\boldsymbol{x} \in D$. The temporal dimension is defined as $t \in \mathbb{R}$.

An Initial Boundary Value Problem for a vector function $\boldsymbol{u} : D \times \mathbb{R} \to \mathbb{R}^{N_v}$ of N_v variables is defined as:

$$
\begin{aligned}
&\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x}, t) = \boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t)), && \forall \ \ \boldsymbol{x} \in \Omega, \\
&\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t))\big|_{\partial\Omega} = 0, && \forall \ \ \boldsymbol{x} \in \partial\Omega, \\
&\boldsymbol{u}(\boldsymbol{x}, t_0) = \boldsymbol{u}_0(\boldsymbol{x}), &&
\end{aligned}
$$

where $\boldsymbol{\mathcal{L}}$ is the spatial differential operator, $\boldsymbol{u}_0(\boldsymbol{x})$ is the initial value and $\boldsymbol{h}$ is the boundary conditions operator for the solution at the boundary points $\boldsymbol{u}\big|_{\partial\Omega}$.

6.2 Algorithm to solve IBVPs

If the spatial domain D is discretized in N_D points, the problem extends from vector to tensor, as a tensor system of equations of order p appears from each variable of $\boldsymbol{u}$. The order of the complete tensor system is $p+1$ and its number of elements N is $N = N_v \times N_D$. The number of points in the spatial domain N_D can be divided on inner points N_Ω and on boundary points $N_{\partial\Omega}$, satisfying: $N_D = N_\Omega + N_{\partial\Omega}$. Thus, the number of elements of the tensor system evaluated on the boundary points is $N_C = N_v \times N_{\partial\Omega}$. Once the spatial discretization is done, even though the system emerges as a tensor Cauchy Problem, it can be rearranged into a vector system of N equations. Particularly, two systems of equations appear: one of $N - N_C$ ordinary differential equations and N_C algebraic equations related to the boundary conditions. These equations can be expressed in the following way:

$$\frac{\mathrm{d}U_\Omega}{\mathrm{d}t} = F(U;t), \qquad H(U;t)\big|_{\partial\Omega} = 0,$$

$$U(t_0) = U^0,$$

where $U \in \mathbb{R}^N$ is at inner and boundary points, $U_\Omega \in \mathbb{R}^{N-N_C}$ is the solution at inner point, $U\big|_{\partial\Omega} \in \mathbb{R}^{N_C}$ is the solution at boundary points, $U^0 \in \mathbb{R}^N$ is the discretized initial value, $F : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^{N-N_C}$ is the difference operator associated to the differential operator and $H : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^{N_C}$ is the difference operator of the boundary conditions.

Hence, once the spatial discretization is carried out, the resulting problem comprises a system of $N - N_C$ first order ordinary differential equations and N_C algebraic equations. This differential-algebraic system of equations (DAEs) contains differential equations (ODEs) and algebraic equations are generally more difficult to solve than ODEs. Since the algebraic equations must be verified for all time, the algorithm to solve an initial boundary value problem comprises the following three steps:

1. Determination of the solution at boundaries.

   If the initial condition or the values U_Ω at a given t_n are given, boundary conditions can be discretized at boundaries. The number of the discretized equations must be the number of the unknowns $U_{\partial\Omega}$ at boundaries. In these equations the inner points act as forcing term or a parameter.

2. Spatial discretization of the differential operator at inner points.

   Once inner and boundary values U_D are known, the spatial discretization at inner points allows building a system of ODEs for the valuesof the inner points.

3. Temporal step to update the evolving solution.

Once the vector function is known, a validated temporal scheme is used to determine the next time step.

The sequence of the algorithm is represented in figure 6.1. This algorithm is called method of lines.

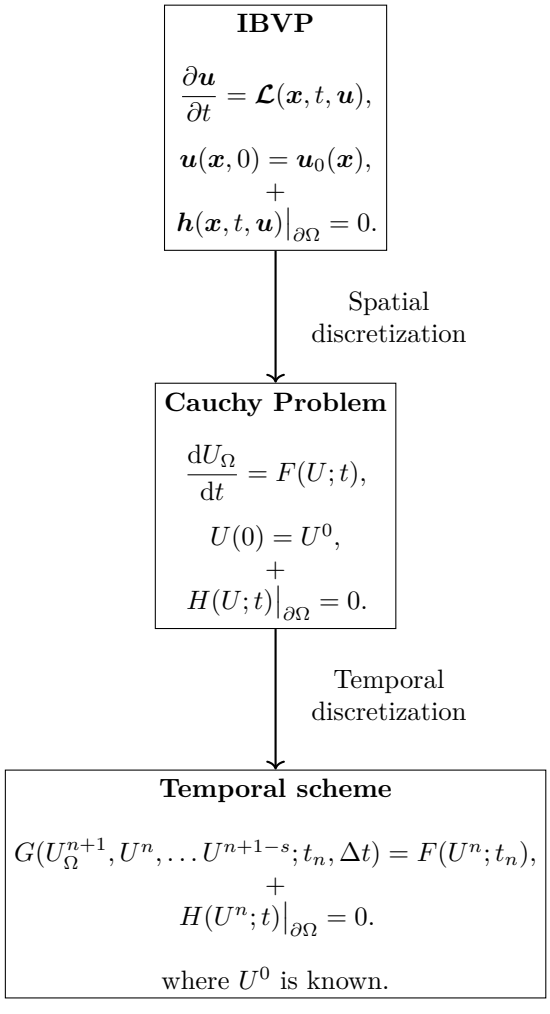

Figure 6.1: Line method for initial value boundary problems.

6.3 From classical to modern approaches

This section is intended to show to advantages of implementing with a high level of abstraction avoiding tedious implementations, sources of errors and misleading results. To explain the algorithm and the different levels of abstraction when implementing a programming code, a one-dimensional initial boundary value problem is considered. The 1D heat equation with the following initial and boundary conditions in the domain $x \in [-1, 1]$ is chosen:

$$\begin{aligned} &\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \\ &u(x, 0) = 0, \\ &u(-1, t) = 1, \qquad \frac{\partial u}{\partial x}(1, t) = 0. \end{aligned}$$

The algorithm to solve this problem, based on second order finite differences formulas, consists on defining a equispaced mesh with Δx spatial size

$$\{x_i, \quad i = 0, \ldots, N\}.$$

If $u_i(t)$ denotes the approximate value of the function $u(x, t)$ at the nodal point x_i, the partial differential equation is imposed in these points by expressing their spatial derivatives with finite difference formulas

$$\frac{du_i}{dt} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}, \quad i = 1, \ldots, N-1.$$

The discretized boundary conditions are also imposed by:

$$\begin{aligned} &u_0(t) = 1, \\ &\frac{1}{2\Delta x}(3\ u_N - 4\ u_{N-1} + u_{N-2}) = 0. \end{aligned}$$

These equations constitute a differential-algebraic set of equations (DAEs). There are two algebraic associated to the boundary conditions and $N-1$ evolution equations governing the temperature of the inner points $i = 1, \ldots, N-1$. Finally, a temporal scheme such as the Euler method should be used to determine the evolution in time. If u_j^n denotes the approximate value of $u(x, t)$ at the point x_i and the instant t_n, the following difference set of equations governs the evolution of the temperature

$$u_0^n = 1,$$

$$u_N^n = \frac{4}{3}\ u_{N-1}^n - \frac{1}{3}\ u_{N-2}^n$$

$$u_i^n = u_i^n + \frac{\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \quad i = 1, \ldots, N-1.$$

The above approach is tedious and requires extra analytical work if we change the equation, the temporal scheme or the order of the finite differences formulas. One of the main objectives of our `NumericalHUB` is to allow such a level of abstraction that these numerical details are hidden, focusing on more important issues related to the physical behavior or the numerical scheme error. This abstraction level allows integrating any initial boundary value problem with different finite-difference orders or with different temporal schemes by making a very low effort from the algorithm and implementation point of view.

The following abstraction levels will be considered when implementing the solution of the discretized initial boundary value problem:

1. Since Fortran is vector language, the solution can be obtained by performing vector operations. That is, U^n is a vector whose components are the scalar values u_i^n.
$$U^{n+1} = U^n + A\ U^n, \quad n = 0, \ldots$$

2. To decouple the spatial discretization form the temporal discretization and to allow reusing the spatial discretizations effort with different temporal schemes, a vector function can be defined to hold the spatial discretization. That is, $F : \mathbb{R}^{N+1} \to \mathbb{R}^{N+1}$ whose components are the equations resulting of the spatial semi-discretization.
$$U^{n+1} = U^n + \Delta t\ F(U^n), \quad n = 0, \ldots$$

3. To reuse the implementation of a complex and validated temporal scheme, a common interface of temporal schemes can be defined to deal with a first order Cauchy problem. That is, a temporal scheme can be a subroutine which gives the next time step of the vector `U2` from the initial vector `U1` and the vector function $F(U)$

```
call Temporal_scheme( F, U1, U2 )
```

4. To reuse the implementation of a complex and validated spatial discretization, a common interface of spatial derivatives can be defined to deal with a partial differential equations written as a second order systems in space. That is, a derivative subroutine can be defined to give the first or second order derivative from a set of nodal points `U`. The results can be held in the vector `Ux`. For example, to calculate the first order derivative of `U` in the `"x"` direction

```
call Derivative( "x", 1, U, Ux )
```

   With these different levels of abstraction, modern Fortran programming becomes reliable, reusable and easy to maintain.

6.4 Overloading the IBVP

Initial boundary value problems can be expressed in spatial domains $\Omega \subset \mathbb{R}^p$ with $d = 1, 2, 3$. From the conceptual point of view, there is no difference in the algorithm explained before. However, from the implementation point of view, tensor variables are of order $p+1$ which makes the implementation slightly different. To make a user friendly interface for the user, the initial boundary value problem has been overloaded. It means that the subroutine to solve the boundary value problem is named `Initial_Boundary_Value_Problem` for all values of d and for different number of variables of $\boldsymbol{u}$. The overloading is implemented in the following code,

```
module Initial_Boundary_Value_Problems
 use Initial_Boundary_Value_Problem1D
 use Initial_Boundary_Value_Problem2D
 use Utilities
 use Temporal_Schemes
 use Finite_differences

 implicit none
 private
 public :: Initial_Boundary_Value_Problem

 interface  Initial_Boundary_Value_Problem
        module procedure IBVP1D, IBVP1D_system, IBVP2D, IBVP2D_system
 end interface

end module
```

Listing 6.1: `Initial_Boundary_value_problems.f90`

For example, if a scalar 2D problem is solved, the software recognizes automatically associated to the interface of $\boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}))$ and $\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}))$. If the given interface of $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{h}$ does not match the implemented existing interfaces, the compiler will complain saying that this problem is not implemented. As an example, the following code shows the interface of 1D and 2D differential operators $\boldsymbol{\mathcal{L}}$

```
      real function DifferentialOperator1D(x, t, u, ux, uxx)
                       real, intent(in) :: x, t, u, ux, uxx
      end function
```

Listing 6.2: `Initial_Boundary_Value_Problem1D.f90`

```
  real function DifferentialOperator2D(x, y, t, u, ux, uy, uxx, uyy, uxy)
                   real, intent(in) :: x, y, t, u, ux, uy, uxx, uyy, uxy
  end function
```

Listing 6.3: `Initial_Boundary_value_problem2D.f90`

6.5 Initial Boundary Value Problem in 1D

For the sake of simplicity, the implementation of the algorithm provided below is only shown 1D problems. Once the program matches the interface of a 1D boundary value problem, the code will use the following subroutine:

```
subroutine IBVP1D( Time_Domain, x_nodes,  Differential_operator,  &
                   Boundary_conditions, Solution, Scheme)

     real, intent(in) :: Time_Domain(:)
     real, intent(inout) :: x_nodes(0:)
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     real, intent(out) :: Solution(0:,0:)
     procedure (Temporal_Scheme), optional :: Scheme

     real :: t_BC
     logical :: dU(2) ! matrix of dependencies( order )
     integer :: Nx, Nt, it

     Nx = size(x_nodes) - 1
     Nt = size(Time_Domain) - 1

     dU = Dependencies_IBVP_1D( Differential_operator )

    call Cauchy_ProblemS( Time_Domain, Space_discretization, &
                          Solution, Scheme                   )

contains

function Space_discretization( U, t ) result(F)
          real ::  U(:), t
          real :: F(size(U))

         call Space_discretization1D( U, t, F )


end function
```

Listing 6.4: `Initial_Boundary_Value_Problem1D.f90`

In order to speed up the calculation, the subroutine `Dependencies_IBVP_1D` checks if the differential operator $\mathcal{L}$ depends on first or second derivative of $u(x)$. If the differential operator does not depend on the first derivative, only second derivative will be calculated to build the difference operator. The same applies if no dependency on the second derivative is encountered.

The spatial discretization is done by the subroutine `Space_discretization1D` and it is implemented in the following code:

```
subroutine Space_discretization1D( U, t, F )
          real ::  U(0:Nx), t, F(0:Nx)

    integer :: k, N_int
    real :: Ux(0:Nx), Uxx(0:Nx)
    real, allocatable :: Ub(:)

        t_BC = t
        call Binary_search(t_BC, Time_Domain, it)
        Solution(it, :)  = U(:)

!  ***  It solves one or two equations to yield values at  boundaries
        if (Boundary_conditions( x_nodes(0),  t_BC, 1.,  2.) ==       &
                                 PERIODIC_BOUNDARY_CONDITION) then
            N_int = Nx
            U(0)  = U(Nx)

       else if (Boundary_conditions( x_nodes(Nx),  t_BC, 1.,  2.) ==  &
                                     FREE_BOUNDARY_CONDITION) then
            N_int = Nx
            allocate( Ub(1) )
            Ub = [ U(0) ]
            call Newton( BCs1, Ub )
            U(0)  = Ub(1)

       else
            N_int = Nx-1
            allocate( Ub(2) )
            Ub = [ U(0), U(Nx) ]
            call Newton( BCs2, Ub )
            U(0)  = Ub(1)
            F(0) = U(0) - Ub(1)
            U(Nx)  = Ub(2)
            F(Nx) = U(Nx) - Ub(2)

        end if
!  *** inner grid points
        if (dU(1)) call Derivative( "x", 1, U, Ux)
        if (dU(2)) call Derivative( "x", 2, U, Uxx)

        do k=1, N_int
         F(k) = Differential_operator(x_nodes(k), t, U(k), Ux(k), Uxx(k))
        enddo

        if (allocated(Ub)) deallocate( Ub )
end subroutine
```

Listing 6.5: `Initial_Boundary_Value_Problem1D.f90`

As it was mentioned, the algorithm to solve a IBVP has to deal with differential-algebraic equations. Hence, any time the vector function is evaluated by the temporal scheme, boundary values are determined by solving a linear or nonlinear system of equations involving the unknowns at the boundaries. Once these values are known, the inner components of $F(U)$ are evaluated.

