

Feat_FloWer Manual

Raphael Münster, Robert Jendrny

April 28, 2020

Table Of Contents

1	Installation	2
1.1	Prerequisites	2
1.2	Checking out the Code from the Repository	2
1.3	Checking out Mesh Repository	3
1.4	Building the Code from Source	5
1.4.1	Linux Systems	5
1.4.2	Windows Systems	6
1.4.3	Installation of the Partitioner	6
1.4.4	Using CMake-gui to build the Code	7
1.4.5	Commonly Used CMake Options Overview	8
2	Running a FeatFlower Application	9
2.1	Introduction to the Application Framework	9
2.1.1	Preprocessing and Mesh Generation	9
2.1.1.1	Mesh Decomposition using the Python Partitioner	10
2.1.2	Running the Application	10
2.1.3	Application Output and Postprocessing	11
2.2	q2p1_cc (by R. Jendrny)	11
2.2.1	Systems that can be solved	11
2.2.2	q2p1_param.dat	12
2.2.3	Source files	13
2.2.4	How to compile and run the code	16
2.3	Appendix	16
2.3.1	Recommended Shell Customizations For Linux Systems	16

Chapter 1

Installation

1.1 Prerequisites

The Feat_FloWer software package is accessible through the version control system *Git*. The main repository of the software is hosted on the servers of the TU Dortmund. In order to checkout a copy of the code an account that provides access to the LS3 servers is required, so it is recommended to get such an account. Furthermore, the code repository does not include the meshes used in the example applications. The meshes are stored in a different repository. Another prerequisite for FeatFloWer is the partitioner software. The partitioner is a tool to decompose a computational mesh into a set of submeshes. In a parallel computation each of the MPI processes gets assigned a mesh on which a solution will be computed. In this document we will describe how to build or install the following prerequisites:

- The mesh repository
- The partitioner tool

To get started with this task we will check out the FeatFloWer code from the repository in the next section.

1.2 Checking out the Code from the Repository

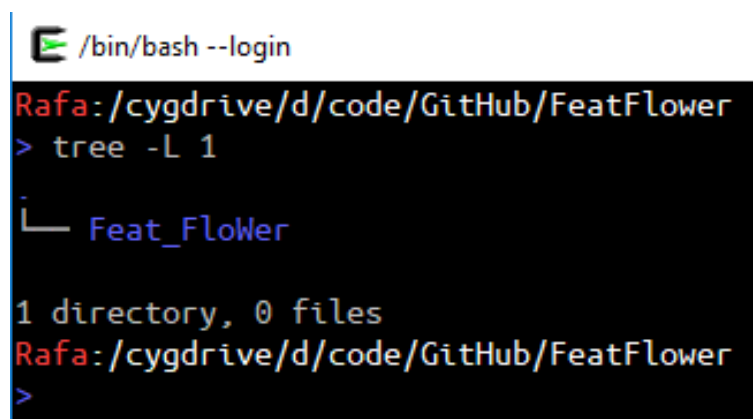
The code from the FeatFloWer repository contains the source code of the partitioner tool, so we need it to build the partitioner. We recommend that you to set up a new folder for your FeatFloWer installation. Create a new folder for your FeatFloWer installation at your chosen location and change to this folder by the following command:

```
> mkdir FeatFlower && cd FeatFlower
```

When you have acquired an appropriate account you can clone the FeatFloWer repository by following command:

```
> git clone --recursive ssh://username@server/home/user/git/Feat_FloWer.git
```

This will create a parent folder called FeatFlower and within this folder the source code will be contained. This is a preparation for building the code from source. The FeatFlower code supports only out-of-source builds, meaning the binary files are not built in the same directory as the source code. This is done in order to prevent that binary files or object files clutter the source directory or that these files show up in the version control system. You should now have a directory structure that looks like in fig. 1.1:



```
/bin/bash --login
Rafa:/cygdrive/d/code/GitHub/FeatFlower
> tree -L 1
.
└── Feat_Flower

1 directory, 0 files
Rafa:/cygdrive/d/code/GitHub/FeatFlower
>
```

