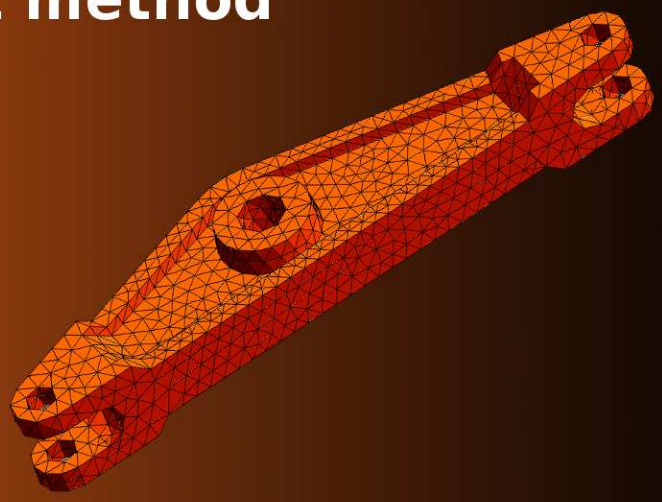
Finite element method for structures

Antoine Legay
Cnam-Paris



XII — Programming Introduction

XII.1 Programming language

A programming language is an interface between a human and a computer which enables to give a list of instructions to it. The instructions are written in a text file, called source file, through a text editor (do not be confused with a word processing program). This source file is read and translated into a machine language by the computer. The computer can then run it.

Languages have been improved since the first programming language (Assembly language) which was quite hard to understand. Nowdays, modern languages enable to use functions and are, for most of them, object-oriented.

Languages can be interpreted or compiled:

— Interpreted language: instructions are read, interpreted into a machine language and run on the fly. The advantage is the ease of use, the drawback is the slow program execution speed.

— Compiled language: instructions are read, and translated into a machine language once by a compiler in order to create an executable file. The advantage is the fast program execution speed, the drawback is that the code has to be compiled after each modification. An other advantage is that one person can give the executable file to an other one while keeping secret the source code.

```
# import the scientific library "scipy"
import scipy

print("My first Python program")

a=3.14
b=2.78
i=3
w=1+1.0j

print("a is a real :",a)
print("b is a real :",b)
print("i is an integer :",i)
print("w is a complex :",w)

c=a*b
print("a*b = :",c)

c=scipy.sqrt(a)
print("square root of a = :",c)

i=i+1
print("let's add 1 to i :",i)
i=i+1
print("let's add an other 1 to i :",i)
```

Figure XII.1 – First Python program

XII.2 Python language

XII.2.1 Python basis and installation

Python language is an object-oriented interpreted language. Several scientific libraries are available. These libraries are compiled, which is a good point for the speed-up of the program execution speed when they are used within a finite element program.

A first Python program (3.4 version) is proposed on figure XII.1. The text indentation is very important with Python, for this example, which is just a list of instructions with neither loop nor condition, all the lines start at the first column. The text after the `#` letter is not read by Python, this is comment. The scientific library `scipy` is firstly called in the memory. We write a text string on the screen, then we give real values to the `a` and `b` variables (`float` type). The memory allocation as well as the variable declaration are automatically done by Python, which is convenient but slows down the program execution speed. We give an integer value to the `i` variable (`int` type). If we would have given the value 3.0, this variable would have been a `float` type. We give a complex value to the `w` variable (`complex` type). We print on the screen these 3 variables, then we do a few computations using the square root function `sqrt` which is in the `scipy` library. Finally, we increment `i` by 1 twice. The sign `=` in the program is interpreted as a variable affectation, in other

```
My first Python program
a is a real :  3.14
b is a real :  2.78
i is an integer :  3
w is a complex :  (1+1j)
a*b = :  8.7292
square root of a = :  1.77200451467
let's add 1 to i :  4
let's add an other 1 to i :  5
```

Figure XII.2 – Screen result of the first Python program

words, the instruction `i=i+1` adds 1 to `i`: it is not an equation with one unknown to be solved. The screen output of this first program is given on figure XII.2.