Chapter 7

Mixed Boundary and Initial Value Problems

7.1 Overview

In the present chapter, the numerical resolution and implementation of the initial boundary value problem for an unknown variable $\boldsymbol{u}$ coupled with an elliptic problem for another unknown variable $\boldsymbol{v}$ are considered. Prior to the numerical resolution of the problem by means of an algorithm, a brief mathematical presentation must be given.

Evolution problems coupled with elliptic problems are common in applied physics. for example, when considering an incompressible flow, the information travels in the fluid at infinite velocity. This means that the pressure adapts instantaneously to the change of velocities. From the mathematical point of view, it means that the pressure is governed by an elliptic equation. Hence, the velocity and the temperature of fluid evolve subjected to the pressure field which adapts instantaneously to velocity changes.

The chapter shall be structured in the following manner. First, the mathematical presentation and both spatial and temporal discretizations will be described. Then, an algorithm to solve the discretized algebraic problem is presented. Finally, the implementation of this algorithm is explained. Thus, the intention of this chapter is to show how these generic problems can be implemented and solved form an elegant and mathematical point of view using modern Fortran.

Let $\Omega \subset \mathbb{R}^p$ be an open and connected set, and $\partial\Omega$ its boundary. The spatial domain D is defined as its closure, $D \equiv \{\Omega \cup \partial\Omega\}$. Each element of the spatial domain is called $\boldsymbol{x} \in D$. The temporal dimension is defined as $t \in \mathbb{R}$.

The intention of this section is to state an evolution problem coupled with a boundary value. The unknowns of the problem are two following vector functions:

$$\boldsymbol{u} : D \times \mathbb{R} \to \mathbb{R}^{N_u}$$

of N_u variables and

$$\boldsymbol{v} : D \times \mathbb{R} \to \mathbb{R}^{N_v}$$

of N_v variables. These functions are governed by the following set of equations:

$$\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x}, t) = \boldsymbol{\mathcal{L}}_u(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t), \boldsymbol{v}(\boldsymbol{x}, t)), \qquad \forall\ \boldsymbol{x} \in \Omega, \tag{7.1}$$

$$\boldsymbol{h}_u(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t))\big|_{\partial\Omega} = 0, \qquad \forall\ \boldsymbol{x} \in \partial\Omega, \tag{7.2}$$

$$\boldsymbol{u}(\boldsymbol{x}, t_0) = \boldsymbol{u}_0(\boldsymbol{x}), \qquad \forall\ \boldsymbol{x} \in D, \tag{7.3}$$

$$\tag{7.4}$$

$$\boldsymbol{\mathcal{L}}_v(\boldsymbol{x}, t, \boldsymbol{v}(\boldsymbol{x}, t), \boldsymbol{u}(\boldsymbol{x}, t)) = 0, \qquad \forall\ \boldsymbol{x} \in \Omega, \tag{7.5}$$

$$\boldsymbol{h}_v(\boldsymbol{x}, t, \boldsymbol{v}(\boldsymbol{x}, t))\big|_{\partial\Omega} = 0, \qquad \forall\ \boldsymbol{x} \in \partial\Omega, \tag{7.6}$$

where $\boldsymbol{\mathcal{L}}_u$ is the spatial differential operator of the initial boundary value problem of N_u equations, $\boldsymbol{u}_0(\boldsymbol{x})$ is the initial value, $\boldsymbol{h}_u$ is the boundary conditions operator for the solution at the boundary points $\boldsymbol{u}\big|_{\partial\Omega}$, $\boldsymbol{\mathcal{L}}_v$ is the spatial differential operator of the boundary value problem of N_v equations and $\boldsymbol{h}_v$ is the boundary conditions operator for $\boldsymbol{v}$ at the boundary points $\boldsymbol{v}\big|_{\partial\Omega}$.

It can be seen that both problems are coupled by the differential operators since these operators depend on both variables. The order in which appear in the differential operators $\boldsymbol{u}$ and $\boldsymbol{v}$ indicates its number of equations, for example: $\boldsymbol{\mathcal{L}}_v(\boldsymbol{x}, t, \boldsymbol{v}, \boldsymbol{u})$ and $\boldsymbol{v}$ are of the same size as it appears first in the list of variables from which the operator depends on.

It can also be observed that the initial value for $\boldsymbol{u}$ appears explicitly, while there is no initial value expression for $\boldsymbol{v}$. This is so, as the problem must be interpreted in the following manner: for each instant of time t, $\boldsymbol{v}$ is such that verifies $\boldsymbol{\mathcal{L}}_v(\boldsymbol{x}, t, \boldsymbol{v}, \boldsymbol{u}) = 0$ in which $\boldsymbol{u}$ acts as a known vector field for each instant of time. This interpretation implies that the initial value $\boldsymbol{v}(\boldsymbol{x}, t_0) = \boldsymbol{v}_0(\boldsymbol{x})$, is the solution of the problem $\boldsymbol{\mathcal{L}}_v(\boldsymbol{x}, t_0, \boldsymbol{v}_0, \boldsymbol{u}_0) = 0$. This means that the initial value for $\boldsymbol{v}$ is given implicitly in the problem. Hence, the solutions must verify both operators and boundary conditions at each instant of time, which forces the resolution of them to be simultaneous.

7.2 Algorithm to solve a coupled IBVP-BVP

If the spatial domain D is discretized in N_D points, both problems extend from vectors to tensors, as a tensor system of equations of order p appears from each variable of $\boldsymbol{u}$ and $\boldsymbol{v}$. The order of the tensor system for $\boldsymbol{u}$ and $\boldsymbol{v}$ is $p+1$.

The number of elements for both are respectively: $N_{e,u} = N_u \times N_D$ and $N_{e,v} = N_v \times N_D$. The number of points in the spatial domain N_D can be divided on inner points N_Ω and on boundary points $N_{\partial\Omega}$, satisfying: $N_D = N_\Omega + N_{\partial\Omega}$. Thus, the number of elements of each tensor system evaluated on the boundary points are $N_{C,u} = N_u \times N_{\partial\Omega}$ and $N_{C,v} = N_v \times N_{\partial\Omega}$.

Once the spatial discretization is done, the initial boundary value problem and the boundary value problem transform. The differential operator for $\boldsymbol{u}$ emerges as a tensor Cauchy Problem of $N_{e,u} - N_{C,u}$ elements, and its boundary conditions as a difference operator of $N_{C,u}$ equations. The operator for $\boldsymbol{v}$ is transformed into a tensor difference equation of $N_{e,v} - N_{C,v}$ elements and its boundary conditions in a difference operator of $N_{C,v}$ equations. Notice that even though they emerge as tensors is indifferent to treat them as vectors as the only difference is the arrange between of the elements which conform the systems of equations. Thus, the spatially discretized problem can be written:

$$\frac{\mathrm{d}U_\Omega}{\mathrm{d}t} = F_U(U,V;t), \qquad H_U(U;t)\big|_{\partial\Omega} = 0,$$

$$U(t_0) = U^0,$$

$$F_V(U,V;t) = 0, \qquad H_V(V;t)\big|_{\partial\Omega} = 0,$$

where $U \in \mathbb{R}^{N_{e,u}}$ and $V \in \mathbb{R}^{N_{e,u}}$ are the solutions comprising inner and boundary points, $U_\Omega \in \mathbb{R}^{N_{e,u}-N_{C,u}}$ is the solution of inner points, $U\big|_{\partial\Omega} \in \mathbb{R}^{N_{C,u}}$ and $V\big|_{\partial\Omega} \in \mathbb{R}^{N_{C,v}}$ are the solutions at the boundary points, $U^0 \in \mathbb{R}^{N_{e,u}}$ is the discretized initial value, the difference operators associated to both differential operators are,

$$F_U : \mathbb{R}^{N_{e,u}} \times \mathbb{R}^{N_{e,v}} \times \mathbb{R} \to \mathbb{R}^{N_{e,u}-N_{C,u}},$$

$$F_V : \mathbb{R}^{N_{e,v}} \times \mathbb{R}^{N_{e,u}} \times \mathbb{R} \to \mathbb{R}^{N_{e,v}-N_{C,v}},$$

and

$$H_U : \mathbb{R}^{N_{e,u}} \times \mathbb{R} \to \mathbb{R}^{N_{C,u}},$$

$$H_V : \mathbb{R}^{N_{e,v}} \times \mathbb{R} \to \mathbb{R}^{N_{C,v}},$$

are the difference operators of the boundary conditions.

Hence, the resolution of the problem requires solving a Cauchy problem and algebraic systems of equations for the discretized variables U and V. To solve the Cauchy Problem, the time is discretized in $t = t_n \boldsymbol{e}_n$. The term $n \in \mathbb{Z}$ is the index of every temporal step that runs over $[0, N_t]$, where N_t is the number of temporal steps. The algorithm will be divided into three steps that will be repeated for every n of the temporal discretization. As the solution is evaluated only in these discrete time points, from now on it will be used the notation for every temporal step t_n: $U_\Omega(t_n) = U^n_\Omega$, $U(t_n) = U^n$ and $V(t_n) = V^n$.

The Cauchy Problem transforms a system of ordinary differential equations into a system of difference equations system by means of a a s-steps temporal scheme:

$$G(U^{n+1}_\Omega, \underbrace{U^n, \dots U^{n+1-s}}_{s \ steps}; t_n, \Delta t) = F_U(U^n, V^n; t_n),$$

$$U(t_0) = U^0, \qquad H_U(U^n; t_n)\big|_{\partial\Omega} = 0,$$

$$F_V(U^n, V^n; t_n) = 0, \qquad H_V(V^n; t_n)\big|_{\partial\Omega} = 0,$$

where

$$G : \mathbb{R}^{N_{e,u}-N_{C,u}} \times \underbrace{\mathbb{R}^{N_{e,u}} \times \ldots \times \mathbb{R}^{N_{e,u}}}_{s \ steps} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{N_{e,u}-N_{C,u}},$$

is the difference operator associated to the temporal scheme and Δt is the temporal step. Thus, at each temporal step four systems of $N_{e,u} - N_{C,u}$, $N_{C,u}$, $N_{e,v} - N_{C,v}$ and $N_{C,v}$ equations appear. In total a system of $N_{e,u} + N_{e,v}$ equations appear at each temporal step for all components of U^n and V^n.

Once the spatial discretizations are done, it is proceeded to integrate in time. This method is called the method of lines and it is represented in figure 7.1.

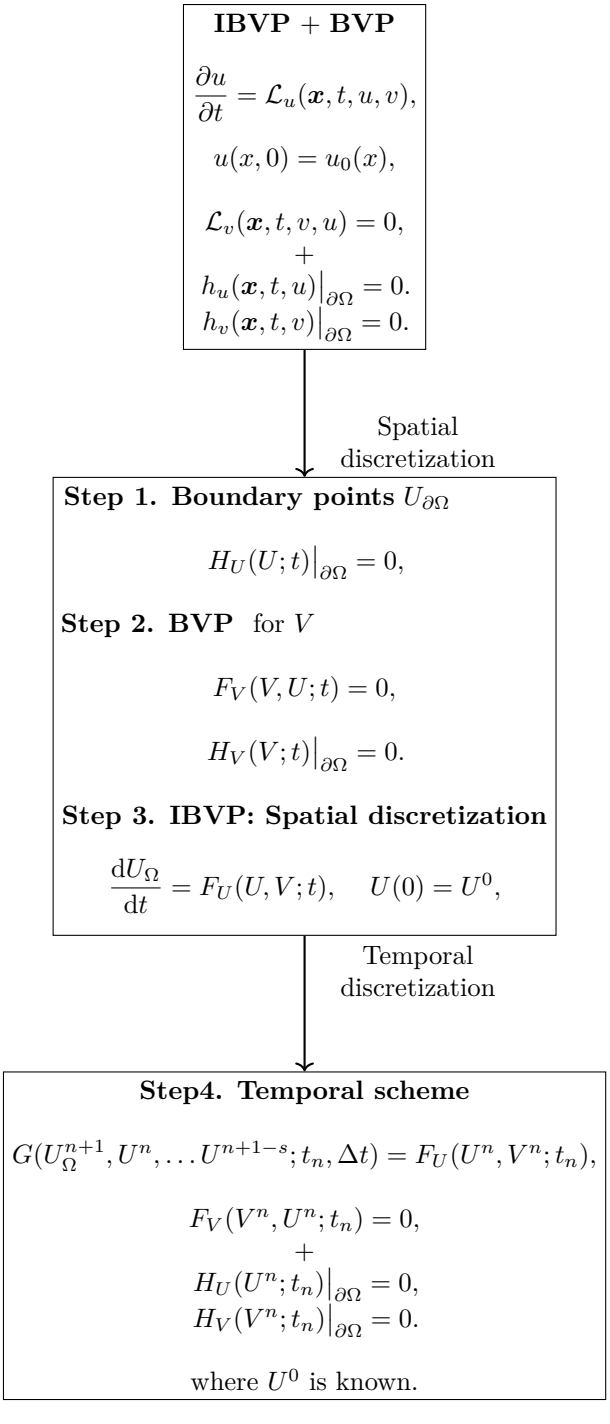

Figure 7.1: Method of lines for mixed initial and boundary value problems.