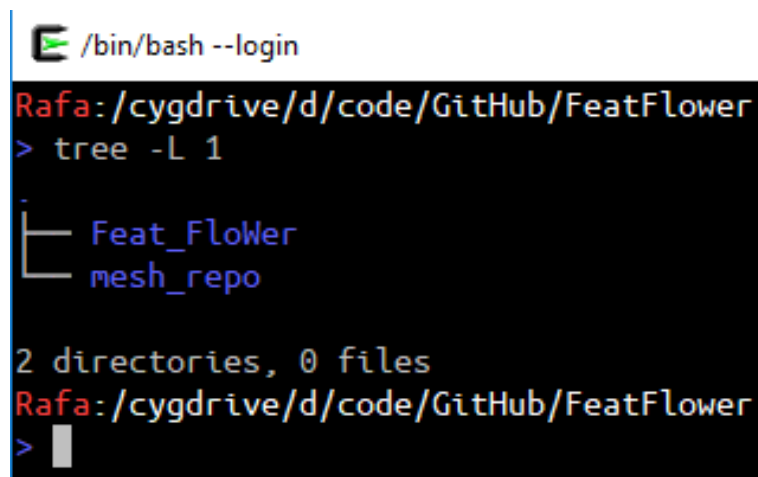
Figure 1.1: Directory structure after cloning the source repository

1.3 Checking out Mesh Repository

As we said, the meshes are stored in a separate git repository that can be cloned by the following command:

```
> git clone ssh://username@server/home/user/git/mesh_repo.git
```

If the above command does not complete successfully it is very likely that you do not yet have access rights to the git mesh repository. In this case please ask your administrator to grant you the required access rights. Your folder structure should look like in fig. 1.2 after cloning the mesh repository:

A terminal window with a dark background and light-colored text. The prompt is 'Rafa:/cygdrive/d/code/GitHub/FeatFlower'. The command 'tree -L 1' has been executed, showing a directory tree with two entries: 'Feat_Flower' and 'mesh_repo'. Below the tree, it says '2 directories, 0 files'. The prompt is now 'Rafa:/cygdrive/d/code/GitHub/FeatFlower' followed by a greater-than sign and a cursor.

```
/bin/bash --login
Rafa:/cygdrive/d/code/GitHub/FeatFlower
> tree -L 1
.
├── Feat_Flower
└── mesh_repo

2 directories, 0 files
Rafa:/cygdrive/d/code/GitHub/FeatFlower
> 
```

Figure 1.2: Directory structure after cloning the mesh repository

After you have checked out the mesh repository, it is time to set up an environment variable for the mesh directory. The environment variable is used during the installation of FeatFloWer, so that the mesh repository can be found. For the `BASH` or `ZSH` shells this can be done by adding the following line to your `.bashrc` or respectively `.zsh`:

```
> export Q2P1_MESH_DIR=/path/to/meshrepo/mesh_repo
```

If you are unsure how to set environment variables please learn how to set up and use environment variables in the terminal environment that you are using. Examples of shell configuration files can be found in section 2.3.1.

1.4 Building the Code from Source

The FeatFloWer code can be build on Linux and Windows operating systems (and probably on MacOS, but this is not tested yet). On Linux systems an installation of the GNU compiler is needed, including the gfortran compiler and the matching OpenMPI libraries. The minimum requirement for the GCC is version 4.9.2. On Linux systems the Intel compiler can be used as an alternative compiler. On Windows systems the Intel compiler is a necessary requirement. For Linux and derivatives we recommended setting a number of environment variables and adding some customization to your shell configuration files in order to work more efficiently within the FeatFloWer framework. See section 2.3 for more details.

1.4.1 Linux Systems

After you have cloned the repository, navigate to the *FeatFlow* folder that you have created in the previous step and create another folder that will contain the binaries:

```
> mkdir bin
```

Verify with the command **pwd** that your FeatFlow folder now contains two subfolders *bin* and *Feat_FloWer*. At the core of the Feat_FloWer build system is the multi-platform build files generator CMake. On the servers of the TU Dortmund CMake is available as a loadable module. In order to successfully compile the code with the GNU compiler the following components are needed:

- GCC the GNU Compiler Collection including the gfortran compiler
- A matching OpenMPI installation
- CMake with a minimum version of 2.8

On the servers of the TU Dortmund you need modules for the following software:

- A module for a gcc version (> 4.9.2)
- A module for openmpi (> 2.1.5)

- A module for cmake (> 2.8).

