## Contents

1 Getting started with Fortran

1.1 Introduction ............................................. 7
  1.1.1 Historical facts .................................. 7
  1.1.2 General Principles ................................. 7
1.2 Getting started .......................................... 8
1.3 Basic Statements ...................................... 10
  1.3.1 DO, WHILE statements ............................ 10
  1.3.2 IF statements ..................................... 12
  1.3.3 SELECT statements ............................... 12
  1.3.4 Interface procedure ............................... 15
1.4 Pointers and Targets ................................ 16
1.5 Input/Output .......................................... 17
1.6 Tips .................................................. 17

2 Modern Fortran: Object-Oriented Programming ................. 19

2.1 Introduction ........................................... 19
2.2 Type extensions and parameters .......................... 19
2.3 Polymorphism .......................................... 20
2.4 The ASSOCIATE constructor ............................ 20
2.5 The SELECT TYPE command ............................. 21
2.6 Type-Bound Procedures .................................. 21
  2.6.1 Prototype approach: using subroutine/functions pointers .... 22
  2.6.2 Contract approach: ABSTRACT types ................. 22
2.7 Operators ............................................. 22

3 Project: A Finite Element Library .......................... 25

3.1 Introduction ........................................... 25
  3.1.1 The 1D case: code description .................... 25
3.2 Code Design ........................................... 26
  3.2.1 First Specification ................................ 26
  3.2.2 Second specification ............................. 26
  3.2.3 Third specification ............................... 26
3.3 A case study: A Finite Element Method Project ......... 26

A Developer tools .......................................... 27

A.1 Installation and Compilation ............................. 27
A.2 Editors ................................................ 27
A.3 Versioning: svn & git .................................. 27
A.4 CMake ................................................ 27
A.5 Unit-tests ............................................ 27
A.6 Calling Fortran codes from Python: f2py ................. 27
A.7 Documentation ........................................ 27
B  Numerical Integration  
B.1  Quadrature formulae  .................................................. 29

C  Applied Linear Algebra  .................................................. 31
C.1  Kronecker algebra  ...................................................... 31
   C.1.1  Kronecker sum  ...................................................... 34
   C.1.2  Solving $AX + XB = C$  ......................................... 34
   C.1.3  Solving $AXB = C$  .................................................. 34
   C.1.4  Solving $\sum_{i=1}^{r} A_i XB_i = C$  ......................... 34
C.2  Sparse Matrices  ....................................................... 35
C.3  Linear solvers  .......................................................... 35
   C.3.1  Direct solvers  ...................................................... 35
   C.3.2  Iterative solvers  ................................................... 35
Introduction

How to use this document

This document is not a reference manual for Fortran, there are many good books for that [7, 6, 2]. You will not, and you don’t, need to learn all Fortran features or capabilities. However you will learn, step by step, new concepts through some practical examples. Think of this document as a way to start writing clean codes. Sometimes, we will intentionally write bad codes, as if we are writing them for the first time. We will give bad naming for our variables, use some wrong design. But through these notes we will learn from our mistakes. We will often refer to our commandments [3]. These notes are not finished yet and all remarks are welcome.

Getting and Compiling the source code

All Fortran codes that are presented in these notes can be found on github. If you have git already installed, you can run the following lines in your Terminal:

```
1. git clone git@github.com:ratnania/lessons.git
```

this will create a directory lessons that contains a CMakeLists.txt and source codes for each chapter. Using cmake, you need to create a build directory and run the following lines:

```
1. mkdir -p build
2. cd build
3. # cmake must points to the path of CMakeLists.txt file
4. cmake ..
5. make
```
Chapter 1

Getting started with Fortran

1.1 Introduction

In this lecture, we will learn how to write Modern Fortran codes. By modern, we mean Object-Oriented Programming. But why using an old language like Fortran, while we can use C++?

Many reasons for that. First of all

Legacy codes Through the years, the scientific community wrote a lot of good codes.

OOP What a mathematician or a physicist is asking from Fortran and the OOP, is not for sure what the C++ can offer!

In these notes, you will not learn all the features of Fortran. For more details about the language capabilities, we refer the reader to the books [7, 2, 6, 1]. However, we will introduce, step by step, the features we need. Remember to always keep it simple (and stupid).

1.1.1 Historical facts

Fortran was one of the first high level languages developed for computers. It stands for Mathematical FORmula TRANslating System. Fortran was started by John Backus at IBM in 1954. The first release of the compiler was in 1957. Since then, many versions have been released: Fortran 66 (1966), Fortran 77 (1977), Fortran 90 (1991), Fortran 95 (1997), Fortran 2003 (2004) and finally Fortran 2008 (2010).

1.1.2 General Principles

Coding should be less important than modeling. The sustainability of a code depends on how it is well designed. Usually, it’s not straightforward to get a good model (and not the best, if it exists). It also takes a lot of time. However, it’s not a waste of time. In fact, once the code begins to get mature, it’s much more easy to extend it. However, you should follow some rules, such as using a Test Driven Development approach, to test every little part of it, in order to prevent from a regression.
1.2 Getting started

Let’s start with the most classical example. Open your favorite Editor and copy paste the following lines:

```fortran
PROGRAM EX_1
IMPLICIT NONE
PRINT *, 'Hello From Fortran !'
END PROGRAM EX_1
```

You can compile the last example using:
```
gfortran ex_1.F90 -o ex_1.exe
```

You can also use the `CMake` tool. All our examples make use of it. Now let’s create a `procedure/subroutine` that prints the previous message.

```fortran
PROGRAM EX_2
IMPLICIT NONE
CALL MY_PRINT()
CONTAINS
SUBROUTINE MY_PRINT()
IMPLICIT NONE
PRINT *, 'Hello From Fortran Procedure !'
END SUBROUTINE MY_PRINT
END PROGRAM EX_2
```

