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)

December 16, 2018

Part I

User Manual

Chapter 1 Linear algebra

1.1 Overview

In this section, some of the classical problems of linear algebra shall be presented, all of them related to operations with matrices. The first of them is the solution of a linear system of algebraic equations, which will be solved by the LU method in the subroutine `LU_Solution`. The natural step to follow from this point is to extend the solution to a non linear system, which will be solved by the Newton method in the subroutine `Newton_Solution`. The eigenvalues problem for a matrix is considered in the subroutine `Test_Power_Method` in which they are computed by a recursive 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 embedded in the subroutine `Systems_of_Equations_examples` which can be called writing `call Systems_of_Equations_examples`.

```fortran
subroutine Systems_of_Equations_examples

    call LU_Solution
    call Newton_Solution
    call Test_Power_Method
    call Vandermonde_condition_number


end subroutine
```

Listing 1.1: `API_Example_Systems_of_Equations.f90`

1.2 LU solution example

As a first example of linear algebra the linear system

$$A \cdot x = b,$$

will be solved for $x \in \mathbb{R}^4$, where A and b 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 solve this problem numerically is the LU method. This method consists on the decomposition of the matrix A into two simpler matrices L and U for which A satisfies:

$$A = LU,$$

where L is a lower triangular matrix and U is an upper triangular matrix. This decompositions permits to calculate with a much lesser computational cost the solution x.

The implementation of this problem is done as follows:

```fortran
subroutine LU_Solution

    real :: A(4,4), b(4), x(4)


    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]

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

    write (*,*) 'The solution is = ', x

end subroutine
```

Listing 1.2: `API_Example_Systems_of_Equations.f90`

Once the problem is implemented, the computed 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 classical problem can be the approximated solution of a non linear system of equations, whose resolution must be calculated through an iterative method. The most iconic method for this task is the Newton-Rhapson method. To illustrate the use of this method an example of a vectorial non linear function $F : \mathbb{R}^3 \to \mathbb{R}^3$, whose components are $F = (F_1, F_2, F_3)$ such as:

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

The problem to be solved for (x, y, z) is the set of equations:

$$F = 0.$$

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

```fortran
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`

This function will be used as an input argument for the Newton method subroutine, and an initial approximation for the numerical solution shall be given.

```
subroutine Newton_Solution

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

    call Newton( F, x0 )

    write(*,*)  'Zeroes of F(x) are x = ', x0

end subroutine
```

Listing 1.4: `API_Example_Systems_of_Equations.f90`

Once the problem is implemented, the computed solution results:

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

1.4 Power method example

Another classical problem for square matrices is the calculus of eigenvalues and eigenvectors for normal matrices ($A \cdot A^T = A^T \cdot A$). The eigenvalues and eigenvectors problem for a square matrix A, consists on finding a scalar λ and a vector v such that:

$$(A - \lambda I) \cdot v = 0.$$

A method to compute the numerical solution of this problem is the power method, which gives back the eigenvalue of greater module and its associated eigenvector. The method consists on computing the nth power of the matrix following a recursive procedure.

$$v^{n+1} = \frac{v^n}{\|Av^n\|},$$

whenever the iteration n tends to infinity, calling $x = v^{n+1}$, the following relations are satisfied for the maximum module eigenvalue and its associated eigenvector:

$$x \to v, \qquad \frac{x^T A x}{\|x\| \|x\|} \to \lambda.$$

It is important to highlight that this method does not work for eigenvalues which satisfy $|\lambda| = 1$, for non real eigenvalues, and for non normal matrices.

For example if we considered the matrix:

$$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},$$

its easy to prove that its eigenvalues are $\lambda = \pm i$. However if the power method is applied to this matrix, the result given is that its eigenvalue of maximum module is $\tilde{\lambda} = 1$, which is an erroneous solution. Hence, these considerations must be taken in account to obtain a proper solution for the eigenvalues and eigenvectors if the power method is going to be used.

The example considiered will be the calculation of the eigenvalues and eigenvectors from the matrix:

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

The implementation of the problem is done as follows:

```
subroutine Test_Power_Method

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


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

      U = 1

      call power_method(A, lambda, U)

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

      B = matmul( A, transpose(A) )

      B = matmul( transpose(A), A )

      U = [-1, 0,  1 ]

      call power_method(A, lambda, U)

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

      U = [1, 2,  1 ]

      call power_method(A, lambda, U)

end subroutine
```

Listing 1.5: `API_Example_Systems_of_Equations.f90`

Once the problem is implemented the computed solution for the eigenvalues is:

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

and the associated eigenvectors are:

$$v_1 = \begin{pmatrix} 0.5773 \\ 0.5773 \\ 0.5773 \end{pmatrix}, \qquad v_2 = \begin{pmatrix} -0.7071 \\ -8.5758 \cdot 10^{-13} \\ 0.7071 \end{pmatrix}, \qquad v_3 = \begin{pmatrix} 0.5773 \\ 0.5773 \\ 0.5773 \end{pmatrix},$$

Notice that as λ_3 is null, the third eigenvector v_3 is the same as the first eigenvector v_1.

1.5 Condition number example

Another interesting topic for matrices is the condition number of a matrix, which is linked to its SVD decomposition.

In order for the reader to understand the motivation of this concept let us be a linear system of equations such as:

$$A \cdot x = b,$$

where x, b are vectors from the vectorial space V, for which is defined the norm $\|\cdot\|$ and A is a square matrix.

If an induced norm is defined for the matrices the previous equation can give us a measurable relation from the system of equations. In particular we define the quadratic norm for square matrices as:

$$\|A\| = \sup \frac{\|A \cdot v\|}{\|v\|},, \qquad \forall\ v \neq 0 \in V.$$

In these conditions, the following order relation is satisfied:

$$\|b\| \leq \|A\| \|x\|.$$

Given the linearity of the system, if the vector b is perturbed with a perturbation δb, the solution will be as well with the perturbation δx and if A is non singular, the following order relation is satisfied:

$$\|\delta x\| \leq \|A^{-1}\|\, \|\delta b\|.$$

Combining both order relations it is obtained an upper bound for the relative perturbation of the solution, that is:

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \frac{\|\delta b\|}{\|b\|},$$

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 quadratical 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.

An example of the computation of the condition number, shall be given for the 6×6 Vandermonde matrix A.

```fortran
subroutine Vandermonde_condition_number

    integer :: i, j, k
    integer, parameter :: N = 6
    real :: A(0:N, 0:N), sigma(0:N), U(0:N, 0:N), V(0:N, 0:N)
    real :: A_SVD(0:N, 0:N), D(0:N, 0:N)
    real :: kappa



    do i=0, N; do j=0, N
        A(i,j) = (i/real(N))**j
    end do; end do


    call SVD(A, sigma, U, V)
    D = 0
    do i=0, N
        D(i,i) = sigma(i)
    end do


    A_SVD = matmul(U , matmul( D ,  transpose(V) ) )

    call print_matrix("A", A)
    call print_matrix("A_SVD", A_SVD)
    call print_matrix("D", D)

    kappa = Condition_number(A)

    write(*,*) " Condition_ number =", sigma(0)/sigma(N)
    write(*,*) " Condition_ number =", kappa

end subroutine
```

Listing 1.6: `API_Example_Systems_of_Equations.f90`

Once implemented and executed, the result given for the condition number is:

$$\kappa(A) = 36061.16067,$$

which indicates that the Vandermonde matrix is *ill-conditioned*.

Chapter 2

Lagrange interpolation

2.1 Overview

In this chapter, the Lagrange polynomial interpolations are discussed for equispaced and non uniform grid or nodal points. A set of nodal points with their images is considered to build a polynomial interpolant. Examples are shown to alert of numerical problems associated to ill posed interpolation problems based on equispaced grid points. At the end of this chapter, the solution based on Chebyshev points is shown. A main program including different examples of interpolation issues is shown in the next listing.

```fortran
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 Chebyshev_expansion( t_kind = 1 )
     call Chebyshev_expansion( t_kind = 2 )

end subroutine
```

Listing 2.1: `API_Example_Lagrange_interpolation.f90`

All functions and subroutine 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 four points is given:

$$x = \{0, 0.1, 0.2, 0.5\}$$

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

$$f = \{0.3, 0.5, 0.8, 0.2\}$$

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 by calling the function: `Interpolated_value`

```
subroutine Interpolated_value_example

    real :: x(4) = [ 0.0, 0.1, 0.2, 0.5 ]
    real :: f(4) = [ 0.3, 0.5, 0.8, 0.2 ]

    real :: xp = 0.15
    real :: yp1, yp2


    yp1 = Interpolated_value( x , f , xp )
    yp2 = Interpolated_value( x , f , xp, 3)

    write (*,*) 'The interpolated value (order 2)  at xp is = ', yp1
    write (*,*) 'The interpolated value (order 3)  at xp is = ', yp2

end subroutine
```

Listing 2.2: `API_Example_Lagrange_interpolation.f90`

The first argument of the 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:

$$\{\ (x_i, f_i), \qquad i = 0, \ldots, 3\}$$

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

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

```
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)

   call scrmod("reverse")
   call qplot( xp, I_N(0, :), M+1) ! plot the Interplant
   call qplot( xp, I_N(1, :), M+1) ! first derivative of the interpolant


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. On figure 2.1 , the interpolant and its first derivative are plotted.

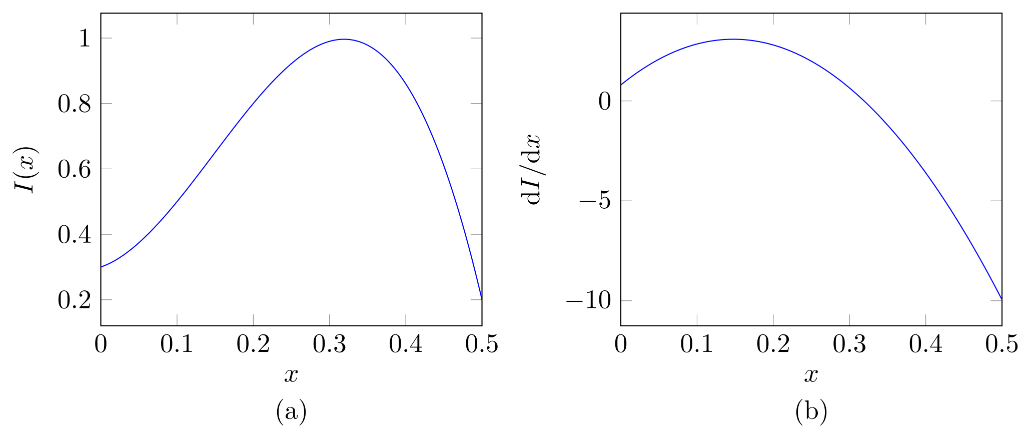


Figure 2.1: Lagrange interpolation with 10 nodal points. (a) Interpolant function. (b) First derivative of the interpolant

2.4 Integral of a function

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

$$I_0 = \int_{-1}^{+1} \sin(\pi x) dx.$$

To carry out this definite integral an interpolant should be built and later by integrating this interpolant to obtain the required value. The interpolant should a piecewise polynomial interpolation of order $q < N$ or it can a unique interpolant of order N. The function `Integral` has three arguments: nodal points,images of the function in these nodal points and the order of the interpolant.

The following subroutine determines the integral value and the results is compared with the exact value to know the error this numerical integration is done.

```fortran
subroutine Integral_example

 integer, parameter :: N=6
 real :: x(0:N), f(0:N)
 real :: 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

end subroutine
```

Listing 2.4: `API_Example_Lagrange_interpolation.f90`

2.5 Lagrange polynomials

A polynomial interpolation $I_N(x)$ can be expressed in terms of the Lagrange functions $(\ell_j(x))$ 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 the nodal point 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.

In this section, the Lagrange polynomial are plotted for $M = 400$ for a equispaced grid of $N = 4$ nodal points. The function `Lagrange_polynomials` allows to calculate all Lagrange polynomials together with their derivatives:

$$\{\ell_0(x), \ \ell_1(x), \ldots, \ell_N(x)\}$$

$$\{\ell'_0(x), \ \ell'_1(x), \ldots, \ell'_N(x)\}$$

The two arguments of the `Lagrange_polynomials` function are: the nodal points x_j and the point in which Lagrange polynomials and their derivatives are evaluated x_{pj}.

```fortran
subroutine Lagrange_polynomial_example

 integer, parameter :: N=4, M=400
 real :: x(0:N), f(0:N), xp(0:M)
 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)

 integer :: i
 real :: a=-1, b=1

 x  = [ (a + (b-a)*i/N, i=0, N) ]
 f = 1
 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 )

end subroutine
```

Listing 2.5: `API_Example_Lagrange_interpolation.f90`

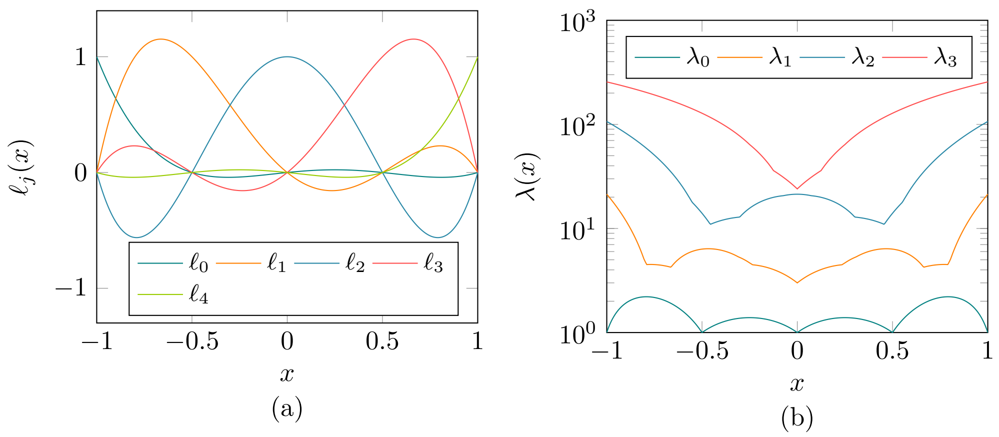


Figure 2.2: First five lagrange polynomials $\ell_j(x)$ and Lebesgue functions $\lambda_j(x)$. (a) Lagrange polynomials $\ell_j(x)$ for $j = 0, 1, \ldots, 4$. (b) Lebesgue functions $\lambda_j(x)$ for $j = 0, 1, \ldots, 3$.

On Figure 2.2 the lagrange polynomials are exposed. For each interpolant the point x_i in which every polynomial ℓ_j with $j \neq i$ annuls and $\ell_i(x_i) = 1$ is highlighted.

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 yields big errors in the interpolation result. In this section an example for the inoffensive $f(x) = \sin(\pi x)$ is plotted to show how the interpolated function with $N = 64$ nodal points has noticeable errors at boundaries. Since the program is compiled in double precession, the round–off is of order 10^{-15}. The sensitivity of the ill– posed problem is measured by the Lebesgue function which is defined by:

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

The function `Lebesgue_functions` is used to analyze the round–off errors at boundaries. It has two arguments: the nodal points x_j and the set points in which Lebesgue functions and their derivatives are evaluated x_{pj}.

```fortran
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 )

 call scrmod("reverse")
 call qplot( xp, I_N(0, :), M+1)
 write(*,*) " maxval Lebesgue =", maxval( Lebesgue_N(0,:) )

end subroutine
```

Listing 2.6: `API_Example_Lagrange_interpolation.f90`

On figure 2.3 it can be seen how the round-off error increases significantly nearby the boundaries, $x = \pm 1$ and the value of this error is related to the value of the Lebesgue function, whose maximum value is $4.393 \cdot 10^{16}$.

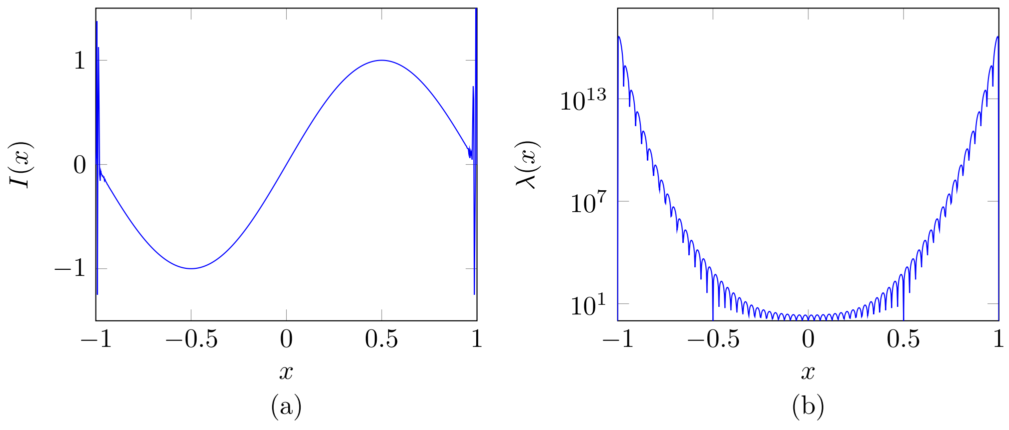


Figure 2.3: (a) Ill posed interpolation with an equispaced grid. (b) Lebesgue function $\lambda(x)$ for the interpolation.

2.7 Lebesgue function and error function

The Lagrange interpolation error has two main contributions: the round-off error measured by the Lebesgue function $\lambda_N(x)$ and the truncation error with the following expression:

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

where π_{N+1} is a polynomial of degree $N+1$ that vanishes in all nodal points.

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.

The function `PI_error_polynomial` allows to compute the error function $\pi_{N+1}(x)$ from the nodal points evaluated in an arbitrary set of different points.

```fortran
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 )

  call scrmod("reverse")

  do k=0, 2       ! k=0 function, k=1 first derivative, k=2 second derivative
    call qplot( xp, Lebesgue_N(k, :), M+1)
    call qplot( xp, PI_N(k, :), M+1)
  end do

end subroutine
```

Listing 2.7: `API_Example_Lagrange_interpolation.f90`

On figures 2.4, 2.5, 2.6 are shown the error functions $\pi_N(x)$ and the Lebesgue functions $\lambda_N(x)$ for $N = 1, 2, 3$.

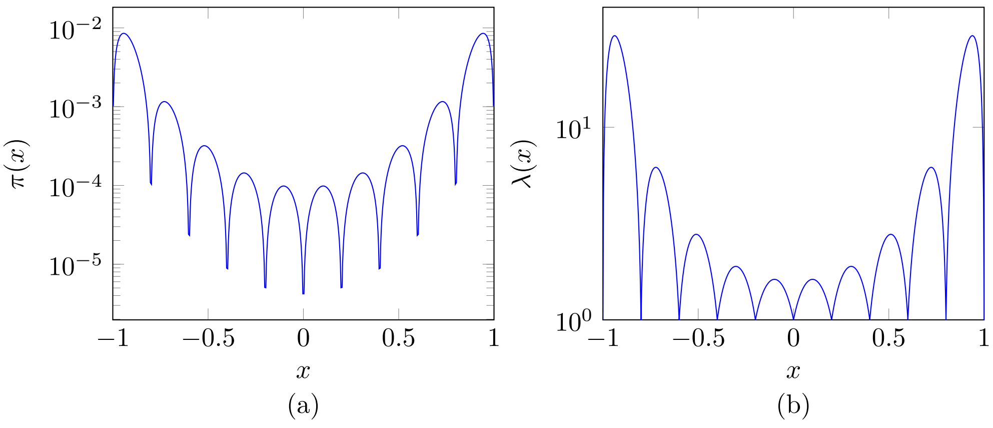


Figure 2.4: Error function $\pi_N(x)$ and Lebesgue function $\lambda_N(x)$ for $N = 10$. (a) Function $\pi_N(x)$. (b) Lebesgue function $\lambda_N(x)$

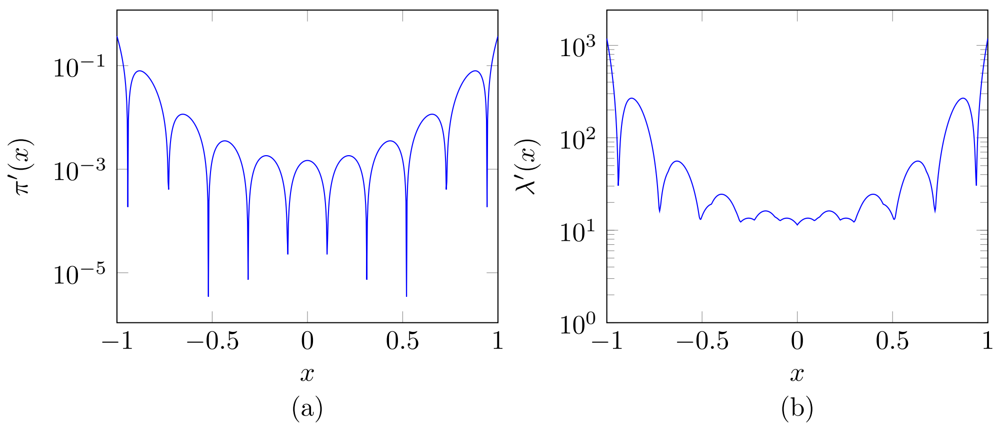


Figure 2.5: First derivative of the error function $\pi'_N(x)$ and Lebesgue function $\lambda'_N(x)$ for $N = 10$. (a) Function $\pi'_N(x)$. (b) Lebesgue function $\lambda'_N(x)$

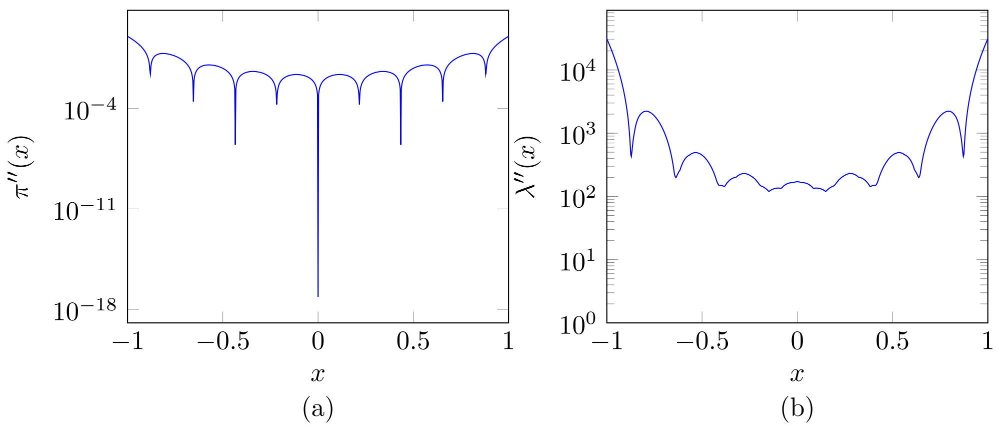


Figure 2.6: Second derivative of the error function $\pi''_N(x)$ and Lebesgue function $\lambda''_N(x)$ for $N = 10$. (a) Function $\pi''_N(x)$. (b) Lebesgue function $\lambda''_N(x)$

2.8 Chebyshev polynomials

There are basically two ways of approximate functions: by interpolants or by series expansions. Once a orthogonal basis of infinite dimensional space is chosen, the approximated function can be expressed as:

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

where $\phi_k(x)$ are the basis functions and $\hat{c}_k$ are the projections of $f(x)$ through basis functions. The are some special orthogonal basis very noticeable like Chebyshev polynomials. The first kind:

$$T_k(x) = \cos(k\theta),$$

and second kind :

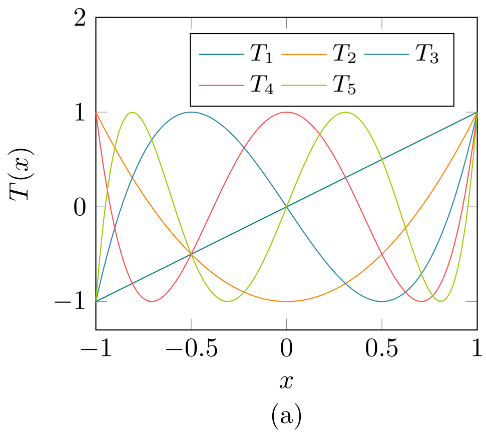
$$U_k(x) = \frac{\sin(k\theta)}{\sin\theta},$$

with $\cos(\theta) = x$.

In this section, Chebyshev functions are plotted to different values of k. The function `Chebyshev` has three arguments: the first one is the kind of the Chebyshev polynomial, the second argument is k and the third one is independent variable x.

```
subroutine Chebyshev_polynomials

    integer, parameter :: N = 100
    integer, parameter :: M = 5
    real :: x(0:N), y1(0:N), y2(0:N)
    real :: x0=-0.99, xf= 0.99
    integer :: i, k

    call scrmod("reverse")

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

    do k=1,  M
        y1 = Chebyshev( 1, k, x)
        y2 = Chebyshev( 2, k, x);
        call qplot(x, y1, N+1 )
        call qplot(x, y2, N+1 )
    end do

end subroutine
```

Listing 2.8: `API_Example_Lagrange_interpolation.f90`



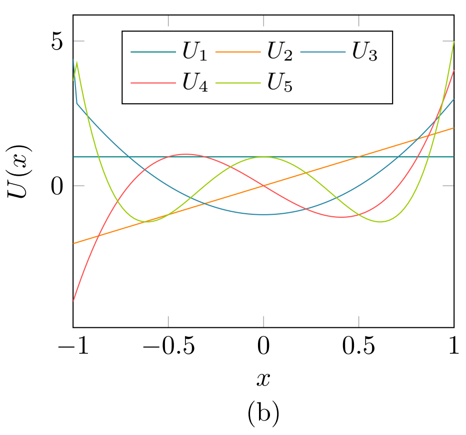


Figure 2.7: First kind and second kind Chebyshev polynomials. (a) First kind Chebyshev polynomials $T_k(x)$. (b) Second kind Chebyshev polynomials $U_k(x)$.

2.9 Chebyshev expansion and Lagrange interpolant

To palliate the ill posed behavior of the Lagrange interpolation in equispaced grids, non uniform grids are used to interpolate functions. In this section, Chebyshev extrema

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

are used to interpolate the function $f(x) = \sin(\pi x)$ over the domain $x \in [-1, 1]$.

The previous results are computed through the following implementation:

```
subroutine Chebyshev_expansion( t_kind )
    integer, intent(in ) :: t_kind

    integer, parameter :: N = 6, M = 500
    real :: x(0:N), f(0:N)
    real :: I_N(0:N, 0:M)
    real :: xp(0:M), yp(0:M), fp(0:M), Integrand(0:M)

    integer :: i, k
    real :: c_k
    real :: a=-0.999999, b = 0.999999, ymax, gamma

    xp = [ (cos( (PI/2 + PI*i)/(M+1)  ), i=M, 0, -1) ]
    fp = f_Chebyshev( xp )

    yp = 0
    do k=0, N

        Integrand = fp * Chebyshev( t_kind, k, xp ) * w_Chebyshev( t_kind, xp
    )

        if (k==0) then;    gamma = 1 / PI;
                  else;    gamma = 2 / PI;
        end if

        c_k = Integral( xp , Integrand )  * gamma

        yp = yp + c_k * Chebyshev( t_kind, k, xp)
                                   ! Truncated Chebyshev series
    end do

    if (t_kind == 1) then
                         x = [ (cos( (PI/2 + PI*i)/(N+1)  ), i=N, 0, -1) ]
    else if ( t_kind == 2) then
                         x = [ (cos(PI*i/N), i=N, 0, -1) ]
    end if

    f = f_Chebyshev( x )

    I_N = Interpolant(x, f, N, xp) ! Equal to Discrete Chebyshev series

end subroutine
```

Listing 2.9: `API_Example_Lagrange_interpolation.f90`

On figure 2.8 the interpolation of the function $f(x)$ is exposed using both truncated and discrete Chebyshev series and the error in the interpolation as well.

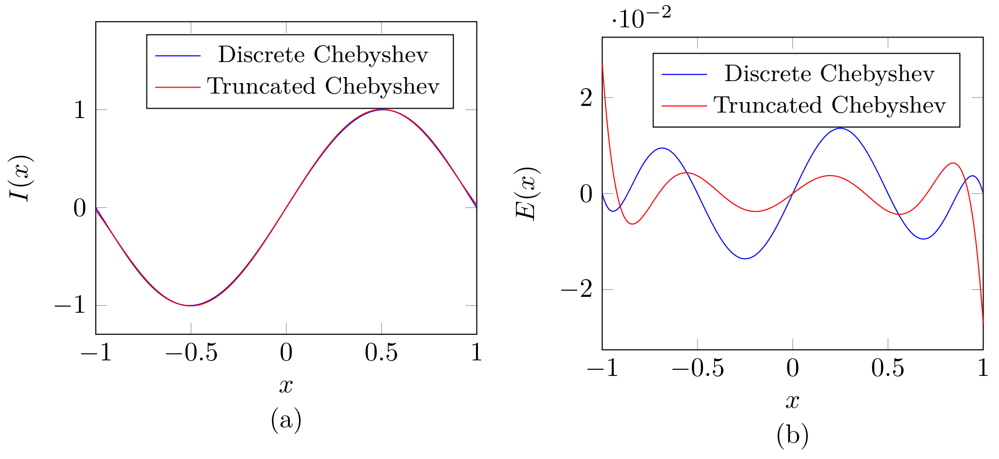


Figure 2.8: Chebyshev discrete expansion and truncated series. (a) Chebyshev expansion. (b) Chebyshev expansion error.

Chapter 3

Finite Differences

3.1 Overview

Whenever is necessary to approximate the value of derivatives of a function, the use of finite differences is mandatory. In this chapter, several examples of the use of finite differences will be presented. To start, the derivatives of a function $f : \mathbb{R} \to \mathbb{R}$ are computed in the subroutine `Derivative_function_x` and also its error is evaluated. The analogous operation for a function $f : \mathbb{R}^2 \to \mathbb{R}$ is performed in the subroutine `Derivative_function_xy`. Then, the influence of the truncation and round off errors is highlighted in the subroutine `Derivative_error`. Finally, as a very common use of finite differences is its application in the spatial discretization of differential equations, the resolution of a boundary value problem is presented in `BVP_FD`. All these subroutines are embedded in the subroutine `Finite_difference_examples`.

```
subroutine Finite_difference_examples


   call Derivative_function_x
   call Derivative_function_xy
   call Derivative_error
   call BVP_FD


end subroutine
```

Listing 3.1: `API_Example_Finite_Differences.f90`

3.2 Example: Derivatives of a 1D function

In order to illustrate how the `Finite_Differences` module works an example will be given. Several derivatives of a known function will be computed by fourth order finite differences and compared to the analytical derivatives. The function will be:

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

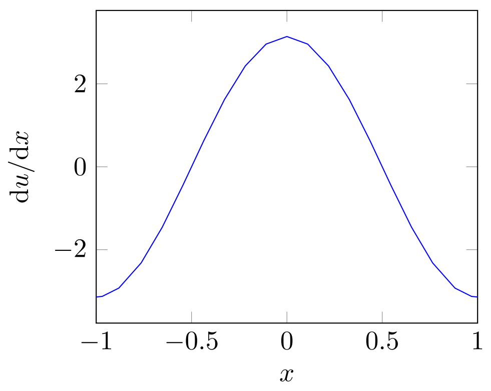
For example, we shall calculate the derivatives $(\mathrm{d}u/\mathrm{d}x, \mathrm{d}^2u/\mathrm{d}x^2)$. This derivatives are expressed analitically as:

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \pi \cos(\pi x), \qquad \frac{\mathrm{d}^2u}{\mathrm{d}x^2} = -\pi^2 \sin(\pi x).$$

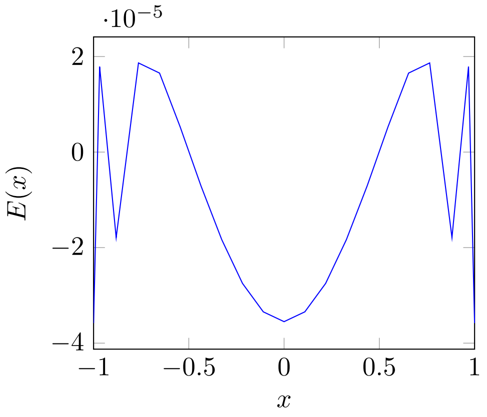
The implementation of these calculations is done as follows:

```fortran
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), ux(0:Nx), uxx(0:Nx), ErrorUx(0:Nx), ErrorUxx(0:Nx)

    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 , ux )
    call Derivative( 'x' , 2 , u , uxx )


    ErrorUx =  ux - pi* cos(pi * x)
    ErrorUxx = uxx + pi**2 * u

    call scrmod("reverse")
    call qplot(x, ErrorUx , Nx+1)
    call qplot(x, ErrorUxx, Nx+1)



end subroutine
```

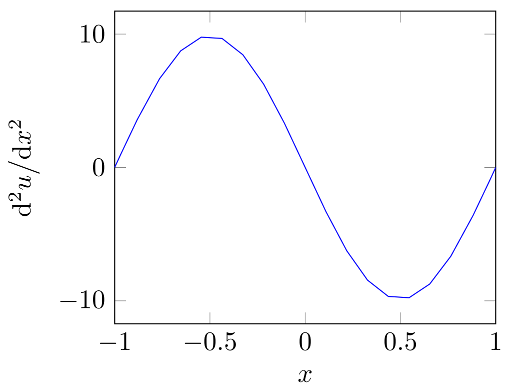
Listing 3.2: `API_Example_Finite_Differences.f90`



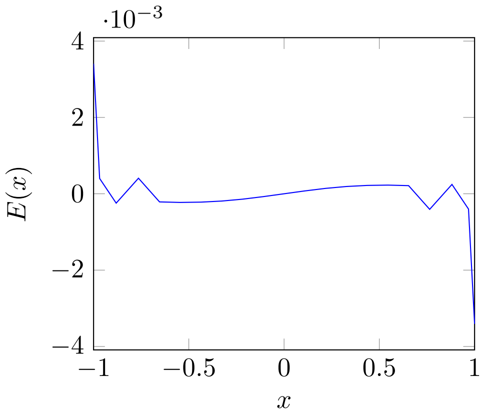
(a)



(b)



(c)



(d)

Figure 3.1: Numerical derivatives $\mathrm{d}u/\mathrm{d}x$, $\mathrm{d}^2u\big/\mathrm{d}x^2$ and error on their computation for 21 nodal points and order $q = 4$. (a) Numerical derivative $\mathrm{d}u/\mathrm{d}x$. (b) Error on the calculation of $\mathrm{d}u/\mathrm{d}x$. (c) Numerical derivative $\mathrm{d}^2u\big/\mathrm{d}x^2$. (d) Error on the calculation of $\mathrm{d}^2u\big/\mathrm{d}x^2$.

