Ani2D

Advanced Numerical Instruments 2D

The package Ani2D is designated for **generating** unstructured triangular meshes, **adapting** them isotropically and anisotropically, **discretizing** systems of PDEs, **solving** linear and nonlinear systems, and **visualizing** meshes and associated solutions. It is a set of independent libraries with different tasks. The libraries may be combined to solve complex problems. The libraries were designed to be included easily in other packages. Extensive tutorials represent powerful capabilities of the package Ani2D.

The package Ani2D has been developed by a team of researchers since 1997. The team is headed by the two principle investigators:

- Konstantin Lipnikov[1]
- Yuri Vassilevski[2].

Ideas and technologies, as well as packages Ani2D-MBA, Ani2D-FEM, Ani2D-LMR, Ani2D-PRJ and Ani2D-VIEW have been developed by the principal investigators.

The package Ani2D-AFT has been developed by

- Alexander Danilov[2]

under the supervision of the principal investigators.

The package Ani2D-RCB has been developed by

- Vadim Chugunov[2]
- Yuri Vassilevski[2].

The package Ani2D-ILU has been developed by

- Sergei Goreinov[2]
- Vadim Chugunov[2]
- Yuri Vassilevski[2].

The package Ani2D-INB has been developed by

- Alexey Chernyshenko[2]

under the supervision of the principal investigators.

Adaptation of the package Ani2D for OS Windows has been developed by

- Nazar Andrienko[2].

Besides the original software, the package Ani2D incorporates a number of public libraries such as BLAS, LAPACK, UMFPACK and AMD.

[1]Los Alamos National Laboratory, Theoretical Division, MS-284, Los Alamos, NM 87545, USA.
[2]Institute of Numerical Mathematics RAS, 8 Gubkina St., 119333 Moscow, RUSSIA.

Copyright and Usage Restrictions

Contact Information

The authors are interested in your feedback. To report a problem, make a suggestion, or request a new feature, please contact us by email:

ani2d.help@gmail.com

Structure of our packages

After package installation, the user will get the following subdirectories

```
bin/  data/  doc/  lib/  src/  cmake/  python/
```

The executable files are always placed in `bin/`. Examples of simple meshes are located in `data/`. A PDF documentation for the package is in `doc/`. The source code is located in `src/` and the usage of the libraries is demonstrated in `src/Tutorials`. Examples of short cmake scripts are in directory `cmake/`. The result of installation is a set of libraries placed in `lib/` and executables placed in `bin/`. An experimental python code illustrating some of the package features is in directory `python/`.

The directory `src/` contains libraries and tutorials:

```
aniXXX/  lapack/  blas/  Tutorials/  Rules.make
```

The source code for libraries is located in various directories `aniXXX`. An incomplete versions of `LAPACK`[3] and `BLAS`[4] libraries are located in directories with the same names. File `Rules.make` is used for the first installation method, see a section below.

The tutorials are split into two groups

```
PackageXXX/   MultiPackage/
```

The first set of directories contains simple examples of using our libraries individually. The directory `MultiPackage` contains more complex examples using multiple packages. The main focus in `MultiPackage` is the solution of linear and nonlinear PDEs. Most of the directories are equipped with `README`s to help the user to navigate through the code.

[3] http://www.netlib.org/lapack
[4] http://www.netlib.org/blas

Code portability

The package is tested regularly under various operational systems:

- Linux (Fedora, Ubuntu, ArchLinux, OpenSUSE)
- Windows (XP, 7)
- Unix (FreeBSD, OpenBSD)
- MaxOSX (Leopard, Snow Leopard)
- Oracle Solaris

This release was tested with various compilers:

- GCC[5] 4.3.4, 4.7.3 and 4.8.3
- Intel[6] 13.1.3 and 14.0
- PGI[7] 13.6

[5] https://gcc.gnu.org/fortran
[6] https://software.intel.com/en-us/fortran-compilers
[7] http://www.pgroup.com

Two alternative installation methods

We provide two methods for package installation. The first method uses `CMake`. It requires to execute the following commands:

```
$ mkdir build
$ cd build
$ cmake ../  (or a specilized script from directory cmake/)
$ make install
$ make test  (optional)
$ cd ..
$ ./DEMO
```

The packages will be compiled with a local copies of `LAPACK` and `BLAS`. To use system libraries, provide the following option to `CMake`:

```
$ cmake -DENABLE_SYSTEM_LAPACK:BOOL=TRUE ../
```

The second installation method is based on a set of simple `Makefiles`. Support of this installation method will be significantly reduced with time. In order to compile the code, the user has to set up the compilers names in `src/Rules.make` and then to execute the following commands:

```
$ make libs
$ make packages
$ ./DEMO
```

WARNING. Problems with automatic linking of C and FORTRAN libraries can be found on some OS and for some compilers. The simplest solution is to skip three tests in installation of our packages by adding the following option to `CMake` command:

```
$ cd build
$ cmake -DDISABLE_C2F_TESTS:BOOL=TRUE [other options] ../
```

Some compilers have special flags to link the code with all required libraries. A few hints can be found in file `Rules.make`; however, we do not guarantee that they will work for your particular OS.

Additional installation notes for Windows XP and 7

The information below was collected over a few years and should be considered as a general guidance for Windows users. In order to use package Ani2D under Windows, we propose to compile libraries separately and then use them in user's projects. First, the user has to install the following software packages:

1. MinGW with gcc, gfortran (g77 in earlier versions), and make
2. MSYS base system

MinGW (Minimalist GNU for Windows) provides Windows port of GNU binutils and GNU compiler collection.[8] MSYS (Minimal SYStem) adds UNIX terminal emulator to MinGW. MSYS can be downloaded from the same site as MinGW.

Optionally, to visualize meshes and solutions, the user has to install a package for viewing Postscript files, such as GSview[9] and GhostScript.

In general, there is no difference between compiling Ani2D under MinGW and UNIX, so we refer to above documentation on compilation.

Compilation in MinGW with cmake

In order to compile package Ani2D under Windows with `CMake`, the user has to install MinGW and additional package `CMake`. To find automatically `C` and `FORTRAN` compilers, we call:

```
$ cd build
$ cmake ../ -G "MSYS Makefiles"
```

It is recommended to execute the cmake command under MSYS terminal.

Compilation in MinGW with Makefiles

An alternative (old style) compilation of package Ani2D uses a set of manually created `Makefiles`. Again `MinGW` has to be installed. The compilers names are set up in file `src/Rules.make`:

```
F77 = gfortran  # Fortran compiler
CC = gcc        #  C compiler
PS = "C:\Program Files\GhostScript\Ghostgum\gsview\gsview32.exe"
```

Please notice the presence of quotes in PS variable assignment: you should use them when path to gsview32.exe contains spaces. Now everything is ready for compiling libraries and packages:

```
$ make libs
$ make packages
```

The user can run executables in directory `bin` and examine the created Postscript files. Alternatively, the user can run script `DEMO`.

[8] `http://sourceforge.net/projects/mingw`
[9] `http://pages.cs.wisc.edu/~ghost/gsview/`

Creation of a Visual Studio project

The user can check tutorials located in directory `src/Tutorials`. We put there examples can be used to create a new Visual Studio project. Let us consider a tutorial in `src/Turorials/MultiPackage/Stokes` that solves the Stokes equations in a nasal with a circle obstacle. The important steps, we want to cover in a short review are:

- Create a new project and populate it with files from directory `Stokes`.
- Open Project Properties Pages and set up the following options (based on VS 2010 with VF 2011):

Option Name	
Additional Include Directories	c:\MinGW\msys\1.0\home\user\ani2D-3.0\src\aniFEM
Name Case Interpretation	Lower Case (/names:lowercase)
Append Underscore to External Names	Yes (/assume:underscore)
Enable Incremental Linking	Yes (/INCREMENTAL)
Additional Library Directories	c:\MinGW\msys\1.0\home\user\ani2D-3.0\lib; c:\MinGW\lib; c:\MinGW\lib\gcc\4.5.2
Additional Dependencies	`libc2f2D-3.0.a libview2D-3.0.a libmba2D-3.0.a libblas-3.2.a liblapack-3.2.a liblapack_ext-3.2.a libgfortran.dll.a libgcc.a libmingwex.a`

  The exact path to libraries may vary slightly depending on the `MinGW` version. The library `libgfortran.dll.a` must be replaced by `libg2c.a` in earlier versions of `MinGW`.

- The executable created by the Visual Fortran can be run in a `MSYS` terminal.

References

1. Yu.Vassilevski, A.Danilov, K.Lipnikov, and V.Chugunov. Automated technologies for generation of unstructured computational meshes. Fizmatlit, Moscow, (2014), in production.

2. A.Agouzal, K.Lipnikov, Yu.Vassilevski. On optimal convergence rate of finite element solutions of boundary value problems on adaptive anisotropic meshes. *Mathematics and Computers in Simulation* (2011) **81**, 1949–1961.

3. A.Agouzal, Yu.Vassilevski. Minimization of gradient errors of piecewise linear interpolation on simplicial meshes. *Comp.Meth.Appl.Mech.Engnr.* (2010) **199**, 2195–2203.

4. A.Agouzal, K.Lipnikov, Yu.Vassilevski. Hessian-free metric-based mesh adaptation via geometry of interpolation error. *J. Computational Mathematics and Mathematical Physics* (2010) **50**, 124–138.

5. A.Agouzal, K.Lipnikov, Yu.Vassilevski. Edge-based a posteriori error estimators for generating quasi-optimal simplicial meshes. *Math.Model.Nat.Phenom.* (2010) **5**, 91–96.

6. K.Lipnikov, Yu.Vassilevski. On discrete boundaries and solution accuracy in anisotropic adaptive meshing. *Engineering with Computers* (2010) **26**, 281–288.

7. V.Dyadechko, K.Lipnikov, Yu.Vassilevski. Hessian based anisotropic mesh adaptation in domains with discrete boundaries. *Russ.J.Numer.Anal.Math.Modelling* (2005) **20**, 391–402.

8. A.Agouzal, Yu.Vassilevski. An unified asymptotical analysis of interpolation errors for optimal meshes. *Doklady Mathematics* (2005) **72**, 879-882.

9. K.Lipnikov, Yu.Vassilevski. Error bounds for controllable adaptive algorithms based on a Hessian recovery. *J. Computational Mathematics and Mathematical Physics* (2005) **45**, 1374–1384.

10. K.Lipnikov, Yu.Vassilevski. Parallel adaptive solution of 3D boundary value problems by Hessian recovery. *Comp.Methods Appl.Mech.Engnr.* (2003) **192**, 1495–1513.

11. K.Lipnikov, Yu.Vassilevski. Optimal triangulations: existence, approximation and double differentiation of P_1 finite element functions. *J. Computational Mathematics and Mathematical Physics* (2003) **43**, 827–835.

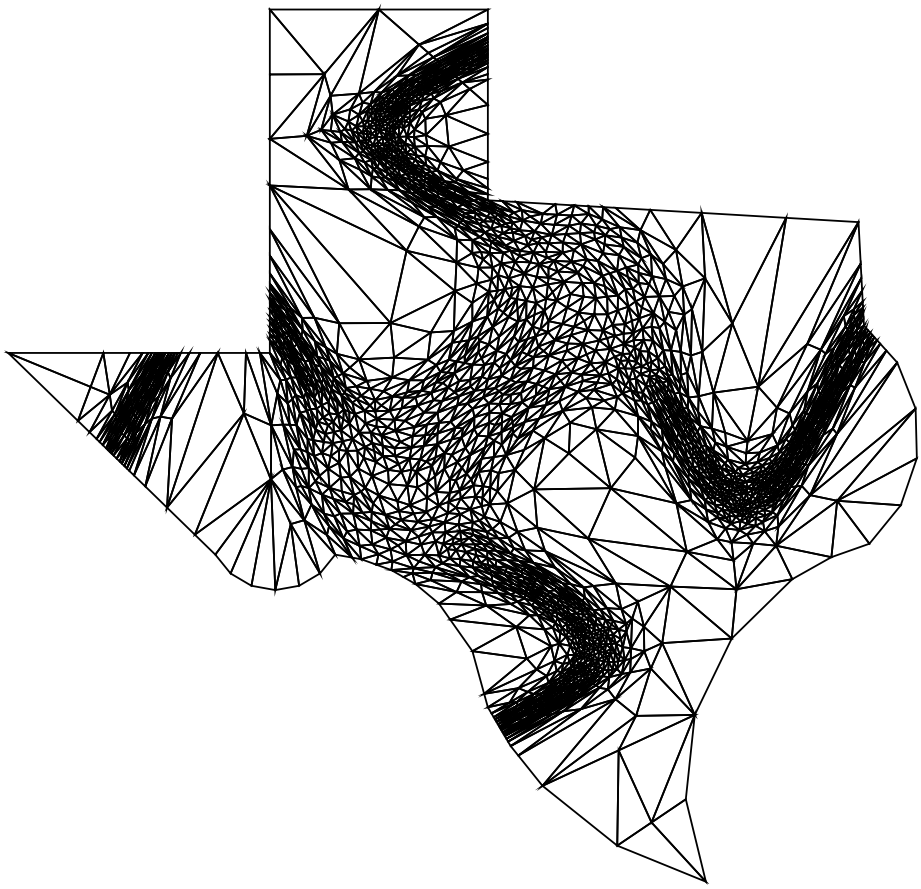
12. A.Agouzal, K.Lipnikov, Yu.Vassilevski. Adaptive generation of quasi-optimal tetrahedral meshes. *East-West J. Numer. Math.* (1999) **7** 223–244.

13. K.Lipnikov, Yu.Vassilevski. An adaptive algorithm for quasioptimal mesh generation. *J. Computational Mathematics and Mathematical Physics* (1999) **39**, 1468–1486.

Contents

Chapter 1

MESHING PACKAGEs



MESH STRUCTURE

All packages use common definition of a mesh; however, the names may vary from package to package for historic reasons. Hereafter, the `mesh` means either all arrays shown below or their various subsets.

```
Points:
  nv    - the actual number of mesh points
  nvmax - the maximal allowed number of mesh points
  nvfix - the number of fixed points

  vrt(2, nvmax) - the Cartesian coordinates of mesh points
  labelV(nvmax) - point id, a non-negative number
  fixedV(nvmax) - a list of fixed points


Boundary and material interface edges:
  nb    - the actual number of boundary and interface edges
  nbmax - the maximal number of boundary and interface edges
  nbfix - the number of fixed edges

  bnd(2, nbmax) - connectivity list of boundary and internal
                  interface edges
  labelB(nbmax) - boundary/interface edge id, a positive number
  fixedB(nbmax) - a list of fixed edges

  nc - the number of curved edges, nc <= nb
  crv(2, nbmax) - parameterizations of curvilinear edges
                  column 1 - parameters for first points
                  column 2 - parameters for last points
  labelC(nbmax) - zero or positive function id for computing
                  the Cartesian coordinates of points on mesh edges


Triangles:
  nt    - the actual number of triangles
  ntmax - the maximal number of triangles
  ntfix - the number of fixed triangles

  tri(3, ntmax) - connectivity list of triangles
  labelT(ntmax) - triangle id, a positive number
  fixedT(ntmax) - a list of fixed triangles
```

Ani2D-AFT version 3.1 "*Forget-me-not*"

Flexible Triangular Mesh Generator Using Advancing Front Technique

User's Guide for libaft2D-3.1.a

1.1 Basic features of the library

The C package Ani2D-AFT is a part of the package Ani2D. Ani2D-AFT was developed by Alexander Danilov under the supervision of Yuri Vassilevski. It generates triangular meshes in arbitrary 2D domains.

The library *libaft2D-3.1.a* can be easily incorporated in other packages. Its basic features are listed below.

Domain type : single or multiple component, simply or multiply connected finite domains

Boundary type : piecewise smooth

Data input : set of linear/curvilinear intervals representing the boundary, or the boundary mesh

Data format : double precision and integer arrays. Enumeration starts from 1.

1.2 Analytical representation of the boundary

If the domain boundary is given analytically, the user should provide additional routine, e.g. `userboundary`, describing this boundary.

```
 integer  aft2dboundary
 external aft2dboundary

 external userboundary
 call registeruserfn(userboundary)
 ...
 ierr = aft2dboundary(Nbv, bv, Nbl, bl, bltail,  h,
&                     nv, vrt,
&                     nt, tri, labelT,
&                     nb, bnd, labelB,
&                     nc, crv, iFNC)
 if (ierr.ne.0) stop 'error in function aft2dboundary'
```

1.2.1 Input for routine `aft2boundary`

The piecewise smooth boundary is represented in terms of the union of a finite number of intervals. Each interval is a smooth curve. It may be split into several subcurves. End points of the intervals are called the V-points. If the domain is multi-connected, each simply connected subdomain has a boundary composed of the given intervals. The input domain is specified by three arrays. Array `bv` describes all V-points, whereas arrays `bl` and `bltail` describe the intervals. `Nbv` is the number of V-points, i-th column in the array `bv(2,Nbv)` has coordinates of the i-th V-point, $i = 1, \ldots,$ `Nbv`. `Nbl` is the number of intervals, i-th column of the integer array `bl(7,Nbl)` describes the i-th interval, $i = 1, \ldots,$ `Nbl` using 7 parameters. The integers from the 7-parameter column are explained below.

1. The index of the V-point at which the interval begins.

2. The index of the V-point at which the interval terminates.

3. If the interval is linear, it is zero. Otherwise it is a positive number defining the type of parameterization in the user defined function `userboundary` which is in the file `crv_model.c`.

4. A dummy integer.

5. The label of the interval. It does not affect the mesh generation. All the mesh edges from the interval will inherit this number.

6. The index of the subdomain, for which the interval is a boundary part.

7. The slit marker. If the interval is the outer part of the domain boundary, it is zero. Otherwise, it is the index of a subdomain that shares the interval with the subdomain indicated by the 6th parameter.

The i-th column of the array `bltail(2,Nbl)` has two zeros when the interval is linear. Otherwise this column contains two parameters corresponding the starting and the terminal points of the interval. These parameters define the Cartesian coordinates of the V-point.

The rules for interval specification are as follows:

1. When moving along the interval from the starting point to the terminal point, the subdomain is located on the right. In other words, the intervals are given clock-wise.

2. For slit intervals, the subdomain indicated by the 6th parameter must be located on the right.

3. For slit intervals shared by two subdomains, the order of the V-points is arbitrary.

4. Coordinates of V-points defined via parametric functions using the data in array `bltail(2,Nbl)` may be different from the corresponding entries in the array `bv`. The latter entries are not used in this case.

Boundary parameterization is provided via a user routine, e.g. `userboundary`. The name of this routine must be registered in the library before the call of `aft2dboundary`:

```
external userboundary
call registeruserfn(userboundary)
...
ierr = aft2dboundary(...)
```

An example of the user defined function describing the complement of a wing to the unit square is in file `src/Tutorials/PackageAFT/crv_model.c`.

The generator produces a quasi-uniform mesh of the given mesh size `h`. However user may override this behavior by providing his own mesh size function. In this case parameter `h` is not used.

```
external usermeshsize
call registersizefn( usermeshsize )
```

An example of the simple user defined mesh size function is in the tutorial example `src/Tutorials/PackageAFT/main_boundary_square.f`.

1.2.2 Output for routine `aft2boundary`

The routines output the mesh as described on page 14. The number of mesh nodes is `nv`. Their Cartesian coordinates are stored in the two-dimensional array `vrt(2,nv)`.

The number of mesh triangles is `nt`. The connectivity list of triangles is stored in the two-dimensional array `tri(3,nt)`. The triangle identificators (labels) are stored in array `labelT(nt)`.

The number of boundary edges is `nb`. The connectivity list of these edges is stored in the two-dimensional array `bnd(2,nb)`. Their identificators (labels) are stored in array `labelB(nb)`.

The number of curved (parametrized) boundary edges is `nc`. Their parameterization is stored in the corresponding column of two-dimensional array `crv(2,nb)` and the corresponding entry of array `iFNC(nb)`. [1] For example, the j-th parametrized edge uses parameter `crv(1,j)` for its starting point and parameter `crv(2,j)` for its terminal point, as well as the identificator `iFNC(j)` of the function that calculates the Cartesian coordinates of interior edge points.

1.3 Grid representation of the boundary

If the domain boundary is given by a set of mesh edges, there is no need in a user defined function for the boundary parameterization. Instead, the user must call the following routine.

```
integer  aft2dfront
external aft2dfront

ierr = aft2dfront(Nbr, brd, Nvr, vbr,
&                 nv,  vrt,
&                 nt,  tri, labelT,
&                 nb,  bnd, labelB)
if (ierr.ne.0) stop 'error in function aft2dfront'
```

1.3.1 Input for routine `aft2dfront`

The domain boundary is described by mesh edges. The total number of the boundary edges is `Nbr`. The number of boundary nodes is `Nvr`. The i-th column of the array `vbr(2,Nvr)` contains Cartesian coordinates of the i-th boundary node. The i-th column of the two-dimensional array `brd(2,Nbr)` contains the node indexes of the starting and terminal points of the edge.

