FLEXI Project Documentation

Numerics Research Group
Institute for Aerodynamics and Gas Dynamics
University of Stuttgart, Germany

March 2, 2023

Contents

1 Introduction **6**
1.1 How to use the user guide . . . 6

2 Getting started **8**
2.1 Installation . . . 8
2.1.1 Prerequisites . . . 8
2.1.2 Obtaining the source . . . 9
2.1.3 Compiling the code . . . 10
2.1.4 Running the code . . . 11
2.2 Basic Usage . . . 11
HOPR . . . 12
FLEXI . . . 12
Parameter file . . . 13
posti_visu tool . . . 13
HDF5 . . . 14
2.3 Feature list . . . 14

3 Code options **16**
3.1 Compiler options . . . 16
3.2 Parameter file options . . . 20
3.3 Parameterfile . . . 20

4 Workflow **31**
4.1 Mesh generation using HOPR . . . 31
4.2 Solver settings . . . 31
4.3 Initial and boundary conditions . . . 33
4.3.1 Initial conditions . . . 33
4.3.2 Boundary conditions . . . 33
4.4 Material properties . . . 34
4.5 Output time interval . . . 35
Restart the simulation . . . 35
4.6 Evaluation during runtime . . . 35
4.7 Parallel execution . . . 35
4.7.1 Domain decomposition . . . 36
4.7.2 Choosing the number of cores . . . 36
4.8 Test case environment . . . 36

4.9 Post processing / posti_visu tool . . . 37

5 Tools Overview **39**
5.1 POSTI . . . 39
5.1.1 Visualization . . . 39
5.1.2 Swap meshes . . . 40
5.1.3 Record points . . . 40
5.1.4 Time averaging . . . 41
5.1.5 Test cases . . . 41
5.2 `tools` folder . . . 43
5.2.1 `animate` . . . 43
5.2.2 `convergence_test` . . . 44
5.2.3 `userblock` . . . 44
5.2.4 `sortfiles.sh` . . . 45
5.2.5 `getload.py` . . . 45
5.2.6 `testcases` . . . 45

6 Tutorials **46**
6.1 Freestream . . . 46
6.1.1 Mesh Generation with HOPR . . . 46
6.1.2 Flow Simulation with FLEXI . . . 48
6.1.3 Numerical settings . . . 49
6.2 Lid-driven Cavity . . . 51
6.2.1 Flow description . . . 51
6.2.2 Compiler options . . . 51
6.2.3 Part I: Basic Tutorial - Flow at $Re = 100$ . . . 53
6.2.4 Part II: Advanced Tutorial - Flow at $Re = 400$ . . . 66
6.3 Flow around a NACA 0012 airfoil . . . 73
6.3.1 Compiler options . . . 76
6.3.2 Mesh Generation with HOPR . . . 76
6.3.3 Flow Simulation with FLEXI . . . 76
6.3.4 Numerical settings . . . 77
6.3.5 Boundary conditions . . . 78
6.3.6 Running the code . . . 79
6.3.7 Evaluation of the lift and drag forces . . . 79
6.3.8 Wall velocities . . . 81
6.3.9 Visualization . . . 81
6.3.10 Remove outflow reflections using the sponge zone . . . 83
6.3.11 Restarting the simulation . . . 85
6.3.12 Two-dimensional computation . . . 85
6.3.13 Using record points . . . 89
6.4 Convergence Test . . . 92
6.4.1 Manufactured Solution . . . 93
6.4.2 Inviscid Convergence Test . . . 93