Starting from the initial value U^0, the initial value V^0 is calculated by means of the BVP that governs the variable V. Using both values U^0 and V^0, the difference operator F_U at that instant is constructed. With this difference operator, the temporal scheme yields the next temporal step U^1_Ω. Then, the boundary conditions of the IBVP are imposed to obtain the solution U^1. This solution will be used as the initial value to solve the next temporal step. In this way, the algorithm consists of a sequence of four steps that are carried out iteratively.

Step 1. Determination of boundary points $U^n_{\partial\Omega}$ from inner points U^n_Ω.

In the first place, the known initial value at the inner points U^n_Ω is used to impose the boundary conditions determining the boundary points $U^n_{\partial\Omega}$. That is, solving the system of equations:

$$H_U(U^n; t_n)\big|_{\partial\Omega} = 0.$$

Even though this might look redundant for the initial value U^0 (which is supposed to satisfy the boundary conditions), it is not for every other temporal step as the Cauchy Problem is defined only for the inner points U^n_Ω. This means that to construct the solution U^n its value at the boundaries $U^n\big|_{\partial\Omega}$ must be calculated satisfying the boundary constraints.

Step 2. Boundary Value Problem for V^n.

Once the value U^n is updated, the difference operator $F_V(V^n, U^n; t_n)$ is calculated by means of its derivatives. The known value U^n is introduced as a parameter in this operator. When U^n and the time t_n is introduced in such manner, the system of equations defined by the difference operator is invertible, a required condition to be solvable. The difference operator F_V is used along with the boundary conditions operator H_V, to solve the boundary value problem for V^n. It is precisely defined by:

$$F_V(V^n, U^n; t_n) = 0,$$

$$H_V(V^n; t_n)\big|_{\partial\Omega} = 0.$$

Since U^n and t_n act as a parameter, this problem can be solved by using the subroutines to solve a classical boundary value problem. However, to reuse the same interface than the classical BVP uses, the operator F_V and the boundary conditions H must be transformed into functions

$$F_{V,R} : \mathbb{R}^{N_{e,v}} \to \mathbb{R}^{N_{e,v}-N_{C,v}},$$

$$H_{V,R} : \mathbb{R}^{N_{e,v}} \to \mathbb{R}^{N_{C,v}}.$$

This is achieved by restricting these functions considering U^n and t_n as external parameters.

Once this is done, the problem can be written as:

$$F_{V,R}(V^n) = 0,$$

$$H_{V,R}(V^n)\big|_{\partial\Omega} = 0,$$

which is solvable in the same manner as explained in the chapter of Boundary Value Problems. Since this algorithm reuses the BVP software layer which has been explained previously, the details of this step will not be included. By the end of this step, both solutions U^n and V^n are known.

Step 3. Spatial discretization of the IBVP.

Once U^n and V^n are known, their derivatives are calculated by the selected finite differences formulas. Once calculated, the difference operator $F_U(U^n, V^n; t_n)$ is built.

Step4. Temporal step for U^n.

Finally, the difference operator previously calculated F_U acts as the evolution function of a Cauchy problem. Once the following step is evaluated, the solution U_Ω^{n+1} at inner points is yielded. This means solving the system:

$$G(U_\Omega^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t) = F_U(U^n, V^n; t_n).$$

In this system, the values of the solution at the s steps are known and therefore, the solution of the system is the solution at the next temporal step U_Ω^{n+1}. However, the temporal scheme G in general is a function that needs to be restricted in order to be invertible. In particular a restricted function $\tilde{G}$ must be obtained:

$$\tilde{G}(U_\Omega^{n+1}) = G(U_\Omega^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t)\Big|_{(U^n, \dots U^{n+1-s}; t_n, \Delta t)}$$

such that,

$$\tilde{G} : \mathbb{R}^{N_{e,u}-N_{C,u}} \to \mathbb{R}^{N_{e,u}-N_{C,u}}.$$

Hence, the solution at the next temporal step for the inner points results:

$$U_\Omega^{n+1} = \tilde{G}^{-1}(F_U(U^n, V^n; t_n)).$$

This value will be used as an initial value for the next iteration. The philosophy for other temporal schemes is the same, the result is the solution at the next temporal step.

7.3 Implementation: the upper abstraction layer

Once the algorithm is set with a precise notation, it is very easy to implement following rigorously the steps provided by the algorithm. The ingredients that are used to solve the IBVP coupled with an BVP are given by the equations (7.1)-(7.6). Hence, the arguments of the subroutine `IBVP_BVP` to solve this problem is implemented in the following way:

```
subroutine IBVP_and_BVP(  Time, x, y, L_u, L_v, BC_u, BC_v, &
                          Ut, Vt, Scheme )

   real, intent(in) :: Time(0:), x(0:), y(0:)
   procedure (L_uv) ::   L_u, L_v
   procedure (BC2D_system) :: BC_u, BC_v
   real, intent(out) :: Ut(0:, 0:, 0:, :),  Vt(0:, 0:, 0:, :)
   procedure (Temporal_Scheme), optional :: Scheme

   real, pointer :: pUt(:, :)
   real :: t_BC
   integer ::  it, Nx, Ny, Nt,  Nu, Nv, M1, M2, M3, M4
   real, allocatable :: Ux(:,:,:), Uxx(:,:,:), Uxy(:,:,:),            &
                        Uy(:,:,:), Uyy(:,:,:)

   Nx = size(x) - 1 ; Ny = size(y) - 1
   Nt = size(Time) - 1
   Nu = size(Ut, dim=4); Nv = size(Vt, dim=4)

   M1 = Nu*(Ny-1); M2 = Nu*(Ny-1); M3 = Nu*(Nx-1); M4 = Nu*(Nx-1)

   allocate( Ux(0:Nx,0:Ny, Nu), Uxx(0:Nx,0:Ny, Nu), Uxy(0:Nx,0:Ny, Nu), &
             Uy(0:Nx,0:Ny, Nu), Uyy(0:Nx,0:Ny, Nu) )

   call Data_pointer( N1 = Nt+1, N2 = Nu*(Nx+1)*(Ny+1),               &
                      U = Ut(0:Nt, 0:Nx, 0:Ny, 1:Nu) , pU = pUt )

   call Cauchy_ProblemS( Time, BVP_and_IBVP_discretization, pUt )

   deallocate( Ux, Uxx, Uxy, Uy, Uyy  )

contains
```

Listing 7.1: `IBVP_and_BVPs.f90`

These arguments comprise two differential operators `L_u` and `L_v` for the IBVP and the BVP respectively. The boundary conditions operators or functions of these two problems are called `BC_u` and `BC_v`. There are two output arguments `Ut` and `Vt` which the solution of the IBVP and the BVP respectively. The first three steps of the algorithm are carried out in the function `BVP_and_IBVP_discretization` and the four step is carried out in the subroutine `Cauchy-ProblemS`.

7.4 BVP_and_IBVP_discretization

```
subroutine BVP_and_IBVP_discretization_2D( U, t, F_u )
          real :: U(0:Nx,0:Ny, Nu), t, F_u(0:Nx,0:Ny, Nu)

    integer :: i, j, k
    real :: Vx(0:Nx,0:Ny, Nv), Vxx(0:Nx,0:Ny, Nv), Vxy(0:Nx,0:Ny, Nv)
    real :: Vy(0:Nx,0:Ny, Nv), Vyy(0:Nx,0:Ny, Nv), Uc(M1+M2+M3+M4)

        t_BC = t
        call Binary_search(t_BC, Time, it)
        write(*,*) " Time domain index = ", it

 !  *** initial boundary value :  Uc
        call Asign_BV2s( U( 0, 1:Ny-1, 1:Nu ), U( Nx, 1:Ny-1, 1:Nu ),  &
                         U( 1:Nx-1, 0, 1:Nu ), U( 1:Nx-1, Ny, 1:Nu ), Uc )
 !  *** Step1. Boundary points Uc from inner points U
        call Newton( BCs, Uc )

!   *** asign boundary points Uc  to U
        call Asign_BVs(Uc, U( 0, 1:Ny-1, 1:Nu ), U( Nx, 1:Ny-1, 1:Nu  ), &
                           U( 1:Nx-1, 0, 1:Nu ), U( 1:Nx-1, Ny, 1:Nu )  )

!  *** Derivatives of U for inner grid points
       do k=1, Nu
          call Derivative( ["x","y"], 1, 1, U(0:,0:, k),  Ux (0:,0:,k) )
          call Derivative( ["x","y"], 1, 2, U(0:,0:, k),  Uxx(0:,0:,k) )
          call Derivative( ["x","y"], 2, 1, U(0:,0:, k),  Uy (0:,0:,k) )
          call Derivative( ["x","y"], 2, 2, U(0:,0:, k),  Uyy(0:,0:,k) )
          call Derivative( ["x","y"], 2, 1, Ux(0:,0:,k),  Uxy(0:,0:,k) )
       end do
!  *** Step 2. BVP for V
       call Boundary_Value_Problem( x, y, L_v_R, BC_v_R, Vt(it,0:,0:,:))
!  *** Derivatives for V
    do k=1, Nv
      call Derivative( ["x","y"], 1, 1, Vt(it, 0:,0:, k), Vx (0:,0:,k) )
      call Derivative( ["x","y"], 1, 2, Vt(it, 0:,0:, k), Vxx(0:,0:,k) )
      call Derivative( ["x","y"], 2, 1, Vt(it, 0:,0:, k), Vy (0:,0:,k) )
      call Derivative( ["x","y"], 2, 2, Vt(it, 0:,0:, k), Vyy(0:,0:,k) )
      call Derivative( ["x","y"], 2, 1, Vx(0:    ,0:, k), Vxy(0:,0:,k) )
    end do

!  *** Step 3. Differential operator L_u(U,V) at inner grid points
 F_u=0
 do i=1, Nx-1; do j=1, Ny-1
    F_u(i, j, :) = L_u(                                                  &
    x(i), y(j), t, U(i, j, :),                                           &
    Ux(i, j, :), Uy(i, j, :), Uxx(i, j, :), Uyy(i, j, :), Uxy(i, j, :), &
    Vt(it, i, j, :),                                                     &
    Vx(i, j, :), Vy(i, j, :), Vxx(i, j, :), Vyy(i, j, :), Vxy(i, j, :)  )

 end  do; end do
end subroutine
```

Listing 7.2: IBVP_and_BVPs.f90

7.5 Step 1. Boundary values of the IBVP

As it was mentioned, the subroutine `BVP_and_IBVP_discretization` is the core subroutine of the algorithm. As it can be seen written in the code, it comprises the three first step of the algorithm. Step 1 is devoted to solve a system of equations for the boundary points `Uc` by means of Newton solver. The system is equations is constructed in the function `BCs`

```
function BCs(Y) result(G)
        real, intent (in) :: Y(:)
        real :: G(size(Y))

     real :: G1(M1), G2(M2), G3(M3), G4(M4)

! ** Asign Newton's iteration Y to Solution
     call Asign_BVs(Y, Ut(it, 0 ,1:Ny-1, 1:Nu), Ut(it, Nx, 1:Ny-1, 1:Nu),&
                       Ut(it, 1:Nx-1, 0, 1:Nu), Ut(it, 1:Nx-1, Ny, 1:Nu) )
! ** Calculate boundary conditions G
     call Asign_BCs( G1,  G2, G3,  G4 )

     G = [ G1, G2, G3, G4 ]

end function
```

Listing 7.3: `IBVP_and_BVPs.f90`

This subroutine prepares the functions `G` to be solved by Newton solver by packing equations of different edges of the spatial domain. The unknowns are gathered in the subroutine `Asign_BVs` and the equations are imposed in the subroutine `Asign_BCs`

```
subroutine Asign_BVs( Y, U1,  U2,  U3, U4)
  real, intent(in) :: Y(M1+M2+M3+M4)
  real, intent(out) :: U1(M1), U2(M2), U3(M3), U4(M4)

    integer :: i1, i2, i3, i4

     i1 = 1  + M1; i2 = i1 + M2; i3 = i2 + M3; i4 = i3 + M4

     U1 = Y(1  : i1-1)
     U2 = Y(i1 : i2-1)
     U3 = Y(i2 : i3-1)
     U4 = Y(i3 : i4-1)
end subroutine
```

Listing 7.4: `IBVP_and_BVPs.f90`

```
subroutine Asign_BCs(  G1, G2,  G3,  G4  )
   real, intent(out) :: G1(1:Ny-1,Nu), G2(1:Ny-1,Nu),                  &
                        G3(1:Nx-1,Nu), G4(1:Nx-1,Nu)

   real :: Wx(0:Nx, 0:Ny, Nu), Wy(0:Nx, 0:Ny, Nu)
   integer :: i, j, k

   do k=1, Nu
    call Derivative( ["x","y"], 1, 1, Ut(it, 0:, 0:, k), Wx(0:, 0:, k) )
    call Derivative( ["x","y"], 2, 1, Ut(it, 0:, 0:, k), Wy(0:, 0:, k) )
   end do

   do j = 1, Ny-1
     G1(j,:) = BC_u( x(0), y(j), t_BC,   &
                     Ut(it, 0, j, : ), Wx(0, j,:), Wy(0, j,:))
     G2(j,:) = BC_u( x(Nx), y(j), t_BC,   &
                     Ut(it, Nx, j, : ), Wx(Nx, j, :), Wy(Nx, j, :))
   end do

   do i = 1, Nx-1
     G3(i,:) = BC_u( x(i), y(0), t_BC,   &
                     Ut(it, i, 0,:), Wx(i, 0, :), Wy(i, 0,:))
     G4(i,:) = BC_u( x(i), y(Ny), t_BC,  &
                     Ut(it, i, Ny, : ), Wx(i, Ny, :), Wy(i, Ny, :))
   end do

end subroutine
```

Listing 7.5: `IBVP_and_BVPs.f90`

As it was mentioned, the function `BC_u` is the boundary conditions operator that is imposed to the IBVP and it is one of the input arguments of subroutine `IBVP_an_BVP`. To sum up, step 1 allows by gathering the unknowns of the boundary points and by building an algebraic system of equations to obtain the boundary values. This system of equations is solved by means of a Newton method.

