How to learn Applied Mathematics trough modern Fortran

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

Contents

Chapter 1

Some basic programs

Chapter 2

First examples: calculus and algebra

2.1 My first program: "Hello world"

batch bsh

  scripts

  gfortran

2.2 sum of a numeric series

Dar el resultado de la suma de los 100 primeros trminos de las siguientes series:

1. Serie de nmeros naturales.
2. Serie de nmeros naturales impares.
3. Serie numrica donde el trmino general de la serie es: $a_n = 1/n^2$ desde $n = 1$.
4. Serie numrica donde el trmino general de la serie es $a_n = 1/n!$ desde $n = 1$.
5. Serie numrica donde el trmino general de la serie es $a_n = (-1)^{n+1}/(2n-1)$ desde $n = 1$.

2.3 Operaciones con matrices y vectores

Considerar los vectores $V, W \in \mathbb{R}^N$ de componentes:

$$\{v_i = \frac{1}{i^2}, \quad i = 1 \dots N\},$$

$$\{w_i = \frac{(-1)^{i+1}}{2i-1}, \quad i = 1 \dots N\}.$$

Considerar la matriz $A \in \mathcal{M}_{N\times N}(\mathbb{R})$ donde su trmino genrico vale $a_{ij} = (i/N)^j$. Escribir un programa para calcular las operaciones siguientes con $N = 100$:

1. Suma de todas las componentes del vector V y del vector W.
2. Suma de todas las componentes de la matriz A.
3. Suma de las componentes del vector W mayores que cero.
4. Producto escalar de los vectores V y W.
5. Producto escalar del vector V y la columna N de la matriz A.
6. Suma de las componentes de vector que resulta de multiplicar la matriz A por el vector V.
7. Traza de la matriz A.

2.4 dynamic allocation of memory

Dada la matriz $A \in \mathcal{M}_{M\times M}(\mathbb{R})$ de trmino genrico

$$\{a_{ij} = (i/M)^j, \quad i = 0, \dots M-1, \quad j = 0, \dots M-1\}.$$

calcular las siguientes operaciones:

1. Calcular
$$\sum_{M=1}^{10} traza(A)$$
2. Calcular
$$\sum_{M=1}^{5} traza(A^2)$$
3. Calcular con $M = 4$
$$traza\left(\sum_{k=1}^{5} A^k\right)$$

2.5 Piecewise functions

Sean los vectores $X, F \in \mathbb{R}^{N+1}$. Las componentes de X almacenan los valores discretos del dominio de definicin y F las imgenes correspondientes de la funcin $F : \mathbb{R} \to \mathbb{R}$ continua a trozos siguiente:

$$F(x) = \begin{cases} 1, & a \le x \le -\dfrac{\pi}{2}, \\ \cos(\pi x), & -\dfrac{\pi}{2} < x < \dfrac{\pi}{2}, \\ 0, & \dfrac{\pi}{2} \le x \le b. \end{cases}$$

Considerar una particin equiespaciada de la forma:

$$\{x_i = a + i\Delta x, \quad i = 0 \dots N\}, \qquad \Delta x = \frac{b-a}{N}, \qquad a < -\frac{\pi}{2}, \qquad b > \frac{\pi}{2}.$$

Se pide calcular la suma;

$$S_N = \sum_{i=0}^{N} F_i \Delta x$$

1. con $N = 10$
2. con $N = 20$
3. con $N = 100$

2.6 Series de funciones

Aproximar mediante un desarrollo en serie de potencias de la forma

$$f(x) = \sum_{k=0}^{M} a_k \; x^k, \qquad\qquad a_k = \frac{f^{(k)}(0)}{k!},$$

las funciones $F : \mathbb{R} \to \mathbb{R}$, siguientes:

1. $f(x) = e^x$ y calcular el valor $f(1)$ con $M = 5$.
2. $f(x) = \sin(x)$ y calcular el valor $f(\pi/2)$ con $M = 8$.
3. $f(x) = \cosh(x)$ y calcular el valor $f(1)$ con $M = 10$.
4. $f(x) = \dfrac{1}{1-x}$ y calcular el valor $f(0.9)$ con $M = 20$.
5. $f(x) = e^x$ y calcular el valor ms preciso de $f(1)$ con doble precisin.
6. $f(x) = \sin(x)$ y calcular el valor ms preciso $f(\pi/2)$ con doble precisin.
7. $f(x) = \cosh(x)$ y calcular el valor ms preciso de $f(1)$ con doble precisin.
8. $f(x) = \dfrac{1}{1-x}$ y calcular el valor ms preciso de $f(0.9)$ con doble precisin.

2.7 Lectura y escritura de ficheros

Crear los ficheros de datos ForTran con nombres `input_1.dat` e `input_2.dat` con la información siguiente:

Contenido del fichero de entrada `input_1.dat` :

```
Datos de entrada 1

1.2      3.4     6.2    -14.0    0.1
-25.2    -8.6    5.1     9.9    17.0
-1.0     -2.0    -5.4    -8.6    0.0
3.14     -11.9   -7.0    -12.1   9.2
6.66     5.32    0.001   0.2     0.001
```

Contenido del fichero de entrada `input_2.dat` :

```
Datos de entrada 2

1.2      3.4     6.2    -14.0    0.1      4.89   7.54
-25.2    -8.6    5.1    12.0     9.9      12.24  17.0
0.0      34.5    -1.0   -2.0     -43.04   -8.6    0.0
3.14     -11.9   71.0   7.0     17.0      -12.1   9.2
6.66     5.32    0.001   0.2     0.001    0.008   -0.027
54.0     77.1    -9.002  -13.2   0.017    65.53   -0.021
23.04    -51.98  -34.2   9.99    5.34     8.87    3.22
```

Escribir un programa que gestione los datos de los ficheros anteriores siguiendo los pasos siguientes:

Declarar las matrices $A \in \mathcal{M}_{N\times N}(\mathbb{R})$, $B \in \mathcal{M}_{N\times 3}(\mathbb{R})$, $C \in \mathcal{M}_{N\times 2}(\mathbb{R})$ y los vectores $U, V, W, T \in \mathbb{R}^N$.