The modules on the TU Dortmund Servers are regularly updated and do undergo frequent name changes, so we do not reference specific module names here. If in doubt, ask a more experienced colleague or the administrator. In order for CMake to use the compiler that has been added by the module system it is recommended to set the environment variables `CC`, `CXX`, `FC` accordingly. In detail this means executing the commands (for `BASH` or `ZSH` shells):

- `> export CC=$(which mpicc)`
- `> export CXX=$(which mpicxx)`
- `> export FC=$(which mpif90)`

for the `TCSH` or `CSH` shells use the syntax:

- `> setenv CC mpicc`
- `> setenv CXX mpicxx`
- `> setenv FC mpif90`

It is advised to create shell aliases for these commands because they are used frequently. After you have entered the load commands, verify that these module are loaded by checking the output of the command **module list**. The build of the basic software package is initiated by navigating into the *bin* folder that you have created before and evoking CMake from there:

```
> cd bin
> cmake -DBUILD_METIS=True ../Feat_FloWer
```

The build process is then started by the command:

```
> make -j 5
```

The option `<-j 5>` starts a parallel build using 5 processes.

1.4.2 Windows Systems

Under construction

1.4.3 Installation of the Partitioner

By installing the partitioner we mean that it is configured in a way that a user can simply call it in a terminal, just as you would call any other terminal command. In order to achieve this, we set environment variables to the location of the necessary

files in the FeatFloWer source tree, so they can be found when called and they get updated when you fetch and apply new commits from the Git repository. The required environment variables for this to work are:

- **FF_HOME**: Source directory of your FeatFloWer installation
- **FF_PY_HOME**: Source directory of the python files of your FeatFloWer installation
- **PATH**: Append the path to the python files to your PATH variable
- **LD_LIBRARY_PATH**: Append the path to the Metis library to your LD_LIBRARY_PATH variable
- **PYTHONPATH**: Append the path /path/to/Feat_FloWer/tools directory to this variable

Instructions and examples of how to set these variables in your shell configuration can be found in Section 2.3.

1.4.4 Using CMake-gui to build the Code

The CMake build framework that is used to build the FeatFloWer software package comes with a GUI component. We will explain in this section how to use the GUI to build CMake from source. The main view of the CMake-gui shows you an overview

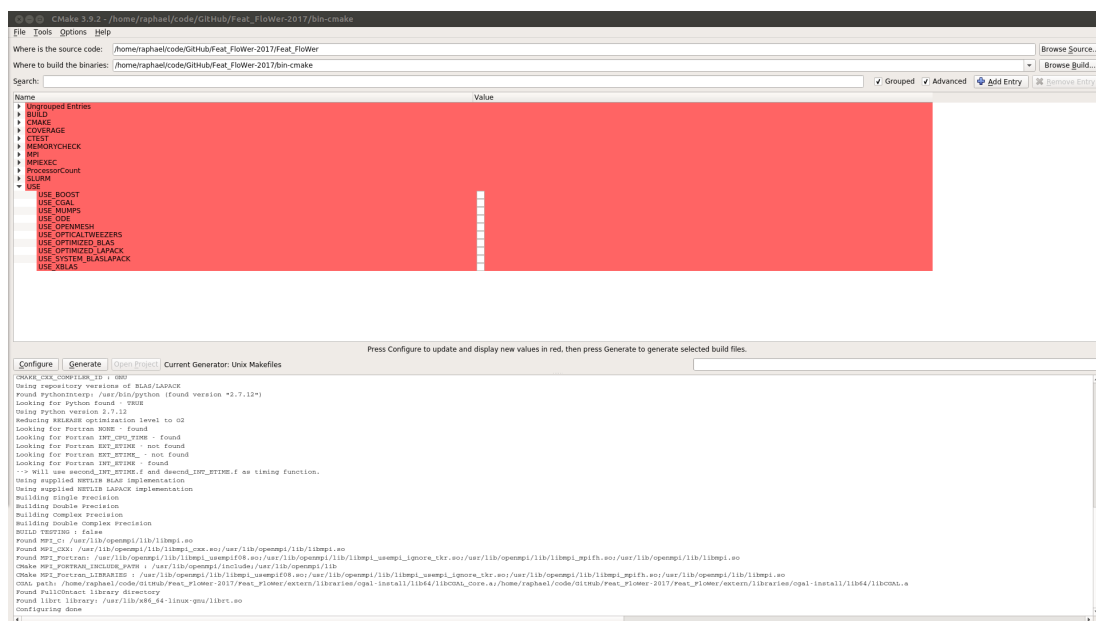


Figure 1.3: The CMake configuration GUI

of the CMake variables and their current values. Once you hit the configure button various tests are run that check for the availability of required libraries the results of these checks are printed to the output window of the CMake GUI. If a problem occurs with the current configuration an error message is printed to the output view and usually it is descriptive enough in order to find and fix the configuration problem. Once there are no more configuration errors you can hit the generate button and an operating system specific build file is generated that you can use to build the code. Additionally, the CMake-gui can give you an overview of the optional components that can be used by FeatFloWer. These optional components give the user access to extended code functionalities, solver libraries or additional simulation libraries. In the *Grouped* of CMake these optional components can be found and enabled in the *USE* group (see figure 1.3).