[1] Note that the size of these arrays is `nb`!

There are two methods for representing the boundary mesh.

The first method assumes that `Nbr > 0`. The boundary nodes and edges are stored in an *arbitrary order*. While moving along an edge from the starting point to the terminal point, the subdomain must be located on the right. Internal slits are allowed. Edges lying on these slits must be defined twice in opposite directions.

The second method assumes that `Nbr = 0`. Only boundary nodes are used to represent the boundary, but their order is important. In this case, array `brd` is not used. The rules for the boundary mesh specification are as follows:

1. The boundary is a union of loops. The loops are stored in `vbr` in the sequential order.
2. In each loop, the first node and the last node must be identical. This fact is used to distinguish different loops.
3. When moving to the next node within one loop, the subdomain must be located on the right.
4. Loops can overlap in any way, in this case the shared node(s) must be presented in each loop.

The resulting mesh will have the trace at the boundary matching to the boundary grid `vbr, brd`. The local mesh size depends on the distance to the boundary: the farther from the boundary, the coarser the mesh is.

Two illustrative examples of using the initial front data may be found in directory `PackageAFT/examples`.

1.3.2 Output for routine `aft2dfront`

The routines output the mesh as described on page 14. The number of mesh nodes is `nv`, their Cartesian coordinates are stored in the array `vrt(2,nv)`. The number of mesh triangles is `nt`, the connectivity list of triangles is stored in the array `tri(3,nt)`. The triangle identificators (labels) are in array `labelT(nt)`.

The number of mesh boundary edges is `nb`. The connectivity list for the edges is stored in array `bnd(2,nb)`. The boundary edge identificators (labels) are in array `labelB(nb)`.

1.4 Three examples

In this section we present three meshes generated by the package as well as data specifying the domain.

The first example uses analytical representation of the boundary. We present the piece of FORTRAN code `src/Tutorials/PackageAFT/main_boundary_wing.f` producing the mesh shown in Fig.1.1.

```
c complement of a wing NACA0012 to the unit square
      double precision bv(2,7), bltail(2,8)
      integer          Nbv, Nbl, bl(7,8)
c number of boundary nodes and number of boundary edges
```

```
      data             Nbv/7/, Nbl/8/
c boundary nodes
      data             bv/0,0, 0,1, 1,1, 1,0, .4,.5, .6,.5, 1,.5/
c outer boundary edges
      data             bl/1,2,0,-1,-1,1,0, 4,1,0,-1,-1,1,0,
     &                    2,3,0,-1,1,1,0,  7,4,0,-1,1,1,0,
     &                    3,7,0,-1,1,1,0,
c slit  boundary edges
     &                    6,7,2,0,11,1,1,
c wing  boundary edges
     &                    6,5,1,-1,2,1,0,  5,6,1,-1,2,1,0/
c curved data for each outer boundary edge
      data             bltail/0,0, 0,0, 0,0, 0,0, 0,0, 0,1, 0,.5, .5,1/

      integer nv, nt, nb, nc
      double precision crv(2,nbmax), vrt(2,nvmax)
      integer          iFNC(nbmax),  labelT(ntmax),
     &                 tri(3,ntmax), bnd(2,nbmax), labelB(nbmax)
      double precision h
      integer  aft2dboundary
      external aft2dboundary

c register name of the user function describing boundary with the library
      external userboundary
      call registeruserfn(userboundary)

c generate quasi-uniform mesh with meshstep h
      h = 0.02
      ierr = aft2dboundary(
     &       Nbv, bv, Nbl, bl, bltail, h,      ! input geometry
     &       nv, vrt, nt, tri, labelT,         ! output mesh
     &       nb, bnd, labelB, nc, crv, iFNC)
      if (ierr.ne.0) stop 'error in function aft2dboundary'
```

The second example uses the discrete representation of a domain boundary. We present a part of the FORTRAN code `src/Tutorials/PackageAFT/examples/main_front1.f` and the data file `src/Tutorials/PackageAFT/examples/front1` producing the mesh shown in Fig.1.2.

```
c list of boundary edges
      double precision vbr(2,nbmax)
      integer   Nbr, Nvr, brd(2,nbmax)

      double precision vrt(2,nvmax)
      integer   nv, nt, nb
      integer   labelT(ntmax), tri(3,ntmax), bnd(2,nbmax), labelB(nbmax)
```

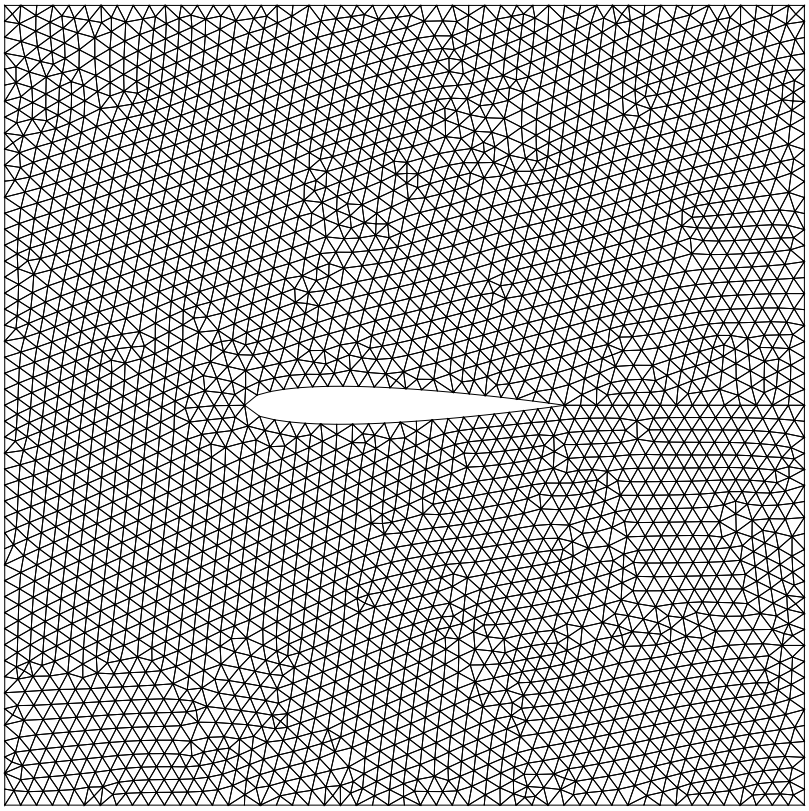


Figure 1.1: Mesh around the wing in the first example.

```
c function from the library
      integer   aft2dfront
      external  aft2dfront

c read input file that contains coordinates of boundary points
      open(1,file='../src/Tutorials/PackageAFT/examples/front1')
      read(1,*) Nvr, Nbr
      do i = 1, Nvr
         read(1,*) (vbr(j,i),j=1,2)
      end do
      do i = 1, Nbr
         read(1,*) (brd(3-j,i),j=1,2)
      end do
      close(1)

c generate a mesh using the advancing front technique
      ierr = aft2dfront(
     &           Nbr, brd, Nbr, vbr,          ! input data
     &           nv,  vrt, nt, tri, labelT,   ! output mesh
     &           nb, bnd, labelB)
```

```
     if (ierr.ne.0) stop 'error in function aft2dfront'
```

The contents of file `front1` is

```
518 518                <- number of vertices and edges
0.  0.064933           <- coordinates of vertices
0.002293  0.059187
0.007467  0.055733
0.01092  0.050573
....
0.648853  0.1954
3 4                    <- connectivity list for edges
4 5
5 6
....
235 236
```

The third example uses again the discrete representation of a domain boundary from the data file `src/Tutorials/PackageAFT/examples/front2`. It package produces the mesh shown in Fig.1.3.

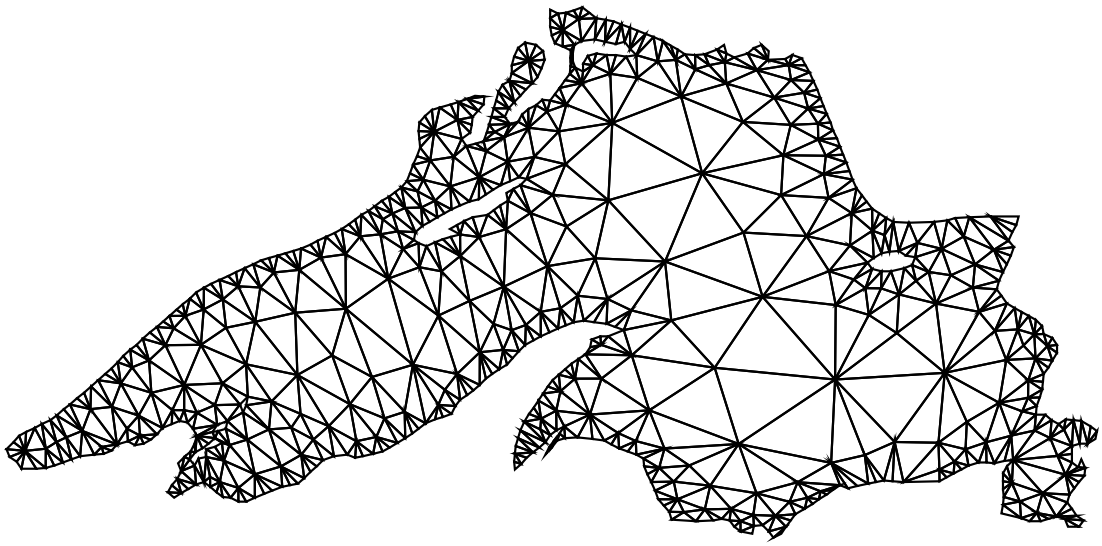


Figure 1.2: Mesh in a complex domain from example two.

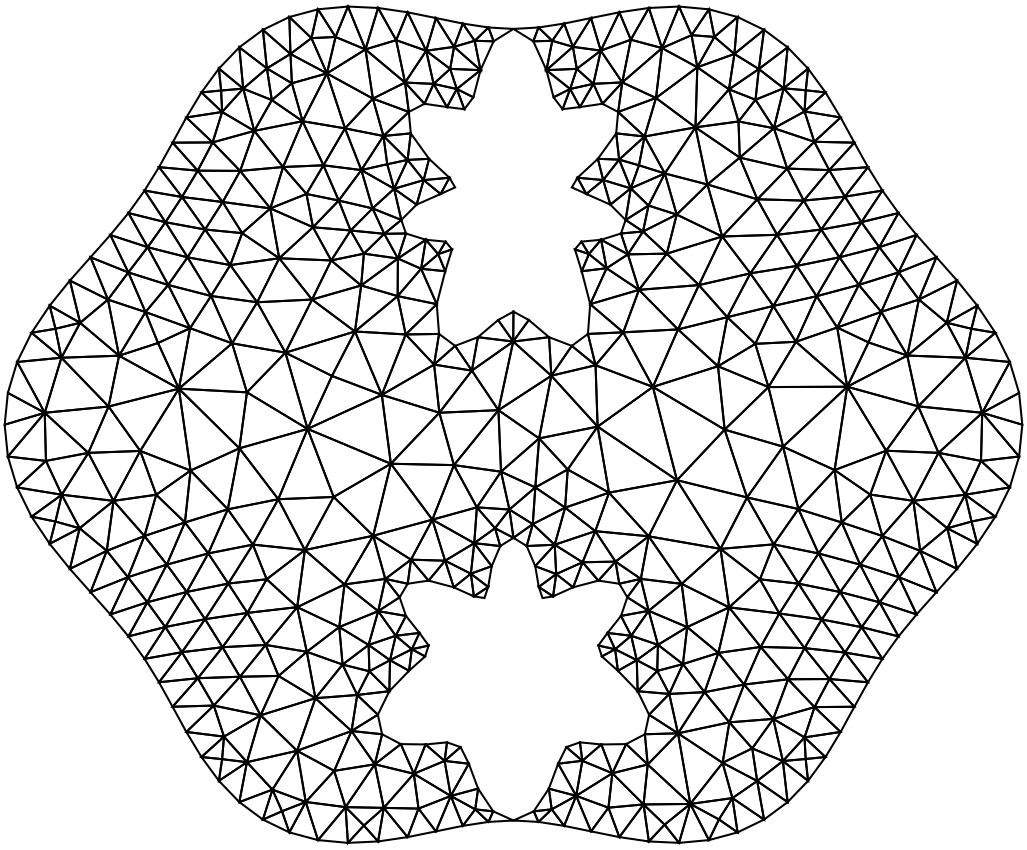


Figure 1.3: Mesh in a domain with two holes from example three.

Ani2D-RCB version 3.1 "*Windflower*"

Flexible Mesh Refining/Coarsening Tool Using Marked Edge Bisection

User's Guide for librcb2D-3.1.a

2.1 Basic features of the library

The FORTRAN-77 package Ani2D-RCB is a part of the package Ani2D. Ani2D-RCB was developed by Vadim Chugunov and Yuri Vassilevski. It is designated for hierarchical refining and coarsening of arbitrary triangular meshes. Basic restriction: prior coarsening the mesh must be refined; no coarsening is applied to an unrefined mesh.

The library *librcb2D-3.1.a* can be easily incorporated in other packages.

The library contains an initialization tool, a refinement tool, and a coarsening tool. An example of calling program is given in `Tutorials/PackageRCB/main.f`.

Mesh data

The mesh output is produced in place of the mesh input. A mesh is represented by the following data. The number of mesh nodes is `nv`. Their Cartesian coordinates are stored in the array `vrt(2,nv)`. The number of mesh triangles is `nt`. The connectivity list of triangles is stored in the array `tri(3,nt)`. The triangle labels (materials labels) are in array `labelT(nt)`.

The number of mesh boundary edges is `nb`. The connectivity list for edges is stored in the array `bnd(2,nb)`. The edge labels (boundary sides) are in array `labelB(nb)`.

2.2 Initialization

The initialization tool (`auxproc.f`) prepares auxiliary structure which defines how to bisect the triangles. In routine `InitializationRCB` all input triangles are marked for bisection according to a specific rule. The rule is based on bisecting the longest edge of each triangle. No care of mesh conformity is necessary during defining this rule. The user may change the rule in routine `InitializeMeshData`. In actual refinement, the user is free to mark for refinement any subset of triangles.

```
iERR = 0
call InitializeRCB(nt, ntmax, vrt, tri, MaxWi, iW, iERR)
if(iERR.GT.0) stop 'size of iW is too small'
```

The size of working integer array `iW` (for routines `InitializeRCB`, `LocalRefine`, and `LocalCoarse`) should be at least `11 ntmax + 7`, where `ntmax` is the maximal number of triangles, as in `tri(3,ntmax)`.

2.3 Refinement

The refinement tool `LocalRefine` (see file `refine.f`) refines the input triangulation according to the user defined rule in routine `RefineRule`. The name of this routine is the input parameter of `LocalRefine`. The output triangulation is put in place of the input triangulation. The by-product of `LocalRefine` is the logical data array `history(maxlevel*ntmax)`. It will be used later in mesh coarsening. The input current index of the refinement level `ilevel` is passed to routine `RefineRule`.

```
c ... user defined procedures
      external  RefineRule
      ...
      nlevel = 5
      Do ilevel = 1, nlevel
         Call LocalRefine (
     &        nv, nvmax, nb, nbmax, nt, ntmax,
     &        vrt, tri, bnd, labelB, labelT,
     &        RefineRule, ilevel,
     &        maxlevel, history,
     &        MaxWi, iW, iERR)
         If(iERR.GT.0) stop 'iERR.gt.0 in LocalRefine'
      End do
```

A key to controlling the refinement process is the user defined routine `RefineRule`. Here, the user defines which triangles have to be refined and how they must be refined, depending on each triangle data and the current level of refinement. The control for refinement is the marker `verf(i)`, where `i` runs from 1 to `nt`. If the marker is 0, then there is no need to refine triangle `i`. If the marker is 1, then the user wants to refine the triangle by a single bisection. If the marker is 2, then the user wants to refine the triangle by two levels of bisection into four similar subtriangles.

```
      Subroutine RefineRule (nt, tri, vrt, verf, ilevel)
      ...
      If (ilevel .le. 0) then
         Do i = 1, nt
            verf(i) =  2 ! two levels of bisection (keep the shape)
         End do
      Else  ! refine towards the diagonal y=x
         Do i = 1, nt
            xy1 = vrt(2,IPE(1,i)) - vrt(1,IPE(1,i))
            xy2 = vrt(2,IPE(2,i)) - vrt(1,IPE(2,i))
            xy3 = vrt(2,IPE(3,i)) - vrt(1,IPE(3,i))
            xy  = (xy1**2 + xy2**2) *
     &            (xy1**2 + xy3**2) *
     &            (xy2**2 + xy3**2)
            If (xy .eq. 0) then ! at least one vertex belongs to y=x
               verf(i) =  2 ! two levels of bisection (keep the shape)
            Else
               verf(i) =  0 ! no need to refine
            End if
         End do
       End if
       End
```

The example of application of the above procedure is shown in Fig.2.4.

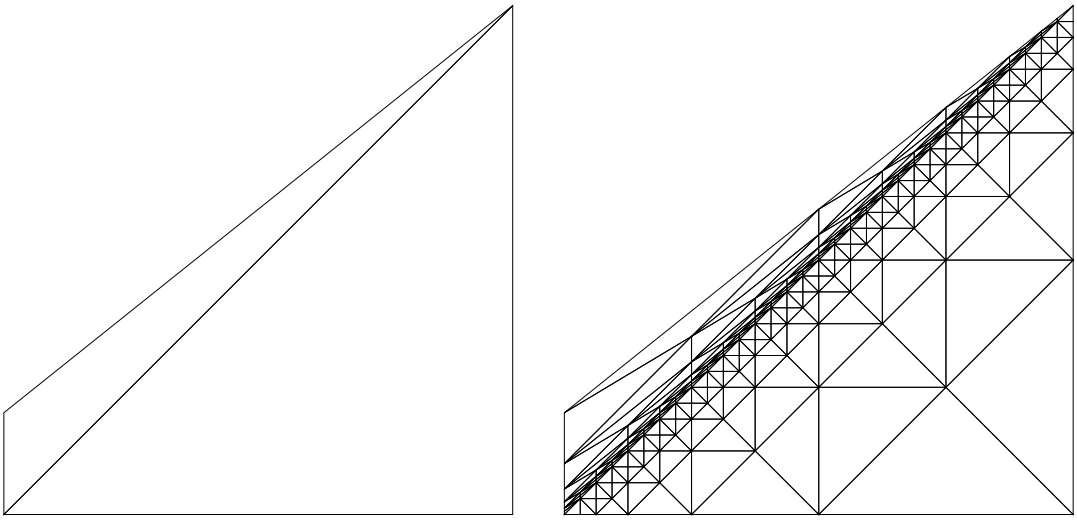


Figure 2.4: Initial mesh and locally refined mesh.

2.4 Coarsening

The coarsening tool `LocalCoarse` (see file `coarse.f`) coarsens the input triangulation according to the user defined rule in routine `CoarseRule`. The name of this routine is the input parameter of `LocalCoarse`. The output triangulation replaces the input triangulation. The by-product of routine `LocalCoarse` is the logical data array `history(maxlevel*ntmax)`. It will be used later in coarsening/refinement. The input current index of the refinement level `ilevel` is passed to `CoarseRule`.

```
c ... user defined procedures
      external  CoarseRule
      ...
      nlevel = 5
      Do ilevel = nlevel, 1, -1
         Call LocalCoarse (
     &        nv, nvmax, nb, nbmax, nt, ntmax,
     &        vrt, tri, bnd, labelB,
     &        CoarseRule, ilevel,
     &        maxlevel, history,
     &        MaxWi, iW, iERR)
         If(iERR.GT.0) stop 'iERR.gt.0 in LocalCoarse'
      End do
```

A key to controlling the coarsening process is the user defined routine `CoarseRule`. Here, the user defines which triangles have to be merged and how they must be merged, depending on each triangle data and the current level of coarsening. The control for coarsening is the marker `verf(i)`, where `i` runs from 1 to `nt`. If the marker is 0, then there is no need to coarse triangle `i`. If the marker is 1, then the user wants to merge the triangle with its

neighbor. If the marker is 2, then the user wants to merge the triangle with its neighbor and then merge the result one more time so that the result be similar to the triangle `i`.

```
 Subroutine CoarseRule (nE, IPE, XYP, verf, ilevel)
 ...
 If (ilevel .le. 0) then
    Do i = 1, nt
       verf(i) =  2 ! two levels of merging (keep the shape)
    End do
 Else ! coarse towards the diagonal y=x
    Do i = 1, nt
       xy1 = vrt(2,IPE(1,i)) - vrt(1,IPE(1,i))
       xy2 = vrt(2,IPE(2,i)) - vrt(1,IPE(2,i))
       xy3 = vrt(2,IPE(3,i)) - vrt(1,IPE(3,i))
       xy  = (xy1**2 + xy2**2) *
&            (xy1**2 + xy3**2) *
&            (xy2**2 + xy3**2)
       If (xy .eq. 0) then
          verf(i) =  2 ! two levels of merging (keep the shape)
       Else
          verf(i) =  0 ! no need to coarse
       End if
    End do
  End if
  End
```

Ani2D-MBA Version 3.1 "*Stone Flower*"[2]

Flexible Mesh Generator Using Metric Based Adaptation

User's Guide for libmba2D-3.1.a

[2]This is our first package. It is hard to carve a flower from a stone.

