CC2D_MOVBC_STRUCTURED

I bet, a lot of people coming from the FEAT world will get irritated when starting with the new CC2D code – because the changes are fundamental. To give a slight help starting with the new code, I will give a short introduction here.

At first let me give a short description about the paradigm change that I realised with that piece of code. The fundamental changes in whole code can be summarised as follows:

DON'T USE COMMON BLOCKS WHEREVER POSSIBLE!

DON'T USE *IMPLICIT* STATEMENTS! USE *IMPLICIT NONE*!

INFORMATION MUST BE STRUCTURED FUNCTIONALLY!

EVERY INITIALISED STRUCTURE MUST BE CLEANED UP ANYWHERE!

THE CODE DESIGN IS BLACK-BOX / TOP-DOWN WHEREVER POSSIBLE, BOTTOM-UP CALLS ARE ONLY ALLOWED WITH CALLBACK ROUTINES.

To realise all this is a little bit tricky, especially the thing with the COMMON blocks – but only at a first glance. Let me describe how this is done:

1. Structured programming

To realise the COMMON-block-less programming style, I'm using *structures* motivated by C/PASCAL, object oriented programming in C++ and the realisation in UMFPACK4. In C/C++, information is typically structured like in the following example:

```
struct STria {
     integer NEL;
     integer NVT;
     integer NMT;
     integer LCORVG;
     ...
}
```

To access information in such a structure, a variable is defined,

```
STria OneTriangulation;
```

and by using an appropriate operator named ".", the sub-information can be accessed, e.g.

```
FOR (int i=0; i < OneTriangulation.NEL, i++) {
  ...
}
```

Such structured programming is also used in Fortran 90, but it's not standard Fortran 77 (although the SUN Fortran 77 compiler allows something like that, but probably no other compiler). So overcome this problem, I recognised a possible solution in the UMFPACK4 package, where a simple array was used instead of a structure:

```
double Control [UMFPACK_CONTROL];
Input argument, not modified.

   Control(1) printing level
   Control(2) dense row parameter
   Control(3) dense column parameter
   Control(4) partial pivoting tolerance
   Control(5) BLAS block size
   ...
```

I glued both ideas together to form a pseudo-structure in Fortran 77. Such a structure is defined simply by one or two arrays: An INTEGER-array for integer values and/or a DOUBLE PRECISION array for double precision values, e.g.

```
INTEGER STRIA(*)
...
STRIA(6)     =    NEL, Number of elements
STRIA(7)     =    NVT, Number of vertices
STRIA(8)     =    NMT, Number of edges
```

Using explicit offsets to access information in such an array (like `STRIA(1)`) is of course not maintainable. Therefore, the position of each entry in such a structure is defined in an INCLUDE file using constants. e.g. in STRIA.INC:

```
INTEGER ONEL,ONVT,ONMT,ONVE,...
...
PARAMETER (ONEL    =  6)
PARAMETER (ONVT    =  7)
PARAMETER (ONMT    =  8)
```

Such constants always start with "`Oxxxxx`". The length of such a structure is always defined by a `SZxxxx`. constant, e.g.

```
INTEGER SZTRIA
PARAMETER (SZTRIA  =  96)
```

Many routines now get such a structure as a parameter, e.g.

```
************************************************************
* Write GMV triangulation
************************************************************

SUBROUTINE GMVTRI (MFILE,TRIA,IEDG,NCELLS,NVERTS)
...
INCLUDE 'stria.inc'

INTEGER TRIA(SZTRIA)
...
```

The INCLUDE-file defines the offset positions of the information in such an array. The routine can use this information now to access the specific information of the structure, e.g.:

```
DO I=0,TRIA(ONVT)-1
  WRITE(MFILE,'(E15.8)') REAL(DCORVG(I,1))
END DO
```

!!! NEVER USE ABSOLUTE OFFSET POSITIONS DIRECTLY !!!

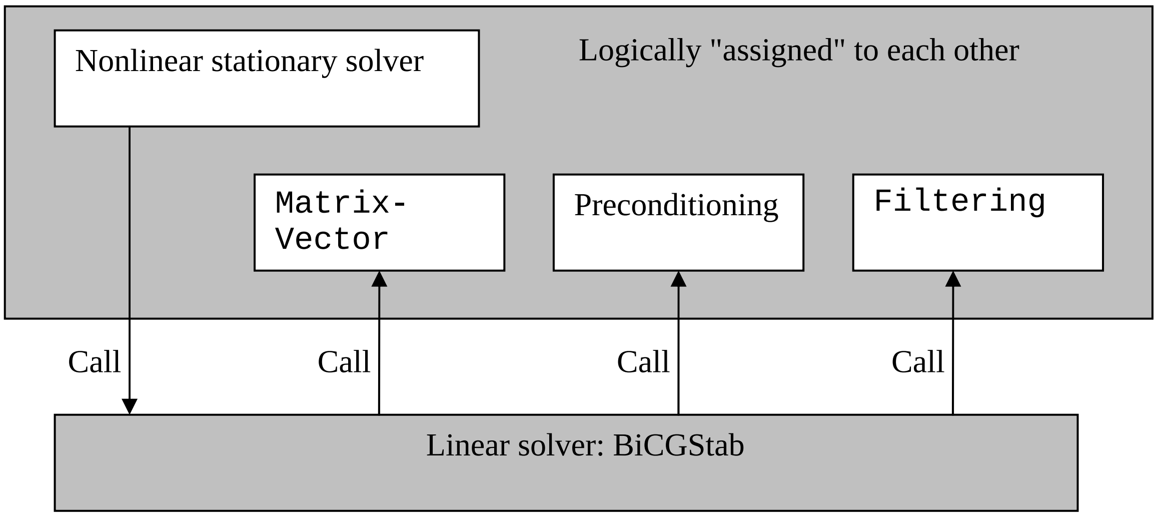
(like e.g. "`DO I=0,TRIA(6)-1`"). It might be that the offset of such an information might have to be changed in later versions (e.g. when adding further variables when going from 2D to 3D). Using this OFFSET-style, only the offset positions in the INCLUDE file have to be changed, while the complete program still keeps running after recompilation!