Leer el fichero de entrada (`input_1.dat` o `input_2.dat`) de la forma siguiente:

1. Cargar el fichero completo en la matriz A.
2. Cargar las cuatro primeras columnas del fichero en los vectores U, V , W y T.
3. Cargar la primera columna en el vector T y las tres últimas columnas en la matriz B.
4. Cargar la segunda columna en el vector U y las dos últimas columnas en la matriz C.
5. Cargar las columnas 1, 2 y 4 en la matriz B.

Además, el programa debe crear el fichero de salida (`output_1.dat` o `output_2.dat`), donde se irán escribiendo las matrices y vectores de los apartados anteriores. El formato de escritura debe ser el de números reales con cinco decimales.

Para el enunciado anterior, escribir los programas siguientes:

1. Programa 1 : Asignación estática de memoria.
   Ejecutar el programa por separado para los ficheros `input_1.dat` e `input_2.dat`. Para ello modificar las dimensiones y en nombre de los ficheros en el programa fuente.

2. Programa 2 : Asignación dinámica de memoria.
   Ejecutar el programa una única vez para gestionar los datos de los ficheros de entrada `input_1.dat` e `input_2.dat`.

2.8 Sistemas lineales de ecuaciones

Implementar un mdulo para la resolucin de sistemas lineales de ecuaciones algebraicas. Los mtodos de resolucin propuestos son el de eliminacin Gaussiana, factorizacin LU, factorizacin LU de la biblioteca *Numerical Recipes* y Jacobi.

Para cada mtodo se pide:

- Validar los resultados con varios casos de prueba con dimensiones distintas.
- Evaluar tiempos de ejecucin.
- Comparar resultados con los mtodos restantes.

Aplicacin : Estudiar el condicionamiento de sistemas lineales de ecuaciones para matrices aleatorias y de Vandermonde.

2.9 Sistemas no lineales de ecuaciones

Implementar un mdulo para la resolucin numrica de ecuaciones no lineales. Para funciones $F : \mathbb{R} \to \mathbb{R}$, los mtodos de resolucin propuestos son el de la biseccin y el de Newton-Raphson. Para funciones $F : \mathbb{R}^N \to \mathbb{R}^N$, se proponen el mtodo de Newton-Raphson con matriz Jacobiana analtica y el mtodo de Newton-Raphson con matriz Jacobiana numrica. Para la validacin de los mtodos propuestos, se pide implementar un mdulo con al menos tres funciones $F : \mathbb{R} \to \mathbb{R}$ y al menos tres funciones $F : \mathbb{R}^N \to \mathbb{R}^N$. Este mdulo debe contener las derivadas y matrices Jacobianas correspondientes de las funciones propuestas.

En el informe correspondiente, presentar tablas de soluciones numricas en cada paso de iteracin para las funciones de prueba propuestas.

2.10 Autovalores y autovectores

Implementar un mdulo para el clculo de autovalores y autovectores de una matriz. Los mtodos de resolucin propuestos son el mtodo de la potencia y el mtodo de la potencia inversa. Implementar el mtodo de la potencia inversa a partir de la matriz inversa y resolviendo el sistema lineal correspondiente.

Para cada mtodo se pide:

- Validar los resultados con varios casos de prueba con dimensiones distintas.
- Evaluar tiempos de ejecucin. Comparar tiempos de ejecucin del mtodo de la potencia inversa mediante los dos algoritmos propuestos : matriz inversa y solucin del sistema lineal.

Aplicacin : Estudiar el condicionamiento de sistemas lineales de ecuaciones para matrices aleatorias y de Vandermonde. Calcular la relacin $\lambda_{max}/\lambda_{min}$ de los casos de prueba presentados en el hito 1 y relacionar y discutir los resultados.

```
subroutine   power_method

integer :: i, j, k
integer, parameter :: PI = 4 * atan(1d0)
integer, parameter :: N = 20
real :: x(0:N), Vandermonde(0:N, 0:N), sigma
real :: a=-1, b=1
real V(0:N), V0(0:N)


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

forall(i=0:N, j=0:N) Vandermonde(i,j) = x(i)**j

V = 1
V0 = 0
do while( abs(norm2(V)-norm2(V0)) > 1d-5 )
V0 = V
V = matmul( Vandermonde, V ) / norm2(V)
write(*,*) maxval(V)
end do
sigma = dot_product( V, V )
write(*,*) "sigma = ", sigma


end subroutine
```