1.4.5 Commonly Used CMake Options Overview

In this section we list some commonly used configure options for the FeatFloWer package. A CMake option MYCMAKE_OPTION can be set from the command line using the syntax:

```
> cmake -DMYCMAKE_OPTION=VALUE [more options] /path/to/Feat_FloWer
```

- `USE_CGAL:BOOL=True/False` : Add the CGAL library, the library will be cloned from a git remote repo.
- `USE_CGAL:BOOL=True USE_CGAL_LOCAL:BOOL=True` : The combination of these options adds the CGAL library, but uses a local install of the library, in this case set also:
 - `CGAL_INCLUDE_DIR:PATH` : The path to the CGAL include directory
 - `CGAL_LIBRARIES:FILEPATH` : The path to the needed CGAL libraries, on unix-like system these are usually called `libCGAL.a` and `libCGAL_Core.a`
- `Q2P1_BUILD_ID:STRING` : Set a build-id (effectively sets compiler, compile flags, build type)

Examples

Setting CGAL libraries:

- `-DCGAL_LIBRARIES="/path/libCGAL.a;/path/libCGAL_Core.a"`

Settings CGAL include directory:

- `-DCGAL_INCLUDE_DIR="/path/cgal/include"`

Chapter 2

Running a FeatFloWer Application

2.1 Introduction to the Application Framework

A FeatFloWer application is a particular simulation case that is based on the underlying Finite-Element-Method framework. The possibility of a FeatFloWer application range from very simple CFD-Setups to well-known benchmark configurations to complex realistic configurations that have an industrial or scientific background. The FeatFloWer code base comes with a number of preconfigured applications that demonstrate the capabilities of the software. The user can then change and adapt these preconfigured applications to his needs or add completely new applications based on the FeatfloWer framework. A FeatFloWer applications roughly consists of a preprocessing step, a solving procedure and an output of the solution. At first we will introduce the tools used in the preprocessing step.

2.1.1 Preprocessing and Mesh Generation

The main task in the preprocessing step is the creation of the mesh used for the FeatFloWer application. A mesh can be acquired by an external mesh generator, by manual construction or by taking a mesh from the FeatFloWer mesh repository. Before a mesh can be used in a parallel simulation it needs to be decomposed or partitioned so that the mpi processes can handle individual parts of the domain in parallel. In FeatFloWer the Metis library is used to partition a mesh into submeshes. The functionality of the Metis library is accessed by a set of Python scripts that is distributed with FeatFloWer. We will now show how to prepare a mesh from the mesh repository for use in a parallel simulation.

2.1.1.1 Mesh Decomposition using the Python Partitioner

In order to start a simulation we need a mesh at first, taking a mesh from the mesh repository is the easiest scenario for a new user. Usually this means that a benchmark configuration or a preconfigured application is run by the user. All the user has to do in this case is to partition the mesh according to the number of MPI processes that the user has to his disposal. The preferred partitioning tool for meshes from the mesh repository is the Python command line program *PyPartitioner.py*. In order to demonstrate the use of the python partitioning tool, we will use the application *q2p1_fc_ext*. The application can be found in the *Feat_FloWer/applications* directory. The python partitioner tool can be found in the *Feat_FloWer/tools* directory. For our first contact with the tool, we will just copy the python files from the *Feat_FloWer/tools* directory to the *bin/applications/q2p1_fc_ext*. So start by navigating to the *bin/applications/q2p1_fc_ext* and copying the python files by:

```
> cp ../../../../Feat_FloWer/tools/*.py .
```

Additionally, we need to copy the metis library so that the PyPartitioner can interface with that library:

```
> cp ../../../../bin/extern/libraries/metis-4.0.3/Lib/libmetis.so .
```

We can now invoke the PyPartitioner by typing:

```
> python PyPartitioner.py 4 1 1 NEWFAC _adc/2D_FAC/2Dbench.prj
```

which will take the mesh referenced in the project file *_adc/2D_FAC/2Dbench.prj* and decompose the domain into 4 partitions which are then stored in the *NEWFAC* directory. The additional 1 1 parameters denote the type of the partitioning method. You have now created a set of partitioned mesh files that are located in the *NEWFAC* directory that can be used in a simulation with 4+1 processes. The syntax 4+1 refers to a setup with 1 master node and 4 slave nodes. You are now ready to run the application.