3.3 Example: Derivatives of a 2D function

In order to illustrate how the *Finite_Differences* module works an example will be given. Several derivatives of a known function will be computed by fourth order finite differences and compared to the analytical derivatives. The function will be:

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

For example, we shall calculate the derivatives $(\partial^2 u/\partial x^2, \partial^2 u/\partial x \partial y)$. This derivatives are expressed analitically as:

$$\frac{\partial^2 u}{\partial x^2} = -\pi^2 \sin(\pi x) \sin(\pi y), \qquad \frac{\partial^2 u}{\partial x \partial y} = \pi^2 \cos(\pi x) \cos(\pi y).$$

The implementation of these calculations is done as follows:

```fortran
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 )


    do i=0, Nx; do j=0, Ny
        u(i,j) = sin(pi * x(i)) * sin(pi * y(j))
    end do; end do

    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

    do i=0, Nx; do j=0, Ny
       Erroruxy(i,j) = uxy(i,j) - pi**2 * cos(pi*x(i))  * cos(pi*y(j))
    end do; end do

end subroutine
```

Listing 3.3: `API_Example_Finite_Differences.f90`

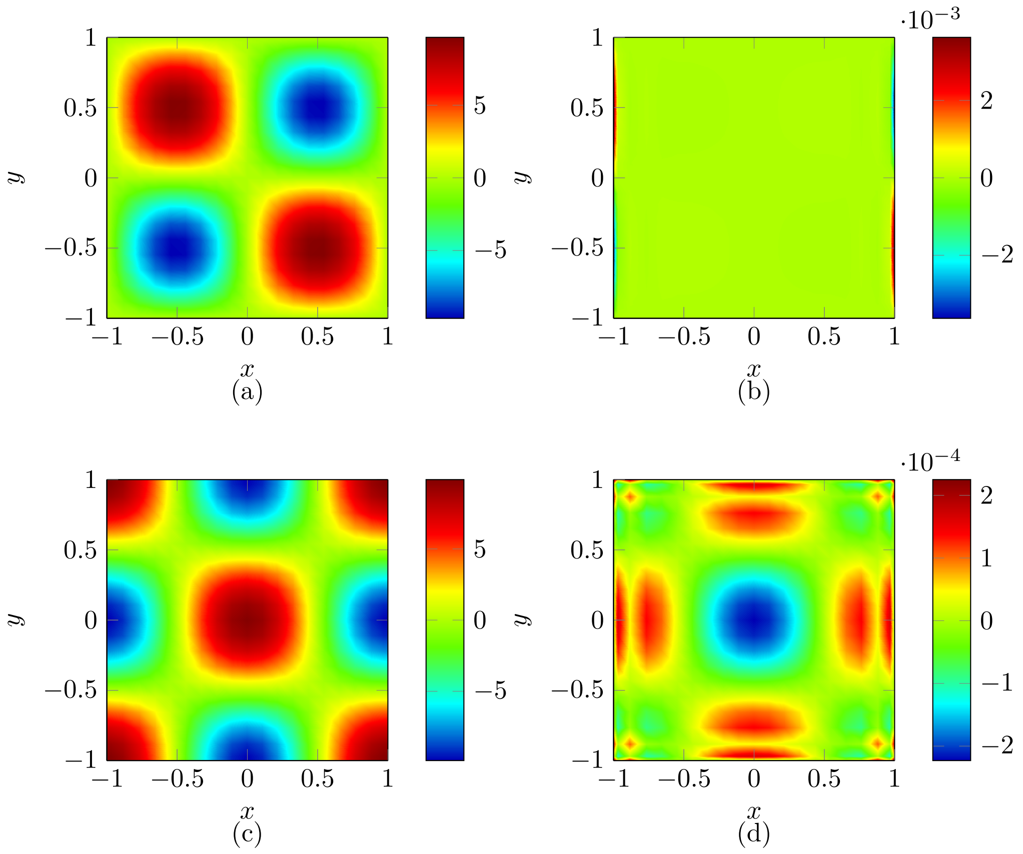


Figure 3.2: Numerical derivatives $\partial^2 u / \partial x^2$, $\partial^2 u / \partial x \partial y$ and error on their computation for 21×21 nodal points and order $q = 6$. (a) Numerical derivative $\partial^2 u / \partial x^2$. (b) Error on the calculation of $\partial^2 u / \partial x^2$. (c) Numerical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

The computed finite differences and the error in the approximation are represented on the Figure 3.2

3.4 Truncation and Round-off errors of derivatives

In order to analyse the effect of the truncation and round-off error produced in the approximation of a derivative by finite differences, the following example will be considered.

Let's suppose that we intend to calculate the derivatives of a known function, $f(x)$ in $x \in [-1, 1]$, for example, $f(x) = \cos(\pi x)$.

When computed this function numerically, as the precision of the machine is finite, the obtained value is disturbed by a function $\varepsilon(x)$ which is of a certain order $\mathcal{O}(10^{-m})$, for example, $m = 12$. In these conditions, the function computed by the machine will be:

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

It is expected that when the mesh size Δx is reduced, the error will be as well. This holds true for a certain region of the mesh size, in which the error committed by truncation overcomes the round-off error. However, when a certain mesh size value is achieved, the error grows when the mesh size is reduced. This is due to the round-off error, which prevents the error to keep reducing. As the precision of the machine is finite, in these region the machine starts to mishandle the values, which provokes the growing of the global error. This behavior can be 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, which indicates that the round-off error starts being relevant for larger mesh sizes.

The implementation of this example is written on the following listing.

```fortran
subroutine Derivative_error

 integer :: q = 2                    ! interpolant order
 integer :: N                        ! # of nodes (piecewise pol. interpolation)
 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),  log_dx(M)    ! Error versus Dx
 real :: epsilon = 1d-12             ! order of the random perturbation

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

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

 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) = log( abs(dIdx(p)-dfdx(p)) )

   deallocate( x, f, dIdx, dfdx )

 end do

 call scrmod("reverse")
 call qplot( log_dx, log_Error, M )

end subroutine
```

Listing 3.4: `API_Example_Finite_Differences.f90`

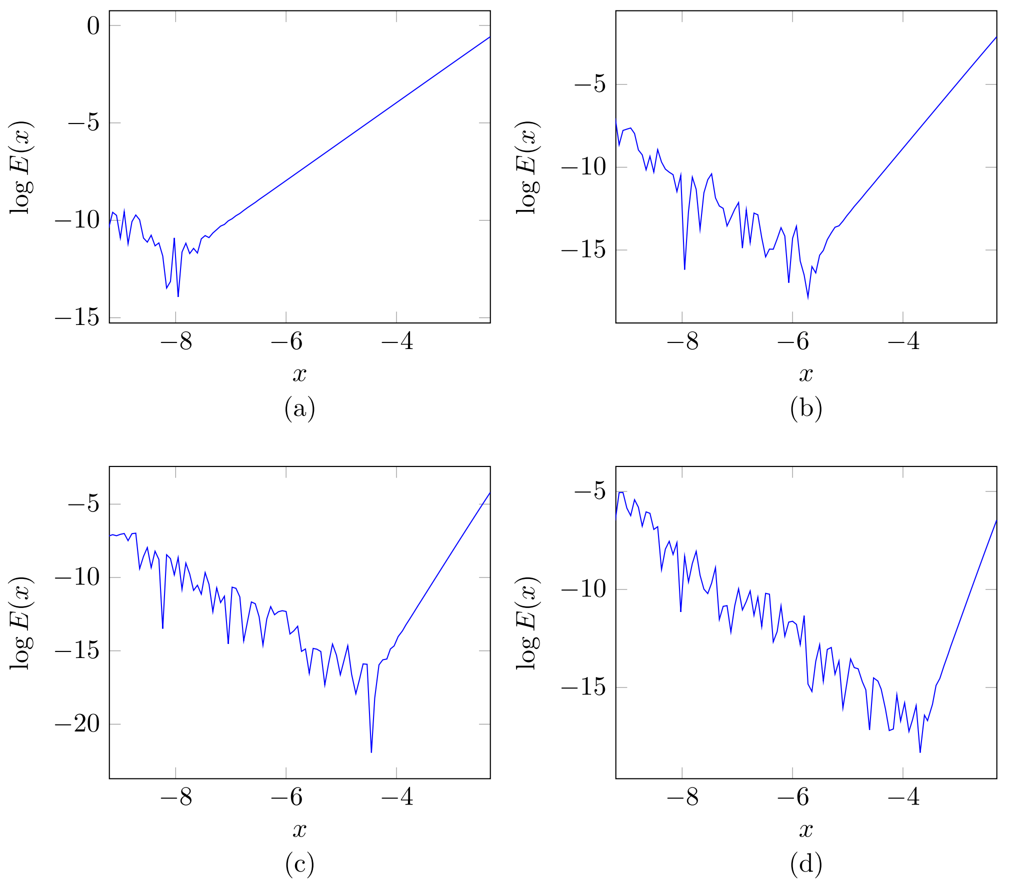


Figure 3.3: Numerical error on the computation of a second order derivative over a perturbed cosine evaluated at $x = -1$. (a) Error for $q = 2$. (b) Error for $q = 4$. (c) Error for $q = 6$. (d) Error for $q = 8$.

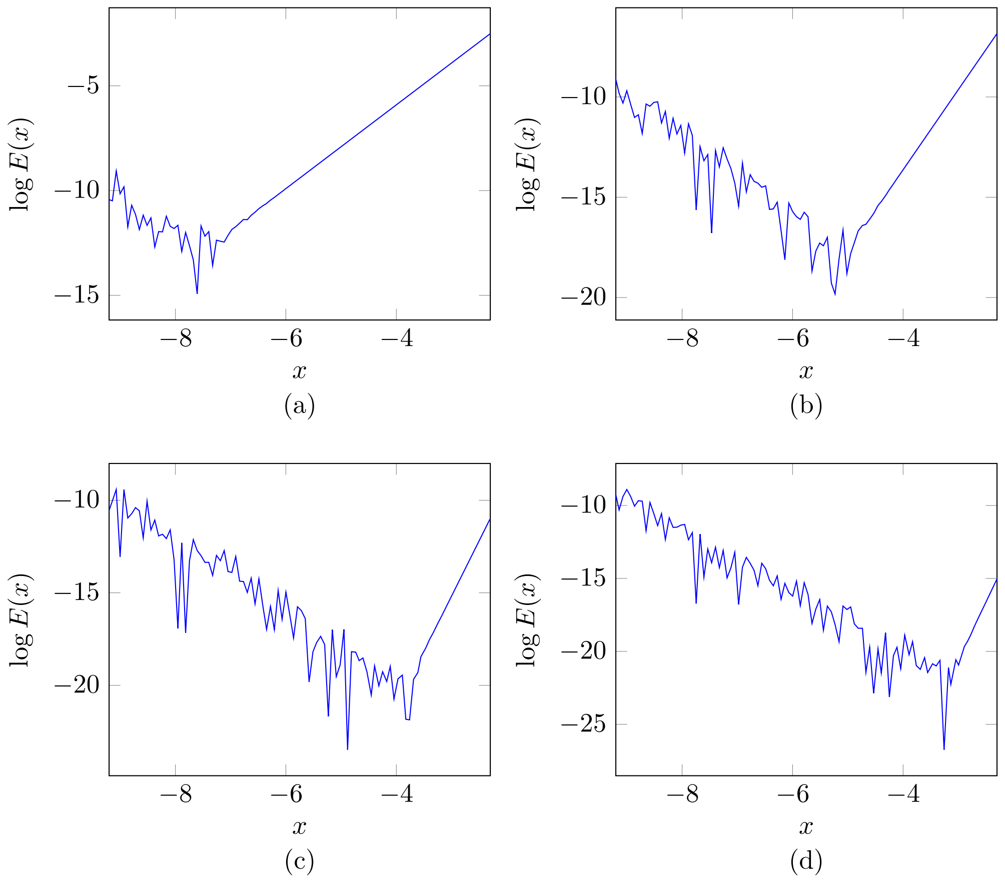


Figure 3.4: Numerical error on the computation of a second order derivative over a perturbed cosine evaluated at $x = 0$. (a) Error for $q = 2$. (b) Error for $q = 4$. (c) Error for $q = 6$. (d) Error for $q = 8$.

3.5 Example: Boundary value problem 1D

A more advanced use of the `Finite_Differences` module is the resolution of a boundary value problem using finite differences. To start, a 1D boundary value problem will be solved in the domain $x \in [-1,1]$. For historical reasons, the differential equation is chosen:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + u = 0,$$

with the boundary conditions:

$$u(-1) = 1, \qquad u(1) = 0.$$

The analytical solution of this problem is:

$$u(x) = A\cos(x) - B\sin(x),$$

where the coefficients are:

$$A = \frac{1}{2\cos(1)}, \qquad B = \frac{1}{2\sin(1)}.$$

The resolution algorithm of this problem by means of finite differences consists on three steps:

1. **Spatial discretization**.

   The first thing to do is defining the grid by selecting a discrete sucesion of points of $x \in [0, N]$:

   $$x_i = -1 + \frac{2i}{N}, \quad i = (0, 1, 2, \ldots N),$$

   where the grid size is $\Delta x = 2/N$.

   The spatial discretization entails also the definition of the solution as a sucesion:

   $$u_i = u(x_i), \quad (i = 0, 1, 2, \ldots N).$$

2. **Differential operator construction.**

Once the domain and the solution are discretized, the derivatives are approximated by finite differences of order q. In general, this finite differences can be seen a linear mapping for the solution in all the discrete points of the domain. Hence, finite differences for a derivative of order n, D_n in the most general case are functions $D_n : \mathbb{R}^{N+1} \to \mathbb{R}$ and the error approximating the derivatives by finite differences of order q is proportional to Δx^q. For the sake of simplicity, in this example the order $q = 2$ will be chosen. Once the derivatives are computed by finite differences, the differential equation for the inner points of the grid can be written:

$$D_2(u_{i-1}, u_i, u_{i+1}) + u_i = 0, \quad (i = 1, 2, \dots N-1).$$

To complete the algebraic system of equations to be determined the boundary conditions must be added which for this problem are:

$$u_0 - 1 = 0, \qquad u_N = 0.$$

Hence, the difference equation system resulting from the differential operator plus the boundary conditions constitute an algebraic system of $N+1$ equations and $N+1$ unknown variables u_i.

3. **Algebraic system of equations resolution.**

The previous step has transformed the differential equation into a set of algebraic equations which must be solved to obtain the succession u_i which is the numerical solution of the problem. This shall be done by means of an iterative method such as Newton-Rhapson.

Once the grid is defined in the interface of the subroutine, a subroutine that computes the difference equation is contained in it, performing the step 1:

```fortran
subroutine Difference_equation ( x, W, F)
   real, intent(in) :: x(0:Nx), W(0:Nx)
   real, intent(out):: F(0:Nx)

   real :: Wxx(0:Nx)

   call Derivative( "x" , 2 , W , Wxx )   ! Derivative computation

   F = Wxx + W    ! D.O. on inner points
   F(0) = W(0) - 1    ! B.C. on x = -1
   F(Nx) = W(Nx)      ! B.C. on x =  1

end subroutine
```

Listing 3.5: `API_Example_Finite_Differences.f90`

To use the subroutine that solves algebraic equations by the Newton-Rhapson method, an intermediate function is defined. The only purpose of this function is to make the difference operator computed suitable as an input argument.

```fortran
function System_BVP(U) result(F)
               real, intent (in) :: U(:)
               real :: F(size(U))

               real :: UU(0:Nx,0:Ny), FF(0:Nx,0:Ny)

               UU = reshape( U, [Nx+1, Ny+1] )

               call Difference_equation(x_nodes, y_nodes, UU, FF)

               F=reshape(FF, [ M ]);

end function
```

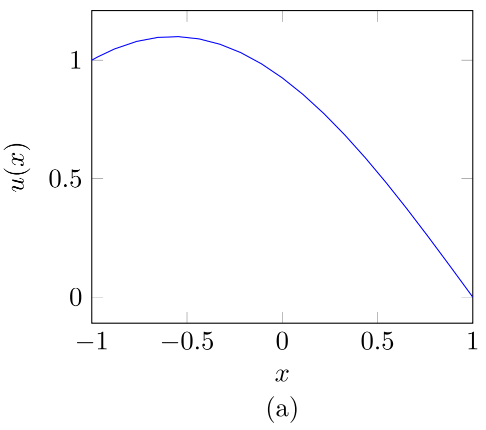
Listing 3.6: `API_Example_Finite_Differences.f90`

These two subprograms are contained on the interface of the subroutine, in which the spatial discretization is carried out, and is implemented as follows:

```fortran
subroutine BVP_FD

    integer, parameter :: Nx = 20,  Order = 6
    real :: x_nodes(0:Nx)        ! Grid definition
    real :: x0 = -1, xf = 1      ! Spatial domain
    integer :: i
    real :: pi = 4 * atan(1.)
    real :: u(0:Nx)              ! Solution u(x)

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

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

    u = 1                        ! Initial guess

    call Newton(System_BVP, u)   ! Difference equation
                                 ! resolution

    call qplot( x_nodes, u, Nx+1)! Solution plot

contains
```

Listing 3.7: `API_Example_Finite_Differences.f90`

Within the interface is also chosen the order of interpolation, which in this case is $q = 6$. Once implemented and executed the result obtained is represented on figure 3.5



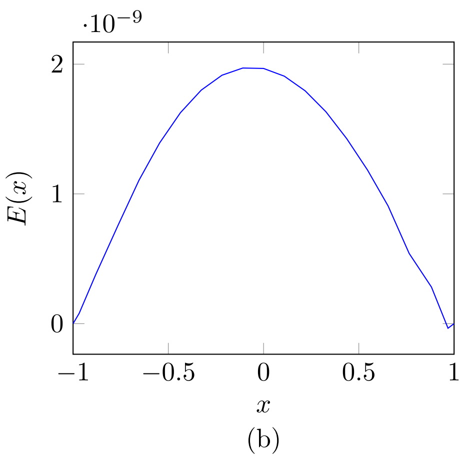


Figure 3.5: Numerical solution and error on the computation of the boundary value problem for $N = 20$ and order $q = 6$. (a) Numerical solution with order $q = 6$ of the boundary value problem. (b) Error of the solution with order $q = 6$ of the boundary value problem.

Chapter 4

Cauchy Problem

4.1 Overview

In this chapter, the implementation of several Cauchy problems will be exposed. The order in which the examples are ordered is in crescent complexity, starting by a scalar first order ordinary differential equation (ODE). Then, an example of a linear disipative spring which is equivalent to a system of differential equations will be exposed. Finally, the Lorenz equations which lead to the Lorenz attractor will be solved.

```fortran
subroutine Cauchy_problem_examples

   call First_order_ODE

   call Linear_Spring

   call Lorenz_Attractor

end subroutine
```

Listing 4.1: `API_Example_Cauchy_Problem.f90`

4.2 Example: First order ODE

The first example shown is a scalar first order ordinary differential equation (ODE). For example, it can be considered a constant coefficient linear equation such as:

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

with the initial condition $u(0) = 1$. The Cauchy problem formulated can represent, for example, the velocity along time of a punctual mass submitted to viscous damping.

This problem has an analytical solution which is written:

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

The implementation of the problem requires the definition of the differential operator as a function.

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

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

end function
```

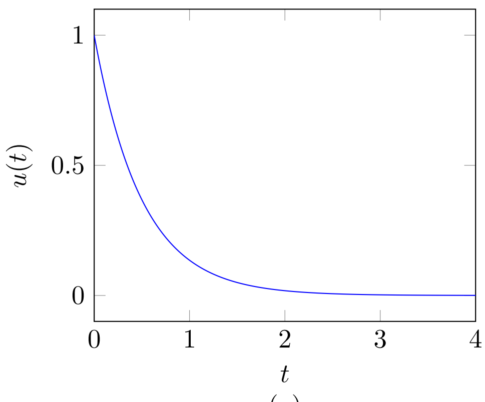
Listing 4.2: `API_Example_Cauchy_Problem.f90`

This function must be called by a subroutine as an input argument of the subroutine that computes the numerical solution.

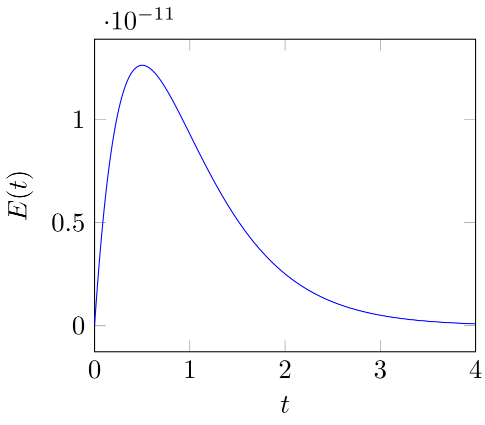
```fortran
subroutine First_order_ODE

    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, &
                          Scheme = Runge_Kutta4, Solution = U )

    call qplot( Time(0:N) , U(0:N,1) , N+1 )

contains
```

Listing 4.3: `API_Example_Cauchy_Problem.f90`

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



(a)



(b)

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.

4.3 Example: Linear spring

The second example to be exposed is a second order ordinary differential equation. It will be solved the vibrations a linear spring whose stiffness increases along time. The problem to be solved is formulated on a temporal domain defined by the set: $\Omega \subset \mathbb{R} : \{t \in [0,5]\}$. The displacement $u(t)$ of the spring is ruled by the equation:

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

To be solved, the equation must be written as 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:

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

It is necessary to give an initial condition of position and velocity. In this example a step on the initial displacement will be considered:

$$\begin{pmatrix} U_1 \\ U_2 \end{pmatrix}_{t=0} = \begin{pmatrix} 5 \\ 0 \end{pmatrix},$$

where $U_1(t)$ is referred to the position and $U_2(t)$ is referred to the velocity.

The implementation of the problem requires the definition of the differential operator as a function:

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

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

    real, parameter :: a = 3.0,b = 0.0

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

end function
```

Listing 4.4: `API_Example_Cauchy_Problem.f90`

This function is used as an input argument for the subroutine `Cauchy_ProblemS`.

```
subroutine Linear_Spring

    real :: t0 = 0, tf = 5
    integer :: i
    integer, parameter :: N = 100   !Time steps
    real :: 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,         &
                          Scheme = Crank_Nicolson , Solution = U )

    call qplot( time(0:N) , U(0:N,1) , N+1 )

contains
```

Listing 4.5: `API_Example_Cauchy_Problem.f90`

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 is satisfied and the oscillatory behaviour of the solution.

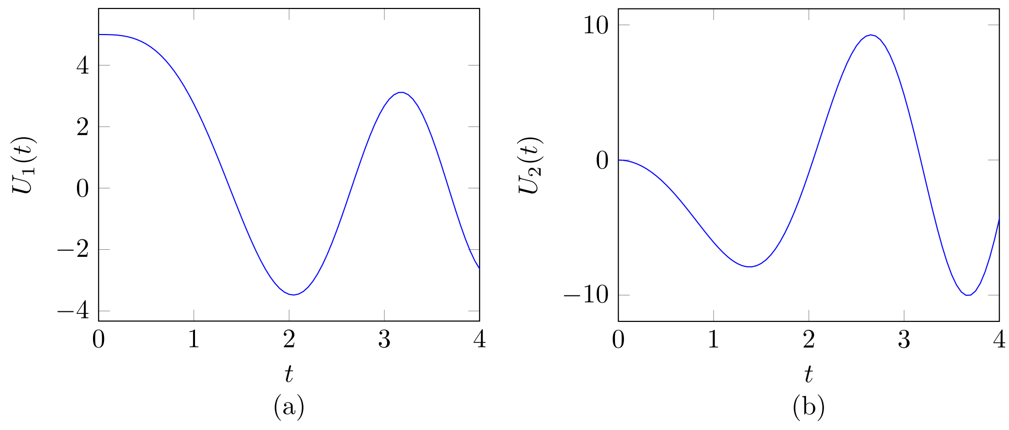


Figure 4.2: Numerical solution of the second order Cauchy Problem. (a) Numerical solution of the second order Cauchy problem. (b) Velocity of the solution along time.

4.4 Example: Lorenz Attractor

Another interesting example can be 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 behaviour along time. The behaviour of the equation's solution is chaotic for certain values of the parameters involved in the equation. The equations are written:

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

along with the initial condition:

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

The implementation of the problem requires the definition of the differential operator as a 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
```

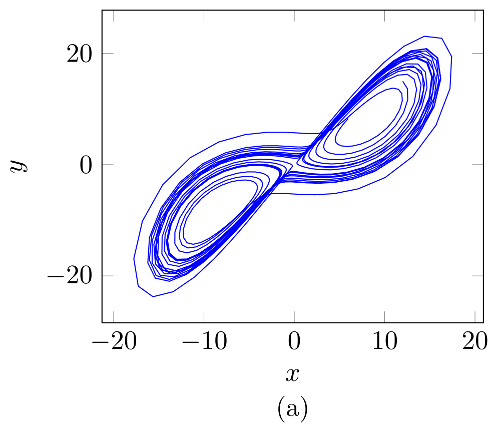
Listing 4.6: `API_Example_Cauchy_Problem.f90`

The previous function will be used as an input argument for the subroutine that solves the Cauchy Problem.

```
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(0,1) = 12
   U(0,2) = 15
   U(0,3) = 30


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

    call qplot(  U(0:N,1) , U(0:N,2) , N+1 )


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.



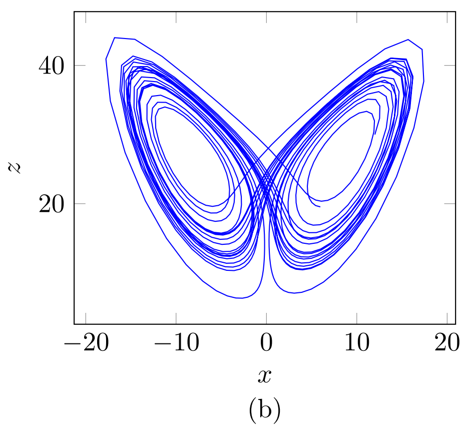


Figure 4.3: Solution of the Lorenz equations. (a) Phase plane (x, y) of the Lorenz attractor. (b) Phase plane (x, z) of the Lorenz attractor.

Chapter 5

Boundary Value Problems

5.1 Overview

In this chapter, the implementation of boundary value problems for its resolution will be explained. These problems can manifest in the shape of ordinary differential equations (ODES) for 1-dimensional spatial domains or in the shape of partial differential equations in 2-dimensional or 3-dimensional spatial domains. For each of these situations the unknown variable can be scalar or vectorial and both cases are included on the code. As it seems natural, the complexity of the examples will be progressive in the chapter. Starting by 1-dimensional scalar boundary value problems, followed by its vectorial extension. After that, 2-dimensional examples will maintain the same structure.

```fortran
subroutine BVP_examples

   call Legendre_1D
   call Linear_BVP2D

   call Linear_Plate_2D
   call Non_Linear_Plate_2D

end subroutine
```

Listing 5.1: `API_example_Boundary_Value_Problem.f90`

5.2 Example: Legendre equation

The first example consists on a boundary value problem for a 1D spatial domain. The selected problem is determined by the Legendre differential equation on a domain defined by the set $\Omega \subset \mathbb{R} : \{x \in [-1, 1]\}$:

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

where $n = 3$ and the boundary conditions are: $y(-1) = -1$ and $y(1) = 1$. This problem has an analytical solution, which is the third order Legendre polynomial:

$$y(x) = \frac{1}{2}(5x^3 - 3x).$$

The numerical solution will be represented in the following figure and as expected it is quite similar to the analytical one.

The implementation of the problem requires the definition of a function for the differential operator:

```fortran
real function Legendre(x, y, yx, yxx) result(L)

    real, intent(in) :: x, y, yx, yxx
    real, parameter :: n = 3.

  ! Legendre differential equation
    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 are also implemented as a function:

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

        real, intent(in) :: x, y, yx

        if (x==x0) then
                           BCs = y + 1
        elseif (x==xf) then
                           BCs = y - 1
        else
            write(*,*) " Error BCs x=", x
            write(*,*) " a, b=", x0, xf
            stop
        endif

    end function
```

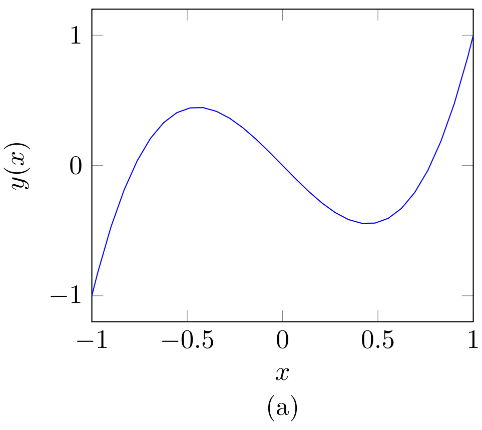
Listing 5.3: `API_example_Boundary_Value_Problem.f90`

These two functions must be contained on a subroutine which uses them as input arguments:

```
subroutine Legendre_1D

    integer, parameter :: N = 30, q = 6
    real :: x(0:N), U(0:N)
    real :: x0 = -1 , xf = 1
    integer :: i
    real :: pi = 4 * atan(1.0)

    x(0) = x0; x(N) = xf
    call Grid_Initialization( grid_spacing = "nonuniform", &
                             direction = "x",   q = q, nodes = x )

    call Boundary_Value_Problem( x_nodes = x, Order = q,              &
                                 Differential_operator = Legendre,    &
                                 Boundary_conditions   = Legendre_BCs, &
                                 Solution = U )
    call scrmod("reverse")
    call qplot(x, U, N+1)

contains
```

Listing 5.4: `API_example_Boundary_Value_Problem.f90`

Once the problem is implemented, compiled and executed, the numerical solution can be observed on figure 5.1(a). The error associated to the numerical computation of the solution is represented on figure 5.1(b).



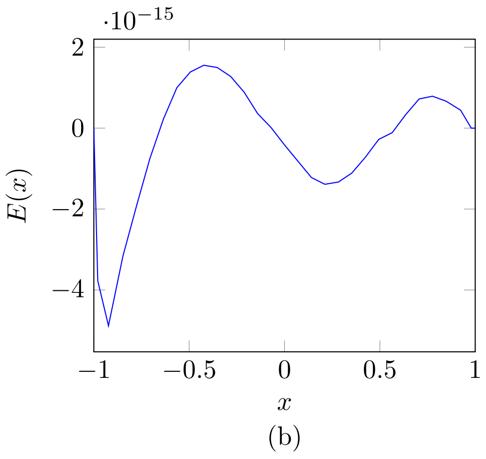


Figure 5.1: Legendre equation solution for $N + 1 = 31$ nodal points. (a) Numerical solution of the third order Legendre equation. (b) Error on the computation with finite differences of order $q = 6$.

5.3 Example: Linear Plate

An example of a boundary value problem can be a linear elastic plate submitted to a distributed load. The deflection w of a bending linear plate is ruled by the biharmonic equation. Considering a plate with simply supported edges with a distributed sinusoidal load, the problem in a domain $\Omega \subset \mathbb{R}^2$:$\{(x, y) \in [-1, 1] \times [-1, 1]\}$ is formulated as:

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

where $\nabla^4 = \nabla^2(\nabla^2)$ is the biharmonic operator. The simply supported boundary conditions on the four edges: $x = \pm 1$, $y = \pm 1$ mean that the displacement and bending moments at the edges annul. It can be proven that the null bending moment condition is equivalent to say that the laplacian of the displacement on the edges satisfies: $\nabla^2 w = 0$. The boundary conditions are represented on the figure below:

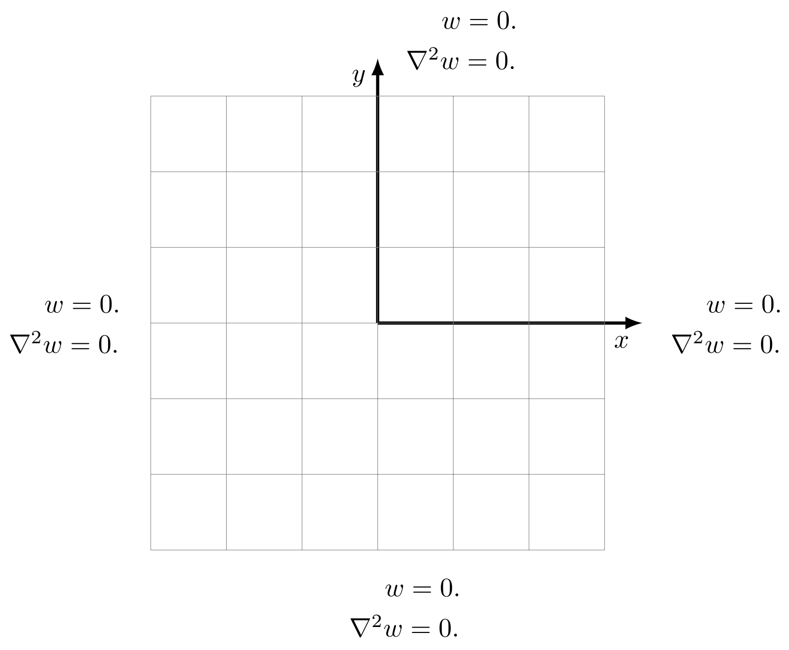


Figure 5.2: Simply supported boundary conditions for a linear plate.

The implementation of the problem requires the definition of a function for the differential operator:

```
function Linear_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 :: v, vxx, vyy, vxy, w, wxx, wyy, q


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

        q =  4*pi**4 * sin(pi*x) * sin(pi*y)

        L(1) =  vxx + vyy - w
        L(2) =  wxx + wyy - q



end function
```

Listing 5.5: `API_example_Boundary_Value_Problem.f90`

And also, the definition of the boundary conditions as another function:

```
function L_Plate_BCs(x, y, u, ux, uy) result(BCs)

        real, intent(in) :: x, y, u(:), ux(:), uy(:)
        real :: BCs(size(u))

        if (x==x0) then
            BCs(1) = u(1)
            BCs(2) = u(2)

        elseif (x==xf) then
            BCs(1) = u(1)
            BCs(2) = u(2)

        elseif (y==y0) then
            BCs(1) = u(1)
            BCs(2) = u(2)

        elseif (y==yf) then
            BCs(1) = u(1)
            BCs(2) = u(2)
        else
            write(*,*) " Error BCs x=", x
            stop
        endif

end function
```

Listing 5.6: `API_example_Boundary_Value_Problem.f90`

These two functions must be framed on a common environment that contains them.

Hence, they must be contained on a subroutine in which these functions are introduced as input arguments of the partial differential equations solver.

```
subroutine Linear_Plate_2D

    integer, parameter :: Nx = 20, Ny = 20, Nv = 2, q= 4
    real :: x(0:Nx), y(0:Ny), U(0:Nx, 0:Ny, Nv)
    real :: x0 = -1 , xf = 1 , y0 = -1  , yf = 1,  PI = 4* atan (1d0)
    integer :: i, j

    x(0) = x0 ; x(Nx) =  xf; y(0) = y0; y(Ny) = yf
    call Grid_Initialization( "nonuniform", "x", q, x )
    call Grid_Initialization( "nonuniform", "y", q, y )

    U = 1

    call Boundary_Value_Problem( x_nodes = x, y_nodes = y, Order = q,  &
                                 N_variables = Nv,                     &
                                 Differential_operator = Linear_Plate, &
                                 Boundary_conditions = L_Plate_BCs,    &
                                 Solution = U )

contains
```

Listing 5.7: `API_example_Boundary_Value_Problem.f90`

The contains command of the last line of the subroutine must be followed by the two previous functions defined for the differential operator and the boundary conditions.

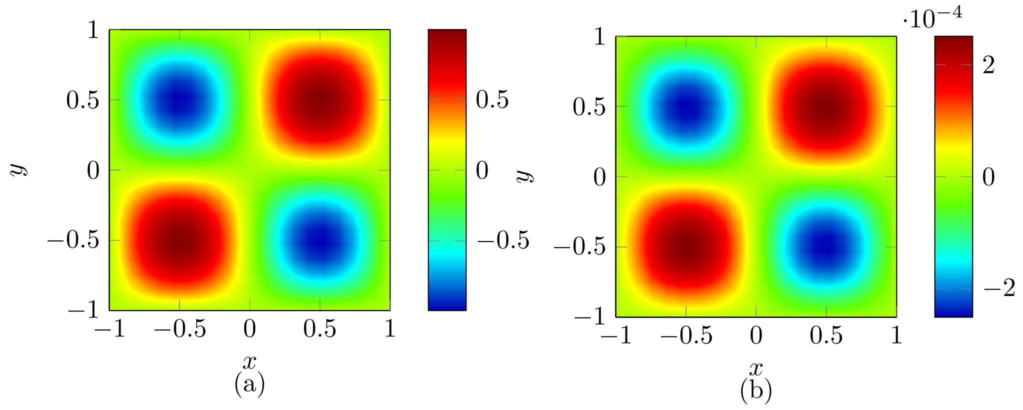


Figure 5.3: Linear plate solution for 21×21 nodal points. (a) Displacement $w(x, y)$ of a linear cartesian plate. (b) Error on the computation with finite differes of order $q = 4$, compared to the analytical solution $w = \sin(\pi x)\sin(\pi y)$.