2.11 Derivacin numrica

1. Obtener las frmulas de las derivadas numricas primeras descentradas, con tres puntos equiespaciados a una distancia Δx.
2. A partir de la funcin $f(x) = \mathrm{e}^x$ en el punto $x = 0$, representar grficamente el error total de las derivadas numricas frente al valor de Δx en precisin simple y doble. En particular, representar grficamente las derivadas primeras adelantada (definicin de derivada), centrada y descentradas y la derivada segunda, con tres puntos equiespaciados a una distancia Δx. Discutir los resultados obtenidos.
3. Resolver los problemas de contorno en ecuaciones diferenciales ordinarias siguientes:
   - **Problema 1:**
   $$u'' + u = 0, \quad x \in [-1, 1], \qquad\qquad u(-1) = 1, \quad u(1) = 0,$$
   - **Problema 2:**
   $$u'' + u' - u = \sin(2\pi x), \quad x \in [-1, 1], \qquad\qquad u(-1) = 0, \quad u'(1) = 0.$$

Para los problemas citados anteriormente se pide:

(a) A partir de las derivadas numricas con tres puntos equiespaciados escribir el sistema de ecuaciones resultante.
(b) Obtener la solucin numrica mediante la resolucin de un sistema lineal de ecuaciones, con $N = 10$ y $N = 100$.
(c) Representar grficamente los resultados obtenidos.

2.12 Integracin numrica

Implementar un mdulo para la resolucin numrica de integrales definidas de funciones $F : \mathbb{R} \to \mathbb{R}$. Los mtodos de resolucin propuestos son las reglas del rectngulo, punto medio, trapecio y Simpson. Implementar un mdulo de funciones $F : \mathbb{R} \to \mathbb{R}$ de prueba para validar los mtodos numricos propuestos. Este mdulo debe contener al menos tres funciones con funciones primitivas conocidas y una funcin cuya funcin primitiva sea desconocida.

Evaluar el error de las soluciones numricas para cada mtodo propuesto y para distintos valores del incremento de la particin.

2.13 to be included

elemental advance = no dummy versus actual assumed shape explicit shape

global, local and scope in modules

1 versus 1.

tab instead of blanks

brackets

mask in intrinsic functions

! and &

;

camel case versus underscore

overloading

forall parallel

enter matrices by row or columns

lower bound: upper bound

array operations C = A + B

public versus private

encapsulamiento y ocultaci

FORALL (I=1:N-1, J=1:N, J¿I) A(I,J) = A(J,I)

```
x   = 1.23456789123456789123456789Q00
xd  = 1.23456789123456789123456789Q00
xdd = 1.23456789123456789123456789Q00
write(*, '(ES)' ) x  ! scientific notation 1.234 (first digit should be greater or equal than one
write(*, '(E)' ) x   ! exponential normalized  notation 0.1234 (first digit is zero )

write(*,*) " Single, double and quadruple precision "
write(*, '(E)' ) x
write(*, '(E)' ) xd
write(*, '(E)' ) xdd




!****************************************************************************
!*
!****************************************************************************
subroutine type_element




interface operator (+)
module procedure element
end interface

character(len=20) :: name
real (kind=4) :: x
real (kind=8) :: xd
real (kind=16) :: xdd

real :: a, b, c;
!  real :: a1, a2, a3;


type (person) :: father = person( "juan"), mother = person("cris")

associate ( name1 => father%name, name2=>mother%name )
name = trim(name1) // trim(name2)
end associate

a = 1; b = 1 ; c = 1;
associate ( a1 =>a, a2 =>b, a3=>c )
a3 = a1 + a2
write(*,*) " a3 = ", a3
end associate
write(*,*) " c = ", c
!    write(*,*) " a3 = ", a3

!   write(*,*) " father =", father % name
```

```
write(*,*) " element =", father + mother
write(*,*) " name =", name

x   = 1.23456789123456789123456789Q00
xd  = 1.23456789123456789123456789Q00
xdd = 1.23456789123456789123456789Q00
write(*, '(ES)' ) x  ! scientific notation 1.234 (first digit should be greater or equal than one
write(*, '(E)' ) x   ! exponential normalized  notation 0.1234 (first digit is zero )

write(*,*) " Single, double and quadruple precision "
write(*, '(E)' ) x
write(*, '(E)' ) xd
write(*, '(E)' ) xdd

write(*, '(3(E, :, ","))' ) x, xd, xdd

end subroutine



!*****************************************************************************
!*
!*****************************************************************************
subroutine Hello_world


write(*,'(a)', advance='no') " Hello world..... "
write(*,'(a)', advance='no') " press enter"
read(*,*)

end subroutine



!*****************************************************************************
!*
!*****************************************************************************
subroutine  commandline

character(len=256) :: line,  enval
integer :: i, iarg, stat, clen, len
integer :: estat, cstat

iarg = command_argument_count()
write(*,*) " iarg = ", iarg
do i=1,iarg
call get_command_argument(i,line,clen,stat)
write (*,'(I0,A,A)') i,': ',line(1:clen)
end do
```

```
call get_command(line,clen,stat)
write (*,'(A)') line(1:clen)

call get_environment_variable('HOSTNAME',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'Host=', enval(1:len)
call get_environment_variable('USER',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'User=', enval(1:len)


! call execute_command_line('ls -al', .TRUE., estat, cstat)
call execute_command_line('dir', .TRUE., estat, cstat)
if (estat==0) write (*,'(A)') "Command completed successfully"

end subroutine

!******************************************************************************
!*
!******************************************************************************
subroutine  allocate_characteristics

real, allocatable :: V(:), A(:, :)
character(:), allocatable :: S
integer :: i, j, N

real, pointer ::  B(:,:), Diagonal(:)
real, pointer :: memory(:)

real :: x, y, z
class(*), pointer :: p1(:)


N = 10
V = [ ( i/real(N), i=1, N ) ]  ! automatic allocation allocate( V(N) )
write(*,* ) " V = ", V

N = 2
A = reshape( [ (  ( (i/real(N))**j ,i=1, N ), j=1, N ) ], [N, N] )
do i=1, N
write(*,* ) " A = ", A(i,:)
end do

N = 4
A = reshape( [ (  ( (i/real(N))**j ,i=1, N ), j=1, N ) ], [N, N] )
do i=1, N
write(*,'(A, 100f6.2)' ) " A = ", A(i,:)
end do

S = "Hello world"
write(*,*) " S = ", S, len(S)
```

```
S = "Hello"
write(*,*) " S = ", S, len(S)


allocate( memory(1:N*N) )
B(1:N,1:N) => memory
memory = 0
forall(i=1:N) B(i,i) = 10.
do i=1, N
write(*,'(A, *(f6.2) )' ) " B = ", B(i,:)
end do
diagonal => memory(::N+1)
write(*,'(A, 100f6.2)' ) " diagonal = ", diagonal
write(*,'(A, 100f6.2)' ) " trace = ", sum(diagonal)
write(*,'(A, 100f6.2)' ) " trace = ", sum(memory(::N+1))

x = 1; y = 2; z = 3;

write(*,'("i=",I0,", REALs=",*(G0,1X),"....")') i, x, y, z  ! C++ style


allocate( integer :: p1(5) ) ! p1 is an array of integers

select type (p1)
type is (integer)
p1 = [( i, i=1, size(p1) ) ]
write(*,'(" p1 = ", *(I0, 1x) )')  p1

class default
stop 'Error in type selection'
end select

deallocate(p1)
allocate( real :: p1(3) ) ! now p1 is an array of reals

select type (p1)
type is (real)
p1 = [( i, i=1, size(p1) )  ]
write(*,'(" p1 = ", *(G0, 1x) )')  p1

class default
stop 'Error in type selection'
end select



end subroutine
```