6.4.3 Viscous Convergence Test . . . 99
6.4.4 Execution of the Convergence Test with non-conform meshes . . . 100
6.4.5 Execution of Convergence Test Outside of Tutorials . . . 100
6.5 Taylor Green Vortex . . . 100
6.5.1 Flow description . . . 101
6.5.2 Compiler options . . . 101
6.5.3 Tutorial - Flow at $Re = 1600$ . . . 102
6.6 SOD Shock tube . . . 107
6.6.1 Mesh Generation with HOPR . . . 107
6.6.2 Flow Simulation with FLEXI . . . 108
6.7 Double Mach Reflection . . . 109
6.7.1 Mesh Generation with HOPR . . . 111
6.7.2 Flow Simulation with FLEXI . . . 111
6.8 Linear Scalar Advection-Diffusion Equation . . . 114
6.8.1 Theoretical Background . . . 115
6.8.2 Compiling FLEXI with LinAdvDiff . . . 117
6.8.3 Setup . . . 118
6.8.4 Simulation and results . . . 120
6.9 Plane Turbulent Channel Flow . . . 124
6.9.1 Flow description . . . 124
6.9.2 Compiler options . . . 124
6.9.3 Tutorial - Flow at $Re_\tau = 180$ . . . 125
6.9.4 Performance improvements . . . 128
6.10 Flow around a cylinder . . . 129
6.10.1 Compiler options . . . 129
6.10.2 Mesh Generation with HOPR . . . 131
6.10.3 Flow Simulation with FLEXI . . . 131
6.10.4 Numerical settings . . . 131
6.10.5 Boundary conditions . . . 132
6.10.6 Running the code . . . 132
6.10.7 Evaluation of Strouhal number . . . 132
6.10.8 Evaluation of the separation angle . . . 133
6.10.9 Dynamic Mode Decomposition . . . 134
6.11 Flow over a flat plate with RANS equations . . . 136
6.11.1 Compiler options . . . 137
6.11.2 Flow Simulation with FLEXI . . . 138
6.11.3 Numerical settings . . . 138
6.11.4 Mesh and boundary conditions . . . 139
6.11.5 Wall distance . . . 139
6.11.6 Running the code . . . 140
6.11.7 Implicit time discretization . . . 140
6.11.8 Runing the simulation with implicit time discretization . . . 143

7 Unit tests **145**
7.1 Integration of unit test with CTest 145
7.2 Implementation of unit tests 146
7.2.1 CMakeLists.txt 146
7.2.2 General unit test structure 146
7.2.3 Generation of reference mesh data 147

8 Installation guidelines **148**
8.1 Cloning and compiling on machines at the HLRS 148
8.1.1 HTTPS 148

References **150**

1 Introduction

FLEXI is a high-order discontinuous Galerkin (DG) simulation code for the solution of the time-dependent compressible Navier-Stokes equations on unstructured hexahedral elements in three space dimensions. The code was specifically designed for very high order accurate simulations on massively parallel systems. It is licensed under GPLv3, written in Fortran and parallelized with MPI. Implemented features are

- Arbitrary order nodal polynomial tensor product basis using Gauss or Gauss Lobatto collocation points
- Various Riemann solvers for inter-element coupling
- Large eddy simulation capabilities through different de-aliasing strategies and subgrid scale models
- Split form DG formulation with various kinetic energy or entropy preserving flux formulations
- Matching high order curved mesh generation from external mesh formats (CGNS, GMSH) or simple analytic blocks via the open source preprocessor **HOPR**
- Nonconforming interfaces based on the mortar approach
- Non-reflecting boundary conditions and damping zones for direct aeroacoustic computations
- Automatic domain decomposition for parallel simulations based on a space filling curve
- High order explicit Runge-Kutta time integration
- I/O using the HDF5 library optimized for massively parallel jobs
- Shock capturing employing finite volume subcells

1.1 How to use the user guide

This user guide is organized to both guide the first steps as well as provide a complete overview of the simulation code's features from a user and a developer point of view.

- Chapter 2 contains step by step instructions from obtaining the source code up to running a first simulation and visualizing the simulation results. In addition, it provides an overview of the whole simulation framework and the currently implemented features.
- Chapter 3 lists the most important compiler and runtime options.

- Chapter 4 is meant as a complete user guide with a detailed description how to use and apply the features of **FLEXI** from a user's point of view. This includes setting up solver settings, initial and boundary conditions, the mesh interface, parallel execution and the currently available post processing capabilities.
- Chapter 5 lists all tools contained in the **FLEXI** repository, including **POSTI** post-processing tools.
- Simulation tutorials are contained in Chapter 6.
- The unit test system used to test key routines with CTest is described in Chapter 7.