3.1 Introduction

The FORTRAN-77 package Ani2D-MBA (Metric Based Adaptation) is a part of the package Ani2D developed by Konstantin Lipnikov and Yuri Vassilevski. Ani2D-MBA package generates conformal anisotropic triangular meshes which are quasi-uniform in a given metric. The metric may be defined either at every point via an analytical formula or only at mesh nodes. In the first case, the user may generate a mesh with desirable properties. In the second case, the given metric is assumed to be piecewise linear, can be generated automatically and thus used in the mesh adaptation procedure.

The library *libmba2D-3.1.a* can be easily incorporated in other packages.

The input data for our generator is an initial conformal triangulation. It may be a very coarse mesh consisting of a few triangles (made by hands), or a very fine mesh produced by another mesh generator. Ani2D-MBA *changes* the initial mesh through a sequence of local modifications. This approach provides a stable algorithm for generating strongly anisotropic grids. Generalization of this approach to tetrahedral meshes has been successfully implemented in the package Ani3D-MBA. This package is freely available at `sourceforge.net/projects/ani3d`.

This document describes the structure of the package, input data, and user-supplied (optional) routines. It explains how the user can control the mesh generation process. It also presents a synthetic example showing the mesh generation process in detail.

3.2 Description of Ani2D-MBA

The aim of package Ani2D-MBA is to generate a mesh with the prescribed number of triangles which is quasi-uniform in a given tensor metric. When the metric is isotropic and constant, Ani2D-MBA will try to generate a mesh consisting of equilateral triangles. A measure of mesh quasi-uniformity is a positive number less or equal to 1 which is called the *mesh quality*. The mesh with a prescribed number of equilateral triangles of the same size (measured in the given metric) has quality 1.

3.2.1 Structure of the package

Two main FORTRAN 77 routines of Ani2D-MBA *mbaAnalytic* and *mbaNodal* are located in files `mba_analytic.f` and `mba_nodal.f`, respectively. The depending routines are contained in other files in directory `src/aniMBA`. The examples using main routines are in directory `src/Tutorials/PackageMBA`. The files

```
main_analytic.f  main_nodal.f  main_fixshape.f  main_tangled.f  main_triangle.f  time.f
```

may be modified by the user. The program in file `main_analytic.f` generates a mesh using an analytic metric. The program in file `main_nodal.f` does the same job using a user-defined metric at mesh nodes. The program in file `main_fixshape.f` makes mesh cosmetics: it fixes the elements with bad shape. The shape quality is understood in the user-defined metric. The program in file `main_tangles.f` fixes a tangled but topologically correct triangulation. The program in file `main_triangle.f` reads a mesh from file generated by J.Shewchuk's code Triangle, modifies it, and saves it in Ani2D-format. The files `main_analytic.f` and `main_nodal.f` contain routine *CrvFunction* describing a parameterization of curved boundaries and routine *MetricFunction* defining a tensor metric. Some of the models do not have curved boundaries. In this case the dummy package routine *ANI_CrvFunction* from the library can be used.

File `time.f` is a wrapper for the system call *etime* that computes CPU time. Generally speaking, this routine depends on the operational system.

For user convenience, package Ani2D-MBA is equipped with auxiliary files

```
loadM.f   loadM_other.f   saveM.f   saveM_other.f
```

Their purpose is to facilitate loading and saving of meshes. For visualization purposes, a simple service library *libview2D-3.1.a* was created. Routine *draw* and *graph_demo* from *libview2D-3.1.a* are used in `main_analytic.f` and `main_nodal.f` for generating PostScript figures. The files

```
aniMBA/Makefile   PackageMBA/Makefile
```

build the library and examples, respectively. The executable programs are put in directory `bin`. The names for compilers are defined in `src/Rules.make`. A few examples of input meshes may be found in directory `data`. This document and other documentation related to package Ani2D-MBA are located in directory `doc`.

3.2.2 Basic things the user should know

The package provides two methods for controlling mesh generation. The first method uses an analytic metric. The second method uses a piecewise linear interpolant of a discrete metric. This discrete metric is defined at mesh nodes. The package contains a few routines for accurate interpolation of functions defined on edges or over triangles to mesh nodes (see Sec. 3.8).

The package is encapsulated in two basic routines *mbaAnalytic* and *mbaNodal* that correspond to two above methods. The comments in file `src/aniMBA/mbaNodal.f` are worth to read!

3.2.3 Input data

The input data may be split into three types: data files, FORTRAN routines and control parameters.

- The input *data files* are the files containing coordinates of mesh nodes, connectivity tables for triangles and boundary edges, a parameterization of curved boundary edges, a list of fixed mesh nodes, a list of fixed mesh edges, and a list of fixed elements. The lists of fixed points, edges and elements may be empty. The list of boundary edges may be also empty. In this case, the boundary edges will be recovered by package routines. A good example illustrating format of the data file is `data/star.ani` (see Section 3.4 for a more complicated example). A data file can be accessed via routine *loadMani*.

  The mesh loader *loadMani* understands the format of input data files located in directory `data`. For other formats, a new mesh loader has to be written.

- The input *routines* are the FORTRAN 77 routines used by the package in the process of mesh generation. Examples of these routines are located in files `main_analytic.f` and `main_nodal.f`.

  An analytical metric has to be supplied for routine *mbaAnalytic*. The user should write a function similar to the function *MetricFunction* in file `PackageMBA/main_analytic.f`. For more details, we refer to comments in this file.

  A routine *CrvFunction* has to be supplied for both routines *mbaAnalytic* and *mbaNodal* if the user model has curved boundaries. If the user model does not have curved boundaries, the dummy routine *ANI_CrvFunction* supplied with the library may be used. *CrvFunction* describes parameterizations of curved boundaries. There is a way to avoid writing this routine. The user may fix the boundary points of the initial mesh provided that they give accurate representation of the boundary. Then, the final mesh approximates curved boundaries with the same accuracy as the initial mesh does.

- The input *control parameters* are the numbers that control the mesh generation. They are defined in files `main_analytic.f` and `main_nodal.f`. Two important input control parameters are

```
nEStar   - [integer] the desired number of triangles
Quality  - [real*8]  target quality for the final grid (between 0 and 1)
```

  The remaining control parameters are collected in integer variable `control`:

```
control(1) - the maximal number of skipped triangles, nEStar / 10
control(2) - the maximal number of local grid modifications, e.g. 10 nEStar
control(3) - advanced control of mesh generation (see file status.fd)
control(4) - automatic recover of missing mesh elements if positive
control(5) - the level of output information (between 0 and 9)
control(6) - flag controlling interior code termination (under development)
```

The mesh generation is an iterative process every step of which is a local modification of the current mesh. The stopping criterion for the iterative process is either the user requested final mesh quality (`Quality`) or the allowed number of local modifications `control(2)`. We recommend to set `Quality` to a value between 0.5 and 0.8 and to choose `control(2)` to be several times bigger than `nEStar`. We also recommend to set `control(1)` to a fraction of `nEStar`.

Mesh representation

Understanding details of the mesh format is one of the first steps in discovering capabilities of Ani2D-MBA. The mesh presentation includes:

```
nv - [integer] the number of points
nb - [integer] the number of boundary and interface edges
nt - [integer] the number of triangles

vrt(2,*)  - [real*8]  the Cartesian coordinates of mesh points
tri(3,*)  - [integer] connectivity list of triangles
labelT(*) - [integer] material identificator (a positive number)

bnd(2,*)  - [integer] connectivity list of boundary edges
labelB(*) - [integer] boundary identificator (a positive number)

nvfix - [integer] the number of fixed points
nbfix - [integer] the number of fixed edges
ntfix - [integer] the number of fixed triangles

fixedV(*) - [integer] list of fixed points
fixedB(*) - [integer] list of fixed edges
fixedT(*) - [integer] list of fixed triangles

CrvFunction(tc, xyc, iFnc) - user-created routine:
   tc     - [input]  parametric coordinate of point xyc
   xyc(2) - [output] Cartesian coordinate of the same point
   iFnc   - [input]  function number associated with a boundary

crv(2,*) - [real*8] linear parameterization of curvilinear edges
                    column 1 - parameter for the starting point
                    column 2 - parameter for the terminal point
iFnc(*)  - [integer] function number for computing the Cartesian coordinates
```

Parameters for interior points of a curved boundary edge are computed by linear interpolation between parameters at edge ends using the function *CrvFunction*.

Since some of the mesh data may be empty lists, the minimal mesh representation may contain only `nv`, `nt`, `vrt`, `tri` and `labelT`.

3.3 Getting started

The source code is stored in `src/aniMBA/`. In order to compile the code, the user has to set up the compilers names in `scr/Rules.make` and then to execute the following commands:

```
$ make libs
$ cd src/Tutorials/PackageMBA
$ make exe
```

After the successful compilation, the user may run one of the executables in `bin/`. The same task can be accomplished with `make run-ana` or `make run-nod`. The output may look like:

```
$ cd bin; ./aniMBA_nodal.exe

Loading mesh ../data/wing.ani
MBA: STONE FLOWER! (1997-2013), version 3.1
     Target: Quality=0.80 (nEStar:  2000, SkipE:  200, maxITR:   15000)
status.fd: +1    [ANIForbidBoundaryElements] [user]

Avg Quality = 0.9011E-02,  R/r(max,avg): 0.193E+03 0.191E+02, status =   9
ITRs:      0  Q=0.4531E-03   #V#B#T:     596     178    1037   tm=   0.00s
ITRs:  15081  Q=0.3345E+00   #V#B#T:    1074     236    1934   tm=   0.43s
Avg Quality = 0.8701E+00,  R/r(max,avg): 0.654E+02 0.113E+02, status =   9

Saving mesh save.ani
```

First, some of the input control parameters are printed out. Then, the quality of the current mesh and the numbers of vertices, edges and triangles are printed. Statistics includes average quality of mesh elements, their maximal and average stretching. Additional output goes into Postscript files `mesh_initial.ps` and `mesh_final.ps` containing figures of initial and final meshes, respectively. The files are located in directory `bin`. One way to check the contents of these files is to run

```
$ make gs-ini gs-fin
```

The program loads the input file `../data/wing.ani`. The user may change the name of the input file in the mesh loader:

```
 Call loadMani(nv, nvfix, nvmax, vrt, labelV, fixedV,
&              nb, nbfix, nbmax, bnd, labelB, fixedB,
&              nc,               Crv, iFnc,
&              nt, ntfix, ntmax, tri, labelT, fixedT,
&              "../data/wing.ani")
```

The user may play with the input control parameters in file `PackageMBA/main_analytic.f` and with the metric defined in routine *MetricFunction*. For instance, changing the metric

$$\mathcal{M}(x,y) \equiv \begin{bmatrix} F(x,y) & H(x,y) \\ H(x,y) & G(x,y) \end{bmatrix}$$

the user will learn how to control the shape of triangles.

3.4 A synthetic example

In this section, we describe in detail a process of creating a new model and generating a quasi-uniform mesh. Let the domain be the union of two circles of radius 0.2 centered at (0.2,0.5) and (0.8,0.5), respectively, and one rectangle defined by vertices (0.2,0.45), (0.2,0.55), (0.8,0.55) and (0.8,0.45). The domain is shown in Fig. 3.5.
The user has to write routine *CrvFunction* describing parameterization of circles. If the user wishes to use the mesh loader *loadMani*, he has to create an input data file. Below, we explain how to produce all these data from scratch.

Step 1. First, we chose a parameterization model. The shape of the domain dictates a natural choice for the parameterization of curvilinear parts of the boundary: each circle is parametrized by trigonometric functions. The input routine *CrvFunction* may be as follows:

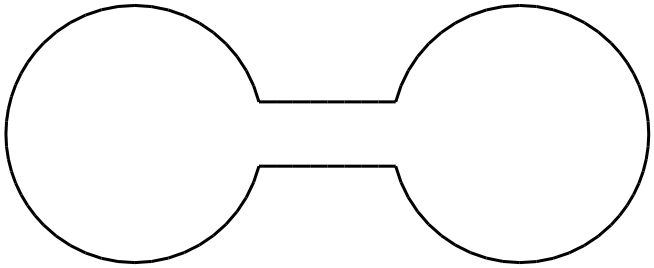


Figure 3.5: The domain to be meshed.

```
      Subroutine CrvFunction(tc, xyc, iFnc)
C  The routine computes the Cartesian coordinates of point
C  xyc from its parametric coordinate tc using function iFnc.

      Real*8  tc, xyc(2), L, H, R

      L = 0.3D0
      H = 0.1D0
      R = 0.2D0

      If(iFnc.EQ.1) Then
         xyc(1) = 5D-1 + L - R * dcos(tc)
         xyc(2) = 5D-1 + R * dsin(tc)
      Else If(iFnc.EQ.2) Then
         xyc(1) = 5D-1 - L + R * dcos(tc)
         xyc(2) = 5D-1 - R * dsin(tc)
      Else
         Write(*,'(A,I5)') 'Undefined function =', iFnc
         Stop
      End if
      Return
      End
```

Step 2. Second, we create input data file containing an initial coarse mesh. It is easy to observe that a simple mesh consisting of 12 triangles will be sufficient, see Fig. 3.6.

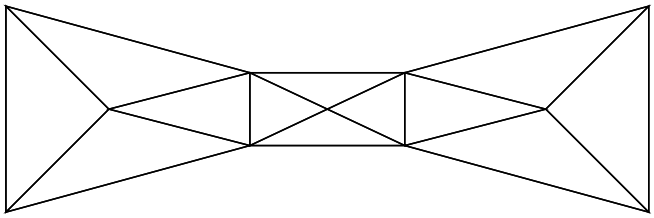


Figure 3.6: The initial coarse mesh.

The file `data/sport.ani` has a header (7 lines), followed by the list of points (11 points), list of edges (8 edges), list of triangles (12 edges) and the list of curved edges (6 edges):

```
T points:             11 (lines  10 - 20)
T edges:               8 (lines  23 - 30)
T elements:           12 (lines  33 - 44)
T curved edges:        6 (lines  47 - 52)
T fixed points:        0
T fixed edges:         0
T fixed elements:      0

 11 # of points
 0.500000000000000      0.500000000000000
 0.800000000000000      0.500000000000000
 0.200000000000000      0.500000000000000
 0.606350832689630      0.550000000000000
 0.941421356237310      0.641421356237310
 0.941421356237310      0.358578643762690
 0.606350832689630      0.450000000000000
 0.393649167310370      0.450000000000000
 5.857864376269000E-002 0.358578643762690
 5.857864376269000E-002 0.641421356237310
 0.393649167310370      0.550000000000000

 8 # of edges
  4  5 1 0 1
  5  6 2 0 1
  6  7 3 0 1
  7  8 0 0 2
 11  4 0 0 2
  8  9 4 0 3
  9 10 5 0 3
 10 11 6 0 3

 12 # of elements
 2  4  5  1
 2  5  6  1
 2  6  7  1
 2  7  4  1
 1  7  8  1
 1  8 11  1
 1 11  4  1
 1  4  7  1
 3  8 11  1
 3 11 10  1
 3 10  9  1
 3  9  8  1

 6 # of curved edges
 0.252680255142080 2.35619449019230  1
 2.35619449019230  3.92699081698720  1
 3.92699081698720  6.03050505203750  1
 0.252680255142080 2.35619449019230  2
 2.35619449019230  3.92699081698720  2
 3.92699081698720  6.03050505203750  2

 0 # number of fixed points
```

```
0 # number of fixed edges

0 # number of fixed elements
```

- Some of the mesh nodes may be relocated or destroyed in a process of the mesh generation. However, the domain boundary requires that four nodes (intersections of the rectangle with the circles) remain untouched. In order to provide this information, we need the list of fixed points. This list may be generated automatically if the boundary is colored properly. More precisely, if a point is shared by two edges with different color, it will be automatically added to the list of fixed points. Note, that we use three colors to mark boundary edges (see the last column).
- The third column in the list of edges indicates that six of boundary edges are part of the curvilinear boundary. It is reasonable to mark the edges with three labels (colors) associated with one rectangle and two circles. In each row, the first two entries are the node indexes, the third entry is the position in the list of curved edges, the fourth is dummy, and the fifth is a label (color) of the edge.
- The list of curved edges contains the starting and ending parameter values and a positive integer which is the function number `iFnc` in routine *CrvFunction*. It is very important to guarantee that evaluation of *CrvFunction* gives exactly the same mesh coordinates as in the input file. For example, let us take `tc = 0.252680255142080` and `iFnc = 1` from the first row. Then, the routine should give the Cartesian coordinates of the 4th mesh node. The acceptable error is 10^{-8}.

Step 3. Third, we have to choose an analytic metric in which the final mesh be quasi-uniform. In other words, we have to write routine *MetricFunction*:

```
Integer Function MetricFunction(x, y, Metric)
Real*8  x, y, Metric(2, 2)

Metric(1,1) = 1D0
Metric(2,2) = 1D0
Metric(1,2) = 0D0
Metric(2,1) = 0D0

MetricFunction = 0
Return
End
```

Step 4. Fourth, we set up the control parameters:

```
Integer   nEStar
Real*8    Quality

nEStar  = 1000
Quality = 8D-1
```

Thus, we plan to generate a mesh with approximately 1000 triangles. Each of the triangles will be very close to an equilateral triangle.

Step 5. The final step is to collect all routines in a single file, e.g. `PackageMBA/main_analytic.f`, compile the package and execute the code (`# make exe run-ana`). We get the mesh shown in Fig. 3.7.

3.5 Two more examples

The first geometric model is shown in Fig.3.8 (left picture). The vertical sides of the model are partly curved. The left part is parametrized as follows:

$$x = 0.2 - 2\,t\,(0.3 - t), \quad y = t, \qquad t \in [0, 0.3].$$

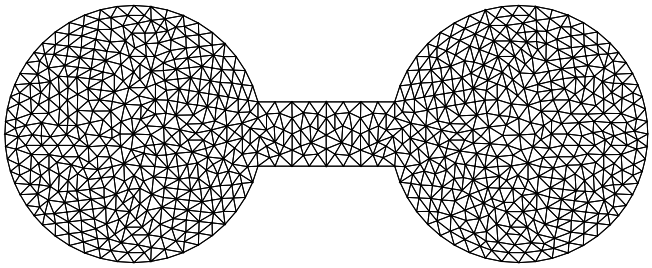


Figure 3.7: The final mesh.

The curved part of the right side of the model is parametrized in a similar way:

$$x = 1 - 2\,(1-t)\,(t-0.7), \quad y = t, \qquad t \in [0.7, 1].$$

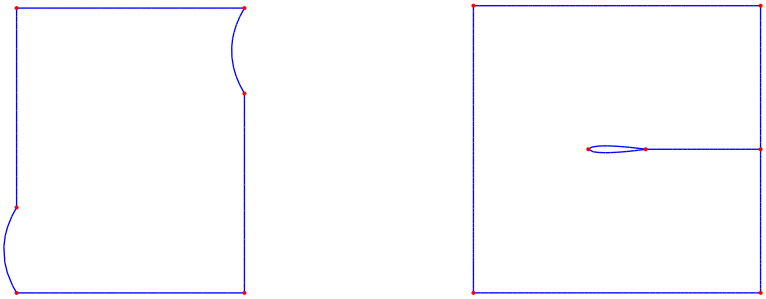


Figure 3.8: Two models: the square with curved sides and the wing.