Chapter 3

System of Equations

3.1 Overview

This is a library designed to solve systems of equations.

It has three modules: **Linear_systems**, **Non_Linear_Systems** and **Jacobian_module**. In spite of this, the API is contained only in the **Linear_systems** and in the **Non_Linear_Systems** modules. With the **Linear_systems** module the user must be able to solve a linear system of equations. With the **Non_Linear_Systems** module the user must be able to solve a linear system of equations.

3.2 Example using the API

For the sake of clarity, a file called **API_Example_Systems_of_Equations.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Linear_systems** and **use Non_Linear_Systems**.

The first example consists of a linear system of equations of four unknowns with four equations. First of all, it is defined the matrix which contains the terms of the equation, and after the solution. In this example:

$$\begin{bmatrix} 4 & 3 & 6 & 9 \\ 2 & 5 & 4 & 2 \\ 1 & 3 & 2 & 7 \\ 2 & 4 & 3 & 8 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 5 \\ 2 \end{pmatrix}$$

The second example consists in the solution of a nonlinear system of equations defined as follows:

$$F_1 = x^2 - y3 - 2$$
$$F_2 = 3xy - z$$
$$F_3 = z^2 - x$$

```fortran
module Equations_example

   use Linear_Systems
   use Non_Linear_Systems

   implicit none

contains

subroutine LU_Solution

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

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

    b = [ 3, 1, 5, 2]

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

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

end subroutine

subroutine Newton_Solution

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

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

end subroutine

function F(v)
    real, intent(in) :: v(:)
    real:: F(size(v))

    real :: x, y, z

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

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

end function

end module

program Example
    use Equations_example

    call LU_solution
    call Newton_solution

end program
```

../sources/Systems_of_equations_example.f90

3.3 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 3.1: *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:

$$\mathbf{A} \cdot \vec{x} = \vec{b}$$

$\mathbf{A}$ and $\vec{b}$ are the given values. The result of the function is:

Function result	Type	Description
x	vector of reals	Solution ($\vec{x}$) of the linear system of equations.

Table 3.2: *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 $\vec{b}$ in the previous equation.

Table 3.3: *Description of* ***Solve_LU*** *arguments*

The dimensions of **A** and $\vec{b}$ must match.

3.4 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 3.4: *Description of* ***Newton*** *arguments*

Chapter 4

Lagrange Interpolation

4.1 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.

4.2 Example using the API

For the sake of clarity, a file called **API_Example_Lagrange_Interpolation.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Interpolation**.

The first subroutine, called **interpolated_Solution**, is devoted to clarify the usage of the function **interpolated_value**. Through this function, one could obtain the value of a function at a certain point by means of the values of that function at other points. For instance, the chosen function is a simple cosine, and the inputs of **interpolated_value** are four values of the function at other points. The final result is the interpolated value of the function at **xp**, which is denoted as **yp**. Since the order of the interpolation has not been defined, it acquires the predefined value of two.