2. Top-down structure – Passing structures

Wherever possible, the programming style is top-down and black-box. Each solver is kept as black-box solver, the geometry library is kept black box, the grid adaption is black box – even the nonlinear stationary and nonstationary solvers are kept black-box!

The only exception to this top-down style is the CALLBACK routine functionality. Most of the solver routines e.g. need callback routines that provide these solvers with information about the problem. But while it's simple to provide a subroutine like the above GMVTRI with necessary information, piping problem related information to callback routines of solver components without using COMMON blocks is a little bit more advanced.

To overcome this problem, we *assign* each callback routine to the caller of the solver, which means that both routines are closely related and must exchange information *through* the solver with each other. Example:



For this purpose, at least one user defined integer and double-precision parameter block is passed to a solver, which is then passed to the callback routines. For instance, analyse the call of the BiCGStab solver:

```
SUBROUTINE II01X(IPARAM, DPARAM, IDATA,DDATA,
                 NEQ,DX,DB,DR,DR0,DP,DPA,DSA,
                 YMVMUL,DCG0C,DFILT)
INTEGER IPARAM(SZSLVI)
DOUBLE PRECISION DPARAM(SZSLVD)
INTEGER IDATA(*)
DOUBLE PRECISION DDATA(*)
...
```

We can see here four different integer / double-precision parameter blocks that do not appear in the BiCGStab-Solver of the standard FEAT2D library: IPARAM, DPARAM, IDATA, and DDATA:

IPARAM and DPARAM are parameter blocks with input- and output variables. Here the user can define e.g. the stopping criterion, the maximum number of steps and so on. When the solver finishes, IPARAM and DPARAM return the status about the success of the solver (number of steps needed,...). These parameter blocks base on the "*general solver structure*", which is a common basis for all linear solvers and can be found in SSOLVERS.INC.

IDATA and DDATA on the other hand are user-defined data blocks that are not used in the BiCGStab solver directly. Instead, these blocks are directly passed to all callback routines which the user has to provide to BiCGStab (`YMVMUL,DCG0C` and `DFILT`), e.g.:

```
YMVMUL : Performs matrix vector multiplication
YMVMUL = SUBROUTINE YMVMUL (DX,DAX,NEQ,A1,A2,IPARAM,DPARAM,
                            IDATA,DDATA)
                            ^^^^^ ^^^^^
```

That way, information is passed to callback-routines. The caller of BiCGStab now typically defines these structures on the stack as local variables, e.g.:

```
INTEGER IDATA(50)
DOUBLE PRECISION DDATA(50)
```

and fills it with information, e.g. the handles about the matrix which is used in a matrix-vector multiplication, then calls the solver:

```
IDATA(1) = LA
IDATA(2) = LCOL
IDATA(3) = LLD
...
CALL II01X(IPARAM, DPARAM, IDATA,DDATA,..., MTML7X,...)
```

On the other hand, the callback routine can use the user-provided data blocks to perform its task, e.g.:

```
SUBROUTINE MTML7X (DX,DAX,NEQ,A1,A2,IPARAM,DPARAM,
                   IDATA,DDATA)
...
INTEGER KA,KCOL,KLD
KA   = L( IDATA(1) )
KCOL = L( IDATA(2) )
KLD  = L( IDATA(3) )
CALL LAX17(DWORK(KA),KWORK(KCOL),KWORK(KLD),NEQ,DX,DAX,A1,A2)
...
```

In the above style you simply write the MV-multiplication on your own, locally to your routines/solver. Arbitrary information can be appended to IDATA/DDATA if you need it (e.g. by realising IDATA/DDATA on the DWORK/KWORK heap) – maybe the complete information for a Multigrid preconditioner with matrices and vectors on all levels...