Now let’s print some basic declarations. For the moment, we will only cover integers, reals, booleans, constants and arrays.

```fortran
PROGRAM EX_3
IMPLICIT NONE
CALL MY_PRINT_1()
CALL MY_PRINT_2()
CONTAINS
SUBROUTINE MY_PRINT_1()
IMPLICIT NONE
INTEGER :: i
INTEGER, PARAMETER :: J = 1
REAL(KIND=8) :: x
REAL(KIND=4) :: y
PRINT *, "<<< Enter MY PRINT_1"
DO i = 1 , J
x = 2.0 * i
y = 2.0 * i
PRINT *, "J : " , J
PRINT *, "I : " , i
PRINT *, "X : " , x
PRINT *, "Y : " , y
PRINT *, "<<< Leave MY PRINT_1"
END SUBROUTINE MY_PRINT_1
```

```fortran
SUBROUTINE MY_PRINT_2()
IMPLICIT NONE
INTEGER, PARAMETER :: N = 5
INTEGER :: i
INTEGER, DIMENSION(5) :: arr_i
REAL(KIND=8), DIMENSION(N) :: arr_x
REAL(KIND=8), DIMENSION(N) :: arr_y
PRINT *, "<<< Enter MY PRINT_2"
do i = 1 , N
arr_i = i
```

Now, let’s consider the classical case when we want to model an unknown (temperature, pressure, ...). Usually, for instance if we use a Galerkin-Ritz method, our unknown will live in a vectorial space and then knowing the basis, one only needs to have the expansion coefficients over it. For this purpose, we introduce the concept of `module` which plays the role of the C++ `namespace`. We introduce also how to construct a data structure using the keyword `type`. The following example describes a first attempt to model an unknown. We call it a `field` and it relies on the use of the `implicit constructor` of Fortran.

```
MODULE MODULE_FIELD
IMPLICIT NONE
PRIVATE :: FIELD, FIELD_DIMENSION, FIELD_PRINT
PUBLIC : : FIELD, FIELD_DIMENSION, FIELD_PRINT
END TYPE FIELD
CONTAINS
FUNCTION FIELD_DIMENSION(self) RESULT(val)
IMPLICIT NONE
TYPE(FIELD), INTENT(IN) :: self
INTEGER :: val
val = self % n_dim
END FUNCTION FIELD_DIMENSION

FUNCTION FIELD_SIZE(self) RESULT(val)
IMPLICIT NONE
TYPE(FIELD), INTENT(IN) :: self
INTEGER :: val
val = self % n_size
END FUNCTION FIELD_SIZE

SUBROUTINE FIELD_PRINT(self)
IMPLICIT NONE
TYPE(FIELD), INTENT(IN) :: self
PRINT *, 'Field : dim', self % n_dim
PRINT *, 'size', self % n_size
END SUBROUTINE FIELD_PRINT
END MODULE MODULE_FIELD
```

```
PROGRAM EX_4
USE MODULE_FIELD
IMPLICIT NONE
TYPE(FIELD) :: U
U = FIELD(1, 10) ! Use the implicit constructor
CALL FIELD_PRINT(U) ! Call a class subroutine
END PROGRAM EX_4
```
CHAPTER 1. GETTING STARTED WITH FORTRAN

Remark 1.2.1 Later, we will see how to add the allocation of the coefficients array directly through the implicit constructor.

1.3 Basic Statements

Through this section, we will present some basic statements through modeling a matrix and the matrix-vector product operation, in the context of Finite Differences.

1.3.1 DO, WHILE statements

Let consider a matrix $M \in \mathcal{M}_{m \times n}(\mathbb{R})$ and vector $x \in \mathbb{R}^m$. The following code implements the matrix-vector product:

```fortran
PROGRAM EX_5
IMPLICIT NONE
INTEGER, PARAMETER :: M = 5
INTEGER, PARAMETER :: N = 5
REAL(KIND=8), DIMENSION(M,N) :: Mat
REAL(KIND=8), DIMENSION(N) :: x
REAL(KIND=8), DIMENSION(M) :: y
INTEGER :: i
INTEGER :: j

Mat = 0.0
DO j = 1, N
  DO i = 1, M
    Mat(i,j) = (i+j) * 1.0
  END DO
END DO

x = 1.0
CALL MATRIX_PRODUCT_VECTOR_DENSE(Mat, x, y)

PRINT *, "y = M x :", y
CONTAINS

SUBROUTINE MATRIX_PRODUCT_VECTOR_DENSE(A, x, y)
IMPLICIT NONE
REAL(KIND=8), DIMENSION(:,::), INTENT(IN) :: A
REAL(KIND=8), DIMENSION(:,::), INTENT(IN) :: x
REAL(KIND=8), DIMENSION(:,::), INTENT(INOUT) :: y
INTEGER :: M
INTEGER :: N
INTEGER :: i
INTEGER :: j

M = SIZE(A,1)
N = SIZE(A,2)
y = 0.0
DO i = 1, M
  DO j = 1, N
    y(i) = y(i) + A(i,j) * x(j)
  END DO
END DO

CALL MATRIX_PRODUCT_VECTOR_DENSE(Mat, x, y)
END PROGRAM EX_5
```
Let us now consider the case of a sparse matrix in *CSR* format. The following code describes the matrix-vector product:

```fortran
1 PROGRAM EX_6
2 IMPLICIT NONE
3 INTEGER, PARAMETER :: N = 4
4 INTEGER, PARAMETER :: NNZ = 9
5 REAL(KIND=8) , DIMENSION(N) :: x
6 REAL(KIND=8) , DIMENSION(N) :: y
7 REAL(KIND=8) , DIMENSION(NNZ) :: A
8 INTEGER(KIND=8) , DIMENSION(NNZ) :: JA
9 INTEGER(KIND=4) , DIMENSION(N+1) :: IA
10 ! ...
11 JA(1) = 1 ; A(1) = 1.0
12 JA(2) = 3 ; A(2) = 2.0
13 JA(3) = 4 ; A(3) = 3.0
14 JA(4) = 2 ; A(4) = 4.0
15 JA(5) = 2 ; A(5) = 5.0
16 JA(6) = 3 ; A(6) = 6.0
17 JA(7) = 2 ; A(7) = 7.0
18 JA(8) = 3 ; A(8) = 8.0
19 JA(9) = 4 ; A(9) = 9.0
20 ! ...
21 IA(1) = 1
22 IA(2) = 4
23 IA(3) = 5
24 IA(4) = 7
25 IA(5) = 10
26 ! ...
27 x = 1.0
28 ! ...
29 CALL MATRIX_PRODUCT_VECTOR_CSR(IA, JA, A, x, y)
30 ! ...
31 PRINT *, "y = M x : " , y
32 CONTAINS
33 ! ...
34 SUBROUTINE MATRIX_PRODUCT_VECTOR_CSR(IA, JA, A, x, y)
35 IMPLICIT NONE
36 REAL(KIND=8) , DIMENSION(:) , INTENT(IN) :: A
37 INTEGER(KIND=8) , DIMENSION(:) , INTENT(IN) :: JA
38 REAL(KIND=8) , DIMENSION(:) , INTENT(INOUT) :: x
39 REAL(KIND=8) , DIMENSION(:) , INTENT(INOUT) :: y
40 ! LOCAL
41 INTEGER :: N
42 INTEGER :: i
43 INTEGER :: k_max
44 INTEGER :: k_min
45 ! ...
46 N = SIZE(IA,1) - 1
47 ! ...
48 y = 0.0
49 DO i = 1, N
50 k_min = IA(i)
51 k_max = IA(i+1) - 1
52 y(i) = DOT_PRODUCT(A(k_min:k_max), X(JA(k_min:k_max)))
53 END DO
54 ! ...
55 END SUBROUTINE MATRIX_PRODUCT_VECTOR_CSR
56 ! ...
57 END PROGRAM EX_6

more details can be found in [11].
1.3.2 IF statements

We now implement the matrix that discretizes the Poisson problem in 1D using finite differences with homogeneous Dirichlet boundary conditions:

```fortran
! The laplacian looks like :
> 2 -1 0
> -1 2 -1 0
> 0 -1 2 -1 0
1/dx^2

INTEGER, PARAMETER :: N = 5
REAL(KIND=8), DIMENSION(N,N) :: Mat
INTEGER :: i
REAL(KIND=8) :: dx

DO i = 1, N
  IF (i == 1) THEN
    Mat(i, i) = 2.0
    Mat(i, i+1) = -1.0
  ELSEIF (i == N) THEN
    Mat(i, i-1) = -1.0
    Mat(i, i) = 2.0
  ELSE
    Mat(i, i-1) = -1.0
    Mat(i, i) = 2.0
    Mat(i, i+1) = -1.0
  END IF
END DO
Mat = (1./dx**2) * Mat
```

1.3.3 SELECT statements

Now, let’s go back to our matrix-vector product. We have implemented two subroutines: `MATRIX_PRODUCT_VECTOR_DENSE` and `MATRIX_PRODUCT_VECTOR_CSR` for a dense and CSR matrix. A first attempt to model a matrix will be to consider a data-structure that contains all data needed for a dense/CSR matrix. The following code describes this data-structure and makes use of the SELECT keyword to take the right matrix-vector product implementation depending on the matrix type.

```fortran
TYPE MATRIX
INTEGER :: matrix_type
INTEGER :: N
INTEGER :: NNZ
REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dense_A
REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: csr_A
INTEGER(KIND=4), DIMENSION(:), ALLOCATABLE :: csr_IA
```

1.3. BASIC STATEMENTS

 INTEGER(KIND=8), DIMENSION(:) , ALLOCATABLE :: csr_JA
 END TYPE MATRIX

 CONTAINS

 SUBROUTINE MATRIX_CREATE_DENSE(self, Mat)
 IMPLICIT NONE
 TYPE(MATRIX), INTENT(INOUT) :: self
 REAL(KIND=8), DIMENSION(:, :) , INTENT(IN) :: Mat
 ! LOCAL
 IF ( SIZE(Mat, 1) .NE. SIZE(Mat, 2) ) THEN
 STOP ' Only square matrices can be used'
 END IF
 self % N = SIZE(Mat, 1)
 ALLOCATE(self % dense_A(self % N, self % N))
 self % dense_A = Mat
 self % matrix_type = 0
 END SUBROUTINE

 SUBROUTINE MATRIX_FREE_DENSE(self)
 IMPLICIT NONE
 TYPE(MATRIX), INTENT(INOUT) :: self
 ! LOCAL
 DEALLOCATE(self % dense_A)
 END SUBROUTINE

 SUBROUTINE MATRIX_PRODUCT_VECTOR_DENSE(A, x, y)
 IMPLICIT NONE
 REAL(KIND=8), DIMENSION(:, :) , INTENT(IN) :: A
 REAL(KIND=8), DIMENSION(:) , INTENT(IN) :: x
 REAL(KIND=8), DIMENSION(:) , INTENT(INOUT) :: y
 ! LOCAL
 INTEGER :: M
 INTEGER :: N
 INTEGER :: i
 INTEGER :: j
 M = SIZE(A, 1)
 N = SIZE(A, 2)
 y = 0.0
 DO i = 1, M
 DO j = 1, N
 y(i) = y(i) + A(i, j) * x(j)
 END DO
 END DO
 END SUBROUTINE

 SUBROUTINE MATRIX_CREATE_CSR(self, IA, JA, A)
 IMPLICIT NONE
 TYPE(MATRIX), INTENT(INOUT) :: self
 INTEGER(KIND=8), DIMENSION(:) , INTENT(IN) :: IA
 INTEGER(KIND=4), DIMENSION(:) , INTENT(IN) :: JA
 REAL(KIND=8) , DIMENSION(:) , INTENT(IN) :: A
 ! LOCAL
 self % N = SIZE(IA, 1) - 1
 self % NNZ = SIZE(JA, 1)
 ALLOCATE(self % csr_IA(self % N + 1))
 ALLOCATE(self % csr_JA(self % NNZ))
 ALLOCATE(self % csr_A (self % NNZ))
 self % csr_IA = IA
 self % csr_JA = JA

self % csr_A = A
self % matrix_type = 1

END SUBROUTINE MATRIX_CREATE_CSR

! . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

SUBROUTINE MATRIX_FREE_CSR(self)
IMPLICIT NONE
TYPE(MATRIX), INTENT(INOUT) :: self
! LOCAL
DEALLOCATE(self % csr_IA)
DEALLOCATE(self % csr_JA)
DEALLOCATE(self % csr_A)

END SUBROUTINE MATRIX_FREE_CSR

! . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

SUBROUTINE MATRIX_PRODUCT_VECTOR_CSR(IA, JA, A, x, y)
IMPLICIT NONE
REAL(KIND=8) , DIMENSION ( : ) , INTENT( IN ) :: A
INTEGER(KIND=8) , DIMENSION ( : ) , INTENT( IN ) :: JA
INTEGER(KIND=4) , DIMENSION ( : ) , INTENT( IN ) :: IA
REAL(KIND=8) , DIMENSION ( : ) , INTENT(IN) :: x
REAL(KIND=8) , DIMENSION ( : ) , INTENT(INOUT) :: y
! LOCAL
INTEGER :: N
INTEGER :: i
INTEGER :: k_max
INTEGER :: k_min

N = SIZE(IA, 1) − 1
y = 0.0
DO i = 1, N
k_min = IA(i)
k_max = IA(i+1) − 1
y(i) = DOT_PRODUCT(A(k_min:k_max), X(JA(k_min:k_max)))
END DO

END SUBROUTINE MATRIX_PRODUCT_VECTOR_CSR

! . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

SUBROUTINE MATRIX_PRODUCT_VECTOR_CSR(self, x, y)
IMPLICIT NONE
TYPE(MATRIX), INTENT(IN) :: self
REAL(KIND=8) , DIMENSION ( : ) , INTENT(IN) :: x
REAL(KIND=8) , DIMENSION ( : ) , INTENT(INOUT) :: y
! LOCAL
SELECT CASE(self % matrix_type)
CASE(0)
CALL MATRIX_PRODUCT_VECTOR_DENSE(self % dense_A, x, y)
CASE(1)
CALL MATRIX_PRODUCT_VECTOR_CSR(self % csr_IA, self % csr_JA, self % csr_A ← x, y)
CASE DEFAULT
PRINT *, "MATRIX_PRODUCT_VECTOR: matrix-type not yet implemented"
END SELECT

END SUBROUTINE MATRIX_PRODUCT_VECTOR

END MODULE MATRIX_UTILITIES

PROGRAM EX_8
USE MATRIX_UTILITIES
IMPLICIT NONE
INTEGER , PARAMETER :: M = 5
INTEGER , PARAMETER :: N = 4
INTEGER , PARAMETER :: NNZ = 9
REAL(KIND=8) , DIMENSION(NNZ) :: A
INTEGER(KIND=8) , DIMENSION(NNZ) :: JA
REAL(KIND=8) , DIMENSION(M+1) :: IA
REAL(KIND=8) , DIMENSION(M,M) :: Mat
1.3. BASIC STATEMENTS