7.6 Step 2. Solution of the BVP

As it can be seen in the subroutine `BVP_and_IBVP_discretization`, step 2 is carried out by the using the subroutine `Boundary_Value_Problem` which was developed in chapter devoted to the Boundary Value Problem. Since the interface of the differential operator argument `L_v` is not as the interface of the argument that uses `Boundary_Value_Problem`, some restrictions must be done. This restrictions for `L_v` and `BC_v` are done by menas of the functions `L_v_R` and `BC_v_R`.

```
function L_v_R(xr, yr, V, Vx, Vy, Vxx, Vyy, Vxy)   &
     result(Fv)
     real, intent(in) :: xr, yr, V(:), Vx(:), Vy(:), Vxx(:), Vyy(:), Vxy
         (:)
     real :: Fv(size(V))

     integer :: ix, iy

     call Binary_search(xr, x, ix)
     call Binary_search(yr, y, iy)

     Fv = L_v( xr, yr, t_BC, V(:),  Vx(:),  Vy(:),  &
               Vxx(:), Vyy(:), Vxy(:),              &
               Ut(it, ix, iy, :),  Ux(ix, iy, :),   &
               Uy(ix, iy, :), Uxx(ix, iy, :),       &
               Uyy(ix, iy, :), Uxy(ix, iy, :)    )

end function
```

Listing 7.6: `IBVP_and_BVPs.f90`

```
function BC_v_R(xr, yr, V, Vx, Vy) result (G_v)
     real, intent(in) :: xr, yr, V(:), Vx(:), Vy(:)
     real :: G_v(size(V))

    G_v = BC_v (xr, yr, t_BC, V, Vx, Vy )


end function
```

Listing 7.7: `IBVP_and_BVPs.f90`

As it can be observed, the interface of `L_v_R` and `BC_v_R` comply the requirements of the subroutine `Boundary_Value_Problem`. The extra arguments that `L_v` and `BC_v` require are accessed as external variables using the lexical scoping of subroutines inside another by means of the instruction `contains`.

7.7 Step 3. Spatial discretization of the IBVP

The spatial discretization of the IBVP is done in the last part of the subroutine `BVP_and_IBVP_discretization` by means of the differential operator `L_u` which is an input argument of the subroutine `BVP_and_IBVP`. Once derivatives of `U` and `V` are calculated in all grid points, the subroutine calculates the discrete or difference operator in each point of the domain. It is copied code snippet of the subroutine `BVP_and_IBVP_discretization` to follow easily these explanations.

```
!  *** Step 3. Differential operator L_u(U,V) at inner grid points
 F_u=0
 do i=1, Nx-1; do j=1, Ny-1
    F_u(i, j, :) = L_u(                                                    &
    x(i), y(j), t, U(i, j, :),                                             &
    Ux(i, j, :), Uy(i, j, :), Uxx(i, j, :), Uyy(i, j, :), Uxy(i, j, :), &
    Vt(it, i, j, :),                                                       &
    Vx(i, j, :), Vy(i, j, :), Vxx(i, j, :), Vyy(i, j, :), Vxy(i, j, :)  )

 end  do; end do
end subroutine
```

Listing 7.8: `IBVP_and_BVPs.f90`

As it observed, two loops run through all inner grid points of `U` variable. In step 1, the boundary values of `U` are obtained by imposing the boundary conditions. It is important also to notice that the evolution of the inner grid points `U` depends on the values of `U`, `V` and their derivatives that are calculated previously

```
!  *** Derivatives of U for inner grid points
       do k=1, Nu
          call Derivative( ["x","y"], 1, 1, U(0:,0:, k),  Ux (0:,0:,k) )
          call Derivative( ["x","y"], 1, 2, U(0:,0:, k),  Uxx(0:,0:,k) )
          call Derivative( ["x","y"], 2, 1, U(0:,0:, k),  Uy (0:,0:,k) )
          call Derivative( ["x","y"], 2, 2, U(0:,0:, k),  Uyy(0:,0:,k) )
          call Derivative( ["x","y"], 2, 1, Ux(0:,0:,k),  Uxy(0:,0:,k) )
       end do
!  *** Step 2. BVP for V
       call Boundary_Value_Problem( x, y, L_v_R, BC_v_R, Vt(it,0:,0:,:))
!  *** Derivatives for V
    do k=1, Nv
      call Derivative( ["x","y"], 1, 1, Vt(it, 0:,0:, k), Vx (0:,0:,k) )
      call Derivative( ["x","y"], 1, 2, Vt(it, 0:,0:, k), Vxx(0:,0:,k) )
      call Derivative( ["x","y"], 2, 1, Vt(it, 0:,0:, k), Vy (0:,0:,k) )
      call Derivative( ["x","y"], 2, 2, Vt(it, 0:,0:, k), Vyy(0:,0:,k) )
      call Derivative( ["x","y"], 2, 1, Vx(0:    ,0:, k), Vxy(0:,0:,k) )
    end do

!  *** Step 3. Differential operator L_u(U,V) at inner grid points
```

Listing 7.9: `IBVP_and_BVPs.f90`

7.8 Step 4. Temporal evolution of the IBVP

Finally, the state of the system in the next temporal step $n + 1$ is calculated by using the classical subroutine `Cauchy_ProblemS`. A code snippet of the subroutine `BVP_and_IBVP` is copied here to follow the explanations.

```
   call Cauchy_ProblemS( Time, BVP_and_IBVP_discretization, pUt )

   deallocate( Ux, Uxx, Uxy, Uy, Uyy  )

contains
```

Listing 7.10: `IBVP_and_BVPs.f90`

To reuse the subroutine `Cauchy_ProblemS` and since the interface of the function `BVP_and_IBVP_discretization` does not comply the requirements of the subroutine `Cauchy_ProblemS`, a restriction is used by means of the subroutine `BVP_and_IBVP_discretization_2D`

```
function BVP_and_IBVP_discretization( U, t ) result(F)
      real ::  U(:), t
      real :: F(size(U))

         call BVP_and_IBVP_discretization_2D( U, t, F )

end function
```

Listing 7.11: `IBVP_and_BVPs.f90`

```
subroutine BVP_and_IBVP_discretization_2D( U, t, F_u )
          real :: U(0:Nx,0:Ny, Nu), t, F_u(0:Nx,0:Ny, Nu)

    integer :: i, j, k
    real :: Vx(0:Nx,0:Ny, Nv), Vxx(0:Nx,0:Ny, Nv), Vxy(0:Nx,0:Ny, Nv)
    real :: Vy(0:Nx,0:Ny, Nv), Vyy(0:Nx,0:Ny, Nv), Uc(M1+M2+M3+M4)
```

Listing 7.12: `IBVP_and_BVPs.f90`

As it can be observed, whereas `U` is a vector of rank one for the subroutine `Cauchy_ProblemS`, the rank of `U` for the subroutine `BVP_and_IBVP_discretization_2D` is three. This association between dummy argument and actual argument violates the TKR (Type-Kind-Rank) rule but allows pointing to the same memory space without duplicating or reshaping variables.

Nomenclature

$\mathbb{R}^N$	: N-dimensional real numbers field.
$\mathcal{M}_{N\times N}$	: Set of all square matrices of dimension $N \times N$.
I	: Identity matrix.
$\det(A)$	: Determinant of a square matrix A.
$\boldsymbol{e}_i$	: Base vector of a vectorial space.
$\boldsymbol{e}_i \otimes \boldsymbol{e}_j$	: Base tensor of a tensorial space.
δ_{ij}	: Delta Kronecker function.
L	: Lower triangular matrix on a LU factorization.
U	: Upper triangular matrix on a LU factorization.
$\boldsymbol{f}$	: Application $\boldsymbol{f} : \mathbb{R}^p \longrightarrow \mathbb{R}^p$.
J_f	: Jacobian matrix of an application $\boldsymbol{f}$.
∇	: Nabla operator $\partial/\partial x_i \ \boldsymbol{e}_i$.
λ	: Eigenvalue of a square matrix.
ϕ	: Eigenvector of a square matrix.
$\Lambda(A)$	: Spectra of a matrix A.
$\kappa(A)$	: Condition number of a matrix A.
ℓ_j	: Lagrange polynomial centered at x_j for global interpolation.
ℓ_{jk}	: k grade Lagrange polynomial centered at x_j for piecewise interpolation.
$\boldsymbol{\ell}_x$	: Interpolation vector along OX $\boldsymbol{\ell}_x = \ell_i(x)\boldsymbol{e}_i$.
$\boldsymbol{\ell}_y$	: Interpolation vector along OY $\boldsymbol{\ell}_y = \ell_j(y)\boldsymbol{e}_j$.
$\mathcal{F}$	: Second order tensor for 2-dimensional interpolation.
$\mathrm{d}\boldsymbol{u}/\mathrm{d}t$	: Temporal derivative for a function $\boldsymbol{u} : \mathbb{R} \longrightarrow \mathbb{R}^p$.
$\boldsymbol{F}(\boldsymbol{u}, t)$	: Differential operator $\boldsymbol{F} : \mathbb{R}^p \times \mathbb{R} \longrightarrow \mathbb{R}^p$ for a Cauchy problem.
$\boldsymbol{u}_0$	: Initial condition for a Cauchy problem.
t_n	: n-th instant of the temporal mesh.
$\boldsymbol{u}^n$	: Discrete solution of the Cauchy problem, $\boldsymbol{u}^n \in \mathbb{R}^p$.
s	: Number of time steps for any numerical scheme.
$\boldsymbol{G}$	: Temporal scheme $\boldsymbol{G} : \mathbb{R}^p \times \mathbb{R}^p \times \ldots \times \mathbb{R}^p \times \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}^p$.
$\tilde{\boldsymbol{G}}$	: Restricted temporal scheme, $\boldsymbol{G}\big\vert_{(\boldsymbol{u}^n,\ldots\boldsymbol{u}^{n+1-s};t_n,\Delta t)}$, $\tilde{\boldsymbol{G}} : \mathbb{R}^p \longrightarrow \mathbb{R}^p$.
$\tilde{\boldsymbol{G}}^{-1}$	: Restricted temporal scheme inverse, $\tilde{\boldsymbol{G}}^{-1} : \mathbb{R}^p \longrightarrow \mathbb{R}^p$.

Ω : Interior of the spatial domain of PDE problem.
$\partial\Omega$: Boundary of the spatial domain of a PDE problem.
D : Domain of a PDE problem, $D \equiv \{\Omega \cup \partial\Omega\}$.
$\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u})$: Differential operator of a BVP, $\boldsymbol{\mathcal{L}} : D \times \mathbb{R}^{N_v} \longrightarrow \mathbb{R}^{N_v}$.
$\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u})\big|_{\partial\Omega}$: Boundary conditions operator for a BVP.
U : Discrete solution of a BVP.
$F(U)$: Inner points difference operator for a BVP.
$H(U)\big|_{\partial\Omega}$: Boundary points difference operator for a BVP.
$S(U)$: Inner and boundary points difference operator for a BVP.
$\partial \boldsymbol{u}/\partial t$: Temporal partial derivative of an IVBP $\boldsymbol{u} : \mathbb{R} \longrightarrow \mathbb{R}^{N_v}$.
$\boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u})$: Differential operator of an IVBP, $\boldsymbol{\mathcal{L}} : D \times \mathbb{R} \times \mathbb{R}^{N_v} \longrightarrow \mathbb{R}^{N_v}$.
$\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u})\big|_{\partial\Omega}$: Boundary conditions operator of an IVBP.
$\boldsymbol{u}_0(\boldsymbol{x})$: Initial condition for an IVBP.
U : Spatially discretized solution of an IVBP, $U : \mathbb{R} \longrightarrow \mathbb{R}^N$
U_Ω : Inner points, $U : \mathbb{R} \longrightarrow \mathbb{R}^{N-N_C}$
$U\big|_{\partial\Omega}$: Boundary points, $U : \mathbb{R} \longrightarrow \mathbb{R}^{N-N_C}$
$\mathrm{d}U_\Omega/\mathrm{d}t$: Temporal derivative for inner points.
$F(U; t)$: Difference operator for a spatially discretized IVBP
$H(U; t)\big|_{\partial\Omega}$: Difference operator for boundary points conditions.
U^n : Discretized solution of an IVBP solution at the instant t_n.
U^n_Ω : Inner points of a discretized solution.
$U^n_{\partial\Omega}$: Boundary points of a discretized solution.
E^n : Temporal discretization error for an IVBP.
E : Spatial discretization error for an IVBP.
$\boldsymbol{\varepsilon}_i$: Spatial discretization error at $\boldsymbol{x}_i$, $\boldsymbol{\varepsilon}_i : \mathbb{R} \longrightarrow \mathbb{R}$.
$\boldsymbol{\varepsilon}^n_i$: Temporal discretization error at $\boldsymbol{x}_i$.
$\boldsymbol{\varepsilon}^n_{T,i}$: Total error at $\boldsymbol{x}_i$.
E_T : Total error on the resolution of a linear IVBP.
$\boldsymbol{r}_i$: Truncation error at $\boldsymbol{x}_i$, $\boldsymbol{r}_i : \mathbb{R} \longrightarrow \mathbb{R}^{N_v}$
R : Truncation error for an IVBP, $R : \mathbb{R} \longrightarrow \mathbb{R}^{N-N_C}$.
Φ : Fundamental matrix, $\Phi : \mathbb{R} \longrightarrow \mathcal{M}_{N-N_C \times N-N_C}$.
$\exp(A)$: Exponential of the matrix A.
$\sup K$: Supreme element of a set K.
$\alpha(A)$: Spectral abscissa of a matrix A.
T^n : Truncation temporal error at instant t_n.
$\rho(A)$: Spectral radius of a matrix A.
$\boldsymbol{\mathcal{L}}_u$: Differential operator for an evolution variable $\boldsymbol{u} : D \times \mathbb{R} \longrightarrow \mathbb{R}^{N_u}$ of an IVBP and BVP mixed problem, $\boldsymbol{\mathcal{L}}_u : D \times \mathbb{R} \times \mathbb{R}^{N_u} \times \mathbb{R}^{N_v} \longrightarrow \mathbb{R}^{N_u}$.
$\boldsymbol{\mathcal{L}}_v$: Differential operator for a variable $\boldsymbol{v} : D \times \mathbb{R} \longrightarrow \mathbb{R}^{N_v}$ of an IVBP and BVP mixed problem, $\boldsymbol{\mathcal{L}}_v : D \times \mathbb{R} \times \mathbb{R}^{N_v} \times \mathbb{R}^{N_u} \longrightarrow \mathbb{R}^{N_v}$.
$\boldsymbol{h}_u$: Boundary conditions operator for a mixed problem.
$\boldsymbol{h}_v$: Boundary conditions operator for a mixed problem.