5.4 Example: Non Linear Plate

A more complex example of a boundary value problem can be a non linear elastic plate submitted to a distributed load simply supported on its four edges. The deflection w of a bending linear plate is ruled by the biharmonic equation plus a non linear term which is dependant of the stress Airy function ϕ. Hence, in addition, the non linear strain compatibility equation must be satisfied. The laplacian of ϕ is the trace of the plane stress tensor. The simply supported boundary conditions are equivalent to say that the displacement, its laplacian and the laplacian of the stress function$\nabla^2\phi$ annul on the edges. Therefore, the problem in a domain $\Omega \subset \mathbb{R}^2 : \{(x, y) \in [-1, 1] \times [-1, 1]\}$ is formulated as:

$$\begin{aligned} &\nabla^4 w = 4\pi^4 \sin(\pi x)\sin(\pi y) + \frac{e}{D}\mathcal{L}(w, \phi), \\ &\nabla^4 \phi + \frac{E}{2}\mathcal{L}(w, w) = 0. \end{aligned}$$

The boundary conditions are represented on the figure below:

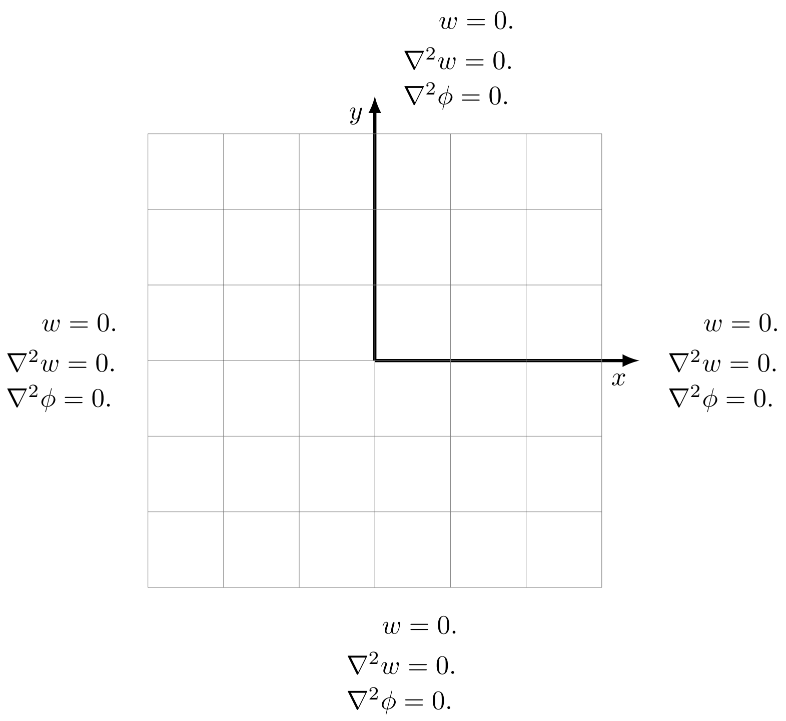


Figure 5.4: Simply supported boundary conditions for a non linear plate.

The implementation of the problem requires the definition of a function for the differential operator:

```fortran
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 ::   v,   vxx,   vyy,   vxy, w, wxx, wyy
  real :: phi, phixx, phiyy, phixy, F, Fxx, Fyy, q

  q = 1d-3 *D*((4*pi**4)) * sin(pi*x) * sin(pi*y)

  v = u(1);   vxx = uxx(1);   vyy = uyy(1) ;   vxy = uxy(1)
  w = u(2);   wxx = uxx(2);   wyy = 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) =  vxx + vyy - w
  L(2) =  wxx + wyy - q/D + (thick/D)*(phiyy*vxx + phixx*vyy - 2*phixy*vxy)
  L(3) =  phixx + phiyy - F
  L(4) =    Fxx + Fyy  + (E/2) * ( vyy*vxx + vxx*vyy - 2* vxy * vxy )

end function
```

Listing 5.8: `API_example_Boundary_Value_Problem.f90`

And also, the definition of the boundary conditions as another function:

```
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) then

            BCs(1) = u(1)
            BCs(2) = u(2)
            BCs(3) = u(3)
            BCs(4) = u(4)

        elseif (x==xf) then

            BCs(1) = u(1)
            BCs(2) = u(2)
            BCs(3) = u(3)
            BCs(4) = u(4)

        elseif (y==y0) then

            BCs(1) = u(1)
            BCs(2) = u(2)
            BCs(3) = u(3)
            BCs(4) = u(4)
        elseif (y==yf) then

            BCs(1) = u(1)
            BCs(2) = u(2)
            BCs(3) = u(3)
            BCs(4) = u(4)
        else
            write(*,*) " Error BCs x=", x
            write(*,*) " x0, xf=", x0, xf
            write(*,*) " y0, yf=", y0, yf

            stop
        endif

    end function
```

Listing 5.9: `API_example_Boundary_Value_Problem.f90`

Hence, they must be contained on a subroutine in which these functions are introduced as input arguments of the partial differential equations solver.

```
subroutine Non_Linear_Plate_2D

    integer, parameter :: Nx = 20, Ny = 20, Nv = 4
    real :: x(0:Nx), y(0:Ny), U(0:Nx, 0:Ny, Nv)
    real :: thick = 16d-3
    real :: x0 = -1 , xf = 1 , y0 = -1 , yf = 1
    integer :: i, j
    integer :: q = 6 ! Order
    real :: E = 72d9 , poisson = 0.22
    real :: mu, lambda, D, pi

    pi = acos(-1d0)




     x(0) = x0 ; x(Nx) =  xf; y(0) = y0; y(Ny) = yf
     call Grid_Initialization( "nonuniform", "x", q, x )
     call Grid_Initialization( "nonuniform", "y", q, y )

    mu= 0.5 * E / (1+poisson)
    lambda = E * poisson / ((1 - 2*poisson)*(1 + poisson))
    D = E * (thick**3) / (12 * (1- poisson**2))

    U=0

    linear2D = .false.  !TO eliminate ; do it automatically

    call Boundary_Value_Problem( x_nodes = x, y_nodes = y,             &
                                 Order = q,                            &
                                 N_variables = Nv,                     &
                                 Differential_operator = NL_Plate,     &
                                 Boundary_conditions   = NL_Plate_BCs, &
                                 Solution = U )

    call scrmod("reverse")
    call qplcon( U, Nx+1, Ny+1, 20)

contains
```

Listing 5.10: `API_example_Boundary_Value_Problem.f90`

The contains command of the last line of the subroutine must be followed by the two previous functions defined for the differential operator and the boundary conditions.

Once the problem is implemented, the solution obtained is represented on figure 5.5:

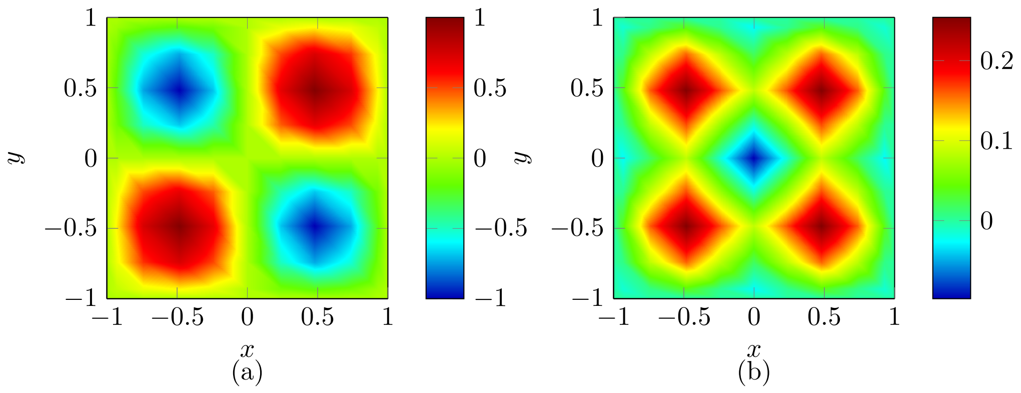


Figure 5.5: Non linear plate solution for 11×11 nodal points and order $q = 6$. (a) Displacement $w(x, y)$ of a non linear cartesian plate. (b) Laplacian of the Airy function $\nabla^2 \phi$.

Chapter 6

Initial Value Boundary Problems

6.1 Overview

In this chapter, the implementation of several initial value boundary problems will be exposed. The order in which the examples are ordered is in crescent complexity, starting by scalar parabollic equations (heat equation and advection-diffusion) in 1-dimensional and 2-dimensional spatial domains. Then, hyperbollic equations will be considered, starting by the waves equation in 1D and 2D spatial domains and finally, two examples of the vibrations of a linear cartesian plate.

Each problem is presented with the same structure. First, the differential equation with its boundary and initial conditions will be stated. After that, pieces of code will be extracted to explain how the problem is implemented. Finally, the results will be exposed in the shape of figures.

Each example that will be presented, can be executed by a call to the subroutine `IVBP_examples`, which at the same time calls each example.

```fortran
subroutine IVBP_examples


  implicit none

   call Heat_equation_1D
   call Advection_Diffusion_1D

   call Heat_equation_2D
   call Advection_Diffusion_2D

   call Waves_equation_1D
   call Waves_equation_2D
   call Plate_vibration

end subroutine
```

Listing 6.1: `API_Example_Initial_Value_Boundary_Problem.f90`

6.2 Example: Heat equation 1D

When a unidimensional continuum medium transfers heat by conduction, the evolution of the distribution of temperatures $u(x,t)$ is ruled by the heat equation. This equation will be solved for a 1D spatial domain determined by the set $\Omega \subset \mathbb{R} : \{x \in [-1,1]\}$ in a temporal domain $t \in [0,4]$:

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

where the thermal diffusivity value is $\nu = \frac{k}{\rho c} = 0.02$. The chosen boundary conditions are homogeneous:

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

and the initial condition is a pulse on the temperature:

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

The implementation of the problem requires the definition of the differential operator as a function:

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

            real :: nu = 0.02

            F =  nu * uxx

end function
```

Listing 6.2: `API_Example_Initial_Value_Boundary_Problem.f90`

And also, the boundary conditions must be defined as a function:

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

        if (x==x0) then
                            BC = u

        else if (x==xf) then
                            BC = u

        else
             write(*,*)  "Error in Heat_BC"
             write(*,*) " x0 =", x0, " xf =", xf
             write(*,*) " x = ", x
             stop
        endif

end function
```

Listing 6.3: `API_Example_Initial_Value_Boundary_Problem.f90`

The previous two functions will be used as an input argument for the subroutine that solves the heat equation.

```
subroutine Heat_equation_1D

       integer, parameter :: Nx = 20, Nt = 1000
       real ::  x(0:Nx)
       real :: Time(0:Nt), U(0:Nt,0:Nx)

       real ::  x0 = -1, xf = 1
       real :: t0 = 0, tf = 4
       integer :: i, j, Order = 6

     Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
     x(0) = x0; x(Nx) = xf
     call Grid_Initialization( "nonuniform", "x", Order, x )


     U(0, :)  =  exp(-25*x**2 )


     call Initial_Value_Boundary_ProblemS(                               &

                       Time_Domain = Time, x_nodes = x,  Order = Order,    &
                       Differential_operator =  Heat_equation1D,           &
                       Boundary_conditions   =  Heat_BC1D,                 &
                       Solution = U )

     call scrmod("reverse")
     call qplot(x, U(0,:),  Nx+1)
     call qplot(x, U(Nt,:), Nx+1)
contains
```

Listing 6.4: `API_Example_Initial_Value_Boundary_Problem.f90`

Once the problem is implemented and the code is compiled and executed the solution at all instants of time for the whole spatial domain is known.

On figure 6.1 it can be seen how the initial pulse (perturbation) on the temperature field is attenuated along time until eventually it reaches a linear distribution, which in this case will be $u(x,t) = 0$.

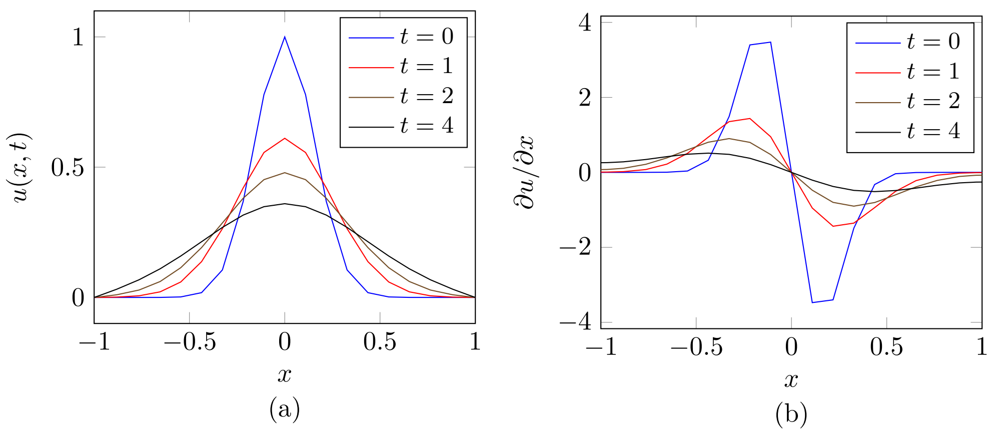


Figure 6.1: Heat equation solution for 21 nodal points and order $q = 6$. (a) Numerical solution for the temperature $u(x,t)$ at four instants of time. (b) Numerical solution for the heat flux $\partial u/\partial x$ at four instants of time.

6.3 Example: Advection Diffusion equation 1D

When a continuum medium transfers heat by conduction and convection, the evolution of the distribution of temperatures $u(x,t)$ is ruled by the advection diffusion equation. This equation for a 1D spatial domain determined by the set $\Omega \subset \mathbb{R} : \{x \in [-1,1]\}$ in a temporal domain $t \in [0,2]$:

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

where the thermal diffusivity value is $\nu = \frac{k}{\rho c} = 0.02$.

The chosen boundary conditions are homogeneous:

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

and the initial condition is a pulse on the temperature:

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

The implementation of the problem requires the definition of the differential operator as a function:

```fortran
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 6.5: `API_Example_Initial_Value_Boundary_Problem.f90`

Also, the boundary conditions must be defined as functions:

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

        if (x==x0) then
                        BC = u

        else if (x==xf) then
                        BC = u
        else
             write(*,*)  "Error in Advection_BC"
             write(*,*) " x = ", x
             stop
        endif

end function
```

Listing 6.6: `API_Example_Initial_Value_Boundary_Problem.f90`

These functions are used as input arguments of the subroutine that computes the solution of the advection-diffusion equation:

```fortran
subroutine Advection_diffusion_1D

       integer, parameter :: Nx = 30, Nt = 1000
       real ::  x(0:Nx)
       real :: Time(0:Nt), U(0:Nt,0:Nx)

       real ::  x0 = -1, xf = 1
       real :: t0 = 0, tf = 0.5
       integer :: i, Order = 4


     Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
     x(0) = x0; x(Nx) = xf
     call Grid_Initialization( "nonuniform", "x", Order, x )



     U(0, :)  =  exp( -25*x**2 )

     call Initial_Value_Boundary_ProblemS(                                  &

                Time_Domain = Time, x_nodes = x, Order = Order,             &
                Differential_operator =  Advection_equation1D,              &
                Boundary_conditions   =  Advection_BC1D,  Solution = U )

     call scrmod("reverse")
     call qplot(x, U(0,:),  Nx+1)
     call qplot(x, U(Nt,:), Nx+1)

contains
```
Listing 6.7: `API_Example_Initial_Value_Boundary_Problem.f90`

Once the problem is implemented and the code compiled and executed, the numerical solution is obtained. On figure 6.2 it is shown the solution evaluated at four instants of time:

In the figures above it can be seen how the attenuation caused by the diffusion is combined with the transport effect along OX produced by the partial derivative along the same direction. When compared to the heat equation solution it can be seen that the steady solution is achieved faster as the convection speeds the heat exchange.

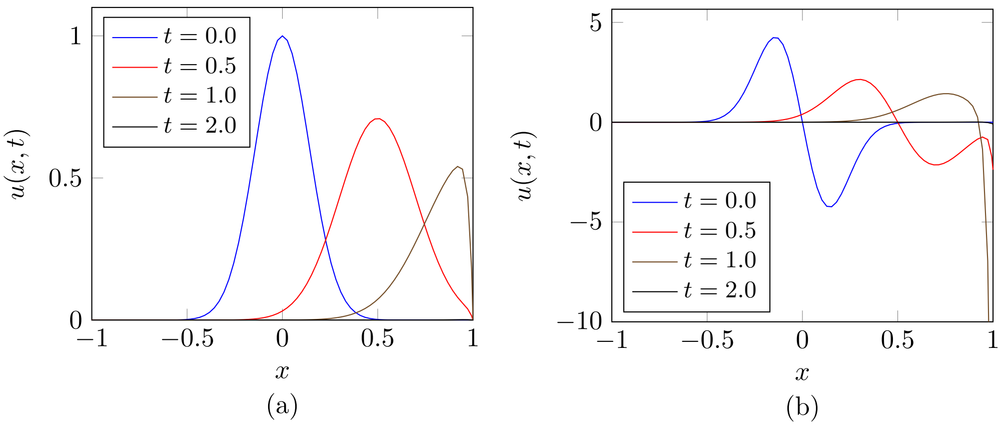


Figure 6.2: Advection-diffusion equation solution for 31 nodal points and order $q = 4$. (a) Numerical solution for the temperature $u(x,t)$ at four instants of time. (b) Numerical solution for the heat flux $q(x,t)$ at four instants of time.

6.4 Example: Heat equation 2D

When a continuum medium transfers heat by conduction, the evolution of the distribution of temperatures $u(x,y,t)$ is ruled by the advection diffusion equation. This equation will be solved for a 1D spatial domain determined by the set $\Omega \subset \mathbb{R}^2$: $\{(x,y) \in [-1,1] \times [-1,1]\}$ in a temporal domain $t \in [0,4]$ formulated as:

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

where the thermal diffusivity value is $\nu = \frac{k}{\rho c} = 0.02$, and is given an initial condition:

$$u(x,y,0) = \exp\bigl(-25x^2 - 25y^2\bigr), \tag{6.1}$$

and the boundary conditions are homogeneous and represented on the figure below:

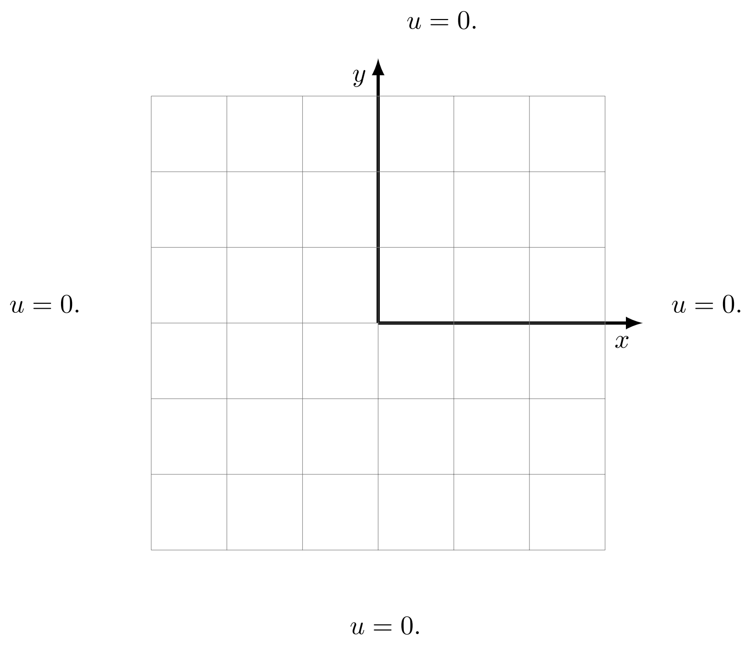


Figure 6.3: Uniform temperature at boundaries of a rectangular domain.

The implementation of the problem requires the definition of the differential operator as a function:

```
function Heat_equation2D( x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy ) result(F)
        real,intent(in) :: x, y, t
        real, intent(in) ::   U, Ux, Uy, Uxx, Uyy, Uxy
        real :: F
             real :: nu = 0.02

             F =   nu * ( Uxx + Uyy )
end function
```

Listing 6.8: `API_Example_Initial_Value_Boundary_Problem.f90`

Also, the boundary conditions must be defined as a function:

```
function Heat_BC2D( x, y, t, U, Ux, Uy ) result (BC)

      real, intent(in) :: x, y, t
      real, intent(in) :: U, Ux, Uy
      real :: BC

        if (x==x0) then
                                 BC = U
        else if (x==xf) then
                                 BC = U
        else if (y==y0) then
                                 BC = U
        else if (y==yf) then
                                 BC = U
        else
            write(*,*)  "Error in Advection_BC "
        end if

end function
```

Listing 6.9: API_Example_Initial_Value_Boundary_Problem.f90

These functions are used as input arguments of the subroutine that computes the solution of the heat equation:

```
subroutine Heat_equation_2D

      integer, parameter :: Nx = 20, Ny = 20, Nt = 200
      real :: x(0:Nx), y(0:Ny), Time(0:Nt), U(0:Nt, 0:Nx, 0:Ny)

       real :: x0 = -1, xf = 1, y0 = -1, yf = 1
       real :: t0 = 0, tf = 4
       integer :: i, j, Order = 4

    Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
    x(0) = x0;  x(Nx) = xf; y(0) = y0;  y(Ny) = yf;

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

    U(0, :, :) = Tensor_product( exp(-25*x**2), exp(-25*y**2) )

    call Initial_Value_Boundary_ProblemS(                                &
                Time_Domain = Time, x_nodes = x, y_nodes = y, Order = Order, &
                Differential_operator =  Heat_equation2D,                &
                Boundary_conditions   =  Heat_BC2D,  Solution = U )

    call scrmod("reverse")
    call qplcon( U(0,:, :),   Nx+1, Ny+1, 20)
    call qplcon( U(Nt, :, :), Nx+1, Ny+1, 20)

contains
```

Listing 6.10: API_Example_Initial_Value_Boundary_Problem.f90

Once the problem is implemented and the code compiled and executed, the numerical solution is obtained. In the figure below it is shown the solution evaluated at four instants of time:

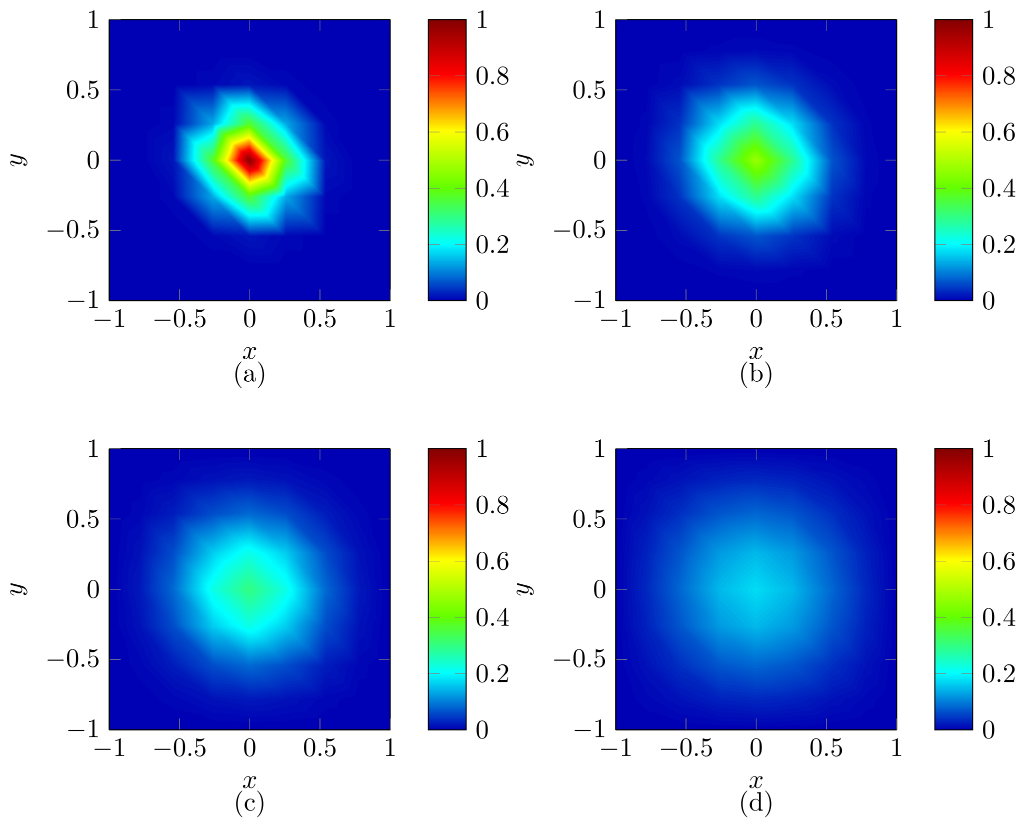


Figure 6.4: Heat equation solution for 21×21 nodal points and order $q = 4$. (a) Initial value $u(x, y, 0)$. (b) Numerical solution $u(x, y, 1)$. (c) Numerical solution $u(x, y, 2)$. (d) Numerical solution $u(x, y, 4)$

In the figure 6.4 it can be observed the attenuation of the perturbation along time due to the diffusion effect.

6.5 Example: Advection-Diffusion equation 2D

When a continuum medium transfers heat by conduction and convection, the evolution of the distribution of temperatures $u(x, y, t)$ is ruled by the advection diffusion equation. This equation for a 1D spatial domain determined by the set $\Omega \subset \mathbb{R}^2 : \{(x, y) \in [-1, 1] \times [-1, 1]\}$ in a temporal domain $t \in [0, 2]$:

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

where the thermal diffusivity value is $\nu = \frac{k}{\rho c} = 0.02$, the initial condition is $u(x, y, 0) = \exp\left(-25x^2 - 25y^2\right)$ and the boundary conditions are homogeneous:

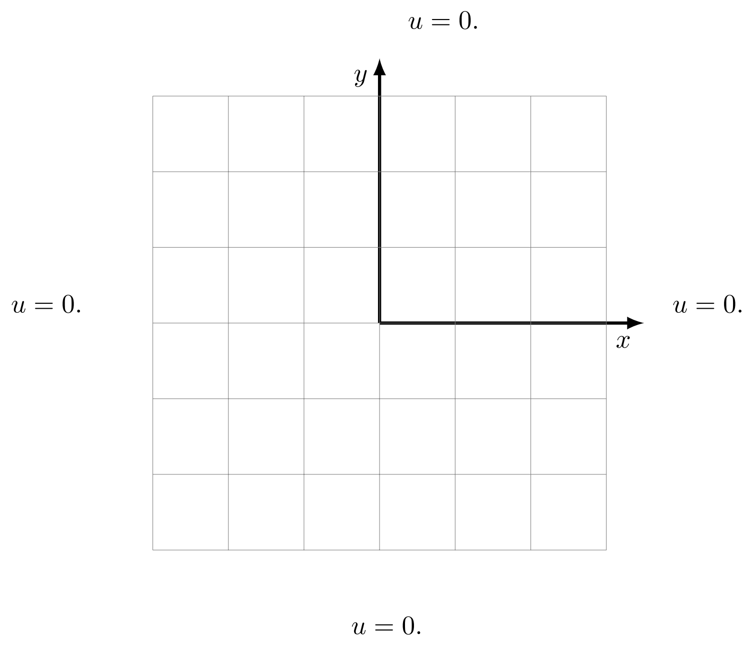


Figure 6.5: Simply supported boundary conditions for a forced vibrating linear plate.

The implementation of the problem requires the definition of the differential operator as a function:

```
function Advection_equation2D( x, y, t, U, Ux, Uy, Uxx, Uyy, Uxy ) result(F)
        real,intent(in) :: x, y, t
        real, intent(in) ::   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_Value_Boundary_Problem.f90`

Also, the boundary conditions must be defined as another function:

```
function Advection_BC2D( x, y, t, U, Ux, Uy ) result (BC)

      real, intent(in) :: x, y, t
      real, intent(in) :: U, Ux, Uy
      real :: BC

        if (x==x0) then
                                 BC = U
        else if (x==xf) then
                                 BC = U
        else if (y==y0) then
                                 BC = U
        else if (y==yf) then
                                 BC = U
        else
            Write(*,*)  "Error in Advection_BC "
        end if

end function
```

Listing 6.12: `API_Example_Initial_Value_Boundary_Problem.f90`

The functions previously defined must be used as input arguments of the subroutine which solves the advection diffusion equation

```fortran
subroutine Advection_diffusion_2D

      integer, parameter :: Nx = 20, Ny = 20, Nt = 200
      real :: x(0:Nx), y(0:Ny), Time(0:Nt), U(0:Nt, 0:Nx, 0:Ny)

       real :: x0 = -1, xf = 1
       real :: y0 = -1, yf = 1
       real :: t0 = 0, tf = 1.0
       integer :: i, j, Order = 8

     Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
     x(0)= x0; x(Nx) = xf; y(0)=y0; y(Ny) = yf
     call Grid_Initialization( "nonuniform", "x", Order, x )
      call Grid_Initialization( "nonuniform", "y", Order, y )

     U(0, :, :) = Tensor_product( exp(-25*x**2), exp(-25*y**2) )

     call Initial_Value_Boundary_ProblemS(                                   &
               Time_Domain = Time, x_nodes = x, y_nodes = y, Order = Order, &
               Differential_operator =  Advection_equation2D,               &
               Boundary_conditions   =  Advection_BC2D,  Solution = U )

     call scrmod("reverse")
     call qplcon( U(0,:, :),   Nx+1, Ny+1, 20)
     call qplcon( U(Nt, :, :), Nx+1, Ny+1, 20)

contains
```

Listing 6.13: `API_Example_Initial_Value_Boundary_Problem.f90`

Once the problem is implemented and the code compiled and executed, the numerical solution is obtained. In the figure below it is shown the solution evaluated at four instants of time:

On figure 6.6 above it can be seen how the attenuation caused by the diffusion is combined with the transport effect along OX produced by the partial derivative along the same direction. When compared to the heat equation solution it can be seen that the steady solution is achieved faster as the convection speeds the heat exchange.

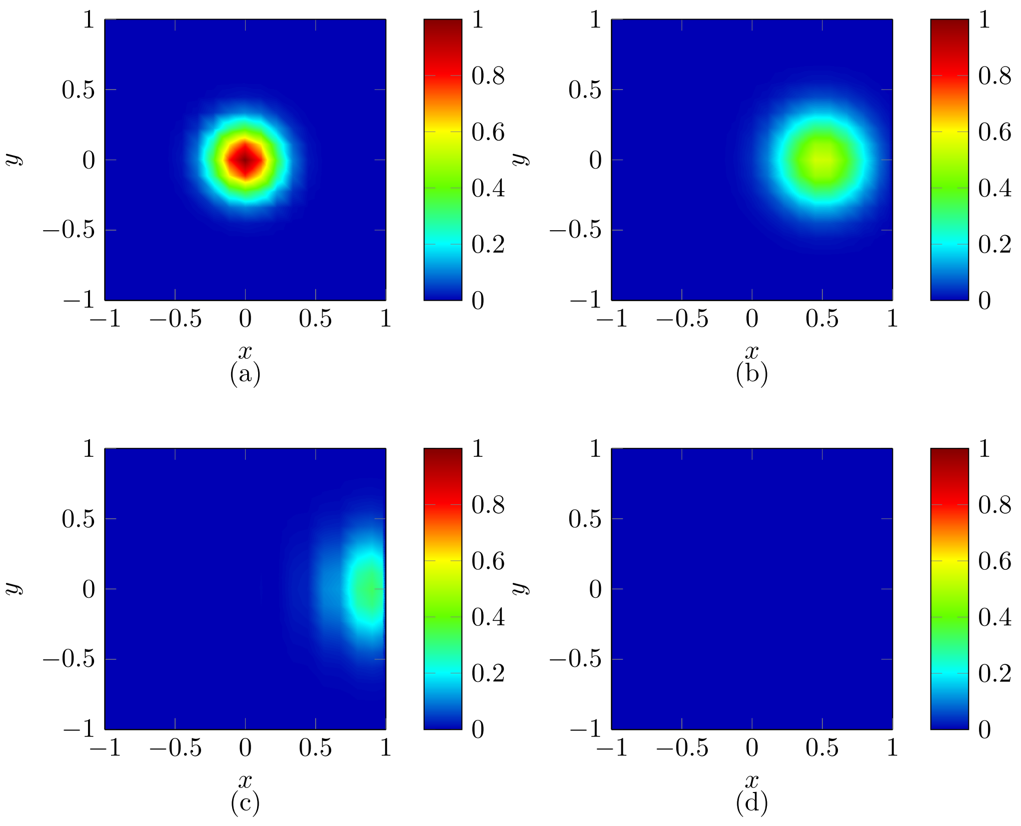


Figure 6.6: Advection-diffusion equation solution for 21×21 nodal points and order $q = 8$. (a) Initial value $u(x, y, 0)$. (b) Numerical solution $u(x, y, 1)$. (c) Numerical solution $u(x, y, 2)$. (d) Numerical solution $u(x, y, 4)$

6.6 Example: Waves equation 1D

To illustrate how to use the code, we shall expose one example for Initial Value Boundary Problem system for 1D and 2D problems.

We will solve the unidimensional waves equation:

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

To solve it we shall write it in its order 1 system form:

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

With the initial values:

$$u(x,0) = \exp\left(-15x^2\right)$$
$$v(x,0) = 0$$

And the boundary conditions:

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

The implementation of the problem requires the definition of the differential operator as a function:

```fortran
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);   w = u(2)
            vxx = uxx(1)

            F(1)  = w
            F(2)  = vxx

end function
```

Listing 6.14: `API_Example_Initial_Value_Boundary_Problem.f90`

Also, the boundary conditions must be defined as a function:

```
function Waves_BC1D(x, t, u, ux) result(BC)
  real, intent(in) :: x, t, u(:), ux(:)
  real :: BC( size(u) )

       real :: v

       v = u(1)

        if (x==x0) then
                         BC(1) = v
                         BC(2) = FREE_BOUNDARY_CONDITION

        else if (x==xf) then
                         BC(1) = v
                         BC(2) =  FREE_BOUNDARY_CONDITION

        else
             write(*,*)  "Error in BC_Burgers"
             write(*,*) " x0 =", x0, " xf =", xf
             write(*,*) " x = ", x
             stop
        endif

end function
```

Listing 6.15: `API_Example_Initial_Value_Boundary_Problem.f90`

These two subroutines are used as input arguments of the subroutine which computes the solution of the waves equation:

```
subroutine Waves_equation_1D

       integer, parameter :: Nx = 40, Nt = 1000, Nv = 2
       real :: Time(0:Nt), U(0:Nt,0:Nx, Nv), x(0:Nx)
       real ::  x0 = -1, xf = 1, t0 = 0, tf = 2.
       integer :: i, Order = 4;

     Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
     x(0) = x0; x(Nx) = xf;
     call Grid_Initialization( "nonuniform", "x", Order, x )