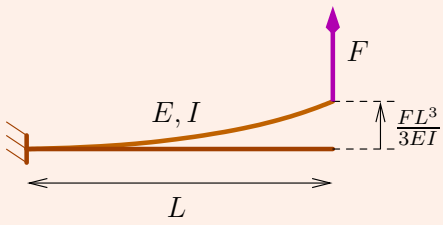
The installation of Python is quite easy:

— With the Ubuntu OS: via synaptic, install: `python3.4`, `idle-python3.4`, `python-scipy`, `python-numpy` and maybe a few others packages if necessary...

— With the Windows and Mac OS: install the package `Anaconda` which include `Python`, the scientific libraries as well as a text editor.

In order to write the program file, the easiest way is to use an integrated development environment (IDE): for instance `idle-python` or `spyder` (comes with `Anaconda`) but there are others. Within the IDE, a window appears, then one can open a file (or create a new one), (`file/open`) and then run it with `run`.

Exercise XII.1 Write a Python program which computes the displacement of a clamped-free bending beam from its length L, its Young modulus E, its second moment of area of the cross section I and the applied load F.



XII.2.2 Conditions and loops with Python

The presented program on figure XII.3 is a simple guessing game. The player has to find the hidden number `i` (here 6). The player makes a guess, we say to him if the number is greater or less than his guess. The player has only 3 tries. There is a `while` loop which ends if the `j` variable becomes greater than 3. There is a first level of indentation for the `while` loop instructions, and then a second level of indentation in each `if` condition. There is no `end` with Python, the end of a loop or a condition is indicated by the end of the indentation. The mandatory use of indentation gives readable programs.

The program on figure XII.4 makes the user revising the multiplication table and gives him a mark out of 20. The `random` library enables to choose 2 numbers randomly between

```
print("Guess my number, you have 3 tries.")

i=6  # number to discover
j=1  # tries counter

while (j <= 3):
    print("Try number ",j)
    print("Guess a number:")
    k = int(input())

    if (k==i):
        print("Congratulations")
        j=99

    if (k<i):
        print("My number is greater than you guess")

    if (k>i):
        print("My number is less than your guess")

    j = j+1

if (j==4):
    print("You have not found my number, my number was ",i)
```

Figure XII.3 – Guessing game with Python

```
import random

print("Multiplication table.")
k=0  # Count of good answers

for i in range(20):
    a=int(random.uniform(2,10))
    b=int(random.uniform(2,10))
    print(a,"*",b," =")
    c = int(input('?'))
    if c==(a*b):
        print("Yes")
        k=k+1
    else:
        print("No")

print("number of good answers out of 20: ",k)
```

Figure XII.4 – Multiplication table with Python

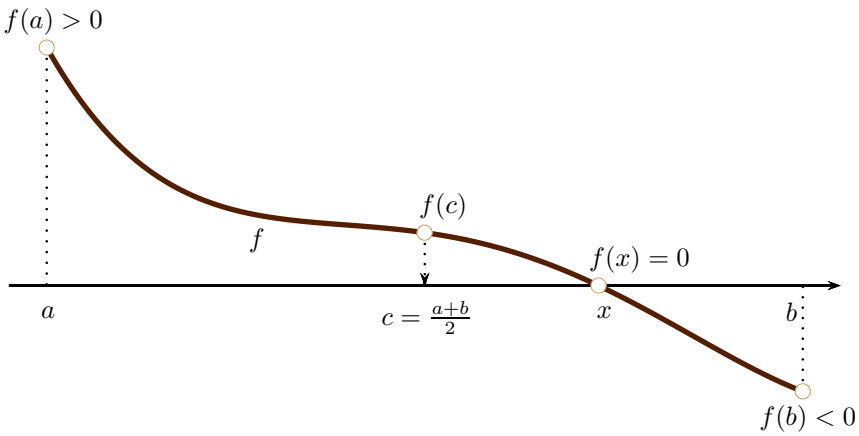


Figure XII.5 – Root-finding algorithm with the bisection method

0 and 10: the `int` function converts the random real into an integer. We print out on the screen the 2 integers `a` and `b`. The user enters a proposition, we store it in the `c` variable. We compare it to the good answer. If it is good, we print out `Yes` and add 1 to the counter `k`, otherwise, we print out `No`. The counter `k` has to be initialized to 0 at the beginning of the program. At the end, we print out the number of good answers.