Part III

Application Program Interface

Chapter 1

Systems of equations

1.1 Overview

This is a library designed to solve systems of equations. The module `Linear_systems` has functions and subroutines related to linear algebra.

```
module Linear_systems

implicit none
private
public ::                &
  LU_factorization,      & ! A  = L U (lower, upper triangle matrices)
  Solve_LU,              & ! It solves L U x = b
  Gauss,                 & ! It solves A x = b by Guass elimination
  Condition_number,      & ! Kappa(A)  = norm2(A)  * norm2( inverse(A) )
  Tensor_product,        & ! A_ij = u_i v_j
  Power_method,          & ! It determines to largest eigenvalue of A
  Inverse_Power_method,  & ! It determines the smallest eigenvalue of A
  Eigenvalues_PM,        & ! All eigenvalue of A by the power method
  SVD                      ! A = U S transpose(V)


contains
```

Listing 1.1: `Linear_systems.f90`

1.2 Linear systems module

LU factorization

```
call LU_factorization( A )
```

The subroutine `LU_factorization` returns the inlet matrix which has been factored by the LU method. The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	inout	Square matrix to be factored by the LU method.

Table 1.1: Description of `LU_factorization` arguments

Solve LU

```
x = Solve_LU( A , b )
```

The function `Solve_LU` finds the solution to the linear system of equations $A\ \boldsymbol{x} = \boldsymbol{b}$, where the matrix A has been previously $L\ U$ factorized and $\boldsymbol{b}$ is a given vector. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	inout	Square matrix A previously factorized by `LU_factorization`.
b	vector of reals	in	Independent term $\boldsymbol{b}$.

Table 1.2: Description of `Solve_LU` arguments

Gauss

```
x = Gauss( A , b )
```

The function `Gauss` finds the solution of the linear system of equations $A\ \boldsymbol{x} = \boldsymbol{b}$ by means of a classical Gaussian elimination. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	inout	Square matrix A.
b	vector of reals	in	Independent term $\boldsymbol{b}$.

Table 1.3: Description of `Gauss` arguments

Condition number

```
kappa = Condition_number(A)
```

The function `Condition_number` determines the condition number $\kappa = ||A||_2\ ||A^{-1}||_2$

Tensor product

```
A = Tensor_product(u, v)
```

The function `Tensor_product` determines the matrix $A_{ij} = u_i\ v_j$. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
u	vector of reals	in	Vector $\boldsymbol{u}$.
v	vector of reals	in	Vector $\boldsymbol{v}$.

Table 1.4: Description of `Tensor_product` arguments

Power method

```
call Power_method(A, lambda, U)
```

The function `Power_method` finds the largest eigenvalue of A by the power method. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	inout	Square matrix A.
lambda	real	out	Largest eigenvalue.
U	vector of reals	out	Associated eigenvector.

Table 1.5: Description of `Power_method` arguments

Inverse Power method

```
call Inverse_Power_method(A, lambda, U)
```

The function `Power_method` finds the smallest eigenvalue of A by the inverse power method. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	inout	Square matrix A.
lambda	real	out	Smallest eigenvalue.
U	vector of reals	out	Associated eigenvector.

Table 1.6: Description of `Inverse_power_method` arguments

SVD

```
call SVD(A, sigma, U, V)
```

The function `SVD` finds the decomposition $A = U\ SV^T$. The arguments of the function are described in the following table.

Argument	Type	Intent	Description
A	two-dimensional array of reals	in	Square matrix A.
sigma	vector of reals	out	σ_k^2 eigenvalues of $A^T A$
U	two-dimensional array of reals	out	U_{ik} is the associated eigenvector of $A\ A^T$
V	two-dimensional array of reals	out	V_{ik} is the associated eigenvector of $A^T A$

Table 1.7: Description of `SVD` arguments

1.3 Non Linear Systems module

The module `Non_Linear_Systems` is used to solve non linear system of equations.

```
module Non_Linear_Systems

 use Jacobian_module
 use Linear_systems

implicit none
private
public ::       &
  Newton,  & ! It solves a vectorial system F(x) = 0
  Newtonc    ! It solves a vectorial system G(x) = 0
             ! with M implicit equations < N unknowns
             ! e.g.  G1 = x1 - x2 (implicit) with x1 = 1 (explicit)

contains
```

Listing 1.2: `Non_Linear_Systems.f90`

Newton

```
call Newton( F , x0 )
```

The subroutine `Newton` returns the solution of a non-linear system of equations. The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
F	vector function $F : \mathbb{R}^N \to \mathbb{R}^N$	in	System of equations to be solved.
x0	vector of reals	inout	Initial iteration point. When the iteration reaches convergence, this vector contains the solution of the problem.

Table 1.8: Description of `Newton` arguments

Newtonc

```
call Newtonc( F , x0 )
```

The subroutine `Newtonc` returns the solution of implicit and explicit equations packed in the same function $F(x)$. Hence, the function $F(x)$ has internally the following form:

$$\begin{aligned} x_1 = &\quad g_1(x_2, x_3, \dots x_N), \\ x_2 = &\quad g_2(x_1, x_3, \dots x_N), \\ \vdots &\quad \vdots \\ x_m = &\quad g_m(x_1, x_2, \dots x_N), \\ \\ F_1 = &\quad 0, \\ F_2 = &\quad 0, \\ \vdots &\quad \vdots \\ F_m = &\quad 0, \\ \\ F_{m+1} = &\quad g_{m+1}(x_1, x_2, \dots x_N), \\ \vdots &\quad \vdots \\ F_N = &\quad g_N(x_1, x_2, \dots x_N). \end{aligned}$$

The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
F	vector function	in	System of implicit and explicit equations to be solved.
x0	vector of reals	inout	Initial iteration point. When the iteration reaches convergence, this vector contains the solution of the problem.

Table 1.9: Description of `Newtonc` arguments

Chapter 2

Interpolation

2.1 Overview

This library is intended to solve an interpolation problem. It comprises: Lagrangian interpolation, Chebyshev interpolation and Fourier interpolation. To accomplish this purpose, `Interpolation` module uses three modules as it is shown in the following code:

```
module Interpolation

    use Lagrange_interpolation
    use Chebyshev_interpolation
    use Fourier_interpolation
    implicit none

  private
  public ::           &
  Interpolated_value, & ! It interpolates at xp from (x_i, y_i)
  Integral,           & ! It integrates from x_0 to x_N
  Interpolant           ! It interpolates I(xp) from (x_i, y_i)

contains
```

Listing 2.1: `Interpolation.f90`

The function `Interpolated_value` interpolates the value of a function at a certain point taking into account values of that function at other points. The function `Integral` computes the integral of a function in a certain interval and, finally, the function `Interpolant` calculates the interpolated values at different points.

2.2 Interpolation module

Interpolated value

The function `interpolated_value` is devoted to conduct a piecewise polynomial interpolation of the value of a certain function $y(x)$ in $x = x_p$. The data provided to carry out the interpolation is the value of that function $y(x)$ in a group of nodes.

```
yp = interpolated_value( x , y , xp , degree )
```

Argument	Type	Intent	Description
x	vector of reals	in	Points in which the value of the function $y(x)$ is provided.
y	vector of reals	in	Values of the function $y(x)$ in the group of points denoted by x.
xp	real	in	Point in which the value of the function y will be interpolated.
degree	integer	optional, in	Degree of the polynomial used in the interpolation. If it is not presented, it takes the value 2.

Table 2.1: Description of `interpolated_value` arguments

Integral

```
I = Integral( x , y , degree )
```

The function `Integral` is devoted to conduct a piecewise polynomial integration of a certain function $y(x)$. The data provided to carry out the interpolation is the value of that function $y(x)$ in a group of nodes. The limits of the integral correspond to the minimum and maximum values of the nodes.

The arguments of the function are described in the following table.

Argument	**Type**	**Intent**	**Description**
x	vector of reals	in	Points in which the value of the function $y(x)$ is provided.
y	vector of reals	in	Values of the function $y(x)$ in the group of points denoted by x.
degree	integer	in (optional)	Degree of the polynomial used in the interpolation. If it is not presented, it takes the value 2.

Table 2.2: Description of `Integral` arguments

2.3 Lagrange interpolation module

The Lagrange interpolation module is devoted to determine Lagrange interpolants as well as errors associated to the interpolation. To accomplish this purpose, `Lagrange_interpolation` module comprises the two following functions:

```
module Lagrange_interpolation

implicit none
public ::                &
  Lagrange_polynomials, & ! Lagrange polynomial at xp from (x_i, y_i)
  Lebesgue_functions      ! Lebesgue function at xp from x_i


contains
```

Listing 2.2: `Lagrange_interpolation.f90`

Lagrange polynomials

The function `Lagrange_interpolation` determines the value of the different Lagrange polynomials at some point `xp`. Given a set of nodal or interpolation points `x`, the following sentence determines the Lagrange polynomials:

```
yp =  Lagrange_polynomials( x, xp )
```

The interface of the function is:

```
pure function Lagrange_polynomials( x, xp )
   real, intent(in) :: x(0:), xp
   real Lagrange_polynomials(-1:size(x)-1,0:size(x)-1)
```

Listing 2.3: `Lagrange_interpolation.f90`

The result is a matrix containing all Lagrange polynomials

$$\ell_0(x),\ \ell_1(x), \ldots \ell_N(x)$$

and their derivatives $\ell_j^{(i)}(x)$ (first index of the array) calculated at the scalar point `xp`. The integral of the Lagrange polynomials is taken into account by the first index of the array with value equal to -1. The index 0 means the value of the Lagrange polynomials and an index k greater than 0 represents the "k-th" derivative of the Lagrange polynomial.

Lebesgue functions

The function `Lebesgue_functions` computes the Lebesgue function and its derivatives at different points `xp`. Given a set of nodal or interpolation points `x`, the following sentence determines the Lebesgue function:

```
yp =  Lebesgue_functions( x, xp )
```

The interface of the function is:

```
pure function Lebesgue_functions( x, xp )
   real, intent(in) :: x(0:), xp(0:)
   real Lebesgue_functions(-1:size(x)-1, 0:size(xp)-1)
```

Listing 2.4: `Lagrange_interpolation.f90`

The result is a matrix containing the Lebesgue function:

$$\lambda(x) = |\ell_0(x)| + |\ell_1(x)| + \ldots + |\ell_N(x)|$$

and their derivatives $\lambda^{(i)}(xp_j)$ (first index of the array) calculated at different points point xp_j. The integral of the Lebesgue function is represented by the first index with value equal to -1. The index 0 means the value of the Lebesgue function and an index k greater than 0 represents the "k-th" derivative of the Lebesgue function. The second index of the array takes into account different components of `xp`.

Chapter 3

Finite Differences

3.1 Overview

This library is intended to calculate total or partial derivatives of functions at any specific $x \in \mathbb{R}^1, \mathbb{R}^2, \mathbb{R}^3$. Since the function is known through different data points $(x_i, f(x_i))$, it is necessary to build an interpolant.

$$I(x) = f(x_0)\ \ell_0(x) + f(x_1)\ \ell_1(x) + \ldots + f(x_N)\ \ell_N(x).$$

Once this interpolant is built, the derivatives of the Lagrange polynomials allows to determine the derivative of the function.

$$\frac{dI(x)}{dx} = f(x_0)\ \frac{d\ell_0}{dx}(x) + f(x_1)\ \frac{d\ell_1}{dx}(x) + \ldots + f(x_N)\ \frac{d\ell_N}{dx}(x).$$

Hence, given a set of nodals or interpolation points, the coefficients for different derivatives are calculated by means of the subroutine `Grid_initialization`. Later, the subroutine `Derivative` calculates the derivative by multiplying the function values by this calculated coefficients.

```
module Finite_differences
 use Lagrange_interpolation
 use Non_uniform_grids

  implicit none
  private
  public ::               &
     Grid_Initialization, & ! Coefficients of different order derivatives
     Derivative             ! k-th derivative of u(:)
```

Listing 3.1: `Finite_differences.f90`

3.2 Finite differences module

Grid Initalization

```
call Grid_Initialization( grid_spacing , direction , q , grid_d )
```

Given the desired grid spacing, this subroutine calculates a set of points within the space domain defined with the first point at x_0 and the last point x_N. Later, it builds the interpolant and its derivatives at the same data points x_i and it stores their values for future use by the subroutine `Derivative`. The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
grid_spacing	character	in	It can be `'uniform'` (equally-spaced) or `'nonuniform'`.
direction	character	in	Selected by user. If the name of the direction has already been used along the program, it will be overwritten.
q	integer	in	Degree of the interpolating polynomial. The number of nodes (N) should be greater than the polynomials order (at least $N = \text{order} + 1$).
grid_d	vector of reals	inout	Contains the mesh nodes or nodal points.

Table 3.1: Description of `Grid_Initalization` arguments

If `grid_spacing` is `'nonuniform'`, the nodes are calculated by obtaining the extrema of the polynomial error associated to the polynomial of degree $N-1$ that the unknown nodes form.