2 Getting started

2.1 Installation

2.1.1 Prerequisites

FLEXI has been tested for various Linux distributions. This includes Ubuntu 14.04 LTS, 16.04 LTS and 18.04 LTS, OpenSUSE 42.1 and CentOS 7. The suggested packages in this section can of course be replaced by self compiled versions.

The required packages for the Ubuntu Linux distributions are listed in table 2.1. Under Ubuntu, they can be obtained using the apt environment:

```
sudo apt-get install git
```

Table 2.1: Debian/Ubuntu packages. x: required, o: optional, -: not available

Package	Ubuntu 14.04	Ubuntu 16.04	Ubuntu 18.04
git	x	x	x
cmake	x	x	x
cmake-curses-gui	o	o	o
liblapack3	x	x	x
liblapack-dev	x	x	x
gfortran	x	x	x
g++	x	x	x
mpi-default-dev	x	x	x
zlib1g-dev	-	x	x
exuberant-ctags	o	o	o

The required packages for OpenSUSE and CentOS are listed in table 2.2.

Under OpenSUSE, packages are installed by the following command.

```
sudo zypper install git
```

The PATH variable must be extended by the openmpi path

```
export PATH=$PATH:/usr/lib64/mpi/gcc/openmpi/bin
```

Under CentOS, packages are installed by the following command.

```
sudo yum install git
```

Additionally, the PATH variable must be extended by the openmpi path

```
export PATH=$PATH:/usr/lib64/openmpi/bin
```

Table 2.2: OpenSUSE/CentOS packages. x: required, o: optional, -: not available

Package	OpenSUSE 42.1	CentOS 7
git	x	x
cmake	x	x
lapack-devel	x	x
openmpi	x	x
openmpi-devel	x	x
zlib-devel	x	x
gcc-fortran	x	x
gcc	x	-
gcc-c++	x	x
ctags-etags	-	o

Finally, for certain features of the post-processing tools, we need to install the FFTW library, an open-source library for computing the discrete Fourier transform (DFT) in one or more dimensions. Download the source code of the latest release from http://www.fftw.org/download.html and follow the installation instructions there. Make sure to enable the MPI-flag when configuring, such that the installation process usually boils down to

```
./configure --enable-mpi
make
make install
```

On some systems it may be necessary to increase the size of the stack (part of the memory used to store information about active subroutines) in order to execute **FLEXI** correctly. This is done using the command

```
ulimit -s unlimited
```

from the command line. For convenience, you can add this line to your `.bashrc`.

2.1.2 Obtaining the source

The **FLEXI** repository is available at GitHub. To obtain the most recent version you have two possibilities:

- Clone the **FLEXI** repository from Github

```
git clone https://github.com/flexi-framework/flexi.git
```

- Download **FLEXI** from Github:

```
wget https://github.com/flexi-framework/flexi/archive/master
 .tar.gz
tar xzf master.tar.gz
```

Note that cloning **FLEXI** from GitHub may not be possible on some machines, as e.g. the HLRS at the University of Stuttgart restricts internet access. Please refer to section 8.1 of this user guide.

2.1.3 Compiling the code

- Open a terminal
- Change into the **FLEXI** directory
- Create a new subdirectory and use CMake to configure and compile the code

```
mkdir build; cd build
cmake ../
make
```

The executables **flexi** and **posti_visu** are contained in your **FLEXI** directory in `build/bin/`.

Custom configuration of compiler options may be done using

```
ccmake ../
```

For a list of all compiler options see Section 3.1.

2.1.3.1 Directory paths

In the following, we write `$FLEXIROOT` as a substitute for the path to the **FLEXI** repository. Please replace `$FLEXIROOT` in all following commands with the path to your **FLEXI** repository *or* add an environment variable `$FLEXIROOT`.

Furthermore, the path to executables is omitted in the following, so for example, we write `flexi` instead of `$FLEXIROOT/build/bin/flexi`.

Here is some explanation for Linux beginners:

In order to execute a file, you have to enter the full path to it in the terminal. There are two different ways to enable typing `flexi` instead of the whole path (do not use both at the same time!)