Exercise XII.2 Write a Python program which computes the Syracuse sequence from a given integer. This "well-known" sequence can be obtained as the following operation on an arbitrary positive integer:

— If the number is even, divide it by two.

— If the number is odd, triple it and add one.

This operation is repeated on the result. The Syracuse conjecture is that, from any given positive integer, the sequence always reaches 1. The `x%y` Python instruction can be used: it gives the remainder of the division of x by y. ■

XII.2.3 Functions with Python

A Python function has a name, inputs, outputs, instructions and local variables. It can eventually call other functions. In order to illustrate how to create a function with Python, we write a program that find the root of a 1-variable mathematical function by the bisection method. The algorithm is as following (see figure XII.5):

`Initialization de` a `et` b

While $b - a >$`criterion` **do**

$c = \frac{a+b}{2}$

If $f(a) \times f(c) > 0$ **then** $a \leftarrow c$ **else** $b \leftarrow c$

`Print` c

`Print number of iterations`

The corresponding Python program is given on figure XII.6. The mathematical function for which we want to find the root is in the Python function named `myfunction`. The input is `x`, the output is `value`. Here, we choose to look for the root of the cosine function. The main program follows the root-finding algorithm using the bisection method, it calls the

```
def myfunction(x):
    import scipy
    value = scipy.cos(x)
    return value

print("Root of cosine between 0 and 3.")
a=0.0
b=3.0
j=1  # Counter of iterations
while ((abs(a-b))>1e-8):
    j=j+1
    ya=myfunction(a)
    c=0.5*(a+b)
    yc=myfunction(c)
    if ((ya*yc)>0.0):
        a=c
    else:
        b=c

print("Number of iterations: ",j)
print("Root of the function: ",c)
```

Figure XII.6 – Root-finding algorithm with the bisection method using Python

function `myfynction` several times. In order to change the mathematical function, one has just to change its expression in `myfunction`.

Exercise XII.3 Write a Python program that computes the integral of a function by using the rectangle rule, for instance $\cos x$, from a to b for a given n number of rectangles. ■

XII.2.4 Matrix operations with Python

The scientific librairy `scipy` enables to do matrix operations.

In the case of small matrices, such as elemental stiffness matrices (up to 100×100), they are completely stored using the `array` type. Figure XII.7 presents an example of matrix multiplication. Matrices are written row by row. The `dot` function multiplies matrices, all the other classical operations on matrices are possible.

In the case of large matrices, such as global stiffness matrices (up to 1,000,000×1,000,000), they are stored as sparse matrices. Each non-zero term is stored by associating its row and column indices. For instance, we want to use the sparse storage to store the following matrix:

$$\begin{bmatrix} 4.6 & 6.9 & 0 & 0 & 0 \\ 1.5 & 1 & 3.8 & 9.2 & 0 \\ 0 & 3.0 & 2 & 0 & 0 \\ 0 & 9 & 0 & 5.5 & 7 \\ 0 & 0 & 0 & 7 & 8 \end{bmatrix}$$

We store the matrix value terms in the array V (reals), the corresponding row indexes are

```
import scipy

A=scipy.array([[1.1  ,  0.0],
               [ 8.7  ,  0.0],
               [ 0.0  ,  2.6]])

B=scipy.array([[6.1  ,  4.5  ,  0.0  ,  0.0],
               [0.0  ,  5.5  ,  3.7  ,  1.9]])

C=scipy.dot(A,B)

print C
```

Figure XII.7 – Matrix declaration and matrix operations with Python

```
import scipy
import scipy.sparse

V=scipy.array([4.6,1.0,2.0,5.5,8.0,6.9,1.5
               ,3.8,3.0,9.2,9.0,7.0,7.0])
I=scipy.array([0,1,2,3,4,0,1,1,2,1,3,3,4])
J=scipy.array([0,1,2,3,4,1,0,2,1,3,1,4,3])

A=scipy.sparse.csc_matrix( (V,(I,J)), shape=(5,5) )

print A.todense()
```

Figure XII.8 – Sparse matrix declaration with Python

stored in the array I (integers) while the corresponding column indexes are stored in the array J (integers). Pointers with Python start at 0, that means that the first row of a matrix is the row number 0. We can first store diagonal terms, then the others, the order has no importance:

$$\begin{array}{lcccccccccccc} V = [& 4.6 & 1 & 2 & 5.5 & 8 & 6.9 & 1.5 & 3.8 & 3.0 & 9.2 & 9 & 7 & 7] \\ I = [& 0 & 1 & 2 & 3 & 4 & 0 & 1 & 1 & 2 & 1 & 3 & 3 & 4] \\ J = [& 0 & 1 & 2 & 3 & 4 & 1 & 0 & 2 & 1 & 3 & 1 & 4 & 3] \end{array}$$

The corresponding Python program is given on figure XII.8.