Derivatives for $x \in \mathbb{R}^k$

Since the space domain $\Omega \subset \mathbb{R}^k$ with $k = 1, 2, 3$, derivatives of numerical problems for 1D, 2D and 3D grids. To avoid the definition of different subroutines to deal with different space dimensions, the subroutine `Derivative` is overloaded with the following subroutines:

```
interface Derivative
    module procedure Derivative3D, Derivative2D, Derivative1D
end interface
```

Listing 3.2: `Finite_differences.f90`

Derivative for 1D grids

```
call Derivative1D( direction , derivative_order , W , Wxi )
```

If `derivative_order=1`, then `Wxi` is calculated by the following operation:

$$\frac{dI(x)}{dx}(x_i) = f(x_0)\ \frac{d\ell_0}{dx}(x_i) + f(x_1)\ \frac{d\ell_1}{dx}(x_i) + \ldots + f(x_N)\ \frac{d\ell_N}{dx}(x_i).$$

The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
direction	character	in	It selects the direction which composes the grid from the ones that have already been defined.
derivative_order	integer	in	Order of derivation.
W	vector of reals	in	Given nodal values of function W.
Wxi	vector of reals	out	Value of the k-th derivate at the same nodal values.

Table 3.2: Description of `Derivative` arguments for 1D grids

Derivative for 2D and 3D grids

```
call Derivative2D( direction , coordinate , derivative_order , W , Wxi )
```

If `direction = ["x", "y"]`, `coordinate = 2` and `derivative_order = 1`, then `Wxi` is calculated by the following operation:

$$\frac{\partial I}{\partial y}(x_i, y_j) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} f(x_i, y_j) \ell_i(x_i) \frac{d\ell_j}{dy}(y_j).$$

The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
direction	vector of characters	in	It selects the directions which compose the grid from the ones that have already been defined. The first component of the vector will be the first coordinete and so on.
coordinate	integer	in	Coordinate at which the derivate is calculated. It can be 1 or 2 for 2D grids and 1, 2 or 3 for 3D grids.
derivative_order	integer	in	Order of derivation.
W	N-dimensional array of reals	in	Given nodal values of function W.
Wxi	N-dimensional array of reals	out	Value of the k-th derivate at the same nodal values.

Table 3.3: Description of `Derivative` arguments for 2D and 3D grids

Chapter 4

Cauchy Problem

4.1 Overview

The module `Cauchy_Problem` is designed to solve the following problem:

$$\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = \boldsymbol{f}\ (\boldsymbol{U},\ t), \quad \boldsymbol{U}(0) = \boldsymbol{U}^0, \quad \boldsymbol{f} : \mathbb{R}^{Nv} \times \mathbb{R} \to \mathbb{R}^{Nv}$$

```
module Cauchy_Problem

  use ODE_Interface
  use Temporal_scheme_interface
  use Temporal_Schemes
  implicit none

private
public ::           &
  Cauchy_ProblemS,  & ! It calculates the solution of a Cauchy problem
  set_tolerance,    & ! It sets the error tolerance of the integration
  set_solver,       & ! It defines the family solver and the name solver
  get_effort          ! # function evaluations (ODES) after integration

contains
```

Listing 4.1: `Cauchy_Problem.f90`

The subroutine `Cauchy_ProblemS` is called to calculate the solution $\boldsymbol{U}(t)$. If no numerical method is defined, the system is integrated by means of a fourth order Runge Kutta method. To define the error tolerance, the subroutine `set_tolerance` is used. To specify the discrete temporal method, the subroutine `set_solver` is called.

4.2 Cauchy problem module

Cauchy ProblemS

```
call Cauchy_ProblemS(Time_Domain, Differential_operator, Solution, Scheme)
```

The subroutine `Cauchy_ProblemS` calculates the solution to a Cauchy problem. Previously to using it, the initial conditions must be imposed. The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time_Domain(0:N)	vector of reals	in	Time domain partition where the solution is calculated.
Differential_operator	vector function: $\boldsymbol{f} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$	in	It is the function $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ described in the overview.
Solution(0:N, 1:Nv)	matrix of reals.	out	The first index represents the time and the second index contains the components of the solution.
Scheme	temporal scheme	optional in	Defines the scheme used to solve the problem. If it is not present, the subroutine `set_solver` allows to define the family and the member of the family. If the family is not defined, it uses a Runge Kutta of four stages.

Table 4.1: Description of `Cauchy_ProblemS` arguments

Set solver

```
call set_solver( family_name, scheme_name)
```

The subroutine `set_solver` allows to select the family of the numerical method and some specific member of the family to integrate the Cauchy problem.

Argument	Type	Intent	Description
familiy_name	character array	in	family name of the numerical scheme to integrate the evolution problem.
scheme_name	character array	in	name of a specific member of the family.

Table 4.2: Description of `set_tolerance` arguments

The following list describes new software implementations of the different families and members:

1. Embbeded Runge Kutta familty (`"eRK"`). Specific scheme names:
   (a) `"HeunEuler21".`
   (b) `"RK21".`
   (c) `"BogackiShampine".`
   (d) `"DOPRI54".`
   (e) `"Fehlberg54".`
   (f) `"Cash\_Karp".`
   (g) `"Fehlberg87".`
   (h) `"Verner65".`
   (i) `"RK65".`
   (j) `"RK87".`
2. Gragg, Burlish and Stoer method (`"GBS"`). Specific scheme names:
   (a) `"GBS".`
3. Adams, Bashforth, Moulton methods (`"ABM"`) implemented as multivalue methods. Specific scheme names:
   (a) `"PC_ABM".`

The following list describes wrappers for classical codes for the different families:

1. Wrappers of classical embbeded Runge Kutta (`"weRK"`). Specific scheme names:
   (a) `"WDOP853"`.
   (b) `"WDOPRI5"`.
2. Wrappers of classical Gragg Burlish and (`"wGBS"`). Specific scheme names:
   (a) `"WODEX"`.
3. Wrappers of classical Adams, Bashforth Methods (`"wABM"`). Specific scheme names:
   (a) `"WODE113"`.

Set tolerance

```
call set_tolerance(Tolerance)
```

The subroutine `set_tolerance` allows to fix the relative and absolute error tolerance of the solution. Embedded Runge-Kutta methods, Adams Bashforth or GBS methods are able to modify locally their time step to attain the required error tolerance.

Argument	Type	Intent	Description
tolerance	real	in	Relative or absolute error tolerance of the solution.

Table 4.3: Description of `set_tolerance` arguments

Get effort

```
get_effort()
```

The function `get_effort` determines the number of evaluations of the vector function associated to the Cauchy problem that are done by the numerical scheme to accomplish the required tolerance.

4.3 Temporal schemes

The module `Temporal_schemes` comprises easy examples of temporal schemes and allows checking new methods developed by the user.

```
module Temporal_Schemes


use Non_Linear_Systems
use ODE_Interface
use Temporal_scheme_interface

use Embedded_RKs
use Gragg_Burlisch_Stoer
use Adams_Bashforth_Moulton
use Wrappers

implicit none

  private
  public ::  &
       Euler,               & ! U(n+1) <- U(n) + Dt F(U(n))
       Inverse_Euler,       & ! U(n+1) <- U(n) + Dt F(U(n+1))
       Crank_Nicolson,      & ! U(n+1) <- U(n) + Dt/2 ( F(n+1) + F(n) )
       Leap_Frog,           & ! U(n+1) <- U(n-1) + Dt/2 F(n)
       Runge_Kutta2,        & ! U(n+1) <- U(n) + Dt/2 ( F(n)+F(U_Euler) )
       Runge_Kutta4,        & ! Runge Kutta method of order 4
       Adams_Bashforth2,    & ! U(n+1) <- U(n) + Dt/2 ( 3 F(n)-F(U(n-1) )
       Adams_Bashforth3,    & ! Adams Bashforth method of Order 3
       Predictor_Corrector1,& ! Variable step methods
```

Listing 4.2: `Temporal_schemes.f90`

The `Cauchy_problem` module uses schemes with the following interface:

```
module Temporal_scheme_interface

 implicit none

 abstract interface
   subroutine Temporal_Scheme(F, t1, t2,  U1, U2, ierr )
                use ODE_Interface
                procedure (ODES) :: F
                real, intent(in)    :: t1, t2
                real, intent(in) ::  U1(:)
                real, intent(out) ::  U2(:)
                integer, intent(out) :: ierr
  end subroutine
end interface

end module
```

Listing 4.3: `Temporal_scheme_interface.f90`

4.4 Stability

The module `Stability_regions` allows to calculate the region of absolute stability of any numerical method. This region is defined by the following expression:

$$\mathcal{R} = \{z \in \mathbb{C}, \pi(\rho, \omega) = 0, |\rho| < 1\}$$

```
module Stability_regions

    use Temporal_scheme_interface
    implicit none

private
public :: Absolute_Stability_Region ! For a generic temporal scheme

contains
```

Listing 4.4: `Stability_regions.f90`

```
call Absolute_Stability_Region(Scheme, x, y, Region)
```

Argument	Type	Intent	Description
Scheme	temporal scheme	in	Selects the scheme whose stability region is computed.
x	vector of reals	in	Real domain $\mathrm{Re}\, z$ of the complex plane.
y	vector of reals	in	Imaginary domain $\mathrm{Im}\, z$ of the complex plane.
Region	matrix of reals	in	Maximum value of the roots of the characteristic polynomial for each point of the complex domain.

Table 4.4: Description of `Cauchy_ProblemS` arguments

4.5 Temporal error

The module `Temporal_error` allows to determine, based on Richardson extrapolation, the error of a numerical solution.

```
module Temporal_error


    use Cauchy_Problem
    use Temporal_scheme_interface
    use ODE_interface
    implicit none

private
public ::                       &
  Error_Cauchy_Problem,         & ! Richardson extrapolation
  Temporal_convergence_rate,    & ! log Error versus log time steps
  Temporal_effort_with_tolerance  ! log time steps versus log (1/tolerance
      )


    contains
```

Listing 4.5: `Temporal_error.f90`

The module uses the `Cauchy_Problem` module and comprises three subroutines to analyze the error of the temporal schemes. It is an application layer based on the `Cauchy_Problem` layer. The error is calculated by integrating the same solution in successive time grids. By using the Richardson extrapolation method, the error is determined.

Error of the solution

```
call Error_Cauchy_Problem( Time_Domain, Differential_operator, Scheme, &
                           order, Solution, Error )
```

Argument	Type	Intent	Description
Time_Domain(0:N)	vector of reals	in	Time domain partition where the solution is calculated.
Differential_operator	vector function $\boldsymbol{f} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$	in	It is the function $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ described in the overview.
Scheme	temporal scheme	optional in	Defines the scheme used to solve the problem.
order	integer	in	order of the numerical scheme.
Solution(0:N, 1:Nv)	matrix of reals.	out	The first index represents the time and the second index contains the components of the solution.
Error(0:N, 1:Nv)	matrix of reals.	out	The first index represents the time and the second index contains the components of the solution.

Table 4.5: Description of `Error_Cauchy_Problem` arguments

Convergence rate with time steps

```
call Temporal_convergence_rate( Time_Domain, Differential_operator,   &
                                U0, Scheme, order, log_E, log_N)
```

Argument	Type	Intent	Description
Time_Domain(0:N)	vector of reals	in	Time domain partition where the solution is calculated.
Differential_operator	vector function $\boldsymbol{f} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$	in	It is the function $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ described in the overview.
U0	vector of reals	in	Components of the initial conditions.
Scheme	temporal scheme	in (optional)	Defines the scheme used to solve the problem.
order	integer	in	order of the numerical scheme.
log_E	vector of reals	out	Log of the norm2 of the error solution. Each component represents a different time grid.
log_N	vector of reals	out	Log of number of time steps to integrate the solution. Sequence of time grids N, 2N, 4N... Each component represents a different time grid.

Table 4.6: Description of `Temporal_convergence_rate`

Error behavior with tolerance

```
call Temporal_steps_with_tolerance(Time_Domain, Differential_operator, &
                                   U0, log_mu, log_steps)
```

Argument	Type	Intent	Description
Time_Domain(0:N)	vector of reals	in	Time domain partition where the solution is calculated.
Differential_operator	vector function $\boldsymbol{f} : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$	in	It is the function $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ described in the overview.
U0	vector of reals	in	Initial conditions.
log_mu	vector of reals	in	Log of the 1/tolerances. This vector is given and it allows to integrate internally different simulations with different error tolerances.
log_steps	vector of reals	out	Log of the number of time steps to accomplish a simulation with a given error tolerance. The numerical scheme has to be selected previously with `set_solver`.

Table 4.7: Description of `Temporal_error_with_tolerance`

Chapter 5

Boundary Value Problems

5.1 Overview

This library is intended to solve linear and nonlinear boundary value problems. An equation involving partial derivatives together with some constraints applied to the frontier of its spatial domain constitute a boundary value problem.

```
module Boundary_value_problems
 use Boundary_value_problems1D
 use Boundary_value_problems2D
 use Boundary_value_problems3D

 implicit none
 private
 public :: Boundary_Value_Problem ! It solves a boundary value problem

 interface Boundary_Value_Problem
      module procedure Boundary_Value_Problem1D,        &
                       Boundary_Value_Problem2D,        &
                       Boundary_Value_Problem2D_system, &
                       Boundary_Value_Problem3D_system
 end interface
end module
```

Listing 5.1: `Boundary_value_problems.f90`

Since the space domain $\Omega \subset \mathbb{R}^k$ with $k = 1, 2, 3$, boundary value problems are stated in 1D, 2D and 3D grids. To have the same name interface when dealing with different space dimensions, the subroutine `Boundary_value_problem` has been overloaded.