Subsequently, the second subroutine, coined as **Integral_Solution**, carries out the computation of the integral of a certain function in an interval defined by the given points. Again the function is set to be a cosine, and five values of it are known at different points. The function **Integral** enables to perform the integral computation. Since the degree of the interpolation has not been defined, it acquires the predefined value of two.

```fortran
module Interpolation_example

    use Interpolation

    implicit none

contains

subroutine interpolated_Solution

    real :: x(4), y(4), xp, yp

    x = [ -2., -1., 1., 2.]
    y = cos(x * atan(1.0))
    xp = 0.

    yp = interpolated_value( x , y , xp )

    write (*,*) 'The interpolated value at xp is = ', yp

end subroutine

subroutine Integral_Solution

    real :: x(5), y(5), Integral_computation

    x = [ -2., -1., 0., 1., 2.]
    y = cos(x * atan(1.0))

    Integral_computation = Integral( x , y )

    write (*,*) 'The integral computation through the interpolation is = ', Integral_computation

end subroutine

end module

program Example

    use Interpolation_example

    implicit none

    call interpolated_Solution
    call Integral_Solution

end program
```

../sources/Interpolation_example.f90

4.3 Interpolation module

interpolated_value

```
yp = interpolated_value( x, y, xp, degree )
```

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.

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 4.1: *Output of* ***interpolated_value***

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.
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 4.2: *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 4.3: *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 4.4: *Description of **Integral** arguments*

Chapter 5

Finite Differences

5.1 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.

5.2 Example using the API

For the sake of clarity, a file called **API_Example_Finite_Differences.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Finite_differences**.

In the following subroutine, denoted as **Derivative_example**, two derivatives of a certain function $y(r, \theta)$ are obtained by means of the function **Derivative**. Firstly, through the function **Grid_Initialization**, a discretization of the domain is carried out, both in the variables r and θ. Afterwards, and taking into account the values of the function $y(r, \theta)$ at the nodes of the discretized domain, some derivatives are computed by the function **Derivative**:

- The second derivative with respect to r.
- The first derivative with respect to θ.
- The derivative with restpect to θ and with respect to r.

```fortran
module Derivative_example

use Finite_Differences

implicit none

real :: PI = 4 * atan(1.)

contains

subroutine function_derivative

    integer, parameter :: Nx = 15, Ny = 50
    real :: x(0:Nx), y(0:Ny)
    integer :: i, j

    real :: f(0:Nx, 0:Ny), fxx(0:Nx, 0:Ny), fy(0:Nx, 0:Ny), fyy(0:Nx, 0:Ny)

    ! Grid_Initialization
     x(0) = - 1; x(Nx) = 1; y(0) = - 1; y(Ny) = 1;
     call Check_grid( "x", x, 10, Nx+1 )
     call Check_grid( "y", y, 10, Ny+1 )

    ! Derivative

    forall(i=0:Nx, j=0:Ny) f(i,j) = sin(PI*x(i)) * sin(4*PI*y(j))

    call Derivative(direction =['x','y'], coordinate=1, derivative_order=2 , W = f  , Wxi = fxx)
    call Derivative(direction =['x','y'], coordinate=2, derivative_order=1 , W = f  , Wxi = fy)
    call Derivative(direction =['x','y'], coordinate=2, derivative_order=1 , W = fy , Wxi = fyy)


    write(*,*) 'Error fxx = ',  maxval( PI**2 * f(:,:) + fxx(:,:) )
    write(*,*) 'Error fyy = ',  maxval( 16*PI**2 * f(:,:) + fyy(:,:) )


end subroutine

end module

program Example

  use Derivative_example

  call function_derivative

end program
```

../sources/Derivative_example.f90

5.3 Finite_differences module

Grid_Initalization

```fortran
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 5.1: *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 5.2: *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 5.3: *Description of* ***Derivative*** *arguments for 2D and 3D grids*

The subroutine is prepared to be called equally in 2D and 3D problems ($N = 2$ or 3).

5.3.1 Boundary conditions

This library is capable of discretizating two type of boundary conditions: Dirichlet and Neumann. Previously to calling this libraries, **Grid_Initializartion** must be used. The way they are used will be explained below.

Dirichlet

```
call Dirichlet( coordinate , N , W , f )
```

A boundary condition type Dirichlet is defined as:

$$W(\vec{x}_0, t) = f(\vec{x}_0, t) \qquad \vec{x}_0 \in \partial\Omega$$

The subroutine **Dirichlet** imposes the Dirichlet condition. The arguments of the subroutine are described in the following table.

Argument	Type	Intent	Description
coordinate	integer	in	It can be 1 or 2. If 1, the boundary condition will be imposed along the coordinate 2 with the coordinate 1 fixed and vece versa.
N	integer	in	Boundary point at which the codition is imposed.
W	two-dimensional array of reals	inout	It will contain the solution. After entering the subroutine it will have imposed the boundary condition determined by **f**.
f	vector of reals	in	Value of the boundary condition.

Table 5.4: *Description of* ***Dirichlet*** *arguments*

This subroutine only can work with 2D grids.

Neumann for 1D grids

```
call Neumann( direction , N , W , f )
```

A boundary condition type Neumann is defined as:

$$\frac{dW}{dn}(\vec{x}_0, t) = f(\vec{x}_0, t) \qquad \vec{x}_0 \ \epsilon \ \partial\Omega$$

The subroutine **Neumann** imposes the Neumann condition. 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.
N	integer	in	Boundary point at which the condition is imposed.
W	vector of reals	inout	It will contain the solution. After entering the subroutine it will have imposed the boundary condition determined by **f**.
f	vector of reals	in	Value of the boundary condition.

Table 5.5: *Description of **Neumann** arguments for 1D grids*

Neumann for 2D grids