     U(0, :,1) = exp( - 15 * x**2 );   U(0, :,2) = 0

     call Initial_Value_Boundary_ProblemS(                                  &
          Time_Domain = Time, x_nodes = x, Order = Order, N_variables = Nv, &
          Differential_operator =  Wave_equation1D,                         &
          Boundary_conditions   =  Waves_BC1D,  Solution = U )


contains
```

Listing 6.16: `API_Example_Initial_Value_Boundary_Problem.f90`

Once the code is implemented, compiled and executed the solution is obtained. The solution evaluated at four different instants of time is shown on figure 6.7:

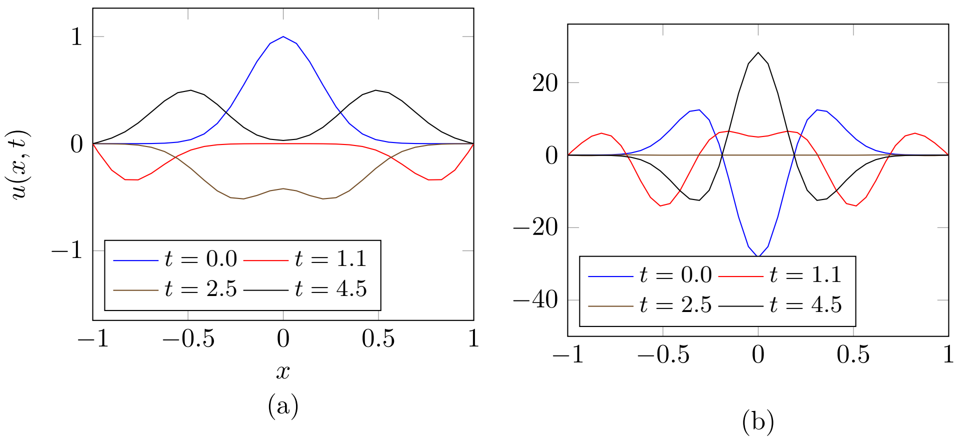


Figure 6.7: Waves equation solution for 41 nodal points and order $q = 4$. (a) Numerical solution $u(x,t)$ at four instants of time. (b) Numerical solution $u_{xx}(x,t)$ at four instants of time.

In the figure it can be seen how the wave propagates through the domain and when it reaches the edges it is reflected.

6.7 Example: Waves equation 2D

We will solve the two dimensional waves equation in the spatial domain $\Omega \equiv \{(x,y) \in [-1,1] \times [-1,1]\}$ for a time interval $t \in [0, t_f]$:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 0$$

To solve it we shall write it in its order 1 system form:

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

With the initial values:

$$u(x,y,0) = \exp\bigl(-15(x^2+y^2)\bigr)$$
$$v(x,y,0) = 0$$

And the boundary conditions, chosen as homogeneous are represented o the figure below:

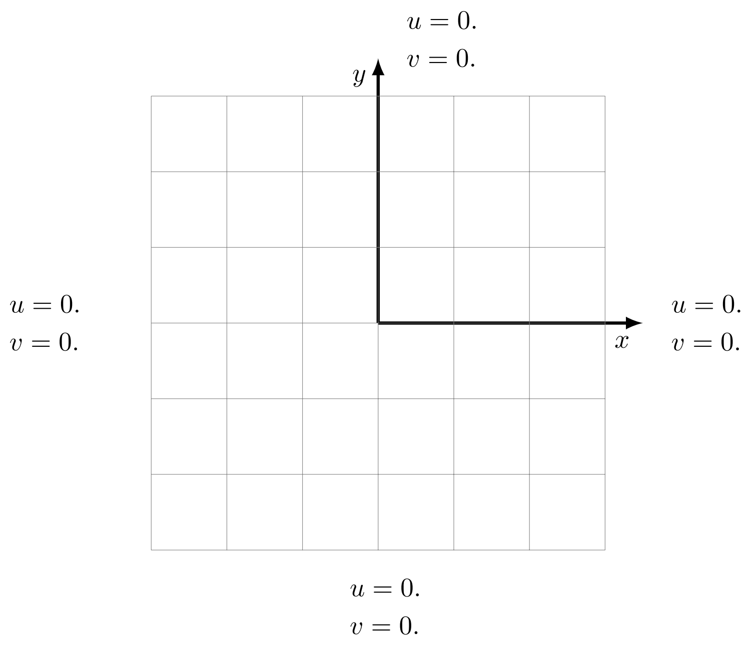


Figure 6.8: Boundary conditions for the 2-dimensional waves equation.

In order to implement the problem it is necessary to define the differential operator as a function in Fortran

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

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

end function
```

Listing 6.17: `API_Example_Initial_Value_Boundary_Problem.f90`

As well, the boundary conditions must be defined in the same manner:

```fortran
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) then
                            BC(1) = v
                            BC(2) = w
        else if (x==xf) then
                            BC(1) = v
                            BC(2) = w
        else if (y==y0) then
                            BC(1) = v
                            BC(2) = w

        else if (y==yf) then
                            BC(1) = v
                            BC(2) = w

        else
             write(*,*)  "Error in BC2D_waves"
             write(*,*) " x0 =", x0, " xf =", xf
             write(*,*) " x = ", x
             stop
        endif

end function
```

Listing 6.18: `API_Example_Initial_Value_Boundary_Problem.f90`

These two functions must be contained within the interface of a subroutine, below a contains command:

```fortran
subroutine Waves_equation_2D

       integer, parameter :: Nx = 20, Ny = 20, Nt = 100, Nv = 2
       real ::  x(0:Nx), y(0:Ny)
       real :: Time(0:Nt), U(0:Nt, 0:Nx, 0:Ny, Nv)

       real ::  x0 = -1, xf = 1, y0 = -1, yf = 1
       real ::  t0 = 0, tf = 2
       integer :: i, j, Order = 8

     Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
     x(0) = x0;  x(Nx) = xf; y(0) = y0;  y(Ny) = yf;

     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_Value_Boundary_ProblemS(                              &

                                Time_Domain = Time,                     &
                                x_nodes = x, y_nodes = y,               &
                                Order = Order, N_variables = Nv,        &
                                Differential_operator = Wave_equation2D,  &
                                Boundary_conditions = Wave_BC2D,        &
                                Solution = U )

     call scrmod('revers')
     call qplclr( U(0, 0:Nx, 0:Ny, 1) ,  Nx+1, Ny+1)
     call qplclr( U(Nt, 0:Nx, 0:Ny, 1) , Nx+1, Ny+1)


contains
```
Listing 6.19: `API_Example_Initial_Value_Boundary_Problem.f90`

Once the problem is implemented and the code compiled and executed, the numerical solution is obtained. In the figure below it is shown the solution evaluated at four instants of time:

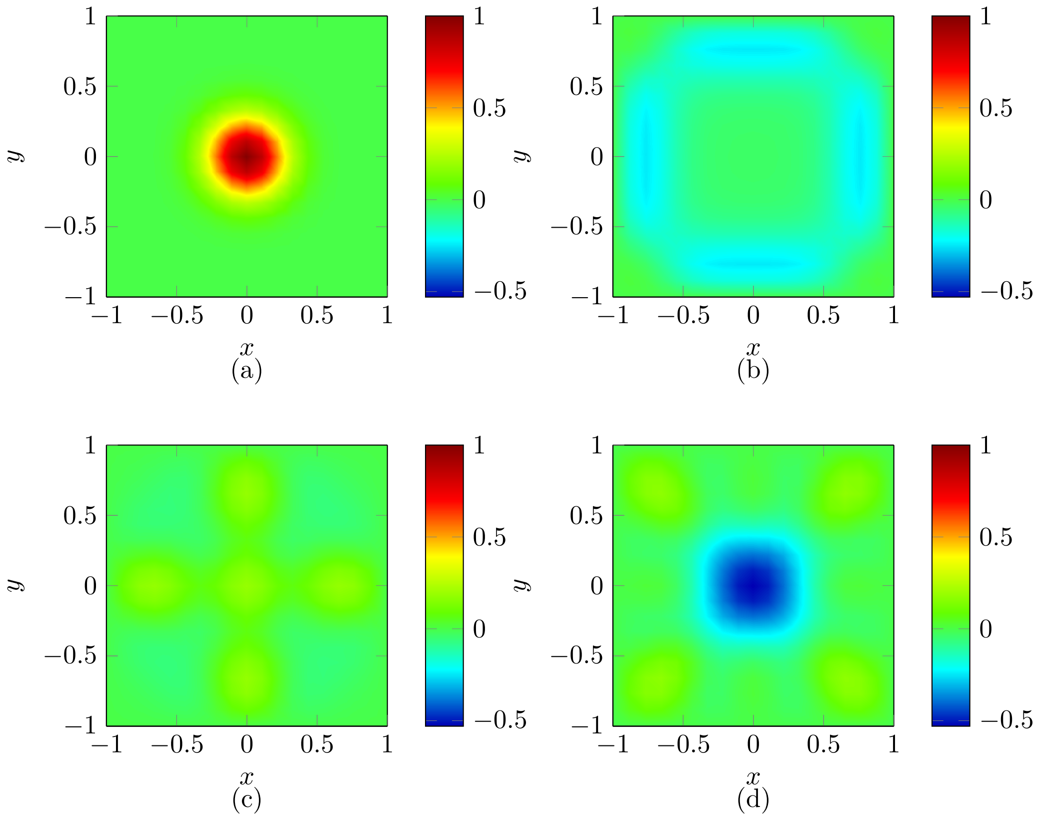


Figure 6.9: Waves equation solution for 21×21 nodal points and order $q = 8$. (a) Initial value $u(x, y, 0)$. (b) Numerical solution at $u(x, y, 0.5)$. (c) Numerical solution at $u(x, y, 1)$. (d)Numerical solution at $u(x, y, 2)$

6.8 Example: Plate Vibration

The vibrations of a linear plate can also be submitted to an initial condition, for example on the field of velocities (pulse). The equation to be solved in the spatial domain $\Omega \equiv \{(x, y) \in [-1, 1] \times [-1, 1]\}$ for a time interval $t \in [0, t_f]$ is:

$$\rho e \frac{\partial^2 w}{\partial t^2} + D\nabla^4 w = 0 \tag{6.2}$$

This equation represents the vibrations of a linear plate submitted to an initial condition, free of loads. The simply supported boundary conditions on its edges can be expressed:

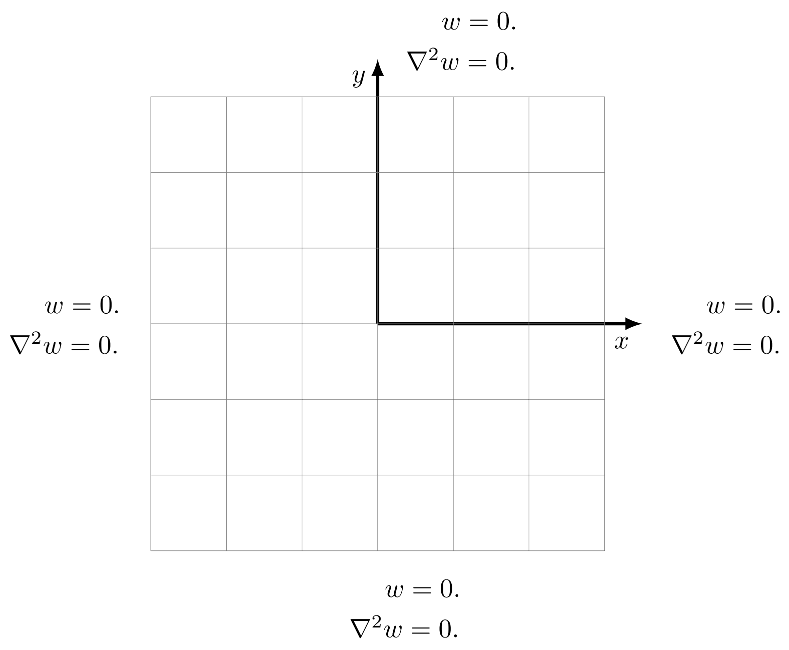


Figure 6.10: Simply supported boundary conditions for a vibrating linear plate.

Along with the boundary conditions, two initial values must be specified:

$$w(x, y, 0) = 0 \tag{6.3}$$

$$\frac{\partial w}{\partial t}(x, y, 0) = e^{-10(x^2+y^2)} \tag{6.4}$$

The implementation of the problem requires the definition of a function for the differential operator:

```
function Plate_equation( 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))

            L(1)  =   u(2)
            L(2)  = -( uxx(3) + uyy(3) ) * D / ( rho_g*thickness )
            L(3)  =  ( uxx(2) + uyy(2) )

end function
```

Listing 6.20: `API_Example_Initial_Value_Boundary_Problem.f90`

And also, the definition of the boundary conditions as another function:

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

        if (x==x0) then
                           BC(1) = u(1)
                           BC(2) = u(2)
                           BC(3) = u(3)

        else if (x==xf) then
                           BC(1) = u(1)
                           BC(2) = u(2)
                           BC(3) = u(3)

        else if (y==y0) then
                           BC(1) = u(1)
                           BC(2) = u(2)
                           BC(3) = u(3)

        else if (y==yf) then
                           BC(1) = u(1)
                           BC(2) = u(2)
                           BC(3) = u(3)
        else
             write(*,*)  "Error in BC1 "
             write(*,*) " x0 =", x0, " xf =", xf
             write(*,*) " x = ", x
             stop
        endif

end function
```

Listing 6.21: `API_Example_Initial_Value_Boundary_Problem.f90`

These two functions must be framed on a common environment that contains them. Hence, they must be contained on a subroutine in which these functions are introduced as input arguments of the partial differential equations solver.

```fortran
subroutine Plate_Vibration

       integer, parameter :: Nx = 10, Ny = 10, Nt = 200, Nv = 3
       real ::  x(0:Nx), y(0:Ny)
       real :: Time(0:Nt), U(0:Nt, 0:Nx, 0:Ny, Nv)
       real ::  x0 = -1 , xf = 1 , y0 = -1 , yf = 1
       real ::  t0 = 0 , tf  , pi
       integer :: i, j,m, Order = 6
       real :: thickness = 16d-3, rho_g = 2500
       real :: E = 72d6 , poisson = 0.22
       real :: mu, lambda, D

       pi = 4 * atan(1d0)
       tf = 1
       Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]

       x(0) = x0;  x(Nx) = xf
       y(0) = y0;  y(Ny) = yf

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

      D = E * thickness**3 / (12 * (1- poisson**2))

      U(0, :, :, 1) =  0; U(0, :, :, 3) =  0

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

      call Initial_Value_Boundary_ProblemS( Time_Domain = Time,             &
                             x_nodes = x, y_nodes = y,                      &
                             Order = Order, N_variables = Nv,               &
                             Differential_operator = Plate_equation,        &
                             Boundary_conditions = Plate_BC,                &
                             Solution = U )

     call scrmod("reverse")
     call qplcon( U(0,:, :, 2),   Nx+1, Ny+1, 20)
     call qplcon( U(Nt, :, :, 2), Nx+1, Ny+1, 20)

contains
```

Listing 6.22: `API_Example_Initial_Value_Boundary_Problem.f90`

The contains command of the last line of the subroutine must be followed by the two previous functions defined for the differential operator and the boundary conditions.

Once the problem has been implemented and the code compiled and executed, the solution obtained at four different instants of time is shown in the following figures:

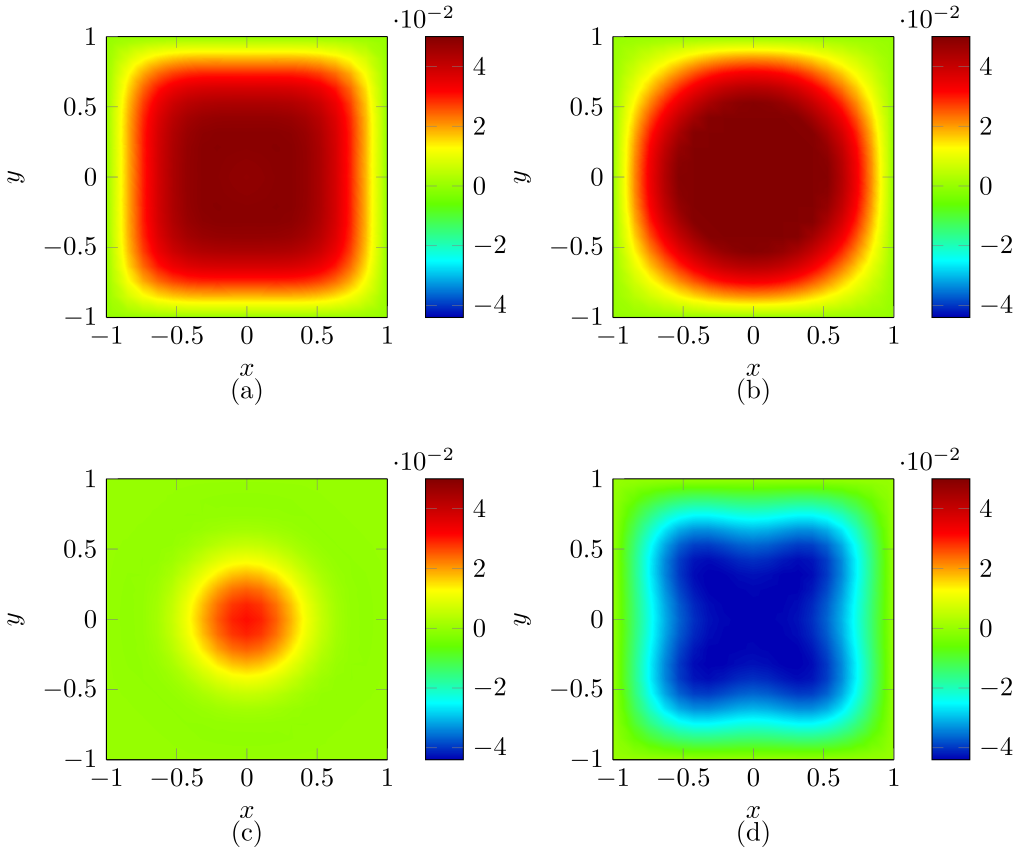


Figure 6.11: Plates vibration solution for 21×21 nodal points and order $q = 6$. (a) Numerical solution at $w(x, y, 0.25)$. (b) Numerical solution at $w(x, y, 0.5)$. (c) Numerical solution at $w(x, y, 0.75)$. (d)Numerical solution at $w(x, y, 1)$.

Chapter 7

Mixed Boundary and Initial Value Problems

7.1 Overview

In this chapter, a mixed boundary and initial value problem will be solved making use of the available libraries. Briefly and non rigourously, these problems can be defined as a parabollic time dependant problem for $\boldsymbol{u}$ and an elliptic problem for $\boldsymbol{v}$. The example considered is a non linear Von Karmann plate. This model appears as a generalization of the linear plate example considered on chapter 6. As before, first a brief mathematical approach to will be exposed, followed by its implementation on Fortran.

7.2 Example: Non Linear Plate Vibration

If an initial condition, for example, on the velocities field is given, the resolution of the problem is quite similar. The equation to be solved in the spatial domain $\Omega \equiv \{(x,y) \in [-1,1] \times [-1,1]\}$ for a time interval $t \in [0,1]$ is:

$$\rho e \frac{\partial^2 w}{\partial t^2} + D\nabla^4 w = (4\pi^4 - 1)\sin(\pi x)\sin(\pi y)\cos(t) + e\mathcal{L}(w,\phi), \tag{7.1}$$

$$\tag{7.2}$$

$$\nabla^4 \phi + \frac{E}{2}\mathcal{L}(w,w) = 0. \tag{7.3}$$

The simply supported boundary conditions are shown on the figure below:

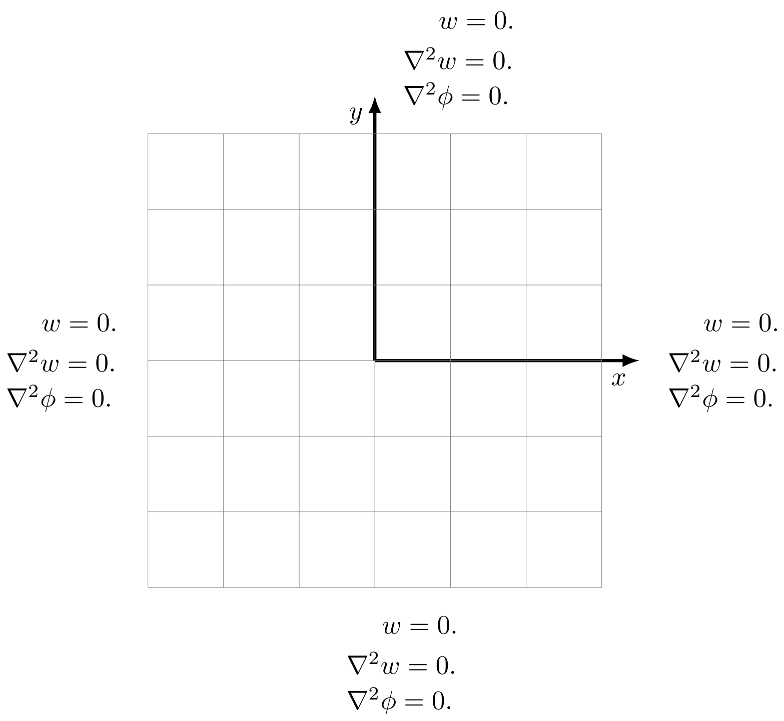


Figure 7.1: Simply supported boundary conditions for a vibrating non linear plate.

Along with the boundary conditions, two initial values must be specified:

$$w(x, y, 0) = 0 \tag{7.4}$$

$$\frac{\partial w}{\partial t}(x, y, 0) = e^{-10(x^2+y^2)} \tag{7.5}$$

The implementation of the problem requires the definition of two functions for the differential operators:

```
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))

  Lu(1)  =   u(2)

  Lu(2)  =    (-(uxx(3) + uyy(3)) * D / (rho_g*thickness) +               &

           +  ( vyy(1)*uxx(1) + vxx(1)*uyy(1) - 2* vxy(1)*uxy(1))/(rho_g) &

            + 1d-3*(4*(D/(rho_g*thickness))*pi**4-4*pi**2)                &
                    * cos(pi*x/2) * cos(pi*y)* cos(2*pi*t))

  Lu(3)  =   uxx(2) + uyy(2)


end function
```

Listing 7.1: API_Example_IVBP_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))


 Lv(1)  = vxx(1) + vyy(1) - v(2)

 Lv(2)  =  vxx(2) + vyy(2)           &

         + 0.5*E*( uyy(1)*uxx(1) + uxx(1)*uyy(1) - 2* uxy(1)*uxy(1) )


end function
```

Listing 7.2: API_Example_IVBP_and_BVP.f90

And also, the two sets of boundary conditions must be defined as functions:

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

        if (x==x0) then
                           BCu(1) = u(1)
                           BCu(2) = u(2)
                           BCu(3) = u(3)

        else if (x==xf) then
                           BCu(1) = u(1)
                           BCu(2) = u(2)
                           BCu(3) = u(3)

        else if (y==y0) then
                           BCu(1) = u(1)
                           BCu(2) = u(2)
                           BCu(3) = u(3)

        else if (y==yf) then
                           BCu(1) = u(1)
                           BCu(2) = u(2)
                           BCu(3) = u(3)
        else
             write(*,*)  "Error in BC1 "
             stop
        endif

end function
```

Listing 7.3: `API_Example_IVBP_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) then
                           BCv(1) = v(1)
                           BCv(2) = v(2)

        else if (x==xf) then
                           BCv(1) = v(1)
                           BCv(2) = v(2)

        else if (y==y0) then
                           BCv(1) = v(1)
                           BCv(2) = v(2)


        else if (y==yf) then
                           BCv(1) = v(1)
                           BCv(2) = v(2)


        else
             write(*,*)  "Error in BC2 "
             stop
        endif

end function
```

Listing 7.4: `API_Example_IVBP_and_BVP.f90`

These two functions must be contained on a subroutine in which these functions are introduced as input arguments of the partial differential equations solver.

```fortran
subroutine Non_Linear_Plate_Vibrations

     integer, parameter :: Nx = 10, Ny = 10, Nt = 200, Nv1 = 3 , Nv2 = 2
     real ::  x(0:Nx), y(0:Ny)
     real :: Time(0:Nt), U(0:Nt, 0:Nx, 0:Ny, Nv1) , V (0:Nt, 0:Nx, 0:Ny, Nv2)
     real ::  x0 = -1, xf = 1, y0 = -1, yf = 1
     real ::  t0 = 0, tf = 1, pi = 4*atan(1d0), thickness = 16d-3, D
     real ::  poisson = 0.22, E = 72d3, rho_g=2.500
     integer :: i, j, m, Order = 4


    Time = [ (t0 + (tf-t0)*i/Nt, i=0, Nt ) ]
    x    = [ (x0 + (xf-x0)*i/Nx, i=0, Nx ) ]
    y    = [ (y0 + (yf-y0)*j/Ny, j=0, Ny ) ]

    D = E * thickness**3 / ( 12*(1-poisson**2)  )


   U(0,:,:,1) =  1d-3 * Tensor_product( sin(pi*x), sin(pi*y) )
   U(0,:,:,2) =  0
   U(0,:,:,3) =  -2*1d-3 * pi**2 * Tensor_product( sin(pi*x), sin(pi*y) )


   call IVBP_and_BVP_Problem( Time_Domain = Time,                  &
                              x_nodes = x, y_nodes = y,            &
                              Order = Order, N_u = Nv1, N_v = Nv2, &
                              Differential_operator_u = Lu,        &
                              Differential_operator_v = Lv,        &
                              Boundary_conditions_u   = BCu,       &
                              Boundary_conditions_v   = BCv,       &
                              Ut = U,  Vt = V )

   call scrmod("reverse")
   call qplcon( U(0,:, :, 1),   Nx+1, Ny+1, 20)
   call qplcon( U(Nt, :, :, 1), Nx+1, Ny+1, 20)

contains
```

Listing 7.5: `API_Example_IVBP_and_BVP.f90`

the contains command of the last line of the subroutine must be followed by the two previous functions defined for the differential operator and the boundary conditions.

Once the problem is implemented, and the code compiled and executed the numerical solution will be represented. In first place the displacement is represented on figure 7.2.

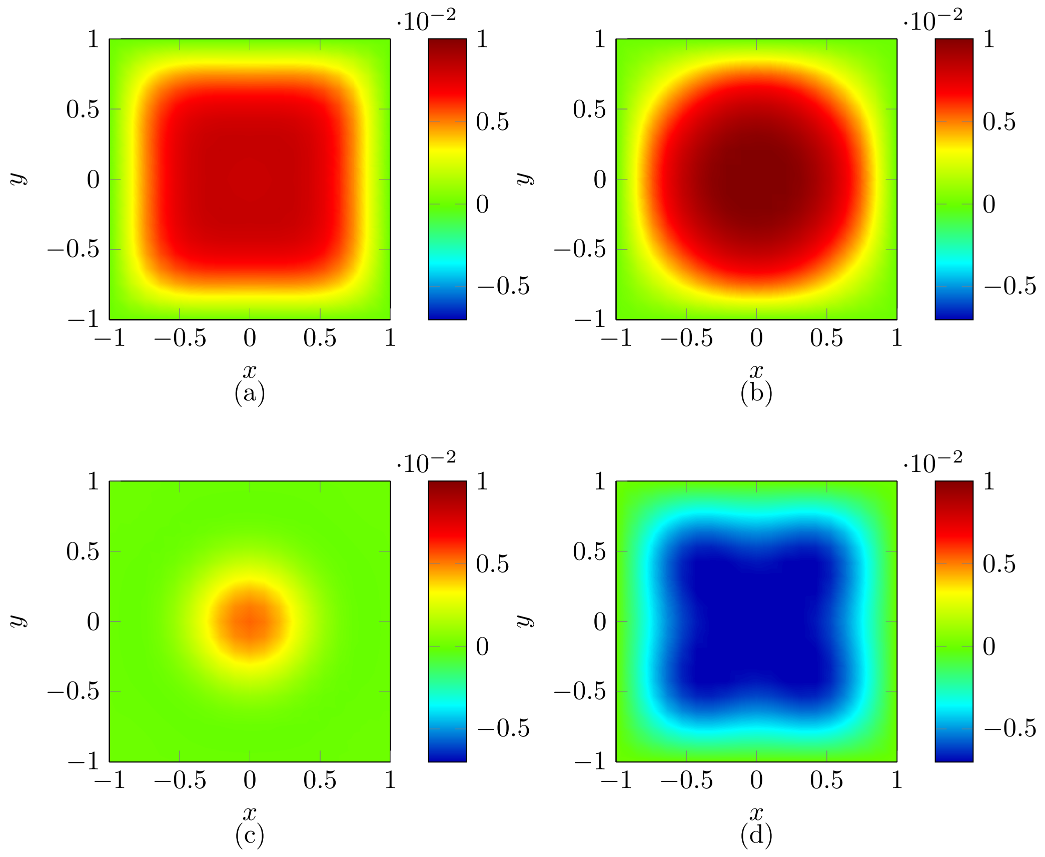


Figure 7.2: Non linear plate vibration solution for 11×11 nodal points and order $q = 8$. (a) Numerical solution at $w(x, y, 0.25)$. (b) Numerical solution at $w(x, y, 0.5)$. (c) Numerical solution at $w(x, y, 0.75)$. (d)Numerical solution at $w(x, y, 1)$.

And finally, the trace of the stress tensor $\nabla^2\phi$ is represented, evaluated at the same instants of time on figure 7.3.

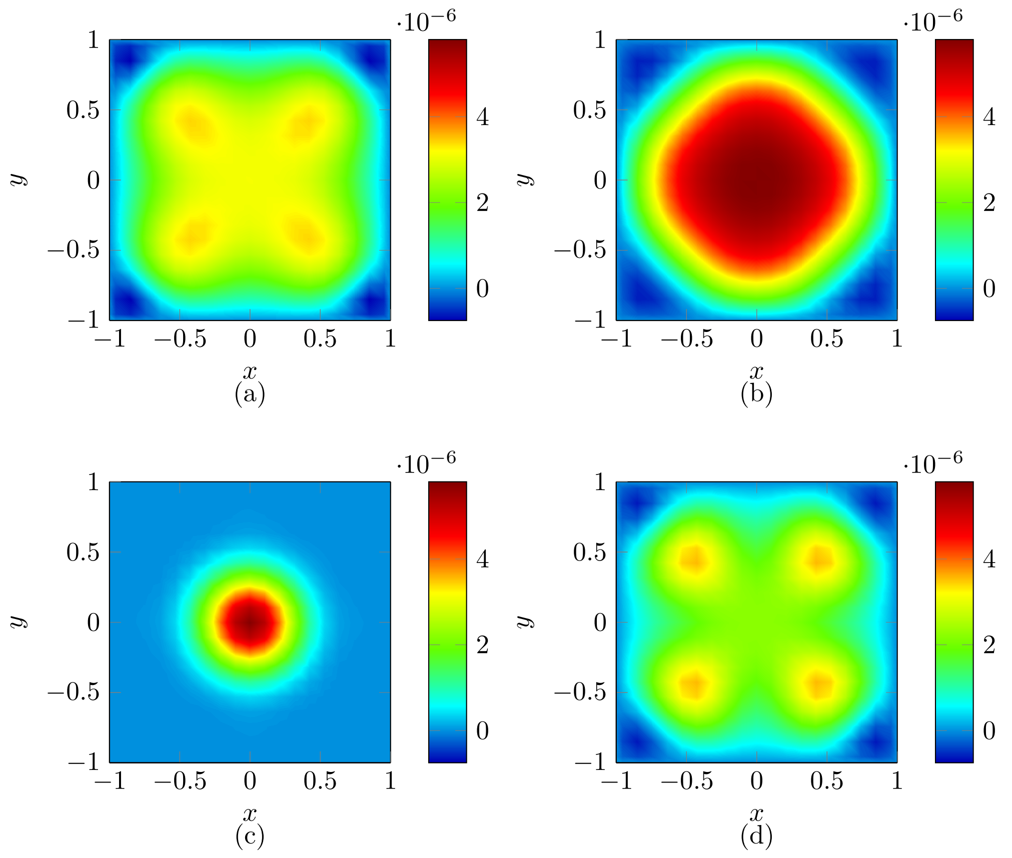


Figure 7.3: (a) Numerical solution at $\nabla^2\phi(x, y, 0.25)$. (b) Numerical solution at $\nabla^2\phi(x, y, 0.5)$. (c) Numerical solution at $\nabla^2\phi(x, y, 0.75)$. (d)Numerical solution at $\nabla^2\phi(x, y, 1)$.

Part II

Developer guidelines

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 $\partial/\partial x_i \ \boldsymbol{e}_i$.
$\mathbb{C}^N$	: N-dimensional complex numbers field.
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 $\boldsymbol{G} : \mathbb{R}^p \times \underbrace{\mathbb{R}^p \times \ldots \times \mathbb{R}^p}_{s\ steps} \times \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}^p$.
$\tilde{\boldsymbol{G}}$	: Restricted temporal scheme, $\boldsymbol{G}\Big\rvert_{(\boldsymbol{u}^n, \ldots \boldsymbol{u}^{n+1-s}; t_n, \Delta t)}$, $\tilde{\boldsymbol{G}} : \mathbb{R}^p \longrightarrow \mathbb{R}^p$.
$\tilde{\boldsymbol{G}}^{-1}$	: 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 the solution 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.

Chapter 1

Linear algebra

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

Within the following pages, the treatment of linear systems will be exposed. It seems natural to follow the order in which the steps to solve linear systems is implemented. Firstly, the factorization to simplify the linear system matrix A will be presented, followed by the resolution of this factorized matrix.

1.2.1 LU Factorization

In order to solve linear problems for an unknown $\boldsymbol{x} \in \mathbb{R}^N$, such as:

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

where $A \in \mathcal{M}_{N\times N}$ verifies $\det(A) \neq 0$ and $\boldsymbol{b} \in \mathbb{R}^N$, it is useful to factorize the matrix A in the form:

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

in which L and U are lower and upper triangular matrices respectively. The three matrices can be expressed, using Einstein's notation as:

$$A = a_{ij}\boldsymbol{e}_i \otimes \boldsymbol{e}_j \quad \text{for} \quad (i,j) \in [1,N] \times [1,N] \tag{1.3}$$
$$L = l_{ij}\boldsymbol{e}_i \otimes \boldsymbol{e}_j \quad \text{for} \quad (i,j) \in [1,N] \times [1,i] \tag{1.4}$$
$$U = u_{ij}\boldsymbol{e}_i \otimes \boldsymbol{e}_j \quad \text{for} \quad (i,j) \in [1,j] \times [1,N]. \tag{1.5}$$

To satisfy the factorization the matrices shall satisfy:

$$a_{ij} = l_{im}u_{mj}, \quad \text{for} \quad m \in [1, \min\{i,j\}]. \tag{1.6}$$

By definition the number of non null terms in L and U are $N(N+1)/2$, which leads to a number of unknown variables of $N_u = N^2 + N$. However, the number of equations supplied by (1.6) is N^2. This makes necessary to fix the value of N unknown variables. The most common variables to fix is either the diagonal of L or U as the unity. The former method is called the Doolittle method is the one which will be presented, that is, we force each element of the diagonal $l_{kk} = 1$. Once this is done, from equation 1.6 it can be obtained a recursion to obtain the $k-th$ element of the diagonal of U if all the previous $k-1$ first rows and columns of L and U are known:

$$u_{kk} = a_{kk} - l_{km}u_{mk}, \quad \text{for} \quad m \in [1, k-1], \tag{1.7}$$

from which u_{kk} is obtained.