2.1.2 Running the Application

As we said before, the simulation requires a master node to control some the slave processes, so the simulation should be launched with 4+1 processes by:

```
> mpirun -np 5 ./q2p1_fc_ext
```

When you start a FeatFloWer application you will see the simulation output on your terminal. Additionally, the simulation output will be stored in a log or protocol file that can be found in the working directory of the particular application you are running under the path *_data/prot.txt*.

2.1.3 Application Output and Postprocessing

As we have mentioned the output of the simulation can be seen on the command line and in the protocol file. Detailed information such as the velocity or pressure fields of the solution are stored in files that are located in the `_vtk/` folder in your application working directory. The files are stored in the VTK-format so they can be opened in our preferred postprocessing software *ParaView*.

2.2 q2p1_cc (by R. Jendrny)

2.2.1 Systems that can be solved

Using this application you can solve the following systems in a coupled way:

- Stationary (Navier-) Stokes equations

$$\begin{pmatrix} L + K(u) & B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix},$$

with $K(v)w \sim v \cdot \nabla w$.

In a defect-correction procedure it is written as:

$$\begin{bmatrix} u^n \\ p^n \end{bmatrix} = \begin{bmatrix} u^{n-1} \\ p^{n-1} \end{bmatrix} + \begin{bmatrix} L + R(u^{n-1}) & B \\ B^T & 0 \end{bmatrix}^{-1} \left(\begin{bmatrix} g \\ 0 \end{bmatrix} - \begin{bmatrix} L + K(u^{n-1}) & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} u^{n-1} \\ p^{n-1} \end{bmatrix} \right)$$

with $R(u^{n-1}) = K(u^{n-1}) + \alpha \bar{M}(u^{n-1})$.

Changing the value α you can switch between Fixpoint and Newton method. If $\alpha = 0$ the pure Fixpoint method will be used. If $\alpha \neq 0$ the code uses an adaptive technique to switch between both methods.

- unsteady (Navier-) Stokes equations

- general θ -scheme

$$\begin{pmatrix} M + \theta \Delta t \{L + K(u^{n+1})\} & \Delta t B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} u^{n+1} \\ p^{n+1} \end{pmatrix} = \begin{pmatrix} M u^n - (1 - \theta) \Delta t \{L + K(u^n)\} u^n \\ 0 \end{pmatrix}$$

- Backward Difference Formula (BDF)

- BDF(1): Backward Euler method is the same.

- BDF(2):

$$\begin{pmatrix} M + \frac{2}{3} \Delta t \{L + K(u^{n+1})\} & \frac{2}{3} \Delta t B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} u^{n+1} \\ p^{n+1} \end{pmatrix} = \begin{pmatrix} \frac{4}{3} M u^n - \frac{1}{3} M u^{n-1} \\ 0 \end{pmatrix}$$

- BDF(3):

$$\begin{pmatrix} M + \frac{6}{11}\Delta t\{L + K(u^{n+1})\} & \frac{6}{11}\Delta t B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} u^{n+1} \\ p^{n+1} \end{pmatrix} = \begin{pmatrix} \frac{18}{11}Mu^n - \frac{9}{11}Mu^{n-1} + \frac{2}{11}Mu^{n-2} \\ 0 \end{pmatrix}$$

2.2.2 q2p1_param.dat

In this parameter file you have to set up all simulation parameters. Since this is a special case of the general program only the differences are listed:

- SimPar@TimeScheme: Choose between BE, CN (or FE); for stationary simulations this has to be BE.
- SimPar@TimeStep: any; for stationary this has to be 1d0.
- SimPar@MaxNumStep: Number of maximum time iterations; for stationary this has to be 1.
- SimPar@MatrixRenewal: Only M, K and S are active in this version; for Stokes K0.
- SimPar@FlowType: nonNewtonian is the only case that is working.
- SimPar@SteadyState: Choose between Yes or No; for stationary this has to be Yes and in SimPar@MatrixRenewal set M to 1.
- CCuvwp@NLmin: One possible choice is 1.
- CCuvwp@NLmax: Maximum of non-linear iteration in each time step; for Stokes this has to be 1.
- CCuvwp@Alpha: In general this is a value between 0d0 (Fixpoint) and 1d0 (Newton); for Stokes it is useful to take a big value (e.g. 11 which results in 1E-12 as the stopping)
- CCuvw@ValAdap: Insert two values for the adaptivity curve (1st greater than 1, 2nd lower than 0.9): Scaling factors for the adaptivity bewtween Fixpoint and Newton
- CCuvwp@Stopping: Relative stopping criterion for each time-step.
- CCuvwp@MGMinLev or MGMedLev: Set this as the value of SimPar@MaxMeshLevel and the linear equations are solved by MUMPS.
- CCuvwp@Vanka: Only 0 is working correctly.
- CCuvwp@BDF: Choose between 0 (for BE/CN or stationary), 2 (for BDF(2)) or 3 (for BDF(3)).

- Prop@PowerLawExp: Together with S in SimPar@MatrixRenewal the value 1.00d0 results in a Newtonian flow

2.2.3 Source files

q2p1_cc.f90

This is the main routine in which the application is configured: The application gets initialized, reads all parameters, runs the simulation and does some post-processing. You can also find the time loop in this file. The initialization for all time-schemes is added compared to the first initial version.

app_init.f90

It is called by the main routine, initializes the parallel structures, prepares the mesh (via decomposing the mesh to subdomains) and reads the simulation parameters SimPar. Here old solutions from older simulations can be read in, too.

assemblies_cc.f

Two important files are included in this file: On the one hand there is the subroutine CC_GetDefect_sub and on the other hand the subroutine CC_Extraction_sub. Both are called in the file q2p1_def_cc.f90 and will be explained in this file.

There is a subroutine which calculates the acting forces, too. But this is an old version because it is not of iso-parametric style.

iso_assemblies.f

In this file all the iso-parametric stuff is included: You can find iso-parametric assemblies of the matrices M, barM, K, S (and its further components), D (which is useless at the moment) and B. Additionally, there is the iso-parametric calculation of the forces (GetForceCyl_cc_iso). Here the acting forces are saved as follows: The first three entries are the components of the drag, the next three entries are the components of the lift and the last entry is the force in z-direction. The components of the drag and lift are sorted. First you have the complete force, then you see the velocity component and finally there is the pressure component of the acting force. The sum of the velocity and the pressure part is the complete force. If you compare the components of each force with a reference one the sum of the component errors can be greater than the error of the complete one since the triangular inequality holds.

lin_transport_cc.f90

???

postprocessing.f90

It provides some helpful subroutines. Firstly, it writes the solution to a vtk or gmV file. Secondly, the solver statistics are output at the end of the simulation. Finally, there are routines that are important for the dump files: Some files are necessary to read old dump files and some need to be called to write dump-files. At the moment you can choose between prf and dmp files: prf needs a lot of memory and dmp outputs the solution of all ids to one file for velocity or pressure. The code is adjusted to use dmp files at the moment.

q2p1_cc_Umfpacksolver.f90

Some files are needed for factorization of the coarse and element matrices. Since MUMPS is set as the coarse grid solver the solver routines of UMFPACK can be useless.

q2p1_def_cc.f90

Creating the system matrices and working with these is one of the main tasks in this file. At the start the system matrices are created: You have a big block matrix A for the non-linear equations and to compute the defect and a block matrix AA to perform the Preconditioner, i.e. $x^n = x^{n-1} + AA^{-1}(b - Ax^{n-1})$. The block matrices will only work with S (D is not working; but with S you can simulate Newtonian fluids as well.). In the bottom of this file all matrix assemblies are called to create these using the iso-parametric concept.

In this file you can find some more basic routines that are important for the CC code: One of them calculates the defect (using CC_GetDefect_sub) and the next calculates the norm of this (non-linear) defect.

Of course, there are routines which are special for the CC version. There is a subroutine which creates special CC structures. These structures are used by CC_Extraction and Special_CC_Coarse. Without these two subroutines the code uses wrong matrices in the solver and you will get wrong results.

Ultimately the MG solver is initialized and called in CC_mgSolve.

q2p1_mg_cc.f90

??

q2p1_transport_cc.f90

Before the main working subroutine is implemented some structures for the CC solver are initialized.