```
call Neumann( direction , coordinate , N , W , f )
```

A boundary condition type Neumann is defined as:

$$\frac{dW}{dn}(\vec{x}_0, t) = f(\vec{x}_0, t) \qquad \vec{x}_0 \,\epsilon\, \partial\Omega$$

The subroutine **Neumann** imposes the Neumann condition. 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 coordinate and so on.
coordinate	integer	in	It can be 1 or 2. If 1, the boundary condition will be imposed along the coordinate 2 with the coordinate 1 fixed and vice versa.
N	integer	in	Boundary point at which the condition is imposed.
W	vector of reals	inout	It will contain the solution. After entering the subroutine it will have imposed the boundary condition determined by **f**.
f	vector of reals	in	Value of the boundary condition.

Table 5.6: *Description of **Neumann** arguments for 2D grids*

Chapter 6

Boundary Value Problem

6.1 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.

6.2 Example using the API

For the sake of clarity, a file called **API_Example_Boundary_Value_Problem.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Boundary_value_problems**.

This example consists of two boundary value problems: a 1D linear problem and a 2D non linear problem. The 1D linear problem is the Legendre differential equation:

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

Where $n = 3$ and the boundary conditions are: $y(-1) = -1$ and $y(1) = 1$. The 2D non linear problem is:

$$\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) u = 0$$

Where the boundary conditions are:

$$u(0,y) = 0 \quad ; \quad u(1,y) = y \quad ; \quad \frac{\partial u}{\partial y}(x,0) = 0 \quad ; \quad u(x,1) = x$$

```fortran
module BVP_examples

  use Boundary_value_problems
  use dislin

  implicit none

contains

subroutine Linear_1D

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

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

    call Linear_Boundary_Value_Problem( x_nodes = x, Order = 4, Differential_operator = L, &
                                        Boundary_conditions = BCs, Solution = U )

    call scrmod('reverse')
    call qplot(x, U, N+1)

    contains

    !Differential operator

    real function L(x, y, yx, yxx)

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

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

    end function

    !Boundary conditions

    real function BCs(x, y, yx)

        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

end subroutine



```

```fortran


subroutine Non_Linear_2D

    integer, parameter :: Nx = 20, Ny = 20
    real :: x(0:Nx), y(0:Ny), U(0:Nx, 0:Ny)
    integer :: i, j
    real :: a=0, b=1, pi = 4 * atan(1.0)

    x = [ (a + (b-a)*i/Nx, i=0, Nx) ]
    y = [ (a + (b-a)*j/Ny, j=0, Ny) ]
    U = 1

    call Non_Linear_Boundary_Value_Problem( x_nodes = x, y_nodes = y, Order = 5,  &
                                            Differential_operator = L, Boundary_conditions = BCs, &
                                            Solution = U )

    call scrmod('reverse')
    call qplclr(U, Nx+1, Ny+1)
    contains

    !Differential operator
    real function L(x, y, u, ux, uy, uxx, uyy, uxy)

        real, intent(in) :: x, y, u, ux, uy, uxx, uyy, uxy

        L = ( uxx + uyy) * u

    end function

    !Boundary conditions
    real function BCs(x, y, u, ux, uy)

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

        if (x==a) then
            BCs = u
        elseif (x==b) then
            BCs = u - y
        elseif (y==a) then
            BCs = uy
        elseif (y==b) then
            BCs = u - x
        else
            write(*,*) " Error BCs x=", x
            write(*,*) " a, b=", a, b
            stop
        endif

    end function

end subroutine

end module
```

../sources/BVP_example.f90

```
program Example

    use BVP_examples

    call Linear_1D
    call Non_Linear_2D

end program
```

../sources/BVP_example.f90

6.3 Boundary_value_problems module

Linear_Boundary_Value_Problem for 1D problems

```
call Linear_Boundary_Value_Problem( x_nodes, Order, Differential_operator, &
                                    Boundary_conditions, Solution )
```

The subroutine **Linear_Boundary_Value_Problem** calculates the solution to a linear boudary value problem such as:

$$\mathscr{L}\left(x,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial^2 U}{\partial x^2}\right) = 0$$

$$f_a\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = a$$

$$f_b\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = b$$

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: $\mathscr{L}\left(x, U, U_x, U_{xx}\right)$	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 6.1: *Description of* ***Linear_Boundary_Value_Problem*** *arguments for 1D problems*

Linear_Boundary_Value_Problem for 2D problems

```
call Linear_Boundary_Value_Problem( x_nodes , y_nodes , Order , Differential_operator ,  &
                                    Boundary_conditions , Solution )
```

The subroutine **Linear_Boundary_Value_Problem** calculates the solution to a linear boudary value problem in a rectangular domain $[a,b] \times [c,d]$:

$$\mathscr{L}\left(x,\ y,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial U}{\partial y},\ \frac{\partial^2 U}{\partial x^2},\ \frac{\partial^2 U}{\partial y^2},\ \frac{\partial^2 U}{\partial x \partial y}\right) = 0$$

$$f_{x=a}\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \quad ; \quad f_{x=b}\left(U,\ \frac{\partial U}{\partial x}\right) = 0$$

$$f_{y=c}\left(U,\ \frac{\partial U}{\partial y}\right) = 0 \quad ; \quad f_{y=d}\left(U,\ \frac{\partial U}{\partial y}\right) = 0$$

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: $\mathscr{L}(x, y, U, U_x, U_y, U_{xx}, U_{yy}, U_{xy})$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f(x, y, U, U_x, U_y)$	in	In this function, the boudary conditions are fixed. The user must use a conditional sentence to do it.
Solution	two-dimensional array of reals	out	Contains the solution, $U = U(x,y)$, of the boundary value problem.

Table 6.2: *Description of* ***Linear_Boundary_Value_Problem*** *arguments for 2D problems*

Non_Linear_Boundary_Value_Problem for 1D problems