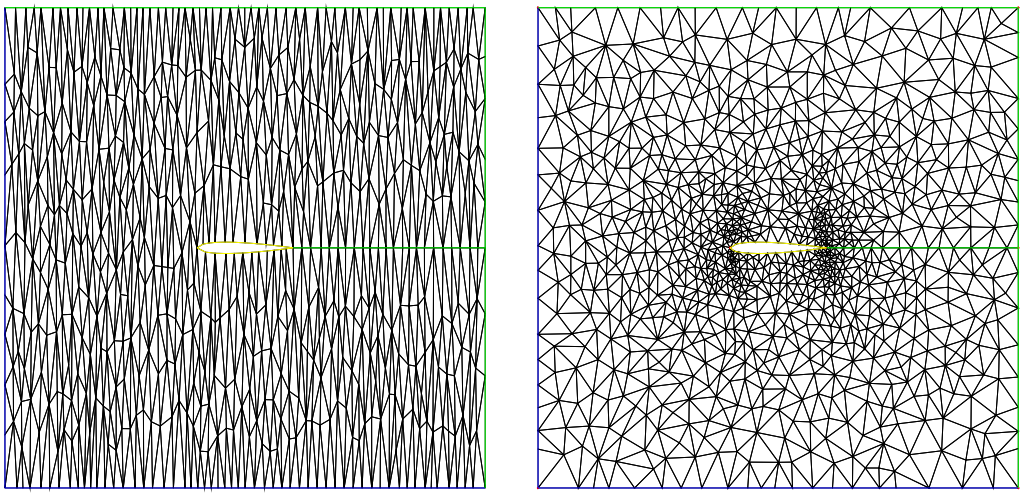


Figure 3.9: The initial and final meshes of the model `data/wing.ani`.

The second model is shown in Fig.3.8 (right picture). We use one parameterization for the wing and the other parameterization for the slit behind the tail. The file `PackageMBA/main_analytic.f` defines a

metric such that the final mesh refines isotropically towards the leading and trailing edges of the wing, see Fig.3.9.

3.6 Useful features of Ani2D-MBA, version 3.1

We improve continuously robustness and efficiency of the code, make it more user friendly and add a few new features in each release. The most important features are listed below:

1. The initial mesh may be tangled. In this case, the user may add `ANIUntangleMesh` defined in `src/aniMBA/status.fd` to the input parameter `control(3)` to untangle the input mesh.
2. The shape of bad element may be fixed by calling a specialized routine `mbaFixShape`. This routine uses different definition of element's quality which controls the shape in the user-defined metric and which is not sensitive to the size of the element.
3. Using two packages Ani2D-MBA and Ani2D-LMR, it is possible to generate meshes that minimize either maximum or L^p-norm of the interpolation error or its gradient, $p > 0$, for a given function.
4. The complete list of available mesh features is in file `src/aniMBA/status.fd`. Here are the most important features:
   - The user may freeze boundary points. This preserves important geometric features for both isotropic and anisotropic metrics. Fig.3.10 illustrates this feature. The fixed boundary points (red dots) prevent sharp boundary from smearing. (*The initial mesh was found on the website of Jonathan Shewchuk.*)
   - The user may freeze boundary edges and/or mesh elements. This allows to preserve mesh structure in important regions (e.g., in boundary layers).
   - The interfaces between materials with different labels (labelT) are recovered and preserved automatically.
   - The vertices of corners smaller than 30° are marked automatically as fixed points.

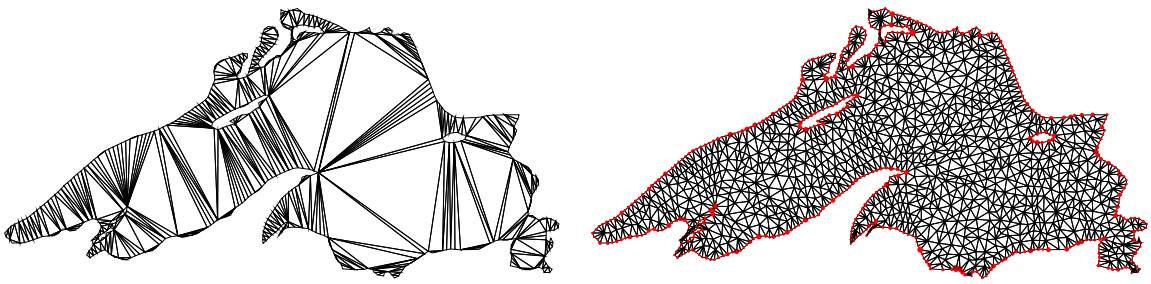


Figure 3.10: The initial and final meshes of the model `data/country.ani`.

5. The library *libmba2D-3.1.a* contains routine *DG2P1* which maps a discontinuous piecewise linear function defined on mesh edges onto a continuous piecewise linear function defined at mesh points (see `src/aniMBA/ZZ.f` for more detail).
6. The library *libmba2D-3.1.a* contains a few routines *listX2Y* which create connectivity lists $X \to Y$ for mesh objects X and Y such as elements, edges, boundary edges, and points (see `src/aniMBA/maps.f` for more detail).
7. If the user have not installed the package LAPACK, the necessary routines are in directories `src/lapack` and `src/blas`. Double precision libraries `liblapack-3.2.a, liblapack_ext-3.2.a, libblas-3.2.a` are generated with the command "`make lib`".

3.7 How to use library libmba2D-3.1

Here we describe one of the main routines, *mbaNodal*, from library `libmba2D-3.1.a`. The other routine, *mbaAnalytic*, is similar to *mbaNodal*, except that the parameter `Metric` (two-dimensional array) is replaced by parameter `MetricFunction` (function defining analytic metric).

```
      Call mbaNodal(
     &      nv, nvfix, nvmax, vrt, labelv, fixedV,
     &      nb, nbfix, nbmax, bnd, labelB, fixedB,
     &      nc,               Crv, iFnc, CrvFunction,
     &      nt, ntfix, ntmax, tri, labelT, fixedT,
     &      nEStar, Quality, control, Metric,
     &      MaxWr, MaxWi, rW, iW, iERR)
```

Most of the parameters were described in Section 3.2 (see file `src/aniMBA/mba_nodal.f` for more detail). The details on the other input parameters are below:

```
I        nvmax - [integer] maximal number of points
N        nbmax - [integer] maximal number of boundary and interface edges
P        ntmax - [integer] maximal number of triangles
U
T        nbfix - [integer] the number of fixed edges
         ntfix - [integer] the number of fixed triangles
P
A        fixedB(nbfix) - [integer] list of fixed edges
R        fixedT(ntfix) - [integer] list of fixed triangles
A
M        nEStar - [integer] the desired number of triangles
E
T        MaxWr - [integer] maximal memory allocation for real*8 array rW
E        MaxWi - [integer] maximal memory allocation for integer array iW
```

Here we collect parameters which are both input and output:

```
I        nv - [integer] the number of points
N        nb - [integer] the number of boundary and interface edges
P        nt - [integer] the number of triangles
U
T        vrt(2, nvmax) - [integer] list of points
/        tri(3, ntmax) - [integer] list of triangles
O        labelT(ntmax) - [integer] labels of triangles (material id)
U
T        bnd(2, nbmax) - [integer] list of boundary and interface edges
P        crv(2, nbmax) - [real*8]  parameterizations of curved edges
U        iFnc(nbmax)   - [integer] list of parameterization functions
T
         nvfix         - [integer] the number of fixed points
         fixedV(nvfix) - [integer] list of fixed points
P
A        Metric(3,nvmax) - Real*8 array containing the metric defined
R               at mesh points. The metric is 2x2 positive definite tensor:
E                    M11   M12
M                    M12   M22
E               Each column of this array stores the upper triangular
T               entries in the following order: M11, M22, and M12.
```

```
E
R       Quality - [real*8] quality of the initial/final mesh
s
        rW(MaxWr) - [real*8]  working array
        iW(MaxWi) - [integer] another working array
```

3.8 Useful routines

The library *libmba2D-3.1.a* has a few routines that can be useful in many other projects. Most of the input parameters in these routines are explained above.

- Uniform mesh refinement and linear interpolation of nodal function `F(LDF, *)`. The size of working integer array `iW` is at least `3 nt + nv` where `nv, nt` are input values. Routines returns the map `map_tr(3, nt)` from triangles to edges.

```
 Subroutine uniformRefinement(nv, nvmax, nb, nbmax, nt, ntmax,
&                             vrt, bnd, labelB, tri, labelT,
&                             CrvFunction, Crv, iFnc, map_tr,
&                             F, LDF, iW, MaxWi)
```

- The experimental routine `Delaunay` builds the Delaunay triangulation from the existing triangulation by swapping edges in pairs of triangles. The size of working integer array `iW` is `6 nt + nv`.

```
Subroutine Delaunay(nv, nt, vrt, tri, MaxWi, iW)
```

- Routine `orientBoundary` orients the external boundary of the input mesh in such a way that the computational domain is located on the left when we move from the first edge point to the second one. In other words `bnd(1, *)` and `bnd(2, *)` are swapped if necessary. The size of working integer array `iW` is `3 nt + 2 nb + nv`.

```
Subroutine orientBoundary(nv, nb, nt, vrt, bnd, tri, iW, MaxWi)
```

- Routine `DG2P1` maps a discontinuous piecewise linear mesh function `fDG` defined on edges onto a continuous piecewise linear mesh function `fP1` defined at vertices. We use the ZZ method for the interpolation. The size of working integer array `iW` is `3 nt + nv`.

```
Subroutine DG2P1(nv, nt, vrt, tri, map_tr, fDG, fP1, MaxWi, iW, iERR)
```

- Routine `listE2R` creates a map `map_tr` from triangles to mesh edges. The routine counts mesh edges. For an element `t`, `map_tr([1:3], t)` give indexes of three edges in the order defined by the connectivity list `tri`. For example, the first edge is `[tri(1,t), tri(2,t)]`. The working integer arrays are `nEP(nv)` and `IEP(3 nt)` (see `src/aniMBA/maps.f` for more detail).

```
Subroutine listE2R(nv, nr, nt, tri, map_tr, nEP, IEP)
```

- Routine `listR2R` creates connectivity lists `nRR` and `IRR` for mesh edges. The routine counts the number of mesh edges, `nr`. Then, `nRR(i) - nRR(i-1)` (`nRR(1)` when `i=1`) gives the total number of edges in triangles sharing the edge `i`. The corresponding edge numbers are saved in array `IRR` in positions `nRR(i-1) + 1` to `nRR(i)`. The size of working integer array `iW` is `9 nt` (see `src/aniMBA/maps.f` for more detail).

```
Subroutine listR2R(nv, nr, nt, MaxL, tri, nRR, IRR, iW)
```

- File `src/aniMBA/maps.f` contains more routines for creating other maps between various mesh objects, for example, edges to points, points to points, elements to boundary edges, elements to elements, etc. The routine *listConv* convolutes two given connectivity lists. The routines *backReferences* and *reverseMap* create reverse maps for a given structured and unstructured maps, respectively. For instance, *backReferences* takes the structured map `tri` from elements to points and creates the unstructured maps `nEP` and `IEP` from points to elements.

- The experimental routine `smoothingMesh` applies the Laplacian smoothing to the mesh. For each mesh vertex, a new position is chosen based on local information (the position of its neighbors) and the vertex is moved there. The size of the working integer array `iW` is `2 nv + 3 nt + 90`.

```
Subroutine smoothingMesh(nv, nt, vrt, tri, MaxWi, iW)
```

- Routine `mbaFixShape` improves shapes of "bad" elements and produces a shape-regular triangulation. An example of using this routine is given in file `Tutorials/PackageMBA/main_fixshape.f`.

3.9 FAQ

- Q. The mesh generator does not refine the input mesh. Why?
  A. There are two cases when the code may do nothing. First, the number of mesh elements whose quality is limited by geometry (e.g. thin layers) is bigger then the control parameter `control(1)`. The remedy is to increase this parameter. Second, a severe anisotropic input metric does not allow to insert new mesh points in a very coarse mesh. The simple remedy is to refine mesh using an isotropic metric and then switch to the anisotropic metric.

- Q. The mesh generator produces different grids on different computers. Why?
  A. The output of the mesh generator may depend on a computer arithmetic. The order of local mesh modifications depends on round-off errors and may be computer-dependent.

- Q. The final mesh quality is very small. Why?
  A. The mesh quality equals to quality of the worst triangle in the mesh. In some cases, the shape of near-boundary triangles is driven mainly by the geometry. A possible remedy is either to increase the number `nEStar` of desired triangles or to fix a possible contradiction between the boundary and the metric. An example of such a contradiction is a quasi-uniform mesh in `data/Dam.*`. Another reason for low mesh quality may be strong jumps in the metric. If the metric is isotropic, the optimal triangles are equilateral ones. The triangle size is defined by the metric value. Therefore, the optimal size is changed strongly across lines of metric discontinuity.

- Q. The mesh generator is stopped immediately with diagnostics saying that the parameterization is wrong. Why?
  A. There is a contradiction between input data in arrays `crv, iFnc` and `vrt`.

- Q. The number of triangles in the final mesh is never equal to `nEStar`. Why?
  A. The equality is achieved if and only if `Quality = 1` and the computational domain may be covered by equilateral (in the user given metric) triangles. Apparently, it is possible only in very special cases.

- Q. Is it possible to use *libmba2D-3.1.a* in an adaptive loop?
  A. Yes. Use `make libs` to generate a few libraries that may be linked with other codes. This release contains a few examples of solving partial differential equations on adaptive grids (see `src/Tutorials/MultiPackage` for more detail).

- Q. Why does *libmba2D-3.1.a* fail to untangle the mesh?
  A. This may happen when the initial mesh is either topologically incorrect or extremely tangled. The second case is curable. Try to run the code with the identity metric or/and change significantly the desired number of mesh elements.

- Q. I do not understand why *libmba2D-3.1.a* fails to generate a mesh.
  A. The authors are interested in any feedback from users. To report a problem, please send an email to either `lipnikov@gmail.com` or `yuri.vasilevski@gmail.com`. To help us to fix the problem, please attach file `main_analytic.f` or `main_nodal.f` and files containing the input mesh.

References

1. Yu.Vassilevski and K.Lipnikov, An adaptive algorithm for quasi-optimal mesh generation, *Computational Mathematics and Mathematical Physics* (1999) **39**, No.9, 1468–1486.

2. A.Agouzal, K.Lipnikov, Yu.Vassilevski, Adaptive Generation of Quasi-optimal Tetrahedral Meshes, *East-West Journal* (1999) **7**, No.4, 223–244.

3. K.Lipnikov, Y.Vassilevski, Parallel adaptive solution of 3D boundary value problems by Hessian recovery, *Comput. Methods Appl. Mech. Engrg.* (2003) **192**, 1495–1513.

4. K.Lipnikov, Yu. Vassilevski, Optimal triangulations: existence, approximation and double differentiation of P_1 finite element functions, *Computational Mathematics and Mathematical Physics* (2003) **43**, No.6, 827–835.

5. K.Lipnikov, Yu.Vassilevski, On a parallel algorithm for controlled Hessian-based mesh adaptation. Proceedings of 3rd Conf. Appl. Geometry, Mesh Generation and High Performance Computing, Moscow, June 28 - July 1, Comp. Center RAS, V.1, 2004, 154–166.

6. K.Lipnikov, Yu.Vassilevski, On control of adaptation in parallel mesh generation. *Engrg. Computers* (2004) **20**, 193–201.

7. K.Lipnikov, Yu.Vassilevski, Error bounds for controllable adaptive algorithms based on a Hessian recovery. *Computational Mathematics and Mathematical Physics* (2005) **45**, 1374–1384.

8. K.Lipnikov, Yu.Vassilevski, Analysis of Hessian recovery methods for generating adaptive meshes. *Proceedings of 15th International Meshing Roundtable*, P.Pebay (Editor), Springer, Berlin, Heidelberg, New York, 2006, pp.163–171.
9. A.Agouzal, Yu.Vassilevski, Minimization of gradient errors of piecewise linear interpolation on simplicial meshes. *Comp.Meth. Appl.Mech.Engnr.* (2005) **199**, 2195–2203.

10. A.Agouzal, K.Lipnikov, Yu.Vassilevski, Hessian-free metric-based mesh adaptation via geometry of interpolation error. *Computational Mathematics and Mathematical Physics* (2010) **50**, 124–138.

11. A.Agouzal, K.Lipnikov, Yu.Vassilevski, On optimal convergence rate of finite element solutions of boundary value problems on adaptive anisotropic meshes. *Mathematics and Computers in Simulation* (2011) **81**, 1949–1961.

Chapter 2

DISCRETIZATION PACKAGEs

Ani2D-FEM version 3.1 "*Sunflower*"

Flexible Generator of Finite Element Systems on Triangular Meshes

User's Guide for libfem2D-3.1.a

4.1 Introduction

The FORTRAN-77 package Ani2D-FEM is developed by Konstantin Lipnikov and Yuri Vassilevski. It is designated for generating finite element matrices on triangular meshes. The package allows to build elemental matrices for variety of finite elements, modify these matrices, assemble them, and impose boundary conditions.

The package Ani2D-FEM differs from other similar packages by providing a very flexible interface for incorporating problem coefficients in elemental matrices. In addition, the elemental matrices are understood in a very broad sense. They may involve different types of finite elements.

Assembling of finite element matrices is the most tedious exercise, especially for high-order methods or system of equations. Our package provides two assembling routines that hide from the user a complicated data management. As a compromise between flexibility and richness of the assembling tools, we decided to move boundary conditions outside of our routines. The user will have to impose them either locally or globally. Our examples explain how to impose boundary conditions on elemental matrices.

The library *libfem2D-3.1.a* can be easily incorporated in other packages.

This document describes the structure of the package, input data, and user-supplied routines. It presents a few examples illustrating details of the package.

4.2 Description of Ani2D-FEM

4.2.1 Elemental finite element matrix

The core of the package is routine *fem2Dtri* which computes elemental matrix corresponding to the bilinear form

$$< D\, Op_A(u),\, Op_B(v) > \tag{4.1}$$

where D is a tensor, Op_A and Op_B are linear first-order or zero-order differential operators, and u and v are finite element basis functions. Let us consider each of these three ingredients in detail.

Finite elements. Below is the list of implemented finite elements (see file `fem2Dtri.f` for more detail). We use the following convention for the order of unknowns in a finite element. The vertex-based unknowns are denoted by V_1, V_2, and V_3, following the order of vertices. The edge-based unknowns are denoted by E_1, E_2, and E_3, following the order of edges in the triangle. The edges are ordered as follows: V_1-V_2, V_2-V_3 and V_3-V_1. The element-based unknown is denoted by T_1.

FEM_P0	The piecewise constant (T_1).
FEM_P1	The continuous piecewise linear (V_1, V_2, V_3).
FEM_P2	The continuous piecewise quadratic $(V_1, V_2, V_3,\ E_1, E_2, E_3)$.
FEM_P3	The continuous piecewise cubic $(V_1, V_2, V_3,\ E_1, E_2, E_3,\ E_1, E_2, E_3,\ T_1)$.
FEM_P4	The continuous piecewise quartic $(V_1, V_2, V_3,\ E_1, E_2, E_3,\ E_1, E_2, E_3,$ $E_1, E_2, E_3,\ T_1, T_1, T_1)$.

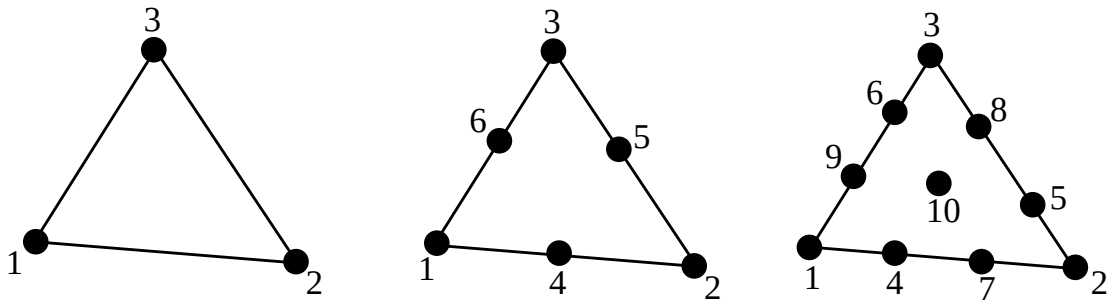


Figure 4.1: From left to right: linear, quadratic and cubic Lagrange elements.

Local enumeration of degrees of freedom in the first three Lagrange elements is shown in Fig. 4.1. Note that enumeration of edge-based unknowns is contiguous and goes in groups of three. These properties are used in our assembling routines.