```fortran
REAL(KIND=8), DIMENSION(N) :: x1
REAL(KIND=8), DIMENSION(N) :: y1
REAL(KIND=8), DIMENSION(M) :: x2
REAL(KIND=8), DIMENSION(M) :: y2
INTEGER :: i
INTEGER :: j
TYPE(MATRIX) :: csr
TYPE(MATRIX) :: dense

! . . .
JA(1) = 1   ; A(1) = 1.0
JA(2) = 3   ; A(2) = 2.0
JA(3) = 4   ; A(3) = 3.0
JA(4) = 2   ; A(4) = 4.0
JA(5) = 2   ; A(5) = 5.0
JA(6) = 3   ; A(6) = 6.0
JA(7) = 2   ; A(7) = 7.0
JA(8) = 3   ; A(8) = 8.0
JA(9) = 4   ; A(9) = 9.0

! . . .
IA(1) = 1
IA(2) = 4
IA(3) = 5
IA(4) = 7
IA(5) = 10

DO j = 1, N
    DO i = 1, M
        Mat(i,j) = (i+j) * 1.0
    END DO
END DO

CALL MATRIX_CREATE_CSR(csr, IA, JA, A)
CALL MATRIX_PRODUCT_VECTOR(csr, x1, y1)
CALL MATRIX_FREE_CSR(csr)

CALL MATRIX_CREATE_DENSE(dense, Mat)
CALL MATRIX_PRODUCT VECTOR(dense, x2, y2)
CALL MATRIX_FREE_DENSE(dense)

PRINT *, "y2 = M2 x2 : ", y2
PRINT *, "y1 = M1 x1 : ", y1
```

1.3.4 Interface procedure

In order to enhance the readability of our last code, we may want to have one single subroutine to create a matrix object no matter what type it is. This can be done through the INTERFACE statement.

```fortran
INTERFACE MATRIX_CREATE
    MODULE PROCEDURE MATRIX_CREATE_DENSE, MATRIX_CREATE_CSR
END INTERFACE
```

Now depending on the actual arguments of your CALL, the compiler will pick the right subroutine.
### 1.4 Pointers and Targets

Unlike the C language for which a pointer contains the object address, in Fortran a pointer is more an alias, usually called reference. It is somehow a high level abstraction of the classical reference in the C language. A Fortran pointer has one of the following states:

- Undefined
- Null
- Associated

```fortran
REAL, POINTER :: p1
INTEGER, DIMENSION (:), POINTER :: p2

TYPE box
  INTEGER i_1
  INTEGER i_2
END TYPE

TYPE (box), POINTER :: p3
```

When a pointer is associated, it should give the reference of a Target. This leads to the following rules:

```fortran
REAL, POINTER :: p1
REAL, POINTER :: p2
REAL, TARGET :: x

x = 2.0

p1 => x  ! OK
p2 => p1  ! OK
! x = p1  ! KO

x = 1.0

PRINT *, p2
```

A pointer can be the reference or alias of more complex objects.

```fortran
REAL, DIMENSION (10,20), TARGET :: a
REAL, DIMENSION (:), POINTER :: p
integer :: i

a = RESHAPE ( source =(/ ( i, i=1,200) /) , shape =shape( a) )

READ (*, *) i
! p points now to a(1), a(4), a(7), a(10)
PRINT *, p (3)
```

A pointer can also be allocated. A distinction between non-allocated and allocated pointers should be done, using for example the prefix `ptr_`.

```fortran```
1.5 Input/Output

1.6 Tips

Let’s go back to our dense matrix. The following example shows that you must pay attention to the order of loops. Indeed, there is an overhead when setting nested loops. The loop range must be increasing from top to down.

```fortran
SUBROUTINE DO_FAST(x, n, m)
  IMPLICIT NONE
  INTEGER, DIMENSION(n,n), INTENT(INOUT) :: x
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(IN) :: m
  ! LOCAL
  INTEGER :: i, j
  x = 0
  DO j=1, m
    DO i=1, n
      x(i,j) = i+j
    END DO
  END DO
END SUBROUTINE DO_FAST

SUBROUTINE DO_SLOW(x, n, m)
  IMPLICIT NONE
  INTEGER, DIMENSION(n,n), INTENT(INOUT) :: x
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(IN) :: m
  ! LOCAL
  INTEGER :: i, j
  x = 0
  DO i=1, n
    DO j=1, m
      x(i,j) = i+j
    END DO
  END DO
END SUBROUTINE DO_SLOW

PROGRAM EXAMPLE
  IMPLICIT NONE
  INTEGER, PARAMETER :: n_loops = 10
  INTEGER :: n
  INTEGER :: m
  INTEGER :: i
  INTEGER, DIMENSION(:, :) , ALLOCATABLE :: x
  REAL :: start, finish
  ALLOCATE(x(n,m))
  CALL CPU_TIME(start)
  DO i=1, n_loops
    CALL DO_FAST(x, n, m)
  END DO
  CALL CPU_TIME(finish)
  print *, "Done with fast."); (finish-start)/n_loops
  CALL CPU_TIME(start)
  DO i=1, n_loops
    CALL DD_SLOW(x, n, m)
  END DO
  CALL CPU_TIME(finish)
  print *, "Done with slow."); (finish-start)/n_loops
END PROGRAM EXAMPLE
```