1. You can add an alias for the path to your executable. Add a command of the form

```
alias flexi='$FLEXIROOT/build/bin/flexi'
```

to the bottom of the file `~/.bashrc`. Source your `~/.bashrc` afterwards with

```
. ~/.bashrc
```

2. You can add the **FLEXI** binary directory to your `$PATH` environment variable by adding

```
export PATH=$PATH:$FLEXIROOT/build/bin
```

to the bottom of the file `~/.bashrc` and sourcing your `~/.bashrc` afterwards.

2.1.4 Running the code

For a first minimal **FLEXI** run, do the following:

- Open a terminal
- Navigate to a directory, in this case *temp*

```
cd temp
```

- Copy the *cavity* tutorial folder

```
cp -r $FLEXIROOT/tutorials/cavity/Basic_Re100 .
cd Basic_Re100
```

- Run flexi

```
flexi parameter_flexi.ini
```

- Convert the output files to the vtu format

```
posti_visu cavity_State_0000000.200000000.h5
```

- Visualize using e.g. ParaView.

2.2 Basic Usage

For a basic overview of the framework and the single components of the flow solver a flowchart is given in Figure 2.1.

Figure 2.1: Flowchart: Basic modules and files used by **FLEXI**

HOPR

A standalone high-order preprocessor **HOPR** has been developed to generate high-order meshes from input data from external linear mesh generators. Different file formats are supported. HOPR has been recently made open source under the GPLv3 license. It generates a **FLEXI** conform mesh format in HDF5 for efficient parallel initialization. For a complete overview of HOPR, see https://www.hopr-project.org.

HOPR can be compiled in the same way as **FLEXI** (see section 2.1.3). The basic command to run HOPR is

```
hopr parameter.ini
```

Note that the path to the **HOPR** executable is omitted in the command (see 2.1.3).

FLEXI

FLEXI, a high order DGSEM based CFD solver, is the core module in the tool chain. Generally **FLEXI** requires two main files as input, a mesh file in HDF5 format generated by **HOPR** and a parameter file where the main settings for the CFD simulation are set. The results files generated by **FLEXI** are also HDF5 files.

The basic command to run **FLEXI** is

```
mpirun -np [no. processors] flexi parameter.ini
```

Note: Adding `mpirun -np [no. processors]` before the **FLEXI** executable starts **FLEXI** in parallel with the specified number of threads. If it is omitted, **FLEXI** is run on one processor without MPI. This also applies to all other tools mentioned below.

Parameter file

The `parameter.ini` file contains the main settings for the CFD simulation. The parameter file defines e.g.

- CFL (Courant-Friedrichs-Lewy) number
- polynomial degree,
- simulation end time and dump/analyze intervals
- boundary conditions
- initial and boundary states

A complete list of all runtime options that can be set in the parameter file can be obtained with the command

```
flexi --help
```

It is also supplied in Section 3.2. The `--help` option also works for most other **FLEXI** tools.

posti_visu tool

To visualize the results e.g. with ParaView, a converter tool is provided. The posti_visu tool takes the HDF5 files generated by **FLEXI**.

The basic command to run the posti_visu tool is

```
mpirun -np [no. processors] posti_visu parameter.ini [
   flexi_outputfile.h5]
```

In this case a parameter file is specified in which options like the type and amount of the visualization nodes and mesh options are defined - see Section 4.9 for all available options. You can also omit the parameter file argument:

```
mpirun -np [no. processors] posti_visu [flexi_outputfile.h5]
```

This runs posti_visu using only standard options, i.e.

- equidistant visualization nodes
- amount of visualizaion nodes equals number of collocation points per element
- allowing for curved meshes
- visualizing the conservative variables

HDF5

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. For further information and to download the software, visit the HDF5 website at https://www.hdfgroup.org.