```
call Non_Linear_Boundary_Value_Problem( x_nodes, Order, Differential_operator, &
                                        Boundary_conditions, Solver, Solution )
```

The subroutine **Non_Linear_Boundary_Value_Problem** calculates the solution to a non linear boudary value problem in a rectangular domain $[a, b] \times [c, d]$:

$$\mathscr{L}\left(x,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial^2 U}{\partial x^2}\right) = 0$$

$$f_a\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = a$$

$$f_b\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = b$$

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: $\mathscr{L}(x, U, U_x, U_{xx})$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f(x, U, U_x)$	in	In this function, the boudary conditions are fixed. The user must include a conditional sentence which sets $f(a,\ U,\ U_x) = f_a$ and $f(b,\ U,\ U_x) = f_b$.
Solution	vector of reals	out	Contains the solution, $U = U(x)$, of the boundary value problem.

Table 6.3: *Description of* ***Non_Linear_Boundary_Value_Problem*** *arguments for 1D problems*

Non_Linear_Boundary_Value_Problem for 2D problems

```
call Non_Linear_Boundary_Value_Problem( x_nodes , y_nodes , Order , Differential_operator ,
                                        Boundary_conditions , Solver , Solution )
```

The subroutine **Non_Linear_Boundary_Value_Problem** calculates the solution to a non linear boudary value problem such as:

$$\mathscr{L}\left(x,\ y,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial U}{\partial y},\ \frac{\partial^2 U}{\partial x^2},\ \frac{\partial^2 U}{\partial y^2},\ \frac{\partial^2 U}{\partial x \partial y}\right) = 0$$

$$f_{x=a}\left(U,\ \frac{\partial U}{\partial x}\right) = 0 \quad ; \quad f_{x=b}\left(U,\ \frac{\partial U}{\partial x}\right) = 0$$

$$f_{y=c}\left(U,\ \frac{\partial U}{\partial y}\right) = 0 \quad ; \quad f_{y=d}\left(U,\ \frac{\partial U}{\partial y}\right) = 0$$

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: $\mathscr{L}\left(x, y, U, U_x, U_y, U_{xx}, U_{yy}, U_{xy}\right)$	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	In this function, the boudary conditions are fixed. The user must use a conditional sentence to do it.
Solution	two-dimensional array of reals	out	Contains the solution, $U = U(x, y)$, of the boundary value problem.

Table 6.4: *Description of* ***Non_Linear_Boundary_Value_Problem*** *arguments for 2D problems*

Chapter 7

Cauchy Problem

7.1 Overview

This library is designed to solve the Cauchy problem. The Cauchy problem is defined as:

$$\frac{\mathrm{d}\vec{U}}{\mathrm{d}t} = \vec{f}\ (\vec{U},\ t)$$
$$\vec{U} = \vec{U}_0$$

The library has two modules: **Cauchy_problem** and **Temporal_Schemes**. However, the API is contained only in the **Cauchy_problem** module.

7.2 Example using the API

For the sake of clarity, a file called **API_Example_Cauchy_Problem.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Cauchy_Problem**.

This example consists of a trajectory. This problem needs to solve a second degree equation. The problem approach is:

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

It is necessary to give an initial condition of position and velocity. In this example:

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

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

```fortran
module Cauchy_example

 use Cauchy_Problem
 use dislin

 implicit none

contains

subroutine Trajectory

    real :: t0 = 0, tf = 10
    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_Trajectory, &
                          Scheme = Crank_Nicolson , Solution = U )

    call scrmod('reverse')
    call qplot(Time, U(:,1), N+1)

end subroutine

function  F_Trajectory( U, t )  result(F)

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

    real, parameter :: a = 3.0
    real, parameter :: b = 10.0

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

end function

end module



program Example

    use Cauchy_example

    call Trajectory

end program
```

../sources/Cauchy_example.f90

7.3 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 $\vec{f}\ (\vec{U},\ t)$ 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 $\vec{U}(t)$. The first index represents the time, the second index contains the components of the solution.

Table 7.1: *Description of **Cauchy_ProblemS** arguments*

7.3.1 Temporal schemes

The schemes that are available in the library are listed below. h denotes the time step.

Scheme	Name (in the code)	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	$U_{n+2} = U_n + 2f(x_{n+1}, U_{n+1})$
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 7.2: *Description of the available schemes*

Chapter 8

Initial Value Boundary Problem

8.1 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.

8.2 Example using the API

For the sake of clarity, a file called **Test_advection_diffusion_equation.f90** contains an example of how to use this library. For using the API it is necessary to write the sentence **use Initial_Value_Boundary_Problem**.

This example consists of two boundary initial value problems: a 1D problem and a 2D problem. The advection diffusion equation is being solved in both cases. The advection diffusion equation for a 1D grid is:

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

The value given for ν is 0.01. The boundary conditions choosen are: $u(-1) = 1$ and $\frac{\partial u}{\partial x}(1) = 0$ and the initial condition: $u(x, t = 0) = 0$. For a 2D grid:

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

Where $\nu = 0.02$, the initial condition is $u(x, y, t = 0) = \exp(-25x^2 - 25y^2)$ and the boundary conditions are:

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

```fortran
module IVBP_example

use Initial_Value_Boundary_Problem

implicit none

contains

subroutine Test_IVBP1D

    integer, parameter :: Nx = 60, Nt = 1000
    real ::  x(0:Nx)
    real :: Time(0:Nt), U(0:Nt,0:Nx)
    real ::  x0 = -1, xf = 1
    real :: t0 = 0, tf = 3
    integer :: i, Order = 4

    U = 0.