In this example, we clearly see the impact of the overhead when \( n > m \):
Let us now change the compilation options. First we use the option -O1

```bash
ahmed $ ./a.out
> Done with fast.
Time = 0.366 seconds.
> Done with slow.
Time = 0.794 seconds.
```

Using the option -O2

```bash
ahmed $ gfortran -O2 ex_9.F90
ahmed $ ./a.out
> Done with fast.
Time = 0.184 seconds.
> Done with slow.
Time = 0.434 seconds.
```

Using the option -O3

```bash
ahmed $ gfortran -O3 ex_9.F90
ahmed $ ./a.out
> Done with fast.
Time = 0.174 seconds.
> Done with slow.
Time = 0.251 seconds.
```

Nowadays compilers are smart enough to optimize our loops but not enough to get the best performance. Always pay attention to the range of nested loops.
Chapter 2

Modern Fortran: Object-Oriented Programming

2.1 Introduction

We will introduce the basic concepts of OOP through our matrix example. As we can notice in the last code (see subsection 1.3.3), we have now two kinds of matrices: dense and CSR. Before going further, let’s start by cleaning our code a little. We split the data-structure into two types: DENSE and CSR. This leads to the following definitions:

```fortran
1 TYPE MATRIX_DENSE
2   INTEGER :: N
3   REAL(KIND=8) , DIMENSION(:,:) , ALLOCATABLE :: A
4 END TYPE MATRIX_DENSE

5 TYPE MATRIX_CSR
6   INTEGER :: N
7   INTEGER :: NNZ
8   REAL(KIND=8) , DIMENSION(:) , ALLOCATABLE :: A
9   INTEGER(KIND=4) , DIMENSION(:) , ALLOCATABLE :: IA
10  INTEGER(KIND=8) , DIMENSION(:) , ALLOCATABLE :: JA
11 END TYPE MATRIX_CSR
```

The OOP paradigm relies on 3 (among others) fundamental ideas:

- Encapsulation
- Inheritance
- Polymorphism

These notions need the concept of an object or class rather than a structure/derived-type.

What is an object?

When you don’t want the external user to access or modify directly some private data in your structure then using a data-structure is meaningless and it’s not the right way to implement your idea. Objects should hide their data behind some abstractions and only expose functions or subroutines that operate on that data. This is a part of what is called Programming by contract.

2.2 Type extensions and parameters

The first thing would be to create a MATRIX object and then make the dense and CSR matrices inherit from it:
CHAPTER 2. MODERN FORTRAN: OBJECT-ORIENTED PROGRAMMING

```fortran
TYPE MATRIX
  INTEGER :: N
END TYPE MATRIX

TYPE, EXTENDS(MATRIX) :: MATRIX_DENSE
  REAL(KIND=8), DIMENSION(:,::), ALLOCATABLE :: coefficients
END TYPE MATRIX_DENSE

TYPE, EXTENDS(MATRIX) :: MATRIX_CSR
  INTEGER :: NNZ
  INTEGER(KIND=4), DIMENSION(:), ALLOCATABLE :: IA
  INTEGER(KIND=8), DIMENSION(:), ALLOCATABLE :: JA
  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: A
END TYPE MATRIX_CSR
```

Now, automatically, the objects `MATRIX_DENSE` and `MATRIX_CSR` will inherit the type parameter `N` which represents the number of rows and columns (we only consider square matrices).

### Type parameters

#### 2.3 Polymorphism

The idea behind the extension of the matrix type, is to collect objects that share some specific behavior for example. In the case of a matrix, what we usually want to do is: matrix-vector product, solve a linear system, compute some norms, ... . Through *polymorphism* the data type of your variable may vary at run time. A typical use would be the following:

```fortran
TYPE(MATRIX_DENSE), TARGET :: dense
TYPE(MATRIX_CSR), TARGET :: csr
CLASS(MATRIX), POINTER :: ptr_matrix

! ... take the dense matrix and do some stuff
ptr_matrix => dense
! ... 

! ... conversion to csr format and then do what you want
ptr_matrix => csr
! ...
```

**Remark 2.3.1** Two very important remarks:

- The use of the `TARGET` attribut for both `DENSE` and `CSR` variables,
- The variables `DENSE` and `CSR` are declared with the keyword `TYPE` while the matrix pointer, the polymorphic variable, is defined with the keyword `CLASS`.

⚠️ Don’t put a `CLASS` object inside your data-structure.

### Unlimited polymorphism

#### 2.4 The `ASSOCIATE` constructor

To enhance the readability of Fortran codes, we can use the `ASSOCIATE` construct that allows us to associate a name with a variable or the value of an expression within the scope of the considered block.
2.5. The SELECT TYPE command

In order to make our code much more clear, we would like to have a single function/subroutine when the action of the matrix object are the same, no matter if it is dense or csr. For this reason, we will use polymorphic variables and in addition we will need to select the type dynamically. This can be done using the keyword SELECT TYPE:

```fortran
SUBROUTINE MATRIX_FREE(self)
IMPLICIT NONE
CLASS(MATRIX), INTENT(INOUT) :: self
!
SELECT TYPE (self)
CLASS IS (MATRIX_DENSE)
CALL MATRIX_FREE_DENSE(self)
CLASS IS (MATRIX_CSR)
CALL MATRIX_FREE_CSR(self)
CLASS DEFAULT
STOP 'MATRIX_FREE: unexpected type for self object!'
END SELECT
END SUBROUTINE MATRIX_FREE
```

The same strategy can be applied for the MATRIX_PRODUCT_VECTOR subroutine.