3. Structure initialisation and cleaning up

The major goals in structure oriented programming like described above are

- a clean definition of input, output and auxiliary information,
- transparency: information if functionally grouped and does not come out of the nowhere,
- deterministic behaviour: a routine called twice with the same input block must produce twice the same output,
- every dynamically generated information must be cleaned up/released somewhere else

As an example for this, let's consider the stationary solver NSDEF2. To configure the solver, one has to call different initialisation routines, where every initialisation routine that allocates memory on the heap is and must be followed by a routine for cleaning up:

```
CALL INSTSL (...)              -> initialise the stationary solver, allocate
                               memory where necessary
  CALL NSDEF2 (...)            -> call the solver
  CALL NSDEF2 (...)            -> probably multiple times

CALL DNSTSL (...)              -> release solver structures and dynamically
                               allocated information
```

If information must be allocated, computed and recomputed frequently (like RHS or nonlinear matrices), the following similar form is used:

```
CALL ININSM (...)              -> Initialise matrices, allocate memory
  CALL GENNSM (...)            -> update/calculate matrices
  CALL GENNSM (...)            -> probably multiple times
CALL DONNSM (...)              -> release matrices again
```

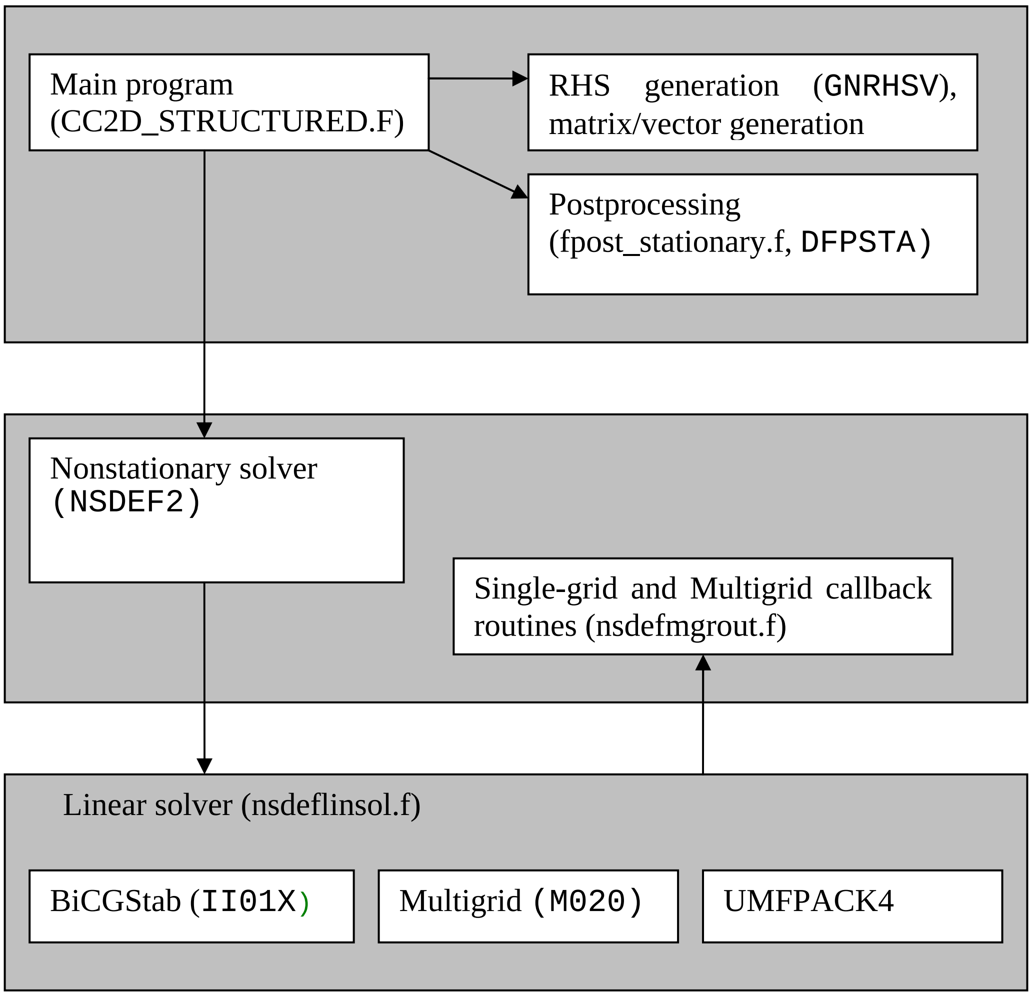
At the end of the program, the `L()`-array with all handles MUST BE EMPTY, otherwise there is a memory leak!!!

4. Structure of the program

Concerning the structure of the main program to solve stationary or nonstationary, one can say that here the most dramatically changes have been made. The two solvers – stationary and nonstationary – are completely capsuled and written in a black-box style (hopefully). The result of this is that there is no more one call to one solver! Solving stationary and nonstationary are two different processes – and so they are handled by the two solvers NSDEF2 and NONST2. The effect of the old "hack" to see the stationary solver as a sub-case of the nonstationary solver (by solving from T=0..1 and getting the stationary solution at T=1) was that only one postprocessing routine was necessary. This is now broken up to have a clean separation between the two cases: There are different postprocessing routines for the stationary and nonstationary case. This is also necessary, as the postprocessing routines differ in their use! We describe the rough overview over the new solver structure in the following subsections:

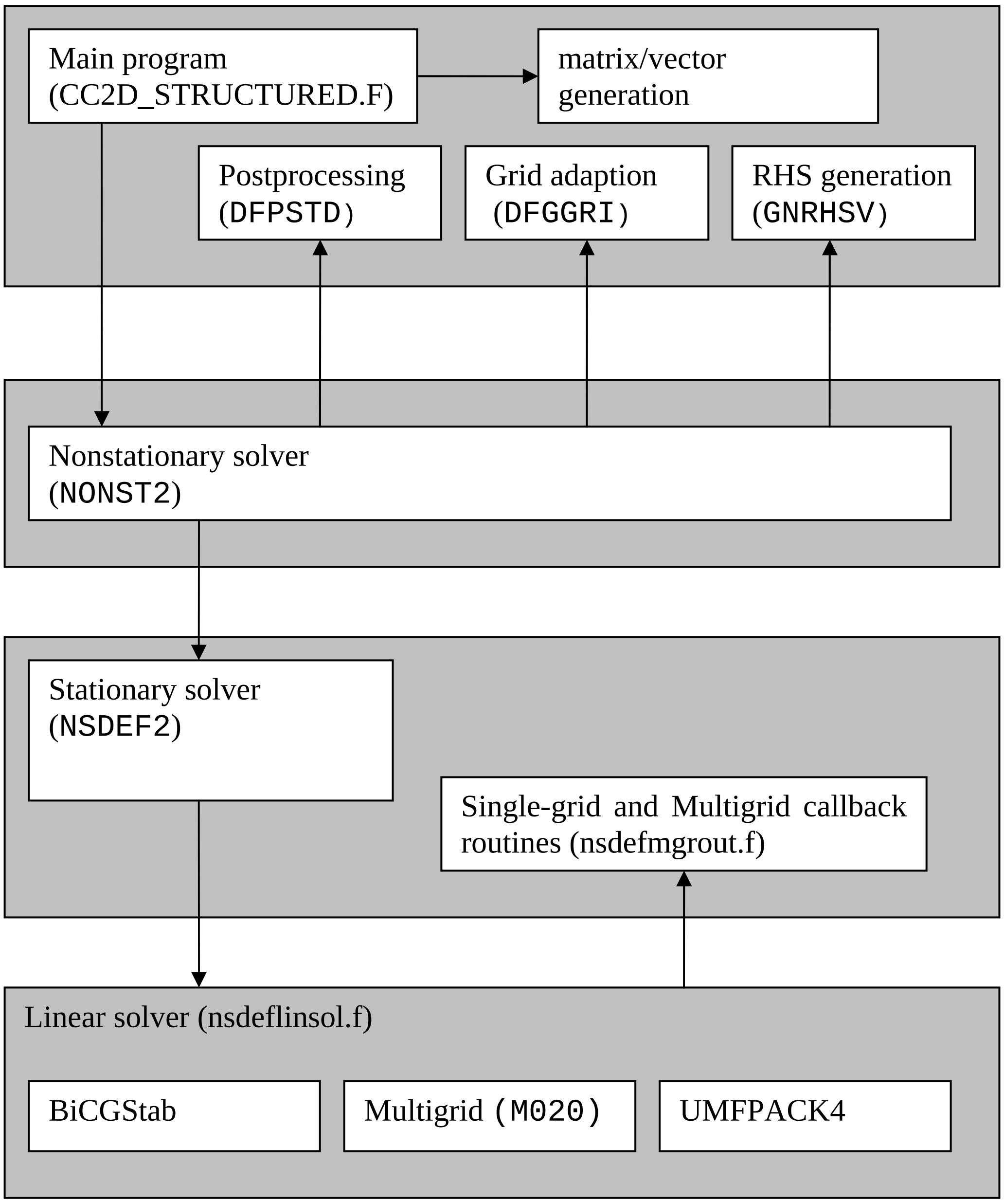
4.1. The stationary solver NSDEF2

The following diagram visualises which main components interact with each other and which are called from where. The main program directly calls the stationary solver `NSDEF2`, gets a solution and then calls the postprocessing routine `DFPSTA`, which performs the postprocessing for the stationary case.


Main program (CC2D_STRUCTURED.F)
RHS generation (GNRHSV), matrix/vector generation
Postprocessing (fpost_stationary.f, DFPSTA)
Nonstationary solver (NSDEF2)
Single-grid and Multigrid callback routines (nsdefmgrout.f)
Linear solver (nsdeflinsol.f)
BiCGStab (II01X)
Multigrid (M020)
UMFPACK4

4.2. The nonstationary solver NONST2

The structure for using the nonstationary sovler `NONST2` is slightly more advanced than the use of the stationary solver. The crucial point is, that – because of the time-dependent nature – the postprocessing routine is realised as callback-routine of the main program. It's called frequently over time with a flag `IFINL`, which indicates the current position inside of the algorithm. Depending on this flag, the postprocessing routine must decide when and how to do the postprocessing.