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

    call Initial_Value_Boundary_ProblemS( Time_Domain = Time, x_nodes = x,          &
                                          Order = Order,                            &
                                          Differential_operator = Burgers_equation, &
                                          Boundary_conditions = Burgers_BC, Solution= U )

    call scrmod('reverse')
    call qplot(x, U(100,:), Nx+1)
    call qplot(x, U(Nt,:), Nx+1)

    contains

    !Differential operator

    real function Burgers_equation( x, t, u, ux, uxx)

        real, intent(in) ::  x, t, u, ux, uxx
        real :: nu = 0.01

        Burgers_equation = - u * ux + nu * uxx

    end function

    !Boundary conditions

    real function Burgers_BC(x, t, u, ux)

        real, intent(in) :: x, t, u, ux

        if (x==x0) then
            Burgers_BC = u - 1
        else if (x==xf) then
            Burgers_BC = ux
        else
            write(*,*)  "Error in BC_Burgers"
        endif

    end function

end subroutine

```

```fortran



subroutine Test_IVBP2D

    integer, parameter :: Nx = 10, Ny = 10, 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 = 2
    integer :: i, j, Order = 4

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

    do i=0,Nx
        do j=0, Ny
            U(0, i, j)  =  exp( -25*x(i)**2  -25*y(j)**2 )
        end do
    end do

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

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

    contains

    function Advection_equation( 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 = -U * Ux + nu * ( Uxx + Uyy )

    end function

    function Advection_BC( 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
```

../sources/IVBP_example.f90

```
        end  if

    end  function

end  subroutine

end  module

program  advection_diffusion_equation

use  IVBP_example

   call  Test_IVBP1D
   call  Test_IVBP2D

end  program
```

../sources/IVBP_example.f90

8.3 Initial_Value_Boundary_Problem module

Initial_Value_Boundary_ProblemS for 1D problems

```
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 such as:

$$\begin{aligned} &\mathscr{L}\left(x,\ t,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial^2 U}{\partial x^2}\right) = \frac{\partial U}{\partial t} \\ &f_a\left(U,\ t,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = a \\ &f_b\left(U,\ t,\ \frac{\partial U}{\partial x}\right) = 0 \qquad x = b \end{aligned}$$

Besides, an initial condition must be established: $U(x, t = t_0) = U_0(x)$. 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	real function: $\mathscr{L}\left(x, t, U, U_x, U_{xx}\right)$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f\left(x, t, U, U_x\right)$	in	In this function, the boundary conditions are fixed. The user must include a conditional sentence which sets $f\left(a,\ t,\ U,\ U_x\right) = f_a$ and $f\left(b,\ t,\ U,\ U_x\right) = f_b$.
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	two-dimensional array of reals	out	Contains the solution, $U = U(x,t)$, of the boundary value problem.

Table 8.1: *Description of* ***Initial_Value_Boundary_ProblemS*** *arguments for 1D problems*

Initial_Value_Boundary_ProblemS 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 $[a, b] \times [c, d]$:

$$\mathscr{L}\left(x,\ y,\ t,\ U,\ \frac{\partial U}{\partial x},\ \frac{\partial U}{\partial y},\ \frac{\partial^2 U}{\partial x^2},\ \frac{\partial^2 U}{\partial y^2},\ \frac{\partial^2 U}{\partial x \partial y}\right) = \frac{\partial U}{\partial t}$$

$$f_{x=a}\left(U,\ t,\ \frac{\partial U}{\partial x}\right) = 0 \quad ; \quad f_{x=b}\left(U,\ t,\ \frac{\partial U}{\partial x}\right) = 0$$

$$f_{y=c}\left(U,\ t,\ \frac{\partial U}{\partial y}\right) = 0 \quad ; \quad f_{y=d}\left(U,\ t,\ \frac{\partial U}{\partial y}\right) = 0$$

Besides, an initial condition must be established: $U(x, y, t = t_0) = U_0(x, y)$. The arguments of the subroutine are described in the following table.

Argument	**Type**	**Intent**	**Description**
Time_Domain	vector of reals	in	Time domain where the solution wants to be calculated.
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: $\mathscr{L}\left(x,y,t,U,U_x,U_y,U_{xx},U_{yy},U_{xy}\right)$	in	This function is the differential operator of the boundary value problem.
Boundary_conditions	real function: $f\left(x,y,t,U,U_x,U_y\right)$	in	In this function, the boudary conditions are fixed. The user must use a conditional sentence to do it.
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	three-dimensional array of reals	out	Contains the solution, $U = U(x,y,t)$, of the boundary value problem.

Table 8.2: *Description of* ***Initial_Value_Boundary_ProblemS*** *arguments for 2D problems*

8.3.1 Temporal schemes

The schemes that are available in the library for both, 1D and 2D problems, are listed below. h denotes the time step.

Scheme	**Name (in the code)**	**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	$U_{n+2} = U_n + 2f(x_{n+1}, U_{n+1})$
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 8.3: *Description of the available schemes*