The advantage of the matrix sparse storage is that the same row and column indexes can be repeated several times, the values are summed up automatically. For instance, we want to assemble the 2 following 2×2 matrices into a 3×3 matrix:

$$\begin{bmatrix} 10 & -10 \\ -10 & 10 \end{bmatrix}$$

```
import scipy
import scipy.sparse

V=scipy.array([10,-10,-10,10,20,-20,-20,20])
I=scipy.array([0,0,1,1,1,1,2,2])
J=scipy.array([0,1,0,1,1,2,1,2])

A=scipy.sparse.csc_matrix( (V,(I,J)), shape=(3,3) )

print A.todense()
```

Figure XII.9 – Sparse matrix assembly with Python

on the rows and the columns [0,1], and

$$\begin{bmatrix} 20 & -20 \\ -20 & 20 \end{bmatrix}$$

on the rows and the columns [1,2]. The expected result is:

$$\begin{bmatrix} 10 & -10 & 0 \\ -10 & 10+20 & -20 \\ 0 & -20 & 20 \end{bmatrix} = \begin{bmatrix} 10 & -10 & 0 \\ -10 & 30 & -20 \\ 0 & -20 & 20 \end{bmatrix}$$

The storage can be done by using the 3 following arrays:

$$\begin{aligned} V &= [\ 10 \quad -10 \quad -10 \quad 10 \quad 20 \quad -20 \quad -20 \quad 20\ ] \\ I &= [\ 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 2\ ] \\ J &= [\ 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 2 \quad 1 \quad 2\ ] \end{aligned}$$

where 10 and 20 are stored separately at the same position [1,1]. The corresponding Python program is given on figure XII.9.

Exercise XII.4 Write a program which assembles automatically the matrix

$$\begin{bmatrix} 10 & -10 \\ -10 & 10 \end{bmatrix}$$

into a 5×5 matrix on the following rows and columns indexes: [0,1],[1,2],[2,3],[0,3],[3,4]. One can advantageously use a connectivity table `element` such that
`elements=scipy.array([[0,1],[1,2],[2,3],[0,3],[3,4]])`. ■

XII.3 Fortran language

XII.3.1 Fortran basis and installation

The Fortran language is a compiled language. It is a relatively old language which has been used for many years by scientists: almost all the finite element softwares are written in Fortran.

A first example of a Fortran program is given on figure XII.10. The Fortran program lines start at the 7th column, they have to stop at the 72nd column. This program runs the

```
      PROGRAM Hello
C comments: variable declarations
      real a,b,c
      integer i
      complex w

C     Main program
C     With Fortran, the instructions start at the 7th column
c                 and stop at the 72nd:
C23456-----------------------------------------------------|
      print*, 'First Fortran program'

      a=3.14
      b=2.78
      i=3
      w=cmplx(1,1)

      print*, 'a is a real : ',a
      print*, 'b is a real : ',b
      print*, 'i is an integer : ',i
      print*, 'w is a complex : ',w

      c=a*b
      print*, 'a*b = : ',c
      c=sqrt(a)
      print*, 'square root of a = : ',c

      i=i+1
      print*, 'let''s add 1 to i : ',i
      i=i+1
      print*, 'let''s add an other 1 to i : ',i
      END
```

Figure XII.10 – First Fortran program

same actions as its equivalent one written with Python (Figure XII.1). The first line of the program starts with the instruction `PROGRAM`. With Fortran, variables have to be declared before their use (sizes and types). Here, `a` and `b` are declared as `real`; the `i` variable is declared as `integer`.