A similar recursion can be obtained for u_{kj}, with $k < j$:

$$u_{kj} = a_{kj} - l_{km}u_{mj}, \quad \text{for} \quad m \in [1, k-1]. \tag{1.8}$$

Finally, a recursion for l_{ik}, with $k < i$ results if all the previous $i-1$ rows and k columns of L and U are known

$$l_{ik} = \frac{a_{ik} - l_{km}u_{mj}}{u_{kk}}, \quad \text{for} \quad m \in [1, k-1]. \tag{1.9}$$

These three recursions can be used to calculate recursively the elements of both L and U. This is checked when it is taken $k = 1$, in which is obtained:

$$u_{11} = a_{11}, \tag{1.10}$$
$$u_{1j} = a_{1j}, \quad \text{for} \quad j \in [1, N], \tag{1.11}$$
$$l_{i1} = \frac{a_{i1}}{u_{11}}, \quad \text{for} \quad i \in [1, N]. \tag{1.12}$$

Hence, from A and the initial value u_{11}, the first row of U and the first column of L are calculated. Thus, doing $k = 2$, recurrence (1.7) permits the calculation of u_{22}, (1.8) gives u_{2j} and (1.9) calculates l_{i2}.

Therefore, an algorithm to obtain both L and U is equivalent to evaluate the recurrences for all $k \in [1, N]$ sequentally in the order:

1. $u_{kk} = a_{kk} - l_{km}u_{mk}, \quad \text{for} \quad m \in [1, k-1].$
2. $u_{kj} = a_{kj} - l_{km}u_{mj}, \quad \text{for} \quad m \in [1, k-1].$
3. $l_{ik} = \frac{a_{ik} - l_{km}u_{mj}}{u_{kk}}, \quad \text{for} \quad m \in [1, k-1].$

1.2.2 Solving LU system

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

$$L\boldsymbol{y} = \boldsymbol{b} \;\Rightarrow\; l_{ij}y_j = b_i, \quad \text{for} \quad (i, j) \in [1, N] \times [1, N]. \tag{1.13}$$

As $l_{ij} = 0$ for $i < j$, the first row of (1.13) gives $y_1 = b_1$ and the value of each y_i can be written on terms of the previous y_j, that is:

$$y_i = b_i - l_{ij}y_j, \quad \text{for} \quad 1 < j < i. \tag{1.14}$$

Hence, sweeping through $i = 2, \dots N$, over (1.14) $\boldsymbol{y}$ is obtained.

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

$$\boldsymbol{y} = U\boldsymbol{x}, \Rightarrow\; y_i = u_{ij}x_j, \quad \text{for} \quad (i, j) \in [1, N] \times [1, N]. \tag{1.15}$$

In a similar manner than before, as $u_{ij} = 0$ for $i > j$, the last row of (1.15) 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 (1.16)[1]

$$x_i = \frac{y_i - u_{ij}x_j}{u_{ii}}, \quad \text{for} \quad i < j \leq N. \tag{1.16}$$

Therefore, evaluating recursively $i = N-1, \ldots 1$ (1.16), the solution $\boldsymbol{x}$ is obtained.

1.2.3 Algorithm implementation of LU method

In the code below the algorithm previously explained to obtain the LU factorization of a matrix A is implemented in Fortran.

```fortran
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 1.1: `Linear_systems.f90`

1.2.4 Algorithm implementation of Solving LU

The algorithm presented to solve LU factorised matrices is implemented on the lines below.

[1]In this equation the term in the denominator u_{ii} makes reference to the i-th term on the diagonal of U not the trace of U.

```fortran
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 1.2: `Linear_systems.f90`

1.3 Non linear systems

Whenever, the system to be solved is not linear, it is necessary the use of an iterative method. The most famous of them all is the Newton method.

Let us be $\boldsymbol{f} : \mathbb{R}^p \to \mathbb{R}^p$ a function and $\boldsymbol{x} \in \mathbb{R}^p$ the independent variable. In the case in which we want to calculate the roots of:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}. \tag{1.17}$$

Starting from $\boldsymbol{x}_i$he following recursion is given by the Newton method:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - J_f^{-1}(\boldsymbol{x}_i) \cdot \boldsymbol{f}(\boldsymbol{x}_i), \tag{1.18}$$

where $J_f(\boldsymbol{x}_i) = \nabla \boldsymbol{f}(\boldsymbol{x}_i)$ is the Jacobian matrix and ∇ is the gradient operator. The recursion can be stopped whenever the function is lesser than a sufficient small value ε for the precision needed. That is, the recursion will stop when $\|\boldsymbol{f}(\boldsymbol{x}_{i+1})\| < \varepsilon$.

1.3.1 Algorithm implementation

To implement the Newton method, first, given the function it is calculated its jacobian.

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



end function
```

Listing 1.3: `Jacobian_module.f90`

The Jacobian is used to compute the next value that approximates the solution, $\boldsymbol{x}_{i+1}$.

```fortran
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


   integer :: N

   N = size(x0)

   Dx = 2 * x0
   iteration = 0
   eps = 1

   do while ( eps > 1d-8 .and. iteration < 1000 )

      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>900) then
      write(*,*) " morm2(J) =", maxval(J),  minval(J)
      write(*,*) " Norm2(Dx) = ", eps, iteration
   endif


end subroutine
```

Listing 1.4: `Non_linear_systems.f90`

1.4 Eigenvalues and eigenvectors

In this section an introduction to eigenvalues and eigenvectors computation will be presented. This problem is extremely difficult to deal with in the general case from the numerical point of view. Iterative methods such as the power method lack of efficacy in delicate situations such as eigenvalues of unitary module or non normal matrices. On figure 1.1 it is given a classification from the spectral point of view of the possibilities for a given matrix. The main characteristic which will classify matrix is whether it is normal or not. A normal matrix verifies $AA^T = A^T A$ (symmetry of AA^T) and is diagonalised by orthonormal vectors.

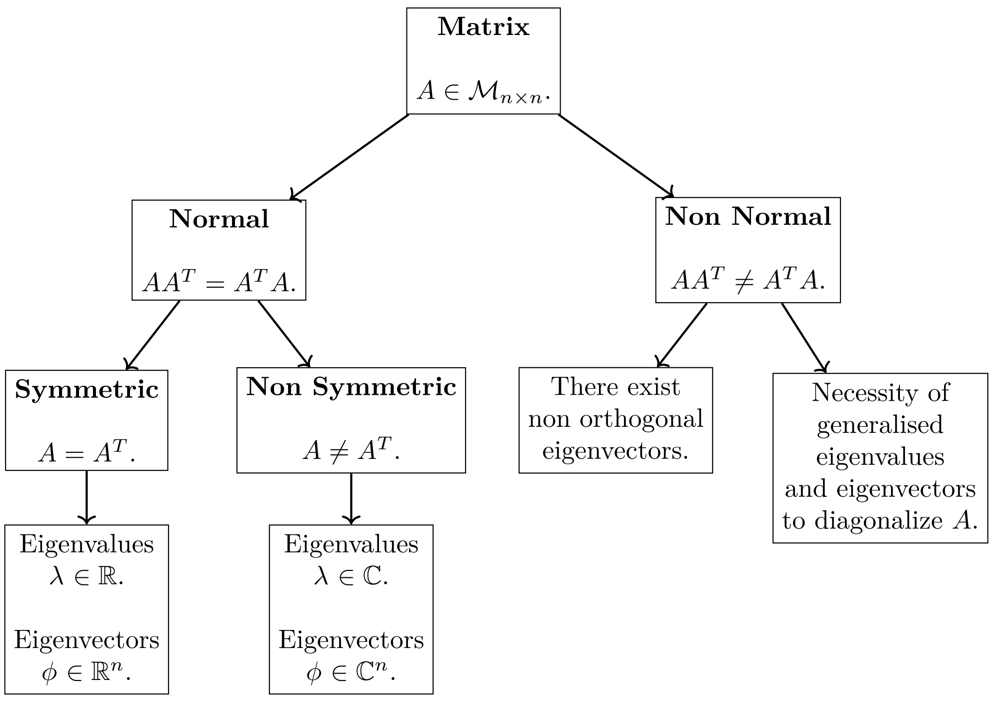

Matrix
$A \in \mathcal{M}_{n\times n}$.
Normal
$AA^T = A^T A$.
Non Normal
$AA^T \neq A^T A$.
Symmetric
$A = A^T$.
Non Symmetric
$A \neq A^T$.
There exist non orthogonal eigenvectors.
Necessity of generalised eigenvalues and eigenvectors to diagonalize A.
Eigenvalues $\lambda \in \mathbb{R}$.
Eigenvectors $\phi \in \mathbb{R}^n$.
Eigenvalues $\lambda \in \mathbb{C}$.
Eigenvectors $\phi \in \mathbb{C}^n$.

Figure 1.1: Normal and non normal matrices classification.

1.4.1 Algorithm implementation

In order to obtain real eigenvalues and eigenvectors of a matrix, the power method is implemented over a subroutine called `Power_method` as follows:

```fortran
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 = norm2(matmul(A, U))


end subroutine
```

Listing 1.5: `Linear_systems.f90`

This subroutine, given a normal matrix A gives back its maximum module eigenvalue λ and its associated eigenvector $\boldsymbol{v}_n$ if $|\lambda_n| \neq 1$. The eigenvalue is yielded on the real `lambda` and the eigenvector in the vector `U`.

In a similar manner the inverse power method is implemented defining the subroutine `Inverse_power_method`:

```fortran
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 1.6: `Linear_systems.f90`

This subroutine, given a normal matrix A gives back its minimum module eigenvalue λ and its associated eigenvector $\boldsymbol{v}_n$ if $|\lambda_n| \neq 1$. The eigenvalue is yielded on the real `lambda` and the eigenvector in the vector `U`.

The two subroutines defined previously are used recurrently to obtain all the eigenvalues of a matrix. For this task the subroutine `Eigenvalues` is defined.

```
subroutine Eigenvalues(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 1.7: `Linear_systems.f90`

The method implemented on it is based on the fact that if λ_1 is the maximum module eigenvalue of A_1, then λ_1 is not eigenvalue of $A_2 = A_1 - \lambda_1 I$, that is $\lambda_1 \notin \Lambda(A_2)$, where $\Lambda(A_2)$ is the spectra of A_2. Note that as it is based on the power method, the scope of this method is for normal matrices which have non unitary eigenvalues. That is $|\lambda| \neq 1$, for all $\lambda 1 \in \Lambda(A)$.

Another functionality that can be implemented using `Eigenvalues` is the SVD decomposition. A subroutine called `SVD` is implemented. Given a matrix A the eigenvalues and eigenvectors of $B = A^T A$ are calculated and the matrix V is composed of the column eigenvectors of B.

```fortran
subroutine SVD(A, sigma, U, V)
      real, intent(inout) :: 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( 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 1.8: `Linear_systems.f90`

A direct application of the power and inverse power method is to compute the condition number $\kappa(A)$ of a normal matrix A with quadratic induced norm. For this, the maximum and minimum (module) eigenvalues of $B = A^T A$, $\sigma_{\max}$ and $\sigma_{\min}$ respectively, are computed. Hence, the condition number is calculated as the division of them. For this, a subroutine called `Condition_number` is defined:

```
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 1.9: `Linear_systems.f90`

Chapter 2

Lagrange Interpolation

2.1 Overview

One of the most relevant matters of the 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.

2.2 Lagrange interpolation

In this chapter, the polynomic interpolation will be presented. Particularly, the interpolation using Lagrange polynomials will be considered, as they are fundamental on the theory of interpolation.

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, the value for $k = 1$ satisfies:

$$\ell_{j0}(x) = 1. \tag{2.5}$$

Hence, it can be obtained recursively the interpolant at x_j for each grade as:

$$\begin{aligned}
\ell_{j1}(x) &= \left(\frac{x - x_0}{x_j - x_0}\right), \\
\ell_{j2}(x) &= \left(\frac{x - x_0}{x_j - x_0}\right)\left(\frac{x - x_1}{x_j - x_1}\right), \\
&\vdots \\
\ell_{jk}(x) &= \underbrace{\left(\frac{x - x_0}{x_j - x_0}\right)\ldots\left(\frac{x - x_{j-1}}{x_j - x_{j-1}}\right)\left(\frac{x - x_{j+1}}{x_j - x_{j+1}}\right)\ldots\left(\frac{x - x_{k-2}}{x_j - x_{k-2}}\right)}_{\ell_{jk-1}(x)}\left(\frac{x - x_{k-1}}{x_j - x_{k-1}}\right).
\end{aligned}$$

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:

$$I(x) = \sum_{j=0}^{N} b_j \ell_j(x). \tag{2.6}$$

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:

$$I(x_i) = f(x_i). \tag{2.7}$$

Taking in account the property (2.2) leads to:

$$I(x_i) = f(x_i) = \sum_{j=0}^{N} b_j \ell_j(x_i) = \sum_{j=0}^{N} b_j \delta_{ij}, \quad \Rightarrow \quad f(x_j) = b_j. \tag{2.8}$$

Hence, the interpolant for $f(x)$ on the nodal points x_j for $j = 0, 1, 2 \ldots, N$ is written:

$$I(x) = \sum_{j=0}^{N} f(x_j) \ell_j(x). \tag{2.9}$$

Notice that the interpolant can be interpreted at each point x as the scalar product:

$$I(x) = \begin{bmatrix} f(x_0), & f(x_1) & \ldots, f(x_N) \end{bmatrix} \cdot \begin{bmatrix} \ell_0(x) & \ell_1(x) & \ldots \ell_N(x) \end{bmatrix}^T \tag{2.10}$$

.

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:

$$I(x,y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} b_{ij} \ell_i(x) \ell_j(y). \tag{2.11}$$

Again, using the property of Lagrange polinomyals (2.2), the coefficients b_{ij} are determined as:

$$b_{ij} = f(x_i, y_j), \tag{2.12}$$

leading to the final expression for the interpolant:

$$I(x,y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} f(x_i, y_j) \ell_i(x) \ell_j(y). \tag{2.13}$$

Notice that when the interpolant is evaluated at a particular coordinate line $x = x_m$ or alternatively at $y = y_n$, it is obtained:

$$I(x_m, y) = \sum_{j=0}^{N_y} f(x_m, y_j) \ell_j(y), \qquad I(x, y_n) = \sum_{i=0}^{N_x} f(x_i, y_n) \ell_i(x), \tag{2.14}$$

which permits writing the interpolant as

$$\begin{aligned} I(x,y) &= \sum_{i=0}^{N_x} I(x_i, y) \ell_i(x) \\ &= \sum_{j=0}^{N_y} I(x, y_j) \ell_j(y). \end{aligned} \tag{2.15}$$

The form in which the interpolant is written in (2.15) suggests a procedure to obtain the interpolant recursively.

Another manner to interpret the equation (2.14) is as a bilinear 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 (2.14) can be written:

$$I(x,y) = \boldsymbol{\ell}_x \, \mathcal{F} \, \boldsymbol{\ell}_y^T. \tag{2.16}$$

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 $y = s$:

$$f(x,y)\Big|_{y=s} = \tilde{f}(x;s) \simeq \tilde{I}(x;s) = \sum_{i=0}^{N_x} b_i(s)\ell_i(x), \tag{2.17}$$

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:

$$b_i(s) = \sum_{j=0}^{N_y} b_{ij}\ell_j(s). \tag{2.18}$$

Hence, the restricted interpolant can be written:

$$\tilde{I}(x;s) = I(x,y)\Big|_{y=s} = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(s), \tag{2.19}$$

and therefore the interpolant $I(x,y)$ can be expressed:

$$I(x,y) = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(y). \tag{2.20}$$

In the same manner, the interpolated value $I(x,y)$ can be achieved restricting the value in $x = s$:

$$f(x,y)\Big|_{x=s} = \tilde{f}(y;s) \simeq \tilde{I}(y;s) = \sum_{j=0}^{N_y} b_j(s)\ell_j(y), \tag{2.21}$$

in which the coefficients $b_j(s)$ can be interpolated aswell:

$$b_j(s) = \sum_{i=0}^{N_x} b_{ij}\ell_i(s). \tag{2.22}$$

This time, the restricted interpolant is expressed as:

$$\tilde{I}(y;s) = I(x,y)\Big|_{x=s} = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(s)\ell_j(y), \tag{2.23}$$

which leads to the interpolated value:

$$I(x,y) = \sum_{i=0}^{N_x}\sum_{j=0}^{N_y} b_{ij}\ell_i(x)\ell_j(y). \tag{2.24}$$

The interpolation procedure and its geometric interpretation can be observed on the figure 2.1. In blue it is 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.

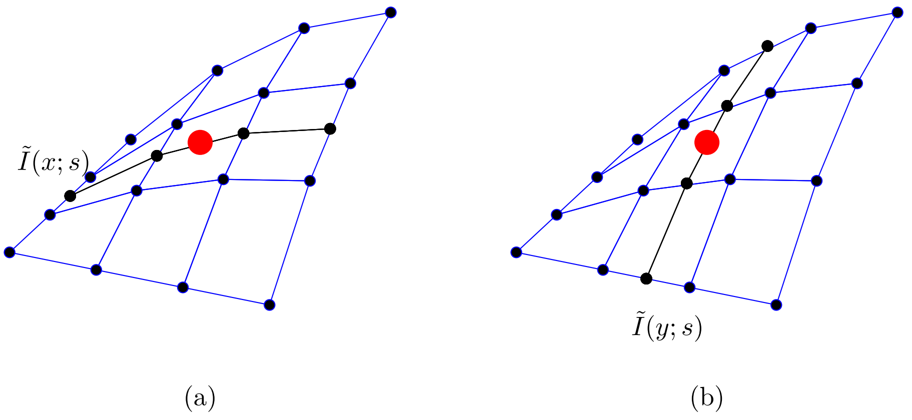


Figure 2.1: Geometric interpretation of the interpolation of a 2D function. (a) Geometric interpretation when restricted along y. (b) Geometric interpretation when restricted along x

2.2.4 Algorithm implementation

The libraries defined for interpolation are all based on the use of a function called `Weights` which gives the coefficients of the lagrange interpolant, its integral and non trivial derivatives.

The function `Lagrange_polynomials` calculates the Lagrange polynomials and its derivatives at a point, given a spatial grid.

```
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 2.1: `Lagrange_interpolation.f90`

The function `Interpolated_value` calculates the interpolated value of a certain degree, given a spatial grid and the function evaluated at that region and the order of the interpolation.

```
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 ! If it is the last, 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 2.2: `Interpolation.f90`

The function `Integral` integrates a function over a given spatial grid, the function at the interval and the approximation order.

```
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 2.3: `Interpolation.f90`

The function `Interpolant` gives the Lagrange interpolant, its integral and the derivatives given a grid, the values of the function over it and the order of the interpolation.

```fortran
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 ! If it is the last, 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 2.4: `Interpolation.f90`

Chapter 3

Cauchy Problem

3.1 Overview

In this chapter a mathematical description of the Cauchy Problem will be presented. This shall be done from the numerical resolution point of view, in where the existence and uniqueness of the solution is supposed. An algorithm for the resolution of this kind of problems will be presented and also, the implementation of this algorithm into Fortran language. With this it is expected for the reader to gain a better insight on how these problems are solved.

3.2 Cauchy Problem

In this section a mathematical presentation of Cauchy problems and also the algorithm to solve them numerically will be presented. A Cauchy problem is composed by a first order ordinary differential equation for $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^p$ with independent variable $t \in [t_0, t_f]$ is defined as:

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{F}(\boldsymbol{u}, t), \qquad \boldsymbol{u}(t_0) = \boldsymbol{u}_0, \tag{3.1}$$

where $\boldsymbol{F} : \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^p$ is a function that determines the value of the temporal derivative and $\boldsymbol{u}_0$ is the initial condition.

The numerical resolution of these kind of problems is mainly based on approximating the derivative for a number $N + 1$ of discrete equispaced points t_n for $n = 0, 1, 2, \ldots, N$, within the interval $[t_0, t_f]$. The temporal discretization entails also that the solution calculated will only be evaluated in the discrete points, which we will call $\boldsymbol{u}^n$. Once this is done, the derivative is substituted by a difference operator $\boldsymbol{G}$ which in general depends on the s previous steps $\boldsymbol{u}^{n+1-j}$ for $j = 1, 2, \ldots, s$, leading to the difference equation system:

$$\boldsymbol{G}(\boldsymbol{u}^{n+1}, \underbrace{\boldsymbol{u}^n, \ldots \boldsymbol{u}^{n+1-s}}_{s\ steps}; t_n, \Delta t) = \boldsymbol{F}(\boldsymbol{u}^n; t_n), \qquad \boldsymbol{u}^0 = \boldsymbol{u}_0, \tag{3.2}$$

where $\Delta t = (t_f - t_0)/N$ is the temporal step. Whenever the number of steps s is greater than 1, then $\boldsymbol{G}$ is called a multi step temporal scheme and in the case $s = 1$ is called single step temporal scheme.

The resolution will be done step by step starting from the known value. For this, is necessary to obtain the value of $\boldsymbol{u}^{n+1}$ from the equation (4.2). A priori as $\boldsymbol{G}$ is a function,

$$\boldsymbol{G} : \mathbb{R}^p \times \underbrace{\mathbb{R}^p \times \ldots \times \mathbb{R}^p}_{s\ steps} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^p, \tag{3.3}$$

it is not possible to obtain the value of $\boldsymbol{u}^{n+1}$. However, as at the instant t_n, all previous steps $\boldsymbol{u}^i$ for $i = 1, 2 \ldots, n$ and particularly the s previous steps $\boldsymbol{u}^{n+1-j}$ for $j = 1, 2, \ldots, s$ are known. Hence, it is possible to restrict the value of the difference operator to obtain a function $\tilde{\boldsymbol{G}} : \mathbb{R}^p \to \mathbb{R}^p$, this is done evaluating $\boldsymbol{G}$ in the manner:

$$\tilde{\boldsymbol{G}} = \boldsymbol{G}(\boldsymbol{u}^{n+1}, \boldsymbol{u}^n, \ldots \boldsymbol{u}^{n+1-s}; t_n, \Delta t)\Big|_{(\boldsymbol{u}^n, \ldots \boldsymbol{u}^{n+1-s}; t_n, \Delta t)}. \tag{3.4}$$

Therefore, the solution can be obtained as

$$\boldsymbol{u}^{n+1} = \tilde{\boldsymbol{G}}^{-1} \left(\boldsymbol{F}(\boldsymbol{u}^n; t_n) \right). \tag{3.5}$$

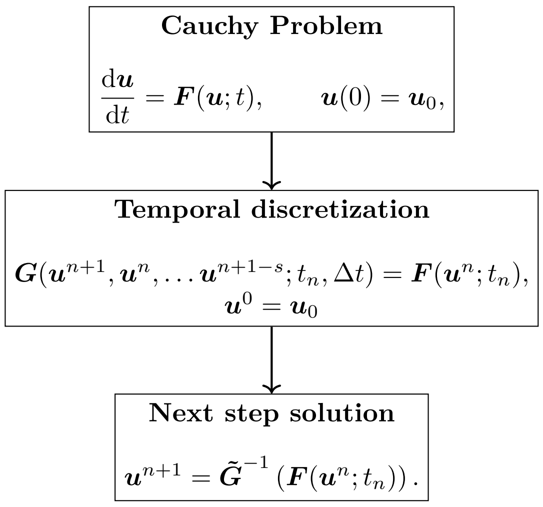

Cauchy Problem
$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{F}(\boldsymbol{u};t), \qquad \boldsymbol{u}(0)=\boldsymbol{u}_0,$
Temporal discretization
$\boldsymbol{G}(\boldsymbol{u}^{n+1},\boldsymbol{u}^{n},\dots \boldsymbol{u}^{n+1-s};t_n,\Delta t) = \boldsymbol{F}(\boldsymbol{u}^n;t_n),$
$\boldsymbol{u}^0=\boldsymbol{u}_0$
Next step solution
$\boldsymbol{u}^{n+1} = \tilde{\boldsymbol{G}}^{-1}\left(\boldsymbol{F}(\boldsymbol{u}^n;t_n)\right).$

Figure 3.1: Numerical resolution of Cauchy problems.

3.2.1 Algorithm

In order to solve a Cauchy problem in a domain $t \in [t_0, t_f]$, such as:

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \boldsymbol{F}(\boldsymbol{u},t), \qquad \boldsymbol{u}(t_0)=\boldsymbol{u}_0,$$

first, the domain of the temporal or evolution variable is discretized transforming it into a vector $\boldsymbol{t} = t_n\boldsymbol{e}_n$, where $\boldsymbol{t} \in \mathbb{R}^{N+1}$. The derivative is approximated by means of a s steps temporal scheme $\boldsymbol{G}$, leading to:

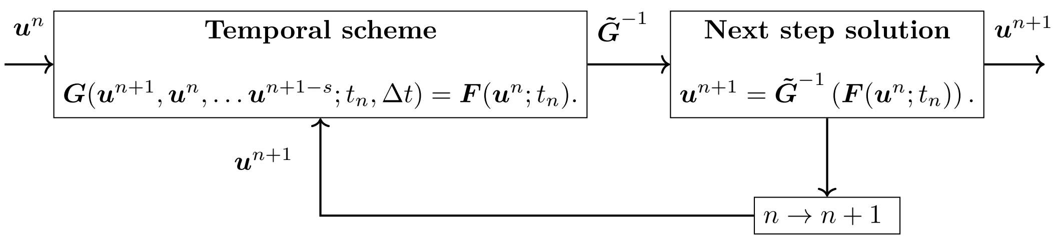
$$\boldsymbol{G}(\boldsymbol{u}^{n+1},\boldsymbol{u}^{n},\dots \boldsymbol{u}^{n+1-s};t_n,\Delta t) = \boldsymbol{F}(\boldsymbol{u}^n;t_n), \qquad \boldsymbol{u}^0=\boldsymbol{u}_0. \tag{3.6}$$

Hence, the solution at a temporal step $n+1$ can be calculated as:

$$\boldsymbol{u}^{n+1} = \tilde{\boldsymbol{G}}^{-1}\left(\boldsymbol{F}(\boldsymbol{u}^n;t_n)\right). \tag{3.7}$$

An algorithm to solve (3.6) will be presented. This algorithm works recursively, using the calculated value $\boldsymbol{u}^{n+1}$ as the new initial condition for the next step through the temporal scheme.

On figure 3.2 an scheme of the algorithm that solves Cauchy problems is shown.



$\boldsymbol{u}^n$
Temporal scheme
$\boldsymbol{G}(\boldsymbol{u}^{n+1}, \boldsymbol{u}^n, \dots \boldsymbol{u}^{n+1-s}; t_n, \Delta t) = \boldsymbol{F}(\boldsymbol{u}^n; t_n).$
$\tilde{\boldsymbol{G}}^{-1}$
Next step solution
$\boldsymbol{u}^{n+1} = \tilde{\boldsymbol{G}}^{-1}(\boldsymbol{F}(\boldsymbol{u}^n; t_n)).$
$\boldsymbol{u}^{n+1}$
$\boldsymbol{u}^{n+1}$
$n \to n+1$

Figure 3.2: Resolution algorithm for Cauchy problems.

3.2.2 Algorithm implementation

The implementation of the algorithm is done within the subroutine called `Cauchy_Problems`. In it, the temporal scheme, which gives back $\tilde{G}^{-1}$ is called recursively, obtaining the solution at each instant.

```fortran
subroutine Cauchy_ProblemS( Time_Domain, Differential_operator, Scheme, &
                            Solution )
     real, intent(in) :: Time_Domain(:)
     procedure (ODES) :: Differential_operator
     procedure (Temporal_Scheme), optional :: Scheme
     real, intent(out) :: Solution(:,:)

!  *** Initial and final time
       real :: start, finish         ! CPU time
       real ::  t1, t2               ! simulation time step
       integer ::  i, N_steps, ierr

       call cpu_time(start)
       N_steps = size(Time_Domain) - 1;

!  *** loop for temporal integration
       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
                  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

end subroutine
```

Listing 3.1: `Cauchy_problem.f90`

For a better understanding of how the temporal scheme restricts G and gives back $\tilde{G}^{-1}$ several examples shall be presented.

1. **Explicit Euler**

For explicit schemes the evaluation and inverse of G are trivial. Such is the case of the Euler explicit, shown below.

```
subroutine Euler(F, t1, t2, U1, U2, ierr )
      procedure (ODES) :: F

      real, intent(in) :: t1, t2
      real, intent(in) ::  U1(:)

      real, intent(out) ::  U2(:)
      integer, intent(out) :: ierr

         real :: t, dt

      dt = t2 - t1
      t = t1
      U2 = U1 + dt * F(U1, t)
      ierr = 0

end subroutine
```

Listing 3.2: `Temporal_Schemes.f90`

2. **Inverse Euler**

Whenever, the scheme is implicit, as for the inverse Euler, the restriction of G is done through a function called `Residual_IE` which will be used as input for the subroutine `Newton`.

```
function Residual_IE(Z) result(G)
         real, intent(in) :: Z(:)
         real :: G(size(Z))

        G = Z - dt *  F( Z, t2) - a

 end function
```

Listing 3.3: `Temporal_Schemes.f90`

The function `Residual_IE` that restricts G is contained within the interface of the subroutine `Inverse_Euler`.

```fortran
subroutine Inverse_Euler(F, t1, t2, U1, U2, ierr )
       procedure (ODES) :: F

       real, intent(in) :: t1, t2
       real, intent(in) ::  U1(:)

       real, intent(out) ::  U2(:)
       integer, intent(out) :: ierr

      real, save :: dt
      real, save, allocatable ::  a(:)


      ierr = 0
      if (.not.allocated(a) )  allocate( a(size(U1)) )

      dt = t2-t1

      a = U1
      U2 = U1

    ! Try to find a zero of the residual of the inverse Euler
      call Newton( F = Residual_IE, x0 = U2 )


     deallocate( a )


  contains
```

Listing 3.4: `Temporal_Schemes.f90`

3. Crank-Nicolson

Another example of implicit scheme is the Crank-Nicolson. Again, the restriction of G is done through a function called `Residual_CN` which will be used as input for the subroutine `Newton`.

```fortran
function Residual_CN(Z) result(G)
         real, intent(in) :: Z(:)
         real :: G(size(Z))

         G = Z - dt/2 *  F( Z, t2) - a

end function
```

Listing 3.5: `Temporal_Schemes.f90`

The function `Residual_CN` that restricts G is contained within the interface of the subroutine `Crank_Nicolson`.

```
subroutine Crank_Nicolson(F, t1, t2, U1, U2, ierr )
       procedure (ODES) :: F

       real, intent(in) :: t1, t2
       real, intent(in) ::  U1(:)

       real, intent(out) ::  U2(:)
       integer, intent(out) :: ierr

      real, save :: dt
      real, save, allocatable ::  a(:)


      ierr = 0
      if (.not.allocated(a) )  allocate( a(size(U1)) )

      dt = t2-t1

      a = U1  +  dt/2 * F( U1, t1)
      U2 = U1

      call Newton( F = Residual_CN, x0 = U2 )

      deallocate( a )


  contains
```

Listing 3.6: `Temporal_schemes.f90`

Chapter 4

Boundary Value Problems

4.1 Overview

In this chapter the mathematical foundations which are necessary to solve boundary problems are exposed. Aswell, and algorithm to solve arbitrary classical problems is exposed and its implementation. It is intended for the reader to be able to develop its own software once these concepts are internalized.

4.2 Boundary Value Problems

Whenever in the considered physical model, the magnitudes vary not only along the spatial dimension but also along time, the equations that result are partial differential equations that involve temporal derivatives. In this case, not only boundary conditions are required for the resolution of the problem but also an initial value for the unknown variable shall be provided. The set of differential equation, boundary conditions and initial value is called *Initial Value Boundary Problem*, and more rigorously can be defined as follows:

Let us be $\Omega \subset \mathbb{R}^p$ 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 $\boldsymbol{x} \in D$.

An Initial Value Boundary Problem for a vectorial function $\boldsymbol{u} : D \to \mathbb{R}^{N_v}$ of N_v

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 for the solution at the boundary points $\boldsymbol{u}\big|_{\partial\Omega}$.

If the spatial domain D is discretized in N_D points, the problem extends from vectorial to tensorial, as a tensorial system of equations appears from each variable of $\boldsymbol{u}$. The order of the tensorial system is $p \times N_v$ and therefore its number of elements N is $N = p \times N_v \times N_D$. 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 tensorial system evaluated on the boundary points is $N_C = p \times N_v \times N_{\partial\Omega}$. Once the spatial discretization is done, the system emerges as a tensorial difference equation that can be rearranged into a vectorial 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:

$$F(U) = 0,$$

$$H(U)\big|_{\partial\Omega} = 0,$$

where $U \in \mathbb{R}^N$ is the solution in all domain, $U\big|_{\partial\Omega} \in \mathbb{R}^{N_C}$ is the solution on the boundary points of the grid, $F : \mathbb{R}^N \to \mathbb{R}^{N-N_C}$ is the difference operator associated to the differential operator and $H : \mathbb{R}^N \to \mathbb{R}^{N_C}$ is the difference operator of the boundary conditions. Notice that the operators can not be solved separately as they are not invertible functions.

To solve both systems they must be joined into a complete difference equation operator $S : \mathbb{R}^N \to \mathbb{R}^N$, such that both equations for F and G are satisfied when:

$$S(U) = 0.$$

Hence, the boundary value problem is transformed into N difference equations for U. The intention of this section is to define and implement an algorithm which solves this system.

The resolution method used for both cases is quite similar but with a slight variation. In the linear case, the difference equation system can be written in matrix form with constant coefficients, and therefore solved by LU method. This affects on how the differential operator is implemented in the code to its posterior resolution. On the other hand in the non linear case, the resolution of the difference equation system must be done by means of an iterative method. Also, it affects the way the differential operator is constructed, which in general cannot be written in matrix form. Nevertheless, this only affects on how the algorithm is implemented and not to its definition. The algorithm is the same for both linear and non linear problems but the way its carried out differs slightly. As this section intends to explain how the algorithm works and is implemented, it will present only the code for non linear cases. However, the reader must take in account that there is a different code specifically for linear problems even though is not included in this section.

4.2.1 Algorithm.

The algorithm solves for a vectorial function $\boldsymbol{u} : D \to \mathbb{R}^{N_v}$ of N_v variables, problems such as:

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

Once the problem is discretized it results:

$$\begin{aligned} F(U) &= 0, \\ H(U)\big|_{\partial\Omega} &= 0, \end{aligned}$$

The main idea of the resolution algorithm is first use F and G to construct the operator H to then solve it.

The algorithm can be explained in two steps:

1. **Difference operator on inner and boundary points.**

In first place we use the difference operators for both inner and boundary points F and G, to construct an operator S for all points of the domain. That is, defining S as:

$$S(U) = \begin{bmatrix} F(U) \\ G(U)\big|_{\partial\Omega} \end{bmatrix}.$$

Defined this way, it is in the required form to be solved.

2. **Difference equation resolution.**

Finally, using the difference operator S, the problem:

$$S(U) = 0,$$

must be solved. For this, as the system in general is not linear, an iterative method is required. Once the system is solved, the solution at all domain U is known.

In the particular case in which both the differential operator and boundary conditions are linear, the difference operator S can be written as a linear algebraic system:

$$S(U) = AU + \boldsymbol{b} = 0. \tag{4.1}$$

Thus, the solution of the boundary value problem in the linear case results:

$$U = -A^{-1} \cdot \boldsymbol{b}.$$

The algorithm for both linear and non linear case are represented schematically on figure 4.1.

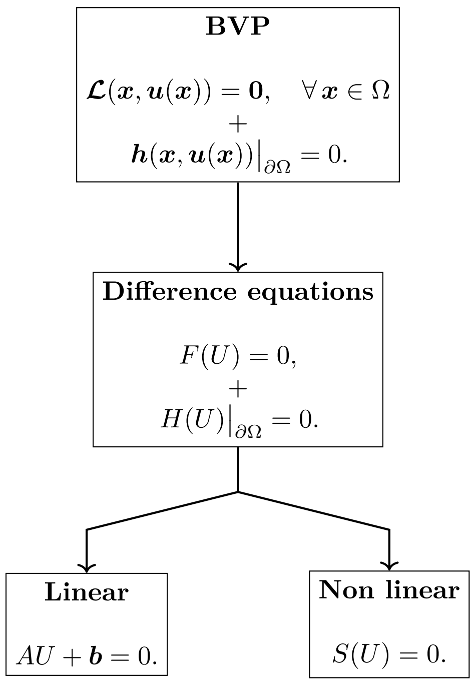

BVP
$\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x})) = \boldsymbol{0}, \quad \forall\, \boldsymbol{x} \in \Omega$
$+$
$\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))\big|_{\partial\Omega} = 0.$
Difference equations
$F(U) = 0,$
$+$
$H(U)\big|_{\partial\Omega} = 0.$
Linear
$AU + \boldsymbol{b} = 0.$
Non linear
$S(U) = 0.$

Figure 4.1: Linear and non linear boundary value problems.

4.3 Algorithm Implementation

In order to implement the resolution of boundary value problems several cases are considered. There are two ways of classifying the problems which are able to be solved by means of the provided code. The first is in terms of the dimension of the spatial domain or in other words the number of independent variables which constitute $\boldsymbol{x}$. Thus, the problems can be 1D, 2D or 3D. The second classification depends on the dimension of the unknown variable $\boldsymbol{u}(\boldsymbol{x})$. The two situations that can be considered are when $\boldsymbol{u}$ is a real scalar, that is $N_v = 1$ and when $\boldsymbol{u}$ is considered a real vectorial field of dimension $N_v > 1$. All these cases are considered separately and then agglutinated in a single module as follows:

```
module Boundary_value_problems

use Boundary_value_problems1D
use Boundary_value_problems2D
use Boundary_value_problems3D

implicit none

private

public :: Boundary_Value_Problem
public :: linear2D, linear1D, linear3D



 interface Boundary_Value_Problem
      module procedure Boundary_Value_Problem1D, &
                       Boundary_Value_Problem2D, &
                       Boundary_Value_Problem2D_system
 end interface



contains



end module
```

Listing 4.1: `Boundary_value_problems.f90`

For the sake of simplicity, as the functioning of the algorithm for each case is quite similar for 1D, 2D and 3D problems, only the 1D case will be explained in order for the reader to understand the basis of the algorithm and its implementation.

4.3.1 Boundary Value Problems 1D

As it was stated before, depending on the linearity or not of the differential operator $\boldsymbol{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{x}))$, the resolution of the problem differs. For this reason, first of all in order to solve 1-dimensional scalar boundary value problems it is necessary to determine whether the problem is linear or not, this is done by means of the subroutine `Boundary_Value_Problem1D`, written as follows:

```
subroutine Boundary_Value_Problem1D( x_nodes, Order, Differential_operator, &
                                     Boundary_conditions, Solution, Solver )

     real, intent(in) :: x_nodes(0:)
     integer, intent(in) :: Order
     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, Order,                 &
                                             Differential_operator,          &
                                             Boundary_conditions, Solution)

     else

       call Non_Linear_Boundary_Value_Problem1D( x_nodes, Order,             &
                                                 Differential_operator,      &
                                                 Boundary_conditions,        &
                                                 Solver, Solution)

     end if


end subroutine
```

Listing 4.2: `Boundary_value_problems1D.f90`

For each linear and non linear case the implementation shall be presented.

Linear Boundary Value Problems 1D

The structure that will be followed is the same as in the algorithm, that is each step will be related to its implementation. The starting point of the explanation will be once the spatial domain is discretized in $N_x + 1$ points. Thus, the solution $U \in \mathbb{R}^{N_x+1}$ is the solution of the difference equations system $S(U) = 0$.

1. **Difference operator on inner and boundary points.**

First of all it is necessary to construct the difference operator $S(U)$, the construction method that is used is based on the delta Kronecker function to recursively construct it. For this, it is defined a subroutine called `Difference_equation1DL` which given a spatial domain `x`, a vector in which is yielded the solution `W`, the first and second derivatives `Wx` and `Wxx` yields the difference operator in `F`.