Main program (CC2D_STRUCTURED.F)
matrix/vector generation
Postprocessing (DFPSTD)
Grid adaption (DFGGRI)
RHS generation (GNRHSV)
Nonstationary solver (NONST2)
Stationary solver (NSDEF2)
Single-grid and Multigrid callback routines (nsdefmgrout.f)
Linear solver (nsdeflinsol.f)
BiCGStab
Multigrid (M020)
UMFPACK4

Remark: The central structure that is known in all blocks (except for the linear solver layer) is the TxxxASSEMBLY structure defined in the file SASSEMBLY.INC. As information about the discretisation of the problem is nearly everywhere necessary, information in this structure is nearly-global. If additional problem-related information must be transferred through the layers, it can be appended to this structure!

5. But there are still COMMON blocks...

Yes, some information is of global nature! There are exactly six circumstances at the moment, where the COMMON block architecture is still used:

- Memory Management
  The realisation of pseudodynamic memory management must be global, this will never change.
- Output to terminal/file, error handling
  For the output to terminal/file and the error handling, the appropriate information/file handles are stored in COMMON block variables, since this is global information; we come to that later.
- Cubature and Finite Element handling
  The cubature and information exchange with Finite Elements Exxx is still handled using the /CUB/ and /ELEM/ COMMON blocks for compatibility – although this is a little bit nasty when switching from one discretisation (Q1~, Q2) to another (Q0, Q1). This might be a matter of change.
- Parametrisation
  At the moment, the parametrisation with PARX/PARY/TMAX (and the new TNBC) is still realised with COMMON blocks as the program is designed to work with only one underlying geometry. This might be a matter of change in the future.
- Generation of triangulation
  On the beginning of each program, the mesh information on all levels is created using the standard refinement routines of the FEAT2D library for compatibility. After the meshes are created, all mesh information is usually kept in a TRIAS(.,.) structure, accessing the /TRIAx/ COMMON blocks is neither used nor necessary.
- Variables in DAT files
  The variables stored in DAT files are read into COMMON block variables on start of the program; we'll come to that later.

6. Initialisation process

6.1. DAT-files for the different purposes

Theses not *one* DAT file anymore for everything. The new CC2D code uses so many different components that it was reasonable also to split the main DAT file. There are now DAT files for discretisation, linear solver, nonlinear solver, time discretisation, geometry,... All the files can be found in the "./data" subdirectory.

6.2. Reading DAT files

When a program is started, it has to read in all parameters from the DAT files and initialise. In contrast to the previous INIT-process, the reading of the parameters and the initialisation is now completely decoupled.

Every sub-component of CC2D (linear solver, nonlinear solver, time-discretisation, post-processing,...) has its own "RDxxxx"-routine, whose only duty is to read parameters of a DAT file into a COMMON block. These routines can be found in files which are named "RDxxxxxxx.f", e.g.

• rddiscretization.f	-> reads discretisation parameters
• rdlinsol_cc2d.f	-> reads parameters of linear solver
• rdnonlinsol_cc2d.f	-> reads parameters of stationary solver
• rdoutput.f	-> reads parameters for output to terminal
• ...	

Most of the files can be found in the `./src/init` subdirectory. Files that belong to components which are independent of CC2D and have nothing to do with the problem (e.g. grid adaption, geometry support) are kept in their own directory – this makes it easier to extract that component to include it into a different program.

These RDxxxx-routines are called in the INIT2 subroutine at the beginning of the program. INIT2 does nothing more! The initialisation of the problem is left to the main program!

After reading into COMMON blocks, the information must be transferred to the structures of the solver components. Every solver component provides a subroutine that is responsible for that purpose. The user has to call the READ-routine and then the INITIALISATION routine, which initialises a structure with COMMON block information. After initialisation, each structure is standalone, can be modified when needed and reset to the initial values by calling the (RELEASE and) INITIALISATION routine again, e.g.:

```
  CALL RDNLCC (...)              -> read nonlinear solver parameters
  ...
  CALL ININSD (...,IC2PAR=1)     -> initialise nonlinear solver parameter block,
                                    transfer variables from COMMON block to
...                                 parameter block
  CALL NSDEF2 (...)              -> solve the problem
```

All these `INIxxx`-routines have a parameter `IC2PAR` which – when set to 1 – lets the initialisation routine transfer COMMON block variables to the structure. If `IC2PAR` is set to 0, the structure is initialised only with standard parameters for the solver!

Remark: Don't be confused in this example with `ININSD` and `INSTSL` from the example above! `ININSD` is called in the `INSTSL` routine, so the actual calling strategy is:

```
  CALL RDNLCC (...)                  -> Read parameters
  ...
  CALL INSTSL (...)                  -> Initialise stationary solver
  -> CALL INISTS (...,IC2PAR=1)          -> Initialise NSDEF in particular
  ...
  CALL NSDEF2 (...)                  -> Call NSDEF
  CALL DNSTSL (...)                  -> Clean up stationary solver
```

The reason is simply, that the nonlinear solver is treated as *exchangeable*! `INSTSL` initialises the nonlinear solver – whichever it is. In the standard example, it's `NSDEF2`, so `INSTSL` calls `ININSD` which belongs to `NSDEF2`.