FEM_P1vector	The continuous vector piecewise linear $(V_1, V_2, V_3,\ V_1, V_2, V_3)$.
FEM_P2vector	The continuous vector piecewise quadratic. The unknowns are ordered first as in the quadratic element and then by the space dimension.
FEM_P2reduced	The Bernardi-Fortin-Raugel finite element, the continuous vector piecewise linear functions enriched by edge bubbles $(V_1, V_2, V_3,\ E_1, E_2, E_3\ V_1, V_2, V_3)$.
FEM_MINI	The continuous vector piecewise linear functions enriched by a central bubble $(V_1, V_2, V_3,\ E_1,\ V_1, V_2, V_3,\ E_1)$
FEM_RT0	The lowest order Raviart-Thomas finite elements
FEM_BDM1	The lowest order Brezzi-Douglas-Marini finite elements
FEM_CR1	The Crouzeix-Raviart finite element.
FEM_CR1vector	The vector Crouzeix-Raviart finite element. The unknowns are ordered first by vertices and then by the space directions (x and y).

Fig. 4.2 shows that local enumeration of edge-based unknowns goes again in groups of three.

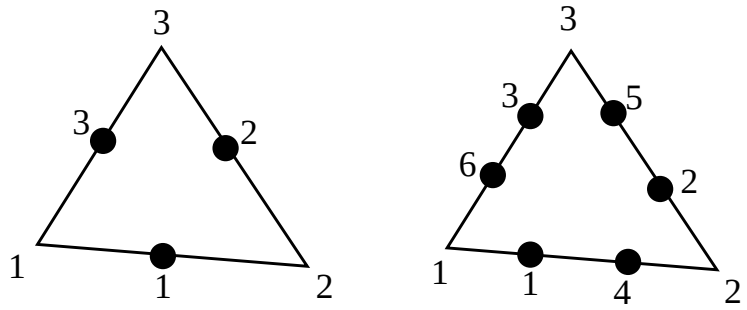


Figure 4.2: From left to right: RT_0 and BDM_1 elements.

Discrete operators Op_A and Op_B. Here is the list of available discrete operators (see comments in file `fem2Dtri.f` for additional details):

IDEN	identity operator
GRAD	gradient operator
DIV	divergence operator
CURL	rotor operator
DUDX	partial derivative d/dx
DUDY	partial derivative d/dy
DUDN	partial derivative in direction of an exterior normal

Not all operators can be applied to all finite elements, for instance `DIV(FEM_P1)` does not make sense because the divergence operator requires a finite element of the vector-type. If this happens, the code execution will be terminated.

Tensors D. The package allows a few types of tensor D to make computations more efficient. Here is the list of supported tensors:

TENSOR_NULL	identity tensor
TENSOR_SCALAR	scalar tensor
TENSOR_SYMMETRIC	symmetric tensor
TENSOR_GENERAL	general (rectangular or non-symmetric) tensor

For the considered first-order differential operators, the maximal rand of a tensor that makes the operations in (4.1) consistent is four. Let (u_x, u_y) be a vector finite element function. The fourth-rank tensor can be represented by a 4×4 matrix that acts on vector of type $(\partial u_x/\partial x, \partial u_x/\partial y, \partial u_y/\partial x, \partial u_y/\partial y)$. For example, for a linear elasticity problem with the Lame coefficients μ and λ, the tensor D is as follows:

$$D = \mu \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} + \lambda \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}.$$

The package uses several quadrature formulae:

order = 1	quadrature formula with one central point
order = 2	quadrature formula with 3 points on triangle edges
order = 5	quadrature formula with 7 points inside triangle
order = 6	quadrature formula with 12 points inside triangle
order = 9	quadrature formula with 19 points inside triangle
order = 13	quadrature formula with 37 points inside triangle

A solution of nonlinear problems is usually based on a Newton-type iterative method. In this case the tensor D may depend on a discrete function (e.g. approximation from the previous iterative step). If so, evaluation of D may be a complex procedure and may require additional data. We provide the flexible machinery for incorporating additional data into the user written function for calculating D. Let *Dcoef* be the name of this function. It has the following format:

```
      Integer Function Dcoef(x, y, label, dDATA, iDATA, iSYS, Coef)

C     The function returns type of the tensor Coef (see the table above).
C
C     (x, y) - [input] Real*8 Cartesian coordinates of a 2D
C              point where tensor Coef should be evaluated
C
C     label  - [input] Integer label of a mesh element
C
C     dDATA(*) - [input] Real*8  user given data
C     iDATA(*) - [input] Integer user given data
C
C     iSYS   - [input/output] integer buffer for information exchange:
C                  iSYS(1) = iD [output] number of rows in Coef
C                  iSYS(2) = jD [output] number of columns in Coef
C
C                  iSYS(3) [input] triangle global index
C                  iSYS(4:6) [input] 1st, 2nd, and 3rd vertex global indexes
C
C                  iSYS(7) [input] 1st edge global index (vertices 1 and 2)
C                  iSYS(8) [input] 2nd edge global index (vertices 2 and 3)
C                  iSYS(9) [input] 3rd edge global index (vertices 3 and 1)
C
C                  iSYS(10) [input] total number of points
C                  iSYS(11) [input] total number of edges
C                  iSYS(12) [input] total number of triangles
C
C                  iSYS(13) : neighboring triangle to edge 12 (for DG methods)
C                  iSYS(14) : neighboring triangle to edge 23
C                  iSYS(15) : neighboring triangle to edge 31
C
```

```
C             iSYS(7:9) and iSYS(11) may be zero if edge degrees of
C             freedom are not used.
C
C    Coef(4,jD) - [output] Real*8 matrix with the leading dimension 4
```

To compute entries of the tensor `Coef`, the user may use the triangle index `iSYS(3)` and arrays `dDATA, iDATA`. Here are a few examples.

- isotropic diffusion coefficient. The user has to set iD = jD = 1, Dcoef = TENSOR_SCALAR and to return the diffusion value Coef(1,1) at the point (x, y).
- anisotropic diffusion coefficient. The user has to set iD = jD = 2, Dcoef = TENSOR_SYMMETRIC, and to return diffusion tensor (2x2 matrix with entries Coef(1,1), Coef(1,2), Coef(2,1), Coef(2,2)) at the point (x, y).
- convection coefficient. The user has to set iD = 2, jD = 1, Dcoef = TENSOR_GENERAL, and to return the velocity transposed vector values Coef(1,1), Coef(2,1) at the point (x, y).

Now we are ready to call routine *fem2Dtri* which computes elemental matrix A:

```
      Call FEM2Dtri(XY1, XY2, XY3,
     &              OpA, FemA, OpB, FemB,
     &              label, Dcoef, dDATA, iDATA, iSYS, order,
     &              LDA, A, nRow, nCol)

C     XYi(2)     - [input] Real*8 Cartesian coordinates of i-th vertex
C     OpA, OpB   - [input] operators in (1), integers
C     FemA, FemB - [input] type of finite elements from (1), integers
C
C     Dcoef      - [input] external integer function using label, dDATA and iDATA
C     order      - [input] order of the numeric quadrature, integer
C
C     LDA        - [input] leading dimension of matrix A(LDA, LDA)
C     A(LDA,LDA) - [output] Real*8 finite element matrix A
C     nRow       - [output] the number of rows of A
C     nCol       - [output] the number of columns of A
```

The following rules are applied for numbering unknowns within the elemental matrix:

- First, basis functions associated with vertices (if any) are numerated in the same order as the vertices r_i, $i = 1, 2, 3$ (input parameters XY1, XY2, XY3).
- Second, basis functions associated with edges (if any) are numerated in the order of edges r_{12}, r_{23} and r_{13}.
- Third, basis functions associated with element (if any) are numerated.
- The vector basis functions with 2 degrees of freedom per a mesh object (vertex, edge) are enumerated first by the corresponding mesh objects and then by the space coordinates, first x and then y. Note that this rule is NOT applied only to element-based degrees of freedom.

For example if `FEM_A = FEM_B = FEM_P2vector`, the following ordering is used:

```
Vdof,Vdof,Vdof, Rdof,Rdof,Rdof, Vdof,Vdof,Vdof, Rdof,Rdof,Rdof,
```

where `V` and `R` stand for vertex-based and edge-based degrees of freedom.

In order to compute a linear form representing an elemental right-hand side, we can use the following trick:

$$f(v) =< D_{rhs}\, FEM\_P0,\, v > \tag{4.2}$$

where D_{rhs} represents the right-hand side function f:

```
      Call FEM2Dtri(XY1, XY2, XY3,
     &              IDEN, FEM_P0, IDEN, FemB,
     &              label, Drhs, dDATA, iDATA, iSYS, order,
     &              LDA, F, nRow, nCol)
```

4.2.2 Extended elemental finite element matrix

Each elemental matrix may be a combination of a few *fem2Dtri* calls reflecting the fact that the bilinear form (4.1) may consist of a few simple forms, for example, refer to the Stokes problem. Degrees of freedom in the extended elemental matrix are characterized by arrays templateR and templateC:

```
      Subroutine FEM2Dext(XY1, XY2, XY3,
     &                    lbE, lbF, lbP, dDATA, iDATA, iSYS,
     &                    LDA, A, F, nRow, nCol,
     &                    template, templateC)

C     XYi(2) - [input] Real*8 Cartesian coordinates of i-th point
C
C     lbE    - [input] ID of the triangle (material label)
C     lbF(3) - [input] 0 or IDs of triangle edges (boundary labels)
C     lbP(3) - [input] IDs of triangle nodes
C
C     dDATA(*) - [input] Real*8  user given data
C     iDATA(*) - [input] Integer user given data
C
C     iSYS   - [input/output] integer buffer for providing triangle
C                information. Its entries are described above
C
C     LDA       - [input] leading dimension of matrix A
C     A(LDA, *) - [output] Real*8 elemental matrix, degrees of freedom
C                 are ordered according to templateR and templateC
C     F(nRow)   - [input] Real*8 vector of the right-hand side
C
C     nRow      - [output] the number of rows in A
C     nCol      - [output] the number of columns in A
C
C     templateR(nRow) - [output] Integer array of degrees of freedom for rows
C     templateC(nCol) - [output] Integer array of degrees of freedom for columns
```

Note the `iSYS(7:9)` and `iSYS(11)` may be zero if edge degrees of freedom are not used. The number of degrees of freedom in `templateR` and `templateC` associated with vertices and edges must be a multiple of three.

Admissible values for arrays `templateR` and `templateC` are defined in file *fem2Dtri.fd*. Including this header file, the user may indicate a nodal degree of freedom as follows

```
templateR(i) = Vdof
```

The degrees of freedom on edges are indicated either by `Rdof` or `RdofOrient`. The former corresponds to a scalar unknown (e.g. a Lagrange multiplier in a hybrid mixed finite element) that has no orientation. The latter corresponds to a vector unknown (e.g. the Raviart-Thomas finite element basis function) that has orientation. Finally, a degree of freedom associated with a mesh element is indicated by `Edof`.

In general, there exists some flexibility in global ordering of unknowns. However, they must be grouped according to their geometric location. For example, the first three unknowns associated with points may go to the first group of point-based unknowns. Next three point-based unknowns may go to the second group. The edge-based unknowns must be also placed in groups of three. The following rule is extremely important, especially for high-order finite element discretizations:

- The multiple edge-based degrees of freedom corresponing to the same scalar finite element must be placed together and marked by the same additional bit, see below. The local ordering in this supergroup must follow the order of edge-based unknowns provided by routine *fem2Dtri*.
- All element-based unknowns must be placed in the same group.

Additional bits are added to entries of arrays `templateR` and `templateC` to indicate global ordering of unknowns in groups. The number of groups is limited by constant `MaxNumGroups = 10` defined in file *fem2Dtri.fd*. If additional bits are omitted, they will be added automatically by the code. For example, for the $P_2 - P_1$ discretization of the Stokes (see `Tutorials/PackageFEM/mainTemplate.f`), we can set (this will be also the automatic ordering):

```
Do i = 1, 3
  templateR(i) = Vdof + Group1      ! u_x at points
  templateR(i + 3) = Rdof + Group2  ! u_x on edges
  templateR(i + 6) = Vdof + Group3  ! u_y at points
  templateR(i + 9) = Rdof + Group4  ! u_y on edges
  templateR(i + 12) = Vdof + Group5 ! pressure at points
Enddo
```

This breaking unknowns into groups will lead to a block structure of the global matrix. In this case, it will be a 5×5 block matrix. The first, third and fifth diagonal blocks will have the same size equal to the number of mesh points. The remaining diagonal blocks will have size equal to the number of mesh edges. The ordering of unknowns inside these blocks is described in the next subsection.

For the third-order finite element discretization of the Laplacian, we can set (the automatic ordering will create four groups and the global matrix will be assembled incorrectly):

```
Do i = 1, 3
  templateR(i) = Vdof + Group1      ! u at vertices
  templateR(i + 3) = Rdof + Group2  ! u on edges
  templateR(i + 6) = Rdof + Group2  ! u on edges
Enddo
templateR(10) = Edof + Group3       ! u inside element
```

This breaking unknowns into groups will lead to a block structure of the global matrix. In this case, it will be a 3×3 block matrix. The first diagonal block will have size equal to the number of mesh points. The second diagonal block will have size equal to twice the number of mesh edges. The last diagonal block will have size equal to the number of mesh triangles. The ordering of unknowns inside these blocks is described in the next subsection.

Here are a few examples where the extended elemental matrix may be useful. For the diffusion-reaction equation, we may sum elemental matrices corresponding to diffusion and reaction in one local matrix. For the diffusion equation written in a mixed form using Lagrange multipliers, we may use hybridization algorithm inside `FEM2Dext`. We may also incorporate boundary conditions in each elemental matrix.

4.2.3 Assembling utilities

The package provides a utility for assembling elemental matrices and right-hand sides. The assemble routine returns a sparse matrix in the formats required by many solvers, the compressed sparse row (CSR) and column (CSC) formats, as well as format used in AMG by K.Stuben, J.Ruge. Other formats will be supported in the nearest future or by request (see converters in file `algebra.f`). Here is the header of the assembling routine. We describe only the new parameters.

The assembling routine must be used for the extended elemental matrices described in Section 4.2.2. All but one parameters were described above. The new parameter `lbP` is optional labels of mesh points. They may be useful for assigning Dirichlet boundary conditions.

```
      Subroutine BilinearFormTemplate(
     &           nP, nF, nE, XYP, lbP, IPF, lbF, IPE, lbE,
     &           FEM2Dext, dDATA, iDATA, control,
     &           MaxF, MaxA, IA, JA, A, F, nRow, nCol,
     &           MaxWi, MaxWr, iW, rW)

C     nP - [input] the number of points (P)
C     nF - [input] the number of edges (F)
C     nE - [input] the number of elements (E)
C
C     XYP(2, nP) - [input] Real*8 Cartesian coordinates of mesh points
C     IPF(2, nF) - [input] connectivity list of boundary faces
C     IPE(3, nE) - [input/output] connectivity list of elements.
C
C     lbP(nP) - [input] point labels
C     lbF(nF) - [input] boundary labels
C     lbE(nE) - [input] element  labels
C
C     control(3) - integer array with control parameters (see below)
C
C     MaxF - [input] the maximal number of equations plus one
C     MaxA - [input] the maximal number of nonzero entries in A
C     IA,JA,A - [output] CSR/CSC sparsity structure of matrix A (see template.f)
C
C     nRow - [output] the number of rows in A
C     nCol - [output] the number of columns in A
C
C     iW(MaxWi) - integer working array of size MaxWi
C     rW(MaxWr) - real*8 working array of size MaxWr
```

The global ordering of unknowns (e.g. in the right-hand side vector `F`) follows the following rules:

- The unknowns are ordered by groups specified in arrays `templateR` and `templateC` of function *FEM2Dext*. If no ordering is provided, the default one is generated by the code. The user may check the returned values of these arrays.
- Inside each group, the unknowns are ordered using different rules (for historic reasons):
  - The point-based unknowns are ordered by the point id. The above rules imply that the total number of unknowns in such a group is equal to the total number of mesh points.
  - The edge-based unknowns are ordered first by the edge id and then by the local position on an edge. The local ordering of unknowns starts from unknown closest to the vertex with the smallest id. For example, the global ids of edge-based unknowns, labeled 1 and 4 in Fig. 4.2, will differ by the number of mesh edges and the first one will have a smaller global id.
  - The element based unknowns are ordered first by the local id and then by the element id. Note this differ from the ordering of edge-based unknowns.

The matrix `A` is assembled in one of the sparse row formats. This is controlled via `control(1)`. Available values for this parameter are located in file `assemble.fd`. A logical AND can be used to combine a few parameters from the following list

```
MATRIX_SYMMETRIC - symmetric matrix
MATRIX_GENERAL   - general matrix
FORMAT_AMG       - format used in AMG
FORMAT_CSR       - compressed sparse row format
FORMAT_CSC       - compressed sparse column format
```

Possible contradictions will be checked by the code.

The second parameter `control(2)` sets up the verbosity level. This is a number from 0 to 9. The third control parameter is not used in this release.

The discontinuous Galerkin finite element method requires more sophisticated data structures than the other finite elements. We developed a new assembling routine that extends capabilities of the previous routine. The list of input parameters mimics that of the previous routine.

```
 Subroutine AssembleFromTemplate(
&           nP, nF, nE, XYP, lbP, IPF, lbF, IPE, lbE,
&           FEM2Dext, DDATA, IDATA, control,
&           MaxF, MaxA, IA, JA, A, F, nRow, nCol,
&           MaxWi, MaxWr, iW, rW)
```

4.3 Discontinuous Galerkin

Our approach to the DG methods is to break generation of 'elemental' matrices into three parts. Discretization of jump conditions of mesh edges requires to consider a patch of at most four triangles, which leads to generation of local patch matrices. The size of a patch matrix is at most four times bigger than the size of an elemental matrix.

The library *libfem2D-3.1.a* implements jump terms appeared in various DG methods. The user has to assemble the elemental matrix for the central triangle (see Fig. 4.3) in the same way as it is done for routine `BilinearFormTemplate` and to impose boundary conditions either locally for the whole patch matrix or globally. The local implementation of boundary condition must rely on the following local order of vertices shown in Fig. 4.3. The assembling routine `AssembleFromTemplate()` will take care of global orientation of triangles.

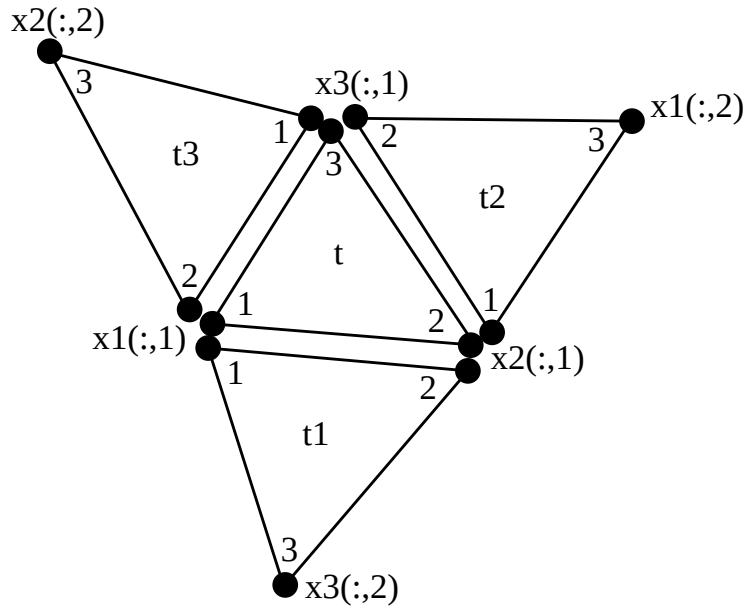


Figure 4.3: Local enumeration of mesh vertices in DG methods.

To stress that degrees of freedom are discontinuous, the neighboring triangles are shifted slightly apart from the central triangle in Fig. 4.3. The central triangle has at most three neighbors. The i-th neighbor shares the edge $x_i - x_{i+1}$ with the central triangle.

The list of parameters in the user routine `FEM2Dext` is not changed. To provide additional information about the patch of four elements, we pass two-dimensional arrays $XYi(2,2)$ instead of one-dimensional

arrays described above. The columns of these arrays are shown in Fig. 4.3. Similarly, we added the second dimension to arrays `lbPloc(3,2)` and `lbFloc(3,3)`. Entries `lbPloc(i,1)` and `lbPloc(i,2)` in the first array correspond to vertices `XYi(:,1)` and `XYi(:,2)`, respectively. Entries `lbFloc(i,2)` and `lbFloc(i,3)` correspond to edges 12 and 23 of the i-th neighboring triangle. Note that entries of these arrays corresponding to a missing triangle are not defined.