```fortran
subroutine Difference_equation1DL(x, W, Wx, Wxx, F)
           real, intent(in) :: x(0:), W(0:)
           real, intent(out) :: Wx(0:), Wxx(0:), F(0:)

    integer :: i
    real :: D, C


        if (dU(1)) call Derivative( "x", 1, W, Wx)
        if (dU(2)) call Derivative( "x", 2, W, Wxx)

!  ***  boundary conditions
        do i=0, N, N
             D = Differential_operator( x(i), W(i), Wx(i), Wxx(i) )
             C = Boundary_conditions( x(i), W(i), Wx(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(i), W(i), Wx(i), Wxx(i) )
        enddo

end subroutine
```

Listing 4.3: `Boundary_value_problems1D.f90`

The delta Kronecker method constructs the linear difference operator $S(U)$ by giving values to the solution U. We shall asign the subindex m to the variables of the m-th step of the Delta Kronecker. In first place, for $m = 0$ it is assigned the null value $U_0 = 0$. Hence, the first value S_0 is the non homogenous or forcing term of the equation

$$S_0 = \boldsymbol{b}. \tag{4.2}$$

For the rest of steps the value given to U_m is all components equal to zero excepting the m-th term of the vector which is equal to 1. Therefore, it can be expressed in terms of the delta Kronecker function δ_{mj} as:

$$U_m = \delta_{mj}\boldsymbol{e}_j, \qquad \text{for} \quad j = 1, 2 \dots, N_x + 1. \tag{4.3}$$

When evaluated in the $m{-}th$ step the image of the difference operator $S_m = S(U_m)$ takes the value of the $m - th$ column of the matrix $A = A_{ij}\boldsymbol{e}_i \otimes \boldsymbol{e}_j$ minus the forcing term. In other words:

$$S_m = A_{im}\boldsymbol{e}_i + \boldsymbol{b}, \qquad \text{for} \quad i, m = 1, 2 \dots, N_x + 1. \tag{4.4}$$

Hence, recursively the matrix A can be constructed as:

$$A = \left[S_1 | S_2 \dots | S_{N_x+1} \right]. \tag{4.5}$$

The method is implemented by calling recursively a subroutine which performs the previous operations, called `Difference_equation`.

```
!   *** independent term  A U = b  ( U = inverse(A) b )
       U = 0
       call Difference_equation1DL(x_nodes, U, Ux, Uxx, bi)



!   *** Delta kronecker to calculate the difference operator
       do i=0, N
          U(0:N) = 0
          U(i) = 1.0

          call Difference_equation1DL(x_nodes, U, Ux, Uxx, F)
          Difference_operator(0:N, i) = F - bi
       enddo
```

Listing 4.4: `Boundary_value_problems1D.f90`

Once this is done the matrix A is known and yielded on the array `Difference_operator` ready to be solved.

2. **Difference equation resolution.**

Once known A the last step is to solve the problem by the LU method:

```
!   *** solve the linear system of equations
       call LU_Factorization(Difference_operator)

       Solution = Solve_LU( Difference_operator, -bi )
```

Listing 4.5: `Boundary_value_problems1D.f90`

And the solution will be yielded on the array `Solution`.

In order to perform the delta Kronecker method, the previously defined subroutine called `Difference_equation` must be contained on the interface of a subroutine called `Linear_Boundary_Value_Problem1D`.

```fortran
subroutine Linear_Boundary_Value_Problem1D( x_nodes, Order,
    Differential_operator, &
                                        Boundary_conditions, Solution)

     real, intent(in) :: x_nodes(0:)
     integer, intent(in) :: Order
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     real, intent(out) :: Solution(0:)


!  *** auxiliary variables
       integer ::  i, N

       real, allocatable ::  bi(:), F(:), U(:), Ux(:), Uxx(:),  &
                             Difference_operator(:,:)

!  *** Integration domain
       N = size(x_nodes) - 1
       allocate( bi(0:N), F(0:N), U(0:N), Ux(0:N), Uxx(0:N),    &
                 Difference_operator(0:N, 0:N) )

!  *** independent term  A U = b  ( U = inverse(A) b )
       U = 0
       call Difference_equation1DL(x_nodes, U, Ux, Uxx, bi)



!  *** Delta kronecker to calculate the difference operator
       do i=0, N
          U(0:N) = 0
          U(i) = 1.0

          call Difference_equation1DL(x_nodes, U, Ux, Uxx, F)
          Difference_operator(0:N, i) = F - bi
       enddo

!  *** solve the linear system of equations
       call LU_Factorization(Difference_operator)

       Solution = Solve_LU( Difference_operator, -bi )

      deallocate( U, Ux, Uxx, F, Difference_operator, bi )

contains
```

Listing 4.6: `Boundary_value_problems1D.f90`

Non Linear Boundary Value Problems 1D

The implementation of the algorithm for non linear boundary value problems is simpler from the programming point of view (however more costly computationally). The reason is that in this case the subroutine which calculates the differential operator on each point of the spatial domain `Difference_equation1DNL` is recursively called when its restriction is introduced on the `Newton` subroutine as an input argument. In the linear case as the delta Kronecker method is used, the differential operator is obtained recursively which requires writing more lines of code.

As before, the subroutine `Difference_equation1DNL` is defined in a similar manner as the equivalent on the linear case. However, in this case it is not necessary to obtain the values of the derivatives, yielded on `Wx` and `Wxx`.

```fortran
subroutine Difference_equation1DNL(x, W, F)
           real, intent(in) :: x(0:), W(0:)
           real, intent(out) :: F(0:)

    real :: Wx(0:N), Wxx(0:N)
    integer :: i
    real :: C, D

     if (dU(1)) call Derivative( "x", 1, W, Wx)
     if (dU(2)) call Derivative( "x", 2, W, Wxx)

 !  ***  boundary conditions
         do i=0, N, N
             D = Differential_operator( x(i), W(i), Wx(i), Wxx(i) )
             C = Boundary_conditions( x(i), W(i), Wx(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(i), W(i), Wx(i), Wxx(i) )
         enddo


end subroutine
```

Listing 4.7: `Boundary_value_problems1D.f90`

In order to make this subroutine suitable for the `Newton` solver it is precise to define a restricting intermediate function called `System_BVP` defined as:

```
 function System_BVP(U) result(F)
        real, intent (in) :: U(:)
        real :: F(size(U))


            call Difference_equation1DNL(x_nodes, U, F)

end function
```

Listing 4.8: `Boundary_value_problems1D.f90`

Thus, the problem can be solved by a simple call:

```
!  *** Non linear solver
      if (present(Solver))  then
                  call Solver(System_BVP, Solution)
      else
                  call Newton(System_BVP, Solution)
      end if
```

Listing 4.9: `Boundary_value_problems1D.f90`

With this it is just necessary to give the code an structure which permits to call both procedures from the interface of the global subroutine.

To permit the lexical scope, both the function `System_BVP` and the subroutine `Difference_equation1DNL` must be contained within the interface of a higher level subroutine which is called `Non_Linear_Boundary_Value_Problem1D`:

```fortran
subroutine Non_Linear_Boundary_Value_Problem1D( x_nodes, Order,             &
              Differential_operator,  Boundary_conditions, Solver, Solution)

     real, intent(in) :: x_nodes(0:)
     integer, intent(in) :: Order
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     procedure (NonLinearSolver), optional:: Solver
     real, intent(inout) :: Solution(0:)


!  *** variables specification
       integer ::   N


!  *** Integration domain
       N = size(x_nodes) - 1

!  *** Non linear solver
       if (present(Solver))  then
                   call Solver(System_BVP, Solution)
       else
                   call Newton(System_BVP, Solution)
       end if


    contains
```
Listing 4.10: `Boundary_value_problems1D.f90`

Chapter 5

Initial Value Boundary Problems

5.1 Overview

In this chapter, a resolution method for initial value boundary problems will be exposed. First, a brief mathematical presentation shall be given, followed by an algorithm which solves the kind of problem presented. To end, how this algorithm is implemented in Fortran language is described. With this, it is intended for the reader to be able to understand and assimilate the necessary concepts which permit to solve initial value boundary problems.

5.2 Initial Value Boundary Problems

Whenever in the considered physical model, the magnitudes vary not only along the spatial dimension but also along time, the equations that result are usually partial differential equations that involve temporal derivatives. In this case, not only boundary conditions are required for the resolution of the problem but also an initial value for the unknown variable shall be provided. The set of differential equation, boundary conditions and initial value is called *Initial Value Boundary Problem*, and more rigorously can be defined as follows:

Let us be $\Omega \subset \mathbb{R}^p$ 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 Value Boundary Problem for a vectorial 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 $\boldsymbol{u}\big|_{\partial\Omega}$.

If the spatial domain D is discretized in N_D points, the problem extends from vectorial to tensorial, as a tensorial system of equations appears from each variable of $\boldsymbol{u}$. The order of the tensorial system is $p \times N_v$ and therefore its number of elements N is $N = p \times N_v \times N_D$. 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 tensorial system evaluated on the boundary points is $N_C = p \times N_v \times N_{\partial\Omega}$. Once the spatial discretization is done, even though the system emerges as a tensorial Cauchy Problem, it can be rearranged into a vectorial system of N equations. Particularly two systems appear: one of $N - N_C$ equations from the differential operator and another of N_C equations from the boundary conditions. The Cauchy Problem, this way results:

$$
\frac{dU_\Omega}{dt} = F(U;t), \qquad H(U;t)\big|_{\partial\Omega} = 0,
$$

$$
U(t_0) = U^0,
$$

where $U \in \mathbb{R}^N$ is the solution in all domain, $U_\Omega \in \mathbb{R}^{N-N_C}$ is the solution on the inner points of the grid, $U\big|_{\partial\Omega} \in \mathbb{R}^{N_C}$ is the solution on the boundary points of the grid, $U^0 \in \mathbb{R}^N$ 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, the differential equations system is transformed into a Cauchy Problem of N equations that in general is not linear.

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. As the solution is evaluated only in these points, from now on we will use the notation for every temporal step t_n: $U_\Omega(t_n) = U_\Omega^n$ and $U(t_n) = U^n$. The Cauchy Problem transforms into a difference equation system introducing a s-steps temporal scheme:

$$G(U_\Omega^{n+1}, \underbrace{U^n, \dots U^{n+1-s}}_{s\ steps}; t_n, \Delta t) = F(U^n; t_n),$$

$$H(U^n; t_n)\big|_{\partial\Omega} = 0, \qquad U(t_0) = U^0,$$

where

$$G : \mathbb{R}^{N-N_C} \times \underbrace{\mathbb{R}^N \times \dots \times \mathbb{R}^N}_{s\ steps} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{N-N_C},$$

is the difference operator associated to the temporal derivative for the temporal scheme and Δt is the temporal step. Thus, at each temporal step two systems of N_C and $N - N_C$ equations appear.

Once the temporal discretization is done, the problem reduces to two difference equations systems, that in total sum N equations at each temporal step.

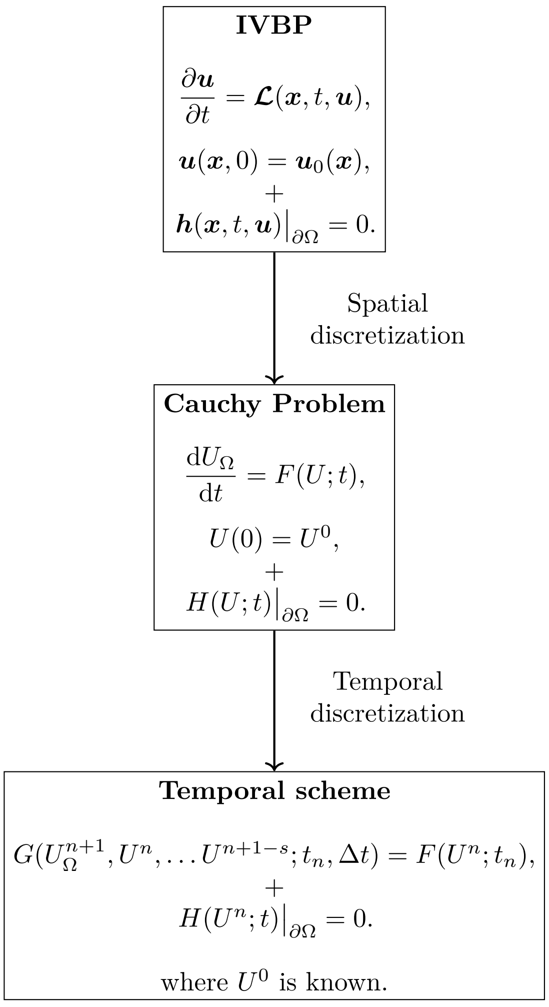

IVBP
$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}),$
$\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}),$
$+$
$\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u})\big|_{\partial\Omega} = 0.$
Spatial discretization
Cauchy Problem
$\frac{\mathrm{d}U_\Omega}{\mathrm{d}t} = F(U; t),$
$U(0) = U^0,$
$+$
$H(U; t)\big|_{\partial\Omega} = 0.$
Temporal discretization
Temporal scheme
$G(U_\Omega^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t) = F(U^n; t_n),$
$+$
$H(U^n; t)\big|_{\partial\Omega} = 0.$
where U^0 is known.

Figure 5.1: Line method for initial value boundary problems.

5.2.1 Algorithm.

The algorithm solves for a function $\boldsymbol{u} : D \times \mathbb{R} \to \mathbb{R}^{N_v}$, the evolution problems of the type:

$$\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x}, t) = \boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t)), \qquad \forall \;\; \boldsymbol{x} \in \Omega,$$
$$\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t))\big|_{\partial\Omega} = 0, \qquad \forall \;\; \boldsymbol{x} \in \partial\Omega,$$
$$\boldsymbol{u}(\boldsymbol{x}, t_0) = \boldsymbol{u}_0(\boldsymbol{x}),$$

This problem will be solved numerically, hence, a distinction between spatial and temporal dependence and their discretizations shall be done. Starting from the spatial discretization, the differential equations systems transforms into:

$$\frac{dU_\Omega}{dt} = F(U; t), \qquad H(U; t)\big|_{\partial\Omega} = 0,$$
$$U(t_0) = U^0.$$

The spatially discretized problem behaves as a Cauchy Problem system of $N - N_C$ equations for $U_\Omega(t)$ and a system of N_C equations for $U(t)\big|_{\partial\Omega}$.

Discretizing temporarily the problem using a s-step scheme, it transforms in a difference equations system of N equations for each temporal step:

$$G(U_\Omega^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t) = F(U^n; t_n),$$
$$H(U^n; t_n)\big|_{\partial\Omega} = 0, \qquad U(t_0) = U^0.$$

The main idea of the resolution algorithm is first to discretize the spatial domain with the initial value U^0. Then construct the difference operator F. Finally, using a temporal scheme obtain the value of the next temporal step $U_\Omega(t_1) = U_\Omega^1$. Hence, recursively the problem can be solved introducing U_Ω^1 as the initial value for the next iteration. As the algorithm works recursively it shall be explained for a generic step, taking U_Ω^n as initial value.

The algorithm can be explained in three steps:

1. **Boundary points from inner points.**

In first place, the known initial value at the inner points, U_Ω^n, is used to calculate the solution for the boundary conditions. That is, solving the system of equations:

$$H(U^n; t_n)\big|_{\partial\Omega} = 0,$$

which gives back the value of the solution at boundaries $U^n\big|_{\partial\Omega}$.

Even though this might look redundant for the initial value U^0(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.

2. **Difference operator for inner points.**

Next, using the previous solution $U^n\big|_{\partial\Omega}$ at the boundary points and the initial value U^n_Ω the derivatives at the inner points are calculated. These derivatives are used to construct the difference operator $F(U^n; t_n)$ for the inner points.

3. **Temporal step.**

Finally, the difference operator previously obtained, F, is used to calculate the solution at inner points at the next temporal step U^{n+1}_Ω. This means solving the system:

$$G(U^{n+1}_\Omega, U^n, \dots U^{n+1-s}; t_n, \Delta t) = F_U(U^n; t_n).$$

In this system, the values of the solution at the s steps are known and therefore, the unknown quantity of the system is the solution at the next temporal step U^{n+1}_Ω. 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:

$$\tilde{G}(U^{n+1}_\Omega) = G(U^{n+1}_\Omega, U^n, \dots U^{n+1-s}; t_n, \Delta t)\bigg|_{(U^n, \dots U^{n+1-s}; t_n, \Delta t)}$$

such that,

$$\tilde{G} : \mathbb{R}^{N-N_C} \to \mathbb{R}^{N-N_C}.$$

Hence, the solution at the next temporal step for the inner points results:

$$U^{n+1}_\Omega = \tilde{G}^{-1}(F(U^n; t_n)).$$

For example, if it is used an explicit Euler scheme over an equispace domain with temporal step Δt the problem transforms into:

$$U^{n+1}_\Omega = U^n_\Omega + F(U^n; t_n)\Delta t,$$

which gives the value of the solution at the next temporal step U^{n+1}_Ω. This value, will be used as an initial value for the next iteration, closing the loop of the algorithm. The philosophy for other temporal schemes is the same, the result is the solution at the next temporal step.

The sequence of the algorithm is represented on figure 5.2, with the inputs and outputs of each step.

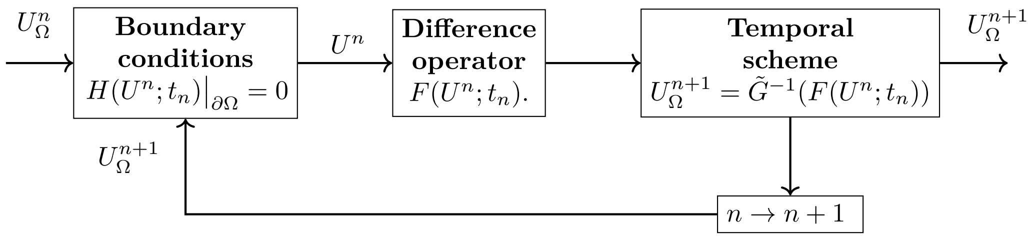


Figure 5.2: Resolution algorithm for initial value boundary problems.

5.2.2 Algorithm implementation

In the following pages the implementation of the algorithm previously presented will be explained. The scheme followed in the explanation shall be the same as in the algorithm, following it step by step.

Initial value boundary problems 1D

1. **Boundary points from inner points.**

The first step is to solve the values of the solution at the boundary points $U^n\big|_{\partial\Omega}$ known the inner points U_Ω^n. For this, the boundary conditions must be introduced by the user as a function which satisfy a certain interface given by the procedure:

```
real function BC1D(x, t, u, ux)
    real, intent(in) :: x, t, u, ux
end function
```

Listing 5.1: `Initial_Value_Boundary_Problem1D.f90`

On the function defined by the user, the boundary conditions can be specified through the operator $\boldsymbol{h}(\boldsymbol{x},t,\boldsymbol{u}(\boldsymbol{x},t))\big|_{\partial\Omega}$ written explicitly or by means of the keywords `FREE_BOUNDARY_CONDITION` and `PERIODIC_BOUNDARY_CONDITION` which correspond to open and periodic boundary condition respectively.

Once introduced the resolution of the boundary conditions must distinguish between these three situations by a conditional sentence. In the case in which the boundary conditions are periodic there is no need of computing any resolution as is imposed the equality of the solution value at the extremes. In the rest of cases the solution is yielded on an allocatable vector `Ub`.

```fortran
!  ***  solve a system of two equations to yield the values at the 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:Nx:Nx)  = Ub(1:2)

        end if
```

Listing 5.2: `Initial_Value_Boundary_Problem1D.f90`

In the case in which there is an open boundary in $x = x_f$ there is a defined function called `BCs1` which receives the procedure. In this function, the array which will yield the solution U, called `Solution` must point the automatic variable of the function and the derivative at the boundary must be computed.

```
function BCs1(Y) result(G)
        real, intent (in) :: Y(:)
        real :: G(size(Y))

    real :: Wx(0:Nx)

    Solution(it, 0) = Y(1)

    call Derivative( "x", 1, Solution(it,:), Wx)

    G(1)= Boundary_conditions( x_nodes(0),  t_BC, Solution(it,0),  Wx(0)  )

end function
```

Listing 5.3: `Initial_Value_Boundary_Problem1D.f90`

Whenever the boundary conditions are given explicitly by a function a function called `BCs2` which receives the procedure is defined. Again the solution must be pointed and the derivative computed.

```
function BCs2(Y) result(G)
        real, intent (in) :: Y(:)
        real :: G(size(Y))

    real :: Wx(0:Nx)


    Solution(it, 0)  = Y(1)
    Solution(it, Nx) = Y(2)

    call Derivative( "x", 1, Solution(it,:), Wx)

    G(1)= Boundary_conditions( x_nodes(0),  t_BC, Solution(it,0),  Wx(0)  )
    G(2)= Boundary_conditions( x_nodes(Nx), t_BC, Solution(it, Nx), Wx(Nx) )

end function
```

Listing 5.4: `Initial_Value_Boundary_Problem1D.f90`

At this point, the solution U^n in all spatial domain is known, which will permit the calculation of the difference operator.

2. **Difference operator for inner points.**

Once known U^n the difference operator can be computed from an user defined function for the differential operator which must suit the interface:

```
      real function DifferentialOperator1D(x, t, u, ux, uxx)
                       real, intent(in) :: x, t, u, ux, uxx
      end function
```

Listing 5.5: `Initial_Value_Boundary_Problem1D.f90`

Before receiving it, in order to save memory and optimize the functioning of the code a logical array `dU` is checked in order to calculate only the derivatives present on the differential operator. Once done, the value of the difference operator F^n is yielded at the array `F`.

```
!  *** inner grid points
        F = 0
        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
```

Listing 5.6: `Initial_Value_Boundary_Problem1D.f90`

The first two steps of this algorithm are contained in a higher level subroutine called `Space_Discretization1D`, which will be called recursively for each temporal step.

```
subroutine Space_discretization1D( U, t, F )
          real ::  U(0:Nx), t
          real ::  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(:)

!  ***  solve a system of two equations to yield the values at the 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:Nx:Nx)   = Ub(1:2)

        end if


!  *** inner grid points
        F = 0
        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 5.7: `Initial_Value_Boundary_Problem1D.f90`

3. **Temporal step.**

Once known the difference operator, it is possible to calculate the solution at the next step for the inner points U_Ω^{n+1}. For this we will reuse the subroutine `Cauchy_Problems` explained on 3. In order to do it, is necessary to define a function which restricts the subroutine `Space_Discretization1D` to satisfy the interface of `Cauchy_Problems`. This function is written as follows and called `Space_Discretization`:

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

         call Space_discretization1D( U, t, F )


end function
```

Listing 5.8: `Initial_Value_Boundary_Problem1D.f90`

Hence, the next step solution is computed by calling `Cauchy_Problems` with the temporal domain and `Space_Discretization` as input arguments:

```fortran
    call Cauchy_ProblemS(  Time_Domain = Time_Domain,                    &
                           Differential_operator = Space_discretization, &
                           Scheme = Scheme, Solution = Solution )
```

Listing 5.9: `Initial_Value_Boundary_Problem1D.f90`

This will give back the solution at the next step U_Ω^{n+1} yielded on `Solution`. This value will be the new initial condition that closes the loop of the algorithm.

In order to structure all this operations the previous functions and subroutines are contained on a higher level subroutine called IVBP1D:

```fortran
subroutine IVBP1D( Time_Domain, x_nodes, Order, Differential_operator,
    Boundary_conditions, Scheme, Solution)

     real, intent(in) :: Time_Domain(:)
     real, intent(inout) :: x_nodes(0:)
     integer, intent(in) :: Order
     procedure (DifferentialOperator1D) :: Differential_operator
     procedure (BC1D) ::  Boundary_conditions
     procedure (Temporal_Scheme), optional :: Scheme
     real, intent(out) :: Solution(0:,0:)
     real :: t_BC
     integer ::  it
     logical :: dU(2) ! matrix of dependencies( direction, order )

     integer :: Nx, Nt, Nt_u, Nx_u
     Nx = size(x_nodes) - 1
     Nt = size(Time_Domain) - 1

     Nx_u = size( Solution(0, :) ) - 1
     Nt_u = size( Solution(:, 0) ) - 1

     dU = Dependencies_IVBP_1D( Differential_operator )

     if (Nx /= Nx_u .or. Nt /= Nt_u ) then

             write(*,*) " ERROR : Non conformal dimensions Time_Domain,
    x_nodes, Solution"
             write(*,*) " Nx = ", Nx, " Nt = ", Nt
             write(*,*) " Nx_u = ", Nx_u, " Nt_u = ", Nt_u
             stop

     end if

    call Cauchy_ProblemS(  Time_Domain = Time_Domain,                    &
                           Differential_operator = Space_discretization, &
                           Scheme = Scheme, Solution = Solution )