The main working/solving routine is Transport_q2p1_UxyzP_cc. The following sketch will show how this routine is working:

- Generate the adaptive function to have a background how to switch between Fixpoint and Newton method.
- Assemble the right hand side.
- Regarding time schemes set the correct scaling factor.
- Assemble the system matrices
- Factorize the element matrices.
- Compute the initial defect.
- Call the MG solver.
- Perform a full update of the solution
- Depending on the change of the defects the code decides if it can endure more Newton influence or less. The scaling is influenced by the adaptive function.
- Depending on the change of the defects and the use of Newton or Fixpoint the stopping criterion for MG is changed.
- Calculate the acting forces.
- Does the solution fulfill the non-linear stopping criterion?

In this file you can also find two subroutines which calculate the acting forces: On the one hand FAC_GetForces_CC_iso that is calling GetForceCyl_cc_iso and is the default decision; on the other hand there is a subroutine that calculates the forces using $Su + Bp$ (myFAC_GetForces).

The last important subroutine reads the CCuvwp parameters which are set in the param file.

q2p1_var_newton.f90

Additionally to QuadSc_var.f90 you can find a few new variables:

- Variables for the Newton-Preconditioner: barMXYmat, mg_barMXYmat, AAXYmat and mg_AAXYmat,

- variables for the force calculation using myFAC_GetForces: BXMat_new, mg_BXMat_new (Since the general B matrices are influenced by the Dirichlet BC there was a need to have B matrices without these BC to have correct forces.),
- a new type for the CC parameters used in the param.dat: TYPE tParamCC and
- additional variables for the BDF schemes

2.2.4 How to compile and run the code

The following comments compile the code and run it:

```
> cmake -DQ2P1_BUILD_ID=xeon-linux-intel-release -DBUILD_APPLICATIONS=True -DUSE_MUMPS=True ../Feat_FloWer/
> make -j4
> PyPartitioner.py 8 1 1 NEWFAC 'FILE.prj'
> mpirun -np 9 ./q2p1_cc
```

2.3 Appendix

TODO

2.3.1 Recommended Shell Customizations For Linux Systems

In order to work more efficiently with the FeatFloWer software, we recommended to add some environment variables and aliases to your shell configuration files in order to make your experience with the software more pleasant. Sample configuration files are provided in the FeatFloWer repository you have cloned in the first steps. The configuration files can be found in the repository under the following path:

```
> /path/to/Feat_FloWer/tools/shell_scripts/featflower_scripts
```

There sample configuration for different shells can be found. As can be seen in the following listings, the configuration files contain aliases for loading compiler settings, customizations for the command line for the Git version control system and setting environment variables so that the different FeatFloWer auxilliary scripts are contained in the path and can be easily called from the command line.

TCSH and CSH

	TCSH
1	<code>#!/bin/csh</code>
2	
3	<code>set prompt = "%B%m%b:%~%# "</code>
4	

```

5  #=====
6  #           Environment variables for Feat_FloWer
7  #=====
8  # Adjust these paths to point them to the directories of
9  # your local installation.
10 setenv Q2P1_MESH_DIR /path/to/mesh_repo
11 setenv FF_HOME /path/to/Feat_FloWer
12 setenv FF_PY_HOME /path/to/Feat_FloWer/tools
13
14 #=====
15 #           GCC Aliases
16 #=====
17 alias load-gcc-mpi51 'module purge ;; \
18 module load cmake/3.5.2 \
19 gcc/5.1.0 \
20 openmpi/gcc5.1.x/1.10.2/threaded/no-cuda/ethernet \
21 boost/1.56 python/2.7.11 ;; \
22 setenv CC mpicc ;; setenv CXX mpicxx ;; setenv FC mpif90'
23
24 alias load-gcc-mpi61 'module purge ;; \
25 module load cmake/3.4.0 \
26 gcc/6.1.0 \
27 openmpi/gcc6.1.x/1.10.2/non-threaded/no-cuda/ethernet \
28 boost/1.56 python/2.7.11 ;; \
29 setenv CC mpicc ;; setenv CXX mpicxx ;; setenv FC mpif90'
30
31 #=====
32 #           Intel Compiler Aliases
33 #=====
34 alias load-intel-mpi16 'module purge ;; \
35 module load cmake/3.5.2-ssl \
36 intel/studio-xe/16.0.4.258 \
37 openmpi/intel16.0.x/1.8.8/non-threaded/no-cuda/ethernet \
38 gcc/6.1.0 boost/1.56 python/2.7.11 ;; \
39 setenv CC mpicc ;; \
40 setenv CXX mpicxx ;; \
41 setenv FC mpif90'
42
43
44 #=====