There is no `DONNSD` routine for cleaning up here, since no dynamic information is kept in the standard parameter blocks on `NSDEF2`. Dynamic information is only allocated/released by INSTSL/DNSTSL. In a future version, NSDEF2 might be exchanged by another solver and then INISTSL has to call another initialisation routine, and DNSTSL might probably have to call an appropriate DONxxx-routine for that solver.

6.3. Version tag

Every DAT-file has a version tag, which can be found at the very beginning of the file, e.g.

```
-------------------------------------------------------------
----- Spatial discretization
...
-------------------------------------------------------------
100      Version information tag. Must fit to program version.
-------------------------------------------------------------
```

This version tag is compared with the hardcoded version tag in the RDxxxx-routine that reads the DAT file into memory. The initialisation will fail if the version tag is different from the hardcoded one. If any substantial change happens to a DAT file, it's advisable to increase/change the version tag in the DAT file and in the RDxxxx-routine. This prevents the DAT file from being able to be read into another, previous (or later) version of the CC2D software. The user will get an error message, is forced to check which changes the DAT file has undergone and will have to adapt his/her old DAT file to match the requirements of the new one.

7. Output and string handling

The handling of the output to terminal/file has been greatly improved (hopefully). There are now two COMMON blocks responsible for the output:

```
  COUT.INC
  ->      COMMON /OUTPUT/ ...,MT,...,MTERM,...

  CFILEOUT.INC
  ->      COMMON /FILOUT/ MFILE,CFLPRT
```

At the beginning of the program, the variables here are initialised in the INIT2 routine:

- MT is the message level of the program
- MTERM is the file handle for writing output to terminal
- MFILE is the file handle when writing to a file
- CFLPRT is the name of the protocol file

When output should be written to screen/terminal, it's no more necessary to double the code with once writing to MFILE and once writing to MTERM. Using the central output routines CNOUTS/CNOUTD simplify this, especially in combination with the string handling library. CNOUTS/CNOUTD accept file handles to the terminal, to a file, a message level indicator and the message to write. The following example demonstrates how to write output:

```
INCLUDE 'cout.inc'
INCLUDE 'cfileout.inc'

CHARACTER CSTR*(255)
INTEGER I

DO I=1,100
  WRITE (CSTR,'(A,I4)') 'Call number ',I
  CALL CNOUTS (MT-2,MTERM,MFILE,.TRUE.,CSTR)
END DO
```

This example writes to screen "Call number xxx" with xxx=1..100, but only if MT is high enough. For MT=0,1,2, nothing is printed, MT=3 prints the output to the file and MT=4 prints it to file and screen.

Alternatively if multiple string components must be concatenated, one can use CNOUTD:

```
INCLUDE 'cout.inc'
INCLUDE 'cfileout.inc'

INTEGER LSTR,I

DO I=1,100
  LSTR = STNEWC (.FALSE.,'Call number ')
  CALL STCATI (LSTR,I,0,.FALSE.)
  CALL CNOUTD (MT-2,MTERM,MFILE,.TRUE.,LSTR)
  CALL STDIS (LSTR)
END DO
```

Note that because of the string handling, leading spaces of the number are trimmed away!

8. Parametrisation, fictitious boundaries, geometries

8.1. *The global parametrisation*

The global parametrisation is still handled by PARX/PARY/TMAX. However, these routines are not in the PARQ2D.F-file anymore. They are now capsuled in the file PARQ2DWRAP.F and serve as a wrapper for the FEAT2D and OMEGA2D parametrisation!

The parameter `IMESH` in the PARAMTRIANG.DAT DAT file is a switch that decides on which type of parametrisation is active. For IMESH=1, the OMEGA2D parametrisation is active. In this case, PARX/PARY/TMAX will *always* use the standard .PRM/.TRI parametrisation technique, which were formally realised by PARPRE.F. The appropriate routines can be found in PARQ2DOMEGA.F.

With IMESH=0, the FEAT2D parametrisation can be activated. In this case, PARX/PARY/TMAX will switch to use the user provided routines FPARX/FPARY/FTMAX in the user defined file PARQ2D.F.

Furthermore there was another routine added. The routine TNBC (or FTNBC in the user provided file PARQ2D.F) allows to determine the number of boundary components in the geometry, independent from a triangulation! That way, the parametrisation is completely decoupled any triangulation and completely capsuled in the PARQxxx-files.

8.2. *Fictitious boundary support*

To deal with fictitious boundaries, the files FICTBDRY.F and FICTBDRYPC.F can be modified by the user. Like the PARQxx-files, these files capsule the complete interface to the user, like determining number of fictitious boundary components, whether a point is inside of a fictitious boundary component or not and so on. FICTBDRY.F defines the standard routines for "simple-type" objects like circles, rectangles and so on, where information (like distance e.g.) can be computed on the fly. FICTBDRYPC.F on the other hand adds the more advanced handling of *precomputed* fictitious boundary components and is designed for more complex objects. For precomputed fictitious boundary objects, CC2D allows calculating important quantities *in advance* (e.g. the distance of an object to all points in a triangulation using level-set or other techniques), so they can be used later without any computational effort.