contains
```

Listing 5.10: Initial_Value_Boundary_Problem1D.f90

5.3 Error on linear problems

Whenever an initial value boundary problem is discretized spatially and temporarily, two kinds of error appear in the resolution.

1. An error associated to the spatial discretization E.

2. An error associated to the temporal discretization E^n.

If it is defined $\boldsymbol{u}(\boldsymbol{x}_i, t)$ as the exact value of the solution on the point $\boldsymbol{x}_i$ and $u_i(t)$ is the approximated value. Then,

$$\boldsymbol{\varepsilon}_i(t) = \boldsymbol{u}(\boldsymbol{x}_i, t) - \boldsymbol{u}_i(t).$$

is the spatial error at the point $\boldsymbol{x}_i$.

Besides, if $\boldsymbol{u}_i(t)$ is the exact solution of an ordinary differential equations system and $\boldsymbol{u}_i^n$ represents the value of the approximated solution in a point $\boldsymbol{x}_i$ of the grid at an instant $t = t_n$, the error of both spatial and temporal discretization is defined as:

$$\boldsymbol{\varepsilon}_{T,i}^n = \boldsymbol{u}(\boldsymbol{x}_i, t_n) - \boldsymbol{u}_i^n,$$

which can be written as

$$\boldsymbol{\varepsilon}_{T,i}^n = \underbrace{\boldsymbol{u}(\boldsymbol{x}_i, t_n) - \boldsymbol{u}_i(t_n)}_{\boldsymbol{\varepsilon}_i(t_n)} + \underbrace{\boldsymbol{u}_i(t_n) - \boldsymbol{u}_i^n}_{\boldsymbol{\varepsilon}_i^n},$$

or written in vectorial form:

$$E_T = E + E^n,$$

where E is the spatial error and E^n is the temporal error.

Once defined is interesting analysing the behaviour of both E and E^n. In the case in which the operators $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{h}$ are linear, the initial value boundary problem can be spatially discretized as a linear Cauchy Problem. As before, starting from an initial value boundary problem with Dirichlet boundary conditions of the shape:

$$\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x}, t) = \boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t)), \qquad \forall \;\; \boldsymbol{x} \in \Omega,$$
$$\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u}(\boldsymbol{x}, t))\big|_{\partial\Omega} = 0, \qquad \forall \;\; \boldsymbol{x} \in \partial\Omega,$$
$$\boldsymbol{u}(\boldsymbol{x}, t_0) = \boldsymbol{u}_0(\boldsymbol{x}),$$

when the spatial domain is discretized in $N_D = N_\Omega + N_{\partial\Omega}$ a linear Cauchy Problem results:

$$\frac{d\boldsymbol{u}_i}{dt} = A_i(t)[\boldsymbol{u}_j], \qquad \boldsymbol{u}_i(t_0) = \boldsymbol{u}_i^0, \quad \text{for} \quad i = 1, 2, ..., N_\Omega$$

where $\boldsymbol{u}_i : \mathbb{R} \to \mathbb{R}^{N_v}$, $[\boldsymbol{u}_j] = \left(\boldsymbol{u}_1^T|...|\boldsymbol{u}_N^T\right)^T$ / $[\boldsymbol{u}_j] : \mathbb{R} \to \mathbb{R}^{N_\Omega}$ and $A_i : \mathbb{R} \to \mathcal{M}_{N_v \times N_\Omega}$. If the exact partial differential equations system is evaluated at $\boldsymbol{x}_i$ and it is substracted the Cauchy Problem, it results:

$$\frac{\partial(\boldsymbol{u}(\boldsymbol{x}_i,t) - \boldsymbol{u}_i(t))}{\partial t} = \frac{\mathrm{d}\boldsymbol{\varepsilon}_i}{\mathrm{d}t} = \boldsymbol{\mathcal{L}}(\boldsymbol{x}_i, t, \boldsymbol{u}(\boldsymbol{x}_i,t)) - A_i(t)[\boldsymbol{u}_j],$$

which adding and substracting the exact solution evaluated at the inner points $[\boldsymbol{u}(\boldsymbol{x}_j,t)] = \left(\boldsymbol{u}^T(\boldsymbol{x}_1,t)|...|\boldsymbol{u}^T(\boldsymbol{x}_{N_\Omega},t)\right)^T$ / $[\boldsymbol{u}_j] : \mathbb{R} \to \mathbb{R}^{N_\Omega}$ multiplied by $A_i(t)$ can be written:

$$\frac{\mathrm{d}\boldsymbol{\varepsilon}_i}{\mathrm{d}t} = \boldsymbol{\mathcal{L}}(\boldsymbol{x}_i, t, \boldsymbol{u}(\boldsymbol{x}_i,t)) - A_i(t)[\boldsymbol{u}_j] + A_i(t)[\boldsymbol{u}(\boldsymbol{x}_j,t)] - A_i(t)[\boldsymbol{u}(\boldsymbol{x}_j,t)],$$

$$\frac{\mathrm{d}\boldsymbol{\varepsilon}_i}{\mathrm{d}t} = A_i(t)[\boldsymbol{u}(\boldsymbol{x}_j,t)] - A_i(t)[\boldsymbol{u}_j] + \underbrace{\boldsymbol{\mathcal{L}}(\boldsymbol{x}_i, t, \boldsymbol{u}(\boldsymbol{x}_i,t)) - A_i(t)[\boldsymbol{u}(\boldsymbol{x}_j,t)]}_{\boldsymbol{r}_i(t)},$$

where $\boldsymbol{r}_i(t)$ is the truncation error, associated to the error on the finite differences. The terms of this error are of order $\mathcal{O}(1/\Delta x^q)$, where q is the finite difference scheme order. Thus, the spatial error can be written:

$$\frac{\mathrm{d}\boldsymbol{\varepsilon}_i}{\mathrm{d}t} = A_i(t) \cdot [\boldsymbol{\varepsilon}_j] + \boldsymbol{r}_i(t),$$

which can be written as vectors of $\mathbb{R}^{N-N_C}$, where $N = N_v \cdot N_D$ and $N_C = N_v \cdot N_{\partial\Omega}$:

$$\frac{\mathrm{d}E}{\mathrm{d}t} = A(t) \cdot E(t) + R(t).$$

where $E : \mathbb{R} \to \mathbb{R}^{N-N_C}$ is the spatial error, $R : \mathbb{R} \to \mathbb{R}^{N-N_C}$ is the truncation error and $A : \mathbb{R} \to \mathcal{M}_{N-N_C \times N-N_C}$.

Thus, a linear system of ordinary differential equations appears for the spatial error E, whose solution is:

$$E(t) = \Phi(t) \cdot E(t_0) + \Phi(t) \int_{t_0}^{t} \Phi(s)^{-1} \cdot R(s) \mathrm{d}s,$$

where $\Phi(t)$ is the fundamental matrix, which satisfies:

$$\frac{\mathrm{d}\Phi}{\mathrm{d}t} = A(t)\Phi(t), \qquad \Phi(t_0) = I.$$

From the general solution for $E(t)$ an upper bound for its norm $\|E(t)\|$ can be stablished:

$$\begin{aligned}\|E\| &= \left\|\Phi(t)\cdot E(t_0) + \Phi(t)\int_{t_0}^{t}\Phi(s)^{-1}R(s)\mathrm{d}s\right\| \leq \\ &\leq \|\Phi(t)\|\|E(t_0)\| + \|\Phi(t)\|\left\|\int_{t_0}^{t}\Phi(s)^{-1}\cdot R(s)\mathrm{d}s\right\| \\ &\leq \|\Phi(t)\|\|E(t_0)\| + \|\Phi(t)\|\sup_{\forall\, t_0\leq s\leq t}\|R(s)\|\int_{t_0}^{t}\left\|\Phi(s)^{-1}\right\|\mathrm{d}s.\end{aligned}$$

It can be proven that in the case in which the matrix $A(t)$ commutes with its primitive[1] $I_A(t) = \int_{t_0}^{t} A(s)\mathrm{d}s$, that is $A\cdot I_A = I_A\cdot A$, the fundamental matrix takes the form:

$$\Phi(t) = \exp\left(\int_{t_0}^{t} A(s)\mathrm{d}s\right) = \exp\left(I_A(t)\right).$$

In the case in which the matrix $I_A(t)$ is normal, that is $I_A I_A^T = I_A^T I_A$, then the exponential matrix norm can be written:

$$\|\exp\left(I_A(t)\right)\| = e^{\alpha(I_A(t))}, \tag{5.1}$$

where $\alpha(I_A(t))$ is the spectral abscissa of $I_A(t)$,

$$\alpha(I_A(t)) = \sup_{z\,\in\Lambda(I_A(t))} \mathrm{Re}(z), \tag{5.2}$$

in which $\Lambda(I_A(t))$ is the spectrum or set of all eigenvalues of $I_A(t)$. Even though this is the most general case that can be considered to study the error, it is not very common that such matrices appear as a result of a spatial discretization. Nevertheless the previous equation is stated as a warning for the general considerations. In practice, the vast majority of the problems that appear on mathematical modelling of physical phenomena satisfy that the differential operator $\boldsymbol{\mathcal{L}}$ does not depend explicitly of t. In these problems the matrix A merges as a constant function of t, which means:

[1]In general, the exponential of a matrix function $\exp(B(t)) = \sum_{n=0}^{\infty} B(t)^n/n!$ does not satisfy $\mathrm{d}(\exp(B))/\mathrm{d}t = \mathrm{d}B/\mathrm{d}t\,\exp(B)$ as the matricial product is not commutative. For example: $\mathrm{d}(B^2)\big/\mathrm{d}t = \mathrm{d}B/\mathrm{d}t\; B + B\;\mathrm{d}B/\mathrm{d}t \neq 2B\;\mathrm{d}B/\mathrm{d}t$ if B and $\mathrm{d}B/\mathrm{d}t$ do not commute.

$$I_A(t) = A(t - t_0).$$

In these problems, the fundamental matrix is takes the form of the exponential[2] of $A(t - t_0)$:

$$\Phi(t) = \exp\left(A(t - t_0)\right).$$

Therefore, the solution for $E(t)$ results as [3]:

$$E(t) = \exp\left(A(t - t_0)\right) \cdot E(t_0) + \int_{t_0}^{t} \exp\left(A(t - s)\right) \cdot R(s) \mathrm{d}s. \tag{5.3}$$

And the norm of it is upperly bounded by:

$$\begin{aligned}
\|E\| &= \left\| \exp\left(A(t - t_0)\right) \cdot E(t_0) + \int_{t_0}^{t} \exp\left(A(t - s)\right) \cdot R(s) \mathrm{d}s \right\| \leq \\
&\leq \|\exp\left(A(t - t_0)\right)\| \|E(t_0)\| + \left\| \int_{t_0}^{t} \exp\left(A(t - s)\right) \cdot R(s) \mathrm{d}s \right\| \\
&\leq \|\exp\left(A(t - t_0)\right)\| \|E(t_0)\| + \sup_{\forall\, t_0 \leq s \leq t} \|R(s)\| \int_{t_0}^{t} \|\exp\left(A(t - t_0)\right)\| \mathrm{d}s.
\end{aligned}$$

Supposing that A is normal, then

$$\|\exp\left(A(t - t_0)\right)\| = e^{\alpha(A)(t - t_0)},$$

and

$$\int_{t_0}^{t} \|\exp\left(A(t - s)\right)\| \mathrm{d}s = \int_{t_0}^{t} e^{\alpha(A)(t-s)} \mathrm{d}s = \frac{1}{|\alpha|} \left[e^{\alpha(t - t_0)} - 1 \right]. \tag{5.4}$$

Hence, the bound for E is written:

$$\|E\| \leq \|E(t_0)\| e^{\alpha(t - t_0)} + \frac{1}{|\alpha|} \left[e^{\alpha(t - t_0)} - 1 \right] \sup_{t_0 \leq s \leq t} \|R(s)\|. \tag{5.5}$$

[2]Note that a matrix A always commutes with itself.

[3]Whenever A and B commute, $\exp(A + B) = \exp(A)\exp(B)$ and $\exp\left(A^{-1}\right) = \exp(-A)$

From equation (5.3) several conclusions about the error behaviour can be extracted for the different values that can have α. Whenever $\alpha < 0$ the spatial error is bounded by $(1/\alpha)[e^{\alpha(t-t_0)} - 1] \sup_{t_0 \leq t \leq t_f} \|R\|$ and if $\alpha > 0$ the spatial error amplifies indefinitely along time. For this reason, it is crucial to ensure that the spatial discretization preserves the stability nature of the differential problem. Generally, when discretizing hyperbollic problems such as the waves equation (which posseses pure imaginary eigenvalues), by means of centered high order finite differences over equispaced grids, leads to eigenvalues with positive real parts. These schemes or formulas of finite differences are not appropriated for the spatial discretization. This issue is associated to the bad behaviour of high order interpolants nearby the boundaries. Evidence suggests that this problem can be mitigated concentrating points in the surroundings of the boundaries as in the case of Chebyshev grids.

Once studied the spatial error, in order to analyse the temporal error lets consider the constant coefficient system of ordinary equations resultant of the discretization of an initial value boundary problem ($\partial \boldsymbol{\mathcal{L}}/\partial t = \mathbf{0}$) with a scheme of finite differences of order q:

$$\frac{\mathrm{d}U}{\mathrm{d}t} = AU + b, \qquad U = \begin{pmatrix} U_1 \\ \vdots \\ U_{N+1} \end{pmatrix}, \qquad b = \begin{pmatrix} \frac{1}{\Delta x^q} \\ \vdots \\ \frac{1}{\Delta x^q}. \end{pmatrix} \tag{5.6}$$

If an explicit Euler scheme is introduced to approximate the derivative, results:

$$\begin{aligned} U^{n+1} &= U^n + AU\Delta t + b\Delta t \\ &= \underbrace{(I + A\Delta t)}_{B} U^n + b\Delta t, \\ &= BU^n + b\Delta t. \end{aligned} \tag{5.7}$$

From introducing the analytical solution $U(t)$ of 5.6 on the temporal scheme, the truncation error T^{n+1} can be defined as:

$$T^{n+1} = \left(U(t_{n+1}) - U(t_n)\right) - \left(BU(t_n) + b\Delta t\right). \tag{5.8}$$

Adding (5.6) to (5.8) a difference equation for the temporal error merges:

$$\begin{aligned} E^{n+1} &= U(t_{n+1}) - U^{n+1}, \\ &= BE^n + T^{n+1}. \end{aligned} \tag{5.9}$$

The solution of 5.9, is:

$$E^n = B^n E^0 + \sum_{k=0}^{n} B^{n-k} T^k, \tag{5.10}$$

in which the term $B^n E^0$ is the initial condition error and $\sum_{k=0}^{n} B^{n-k} T^k$ is the temporal discretization error.

Hence, an upper bound can be calculated for E^n:

$$\|E^n\| \leq \|B^n\| \|E^0\| + \sup_{0\leq k\leq n} \|T^k\| \sum_{k=0}^{n} \|B^{n-k}\|. \tag{5.11}$$

If B is normal ($BB^T = B^T B$) there exist an orthonormal set of eigenvectors and eigenvalues which diagonalize B and the norm:

$$\|B^n\| = \rho^n, \qquad \text{with} \quad \rho = \sup_{z \,\in\, \Lambda(B)} z, \tag{5.12}$$

where ρ is the spectral radius and $\Lambda(B)$ is the set of all eigenvalues of B or spectra. Using the definition of this norm:

$$\|B^n\| = \rho^n, \qquad \sum_{k=0}^{n} \|B^{n-k}\| = \sum_{k=0}^{n} \rho^{n-k}. \tag{5.13}$$

And the upper bound for temporal error can be written [4]:

$$\begin{aligned} \|E^n\| &\leq \rho^n \|E^0\| + \sup_{0\leq k\leq n} \|T^k\| \sum_{k=0}^{n} \rho^{n-k}, \\ &\leq \rho^n \|E^0\| + \sup_{0\leq k\leq n} \|T^k\| \frac{(1-\rho^{n+1})}{(1-\rho)}. \end{aligned} \tag{5.14}$$

From 5.14 it can be inferred that if $\rho < 1$ the error on the initial condition annuls when $t \to \infty$ and the temporal error is bounded by the error of the temporal discretization

[4]Taking in account the solution of the geometric sum, given by $S_n = \sum_{k=0}^{n} \rho^{n-k} = 1+\rho+\cdots+\rho^n$ and $\rho S_n = \sum_{k=0}^{n} \rho^{n-k+1} = \rho + \rho^2 + \cdots + \rho^{n+1} = S_n - 1 + \rho^{n+1} \Rightarrow S_n = (1-\rho^{n+1})/(1-\rho)$.

$$\|E^n\| \leq \sup_{0\leq k\leq n} \left\|T^k\right\| \frac{1}{(1-\rho)}. \tag{5.15}$$

In the case in which $\rho > 1$ any error on the initial conditions amplifies as a power of grade n when $t \to \infty$, masking any truncation error.

The interrogant that now arises is to determine under which circumstances $\rho < 1$. In order to analyse the eigenvalues of B in terms of the eigenvalues of A and therefore calculating the spectral radius it is used the spectral transformation theorem.

A s steps linear temporal scheme G can be written:

$$\begin{aligned} G(U^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t)\, \Delta t &= \sum_{j=0}^{s} \alpha_j U^{n+1-j} \\ &= \Delta t \sum_{j=0}^{s} \beta_j F(U^{n+1-j}; t_{n+1-j}) \\ &= \sum_{j=0}^{s} \beta_j \Delta t A U^{n+1-j}, \end{aligned} \tag{5.16}$$

which means that

$$\sum_{j=0}^{s} \left(\alpha_j I - \beta_j \Delta t A\right) U^{n+1-j} = 0, \tag{5.17}$$

Definition 1. *The stability characteristic polynomial of a linear s steps temporal scheme is defined as:*

$$\pi(r, w) = \sum_{j=0}^{s} (\alpha_j - w\beta_j) r^{s-j}. \tag{5.18}$$

Definition 2. *The stability characteristic polynomial of a linear 1 step temporal scheme is defined as:*

$$\pi(r, w) = r - kw. \tag{5.19}$$

The spectral decomposition theorem stablishes that if $\lambda\Delta t \in \Lambda(\Delta t A)$, then each root of the stability characteristic polynomial with $w = \lambda\Delta t$ belongs to the spectra of B.

Definition 3. *The absolute stability region $\mathcal{R}_A$ of a numerical scheme is the set of number $w \in \mathbb{C}$ for which all roots of the stability characteristic polynomial satisfy that $|w| \leq 1$ and those roots which verify $|w| = 1$ are simple.*

Therefore, if $\lambda \Delta t \in \mathcal{R}_A$, then the eigenvalue $r \in \Lambda(B)$ verifies $|r| \leq 1$ and hence, the temporal error is bounded.

To summarize the general idea of this section, on the figure 5.3 is represented the errors which are produced during the numerical resolution of an initial value boundary problem.

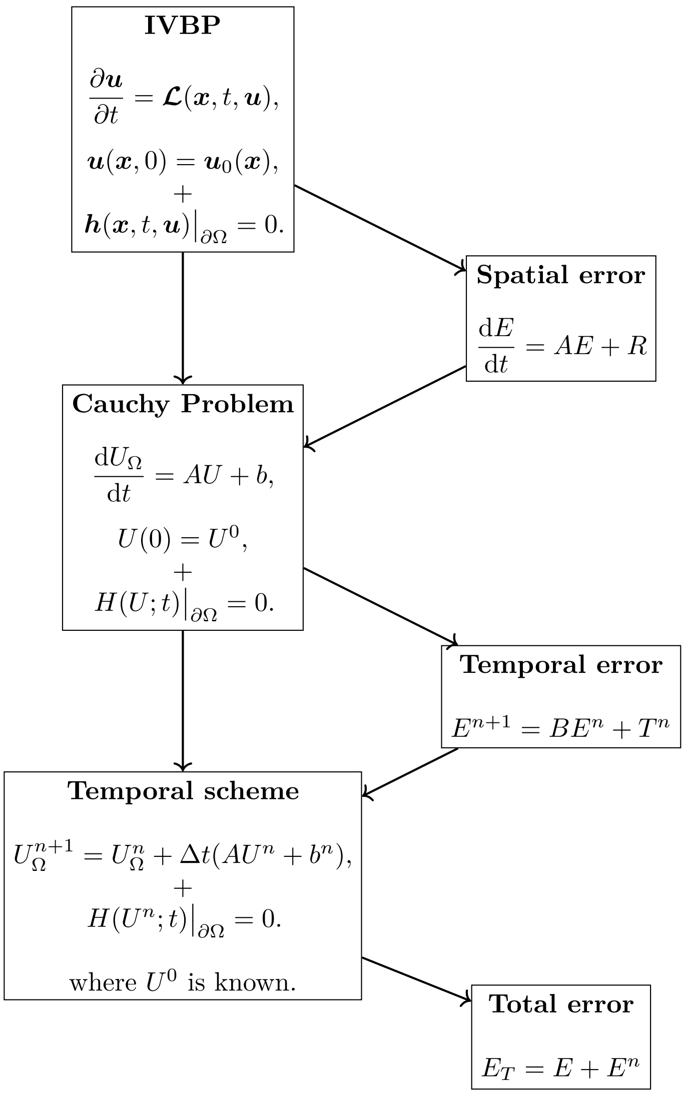

IVBP
$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}(\boldsymbol{x}, t, \boldsymbol{u}),$
$\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}),$
$+$
$\boldsymbol{h}(\boldsymbol{x}, t, \boldsymbol{u})\big|_{\partial\Omega} = 0.$
Spatial error
$\frac{\mathrm{d}E}{\mathrm{d}t} = AE + R$
Cauchy Problem
$\frac{\mathrm{d}U_\Omega}{\mathrm{d}t} = AU + b,$
$U(0) = U^0,$
$+$
$H(U;t)\big|_{\partial\Omega} = 0.$
Temporal error
$E^{n+1} = BE^n + T^n$
Temporal scheme
$U_\Omega^{n+1} = U_\Omega^n + \Delta t(AU^n + b^n),$
$+$
$H(U^n;t)\big|_{\partial\Omega} = 0.$
where U^0 is known.
Total error
$E_T = E + E^n$

Figure 5.3: Errors during the numerical resolution of initial value boundary problems.

Chapter 6

Mixed Boundary and Initial Value Problems

6.1 Overview

In the present chapter, the numerical resolution and implementation of initial value boundary problem for an unknown variable $\boldsymbol{u}$ coupled with an elliptic problem for another unknown variable $\boldsymbol{v}$. Prior to the numerical resolution of the problem by means of an algorithm a brief mathematical presentation must be given. Hence, 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, how this algorithm is implemented and the code organized in Fortran language is explained.

Thus, it is expected for the reader to comprehend how this generic problems are solved and conceptually understand how it is possible to take this to practice using Fortran language, although the concepts can be implemented in any desired language, however, probably not in such an intuitively manner.

6.2 Mixed Problems

Let us be $\Omega \subset \mathbb{R}^p$ 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 $t \in \mathbb{R}$.

The intention of this section is to state and solve a temporal evolution problem for two vectorial 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, such as:

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

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, $\boldsymbol{h}_u$ 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 as the differential operators are defined for 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 vectorial 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.

If the spatial domain D is discretized in N_D points, both problems extend from vectorial to tensorial, as a tensorial system of equations appears from each variable of $\boldsymbol{u}$ and $\boldsymbol{v}$. The order of the tensorial system for $\boldsymbol{u}$ and $\boldsymbol{v}$ are respectively $p \times N_u$ and $p \times N_v$. Therefore the number of elements for both of them are: $N_{e,u} = p \times N_u \times N_D$ and $N_{e,v} = p \times N_v \times N_D$. 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 tensorial system evaluated on the boundary points are $N_{C,u} = p \times N_u \times N_{\partial\Omega}$ and $N_{C,v} = p \times N_v \times N_{\partial\Omega}$.

Once the spatial discretization is done, the initial value boundary problem and the boundary value problem transform. The differential operator for $\boldsymbol{u}$ emerges as a tensorial 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 tensorial 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:

$$\begin{aligned}
&\frac{dU_\Omega}{dt} = F_U(U,V;t), && G(U;t)\big|_{\partial\Omega} = 0,\\
&U(t_0) = U^0, && \\
&F_V(U,V;t) = 0, && H(V;t)\big|_{\partial\Omega} = 0,
\end{aligned}$$

where $U \in \mathbb{R}^{N_{e,u}}$ and $V \in \mathbb{R}^{N_{e,u}}$ are the solutions in all domain, $U_\Omega \in \mathbb{R}^{N_{e,u}-N_{C,u}}$ is the solution on grid 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 on the boundary points of the grid, $U^0 \in \mathbb{R}^{N_{e,u}}$ is the discretized initial value,

$$\begin{aligned}
F_U &: \mathbb{R}^{N_{e,u}} \times \mathbb{R}^{N_{e,v}} \times \mathbb{R} \to \mathbb{R}^{N_{e,u}-N_{C,u}},\\
F_V &: \mathbb{R}^{N_{e,v}} \times \mathbb{R}^{N_{e,u}} \times \mathbb{R} \to \mathbb{R}^{N_{e,v}-N_{C,v}},
\end{aligned}$$

are the difference operators associated to both differential operators and

$$\begin{aligned}
H_U &: \mathbb{R}^{N_{e,u}} \times \mathbb{R} \to \mathbb{R}^{N_{C,u}},\\
H_V &: \mathbb{R}^{N_{e,v}} \times \mathbb{R} \to \mathbb{R}^{N_{C,v}},
\end{aligned}$$

are the difference operators of the boundary conditions.

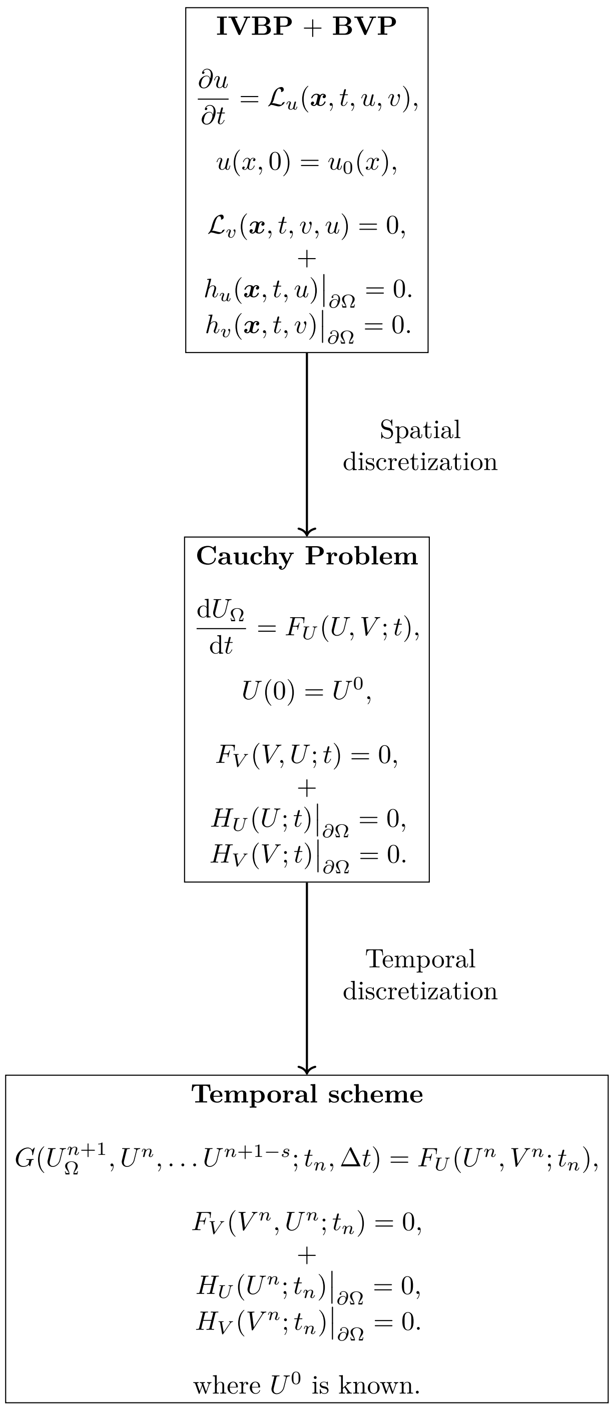
Hence, the resolution of the problem requires solving a Cauchy problem and three systems of equations for the discretized variables U and V. To solve the Cauchy Problem, the time is discretized in $t = t_n \boldsymbol{e}_n$. The term $n \in \mathbb{Z}$ is the index of every temporal step that runs over $[0, N_t]$, where N_t is the number of temporal steps. Later will be seen that the algorithm can be resumed in five recursive steps that repeat 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: $U_\Omega(t_n) = U_\Omega^n$, $U(t_n) = U^n$ and $V(t_n) = V^n$. The Cauchy Problem transforms into a difference equation system introducing a s-steps temporal scheme:

$$\begin{aligned}
G(U_\Omega^{n+1}, \underbrace{U^n, \dots U^{n+1-s}}_{s\ steps}; t_n, \Delta t) &= F_U(U^n, V^n; t_n),\\
U(t_0) = U^0, \qquad G(U^n;t_n)\big|_{\partial\Omega} &= 0,\\
F_V(U^n, V^n; t_n) = 0, \qquad H(V^n;t_n)\big|_{\partial\Omega} &= 0,
\end{aligned}$$

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.



IVBP + BVP

$$\frac{\partial u}{\partial t} = \mathcal{L}_u(\boldsymbol{x}, t, u, v),$$

$$u(x,0) = u_0(x),$$

$$\mathcal{L}_v(\boldsymbol{x}, t, v, u) = 0,$$

$$+$$

$$h_u(\boldsymbol{x}, t, u)\big|_{\partial\Omega} = 0.$$

$$h_v(\boldsymbol{x}, t, v)\big|_{\partial\Omega} = 0.$$

Cauchy Problem

$$\frac{\mathrm{d}U_\Omega}{\mathrm{d}t} = F_U(U, V; t),$$

$$U(0) = U^0,$$

$$F_V(V, U; t) = 0,$$

$$+$$

$$H_U(U; t)\big|_{\partial\Omega} = 0,$$

$$H_V(V; t)\big|_{\partial\Omega} = 0.$$

Temporal scheme

$$G(U_\Omega^{n+1}, U^n, \dots U^{n+1-s}; t_n, \Delta t) = F_U(U^n, V^n; t_n),$$

$$F_V(V^n, U^n; t_n) = 0,$$

$$+$$

$$H_U(U^n; t_n)\big|_{\partial\Omega} = 0,$$

$$H_V(V^n; t_n)\big|_{\partial\Omega} = 0.$$

where U^0 is known.

Figure 6.1: Method of lines for mixed initial and boundary value problems.

6.2.1 Algorithm.

The aim of the algorithm, is to solve numerically evolution problems for two functions $\boldsymbol{u} : D \times \mathbb{R} \to \mathbb{R}^{N_u}$ and $\boldsymbol{v} : D \times \mathbb{R} \to \mathbb{R}^{N_v}$, such as:

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

When the spatial domain D is discretized the problem results:

$$\begin{aligned}
&\frac{dU_\Omega}{dt} = F_U(U, V; t), && G(U; t)\big|_{\partial\Omega} = 0,\\
&U(t_0) = U^0, &&\\
&F_V(V, U; t) = 0, && H(V; t)\big|_{\partial\Omega} = 0.
\end{aligned}$$

Finally, to start with the algorithm the time is discretized and the derivative approximated using a s-steps temporal scheme:

$$G(U_\Omega^{n+1}, \underbrace{U^n, \ldots U^{n+1-s}}_{s\ steps}; t_n, \Delta t) = F_U(U^n, V^n; t_n),$$

$$\begin{aligned}
U(t_0) = U^0, &\qquad H_U(U^n; t_n)\big|_{\partial\Omega} = 0,\\
F_V(V^n, U^n; t_n) = 0, &\qquad H_V(V^n; t_n)\big|_{\partial\Omega} = 0.
\end{aligned}$$

The algorithm must solve this systems of equations. It will be done in a similar manner as in the section 5, but with the addition of an intermediate Boundary Value Problem for V^n, which will be solved using the algorithm described in 4. The main idea of the algorithm is: starting from the initial value U^0, the initial value V^0 is calculated; using both values, the difference operator F_U at that instant is constructed; with the difference operator, the temporal scheme is solved, giving back the solution at inner points at the next temporal step U_Ω^1 for which the boundary conditions must be imposed to obtain the solution U^1. This solution will be used as the initial value to solve the next temporal step, closing the algorithm. Hence, the algorithm consists on several steps that are carried out recursively. Particularly, the algorithm can be resumed in five steps for a generic temporal step, t_n in which the solutions are (U^n, V^n):

1. **Boundary points from inner points for U^n.**

In first place, the known initial value at the inner points, U^n_Ω, is used to calculate the solution for the boundary conditions. That is, solving the system of equations:

$$H_U(U^n;t_n)\big|_{\partial\Omega} = 0,$$

which gives back the value of the solution at boundaries $U^n\big|_{\partial\Omega}$.

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.

2. **Difference operator for V^n.**

Once the value in all domain U^n is calculated, calculating its derivatives it is possible to construct the difference operator $F_V(V^n, U^n; t_n)$. 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.

3. **Boundary Value Problem for V^n.**

The previous difference operator is used along with the boundary conditions operator $H(V^n;t_n)\big|_{\partial\Omega}$, to solve the boundary value problem for V^n defined as:

$$F_V(V^n, U^n; t_n) = 0,$$

$$H_V(V^n;t_n)\big|_{\partial\Omega} = 0.$$

It can be seen that as U^n and t_n are known, the resolution of this problem can be done by similarly to the one detailed in the section 4. However, for this to be possible first 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 restricting them by introducing U^n and t_n as 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 in the section 4.

In order to avoid redundancies, the explanation of this step will not be included in this section as it was explained previously. By the end of this step, both solutions U^n and V^n are known.

4. **Difference operator for U^n.**

With the values of U^n and V^n it is possible to calculate their derivatives by finite differences. Once calculated, it is possible to construct the difference operator for U^n, $F_U(U^n, V^n; t_n)$. Notice that the values of F_U are known for all points as U^n and V^n are known, unlike in step 2, in which the value of V^n was unknown.

5. **Temporal step for U^n.**

Finally, the difference operator previously calculated, F_U, is used to calculate the solution at inner points at the next temporal step U_Ω^{n+1}. 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, closing the loop of the algorithm. The philosophy for other temporal schemes is the same, the result is the solution at the next temporal step.

The sequence of the algorithm is represented on figure 6.2, with the inputs and outputs of each step.

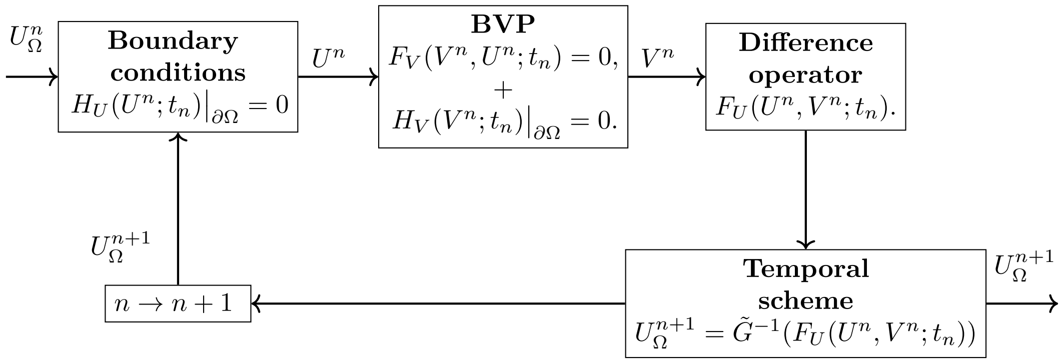


Figure 6.2: Algorithm for mixed initial and boundary value problems.

6.2.2 Algorithm implementation.

The aim of the following pages is to give a description of how it is implemented the algorithm step by step. For this, the structure will be the same as in the algorithm explanation.

Besides, the code will be based on the available codes exposed in the sections 4 and 5, re-using them as much as possible. The explanation will treat the case of a two dimensional problem that evolves along time.

Before starting with the steps of the algorithm it is necessary a presentation of a new abstract interface defined to implement the differential operators that mix both $\boldsymbol{u}$ and $\boldsymbol{v}$. That is, for problems that use differential operators such as:

$$\frac{\partial \boldsymbol{u}}{\partial t}(\boldsymbol{x},t)=\boldsymbol{\mathcal{L}}_u(\boldsymbol{x},t,\boldsymbol{u}(\boldsymbol{x},t),\boldsymbol{v}(\boldsymbol{x},t)),$$

The interface is defined as follows:

```fortran
function DifferentialOperator2D_system_mixed(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 :: DifferentialOperator2D_system_mixed (size(u))
end function
```

Listing 6.1: `IVBP_and_BVP.f90`

As it was mentioned before, it can be seen that the order of the input arguments is relevant, as the size of the differential operator is the same as the vectorial input u.

Defined in such manner, the differential operator recognises the value and derivatives of both u and v as needed.

This said, the implementation can be explained following the steps of the algorithm.

1. **Boundary points from inner points for U^n.**

This step requires solving the system of equations $G(U^n; t_n)\big|_{\partial\Omega} = 0$, that in general is not linear. This step is essentially the same as the equivalent one from the section 5. The subroutine that will solve the system will be the same as the one used in the mentioned section. Which means that the restriction imposed to this function shall be the same as the one explained there and its result is the same function G_r. In order to avoid redundancies the detailed explanation of this restriction will be omitted and the reader can consult it in the subsection Algorithm implementation from the section 5.

Nevertheless, the steps in order of execution will be exposed.

First, the initial approximation for the iterative method is assigned, extracted from the initial value at the inner points U^n_Ω.

```
    Uc = [ U1, U2, U3, U4 ]
```

Listing 6.2: `IVBP_and_BVP.f90`

In this call, it can be seen a breach on the TKR compatibility (Type, Kind, Rank) in the call of the subroutine. This breach in the rank compatibility is permitted as the vectors `U1`, `U2`, `U3` and `U4` are defined in explicit shape form.

Secondly, the values at the boundaries are calculated over `Uc` using the Newton subroutine. The inputs are `Uc` as initial approximation and `BCs` as the function.

```
        call Newton( BCs, Uc )
```

Listing 6.3: `IVBP_and_BVP.f90`

The function `BCs` results on a vector `G` that yields the values of the restricted difference operator for the boundary points G_r. For this, first uses the subroutine `Asign_BVs` to define the boundary points of an array Ut at each temporal step as the input of `BCs` starting from the initial value. The array `Ut`, defined above the contains command, contains the value of the final solution of the problem:

$$U = U^n_{ijk}(\boldsymbol{e}_n \otimes \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k),$$

where the indices $(n, i, j, k) \in \mathbb{Z}^3$ go over the interval $[0, N_t] \times [0, N_x] \times [0, N_y] \times [1, N_u]$.

Thus, the function BCs is implemented as:

```
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(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),Y )

! ** Calculate boundary conditions G
     call Asign_BCs(  G1,  G2, G3,  G4 )

     G = [ G1, G2, G3, G4 ]

end function
```

Listing 6.4: IVBP_and_BVP.f90

The next step of the function BCs is constructing the result vector G_r. For this, it is used the subroutine Asign_BCs.

This subroutine stores the values of the boundary conditions (four edges) in four vectors G1, G2, G3 and G4 which compound G. These are constructed calculating the solution derivatives at boundaries from the initial value. These derivatives are calculated using an available subroutine. Its interface requires the solution U^n to be expressed as a third order tensor to be consistent with the computational cell of the grid. This tensor can be expressed for a two dimensional problem as:

$$U^n = U^n_{ijk}(\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k),$$

where the indices $(i, j, k) \in \mathbb{Z}^3$ go over the interval $[0, N_x] \times [0, N_y] \times [1, N_u]$. The subroutine can calculate the derivatives if k is fixed. From this expression it can be inferred the relation:

$$U = \boldsymbol{e}_n \otimes U^n$$

To store in the array Ut the solution at each instant of time it is used the subroutine Binary_search in the same manner as in the section 5.

Hence, when the subroutine is called for t_BC, the discrete time domain Time_Domain and it, t_BC stores the value of time at the temporal step it. The idea is use this index to implement the solution U in Fortran as:

```
       Ut(it,0:Nx,0:Ny,1:Nu)
```

For this to be possible, the integer `it` must have the same value in all of the subroutines and functions involved in the algorithm at each temporal step. That means that `it`, `t_BC` and `Time_Domain` have to be defined above the contains command.

The call for the subroutine `Binary_search`, must be done below the contains command to store at every temporal step the value of the time `t_BC`, before the Newton subroutine is called.

```
        call Binary_search(t_BC, Time_Domain, it)
```

Listing 6.5: `IVBP_and_BVP.f90`

Then, using the interface for boundary conditions `Boundary_conditions` written as:

```
function BC2D_system(x, y, t, u, ux, uy)
      real, intent(in) :: x, y, t, u(:), ux(:), uy(:)
      real :: BC2D_system(size(u)) ! maximum number of BCs at each point
end function
```

Listing 6.6: `IVBP_and_BVP.f90`

the vectors are defined. These interfaces must be stated above the contains command as procedures to be possible its call.

This said, it can be understood the functioning of the previously mentioned subroutine, `Asign_BCs`, which is implemented as follows:

```fortran
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,:) = Boundary_conditions_u ( x_nodes(0), y_nodes(j), t_BC,   &
                               Ut(it, 0, j, : ), Wx(0, j,:), Wy(0, j,:))

     G2(j,:) = Boundary_conditions_u (x_nodes(Nx), y_nodes(j), t_BC,   &
                          Ut(it, Nx, j, : ), Wx(Nx, j, :), Wy(Nx, j, :))
   end do

   do i = 1, Nx-1
     G3(i,:) = Boundary_conditions_u ( x_nodes(i), y_nodes(0), t_BC,   &
                                Ut(it, i, 0,:), Wx(i, 0, :), Wy(i, 0,:))

     G4(i,:) = Boundary_conditions_u ( x_nodes(i), y_nodes(Ny), t_BC,  &
                          Ut(it, i, Ny, : ), Wx(i, Ny, :), Wy(i, Ny, :))

   end do

end subroutine
```

Listing 6.7: `IVBP_and_BVP.f90`

As `Ut` varies with time, the values of the boundary conditions adapt to the pass of time, which in the future will permit to construct the difference operator for each instant of time, $F_U(U^n, V^n; t_n)$, over a single third order tensor.

Thus, the function `BCs` gives back the restricted difference operator G_r to be introduced in the Newton subroutine.

At this point, the vector `Uc` yields the values of the solution at the boundary points calculated by the Newton subroutine.

Finally, the values of the boundary points $U^n\big|_{\partial\Omega}$ are assigned to the solution at that temporal step U^n using the subroutine `Asign_BVs`.

