Zohour

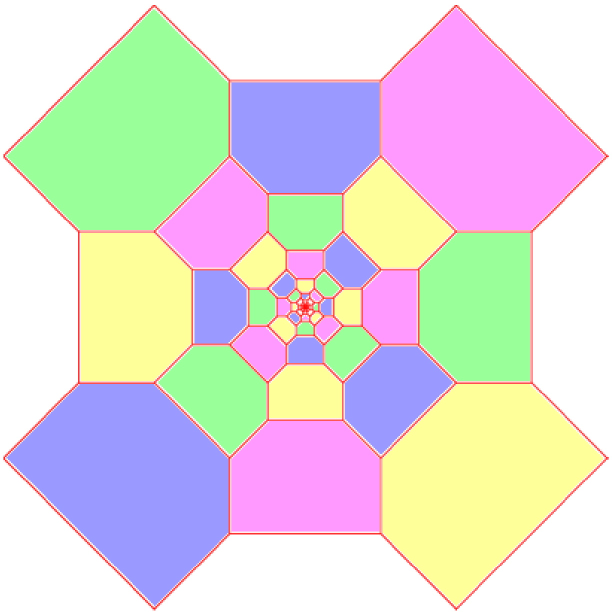
User's Guide

Fortran 2003 implementation

Release 0.9.1

Édouard Canot*

Jun 16, 2019



*IPR/CNRS, Rennes, France

Contents

1 Introduction

This document describes the Zohour Fortran 2003 library.

Zohour is freely available at the following address: https://perso.univ-rennes1.fr/edouard.canot/zohour/

Bugs reports or comments: mailto:Edouard.Canot@univ-rennes1.fr

Thanks to the Zohour users: Corentin Beaucé, Salwa Mansour.

About the name

Zohour: during the mesh modification, new nodes appear; for this reason the name given to this mesh algorithm is “Zohour” (ظهور means “emergence” in arabic language). But “Zohour” also corresponds to زهور, which means “flower”, and this show the strong link with the logo design – see the cover page.

2 A Node-Based Adaptive 2D mesh algorithm

2.1 The zohour_2D module

From the user point-of-view, the Zohour library is seen as a Fortran module. This module has many *private* components and routines and it is distributed under the following form:

– an archive library: `libzohour_2d.a` which contains the binary code of the Zohour algorithm;

– a Fortran precompiled module: `zohour_2d.mod` which contains the interface of all routines available to the user.

These two files are needed to use the Zohour module. Be aware that the precompiled module is compiler dependent and therefore, the user must use the appropriate compiler version.

The following describes the *public* part of the Zohour module.

2.1.1 Available derived types and variables

The `cell` derived type contains all stuff for the 2D mesh algorithm, and is designed as follows:

```
type :: cell

   ! coordinates of the central node
   double precision :: x, y

   double precision, allocatable :: data(:) ! user data

   ! gradient module and hessian value, only for data(1)
   double precision :: grad_data, grad2_data

   ! next cell in the linked list
   type(cell), pointer :: next

   ! these pointers describe the spatial connectivity
   class(*), pointer :: N     ! what is at North
   class(*), pointer :: NE    !   "     "   North-East
   class(*), pointer :: E     !   "     "   East
   class(*), pointer :: SE    !   "     "   South-East
   class(*), pointer :: S     !   "     "   South
   class(*), pointer :: SW    !   "     "   South-West
   class(*), pointer :: W     !   "     "   West
   class(*), pointer :: NW    !   "     "   North-West

   ! edge length in terms of the distance to the cell boundary
   integer :: Nl  = 2  ! length of North      side
   integer :: NEl = 2  !   "     "  North-East  "
   integer :: El  = 2  !   "     "  East        "
   integer :: SEl = 2  !   "     "  South-East  "
   integer :: Sl  = 2  !   "     "  South       "
   integer :: SWl = 2  !   "     "  South-West  "
   integer :: Wl  = 2  !   "     "  West        "
   integer :: NWl = 2  !   "     "  North-West  "

   ! more internal components (not available to the user)
   ...

end type mfArray
```

All the cells are linked in a list, whose entry is the following pointer:

```
type(cell), pointer :: mesh_beg
```

So, to access all the cells of the mesh, one can use:

```
type(cell), pointer :: tmp
...
tmp => mesh_beg
do while( associated(tmp) )
   ! here, access to members of the cell 'tmp'
   ! ...
   tmp => tmp%next
end do
```

The total number of cells is stored in this variable:

```
integer :: nb_nodes
```

The user can store any number of floating-point values in each cell. This number is passed to the *Zohour* library by setting:

```
integer :: n_cell_data
```

`n_cell_data` must be greater than one. The library itself allocates the array `data(:)` of each cell.

During the remeshing, the cells can be divided many times. Initially, the basic mesh (whose dimensions are given by the user) has all its cells as squares, of size `dist_0` . After a subdivision, a cell sees its shape changed, usually not a square. At any time, the user may retrieve the effective subdivision levels used by reading the 2-element array:

```
integer :: level_range(2)
```

`level_range` contains the lower and the upper subdivision levels.
Of course, `level_range(1)` $\leq$ `level_range(2)` ; on the other hand, the minimum value of the subdivision level is 0 (*i. e.* the basic mesh) whereas its maximum value is specified at initialization by the user (see the `subdiv_max` argument of the `init_mesh` routine, next section).

Concerning the geometric description of the computational domain, it uses first a derived type for the boundaries:

```
type :: boundary

   ! coordinates of a line: a*x + b*y = c
   double precision :: a = 0.0d0, b = 0.0d0, c = 0.0d0

   integer :: bc_type = 0

end type
```

As can be seen above, each part of the boundary must be a straight line, defined by the three coefficients a, b and c of the line equation. Moreover, one boundary condition type is attached to the part of boundary, via the component bc_type. This latter boundary condition type should take only one of the following possible values:

```
integer, parameter :: BC_type_Dir  ! Dirichlet
integer, parameter :: BC_type_Neu  ! Neumann
integer, parameter :: BC_type_Oth  ! Other
```

The boundary lines must be horizontal or vertical, so the couple of values (a,b) should be equal to (1,0) or (0,1).

2.1.2 Available routines

```
subroutine init_mesh( BC_type, nx, ny, subdiv_max, y_max )

   interface
      function BC_type( x, y, side ) result( res )
         double precision, intent(in) :: x, y
         character(len=*), intent(in) :: side
         integer                      :: res
         ! on input:
         !   (x,y) : position in the domain
         !   side  : boundary side
         !
         ! on output:
         !   res : the B.C. for the (x,y) point
      end function
   end interface

   integer, intent(in) :: nx, ny, subdiv_max
   double precision, intent(out) :: y_max
```

init_mesh is the routine which must be called first by the user; it creates the mesh and applies the boundary conditions given by the user-defined BC_type routine.

An example of BC_type routine could be:

```
function BC_type( x, y, side ) result( res )
   double precision, intent(in) :: x, y
   character(len=*), intent(in) :: side
   integer                      :: res
   ! on input:
   !   (x,y) : position in the domain
   !   side  : boundary side( "North", "East", "South" or "West" )
   !
   ! on output:
   !   res : the B.C. for the (x,y) point
   !         (BC_type_Dir, BC_type_Neu, BC_type_Oth)
   select case( side )
      case( "North" )
         if( x > heat_length ) then
            res = BC_type_Neu
         else
            res = BC_type_Dir
         end if
```

```
      case( "East" )
         res = BC_type_Dir
      case( "South" )
         res = BC_type_Dir
      case( "West" )
         res = BC_type_Neu
   end select
end function
```