5.2 Boundary value problems module

1D Boundary Value Problems

```
call Boundary_Value_Problem1D( x_nodes, Differential_operator,  &
                               Boundary_conditions , Solution )
```

The subroutine calculates the solution of the following boundary value problem:

$$\mathcal{L}\left(x,\ u,\ \frac{\partial u}{\partial x},\ \frac{\partial^2 u}{\partial x^2}\right) = 0, \qquad h\left(x,\ u,\ \frac{\partial u}{\partial x}\right) = 0$$

The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
x_nodes	vector of reals	inout	Contains the mesh nodes.
Differential_operator	real function: $\mathcal{L}(x, u, u_x, u_{xx})$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $h(x, u, u_x)$	in	In this function, the boundary conditions are fixed. The user must include a conditional sentence which sets $h(a,\ u,\ u_x)$ and $h(b,\ u,\ u_x)$.
Solution	vector of reals	out	Solution $u(x)$ of the boundary value problem.

Table 5.1: Description of `Boundary_Value_Problem` arguments for 1D problems

2D Boundary Value Problems

```
call Boundary_Value_Problem2D( x_nodes, y_nodes,                &
                               Differential_operator,           &
                               Boundary_conditions, Solution )
```

This subroutine calculates the solution to a linear boundary value problem in a rectangular domain $[a, b] \times [c, d]$:

$$\mathcal{L}\left(x,\ y,\ u,\ \frac{\partial u}{\partial x},\ \frac{\partial u}{\partial y},\ \frac{\partial^2 u}{\partial x^2},\ \frac{\partial^2 u}{\partial y^2},\ \frac{\partial^2 u}{\partial x \partial y}\right) = 0,$$

The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
x_nodes	vector of reals	inout	Contains the mesh nodes in the first direction of the mesh.
y_nodes	vector of reals	inout	Contains the mesh nodes in the second direction of the mesh.
Differential_operator	real function: $\mathcal{L}$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $h\left(x, y, u, u_x, u_y\right)$	in	The user must use a conditional sentence to impose BCs.
Solution	two-dimensional array of reals	out	Solution $u = u(x, y)$.

Table 5.2: Description of `Linear_Boundary_Value_Problem` arguments for 2D problems

2D Boundary Value Problems for system of equations

This subroutine calculates the solution of a boundary value problem of N variables in a rectangular domain $[a, b] \times [c, d]$:

$$\boldsymbol{\mathcal{L}}\left(x,\ y,\ \boldsymbol{u},\ \frac{\partial \boldsymbol{u}}{\partial x},\ \frac{\partial \boldsymbol{u}}{\partial y},\ \frac{\partial^2 \boldsymbol{u}}{\partial x^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial y^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial x \partial y}\right) = \mathbf{0}$$

The solution of this problem is calculated using the libraries by a simple call to the subroutine:

```
call Boundary_Value_Problem2D_system(x_nodes, y_nodes,                    &
                                     Differential_operator,               &
                                     Boundary_conditions, Solution)
```

Argument	Type	Intent	Description
x_nodes	vector of reals	inout	Mesh nodes in the direction x.
y_nodes	vector of reals	inout	Mesh nodes in direction y.
Differential_operator	function: $\boldsymbol{\mathcal{L}}$	in	This function is the differential operator.
Boundary_conditions	function: $\boldsymbol{h}$	in	Boundary conditions for all variables.
Solution	three-dimensional array of reals	out	Solution $\boldsymbol{u} = \boldsymbol{u}(x, y)$. Third index: index of the variable.

Table 5.3: Description of `Linear_Boundary_Value_Problem_System` arguments for 2D problems

3D Boundary Value Problems for systems of equations

```
call Boundary_Value_Problem3D_system(x_nodes, y_nodes, z_nodes,          &
                                     Differential_operator, &
                                     Boundary_conditions, Solution )
```

This subroutine calculates the solution of a boundary value problem system of N variables in a rectangular domain $[a, b] \times [c, d] \times [e, f]$:

$$\boldsymbol{\mathcal{L}}\left(x,\ y,\ z,\ \boldsymbol{u},\ \frac{\partial \boldsymbol{u}}{\partial x},\ \frac{\partial \boldsymbol{u}}{\partial y},\ \frac{\partial \boldsymbol{u}}{\partial z},\ \frac{\partial^2 \boldsymbol{u}}{\partial x^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial y^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial z^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial x \partial y},\ \frac{\partial^2 \boldsymbol{u}}{\partial x \partial z},\ \frac{\partial^2 \boldsymbol{u}}{\partial y \partial z}\right) = \boldsymbol{0}$$

Argument	**Type**	**Intent**	**Description**
x_nodes	vector of reals	inout	Nodes in X.
y_nodes	vector of reals	inout	Nodes in y.
z_nodes	vector of reals	inout	Nodes in z.
Differential_operator	function: $\boldsymbol{\mathcal{L}}$	in	Differential operator.
Boundary_conditions	function: $\boldsymbol{h}$	in	Boundary conditions.
Solution	4-dimensional array of reals	out	Solution $\boldsymbol{u} = \boldsymbol{u}(x, y, z)$. Fourth index: index of the variable.

Table 5.4: Description of `Boundary_Value_Problem` arguments for 3D problems

Chapter 6

Initial Value Boundary Problem

6.1 Overview

This library is intended to solve an initial value boundary problem. This problem is governed by a set time evolving partial differential equations together with boundary conditions and an initial condition.

```
module Initial_Boundary_Value_Problems
 use Initial_Boundary_Value_Problem1D
 use Initial_Boundary_Value_Problem2D
 use Utilities
 use Temporal_Schemes
 use Finite_differences

 implicit none
 private
 public :: Initial_Boundary_Value_Problem

 interface  Initial_Boundary_Value_Problem
        module procedure IBVP1D, IBVP1D_system, IBVP2D, IBVP2D_system
 end interface

end module
```

Listing 6.1: `Initial_Boundary_Value_Problems.f90`

Since the space domain $\Omega \subset \mathbb{R}^k$ with $k = 1, 2, 3$, initial value boundary problems are stated in 1D, 2D and 3D grids. To have the same name interface when dealing with different space dimensions, the subroutine `Initial_Value_Boundary_Problem` has been overloaded.

6.2 Initial Value Boundary Problem module

1D Initial Value Boundary Problem

```
call Initial_Boundary_Value_Problem(Time_Domain, x_nodes,                &
                                    Differential_operator,               &
                                    Boundary_conditions, Solution, Scheme)
```

This subroutine calculates the solution to a boundary initial value problem in a domain $x \in [a, b]$ such as:

$$\frac{\partial u}{\partial t} = \mathcal{L}\left(x,\ t,\ u,\ \frac{\partial u}{\partial x},\ \frac{\partial^2 u}{\partial x^2}\right)$$

Besides, an initial condition must be established: $u(x, t = t_0) = u_0(x)$.

Argument	**Type**	**Intent**	**Description**
Time_Domain	vector of reals	in	Time domain where the solution is calculated.
x_nodes	vector of reals	inout	Contains the mesh nodes.
Differential_operator	real function: $\mathcal{L}(x, t, u, u_x, u_{xx})$	in	Differential operator.
Boundary_conditions	real function: $h(x, t, u, u_x)$	in	The user must include a conditional sentenceto impose boundary conditions.
Solution	two-dimensional array of reals	out	Solution $u = u(x, t)$.
Scheme	temporal scheme	optional in	Numerical scheme to integrate in time. If it is not specified, it uses a Runge-Kutta of four stages.

Table 6.1: Description of `Initial_Value_Boundary_ProblemS` arguments for 1D problems

1D Initial Value Boundary Problem for systems of equations

```
call Initial_Boundary_Value_Problem(Time_Domain, x_nodes,                &
                                    Differential_operator,               &
                                    Boundary_conditions, Solution, Scheme)
```

The subroutine `Initial_Value_Boundary_Problem` calculates the solution to a boudary initial value problem in a rectangular domain $x \in [a, b]$ such as:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}\left(x,\ t,\ \boldsymbol{u},\ \frac{\partial \boldsymbol{u}}{\partial x},\ \frac{\partial^2 \boldsymbol{u}}{\partial x^2}\right)$$

Besides, an initial condition must be established: $\boldsymbol{u}(x, t = t_0) = \boldsymbol{u}_0(x)$. The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time_Domain	vector of reals	in	Time domain where the solution wants to be calculated.
x_nodes	vector of reals	inout	Contains the mesh nodes.
Order	integer	in	It indicates the order of the finitte differences.
Differential_operator	function: $\boldsymbol{\mathcal{L}}\left(x, t, \boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial x}, \frac{\partial^2 \boldsymbol{u}}{\partial x^2}\right)$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	function: $\boldsymbol{h}\left(x, t, \boldsymbol{u}, \boldsymbol{u}_x\right)$	in	Boundary conditions.
Solution	three-dimensional array of reals	out	Solution $\boldsymbol{u} = \boldsymbol{u}(x, t)$.
Scheme	temporal scheme	optional in	Optional temporal scheme. Default: Runge Kutta of four stages.

Table 6.2: Description of `Initial_Value_Boundary_ProblemS_System` arguments for 1D problems

2D Initial Value Boundary Problems

```
call Initial_Boundary_Value_Problem(Time_Domain, x_nodes, y_nodes,       &
                                    Differential_operator,               &
                                    Boundary_conditions, Solution, Scheme)
```

This subroutine calculates the solution to a scalar initial value boundary problem in a rectangular domain $(x, y) \in [x_0, x_f] \times [y_0, y_f]$:

$$\frac{\partial u}{\partial t} = \mathcal{L}(x, y, t, u, u_x, u_y, u_{xx}, u_{yy}, u_{xy}), \qquad h(x, y, t, u, u_x, u_y)\Big|_{\partial\Omega} = 0.$$

Besides, an initial condition must be established: $u(x, y, t_0) = u_0(x, y)$. The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time_Domain	vector of reals	in	Time domain.
x_nodes	vector of reals	inout	Mesh nodes along OX.
y_nodes	vector of reals	inout	Mesh nodes along OY.
Order	integer	in	Finite differences order.
Differential_operator	real function: $\mathcal{L}$	in	Differential operator of the problem.
Boundary_conditions	real function: h	in	Boundary conditions for u.
Solution	three-dimensional array of reals	out	Solution of the problem u.
Scheme	temporal scheme	optional in	Scheme used to solve the problem. If not given a Runge Kutta of four stages is used.

Table 6.3: Description of `Initial_Value_Boundary_ProblemS` arguments for 2D problems

Initial Value Boundary Problem System for 2D problems

```
call Initial_Boundary_Value_Problem(                                    &
             Time_Domain, x_nodes, y_nodes,  Differential_operator, &
             Boundary_conditions,  Solution, Scheme                     )
```

The subroutine `Initial_Value_Boundary_ProblemS` calculates the solution to a boundary initial value problem in a rectangular domain $(x, y) \in [x_0, x_f] \times [y_0, y_f]$:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy}), \qquad \boldsymbol{h}(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y)\Big|_{\partial\Omega} = 0.$$

Besides, an initial condition must be established: $\boldsymbol{u}(x, y, t = t_0) = \boldsymbol{u}_0(x, y)$. The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time_Domain	vector of reals	in	Time domain.
x_nodes	vector of reals	inout	Mesh nodes along OX.
y_nodes	vector of reals	inout	Mesh nodes along OY.
Differential_operator	function: $\boldsymbol{\mathcal{L}}$	in	Differential operator.
Boundary_conditions	function: $\boldsymbol{h}$	in	Boundary conditions.
Solution	four-dimensional array of reals	out	Solution of the problem $\boldsymbol{u}$.
Scheme	temporal scheme	optional in	Scheme used to solve the problem. If not given, a Runge Kutta of four stages is used.

Table 6.4: Description of `Initial_Value_Boundary_ProblemS_System` arguments for 2D problems

Chapter 7

Mixed Boundary and Initial Value Problems

7.1 Overview

This library is intended to solve an initial value boundary problem for a vectorial variable $\boldsymbol{u}$ with a coupled boundary value problem for $\boldsymbol{v}$.

```
module IBVPs_and_BVPs

use Cauchy_problem
use Temporal_scheme_interface
use Finite_differences
use Linear_Systems
use Non_Linear_Systems
use Boundary_value_problems
use Utilities

implicit none
private
public :: IBVP_and_BVP
```

Listing 7.1: `IBVP_and_BVPs.f90`

7.2 Mixed BVP and IBVP module

The subroutine `IBVP_and_BVP` calculates the solution to a boundary initial value problem in a rectangular domain $(x, y) \in [x_0, x_f] \times [y_0, y_f]$:

$$\begin{aligned}
&\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}_u(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy}, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_{xx}, \boldsymbol{v}_{yy}, \boldsymbol{v}_{xy}) \\
&\boldsymbol{\mathcal{L}}_v(x, y, t, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_{xx}, \boldsymbol{v}_{yy}, \boldsymbol{v}_{xy}, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy}) = 0, \\
&\boldsymbol{h}_u(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y)\Big|_{\partial\Omega} = 0, \qquad \boldsymbol{h}_v(x, y, t, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y)\Big|_{\partial\Omega} = 0.
\end{aligned}$$

Besides, an initial condition must be stablished: $\boldsymbol{u}(x, y, t_0) = \boldsymbol{u}_0(x, y)$. The problem is solved by means of a simple call to the subroutine:

```
call IBVP_and_BVP( Time, x_nodes, y_nodes, L_u, L_v, BCs_u, BCs_v,  &
                   Ut, Vt, Scheme        )
```

The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time	vector of reals	in	Time domain.
x_nodes	vector of reals	inout	Mesh nodes along OX.
y_nodes	vector of reals	inout	Mesh nodes along OY.
L_u	function: $\boldsymbol{\mathcal{L}}_u$	in	Differential operator for $\boldsymbol{u}$.
L_v	function: $\boldsymbol{\mathcal{L}}_v$	in	Differential operator for $\boldsymbol{v}$.
BCs_u	function: $\boldsymbol{h}_u$	in	Boundary conditions for $\boldsymbol{u}$.
BCs_v	function: $\boldsymbol{h}_v$	in	Boundary conditions for $\boldsymbol{v}$.
Ut	four-dimensional array of reals	out	Solution $\boldsymbol{u}$ of the evolution problem. Fourth index: index of the variable.
Vt	four-dimensional array of reals	out	Solution $\boldsymbol{v}$ of the boundary value problem. Fourth index: index of the variable.
Scheme	temporal scheme	optional in	Scheme used to solve the problem. If not given, a Runge Kutta of four steps is used.