2.6 Type-Bound Procedures

Our code will be better, if we can associate a given procedure like CREATE, FREE or DOT for the matrix-vector product. In Fortran 2003 this is possible using the Type bounding procedures. It is obvious that the following code is much more clear, also through the definition of our object, we can see what are its properties. It is normal to ask a matrix object to provide subroutines to create, free and apply a matrix-vector product. This is again a way of expressing the contract between the developer and the user.

```fortran
TYPE MATRIX
  INTEGER :: N
END TYPE MATRIX

TYPE, EXTENDS(MATRIX) :: MATRIX_DENSE
  REAL(KIND=8) , DIMENSION(:, :), ALLOCATABLE :: coefficients
CONTAINS
  PROCEDURE :: CREATE => MATRIX_CREATE_DENSE
  PROCEDURE :: FREE => MATRIX_FREE_DENSE
  PROCEDURE :: DOT => MATRIX_PRODUCT_VECTOR_DENSE
END TYPE MATRIX_DENSE

TYPE, EXTENDS(MATRIX) :: MATRIX_CSR
  INTEGER :: NNZ
  INTEGER(KIND=4), DIMENSION(:, :), ALLOCATABLE :: IA
END TYPE MATRIX_CSR
```
CHAPTER 2. MODERN FORTRAN: OBJECT-ORIENTED PROGRAMMING

```fortran
INTEGER(KIND=8), DIMENSION(:), ALLOCATABLE :: JA
REAL(KIND=8) , DIMENSION(:), ALLOCATABLE :: A
CONTAINS
  PROCEDURE :: CREATE => MATRIX_CREATE_CSR
  PROCEDURE :: FREE  => MATRIX_FREE_CSR
  PROCEDURE :: DOT   => MATRIX_PRODUCT_VECTOR_CSR
END TYPE MATRIX_CSR
```

Do not use the PASS statement. The resulting code is less clear.

2.6.1 Prototype approach: using subroutine/functions pointers

2.6.2 Contract approach: ABSTRACT types

Now that we decided that any matrix object must provide the subroutines to create, free and apply a matrix-vector product, we can enforce it inside the definition of the parent object MATRIX. To do so, we need to add the ABSTRACT argument in its definition. The following code implements these ideas. However, we still can not add the CREATE subroutine into the ABSTRACT type.

```fortran
TYPE, ABSTRACT :: MATRIX
  INTEGER :: N
CONTAINS
  PROCEDURE(MATRIX_FREE), DEFERRED :: FREE
  PROCEDURE(MATRIX_PRODUCT_VECTOR), DEFERRED :: DOT
END TYPE MATRIX

ABSTRACT INTERFACE
  SUBROUTINE MATRIX_FREE(self)
  IMPORT MATRIX
  IMPLICIT NONE
  CLASS(MATRIX), INTENT(INOUT) :: self
END SUBROUTINE MATRIX_FREE
END INTERFACE

ABSTRACT INTERFACE
  SUBROUTINE MATRIX_PRODUCT_VECTOR(self, x, y)
  IMPORT MATRIX
  IMPLICIT NONE
  CLASS(MATRIX), INTENT(IN) :: self
  REAL(KIND=8), DIMENSION(:), INTENT(IN) :: x
  REAL(KIND=8), DIMENSION(:), INTENT(INOUT) :: y
END SUBROUTINE MATRIX_PRODUCT_VECTOR
END INTERFACE
```

An ABSTRACT type is a derived type that cannot be instantiated.

2.7 Operators

The aim of this section is to make our code more clear through the definition of some basic operators: +, −, ∗, == , != and ask our objects to provide these operations. Before doing so, we need to ask ourselves some questions:

1. what do we mean by the matrix $A$ is equal to the matrix $B$?
2. although we understand what to do when adding two \textit{csr/dense} matrices, can we add a \textit{csr} matrix to a \textit{dense} matrix? what about the opposite?
Chapter 3

Project: A Finite Element Library

3.1 Introduction

3.1.1 The 1D case: code description

We describe here the initial Fortran code for our project. This code solves the 1D Poisson equation. It has two files fem1d_lagrange.F90 and main.F90. The first file contains the following routines:

- **FEM1D_LAGRANGE_STIFFNESS** This subroutine assembles the stiffness and mass matrices and the load.
- **LAGRANGE_DERIVATIVE** evaluates the Lagrange basis derivative.
- **LAGRANGE_VALUE** evaluates the Lagrange basis polynomials.
- **LAGRANGE_SET** sets abscissas and weights for Gauss-Legendre quadrature.
- **R8MAT_FS** factors and solves a system with one right hand side.
- **R8VEC_LINSPACE** creates a vector of linearly spaced values.
- **R8VEC_PRINT** prints an R8VEC.

while the second file contains the following ones:

- **MAIN** the main program
- **LEGENDRE_SET_TEST** tests LEGENDRE_SET.
- **LAGRANGE_VALUE_TEST** tests LAGRANGE_VALUE.
- **LAGRANGE_DERIVATIVE_TEST** tests LAGRANGE_DERIVATIVE.
- **FEM1D_LAGRANGE_STIFFNESS_TEST** tests FEM1D_LAGRANGE_STIFFNESS.

**F** the right hand side function.

**EXACT** the exact solution.

The original code can be found [here](#).
CHAPTER 3. PROJECT: A FINITE ELEMENT LIBRARY

Compilation and execution

In order to compile these files, you need a Fortran compiler (gfortran, ifort, ...). If you are using gfortran, then you can run the following commands:

```
1. gfortran -c fem1d_lagrange.F90
2. gfortran main.F90 fem1d_lagrange.o -o fem.exe
```

Execution can be done by

```
./fem.exe
```

3.2 Code Design

Rather than presenting the specification of our final code, we will present and introduce them one by one, and continuously change, update and redesign our code. We want the reader to face a typical situation that we encounter: we only keep writing and rewriting our codes!

3.2.1 First Specification

3.2.2 Second specification

3.2.3 Third specification

3.3 A case study: A Finite Element Method Project

Through this lecture, we will present the basic concepts from Object-Oriented Programming and apply them to the implementation of a Finite Element Method Library. We decided to only consider the case of B-Splines basis functions as they allow us to create different spaces with a minimal cost. The code can be extended easily to treat other kind of basis functions.