Let us consider a typical step in a user routine `FEM2Dext`. The initialization procedure

```
Call DG_init(XY1, XY2, XY3, iSYS, XYZ, t, IEE)
```

preprocesses the input data for other routines. It copies coordinates of mesh vertices `XYi(2,2)` into the three-dimensional array `XYZ(2,3,2)`, extracts numbers of neighboring triangles from array `iSYS(MAXiSYS)` and copies them into array `IEE(3)`. Finally, it extracts the global number `t` of the central triangle.

Next, we populate the template arrays `templateR(nRow)` and `templateC(nCol)`. For a classical DG method, the user may call

```
      Call DG_template(iE, IEE, FEMtypeA, FEMtypeB, nRow, templateR, ks)

c     FEMtypeA    - type of the Lagrange finite element for triangle t
c     FEMtypeB(3) - types of the Lagrange finite elements for triangles t_i
c     ks(3) - pointers, ks(i)+1 is the 1st dof in templateR() for the i-th
c             neighboring triangle t_i
```

Often, for a square system, `nCol = nRow` and the template for columns, `templateC()`, coincides with that for rows.

The user routine `FEM2Dext` is called by `AssembleFromTemplate` a few times for each triangle. To first two calls require only arrays `templateR` and `templateC`. This is indicated with `iSYS(1) = -1`. Since most of the input parameters may not be populated at these calls, the user is advised to add the following line to the code right after these arrays were populated:

```
If(iSYS(1).LT.0) Return
```

The first DG method is a symmetric interior penalty method. It requires to calculate two types of edge integrals. The first one involves only jumps of function values. For instance, for edge e shared by triangles `t` and `t1` we need to calculate

$$\frac{1}{h_e}\int_e D(x,y)\,[[u]]\,[[v]]\,\mathrm{d}x, \qquad [[u]] = u_t\mathbf{n}_t + u_{t1}\mathbf{n}_{t1},$$

where $\mathbf{n}_{t1}$ indicates the exterior normal for triangle `t`. Theory of DG methods does not specify a simple way to calculate the constant function $D(x,y)$. It should be just big enough. We provide a possibility to make it dependent on coordinates x and y:

```
Call DGjump_AJ(XYZ, iE, IEE, ks,
&              FEMtypeA, FEMtypeB,
&              D, dDATA, iDATA, iSYS, order,
&              LDA, A, nRow, nCol)
```

The list of input parameters in the following call has been explained above. We note that `nRow` and `nCol` are input parameters calculated by routine `DG_template()`.

The symmetric interior penalty method requires a consistency integrals involving jumps of derivatives:

$$-\int_e \{D(x,y)\nabla u\}\,[[v]]\,\mathrm{d}x - \int_e \{D(x,y)\nabla v\}\,[[u]]\,\mathrm{d}x, \qquad \{D(x,y)\nabla u\} = \frac{1}{2}(D\nabla u_t + D\nabla u_{t1}).$$

This integrals are calculated using the following call:

```
      Call DGjump_SIP(XYZ, iE, IEE, ks,
     &                FEMtypeA, FEMtypeB,
     &                D, dDATA, iDATA, iSYS, order,
     &                LDA, A, nRow, nCol)
```

Finally, the user has to sum up all matrices, add its own matrices for the bilinear forms defined on the central triangle `t`, create a right-hand side vector, and optionally impose locally boundary conditions.

The library *libfem2D-3.1.a* allows the user to vary polynomial order across mesh elements.

Non-symmetric interior penalty method requires to call

```
 Call DGjump_NIP(XYZ, iE, IEE, ks,
&                FEMtypeA, FEMtypeB,
&                D, dDATA, iDATA, iSYS, order,
&                LDA, A, nRow, nCol)
```

This is the only difference between using symmetric and non-symmetric interior penalty methods.

4.4 Error calculation

Package Ani2D-FEM uses the same core routines to calculate error of a finite element solution u_h. Let u denote an exact solution. We define the following elemental error:

$$\|u - u_h\|_*^p = \int_\Delta |D\,(Op_A(u_h) - u) \cdot (Op_A(u_h) - u)|^{p/2}\,\mathrm{d}x,$$

where D is a tensor, and Op_A is a linear operator as defined earlier. To calculate this error, we call the following function. Only new parameters are described here.

```
 Call fem2Derr(XY1, XY2, XY3, Lp,
&              operatorA, FEMtypeA, Uh, Fu, dDATAFU, iDATAFU,
&              label, D, dDATA, iDATA, iSYS, order, ERR)

C  Lp - norm for which the error is to be calculated:
C           Lp > 0  means the L^p norm
C           Lp = 0  means the maximum norm (L^infinity)
C
C  Uh - Real*8 vector of discrete solution over triangle. It must have
C       the same size as the matrix of bilinear form <OpA(u), OpA(u)>
C
C  Fu - Integer function for exact solution. The standard format:
C       Fu(x, y, label, dDATAFU, iDATAFU, iSYS, Diff)
C
C       The function returns type of the tensor Diff which is the
C       value of this function. See fem2Dtri.f for more details.
```

4.5 Examples

4.5.1 Elemental matrices

The program `Tutorials/PackageFEM/mainTriangle.f` demonstrates the use of the simplest routine `fem2Dtri` for generating several elemental matrices on a triangle:

$$\int_\Delta \varphi^{\mathrm{BDM1}} \cdot \psi^{\mathrm{BDM1}}\,dx, \qquad \int_\Delta \mathrm{div}(\varphi^{\mathrm{BDM1}})\,dx, \qquad \int_\Delta \mathrm{curl}(\varphi^{\mathrm{P1}})\mathrm{curl}(\psi^{\mathrm{P1}})\,dx, \qquad \int_\Delta \nabla(\varphi^{\mathrm{P3}}) \cdot \nabla(\psi^{\mathrm{P3}})\,dx,$$

where φ and ψ stand for either a vector or scalar functions.

4.5.2 Diffusion-reaction problem with inhomogeneous boundary conditions

The program `Tutorials/PackageFEM/mainBC.f` generates the finite element system for the boundary value problem with continuous piecewise linear finite elements P_1:

$$\begin{aligned} -\mathrm{div}(K \,\mathrm{grad}\, u) + u &= 1 \quad \text{in} \quad \Omega, \\ u &= x + y \quad \text{on} \quad \partial\Omega_D, \\ K\frac{\partial u}{\partial n} &= x - y \quad \text{on} \quad \partial\Omega_N, \\ K\frac{\partial u}{\partial n} + u &= x \quad \text{on} \quad \partial\Omega_R, \end{aligned}$$

where $\Omega = (0,1)^2$ is the unit square, $\partial\Omega_D$ is the union of bottom and right sides of Ω, $\partial\Omega_N$ is the top side of Ω, and $\partial\Omega_R$ is the left side of Ω. The diffusion coefficient K is the full constant tensor:

$$K = \begin{bmatrix} 1 & -1 \\ -1 & 10 \end{bmatrix}.$$

The program generates the finite element system using routine `BilinearFormTemplate`.

4.5.3 Stokes problem

The program `Tutorials/PackageFEM/mainTemplate.f` generates the finite element system of the Stokes problem with the stable $P_2 \times P_1$ pair of finite elements:

$$\begin{aligned} -\mathrm{div}\,\mathrm{grad}\,\mathbf{u} + \nabla p &= 0 \quad \text{in} \quad \Omega, \\ -\mathrm{div}\,\mathbf{u} &= 0 \quad \text{in} \quad \Omega, \\ \mathbf{u} &= \mathbf{u}_0 \quad \text{on} \quad \partial\Omega_1, \\ \mathbf{u} &= 0 \quad \text{on} \quad \partial\Omega_2, \\ \frac{\partial \mathbf{u}}{\partial n} - p &= 0 \quad \text{on} \quad \partial\Omega_3, \end{aligned}$$

where $\Omega = (0,1)^2$, $\partial\Omega_1 = \{(x,y) : \ x = 0, \ 0 < y < 1\}$, $\partial\Omega_3 = \{(x,y) : \ x = 1, \ 0 < y < 1\}$, $\partial\Omega_2 = \partial\Omega \setminus (\partial\Omega_1 \cup \partial\Omega_3)$, and $\mathbf{u}_0 = (4y(1-y), 0)^T$.

4.5.4 DG method for diffusion problem

The program `Tutotials/MutiPackage/DiscontinuousGalerkin/main.f` solves the diffusion problem

$$\begin{aligned} -\mathrm{div}(\mathrm{grad}\, u) &= -2 \quad \text{in} \quad \Omega, \\ u &= x^2 \quad \text{on} \quad \partial\Omega, \end{aligned}$$

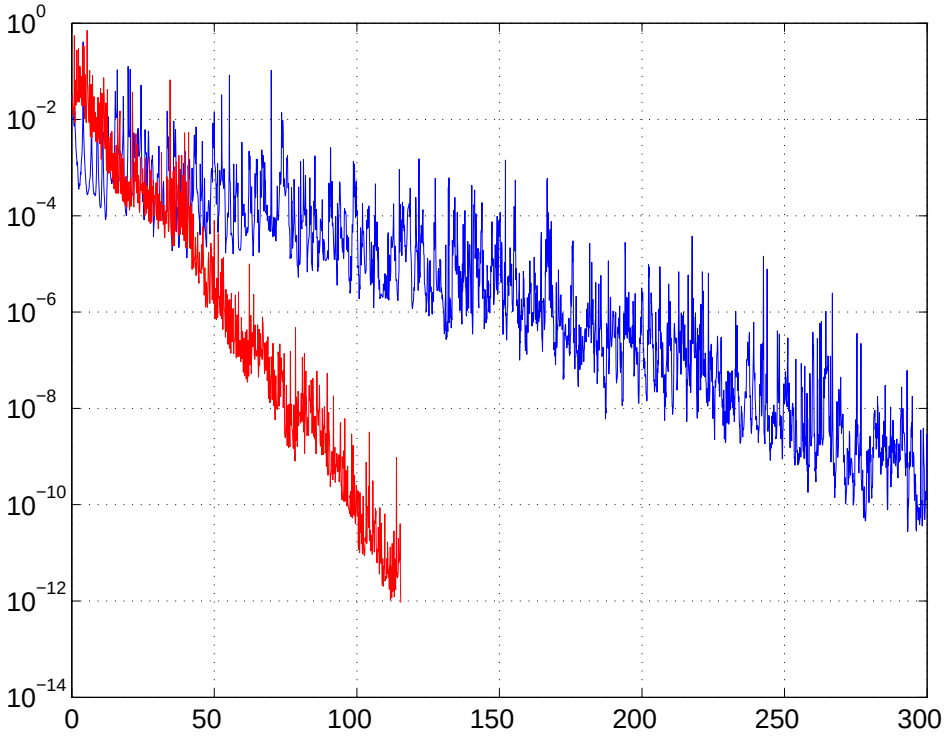
where Ω is a unit disk of radius 1. The exact solution of this problem is $u(x,y) = x^2$.

The problem is solved with a DG finite element method. We use a mixture of P_2 and P_3 finite elements. For triangles with centers belonging to the disk or radius 0.5, we use the quadratic finite elements. For the remaining triangles, we use the cubic finite elements. These elements represent the solution exactly; therefore, the error calculated at mesh vertices is zero.

Note that information about the finite element discretization is passed inside the user-written routines in file `forlibfem.f` via the integer array `iDATA`. Note also a long code for imposing Dirichlet boundary conditions on edges of the DG superelement (see Fig. 4.3). We work on a more user-friendly way to support boundary conditions (see file `aniFEM/bc.f`).

Chapter 3

SOLUTION PACKAGEs



Ani2D-LU "*Twinflower*"

LU Factorization Solver for Sparse Systems

User's Guide for liblu-5.1.a

5.1 A short description of the library

The C package Ani2D-LU is a set of double precision routines of UMFPACK-5.1, AMD, and UFconfig packages developed by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. It is designated for the direct solution of sparse linear systems. Minor modifications related to binding C and Fortran/BLAS were done to compile these packages on various platforms. The source code was not modified which allows us to consider Ani2D-LU is an independent part of the package Ani2D.

Example of using Ani2D-LU in a FORTRAN program is given in file `src/Tutorials/PackageLU/main.f`. For detailed documentation, see `src/aniLU/UMFPACK/UserGuide.pdf`.

Ani2D-ILU version 3.1 "*Bellflower*"

Flexible Iterative Solver Using Incomplete LU Factorization

User's Guide for libilu-3.1.a

6.1 Basic features of the library

The FORTRAN-77 package Ani2D-ILU is an independent part of the package Ani2D. Ani2D-ILU was developed by Yuri Vassilevski, Sergey Goreinov and Vadim Chugunov. It is designated for the iterative solution of sparse linear systems.

Library *libilu-3.1.a* may be easily incorporated in other packages.

The basic features of library *libilu-3.1.a* are listed below.

Iterative method : BiConjugate Gradient Stabilized (BiCGstab), Conjugate Gradient (CG), Generalized minimal residual restarted (GMRES(m))

Preconditioners : ILU0 and ILU2, the second order accurate ILU

Matrix storage format : Compressed Sparse Row-wise, CSR

Data format : double precision or integer arrays. Enumeration starts from 1.

Typical memory requests : for systems with N equations and NZ non-zero matrix elements, BiCGstab (resp., CG, GMRES(m)) needs 8 (resp., 4, $m+3$) work vectors of dimension N, right-hand side and solution vectors. ILU0 requires the same storage as the CSR matrix representation. ILU2 requires up to 2-5-fold memory for the CSR matrix representation.

6.2 Iterative solution

The default iterative solver is BiConjugate Gradient Stabilized method (BiCGstab). This is the Krylov subspace method applicable to non-singular non-symmetric matrices. Therefore, it requires two procedures: matrix-vector multiplication and preconditioner-vector evaluation. If the user is not confident that the matrix is symmetric positive definite, he or she is advised to choose the default method. The call of the method is

```
 Call slpbcgs(prevec, IPREVEC, iW,rW,
&             matvec, IMATVEC, ia,ja,a,
&             WORK, MW, NW,
&             N, RHS, SOL,
&             ITER, RESID,
&             INFO, NUNIT)
```

- `prevec` is the name of a preconditioner-vector multiplication routine and `IPREVEC` is an integer array with user's data which is passed to `prevec`. In the presented examples `IPREVEC` contains a single entry that equals to the system order. The format of `prevec` is:

```
      Subroutine prevec(IPREVEC, ICHANGE, X, Y, iW, rW)
c Input
      Integer IPREVEC(*), ICHANGE, iW(*)
      Real*8  X(*), rW(*)
c Output
      Real*8  Y(*)
```

  This routine solves the system $(LU)\,Y = X$ with L and U being the low and upper triangular factors, respectively, stored in arrays `iW` and `rW`. `ICHANGE` is the flag controlling modification of the preconditioner. It may be useful when the convergence stagnates. We provide two examples of `prevec` corresponding to two preconditioners, `prevec0` (see file ilu0.f) and `prevec2` (see file iluoo.f).

- `iW, rW` are the Integer and Real*8 arrays which store preconditioner's data.

- `matvec` is the name of a generalized matrix-vector multiplication routine and `IMATVEC` is an integer array of user's data which is passed to `matvec`. In the presented example `IMATVEC` contains a single entry that equals to the system order. The format of `matvec` is as follows:

```
      Subroutine matvec(IMATVEC, ALPHA, X, BETA, Y, ia, ja, a)
c Input
      Integer IMATVEC(*), ia(*), ja(*)
      Real*8  X(*), Y(*), a(*), ALPHA, BETA
c Input/Output
      Real*8  Y(*)
```

This routine calculates a matrix-vector product AX and adds the vector βY to it:

$$Y := \alpha AX + \beta Y.$$

For example, when $\alpha = 1$ and $\beta = 0$, `matvec` returns $Y = AX$. The example of this routine is in file `bcg.f`. It uses the compressed sparse row (CSR) representation of matrix A stored in arrays `ia`, `ja`, and `a`.

- `ia,ja,a` are two Integer and one Real*8 arrays containing a matrix in the CSR format.
- `WORK(MW,NW)` is Real*8 working two-dimensional array which stores at least 8 Krylov vectors.
- `MW*NW` the total length of `WORK` which must be not less than `8N`.
- `N` is the system order and length of vectors.
- `RHS` is the right-hand side vector (Real*8).
- `SOL` is the initial guess on input and the iterated solution on output (Real*8).
- `ITER` is the maximal number of iterations on input and the actual number of iterations on output.
- `RESID` is the convergence criterion on input and norm of the final residual on output.
- `INFO` is the performance information, 0 - converged, 1 - did not converge, etc.
- `NUNIT` is the channel number for output (0 - no output).

If the user is not satisfied with the convergence of the BiCGstab method, he or she can use the GMRES(m) method which requires $m + 3$ work vectors of length $N \leq$ `MW`:

```
Integer   IRESTART
Parameter (IRESTART = 20)

Integer   NW, MH, NH
Parameter (NW = IRESTART+3, MH = IRESTART + 1, NH = IRESTART + 6)
Real*8    H(MH,NH), WORK(MW*NW)
...

Call slgmres(prevec, IPREVEC, iW,rW,
&             matvec, IMATVEC, ia,ja,a,
&             WORK, MW, NW, H, MH, NH,
&             N, RHS, SOL,
&             ITER, RESID,
&             INFO, NUNIT)
```

In many applications, the larger parameter m, the faster convergence of GMRES(m). The price of using a large m is high memory requirements. We note that one iteration of BiCGstab costs approximately two iterations of GMRES(m). An example of calling the GMRES(m) solver is in file `src/Tutorials/PackageILU/main_gmres_ilu0.f`.

If the matrix is symmetric and positive definite, the user can save 4 work vectors and probably 10-30% of the CPU time by calling the Conjugate Gradient method (CG):

```
      Call slpcg(prevec, IPREVEC, iW,rW,
     &           matvec, IMATVEC, ia,ja,a,
     &           WORK, MW, NW,
     &           N, RHS, SOL,
     &           ITER, RESID,
     &           INFO, NUNIT)
```

The parameters of this routine are the same as for the above routines, except that `MW*NW` must be not less than $4N$.

6.3 ILU0 preconditioner

The ILU0 preconditioner is the simplest and the most popular incomplete LU preconditioner. It is characterized by very fast and economical factorization. The drawbacks of the method are its slow convergence and a danger to get a zero pivot. Nevertheless, for simple non-stiff problems, it works well. The application of the preconditioner has two stages: initialization and evaluation. The initialization is done with the routine `ilu0`. The evaluation must be performed at each step of the iterative method. It is provided by the routine `prevec0`. The user should provide the name `prevec0` as one of the input parameters:

```
      external prevec0
      ...

      Call slpbcgs(prevec0, IPREVEC, iW,rW,
     &             matvec, IMATVEC, ia,ja,a,
     &             WORK, MW, NW,
     &             N, RHS, SOL,
     &             ITER, RESID,
     &             INFO, NUNIT)
```

The initialization routine has the following parameters

```
      Call ilu0(n, a, ja, ia, alu, jlu, ju, iw, ierr)
```

where

- `n` is the matrix order
- `ja,ia,a` are two Integer and one Real*8 arrays containing a matrix in the CSR format
- `alu,jlu,ju` are one Real*8 and two Integer arrays containing the L and U factors
- `ierr` is the integer error code (0 - successful factorization, k - zero pivot at step k)
- `iw` is the integer working array of length n.

Let us present the basic blocks of a program solving a system of linear equations with a matrix `a, ia, ja` and a right-hand side vector `f` using the BiCGstab method with the ILU0 preconditioner. First we define all necessary arrays and variables:

```
C Matrix arrays in the CSR format
      Integer ia(maxn+1), ja(maxnz)
      Real*8   a(maxnz), f(maxn), u(maxn)

C Work arrays for ILU factors and 8 BCG vectors
      Integer   MaxWr,MaxWi
      Parameter(MaxWr=maxnz+8*maxn, MaxWi=maxnz+2*maxn+1)
      Real*8  rW(MaxWr)
      Integer iW(MaxWi)
```

```
C BiCGStab data
      External  matvec, prevec0
      Integer   ITER, INFO, NUNIT
      Real*8    RESID

C ILU0 data
      Integer   ierr, ipaLU, ipjLU, ipjU, ipiw

C Local variables
      Integer   ipBCG
```

Second, we initialize the preconditioner by computing the L and U factors and saving them in array `rW(1:nz)` and `iW(1:nz+n+1)`:

```
      ipaLU = 1
      ipBCG = ipaLU + nz
      ipjU  = 1
      ipjLU = ipjU + n + 1
      ipiw  = ipjLU + nz   ! work array of length n

      Call ilu0(n, a,ja,ia, rW(ipaLU), iW(ipjLU),iW(ipjU), iW(ipiw), ierr)

      If (ierr.ne.0) Then
         write(*,*)'initialization of  ilu0 failed, zero pivot=',ierr
         stop
      End if
```

Third, once the preconditioner is initialized, we call the iterative solver:

```
      ITER  = 1000      ! max number of iterations
      RESID = 1d-8      ! threshold for \|RESID\|
      INFO  = 0         ! no troubles on input
      NUNIT = 6         ! output channel

      Call slpbcgs(prevec0, n, iW,rW,
     &             matvec, n, ia,ja,a,
     &             rW(ipBCG), n, 8,
     &             n, f, u,
     &             ITER, RESID,
     &             INFO, NUNIT)

      If (INFO.ne.0) Stop 'BiCGStab failed'
```

Examples of programs implementing these three steps are in `src/Tutorials/PackageILU/main_bcg_ilu0.f` and `src/Tutorials/PackageILU/main_gmres_ilu0.f`.