2.3 Feature list

The currently implemented features of **FLEXI** include

- Equation systems:
  - compressible Euler equations
  - compressible Navier-Stokes equations
  - linear scalar advection and diffusion
- Space discretization: DGSEM method (Kopriva and Gassner 2010; Hindenlang et al. 2012)
  - Legendre Gauss
  - Legendre Gauss Lobatto
- Time discretization: explicit Runge-Kutta methods
  - standard RK methods
  - low storage RK methods (Carpenter and Kennedy 1994)
  - strong stability preserving RK methods (Niegemann, Diehl, and Busch 2012)
- Two- or three-dimensional domains
- Riemann solvers:
  - local Lax-Friedrichs
  - HLL
  - HLLC
  - Roe-Pike
- Curved Meshes
- Nonconforming Meshes via mortar interfaces (Kopriva, Woodruff, and Hussaini 2002)
- Shock capturing
  - Employing finite volume subcells by either
    - switching to finite volume subcells (Sonntag and Munz 2017)
    - blending the finite volume operator (Hennemann et al. 2021)
  - Several shock indicators available
- Boundary conditions

  - Various subsonic inflow and outflow conditions (Carlson 2011)
  - exact boundaries (Dirichlet)
  - periodic boundaries
  - slip wall (Euler wall)
  - non-slip walls (Navier-Stokes wall)
    * adiabatic
    * isothermal

- Splitform discontinuous Galerkin schemes (Gassner, Winters, and Kopriva 2016)
- Dealiasing (Gassner and Beck 2013)
  - filtering
  - overintegration
- Lifting methods
  - Bassi Rebay 1 (Bassi and Rebay 1997)
  - Bassi Rebay 2 (Bassi and Rebay 1997)
- Sponge zone (Flad et al. 2014)
- Time averaging

3 Code options

3.1 Compiler options

This section describes the main configuration options which can be set when building **FLEXI** using CMake. Some options are dependent on others being enabled (or disabled), so the available ones may change.

The first set of options describe general CMake behaviour:

- `CMAKE_BUILD_TYPE`:

  This statically specifies what build type (configuration) will be built in this build tree. Possible values are

  - **Release**

    "Normal" execution.

  - **Profile**

    Performance profiling using gprof.

  - **Debug**

    Debug compiler for detailed error messages during code development.

- `CMAKE_HOSTNAME`:

  This will display the host name of the machine you are compiling on.

- `CMAKE_INSTALL_PREFIX`:

  If "make install" is invoked or INSTALL is built, this directory is prepended onto all install directories. This variable defaults to /usr/local on UNIX.

For some external libraries and programs that **FLEXI** uses, the following options apply:

- `CTAGS_PATH`:

  This variable specifies the Ctags install directory, an optional program used to jump between tags in the source file.

- `LIBS_BUILD_HDF5`: ON/OFF

  This will be set to ON if no prebuilt HDF5 installation was found on your machine. In this case a HDF5 version will be build and used instead.

- `HDF5_DIR`:

  If you want to use a prebuilt HDF5 library that has been build using the CMake system, this directory should contain the CMake configuration file for HDF5 (optional).

The following options enable or disable specific features of **FLEXI**. If you want to use a certain feature, make sure to enable it during the build process!

- `FLEXI_2D`: ON/OFF

  If set to ON the code will run in two-dimensional mode. You have to provide a mesh that consists of only one layer of elements in the third dimension.

- `FLEXI_EQNSYSNAME`:

  This variable defines the equation system, which will be compiled and used for the simulation. Possible values are

  - *Navierstokes*
  - *Linearscalaradvection*

- `FLEXI_FV`:

  Set this to enable the corresponding finite volume subcell shock capturing mechanism. Implemented are

  - OFF : No finite volume shock capturing
  - SWITCH : Switching DG elements into a finite volume subcell representation (Sonntag and Munz 2017)
  - BLEND : Blending the finite volume discretization operator to the DG operator. (Hennemann et al. 2021)

- `FLEXI_FV_RECONSTRUCTION`: ON/OFF

  Only available if FLEXI_FV is set either to SWITCH or BLEND. Enables the reconstruction of interface values in the finite volume subcells. Needed for calculation of gradients and to use a second order finite volume scheme.