8.3. *The geometry library*

In the `./src/geometry` subdirectory, a geometry library can be found. This supports simple type objects (circle, rectangle) as well as complex type objects (USS Enterprise, Objects given by line segments, Groups of objects) and offers a wide range of functions that can be applied to these (Rotation, Scaling, Grouping, Distance calculation,...). These routines are typically used in the handling of fictitious boundaries, where a more or less complex object has to be put into the flow.

Apart from this pure geometric stuff, the geometry library also contains routines to deal with triangular and quadrilateral elements, like transformation from the reference element to the real element, back-transformation from the real to the reference element. This might help somewhere...

9. Triangulation

9.1. *Triangulation structures*

Prof. Turek once gave the statement:

"*We must be the masters of the grids.*"

Using the new triangulation structures introduced in this CC2D-version will hopefully allow reaching that goal! What is this thing about? Well, remember how triangulation information was maintained in the old FEAT2D style:

- There were the /TRIAx/ COMMON blocks which hold all information about the current active triangulation
- For Multigrid, all level information were stored sequentially in appropriate COMMON blocks. E.g. there were arrays KLCORVG(NNLEV), KLVERT(NNLEV), KLMID(NNLEV),... for all levels.
- To switch from one level to another, SETLEV was used.

This handling was very inconvenient in my opinion. Imagine, a subroutine calls SETLEV and switches to another level but forgets to return to the previous level, which was always a nasty bug to find.

But there was a very easy trick that simplified the total mesh-handling vastly in my opinion: *Transpose the structures!* Instead of storing first the LCORVG-handles of all levels, then LVERT of all levels, then... behind each other, simply store all information of one level, then the information of the next one and so on. The result was the structure STRIA:

```
INTEGER STRIA(SZTRIA)
...
STRIA(6)     =    NEL, Number of elements
STRIA(7)     =    NVT, Number of vertices
STRIA(8)     =    NMT, Number of edges
...
```

which was already introduced earlier. This array collects all information of one level. To access a specific information, the Oxxxx-constants in the STRIA.INC file are used, e.g.:

```
NVT = STRIA(ONVT)
```

The biggest advantage of this structure is its ability to be passed as parameter to functions – in the single-grid as well as in the Multigrid case! All routines that need a triangulation can now accept one by a parameter, e.g.

```
SUBROUTINE GMVTRI (MFILE,TRIA,IEDG,NCELLS,NVERTS)
...
INCLUDE 'stria.inc'

INTEGER TRIA(SZTRIA)
...
```

and will therefore always work with the "correct" triangulation (that one which is passed to them). Moreover for routines that work on multiple levels (e.g. Multigrid), the triangulations on all levels can be kept in a single INTEGER array,

```
INTEGER TRIAS(SZTRIA,NNLEV)
```

To access NVT on level 5, one can now directly write:

```
NVT = TRIAS(ONVT,5)
```

And if a subroutine only needs information on one level, it can be called directly passing only the information needed. To write out the triangulation of level ILEV to a GMV file, one could write e.g.

```
CALL GMVTRI (65,TRIAS(1,ILEV),0,NCELLS,NVERTS)
```

without doing complicated `SETLEV`-things, which might leave the /TRIAx/ COMMON blocks at a wrong level you don't want it to be...

Although the whole CC2D solution process does therefore not need the `/TRIAx/` COMMON blocks anymore, they are still there – for the initialisation of the triangulations! This is still done using the FEAT2D library. For convenience, the COMMON blocks are defined in the file CTRIA.INC. Specialised "SETLEV" – and "GETLEV" – routines to read a triangulation structure from the COMMON block or write it to them can be found in

INITRIA.F: `C2TRIA`() reads the COMMON block and creates a triangulation, while `TRIA2C` writes the triangulation to the COMMON block.

9.2. Extended triangulation information

For higher convenience, the triangulation structures maintained in the new CC2D-code are not completely kept as they come from the FEAT2D library. After refinement of the coarse grid, additional information is generated to the triangulation structure. Roughly speaking, the following major changes are applied (using the routine `GENETR`) to the standard FEAT2D triangulation before they are given for use to the main program:

- `KNPR` is no more! There was too less information in `KNPR` to be able to fulfil the requirements of the new code with all the fictitious boundary stuff and so on. In the initialisation phase, `KNPR` is replaced by `KXNPR`, which gives more detailed information about the vertices.
- What is `KMID`? Numbers of edge-midpoints on every element? Right and wrong! In the extended triangulation structure, `KMID` is interpreted as the *numbers of the edges* on an element. Ok, this coincides with the numbers of edge-midpoints at a first glance, but not completely. The extended triangulation structure allows more points on an edge than simply the midpoint. It's also possible to regularly distribute n points (e.g. 2 or 3, not only the one and only midpoint) on every edge! In this case, `KMID` will still give numbers to the edges, while the vertex-numbers must be computed using the number of the edge!
- What is `DCORMG`? X/Y-coordinate of the edge-midpoints? Right and wrong. If there's only one point on every edge, yes – to maintain compatibility to the old FEAT style. But if more than one vertex exists per edge, `DCORMG` saves more. In particular, `DCORMG` saves the coordinates of *all* vertices that are not corners of elements! This can even be the midpoints of the elements or Gaussian points on an element or an edge or whatever. A more detailed discussion about the "node numbering" and the definition of `DCORMG` in this case can be found in the comments in STRIA.INC and STRUCTRIA.F!
- If you have the index of a vertex on the boundary, you can use `KVBD` to get its vertex number. But what if you have the vertex number and you need the index??? In the old FEAT style, you had to search the whole `KVBD` array. In the new, you can use `KVBDI` to search much more efficiently for that.
- You can even create and/or use other advanced arrays – a collection of all fictitious boundary vertices, an array with the two vertices adjacent to an edge and some more...
- If you need to duplicate a grid, use the duplication routines. If you are a bit careful in using these routines, they allow even to share some information between different grids. This is used e.g. to realise that `DCORVG` is shared between all levels in the standard implementation.