6.4 ILU2 preconditioner

The ILU2 preconditioner is an ILU factorization with two thresholds proposed by I.Kaporin in 1998. For symmetric positive definite stiff systems, it is shown to be robust and to give better convergence rate compared to other factorizations. It can be applied to non-symmetric matrices as well. The factorization of the input matrix A satisfies the formula

$$A = LU + TU + LR - S$$

where L, U are the first order factors, T, R are the second order factors (kept and used in calculation, neglected after calculation), and S is the residual matrix (neglected during the calculation). The method

seems to be a flexible and powerful tool for constructing efficient preconditioners for stiff matrices. The application of this preconditioner has two stages: initialization and evaluation. The initialization routine is *iluoo*. The evaluation must be performed at each step of an iterative method. It is provided by the routine *prevec2*. The user should provide the name *prevec2* as the first input parameter:

```
      external prevec2
      ...

      Call slpbcgs(prevec2, IPREVEC, iW,rW,
     &               matvec, IMATVEC, ia,ja,a,
     &               WORK, MW, NW,
     &               N, RHS, SOL,
     &               ITER, RESID,
     &               INFO, NUNIT)
```

The initialization routine has the following parameters

```
      Call iluoo(n, ia, ja, a, tau1, tau2, verb,
     &            work, iwork, lendwork, leniwork,
     &            partlur, partlurout,
     &            lendworkout, leniworkout, ierr)
```

where

- `n` is the order of the square matrix A.
- `ia,ja,a` are two Integer and one Real*8 arrays containing the matrix in the CSR format.
- `tau1` is the absolute threshold for entries of L and U. Elements of L and U greater than τ_1 will enter L and U. The recommended values lie in the interval $[0.01; 0.1]$.
- `tau2` is the absolute threshold for entries of T and R. Elements not included in L and U but greater than τ_2 will enter T and R. The recommended values lie in the interval $[\tau_1^2; 10\tau_1^2]$.
- `verb` sets up the verbosity level: 0 means no output, positive means verbose output.
- `work(iwork), lendwork(leniwork)` are working Real*8 and Integer arrays.
- `partlur` is the user defined partition of the available memory `work(iwork)`. The L, U factors occupy first `(1-partlur)* lendwork` positions, while T and R occupy next `partlur*lendwork` positions.
- `partlurout` is the optimal partition computed during the factorization. It may be useful for a next factorization.
- `lendworkout,leniworkout` are the minimal (round-off errors may cause a tiny underestimate) memory demands provided that the optimal partition LU/TR of the available memory is used. It may be useful for a next factorization.
- `ierr` is the integer error code (0 - successful factorization).

Let us present the basic blocks of a program solving a system with a matrix `a, ia, ja` and a right-hand side vector `f` by the BiCGstab method with the ILU2 preconditioner. First, we define all necessary arrays and variables:

```
C Matrix arrays in the CSR format
      Integer ia(maxn+1), ja(maxnz)
      Real*8   a(maxnz), f(maxn), u(maxn)

C Work arrays
      Integer   MaxWr,MaxWi
      Parameter(MaxWr=5*maxnz, MaxWi=6*maxnz)
      Real*8  rW(MaxWr)
```

```
      Integer iW(MaxWi)

C BiCGStab data
      External  matvec, prevec2
      Integer   ITER, INFO, NUNIT
      Real*8    RESID

C ILU data
      Real*8   tau1,tau2,partlur,partlurout
      Integer  verb, ierr, UsedWr, UsedWi

C Local variables
      Integer   ipBCG, ipIFREE
```

Second, once the matrix is stored in the CSR format, we initialize the preconditioner by computing the L and U factors and saving them in `rW, iW`:

```
      verb    = 0      ! verbose no
      tau1    = 1d-2
      tau2    = 1d-3
      partlur = 0.5
      ierr    = 0

      Call iluoo(n, ia, ja, a, tau1, tau2, verb,
     &           rW, iW, MaxWr, MaxWi, partlur, partlurout,
     &           UsedWr, UsedWi, ierr)

      If (ierr.ne.0) Then
         Write(*,*) 'Initialization of iluoo failed, ierr=', ierr
         Stop
      End if

      if (UsedWr+8*n.gt.MaxWr) then
          write(*,*) 'Increase MaxWr to ',UsedWr+8*n
          stop
      end if
      ipBCG  = UsedWr + 1
```

Third, once the preconditioner is initialized, we call the iterative solver:

```
      ITER  = 1000        ! max number of iterations
      RESID = 1d-8        ! threshold for \|RESID\|
      INFO  = 0           ! no troubles on input
      NUNIT = 6           ! output channel

      Call slpbcgs(prevec2, n, iW,rW,
     &             matvec, n, ia,ja,a,
     &             rW(ipBCG), n, 8,
     &             n, f, u,
     &             ITER, RESID,
     &             INFO, NUNIT)

      If (INFO.ne.0) Stop 'BiCGStab failed'
```

An example is given in file `src/Tutorials/PackageILU/main_bcg_ilu2.f`.

Ani2D-INB Version 3.1 "*Starflower*"

Flexible Iterative Solver Using Inexact Newton-Krylov Backtracking

User's Guide for libinb-3.1.a

7.1 Basic features of the library

The FORTRAN-77 package Ani2D-INB is an independent part of the package Ani2D. Ani2D-INB was developed by Alexey Chernyshenko under the supervision of Yuri Vassilevski. It is designated for the iterative solution of nonlinear systems.

The library *libinb-3.1.a* may be easily incorporated in other packages.

The package interfaces to the ILU preconditioners provided by the Ani2D-ILU package or any other preconditioner such as LU sparse factorization solver. The package Ani2D-INB is a deeply processed and essentially simplified version of the NITSOL package by Homer F. Walker.

The basic features of library *libinb-3.1.a* are listed below.

Iterative method : Inexact Newton-Krylov Backtracking (INB), with BiConjugate Gradient Stabilized (BiCGStab) iteration as the interior Krylov subspace solver

Preconditioners : Common interface with ILU0 and ILU2, the second order accurate ILU (provided by the Ani2D-ILU package).

Problem setting : A user defined routine computing a nonlinear residual.

Data format : double precision or integer arrays. Enumeration starts from 1.

Typical memory requests : for systems with N equations Ani2D-INB needs 11 work vectors of dimension N, one solution vector and a room for preconditioner data. If the preconditioner is built by the Ani2D-ILU package, ILU0 requires the same storage as the CSR representation of the Jacobian, ILU2 requires 2-5-fold storage.

7.2 Iterative solution

The exterior iterative solver is the Inexact Newton-Krylov Backtracking (INB) method with the interior linear BiConjugate Gradient Stabilized method (BiCGStab). This is a Newton type method applicable to non-singular nonlinear systems. It requires two procedures: evaluation of the nonlinear residual function and optional preconditioner-vector evaluation. A preconditioner should approximate the inverse of the Jacobian matrix. The Jacobian-free finite difference method is used to evaluate of Jacobian-vector product. The call of the method is

```
      external prevec, funvec
      ....

      call slInexactNewton(prevec, IPREVEC, iWprevec, rWprevec,
     &                     funvec, rpar, ipar,
     &                     N, SOL,
     &                     RESID, STPTOL,
     &                     rWORK, LenrWORK,
     &                     INFO)
```

where

- `prevec` is the name of a preconditioner-vector multiplication routine. `IPREVEC, iWprevec, rWprevec` are two Integer and one Real*8 arrays of user's data which are passed to `prevec`. The arrays `iWprevec, rWprevec` are recommended to keep the preconditioner bulk data (triangular factors, for instance). The array `IPREVEC` may contain control parameters or basic user's data such as the system order and useful pointers. In the presented example, `IPREVEC` contains a single entry equal to the system order. The format of `prevec` coincides with that from the package Ani2D-ILU:

```
      Subroutine prevec(IPREVEC, ICHANGE, X, Y, iW, rW)
c Input
      Integer IPREVEC(*), ICHANGE, iW(*)
```

```
      Real*8  X(*), rW(*)
c Output
      Real*8  Y(*)
```

Here X is the input vector and Y is the output vector. `ICHANGE` is the flag controlling the change of the preconditioner. It is useful when convergence stagnation occurs. `iW, rW` are Integer and Real*8 arrays, respectively, which store the preconditioner data. The package Ani2D-ILU provides two examples of routine `prevec` corresponding to two ILU preconditioners, `prevec0` (see file ilu0.f) and `prevec2` (see file iluoo.f). The details may be found in the user guide for Ani2D-ILU.

- `funvec` is the name of the user routine computing the nonlinear residual $F(X)$ and `ipar, rpar` are Integer and Real*8 arrays of user's data which are passed to `funvec` and used there. The format of `funvec` is as follows:

```
      Subroutine funvec(n, xcur, fcur, rpar, ipar, itrmf)
c INPUT:
      Integer  n       ! dimension of vectors
      Real*8   xcur(*) ! current vector
      Real*8   rpar(*) ! double precision user-supplied parameters
      Integer  ipar(*) ! integer user-supplied parameters
c OUTPUT:
      Integer  fcur(*) ! nonlinear residual vector (zero for the solution)
      Integer  itrmf   ! flag for successful termination of the routine
```

- `N` is order of system and length of vectors.
- `SOL` is the initial guess and the iterated solution (Real*8).
- `RESID` is the convergence criterion for the nonlinear residual on input, the actual norm of the nonlinear residual on output.
- `STPTOL` is the stopping tolerance on the Newton's step length.
- `rWORK(LenrWORK)` is Real*8 working array which stores at least 11 vectors of size N.
- `LenrWORK` is the total length of `rWORK` which must be not less than `11 N`.
- `INFO` is the array of control parameters. On input: `INFO(1)` sets initial value for successful termination flag, `INFO(2)` sets the maximal number of linear iterations per Newton step, `INFO(3)` sets the maximal number of nonlinear iterations, `INFO(4)` sets the maximal number of backtracks, `INFO(5)` sets the printing level (0 none, 1 nonlinear residuals, 2 linear residuals). On output: `INFO(1)` is the value of the termination flag (successful termination corresponds to 0), `INFO(2)` is the number of performed linear iterations, `INFO(3)` is the number of performed nonlinear iterations, `INFO(4)` is the number of actual backtracks, `INFO(5)` is the number of performed function evaluations.

Examples of calling programs are in files `src/Tutorials/PackageINB/main_simple.f`, `src/Tutorials/PackageINB/main_bratu.f`, `src/Tutorials/MultiPackage/StokesNavier/main.f`.

Chapter 4

SERVICE PACKAGEs

Ani2D-LMR version 3.1 "*Cornflower*"

Local Metric Recovery

User's Guide for liblmr2D-3.1.a

8.1 Introduction

The FORTRAN-77 package Ani2D-LMR is developed by Konstantin Lipnikov and Yuri Vassilevski. It is designated for generating continuous tensor metrics. The tensor components are piecewise linear functions defined on nodes of a given triangular mesh. The generated metric may be used further in Metric Based Adaptation package Ani2D-MBA as a control of mesh features.

The input data for metric generation is either a discrete solution defined at mesh nodes, or cell-based error estimates or edge-based error estimates.

The library *liblmr2D-3.1.a* can be easily incorporated in other packages.

This document describes the structure of the package, input data, and user-supplied routines. It has a few examples illustrating details of the package.

8.2 Description of Ani2D-LMR

8.2.1 General structure of package

The package Ani2D-LMR consists of a few FORTRAN files. The routines in these files implement one of the following basic tasks:

1. Recovery of a nodal metric from a continuous function or a discrete nodal function;
2. Recovery of a nodal metric from an edge-based error estimator;
3. Recovery of a nodal metric from a cell-based error estimator;
4. Modification of a metric for error minimization in the L^p norm.

These tasks will be discussed in subsequent sections.

In addition to the library *liblmr2D-3.1.a*, package Ani2D-LMR contains a tutorial directory described in the last section.

8.2.2 Local metric recovery from discrete function

A nodal tensor metric may be recovered from the discrete function defined at nodes of the mesh. The metric is the spectral module of the discrete Hessian of this mesh function. A mesh that is quasi-uniform in this metric minimizes the maximum norm of the P_1 interpolation error of an underlying continuous function. Two methods for the Hessian recovery are implemented in files `Nodal2MetricVAR.f` and `Nodal2MetricZZ.f`.

```
      Subroutine Nodal2MetricVAR(U,
     &                           vrt, nv, tri, nt, nnd, nb, Metric,
     &                           MaxWr, rW, MaxWi, iW)

      Subroutine Nodal2MetricZZ(U,
     &                          vrt, nv, tri, nt, Metric,
     &                          MaxWr, rW, MaxWi, iW)

C  Input: U(nv) - Real*8 array containing function values at mesh vertices
C
C         nv - the number of vertices
C         nb - the number of boundary edges
C         nt - the number of triangles
C
C         vrt(2,nv) - coordinates of these vertices
C         tri(3,nt) - connectivity table for triangles
C         bnd(2,nb) - connectivity table for boundary edges
C
```

```
C  Output: Metric(3, nv) - tensor metric at mesh vertices
C
C  Work arrays: rW(MaxWr) - Real*8 working array
C               iW(MAxWi) - Integer working array
```

For the first method, the input mesh has to satisfy the following condition. Every boundary node can be connected to an interior node with at most *two* mesh edges.

8.2.3 Local metric recovery from edge-based error estimator

Nodal tensor metric may be recovered from edge-based error estimates η_{e_k}. The metric may be anisotropic in this case. Two methods of metric recovery are implemented. The first method based on the Least Squares solution of the local system

$$(M(a_i)e_k, e_k) = \eta_{e_k}.$$

Here $M(a_i)$ is the tensor metric to be recovered at a mesh node a_i, e_k are mesh edges incident to a_i, and η_{e_k} are error estimates prescribed to these edges.

```
      Subroutine EdgeEst2MetricLS(nv, nt, vrt, tri,
     &                            Error, Metric,
     &                            MaxWr, MaxWi, rW, iW)
c Input:
c      Integer nv, nt       ! numbers of mesh nodes and triangles
c      Real*8  vrt(2,nv)    ! coordinates of mesh nodes
c      Integer tri(3,nE)    ! connectivity table of triangles
c      Real*8  Error(3,nt)  ! error estimates prescribed to element edges
c
c Output:
c      Real*8  Metric(3,nv) ! node-based tensor metric
c
c Working arrays:
c      Integer iW(MaxWi)
c      Real*8  rW(MaxWr)
```

The second method is called the method of shifts. First, it recovers a cell-based (piecewise constant) tensor metric and then for each mesh node a_i picks a metric with the maximum determinant among all metrics in elements sharing the node a_i:

```
      Subroutine EdgeEst2MetricMAX(Error, nv, nt, vrt, tri,
     &                             Metric, MaxWr, rW)
```

The above routines recover a tensor metric that can be used to minimize the maximum norm of the P_1 interpolation error. The following routine can be used to minimize the maximum norm of the gradient of the P_1 interpolation error.

```
      Subroutine EdgeEst2GradMetricMAX(Error, nv, nt, vrt, tri,
     &                                 Metric,
     &                                 MaxWr, rW)
```

These routines may be used for any edge-based errors, including interpolation errors and a posteriori error estimates, see the last section.

If an analytical function is available, the interpolation error can be calculated and the tensor metric can be built using the following two routines.

```
      Subroutine Func2MetricMAX(Func,
     &                          nv, nt, vrt, tri,
     &                          Metric, MaxWr, rW)

      Subroutine Func2GradMetricMAX(Func, nv, nt, vrt, tri,
     &                              Metric,
     &                              MaxWr, rW)
c  Input:
c      Func - Real*8 Function f(xy), where xy(2) are point coordinates
```

Both routines use the method of shifts to build the metric.

8.2.4 Local metric recovery from cell-based error estimator

Nodal tensor metric may be recovered from cell-based error estimates η_{Δ_k}:

$$M(\Delta_k) = \eta_{\Delta_k}.$$

The metric will be isotropic (scalar tensor) in this case. The nodal metric is generated by applying the ZZ recovery algorithm to a scalar cell-based metric.

```
      Subroutine CellEst2MetricZZ(nv, nt, vrt, tri,
     &                            Error, Metric,
     &                            MaxWr, MaxWi, rW, iW)
```

This method is recommended for problems with isotropic solutions.

8.2.5 Metric modification for error minimization in L^p

The above routines build tensor metrics to minimize of the maximum (L^∞) norm of error. If the user wants to minimize the L^p norm, he or she should modify the metric using the following routine:

```
      Subroutine Lp_norm(nP, Lp, Metric)
c Input:
c     Real*8  Lp - norm for which the metric is to be adjusted:
c             Lp > 0  means  L_p     norm
c             Lp = 0  means  maximum norm (L_infinity)
c Output:
c     Real*8  Metric(3, nP)
```

8.3 Examples

Examples of usage of the package Ani2D-LMR are located in `src/Tutorials/PackageLMR`.

The program `mainNodal2Metric.f` demonstrates the local metric recovery from discrete function defined at mesh nodes. The metric is recovered by evaluating the discrete Hessian of this mesh function. The metric is built to minimize the L^p norm of interpolation error.

The program `mainFunc2GradMetric.f` demonstrates building the optimal metric for minimizing L^p norm of the gradient of the P_1 interpolation error. The metric is recovered using the analytic representation of interpolated function, since it requires nodal and mid-edge values of this function.

The program `mainEst2Metric.f` builds a metric from either errors defined at centers of mesh elements or mesh edges. This program calculates the maximum norm of the interpolation error on cells or edges for the user-defined function `Func(xy)`.

The errors may be replaced with a posteriori error estimates. Example of the adaptive solution of a BVP using hierarchical estimates is given in `src/Tutorials/MultiPackage/PosterioriEstimates/main.f`.

Ani2D-PRJ version 3.1 "*Feather Flower*"

Finite Element L^2 Projection

User's Guide for libprj2D-3.1.a

9.1 Basic features of the library

The FORTRAN-77 package Ani2D-PRJ is a part of the package Ani2D. It is designated for remapping data between two unstructured meshes using the conventional finite element L^2 projection. Intersection of two meshes (a metamesh) is constructed during assembling of the right-hand side, the most crucial part of the projection algorithm.

The package is organized as a library `libprj2D-3.1`. An example of using the library is `Tutorials/PackagePRJ/main.f`.

9.2 Usage of the library `libprj2D-3.1`

Given a finite element solution $u_h^{(2)} \in V_{h,2}$ on mesh $\Omega_h^{(2)}$, this library finds its finite element projection $u_h^{(1)}$ on to mesh $\Omega_h^{(1)} \in V_{h,1}$. The finite element spaces $V_{h,1}$ and $V_{h,2}$ may be different. We assume that the meshes occupy the same domain; however, the implemented algorithm remains stable even when the domains are different.

Mathematical formulation of the problem is as follows: Find $u_h^{(1)}$ such that

$$\int_\Omega u_h^{(1)} v_h^{(1)} \,\mathrm{d}x = \int_\Omega u_h^{(2)} v_h^{(1)} \,\mathrm{d}x, \qquad v_h^{(1)} \in V_{h,1}.$$

The algorithm consists of calculating a metamesh, the right-hand side vector, the mass matrix, and solution of a linear system. This library performs the first two steps. The last two steps are performed using the packages Ani2D-FEM and Ani2D-ILU or Ani2D-LU.