```

```

45 #                               Library Path for Metis
46 #=====
47 setenv LD_LIBRARY_PATH \
48     $LD_LIBRARY_PATH:/path/to/metis-4.0.3/Lib
49
50
51 #=====
52 #                               Customization for Git Version Control
53 #=====
54 source ~/autocompletion.tcsh
55
56 #git completion
57 source ~/.git-completion.tcsh
58
59 # set prompt; with current git branch if available.
60 alias __git_current_branch \
61
62 'git rev-parse --abbrev-ref HEAD >& /dev/null && \
63 echo "{`git rev-parse --abbrev-ref HEAD`}"'
64
65 alias precmd \
66 'set prompt="%n@m[%c2] `__git_current_branch` "'
67
68 #=====
69 #                               PATH Variable
70 #=====
71 # Do further customization of your PATH here:
72 setenv PATH $PATH:/path/to/Feat_FloWer/tools/partitioner
73
74 module purge ;; module load gcc/6.1.0 \
75 openmpi/gcc6.1.x/1.10.2/non-threaded/no-cuda/ethernet \
76 cmake
77
78 setenv CC mpicc
79 setenv CXX mpicxx
80 setenv FC mpif90

```

BASH and ZSH**BASH**

```

1  #!/bin/bash
2  #=====
3  #Author: Raphael Muenster
4  #This script automizes module loading
5  #and console environment configuration
6  #=====
7
8
9  # Uncomment this line and assign a value to the EDITOR
10 # variable to set your default editor.
11 #export EDITOR=
12
13 #=====
14 #           Environment variables for Feat_FloWer
15 #=====
16 # Adjust these paths to point them to the directories
17 # of your local installation.
18
19 export Q2P1_MESH_DIR=/path/to/mesh_repo
20 export FF_HOME= /path/to/Feat_FloWer
21 export FF_PY_HOME=/path/to/Feat_FloWer/tools
22 export PATH=$PATH:/path/to/Feat_FloWer/tools/partitioner
23
24 #=====
25 #           Cluster Logins
26 #=====
27 alias ssh-lido='ssh username@lidong1.itmc.tu-dortmund.de\
28 -L 11111:lidong1.itmc.tu-dortmund.de:11111'
29 alias ssh-lido3='ssh username@gw02.lido.tu-dortmund.de\
30 -L 11111:gw02.lido.tu-dortmund.de:11111'
31
32 #=====
33 #           TMUX Configuration
34 #=====
35 # Set the screen variable of the tmux terminal multiplexer
36 alias tmux="TERM=screen-256color-bce tmux"
37
38 #=====
39 #           Intel Compiler Aliases

```

```

40 #=====
41 alias load-intel-mpi15='module purge && module load \
42 intel/studio-xe/15.0.7.235 gcc/5.4.0 boost/1.56 \
43 openmpi/intel15.0.x/1.10.2/non-threaded/no-cuda/ethernet \
44 cmake && export CC=mpicc && export CXX=mpicxx && \
45 export FC=mpif90'
46 alias load-intel-mpi16='module purge && module load \
47 intel/studio-xe/16.0.4.258 binutils/2.25 \
48 gcc/6.1.0 \
49 openmpi/intel16.0.x/1.8.8/non-threaded/no-cuda/ethernet \
50 cmake && export CC=mpicc && export CXX=mpicxx && \
51 export FC=mpif90'
52
53 #=====
54 #                               GCC Aliases
55 #=====
56 alias load-gcc-mpi61='module purge && module load cmake \
57 gcc/6.1.0 boost/1.56 \
58 openmpi/gcc6.1.x/1.10.2/non-threaded/no-cuda/ethernet \
59 && export CC=$(which mpicc) && \
60 export CXX=$(which mpicxx) && export FC=$(which mpif90)'
61
62 #=====
63 #                               PATH Variable
64 #=====
65 # Do further customization of your PATH here:
66 #export PATH=$PATH:$HOME/nobackup/bin
67
68 #=====
69 #                               Library Path for Metis
70 #=====
71 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: \
72 /path/to/metis-4.0.3/Lib
73
74 module purge && module load gcc/6.1.0 && \
75 module load \
76 openmpi/gcc6.1.x/1.10.2/non-threaded/no-cuda/ethernet \
77 && module load cmake && export CC=mpicc && \
78 export CXX=mpicxx && export FC=mpif90
79

```