10. Fortran programming standard

When going through the code, one could argue that it's not Standard Fortran 77 code. High level Fortran constructions like "DO..ENDDO", "WHILE..DO", "IMPLICIT NONE", "INCLUDE", bitwise functions like "IAND", "IOR", "NOT()" and other functions things, and one could argue, that the program is less portable and completely non-portable.

Well, that's true and not true. Indeed, these functions do not appear in the basic Fortran 77 standard – but they are not completely out of standard. These commands follow the *Fortran 77 Military Standard* "US MIL-STD-1753" of 1978, which was mandatory for Fortran languages sold to the US (cf. http://www.fortran.com/mil_std_1753.html). Therefore we assume these functions to be supported by every up-to-date compiler.

The only nonstandard difference is that ENDDO-statements are usually not labelled in the code – also a fact that is assumed to be understood by all modern compilers! Constructs like

```
      DO 10 I=1,NEQ
10    DX(I) = 1D0 / DX(I)
```

are usually replaced by

```
      DO I=1,NEQ
        DX(I) = 1D0 / DX(I)
      END DO
```

without the label "10" in front, the sub-block indented. This is much more in the spirit of structured programming than the use of jump marks as it visually helps to recognise what belongs together. GOTO commands are avoided wherever possible. The only exception is the use of GOTO as a replacement for the missing BREAK command in Fortran 77 in constructions like

```
      DO I=1,NVE
        IF (KVERT(I,IEL).EQ.IVE) GOTO 10
      END DO
      RETURN
10    CONTINUE
```

Returning out of functions in case of an error is usually done by the RETURN command instead of jumping with GOTO to the end, e.g.

```
SUBROUTINE MULVC (NEQ,DX,A)

IMPLICIT NONE
INTEGER NEQ,I
DOUBLE PRECISION DX(NEQ),A

IF (NEQ.EQ.0) RETURN

DO I=1,NEQ
  DX(I)=DX(I)*A
END DO

END
```

Note that all variables in this example are declared explicitly after an IMPLICIT NONE. This is a very important new design aspect in comparison to old FEAT routines. DON'T USE IMPLICIT! My personal experience is that it helps avoiding many side effects with undefined variables or typing errors, which are otherwise hard to find even with a debugger... although it's more programming work and is sometimes a bit nasty if one is not familiar with it. But realise: There is no other common high-level language on the market today which supports automatic declaration of variables with something like an IMPLICIT statement – and this has a reason...

11. Other changes and hints

- The "#" in the beginning of "#gmv", ... was removed. The directories are now named "gmv", ...
- The linear extrapolation in time was removed completely.
- For the memory management, the file CMEM.INC should be included. NNWORK is defined there!
- The code supports grid adaption, which is deactivated by default, though. To control it, use the DAT files "gridadapt.dat", "timediscr.dat". If you like to control on your own where to adapt a given grid, modify the file "mon.f".
- The main program can be found in CC2D_STRUCTURED.F in the ./CORE subdirectory.
- I'm testing an optimisation code with this software. It's deactivated by default. If you like to try yourself, modify the Makefile to include CC2D_STRUCTURED_OPT.F instead of CC2D_STRUCTURED.F.
- The ./INCLUDE subdirectory contains include files with global definitions about the problem: COMMON blocks for DAT file parameter, COMMON blocks as interface to the old FEAT2D library (CTRIA.INC), general COMMON blocks which are needed everywhere (CMEM.INC, COUT.INC,...) and general constants which have also global character (e.g. NNVE in CBASICTRIA.INC). Algorithm or component specific information (like triangulation structure, matrix/vector structure, grid adaption parameters, solver parameter blocks, ...) can be found directly in the corresponding ./SRC/xxxx subdirectory.
- All solvers/preconditioners/smoothers are now "children" of a general solver, thus sharing the so called "General solver structure" as common parameter block. Please read the comments in "SSOLVERS.INC" and "GSOLVERS.F" for a detailed description about the handling of linear solvers in extended calling convention.

For further implementation details, also read

- the initial comments in the file INIT2.F, which describe a special customisation of the user-defined field of the triangulation structures,
- the comments in all the .INC include files, which describe the structures that are used in the program,
- the README.TXT file in the ./SRC subdirectory,
- all the comments in all the files which are hopefully enlightening ^^

Happy coding :-)

Michael Köster, 25.02.2006