The metamesh is created by calling the following routine:

```
      Call MetaMesh(nv, vrt, nt, tri, nv2, vrt2, nt2, tri2,
     &              nv12, nvMetaMax, vrt12,
     &              nt12, ntMetaMax, tri12, parents,
     &              MaxWi, MaxWr, iW, rW, iERR)
C
C    nv1, vrt1(2,nv1), nt1, tri1(3,nt1) - the first mesh
C    nv2, vrt2(2,nv2), nt2, tri2(3,nt2) - the second mesh
C    nv12, vrt12(2,nv12), nt12, tri12(3,nt12) - intersection of two meshes
C    parents(2,nt12) - two parents of new triangles
C
C    rW(MaxWr) - Real*8 working memory of size nt + 2*nv2
C    iW(MaxWi) - Integer working memory of size 8*nt2 + 4*nv2
```

The right-hand side vector is calculated from elemental contributions of triangles in the metamesh using the library `libfem2D-3.1.a` of the package Ani2D-FEM. The assembling routine below allows the user to perform finite element projection not only in L^2-norm and also in energy norms. We describe only new parameters:

```
      Call assemble_rhs(nv1, vrt1, nt1, tri1, nv2, vrt2, nt2, tri2,
     &                  nv12, vrt12, nt12, tri12, parents,
     &                  operatorA, FEMtypeA, operatorB, FEMtypeB,
     &                  RHS, U2, MaxWi, iW)
C
C    operatorA - differential operator in front of u_2, e.g. IDEN or GRAD
C                as described in the package AniFEM
C    FEMtypeA  - finite element space V_h1, e.g., FEM_P1
C
C    operatorB - differential operator in front of u_1
C    FEMtypeB  - finite element space V_h2
C
```

```
C    U2(*)  - a given finite element solution on the second mesh
C    RHS(*) - right-hand side on the first mesh
C
C    iW(MaxWi) - integer working memory of size
C                3*(nt1+nt2) + max(nv1,nv2) + 3*max(nt1,nt2)
```

Let `M11` be the mass matrix in space $V_{h,1}$. Then, a finite element vector `U1` corresponding to $u_h^{(1)}$ is calculated by solving the problem `M11 U1 = RHS`.

Ani2D-VIEW version 3.1 "*Coneflower*"

Visualization Toolkit

User's Guide for libview2D-3.1.a

Ani2D-VIEW is a simple visualizing library producing PostScript-files of a mesh and isolines of a discrete solution.

Self-instructive examples of using Ani2D-VIEW are given in `src/Tutorials/PackageVIEW/main.f`, `src/Tutorials/PackageVIEW/main_matrix.f`.

Ani2D-C2F version 3.1 "*Fleeceflower*"

C-wrapper for FORTRAN Packages

User's Guide for libc2f2D-3.1.a

11.1 Introduction

The C package Ani2D-C2F is a C-interface to mesh generation routines from package Ani2D-MBA. This interface was designed to reduce the number of calling parameters and automatize the memory management. The interface to the package Ani2D-MBA is performed via structure `ani2D` defined in header file `ani2D.h` and library *anic2f2D-3.1.a*. This document describes the routines from this library.

In the future releases Ani2D-C2F will be extended by C-wrappers to Ani2D-RCB, Ani2D-FEM, and Ani2D-ILU.

11.2 Main routines

Any application program using libraries *animba2D-3.1.a* and *anic2f2D-3.1.a* must create an instance of the structure `ani2D` and then call the initialization routine:

```
ani2D ani2Real, *ani;
ani = &aniReal;
int ani2D_INIT( ani2D* ani, int nEStar, double mem_factor )
```

Here `nEStar` is the desirable number of elements and `mem_factor` is a factor controlling the size of working memory, `mem_factor` $\geq$ `1`. In general, it is impossible to estimate *a priori* the size of the required working memory, since it depends on alignment of initial mesh and the provided metric. In the future, automatic memory resizing will be implemented.

The next step is to populate the structure `ani2D` either by providing an input file with extension `.ani` or by initialization of all mesh objects one by one using routines described in the next section. The following routines operate with mesh files:

```
int ani2D_load_mesh( ani2D* ani, const char* file_name )
int ani2D_save_mesh( ani2D* ani, const char* file_name )
```

The first routine loads a mesh from file `file_name`. The file must have extension `.ani`. The second routine saves the mesh in the file `file_name`. To verify the mesh correctness, the following visualization routine may be called:

```
int ani2D_draw_mesh( ani2D* ani, const char* file_ps )
```

This routine creates a PostScript file `file_ps` with a mesh associated with the structure `ani`. This routine is the interface to a FORTRAN routine from library *libview2D.a*.

Now it is time to call one of the two mesh generation routines:

```
int ani2D_analytic( ani2D* ani, void* metric_function, void* crvfunc )
int ani2D_nodal(    ani2D* ani, double* metric_table,  void* crvfunc )
```

These are the major routines that build an adaptive mesh. The first routine requires a metric function that returns a tensor metric at a given space point. Here is an example of such a routine:

```
int metric_user( double* x, double* y, double* M )
{
   M[0] = 1;  /* element M11 */
   M[1] = 0;  /* element M21 */
   M[2] = 2;  /* element M22 */
   M[3] = 0;  /* element M12 */
   return 0;
}
```

The second routine requires a metric table that has three rows and and `nv` columns, where `nv` is the number of mesh vertexes. Each column defines a symmetric positive definite matrix at a mesh vertex. The following example shows how to create a simple metric (note that the elements in the table are ordered by columns):

```
for( n=0,i=0; i<nv; i++ ) {
   ani2D_get_point( ani, i, xy );
   metric_table[n++] = 1 + xy[1] * xy[1];   /* element M11 = 1 + y^2  */
   metric_table[n++] = 2 + xy[0] * xy[0];   /* element M22 = 2 + x^2  */
   metric_table[n++] = 0;                   /* element M12 = M_21 = 0 */
}
```

11.3 Supporting routines

The set the number of points, boundary edges edges and elements, the following pointers must be used: `ani->nP`, `ani->nF`, and `ani->nE`, respectively. The verify the number of mesh objects, the following calls can be performed:

```
int ani2D_number_points(   ani2D* ani )
int ani2D_number_edges(    ani2D* ani )
int ani2D_number_elements( ani2D* ani )
```

To limit the number of mesh object that can be generated by package Ani2D-MBA the following calls can be used:

```
void ani2D_set_max_points(   ani2D* ani, int nvmax )
void ani2D_set_max_edges(    ani2D* ani, int nbmax )
void ani2D_set_max_elements( ani2D* ani, int ntmax )
```

These routines limits the number of points, boundary edges and elements in the adapted mesh. Note that the final number of elements will be close to `nEStar`; however, the temporary number of elements may exceed this targeted number many times whenever the initial mesh and the provided metric are not in agreement.

The following operations with a mesh vertex are possible:

```
void ani2D_get_point( ani2D* ani, int i, double* xy )
void ani2D_set_point( ani2D* ani, int i, double* xy )
void ani2D_fix_point( ani2D* ani, int i )
```

The first routine returns Cartesian coordinates `xy` of the i-th mesh vertex. The second routines sets new Cartesian coordinates for the i-th mesh vertex. The third routine adds the i-th mesh vertex to the list of fixed vertexes.

The following operations with a mesh edge are possible:

```
void ani2D_get_edge( ani2D* ani, int i, int* edge, int* icrv, int* label )
void ani2D_set_edge( ani2D* ani, int i, int* edge, int  icrv, int  label )
void ani2D_fix_edge( ani2D* ani, int i )
void ani2D_get_crv(  ani2D* ani, int i, double* par, int* iFnc )
```

The first routine returns end vertexes `edge[2]` of the i-th mesh edge as well as the edge label `label` and the curvature identificator `icrv`. The later is zero for a straight edge. The second routine sets the same information for the i-th mesh edge. The third routine adds the i-th mesh edge to the list of fixed edges. The fourth routine returns the parametrization of the end vertexes `par[2]` and the parametrization function number `iFnc`.

The following operations with a mesh element are possible:

```
void ani2D_get_element( ani2D* ani, int i, int* tri, int* label )
void ani2D_set_element( ani2D* ani, int i, int* tri, int  label )
void ani2D_fix_element( ani2D* ani, int i )
```

The first routine returns the three vertexes `tri[3]` of the i-th mesh element as well the element label `label`. The second routine sets the same information for the i-th mesh element. The third routine adds the i-th element to the list of fixed elements.

To control the mesh generation, the following parameters can be set:

```
void ani2D_get_quality( ani2D* ani, double* Q )
void ani2D_set_quality( ani2D* ani, double  Q )

void ani2D_get_status( ani2D* ani, int* status )
void ani2D_set_status( ani2D* ani, int  status )

void ani2D_get_max_iters( ani2D* ani, int* max_iters )
void ani2D_set_max_iters( ani2D* ani, int  max_iters )

void ani2D_get_max_basket( ani2D* ani, int* max_basket )
void ani2D_set_max_basket( ani2D* ani, int  max_basket )
```

The first pair of routines returns (resp., sets) the desired (resp., final) mesh quality. The second pair of routines returns (resp., sets) internal variable `status` that is described in package Ani2D-MBA. The third pair of routines returns (resp., sets) the maximal allowed number of local mesh modifications. The fourth pair of routines returns (resp., sets) the maximal allowed number of temporary skipped bad elements. Note that without a basket of such elements, the package will try to improve quality of the same element over and over.

11.4 Examples

Examples of using Ani2D-C2F in C programs are given in files `src/Tutorials/PackageC2F/main_nodal.c`, `src/Tutorials/PackageC2F/main_analytic.c`.

Chapter 5

TUTORIALs

12.1 Lid-driven cavity problem

In this section we discuss the application of the package Ani2D to a nonlinear BVP. The source code of this example is located in `src/Tutorials/MultiPackage/StokesNavier`.

The lid-driven cavity problem is used as a validation test for new codes or new solution methods. The problem geometry is the unit square Ω. The boundary conditions are no-flow on three sides of the square, Γ_2, and the prescribed constant tangent velocity on the top side, Γ_1.

The moving lid (top side) creates a strong vortex inside the domain and a cascade of secondary vortexes. The larger the Reynolds number is, the more difficult numerical solution of this problem.

The mathematical formulation of the problem is:

$$\begin{aligned} -\nu\Delta\mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p &= 0 \quad \text{in } \Omega, \\ \mathrm{div}\mathbf{u} &= 0 \quad \text{in } \Omega, \\ \mathbf{u} &= \mathbf{u}_0 \quad \text{on } \Gamma_1, \\ \mathbf{u} &= 0 \quad \text{on } \Gamma_2, \end{aligned}$$

where ν is the kinematic viscosity and $\mathbf{u}_0 = (1,\, 0)^T$.

Let L be a characteristic length and v be the characteristic velocity. Since $L \approx 1$ and $v \approx 1$, the Reynolds number becomes

$$Re = \frac{vL}{\nu} \approx \frac{1}{\nu}.$$

Our major interest is small kinematic viscosity resulting in large Reynolds numbers. In this example $\nu = 3e-4$, i.e. $Re \approx 3,300$.

We solve this problem using the stable pair of finite elements: P_2 for velocity $\mathbf{u}$ and P_1 for the pressure p. Modeling of convection-dominated flows, $Re \gg 1$, requires to use either SUPG stabilized numerical schemes or specially designed adaptive meshes. Here, we employ the second approach.

Step 1. The first step in the numerical solution is the construction of an initial mesh. This can be done using library Ani2D-AFT. For large Reynolds numbers, the mesh has to resolve boundary layers. To avoid generation of a very fine quasi-uniform mesh, we use capability of the package Ani2D-AFT to generate regular meshes with local mesh size given by a user specified function $h(\mathbf{x})$. Let

$$h(\mathbf{x}) = \sqrt{c_0^2\, R^{1.5} + m_0^2}, \quad c_0 = 0.013, \quad m_0 = 0.005,$$

where R is the distance from point $\mathbf{x}$ to the boundary of the unit square. This mesh size function is named as *meshSize* in file `forlibaft.f`. It is registered with the library Ani2D-AFT in `main.f` as follows:

```
Real*8   meshSize
EXTERNAL meshSize
call registersizefn(meshSize)
```

Then, we define four straight boundary edges and generate the mesh using library routine *aft2dboundary*. The mesh is shown on the left in Fig. 12.1. It contains 6522 vertices and 12238 triangles. The boundary edges of this mesh will be marked with integers 1 to 4 corresponding to four sides of the unit square. The mesh is isotropically refined towards the domain boundary where the solution is expected to have boundary layers.

Step 2. The second step is the generation of the discrete system and its iterative solution by a few Picard iterations. Each step of the method requires generation of a linear finite element problem for a linearized convection operator and the solution of it by LU sparse factorization. The generation of a finite element system $Ax = F$ involves usage of a few routines from the library Ani2D-FEM:

1. *markDIR* marks boundary nodes with the maximal color of their parent edges.
2. *BilinearFormTemplate* returns the finite element matrix in the sparse compressed column format set up in the control parameter `controlFEM(1)`. The convection velocity is passed inside this routine via the parameter `SOL`. This velocity is used inside user-written routines in file `forlibfem.f`.

Generation of the finite element matrix A and the right-hand side F is controlled via five routines:

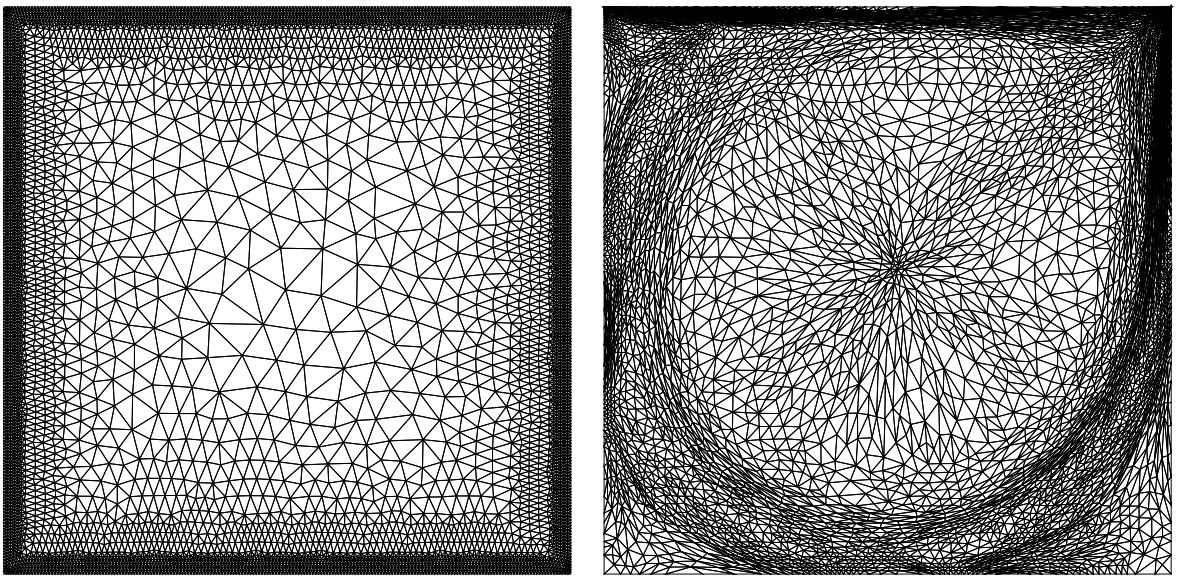


Figure 12.1: Initial and final meshes.

FEM2Dext creates an elemental matrix for velocity and pressure. Each triangle has 6 velocity unknowns and 3 pressure unknowns. Therefore, the size of the elemental matrix is 15. Note that we have to return vectors `templateR(15)` and `templateC(15)` that specify location of our degrees of freedom within the elemental matrix. This routine assembles the saddle point elemental matrix `A`, the corresponding right-hand side `F` and imposes locally boundary conditions. It processes the convection velocity `dDATA` and passes it to routine *Dconv*.

Ddiff returns viscosity coefficient ν. Since this is a scalar, we need to populate only the first diagonal entry in matrix `Coef` with the fixed leading dimension 4.

Dbc returns the Dirichlet boundary condition at point (x, y). Here we use our knowledge of markers assigned to boundary edges. Internal edges have zero marker.

Dconv returns value of the convection velocity at point (x, y) using, for each component, three values at vertices and three values at edge mid-points (degrees of freedom for the P_2 finite element).

Drhs returns the source term which is zero.

Step 3. The inexact Newton method is called after a few Picard iterations. It is a part of the library Ani2D-INB that requires a few user-written routines:

prevec solves a linear system with a preconditioner. The preconditioner is given by the LU decomposition of the last matrix in the Picard iterations.

fnlin evaluates the nonlinear residual. It requires access to some mesh data that we pack into parameters `rW(ipRMesh)` and `iW(ipIMesh)` (see `main.f`) corresponding to the parameters `rpar` and `ipar` in Ani2D-INB.

Step 4. After solving the nonlinear problem we adapt the mesh to its solution. After that the solution process is repeated again. The reason behind such an approach is that (a) error on the adapted mesh will be much smaller than on the original mesh and (b) complexity of the FEM solution on smaller but adapted mesh is lower compared to the FEM solution on the original mesh.

The mesh adaptation requires a specially designed metric generated by the library Ani2D-LMR. We use the velocity module and routine *Nodal2MetricVAR* to build a tensor metric `Metric`. This metric is

aligned with our solution.

Step 5. Unstructured mesh adaptation uses the above metric and library Ani2D-MBA. We call routine *mbaNodal* to build a new mesh with approximately 10000 triangles which is specified via control parameter `control(2)`. The mesh build with this library resembles the mesh shown on the right picture in Fig. 12.1. The shown mesh has been obtained after 5 iterations of the adaptation algorithm.

Step 6. For a steady-state problem, we could start each new iteration with the same initial guess. However, the existing solution will be a much better initial guess. The objective of this step is to interpolate the existing FEM solution to the newly adapted mesh. Data interpolation between two unstructured meshes is performed by the package Ani2D-PRJ. This package implements the conventional finite element L^2 projection. We have to call routine *assemble_rhs* from library Ani2D-PRJ. It assembles the right-hand side vector b for the L^2 projection problem $Pu = b$. The mass matrix P is generated by calling routine *BilinearFormTemplate* from library Ani2D-FEM. Finally, the solution of the linear system is performed with library Ani2D-LU.

Step 7. The discrete solution (streamlines, velocity components, pressure) and the underlying mesh can be visualized using library Ani2D-VIEW.

In summary, the example shows usage of eight packages: Ani2D-AFT, Ani2D-FEM, Ani2D-INB, Ani2D-LMR, Ani2D-MBA, Ani2D-PRJ, Ani2D-LU, and Ani2D-VIEW. Streamlines of solutions after the first and the fifth iterations are shown on Fig. 12.2. We do not have a way to quantify accuracy of these two solution. However, qualitatively the second solution is better: the secondary vortexes are resolved more accurately. The streamlines are smoother on the adaptive mesh which indicates directly proper error equidistribution and implies indirectly smaller error.

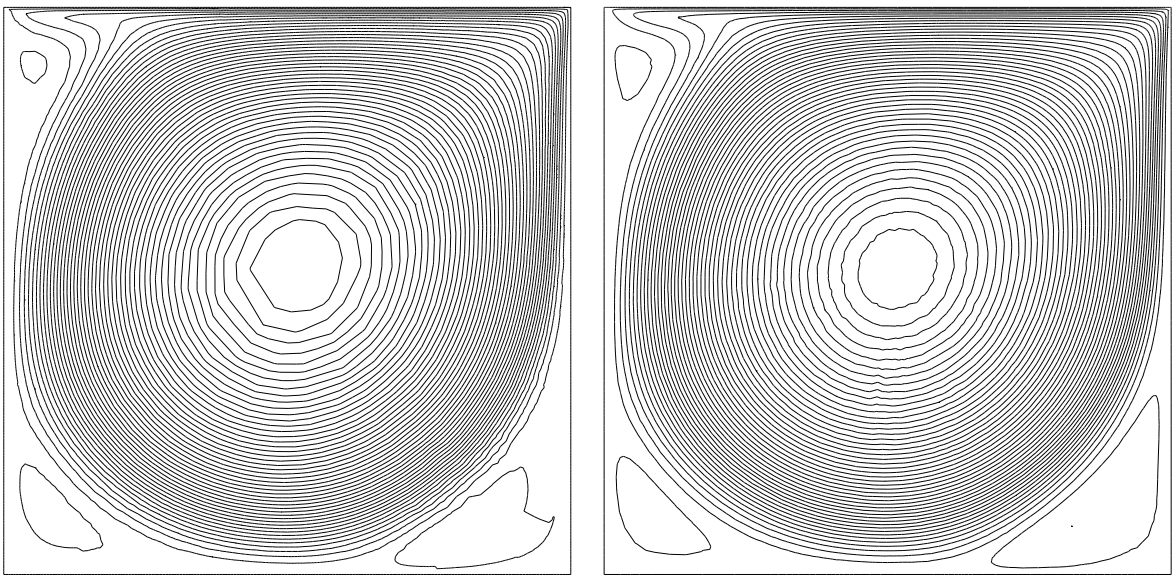


Figure 12.2: Velocity streamlines on initial and final meshes.