The installation of Fortran is relatively simple:

— With the Ubuntu OS: via synaptic, install `gfortran`

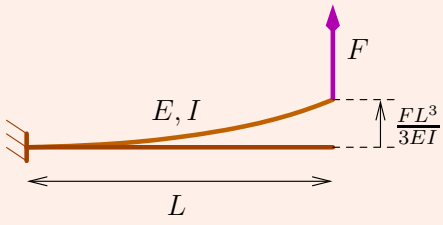
— With the Windows OS: download gfortran from the website
http://gcc.gnu.org/wiki/GFortranBinaries#Windows,
then follow the instructions.

In order to run a Fortran program, it has to be compiled to create an executable file with the following shell command (by assuming that the file name is `hello.f`):

```
gfortran hello.f
```

A new executable file named `a.out` is created. This file can be run with the shell command `./a.out` in the terminal.

Exercise XII.5 Write a Fortran program which computes the displacement of a clamped-free bending beam from its length L, its Young modulus E, its second moment of area of the cross section I and the applied load F.



■

XII.3.2 Conditions and loops with Fortran

The example of figure XII.11 is the Fortran program of the previous guessing game defined in the Python section.

The example of figure XII.12 is the Fortran program of the previous multiplication table program.

Exercise XII.6 Write a Fortran program which computes the Syracuse sequence from a given integer. This "well-known" sequence can be obtained as the following operation on an arbitrary positive integer:

— If the number is even, divide it by two.

— If the number is odd, triple it and add one.

This operation is repeated on the result. The Syracuse conjecture is that, from any given positive integer, the sequence always reaches 1. ■

XII.3.3 Functions with Fortran

The example of figure XII.13 is the Fortran program of the previous root-finding algorithm with the bisection method.

```
PROGRAM Guessing game
integer i,j,k

print*, 'Guess my number, you have 3 tries.'
i=6  ! number to discover
j=1  ! tries counter
do while (j.le.3)
   print*, 'Try number  ',j
   print*, 'Guess a number:'
   read*,k

   if (k.eq.i) then
      print*, 'Congratulations'
      return ! this ends programme
   endif

   if (k.lt.i) then
      print*, 'My number is greater than you guess'
   endif

   if (k.gt.i) then
      print*, 'My number is less than your guess'
   endif

   j = j+1
enddo

print*, 'You have not found my number, my number was ',i
END
```

Figure XII.11 – Guessing game with Fortran

```
PROGRAM Multiplications
 implicit none
 integer i,k,a,b,c
 real rand

 print*, 'Multiplication table.'
 k=0  ! Count of good answers
 do i=1,20
    a=floor(8*rand(0))+2
    b=floor(8*rand(0))+2
    print*, a,'*',b,' = ?'
    read*,c
    if (c.eq.(a*b)) then
       print*, 'Yes'
       print*, ' '
       k=k+1
    else
       print*, 'No'
       print*, ' '
    endif
 enddo
 print*, 'number of good answers out of 20: ',k
END
```

Figure XII.12 – Révision de la table de multiplication en Fortran

```
PROGRAM Bisection method
integer i,j,k
double precision a,b,c,ya,yc,mafonction
print*, 'Root of cosine between 0 and 3.'
a=0.0
b=3.0
j=1  ! Counter of iterations
do while ((abs(a-b)).ge.1e-6)
    j=j+1
    ya=myfunction(a)
    c=0.5*(a+b)
    yc=myfunction(c)
    if ((ya*yc).ge.(0.0)) then
        a=c
    else
        b=c
    endif
enddo
print*, 'Number of iterations: ',j
print*, 'Root of the function: ',c
END

double precision function myfunction(x)
  double precision x
  myfunction = cos(x)
RETURN
END
```

Figure XII.13 – Root-finding algorithm with the bisection method using Fortran

```
PROGRAM Matrices

double precision A(3,2),B(2,4),C(3,4)
integer i,j,k

A(1,1)=1.1 ; A(1,2)=0.0
A(2,1)=8.7 ; A(2,2)=0.0
A(3,1)=0.0 ; A(3,2)=2.6

B(1,1)=6.1 ; B(1,2)=4.5 ; B(1,3)=0.0 ; B(1,4)=0.0
B(2,1)=0.0 ; B(2,2)=5.5 ; B(2,3)=3.7 ; B(2,4)=1.9

do i=1,3
  do j=1,4
    C(i,j)=0.0
    do k=1,2
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    enddo
  enddo
enddo

do i=1,3
    print*, (C(i, j), j=1,4)
enddo

END
```