```fortran
!   *** asign boundary points Uc  to U
        call Asign_BVs( 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 )
```

Listing 6.8: `IVBP_and_BVP.f90`

Thus, the boundary conditions are imposed to the solution U^n from the initial value U^n_Ω.

2. **Difference operator for V^n.**

In this step the goal is to construct the differential operator $F_V(V^n, U^n; t_n)$, in which the known solution U^n and the time t_n act as a mere parameter. First of all, the derivatives of U^n must be calculated. For this task, it is used the available subroutine that calculates derivatives using finite differences. This is implemented as:

```fortran
!  *** 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
```

Listing 6.9: `IVBP_and_BVP.f90`

The derivatives of U^n are stored on arrays defined over the contains command. The location of their definition, will permit to re-use them in all the subroutines and functions involved. The idea is to re-use the subroutine Non Linear Boundary Value Problems presented in the section Boundary Value Problem. To fit its interface it is necessary to restrict the operator and transform it into $F_{V,R}$. The philosophy is the same as all the restrictions in functions done in Fortran.

A function is defined for this purpose:

```
function Differential_operator_v_R(x, y, V, Vx, Vy, Vxx, Vyy, Vxy) result (Fv)
     real, intent(in) :: x, y, V(:), Vx(:), Vy(:), Vxx(:), Vyy(:), Vxy(:)
     real :: Fv(size(V))

     integer :: ix, iy

     call Binary_search(x, x_nodes, ix)
     call Binary_search(y, y_nodes, iy)

     Fv = Differential_operator_v( x, y, 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 6.10: `IVBP_and_BVP.f90`

The interface of this function fits the requirements of the subroutine that solves boundary value problems. First, the values of the difference operator F_V are stored on `F_v` which is also the result of the function. When `F_v` is given back, it holds the difference operator $F_{V,R}$. The arrays containing the derivatives of U^n are defined over the contains command and therefore, the use of `Binary_search` is needed for each spatial dimension. The integer variables which keep track of each spatial point are `ix` and `iy`. Its use will relate the values of the arrays that contain the derivatives of `Ut` with the differential operator.

This function is ready to be introduced in the subroutine that calculates the value of V^n as the differential operator.

3. **Boundary Value Problem for V^n.**

In this step the objective is to obtain the value V^n as the solution of the problem:

$$F_{V,R}(V^n) = 0,$$

$$H_R(V^n)\big|_{\partial\Omega} = 0.$$

In first place, to fit the interface of the subroutine that solves this problem, an additional restriction to H must be done, as the temporal step t_n is only involved as a parameter, transforming it into H_R. For this, the procedure is the same as before, defining a function:

```
function Boundary_conditions_v_R(x, y, V, Vx, Vy) result (G_v)
     real, intent(in) :: x, y, V(:), Vx(:), Vy(:)
     real :: G_v(size(V))

    G_v = Boundary_conditions_v (x, y, t_BC, V, Vx, Vy )


end function
```

Listing 6.11: `IVBP_and_BVP.f90`

The interface of this function is suitable for the subroutine that will solve the boundary conditions.

Once this is done, the resolution of the boundary value problem for $\boldsymbol{v}$ is achieved with a simple call:

```
       call Boundary_Value_Problem(                                  &
                x_nodes = x_nodes,                                   &
                y_nodes = y_nodes,                                   &
                Order = Order, N_variables = Nv,                     &
                Differential_operator = Differential_operator_v_R,   &
                Boundary_conditions = Boundary_conditions_v_R,       &
                Solution = Vt(it, 0:, 0:, :)     )
```

Listing 6.12: `IVBP_and_BVP.f90`

This subroutine gives back the solution V^n stored over the array `Vt`, defined above the contains command. The description of its functioning can be consulted in the section 4 from this part.

4. **Difference operator for U^n.**

Whenever both U^n and V^n are known, and stored respectively on the arrays Ut and Vt, the difference operator F_U can be constructed. The derivatives of Ut are already calculated and known as they are defined over the contains command. Hence, the derivatives of V^n must be calculated from the array Vt. As before, it is used the available subroutine to perform the calculation:

```
!  *** 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
```

Listing 6.13: IVBP_and_BVP.f90

Once all the derivatives involved are known, the differential operator can be stored over an array F_u in the following manner:

```
!  *** Differential operator L_u(U,V) at inner grid points

 do i = 1, Nx - 1
   do j = 1, Ny - 1

   F_u(i,j,:) = Differential_operator_u( x_nodes(i), y_nodes(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
```

Listing 6.14: IVBP_and_BVP.f90

Thus, the difference operator calculated will have the shape:

$$F_U^n = F_{U,ijk}^n (\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k).$$

This differential operator will be used to obtain the solution at the next temporal step.

5. **Temporal step.**

The difference operator now has to be used to calculate the solution at the next temporal step U^{n+1}. For this task there is a subroutine that solves Cauchy Problems systems. The interface of this subroutine requires the difference operator to be ordered as before. With it we intend to construct the solution at all time as a fourth order tensor:

$$U = \boldsymbol{e}_n \otimes U^n = U^n_{ijk}(\boldsymbol{e}_n \otimes \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$$

, where $n \in \mathbb{Z}$ goes over the interval $[0, N_t]$ and N_t is the number of temporal steps.

However, the interface of the Cauchy Problems tool takes an array in which yield the solution at all time in the form:

$$U = U^n_r(\boldsymbol{e}_n \otimes \boldsymbol{e}_r),$$

where $r \in \mathbb{Z}$ goes over the interval $[1, M]$. Hence, to introduce the solution it must be in this form for $M = (N_x + 1)(N_y + 1)N_u$.

Thus, introducing the solution into the subroutine at a certain time step U^n in the form:

$$U^n = U^n_r \boldsymbol{e}_r,$$

gives back its value at the next temporal step

$$U^{n+1} = U^{n+1}_r \boldsymbol{e}_r.$$

To construct the fourth order tensor solution it is necessary a rearrangement or linking between it and the solution expressed as a second order tensor.

In order to link both tensors without waste of memory is defined in Fortran a pointer data structure:

$$P = P^n_r(\boldsymbol{e}_n \otimes \boldsymbol{e}_r),$$

which targets the solution: $U = U^n_{ijk}(\boldsymbol{e}_n \otimes \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$. Doing this, links the values of both arrays in the Fortran environment as they share the memory spaces in which their components are yielded. Mathematically, this means that a bijective application between P and U is defined in such a manner that assigns to each component of P a component of U.

This is done in the same manner as in the section 5, using the subroutine `Data_Pointer`. This subroutine reads two integer values `N1,N2` and an array `U(N1,N2)` and assigns a pointer array `pU(N1,N2)` that targets every element on the array `U`.

```
subroutine Data_pointer( N1, N2, U, pU )
     integer, intent(in) :: N1, N2
     real, target, intent(in) :: U(N1, N2)
     real, pointer, intent(out) :: pU(:,:)

     pU => U

end subroutine
```

Listing 6.15: `IVBP_and_BVP.f90`

This subroutine will be used to target the solution U with a pointer P. Observing the interface of `Data_pointer` it can be seen that both pointer and target must be second order tensors. However U is a fourth order tensor, and the assignation seems to be impossible. To solve this issue, a breach on Type, Kind, Rank compatibility will be used.

Fortran language permits a breach on the Rank compatibility for variables defined in explicit shape. In the subroutine `Data_Pointer` the input variable `U(N1,N2)` is defined with explicit shape. The TKR rule breach permits the subroutine to treat the target as a data structure without taking in account its defined rank:

```
  call Data_pointer( N1 = Nt+1, N2 = Nu* (Nx+1) * (Ny+1),              &
                     U = Ut(0:Nt, 0:Nx, 0:Ny, 1:Nu) , pU = pUt )
```

Listing 6.16: `IVBP_and_BVP.f90`

In this call, `Ut`, which yields U, is targeted by the pointer data structure `pUt` that yields P. This call is equivalent to say that there is a bijective application A that relates every element of

$$U = U^n_{ijk}(\boldsymbol{e}_n \otimes \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k),$$

with each component of

$$P = P^n_r(\boldsymbol{e}_n \otimes \boldsymbol{e}_r)$$

, by the identity function. The ordenation of the elements in P is done by the subroutine and does not influence on the resolution.

When stored in such manner, the pointer is in the rank needed for the subroutine `Cauchy_Problems`, permitting us to re-use it easily. Hence, the subroutine can be called as follows:

```
  call Cauchy_ProblemS( Time_Domain = Time_Domain, &
         Differential_operator = Space_discretization, Solution = pUt )
```

Listing 6.17: `IVBP_and_BVP.f90`

The inputs are: a vector for the time `Time_Domain` that yields the values t_n, the pointer data structure `pUt` which yields the solution P and a function `Space_Discretization` whose result is the difference operator F^n_U in vectorial form. It constructs recursively for each t_n the solution P using the differential operator at that time F^n_U. The number of iterations equals to the number of temporal steps N_t.

This function is constructed only to fit $F^n_U = F^n_{U,ijk}(\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$ to the interface of the subroutine `Cauchy_ProblemS`. It rearranges the elements of U^n and F^n_U defining them in assume shape form as vectors and calls the subroutine that constructs the difference operator as a third order tensor.

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

         call Space_discretization_2D_system( U, t, F )

         if (t>0) then
         end if


end function
```

Listing 6.18: `IVBP_and_BVP.f90`

The function calls the subroutine `Space_Discretization_2D_system` using the time `t` and the vectorial argument `U` as input and the vector `F` as output. As it will seen later, this call entails a breach on TKR compatibility, permitted as the arguments in the subroutine are defined explicitly.

The subroutine in charge of constructing the difference operator $F_U^n = F_{U,ijk}^n(\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$ is Space_Discretization_2D_system. The inputs for it are the solution at the temporal step $U^n = U_{ijk}^n(\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$, the time t_n and the difference operator F_U^n.

```fortran
subroutine Space_discretization_2D_system( U, t, F_u )
          real :: U(0:Nx,0:Ny, Nu),  t
          real :: 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)
    real ::  Uc(M1 + M2 + M3 + M4)

        t_BC = t

        call Binary_search(t_BC, Time_Domain, it)
        write(*,*) " it = ", 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 )

 !  *** Boundary points Uc from inner points U
        call Newton( BCs, Uc )
        F_u=0

!   *** asign boundary points Uc  to U
        call Asign_BVs( 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 )

!  *** 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

       call Boundary_Value_Problem(                                   &
                x_nodes = x_nodes,                                    &
                y_nodes = y_nodes,                                    &
                Order = Order, N_variables = Nv,                      &
                Differential_operator = Differential_operator_v_R,    &
                Boundary_conditions = Boundary_conditions_v_R,        &
                Solution = Vt(it, 0:, 0:, :)    )
```
Listing 6.19: IVBP_and_BVP.f90

```
!  *** 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

!  *** Differential operator L_u(U,V) at inner grid points

 do i = 1, Nx - 1
   do j = 1, Ny - 1

   F_u(i,j,:) = Differential_operator_u( x_nodes(i), y_nodes(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 6.20: `IVBP_and_BVP.f90`

As was mentioned before, the inputs U^n and F^n are third order tensors, and when called in the function `Space_Discretization` they were treated as vectors. This is possible as they are defined in explicit shape form and a breach in rank compatibility is permitted.

This subroutine includes the steps from 1 to 3 of the algorithm, as they are necessary to construct the difference operator.

To sum up, calling the subroutine `Cauchy_ProblemS` at each temporal step gives us the solution at the next temporal step $P^{n+1} = P_r^{n+1}\boldsymbol{e}_r$ and therefore gives also $U^{n+1} = U_{ijk}^{n+1}(\boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k)$. However, only the inner points of these tensors have the definitive values. The values of the boundary points have to be calculated introducing U^{n+1} as the new initial value for the algorithm, closing the loop that conforms it.

6.2.3 Code structure.

In order to carry out the algorithm implementation, three subroutines inside the module `IVBP_and_BVP` have been defined. The first two are the tools previously explained `Data_Pointer` and `Binary_Search`. The third is the subroutine that will perform the five steps of the algorithm using the other two subroutines. This latter subroutine, called `IVBP2D_system_mixed`, is structured in two parts: the interface and the contained functions and subroutines. The contained subroutine have been presented previously in the Algorithm Implementation section. In this pages, how the interface works and uses the contained subroutines will be explained.

The main idea is to contain in the interface the functions and subroutines used to implement the five steps of the algorithm. For this, a sequence on calls of these functions and subroutines is implemented. In this sequence there will be involved variables known in the whole subroutine and local ones of each contained subroutine or function as well.

The interface takes as input arguments: the vectors `Time_Domain`, `x_nodes` and `y_nodes`; the integers `Order`, `N_u` and `N_v`; the functions `Differential_operator_u`, `Differential_operator_v`,
`Boundary_conditions_u`, `Boundary_conditions_v` and `Scheme`; and the arrays `Ut` and `Vt`. The output arguments are the arrays `Ut` and `Vt` which will contain the solutions of the problem. Hence, the complete subroutine acts as a black box in which, introduced the input arguments gives back the numerical solutions U and V stored on the arrays `Ut` and `Vt`.

The interface is implemented as follows:

```
subroutine IVBP2D_system_mixed(Time_Domain, x_nodes, y_nodes,          &
                                     Order,      N_u,      N_v,          &
                                     Differential_operator_u,          &
                                     Differential_operator_v,          &
                                       Boundary_conditions_u,          &
                                       Boundary_conditions_v,          &
                                              Scheme, Ut, Vt           )

   real, intent(in) :: Time_Domain(:)
   real, intent(inout) :: x_nodes(0:), y_nodes(0:)
   integer, intent(in) :: Order, N_u, N_v
   procedure (DifferentialOperator2D_system_mixed) ::                  &
                        Differential_operator_u, Differential_operator_v
   procedure (BC2D_system) :: Boundary_conditions_u, Boundary_conditions_v
   procedure (Temporal_Scheme), optional :: Scheme
   real, intent(out) :: Ut(0:, 0:, 0:, :),  Vt(0:, 0:, 0:, :)

   real, pointer :: pUt(:, :)
   real :: t_BC
   integer ::  it, Nx, Ny, Nt, Nu, Nv, Nt_u, Nx_u, Ny_u, Nu_u, Nv_u
   integer ::  M1, M2, M3, M4

   real, allocatable :: Ux(:,:,:), Uxx(:,:,:), Uxy(:,:,:)
   real, allocatable :: Uy(:,:,:), Uyy(:,:,:)

   Nx = size(x_nodes) - 1 ; Ny = size(y_nodes) - 1
   Nt = size(Time_Domain) - 1
   Nu = N_u   ;   Nv = N_v

   Nt_u = size( Ut(:, 0, 0, 1) ) - 1
   Nx_u = size( Ut(0, :, 0, 1) ) - 1
   Ny_u = size( Ut(0, 0, :, 1) ) - 1
   Nu_u = size( Ut(0, 0, 0, :) )
   Nv_u = size( Vt(0, 0, 0, :) )

   if (Nx /= Nx_u .or. Ny /= Ny_u .or. Nt /= Nt_u .or.                 &
       Nu /= Nu_u .or. Nv /= Nv_u ) then

     write(*,*) " ERROR : Non conformal dimensions" ,                  &
                " Time_Domain, x_nodes, y_nodes, Solution"
     write(*,*) " Nx = ", Nx, " Ny = ", Ny, " Nt = ", Nt,              &
                " Nu = ", Nu, " Nv = ", Nv
     write(*,*) " Nx_u = ", Nx_u, " Ny_u = ", Ny_u, " Nt_u = ", Nt_u,  &
                " Nu_u = ", Nu_u, " Nv_u =", Nv_u
     stop

   end if

   M1 = Nu*(Ny - 1); M2 = Nu*(Ny - 1); M3 = Nu*(Nx - 1); M4 = Nu*(Nx - 1)
```

Listing 6.21: IVBP_and_BVP.f90

```
   allocate( Ux(0:Nx,0:Ny, Nu), Uxx(0:Nx,0:Ny, Nu), Uxy(0:Nx,0:Ny, Nu) )
   allocate( Uy(0:Nx,0:Ny, Nu), Uyy(0:Nx,0:Ny, Nu) )

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

!  call Check_grid( "x", x_nodes, Order, Nx+1 )
!  call Check_grid( "y", y_nodes, Order, Ny+1 )

  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_Domain = Time_Domain, &
        Differential_operator = Space_discretization, Solution = pUt )

   deallocate( Ux, Uxx, Uxy, Uy, Uyy  )

contains
```

Listing 6.22: `IVBP_and_BVP.f90`

It can be seen that the solution variables `Ut`, `Vt`, `pUt` and the derivatives of `Ut` at each instant of time are necessarily defined at this level. Also, as the variables `Time_Domain`, `t_BC` and `it` are defined in the interface, its value is the same in all the functions and subroutines involved in the algorithm. Which permits the relation between `Ut` and the third order tensor that yields the solution at each temporal step U^n, as explained in the Algorithm Implementation section. The integers that define the size of the edges `M1`, `M2`, `M3` and `M4`, are also stated in the interface.

The call for the function `Data_pointer` is done at this level, necessarily before the call for `Cauchy_ProblemS` subroutine.

Finally, the call for `Cauchy_ProblemS` subroutine is done. This call involves the function `Space_Discretization`, which at the same time calls the subroutine that carries out steps from 1 to 4 of the algorithm, `Space_Discretization_2D_system`. The effect of this is recursively execute the algorithm for every temporal step. Once the sequence on calls is performed for all temporal steps, the problem is solved.

Summarizing, the code that solves IVBP and BVP is structured in an interface called `IVBP2D_system_mixed` which contains the functions and subroutines described during the Algorithm Implementation of the present section. The interface is framed in a module called `IVBP_and_BVP` which also contains the subroutines `Binary_Search` and `Data_Pointer`.

Part III

Application Program Interface

Chapter 1

Systems of equations

1.1 Overview

This is a library designed to solve systems of equations.

The API contains two modules: `Linear_systems` and `Non_Linear_Systems`. With the `Linear_systems` module the user is able to solve a linear system of equations. With the `Non_Linear_Systems` module the user is able to solve a non linear system of equations.

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:

$$A \cdot \boldsymbol{x} = \boldsymbol{b}$$

A and $\boldsymbol{b}$ are the given values. The result of the function is:

Function result	**Type**	**Description**
x	vector of reals	Solution ($\boldsymbol{x}$) of the linear system of equations.

Table 1.2: Output of `Solve_LU`

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 in the previous equation, but it must be facotred before using the LU method.
b	vector of reals	in	Vector $\boldsymbol{b}$ in the previous equation.

Table 1.3: Description of `Solve_LU` arguments

The dimensions of A and $\boldsymbol{b}$ must match.

1.3 Non Linear Systems module

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: $\mathbb{R}^N \to \mathbb{R}^N$	in	System of equations that wants to be solved.
x0	vector of reals	inout	Initial point to start the iteration. Besides, this vector will contain the solution of the problem after the call. Its dimension must be N.

Table 1.4: Description of `Newton` arguments

Chapter 2

Lagrange Interpolation

2.1 Overview

This library has been designed to apply lagrangian interpolation in order to carry out different computations. There are two modules, defined as `Interpolation` and `Lagrange_interpolation`. In spite of this, the API is contained only in the `Interpolation` module. On the whole, this module contains two main functions: `interpolated_value` and `Integral`. Each of them is based on a lagrangian interpolation, which is permormed in the support module `Lagrange_interpolation`.

There are two main objectives of this API. The first one, attained by the function `interpolated_value`, computes the value of a function at a certain point taking into account values of that function at other points. The second purpose, carried out by the function `Integral`, is related to the computation of the integral of a function in a certain interval.

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 )
```

The result of the function is the following:

Function result	**Type**	**Description**
yp	real	Interpolated value of the function $y(x)$ in $x = x_p$.

Table 2.1: Output of `interpolated_value`

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	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 `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 result of the function is the following:

Function result	**Type**	**Description**
I	real	Value of the piecewise polynomial integration of $y(x)$.

Table 2.3: Output of `Integral`

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 2.4: Description of `Integral` arguments

Chapter 3

Finite Differences

3.1 Overview

This is a library is designed to prepare a PDE problem for a future resolution. The finite differences library obtains the discretization, interpolation and derivative of a function and boundary conditions needed to solve a PDE problem. This will be achieved with the subroutines `Grid_initialization`, `Derivative`, `Dirichlet` and `Neumann`.

The library has two modules: `Finite_differences` and `Non_uniform_grid`. But the API is contained only in the `Finite_differences` module.

3.2 Finite differences module

Grid Initalization

```
call Grid_Initialization( grid_spacing , direction , q , grid_d )
```

This subroutine will calculate a set of points within the space domain defined; $[-1, 1]$ by default. The arguments of the routine are described in the following table.

Argument	**Type**	**Intent**	**Description**
grid_spacing	character	in	Here the grid structure must be chosen. 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	This is the order chosen for the interpolating polynomials. This label is for the software to be sure that the number of nodes (N) is greater than the polynomials order (at least $N = \text{order} + 1$).
grid_d	vector of reals	inout	Contains the mesh nodes.

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.

Derivative for 1D grids

```
call Derivative ( direction , derivative_order , W , Wxi )
```

The subroutine `Derivative` approximates the derivative of a function by using finite differences. It performs the operation:

$$\frac{\partial^k W}{\partial x^k} = W_{xk}$$

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 ($k = 1$ first derivate, $k = 2$ second derivate and so on).
W	vector of reals	in	Values that the function has at the given points.
Wxi	vector of reals	out	Result. Value of the k-derivate of the given function.

Table 3.2: Description of `Derivative` arguments for 1D grids

Derivative for 2D and 3D grids

```
call Derivative ( direction , coordinate , derivative_order , W , Wxi )
```

The subroutine `Derivative` approximates the derivative of a function by using finite differences. It performs the operation:

$$\frac{\partial^k W}{\partial x^k} = W_{xk}$$

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 ($k = 1$ first derivate, $k = 2$ second derivate and so on).
W	N-dimensional array of reals	in	Values that the function has at the given points.
Wxi	N-dimensional array of reals	out	Result. Value of the k-derivate of the given function.

Table 3.3: Description of `Derivative` arguments for 2D and 3D grids

Chapter 4

Cauchy Problem

4.1 Overview

This library is designed to solve the Cauchy problem. The Cauchy problem is defined as:

$$\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = \boldsymbol{f}\ (\boldsymbol{U},\ t)$$
$$\boldsymbol{U} = \boldsymbol{U}_0$$

The library has two modules: `Cauchy_problem` and `Temporal_Schemes`. However, the API is contained only in the `Cauchy_problem` module.

4.2 Cauchy problem module

Cauchy ProblemS

```
call Cauchy_ProblemS ( Time_Domain, Differential_operator, Scheme, Solution )
```

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	vector of reals	in	Time domain where the solution wants to be calculated.
Differential_operator	vector function: $\mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$	in	It is the funcition $\boldsymbol{f}\ (\boldsymbol{U},\ t)$ described in the overview.
Scheme	temporal scheme	in (optional)	Defines the scheme used to solve the problem. If it is not specified it uses a Runge Kutta of four steps by default.
Solution	vector of reals	out	Contains the solution $\boldsymbol{U}(t)$. The first index represents the time, the second index contains the components of the solution.

Table 4.1: Description of `Cauchy_ProblemS` arguments

4.2.1 Temporal schemes

The schemes that are available in the library are listed below. h denotes the time step.

Scheme	**Name**	**Formula**
Euler	Euler	$U_{n+1} = U_n + hf(t_n, U_n)$
Runge Kutta 2	Runge_Kutta2	$U_{n+1} = U_n + h(\frac{k_1+k_2}{2})$ $k_1 = f(t_n, U_n)$ $k_2 = f(t_n + h, U_n + hk_1)$
Runge Kutta 4	Runge_Kutta4	$U_{n+1} = U_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$ $k_1 = f(t_n, U_n)$ $k_2 = f(t_n + \frac{1}{2}h, U_n + \frac{1}{2}hk_1)$ $k_3 = f(t_n + \frac{1}{2}h, U_n + \frac{1}{2}hk_2)$ $k_4 = f(t_n + h, U_n + hk_3)$
Leap Frog	Leap_Frog	$U_{n+2} = U_n + 2f(t_{n+1}, U_{n+1})$
Adams Bashforth 2	Adams_Bashforth	$U_{n+2} = U_{n+1} +$ $+h\left(\frac{3}{2}f(t_{n+1}, U_{n+1}) - \frac{1}{2}f(t_n, U_n)\right)$
Adams Bashforth 3	Adams_Bashforth3	$U_{n+3} = U_{n+2} +$ $+h\left(\frac{23}{12}f(t_{n+2}, U_{n+2}) -\right.$ $\left.-\frac{4}{3}f(t_{n+1}, U_{n+1}) + \frac{5}{12}f(t_n, U_n)\right)$
Predictor Corrector	Predictor_Corrector1	$\bar{U}_{n+1} = U_n + hf(t_n, U_n)$ $U_{n+1} = U_n + \frac{1}{2}h(f(t_{n+1}, \bar{U}_{n+1}) + f(t_n, U_n))$
Euler Inverso	Inverse_Euler	$U_{n+1} = U_n + hf(t_{n+1}, U_{n+1})$
Crank Nicolson	Crank_Nicolson	$U_{n+1} = U_n + \frac{1}{2}h(f(t_{n+1}, U_{n+1}) + f(t_n, U_n))$

Table 4.2: Description of the available schemes

Chapter 5

Boundary Value Problems

5.1 Overview

This library is designed to solve both linear and non linear boundary value problems. A boundary value problem appears when a equation in partial derivatives is to be solved inside a region (space domain) according to some constraints which applies to the frontier of this domain (boundary conditions). The library has a module: `Boundary_value_problems`, where the API is contained. The API consists of 2 subroutines: one to solve linear problems and the other to solve non linear problems. Finally, depending on the inputs of the subroutines, a 1D problem or a 2D problem is solved.

5.2 Boundary value problems module

1D Boundary Value Problems

```
call Boundary_Value_Problem( x_nodes , Order , Differential_operator ,  &
                             Boundary_conditions , Solution )
```

The subroutine calculates the solution of the following boundary value problem:

$$\mathcal{L}\left(x,\ u,\ \frac{\partial u}{\partial x},\ \frac{\partial^2 u}{\partial x^2}\right) = 0$$

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.
Order	integer	in	It indicates the order of the finitte differences.
Differential_operator	real function: $\mathcal{L}\left(x, u, u_x, u_{xx}\right)$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f\left(x, u, u_x\right)$	in	In this function, the boudary conditions are fixed. The user must include a conditional sentence which sets $f\left(a,\ u,\ u_x\right) = f_a$ and $f\left(b,\ u,\ u_x\right) = f_b$.
Solution	vector of reals	out	Contains the solution, $u = u(x)$, of the boundary value problem.

Table 5.1: Description of `Linear_Boundary_Value_Problem` arguments for 1D problems

2D Boundary Value Problems

```
call Linear_Boundary_Value_Problem( x_nodes , y_nodes , Order ,        &
                                    Differential_operator ,            &
                                    Boundary_conditions , Solution )
```

This subroutine calculates the solution to a linear boudary 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.
Order	integer	in	It indicates the order of the finitte differences.
Differential_operator	real function: $\mathcal{L}\left(x, y, u, u_x, u_y, u_{xx}, u_{yy}, u_{xy}\right)$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f\left(x, y, u, u_x, u_y\right)$	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 $[a, b] \times [c, d]$:

$$\boldsymbol{\mathcal{L}}\left(x,\ y,\ \boldsymbol{u},\ \frac{\partial \boldsymbol{u}}{\partial x},\ \frac{\partial \boldsymbol{u}}{\partial y},\ \frac{\partial^2 \boldsymbol{u}}{\partial x^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial y^2},\ \frac{\partial^2 \boldsymbol{u}}{\partial x \partial y}\right) = \boldsymbol{0}$$

The solution of this problem is calculated using the libraries by a simple call to the subroutine:

```
call Linear_Boundary_Value_Problem(x_nodes, y_nodes, Order, N_variables,
    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.
Order	integer	in	Order of the finite differences.
N_variables	integer	in	Number of variables of the differential equations system.
Differential_operator	function: $\boldsymbol{\mathcal{L}}(x, y, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy})$	in	This function is the differential operator.
Boundary_conditions	function: $\boldsymbol{f}(x, y, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y)$	in	Boundary conditions for all variables.
Solution	three-dimensional array of reals	out	Solution $\boldsymbol{u} = \boldsymbol{u}(x, y)$.

Table 5.3: Description of `Linear_Boundary_Value_Problem_System` arguments for 2D problems

3D Boundary Value Problemsfor systems of equations

```
call Boundary_Value_Problem(x_nodes, y_nodes, z_nodes, Order,   &
                            N_variables, 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.
Order	integer	in	Order of FD formulas.
N_variables	integer	in	Number of variables.
Differential_operator	function: $\boldsymbol{\mathcal{L}}(x,y,\boldsymbol{u},\boldsymbol{u}_x,\boldsymbol{u}_y,\boldsymbol{u}_z,\boldsymbol{u}_{xx},$ $\boldsymbol{u}_{yy},\boldsymbol{u}_{zz},\boldsymbol{u}_{xy},\boldsymbol{u}_{xz},\boldsymbol{u}_{yz})$	in	Differential operator.
Boundary_conditions	function: $\boldsymbol{f}\,(x,y,z,\boldsymbol{u},\boldsymbol{u}_x,\boldsymbol{u}_y,\boldsymbol{u}_z)$	in	Boundary conditions.
Solution	4-dimensional array of reals	out	Solution $\boldsymbol{u}=\boldsymbol{u}(x,y,z)$.

Table 5.4: Description of `Linear_Boundary_Value_Problem_System` arguments for 3D problems

Chapter 6

Initial Value Boundary Problem

6.1 Overview

This library is designed to solve a boundary initial value problem. The initial value boundary problem is composed by equations in partial derivatives which change with time. Then, the complexity of this problem mixes the resolution scheme of a Cauchy problem (in order to solve the temporal evolution) with the procedure for solving a boundary value problem whose unknowns change in every time iteration. The library has a module: `Initial_Value_Boundary_Problem`, where the API is contained.

6.2 Initial Value Boundary Problem module

1D Initial Value Boundary Problem

```
call Initial_Value_Boundary_ProblemS( Time_Domain , x_nodes , Order ,   &
                                      Differential_operator ,           &
                                      Boundary_conditions ,  Solution )
```

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.
Order	integer	in	Order of FD formulas.
Differential_operator	real function: $\mathcal{L}(x, t, u, u_x, u_{xx})$	in	Differential operator.
Boundary_conditions	real function: $f(x, t, u, u_x)$	in	The user must include a conditional sentence which sets $f(a,\ t,\ u,\ u_x) = f_a$ and $f(b,\ t,\ u,\ u_x) = f_b$.
Scheme	temporal scheme	in (optional)	Numerical scheme to integrate in time. If it is not specified, it uses a Runge Kutta of four steps.
Solution	two-dimensional array of reals	out	Solution $u = u(x, t)$.

Table 6.1: Description of `Initial_Value_Boundary_ProblemS` arguments for 1D problems

1D Initial Value Boundary Problem for systems of equations

```
call Initial_Value_Boundary_ProblemS( Time_Domain , x_nodes , Order ,    &
                                      Differential_operator ,            &
                                       Boundary_conditions ,  Solution )
```

The subroutine Initial_Value_Boundary_ProblemS 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: $\boldsymbol{f}\left(x, t, \boldsymbol{u}, \boldsymbol{u}_x\right)$	in	Boundary conditions.
Scheme	temporal scheme	in (optional)	Optional temporal scheme. Default: Runge Kutta of four steps.
Solution	three-dimensional array of reals	out	Solution $\boldsymbol{u} = \boldsymbol{u}(x, t)$.

Table 6.2: Description of Initial_Value_Boundary_ProblemS_System arguments for 1D problems

2D Initial Value Boundary Problems

```
call Initial_Value_Boundary_ProblemS( Time_Domain, x_nodes, y_nodes, Order,  &
                                      Differential_operator,                 &
                                      Boundary_conditions,  Solution )
```

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.
Scheme	temporal scheme	in (optional)	Scheme used to solve the problem. If not given a Runge Kutta of four steps is used.
Solution	three-dimensional array of reals	out	Solution of the problem u.

Table 6.3: Description of `Initial_Value_Boundary_ProblemS` arguments for 2D problems

Initial Value Boundary ProblemS System for 2D problems

```
call Initial_Value_Boundary_ProblemS( Time_Domain , x_nodes , y_nodes , Order
    , Differential_operator , Boundary_conditions ,  Solution )
```

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.
Order	integer	in	It indicates the order of the finitte differences.
Differential_operator	function: $\boldsymbol{\mathcal{L}}$	in	Differential operator.
Boundary_conditions	function: $\boldsymbol{h}$	in	Boundary conditions.
Scheme	temporal scheme	in (optional)	Scheme used to solve the problem. If not given, a Runge Kutta of four steps is used.
Solution	four-dimensional array of reals	out	Solution of the problem $\boldsymbol{u}$.

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 designed to solve an initial value boundary problem for a vectorial variable $\boldsymbol{u}$ with a coupled boundary value problem for $\boldsymbol{v}$.

Then, the complexity of this problem mixes the resolution scheme of an initial value boundary problem (in order to solve the temporal evolution) with the procedure for solving a boundary value problem whose unknowns change in every time iteration. The library has a module: `IVBP_and_BVP`, where the API is contained.

7.2 Mixed Boundary and Initial Value Problems module

Mixed Boundary and Initial Value Problems for 2D problems

The subroutine `IVBP_and_BVP_Problem` calculates the solution to a boundary initial value problem in a rectangular domain $(x, y) \in [x_0, x_f] \times [y_0, y_f]$:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{\mathcal{L}}_u(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy}, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_{xx}, \boldsymbol{v}_{yy}, \boldsymbol{v}_{xy})$$

$$\boldsymbol{\mathcal{L}}_v(x, y, t, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_{xx}, \boldsymbol{v}_{yy}, \boldsymbol{v}_{xy}, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y, \boldsymbol{u}_{xx}, \boldsymbol{u}_{yy}, \boldsymbol{u}_{xy}) = 0,$$

$$\boldsymbol{h}_u(x, y, t, \boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y)\bigg|_{\partial\Omega} = 0, \qquad \boldsymbol{h}_v(x, y, t, \boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y)\bigg|_{\partial\Omega} = 0.$$

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 IVBP_and_BVP_Problem( Time_Domain , x_nodes , y_nodes , Order , N_u , N_v
     , Differential_operator_u ,&  Differential_operator_v ,
    Boundary_conditions_u    , Boundary_conditions_v , Ut ,  Vt  )
```

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.
N_u	integer	in	Dimension of $\boldsymbol{u}$.
N_v	integer	in	Dimension of $\boldsymbol{v}$.
Differential_operator_u	function: $\boldsymbol{\mathcal{L}}_u$	in	Differential operator for $\boldsymbol{u}$.
Differential_operator_v	function: $\boldsymbol{\mathcal{L}}_v$	in	Differential operator for $\boldsymbol{v}$.
Boundary_conditions_u	function: $\boldsymbol{h}_u$	in	Boundary conditions for $\boldsymbol{u}$.
Boundary_conditions_v	function: $\boldsymbol{h}_v$	in	Boundary conditions for $\boldsymbol{v}$.
Scheme	temporal scheme	in (optional)	Scheme used to solve the problem. If not given, a Runge Kutta of four steps is used.
Ut	four-dimensional array of reals	out	Solution, $\boldsymbol{u}$, of the evolution problem.
Vt	four-dimensional array of reals	out	Solution, $\boldsymbol{v}$, of the boundary value problem.

Table 7.1: Description of `IVBP_and_BVP_Problem` arguments for 2D problems