The first notion we need is the **Discrete Vectorial Functions Space**. It is an abstract notion that we will extend later in the case of the well known spaces $H^1, H(\text{div}), H(\text{rot}), L^2$.

```
1. TYPE DEF_ABSTRACT_SPACE
2.   INTEGER :: oi_dim
3. END TYPE DEF_ABSTRACT_SPACE
```

A function that lives in the discrete space will be called a **field**. We can therefore define an abstract field as well as specific fields depending on the spaces $H^1, H(\text{div}), H(\text{rot}), L^2$.

```
1. TYPE DEF_ABSTRACT_FIELD
2.   INTEGER :: oi_dim
3. END TYPE DEF_ABSTRACT_FIELD
```
Appendix A

Developer tools

A.1 Installation and Compilation
A.2 Editors
A.3 Versioning: svn & git
A.4 CMake
A.5 Unit-tests
A.6 Calling Fortran codes from Python: f2py
A.7 Documentation
Appendix B

Numerical Integration

B.1 Quadrature formulae

For more details, we refer the reader to the book [9]. In figure (Fig. B.1), we show the quadrature points formulae for a mesh \( \{x_0 = 0, x_1 = \frac{1}{4}, x_2 = \frac{1}{2}, x_3 = \frac{3}{4}, x_4 = 1\} \) for a quadratic polynomial degree.

![Gauss quadrature points on [0, 0.25, 0.75, 1] for polynomial degree of 2](image)

**Figure B.1:** The quadrature points formulae for a mesh \( \{x_0 = 0, x_1 = \frac{1}{4}, x_2 = \frac{1}{2}, x_3 = \frac{3}{4}, x_4 = 1\} \) for a quadratic polynomial degree.
Appendix C

Applied Linear Algebra

C.1 Kronecker algebra

In this section, we present an overview about an interesting subject, which is the Kronecker Algebra, and which will be of a big interest in the Fast-IGA approach. Most of the presented results were taken from [4, 10].

Definition C.1.1 (The vec operator) Let \( A = (a_{ij}) \in M_{n \times m} \), the vec operator is defined as,

\[
\text{vec} \ A = \begin{pmatrix} A_{:,1} \\ \vdots \\ A_{:,m} \end{pmatrix} \in \mathbb{R}^{mn}
\]  

which is simply a vector composed by stacking all the columns of \( A \). Where we denote \( A_{:,j} \) the \( j \)th column of \( A \). We also define the inverse operator of vec by,

\[
A = \text{vec}^{-1} \text{vec} \ A \]  

Definition C.1.2 (Kronecker product) Let \( A = (a_{ij}) \in M_{m \times n} \) and \( B = (b_{ij}) \in M_{r \times s} \) be two matrices. The Kronecker product of \( A \) and \( B \), denoted by \( A \otimes B \in M_{mr \times ns} \), defines the following matrix:

\[
A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\
 a_{21}B & a_{22}B & \cdots & a_{2n}B \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix}
\]  

Example

Let

\[
A = \begin{pmatrix} a_{11} & a_{12} \\
 a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\
 b_{21} & b_{22} \end{pmatrix}
\]

then their Kronecker product is,

\[
A \otimes B = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\
 a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\
 a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\
 a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}
\]
APPENDIX C. APPLIED LINEAR ALGEBRA

Properties

Proposition C.1.3 If $\alpha$ is a scalar, then
\[
A \otimes \alpha B = \alpha A \otimes B
\]
(C.1.5)

Proposition C.1.4 We have,
\[
(A + B) \otimes C = A \otimes C + B \otimes C
\]
(C.1.6)

\[
A \otimes (B + C) = A \otimes B + A \otimes C
\]
(C.1.7)

Proposition C.1.5 (Associativity)
\[
A \otimes B \otimes C = A \otimes (B \otimes C) = (A \otimes B) \otimes C
\]
(C.1.8)

Proposition C.1.6 (Mixed Product Rule)
\[
(A \otimes B)(C \otimes D) = AC \otimes BD
\]
(C.1.9)

and,
\[
(A \otimes B)^p = A^p \otimes B^p, \quad \forall p \in \mathbb{N}
\]
(C.1.10)

Proposition C.1.7
\[
(A \otimes B)^T = A^T \otimes B^T
\]
(C.1.11)

Proposition C.1.8
\[
(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}
\]
(C.1.12)

Proposition C.1.9
\[
\text{vec}(ABC) = (C^T \otimes A)\text{vec}(B)
\]
(C.1.13)

Proposition C.1.10
\[
\text{tr}(A \otimes B) = \text{tr}(B \otimes A) = \text{tr}(A)\text{tr}(B)
\]
(C.1.14)

Proposition C.1.11 Let $A \in \mathcal{M}_{n \times n}$ and $B \in \mathcal{M}_{m \times m}$, we have,
\[
\text{mspec}(A \otimes B) = \{\lambda \mu, \quad \lambda \in \text{mspec}(A), \quad \mu \in \text{mspec}(B)\}
\]
(C.1.15)

Proposition C.1.12 Let $A \in \mathcal{M}_{n \times n}$ and $B \in \mathcal{M}_{m \times m}$, we have,
\[
\det(A \otimes B) = (\det A)^m (\det B)^n
\]
(C.1.16)

We deduce from C.1.15,

Proposition C.1.13 Let $A \in \mathcal{M}_{n \times n}$, we have,
\[
\rho(A \otimes A) = \rho(A)^2
\]
(C.1.17)

Proposition C.1.14 Let $f$ be an analytic function, $A \in \mathcal{M}_{n \times n}$ such that $f(A)$ exists, then we have,
\[
f(I_m \otimes A) = I_m \otimes f(A)
\]
(C.1.18)

\[
f(A \otimes I_m) = f(A) \otimes I_m
\]
(C.1.19)

Proposition C.1.15 Let $X \in \mathbb{R}^n$ and $Y \in \mathbb{R}^m$, be two vectors. We have,
\[
XY^T = X \otimes (Y^T) = (Y^T) \otimes X
\]
(C.1.20)

moreover, we have,
\[
\text{vec}(XY^T) = Y \otimes X
\]
(C.1.21)
Definition C.1.16 (Kronecker permutation matrix) The Kronecker permutation matrix $P_{n,m} \in M_{nm \times nm}$, is defined by,

$$P_{n,m} = \sum_{i,j=1}^{n,m} E_{i,j,n \times m} \otimes E_{j,i,m \times n}$$  \hspace{1cm} (C.1.22)

Proposition C.1.17 Let $A \in M_{m \times n}$, we have,

$$\text{vec}(A^T) = P_{m,n} \text{vec}(A)$$  \hspace{1cm} (C.1.23)