Figure XII.14 – Matrix declaration and matrix operations with Fortran

Exercise XII.7 Write a Fortran program that computes the integral of a function by using the rectangle rule, for instance $\cos x$, from a to b for a given n number of rectangles. ■

XII.3.4 Matrix operations with Fortran

There are Fortran matrix operation libraries, but one can program the multiplication of 2 matrices by using 3 loops as it is done in the example shown on figure XII.14. It is possible to use the sparse matrix storage with Fortran, but it is not used here since we couple Python and Fortran as it is shown in the next section.

XII.4 Use of Fortran functions within a Python program

A Fortran function can be easily used within a Python program. One can then take advantages of the two languages: program execution speed of Fortran (specially for loops) and easiness of Python (specially for sparse matrices).

For instance, we can write a Fortran subroutine which computes the product of two matrices. This subroutine, given on figure XII.15, is saved with the name `lib_multiplication.f`. From this Fortran routine, we build up a Python library with the

```
      SUBROUTINE prodmat(A,B,C,l,m,n)
      integer l,m,n,i,j,k
      real*8 A(l,m),B(m,n),C(l,n)

C inputs/outputs for Python
Cf2py intent(in) A
Cf2py intent(in) B
Cf2py intent(out) C

      do i=1,l
        do j=1,n
          C(i,j)=0.0
          do k=1,m
            C(i,j)=C(i,j)+A(i,k)*B(k,j)
          enddo
        enddo
      enddo
      return
      END
```

Figure XII.15 – Fortran routine `lib_multiplication.f` for computing the product of two matrices

following shell instruction:

```
f2py3 -c -m lib_multiplication lib_multiplication.f
```

In the Fortran subroutine `promat`, the following instructions (which are comments for Fortran):

```
Cf2py intent(in) A
Cf2py intent(in) B
Cf2py intent(out) C
```

are interpreted by `f2py` in order to build inputs and outputs of the `prodmat` function of the created Python library `lib_multiplication.cpython-34m.so`. This library can then be used within a Python program as it is done on figure XII.16. In this example, the parametrized matrix sizes `l`, `m` and `n` are automatically managed by `f2py` and do not need to be given when the routine is called from Python.

The following Python instruction:

```
print(lib_multiplication.__doc__)
```

enables to know all the functions contained in the `lib_multiplication` library.

The following Python instruction:

```
print(lib_multiplication.prodmat.__doc__)
```

enables to know all the expected inputs and outputs of the `prodmat` function.

The screen output of the Python program given on figure XII.16 which calls a Fortran library is plotted on figure XII.17.

```
import scipy
import lib_multiplication

A=scipy.array([[1.1  , 0.0],
               [ 8.7  , 0.0],
               [ 0.0  , 2.6]])

B=scipy.array([[6.1  , 4.5  , 0.0  , 0.0],
               [0.0  , 5.5  , 3.7  , 1.9]])

print(lib_multiplication.__doc__)
print(lib_multiplication.prodmat.__doc__)

C=lib_multiplication.prodmat(A,B)

print(C)
```

Figure XII.16 – Use of a Fortran library within a Python program

```
This module 'lib_multiplication' is auto-generated with f2py (version:2).
Functions:
c = prodmat(a,b,l=shape(a,0),m=shape(a,1),n=shape(b,1))

c = prodmat(a,b,[l,m,n])

Wrapper for ``prodmat``.

Parameters
----------
a : input rank-2 array('d') with bounds (l,m)
b : input rank-2 array('d') with bounds (m,n)

Other Parameters
----------------
l : input int, optional
    Default: shape(a,0)
m : input int, optional
    Default: shape(a,1)
n : input int, optional
    Default: shape(b,1)

Returns
-------
c : rank-2 array('d') with bounds (l,n)

[[  6.71   4.95   0.     0.  ]
 [ 53.07  39.15   0.     0.  ]
 [  0.    14.3    9.62   4.94]]
```

Figure XII.17 – Screen output of the use of a Fortran library within a Python program