Table 7.1: Description of `IBVP_and_BVP` arguments for 2D problems

Chapter 8

Plotting graphs with Latex

8.1 Overview

This library is designed to plot $x-y$ and contour graphs on the screen or to create automatically Latex files with graphs and figures. The module: `Plots` has two subroutines: `plot_parametrics` and `plot_contour`.

8.2 Plot parametrics

```
call plot_parametrics(x, y, legends, x_label, y_label, title, &
                      path, graph_type)
```

This subroutine plots a given number of parametric curves (x, y) on the screen and creates a Latex file for optimum quality results. This subroutine is overloaded allowing to plot parametric curves sharing x–axis for all curves or with different data booth for x and y axis. That is, x can be a vector the same for all parametric curves or a matrix. In this case, (x_{ij}, y_{ij}) represents the the point i of the parametric curve j. The last three arguments are optional. If they are given, this subroutine creates a plot data file (`path.plt`) and a latex file (`path.tex`) to show the same graphics results by compiling a latex document.

Argument	Type	Intent	Description
x	vector or matrix of reals	in	First index is the point and second index is the parametric curve.
y	matrix of reals	in	First index is the point and second index is the parametric curve.
legends	vector of char strings	in	These are the legends of the parametric curves.
x_label	character string	in	x label of the graph.
y_label	character string	in	y label of the graph.
title	character string	optional in	title of the graph.
path	character string	optional in	path of Latex and data files.
graph_type	character string	optional, in	graph type

Table 8.1: Description of `plot_parametrics` arguments for Latex graphs

```
subroutine myexampleC

    integer, parameter :: N=200, Np = 3
    real :: x(0:N), y(0:N, Np), a = 0, b = 2 * PI
    integer :: i
    character(len=100) :: path(4) =                    &
    ["./results/myexampleCa", "./results/myexampleCb", &
     "./results/myexampleCc", "./results/myexampleCd"  ]

    x  = [ (a + (b-a)*i/N, i=0, N) ]
    y(:, 1)  = sin(x); y(:, 2)  = cos(x); y(:, 3)  = sin(2*x)

   call plot_parametrics( x, y, ["$\sin x$", "$\cos x$", "$\sin 2x$"], &
                          "$x$", "$y$", "(a)", path(1) )
   call plot_parametrics( y(:,1), y(:,:), ["O1", "O2", "O3"],          &
                          "$y_2$", "$y_1$", "(b)", path(2) )
   call plot_parametrics( y(:,1), y(:,2:2), ["O2"],  "$y_2$", "$y_1$", &
                          "(c)", path(3) )
   call plot_parametrics( y(:,1), y(:,3:3), ["O3"], "$y_2$", "$y_1$",  &
                          "(d)", path(4) )
end subroutine
```

Listing 8.1: `my_examples.f90`

The above Fortran example creates automatically four plot files and four latex files. By compiling the following Latex file, the same plots showed on the screen can be included in any latex manuscript.

```
\documentclass[twoside,english]{book}

\usepackage{tikz}
\usepackage{pgfplots}
\pgfplotsset{compat=1.5}

\newcommand{\fourgraphs}[6]
  {
   \begin{figure}[htpb]

      \begin{minipage}[t]{0.5\textwidth} {#1} \end{minipage}
      \begin{minipage}[t]{0.5\textwidth} {#2} \end{minipage}

      \begin{minipage}[t]{0.5\textwidth} {#3} \end{minipage}
      \begin{minipage}[t]{0.5\textwidth} {#4} \end{minipage}
      \caption{#5} \label{#6}
   \end{figure}
  }

\begin{document}

      \fourgraphs
      {\input{./results/myexampleCa.tex} }
      {\input{./results/myexampleCb.tex} }
      {\input{./results/myexampleCc.tex} }
      {\input{./results/myexampleCd.tex} }
      {Heinon-Heiles system solution.
      (a) Trajectory of the star $(x,y)$.
      (b) Projection $(x,\dot{x})$ of the solution.
      (c) Projection $(y,\dot{y})$ of the solution.
      (d) Projection $(\dot{x},\dot{y})$.}
      {fig:exampleCad}
```

Listing 8.2: `Latex.tex`

After compiling the above Latex code, the plot of figure 8.1 is obtained.

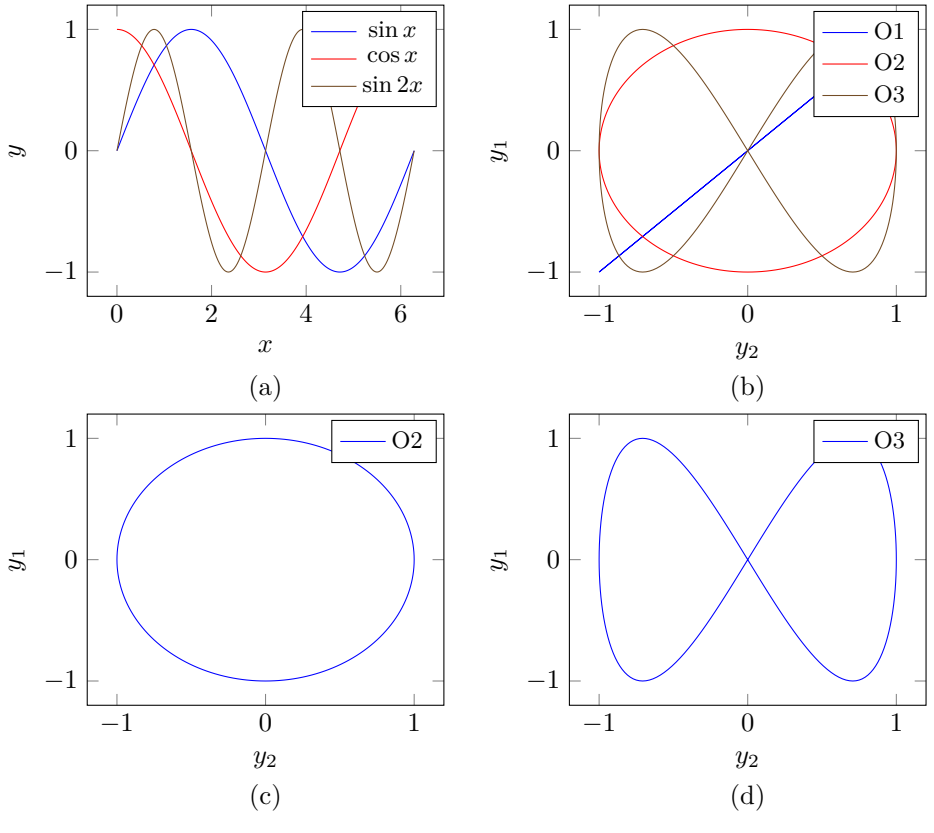

Figure 8.1: Heinon-Heiles system solution. (a) Trajectory of the star (x, y). (b) Projection $(x, \dot{x})$ of the solution. (c) Projection $(y, \dot{y})$ of the solution. (d) Projection $(\dot{x}, \dot{y})$.

8.3 Plot contour

```
call plot_contour(x, y, z, x_label, y_label, levels, legend, &
                  path, graph_type)
```

This subroutine plots a contour map of $z = z(x, y)$ on the screen and creates a Latex file for optimum quality results. Given a set of values x_i and y_j where some function $z(x, y)$ is evaluated, this subroutine plot a contour map. The last three arguments are optional. If they are given, this subroutine creates a plot data file (`path.plt`) and a latex file (`path.tex`) to show the same graphics results by compiling a latex document.

Argument	**Type**	**Intent**	**Description**
x	vector of reals	in	x_i grid values.
y	vector of reals	in	y_j grid values.
z	matrix of reals	in	z_{ij} different evaluations of $z(x, y)$.
x_label	character string	in	x label of the graph.
y_label	character string	in	y label of the graph.
levels	vector of reals	optional in	Levels for the iso–lines.
legend	character string	optional in	title of the graph.
path	character string	optional in	Latex and data files.
graph_type	character string	optional in	”color” or ”isolines”

Table 8.2: Description of `plot_contour` arguments

```
subroutine myexampleD

    integer, parameter :: N=20, Nl = 29
    real :: x(0:N), y(0:N), z(0:N, 0:N)
    real :: levels(0:Nl), a = 0, b = 2 * PI
    integer :: i
    character(len=100) :: path(2) =  ["./results/myexampleDa", &
                                      "./results/myexampleDb"  ]
    x  = [ (a + (b-a)*i/N, i=0, N) ]
    y  = [ (a + (b-a)*i/N, i=0, N) ]
    a = -1; b = 1
    levels  = [ (a + (b-a)*i/Nl, i=0, Nl) ]
    z = Tensor_product( sin(x), sin(y) )

    call plot_contour(x, y, z, "x", "y", levels, "(a)",path(1),"color")
    call plot_contour(x, y, z, "x", "y", levels, "(b)",path(2),"isolines")
end subroutine
```

Listing 8.3: `my_examples.f90`

The above Fortran example creates the following data files and latex files:

`./results/myexampleDa.plt`, `./results/myexampleDa.tex`,

`./results/myexampleDb.plt`, `./results/myexampleDb.tex`.

By compiling the following Latex file, the same plots showed on the screen are included in any latex manuscript.

```
\newcommand{\twographs}[4]
  {
   \begin{figure}[htpb]
      \begin{minipage}[t]{0.5\textwidth} {#1} \end{minipage}
      \begin{minipage}[t]{0.5\textwidth} {#2} \end{minipage}
      \caption{#3} \label{#4}
   \end{figure}
  }

    \twographs
     {\input{./results/myexampleDa.tex}}
     {\input{./results/myexampleDa.tex}}
     {Heinon-Heiles system solution.
     (a) Trajectory of the star $(x,y)$.
     (b) Projection $(x,\dot{x})$ of the solution.
     }
     {fig:exampleDa}
```

Listing 8.4: `Latex.tex`

To compile successfully the above code, `gnuplot` must be installed in the computer. Besides, during installation, the path environmental variable should be

added. If `TexStudio` is used to compile the Latex file, the `lualatex` and `PDFLatex` orders should be modified as follows:

```
pdflatex -synctex=1 -interaction=nonstopmode -shell-escape %.tex

lualatex.exe -synctex=1 -interaction=nonstopmode -shell-escape %.tex
```

The results are shown in figure 8.2.

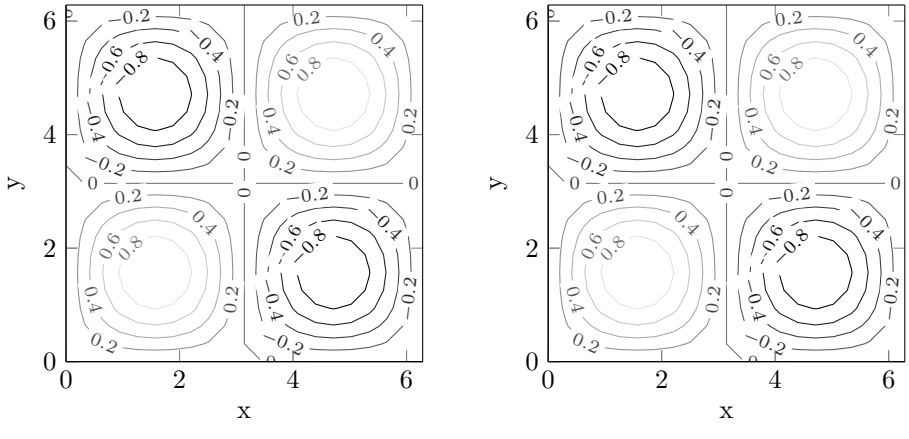

Figure 8.2: Isolines. (a) $z = \sin x \sin y$. (a) $z = \sin x \sin y$.

Bibliography

[1] M.A. Rapado Tamarit, B. Moreno Santamaría, & J.A. Hernández Ramos. Programming with Visual Studio: Fortran & Python & C++. Amazon Kindle Direct Publishing 2020.

[2] Lloyd N. Trefethen & D. Bau. Numerical linear algebra. SIAM 1977.

[3] William H. Press, Saul A. Teukolsky, William T. Vetterling & Brian P. Flannery. Numerical Recipes in Fortran 77, The Art of Scientific Computing Second Edition. Cambridge University Press 1992.

[4] E. Hairer, S. P. Norsett, G.Wanner. Solving Ordinary Differential Equations I: Nonstiff Problems, Second Revised Edition. Springer Series in Computational Mathematics 2008.

[5] E. Hairer, C. Lubich, G.Wanner. Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, Second Revised Edition. Springer Series in Computational Mathematics 1996.

[6] L. Shampine. Numerical solution of ordinary differential equations. Chapman & Hall 1994.

[7] J.A. Hernandez Ramos. Cálculo numérico en Ecuaciones Diferenciales Ordinarias. Second Edition Amazon Kindle Direct Publishing 2020.

[8] J.A. Hernandez Ramos & M. Zamecnik Barros. FORTRAN 95, programación multicapa para la simulación de sistemas físicos. Second Edition Amazon Kindle Direct Publishing 2019.

[9] J.R.Dormand & P.J.Prince. A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics Volume 6, Issue 1, March 1980, Pages 19-26.

[10] J.R.Dormand & P.J.Prince. High order embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics Volume 7, Issue 1, March 1981, Pages 67-75.

[11] Euaggelos E. Zotos. Classifying orbits in the classical Henon-Heiles Hamiltonian system. Nonlinear Dynamics (NODY), 2015, vol. 79, pp. 1665-1677.

[12] J. D. Lambert. Numerical Methods for ordinary Differential systems: The Initial Value Problem. John Wiley & Sons. May, 1993.