Proposition C.1.18 Let us consider the Kronecker permutation matrix $P_{n,m} \in M_{nm \times nm}$. Then we have,

- $P_{n,m}^T = P_{m,n}^{-1} = P_{m,n}$
- $P_{n,m}$ is orthogonal,
- $P_{n,m} P_{m,n} = I_{nm}$
- $P_{n,n}$ is orthogonal, symmetric and involutory,
- $P_{n,n}$ is a reflector,
- $\text{tr}P_{n,n} = n$,
- $P_{1,m} = I_m$, and $P_{n,1} = I_n$
- if $X \in \mathbb{R}^n$ and $Y \in \mathbb{R}^m$, then,
  $$P_{n,m}(Y \otimes X) = X \otimes Y$$  \hspace{1cm} (C.1.24)
- if $A \in M_{n \times m}$ and $B \in M_{r \times s}$, then
  $$P_{r,n}(A \otimes B)P_{m,s} = B \otimes A$$  \hspace{1cm} (C.1.25)
- if $A \in M_{n \times n}$ and $B \in M_{m \times m}$, then
  $$P_{m,n}(A \otimes B)P_{n,m} = P_{m,n}(A \otimes B) P_{m,n}^{-1} = B \otimes A$$  \hspace{1cm} (C.1.26)

Therefor, $A \otimes B$ and $B \otimes A$ are similar.

Proposition C.1.19 Let $A \in M_{n \times n}$ and $B \in M_{m \times m}$, then we have the following properties,

- if $A$ and $B$ are diagonal, then $A \otimes B$ is diagonal,
- if $A$ and $B$ are upper triangular, then $A \otimes B$ is upper triangular,
- if $A$ and $B$ are lower triangular, then $A \otimes B$ is lower triangular,

Proposition C.1.20 Let $A, C \in M_{n \times m}$ and $B, D \in M_{r \times s}$. If $A$ is (left equivalent, right equivalent, equivalent) to $C$, and assume that $B$ is (left equivalent, right equivalent, equivalent) to $D$. Then, $A \otimes B$ is (left equivalent, right equivalent, equivalent) to $C \otimes D$.

Remark C.1.21 The use of Kronecker product preconditioners is well known \cite{13, 8, 12, 5}, it is based on results of the form,

$$\text{Minimizing, } \phi_A(B,C) = \|A - B \otimes C\|^2$$  \hspace{1cm} (C.1.27)

for a chosen norm.
C.1.1 Kronecker sum

Definition C.1.22 (Kronecker sum) Let $A = (a_{ij}) \in \mathcal{M}_{n \times n}$ and $B = (b_{ij}) \in \mathcal{M}_{m \times m}$ be two matrices. The Kronecker sum of $A$ and $B$, denoted by $A \oplus B \in \mathcal{M}_{mn \times mn}$, defines the following matrix:

$$A \oplus B = A \otimes I_m + I_n \otimes B$$  

(C.1.28)

Proposition C.1.23 Let $A \in \mathcal{M}_{n \times n}$ and $B \in \mathcal{M}_{m \times m}$, we have,

$$\text{mspec}(A \oplus B) = \{\lambda + \mu, \ \lambda \in \text{mspec}(A), \ \mu \in \text{mspec}(B)\}$$  

(C.1.29)

C.1.2 Solving $AX + XB = C$

Let $A \in \mathcal{M}_{n \times n}$, $B \in \mathcal{M}_{m \times m}$ and $C \in \mathcal{M}_{n \times m}$. The aim of this section, is to solve the equation:

$$AX + XB = C$$  

(C.1.30)

we can rewrite this equation in term of the Kronecker sum:

$$(B^T \oplus A)\text{vec}(X) = \text{vec}(C)$$  

(C.1.31)

or equivalently,

$$Gx = c$$  

(C.1.32)

where, $G = (B^T \oplus A)$, $x = \text{vec}(X)$, and $c = \text{vec}(C)$.

Using the property [C.1.29], we can easily check that [C.1.30] has a unique solution if and only if $G$ is nonsingular, i.e $\lambda + \mu \neq 0, \ \forall \lambda \in \text{mspec}(A), \ \forall \mu \in \text{mspec}(B)$, which can be written in the form,

$$\text{mspec}(A) \cap \text{mspec}(-B) = \emptyset$$  

(C.1.33)

Proposition C.1.24 If $\text{mspec}(A) \cap \text{mspec}(-B) = \emptyset$, then there exists a unique matrix $X \in \mathcal{M}_{n \times m}$, satisfying [C.1.30]. Moreover, the matrices $\begin{pmatrix} A & C \\ 0 & -B \end{pmatrix}$ and $\begin{pmatrix} A & 0 \\ 0 & -B \end{pmatrix}$ are similar and verify,

$$\begin{pmatrix} A & C \\ 0 & -B \end{pmatrix} = \begin{pmatrix} I & X \\ 0 & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & -B \end{pmatrix} \begin{pmatrix} I & -X \\ 0 & I \end{pmatrix}.$$  

(C.1.34)

C.1.3 Solving $AXB = C$

Let $A, B, C$ and $X \in \mathcal{M}_{n \times n}$. As seen previously, using [C.1.13] the equation

$$AXB = C$$  

(C.1.35)

can be written in the form,

$$Hx = c$$  

(C.1.36)

where, $H = (B^T \otimes A)$, $x = \text{vec}(X)$, and $c = \text{vec}(C)$.

Using the property [C.1.15], we can easily check that [C.1.35] has a unique solution if and only if $H$ is nonsingular, i.e $\lambda \mu \neq 0, \ \forall \lambda \in \text{mspec}(A), \ \forall \mu \in \text{mspec}(B)$, which is equivalent to, $A$ and $B$ are both nonsingular.

C.1.4 Solving $\sum_{i=1}^{r} A_i X B_i = C$

Let $A_i, B_i, C, \ 1 \leq i \leq r$ and $X \in \mathcal{M}_{n \times n}$. Using, the previous result, it is easy to show that the solution of:

$$\sum_{i=1}^{r} A_i X B_i = C$$  

(C.1.37)

can be written in the form,

$$Hx = c$$  

(C.1.38)

where, $H = \sum_{i=1}^{r} (B_i^T \otimes A_i)$, $x = \text{vec}(X)$, and $c = \text{vec}(C)$. 


C.2 Sparse Matrices

C.3 Linear solvers

C.3.1 Direct solvers

C.3.2 Iterative solvers
Bibliography