- `FLEXI_LIFTING`:

  Two different lifting methods for the parabolic part of the equation system are implemented. Possible values are

  - **BR1**: First method of Bassi and Rebay (Bassi and Rebay 1997)
  - **BR2**: Second method of Bassi and Rebay (Bassi et al. 1997)

- `LIBS_USE_MKL`: ON/OFF

  This flag defines, whether Intel's MKL (Math Kernel Library) should be used. This is only meaningful when **FLEXI** is compiled with Intel compiler.

- `LIBS_USE_MPI`: ON/OFF

  This flag defines, whether **FLEXI** is compiled with MPI (necessary for parallel execution).

- `FLEXI_NODETYPE`:

  FLEXI space discretization is based on a DG method. Here two different basis functions could be used, see (Kopriva and Gassner 2010) for details.

  - GAUSS
  - GAUSS-LOBATTO

- `LIBS_USE_PAPI`: ON/OFF

  Enable to use the PAPI library to perform performance measurements (e.g. flop counts).

- `FLEXI_PARABOLIC`: ON/OFF

  This variable defines, whether the parabolic part of the chosen system should be included or not. Practically, this corresponds to setting the viscosity or diffusivity coefficient to zero, but is much more efficient, since the lifting routines are not necessary.

- `FLEXI_POLYNOMIAL_DEGREE`:

  Since **FLEXI** is a high order CFD solver based on polynomial basis function, the polynomial degree can already be chosen in the compile process. If the default value N is chosen, different polynomial degrees can be defined later in the parameter file.

  - N
  - #

- `FLEXI_SPLIT_DG`: ON/OFF

  Enable to use the split form of the discontinuous Galerkin operator. Allows to use kinetic energy or entropy stable flux functions. Only available for Gauss-Lobatto nodes.

- `FLEXI_TESTCASE`:

  Some specific (and often used) simulation setups are encapsulated in test cases. These include e.g. case-specific initialization, analyze routines, boundary conditions etc. The default test case does not include any additions.

  - default
  - taylorgreenvortex
  - phill
  - channel

  - riemann2d

  See section 4.8 for details.

- `FLEXI_VISCOSITY`:

  There are different modeling approaches for the viscosity in **FLEXI**. You can choose

  - constant
  - sutherland
  - powerlaw

The remaining part of the options deal with the post-processing framework **POSTI**.

- `POSTI`: ON/OFF

  Enable to also build the post-processing tools next to the actual simulation software. When this general option is enabled, you will need to enable the specific options for the tools that should be build.

- `POSTI_*`: ON/OFF

  Each of the **POSTI** tools has it's own build option. Enable to build this specific tool.

- `POSTI_VISU_PARAVIEW*`: ON/OFF

  Enable to build the ParaView plugin for visualization of **FLEXI** simulation data. The ParaView libraries etc. must be available on the system and the environment variable $ParaView_DIR set accordingly.

There are some additional options for performance improvements for FLEXI.

- `FLEXI_PERFORMANCE`: ON/OFF

  Enables a set of advanced features to improve the performance of FLEXI.

- `FLEXI_PERFORMANCE_OPTLIFT`: ON/OFF

  Enable to lift only the gradients of the variables that are actually required to evaluate the fluxes of the corresponding equation system. Improves the performance if the parabolic terms are computed, but cannot be used if POSTI is enabled.

- `FLEXI_PERFORMANCE_PGO`: ON/OFF

  Enables profile-guided optimization (PGO) for compilation. Currently, only the GNU compiler is supported to use PGO with FLEXI. The required two-step compilation process is detailed in Section 6.9.4.

3.2 Parameter file options

A `parameter.ini` file is needed to control the code. An overview of all options in the parameter file can be generated by following command in the terminal:

```
flexi --help
```

Generally following types are used:

```
INTEGER = 1
REAL    = 1.23456
LOGICAL = T          ! True
LOGICAL = F          ! False
STRING  = FLEXI
VECTOR  = (/1.0,2.0,3.0/)
```

The concept of the parameter file is described as followed:

- each single line is saved and examined for specific variable names
- the examination is case-insensitive
- comments can be set with symbol "!" in front of the text

```
! commented text
```

- numbers can also be set by using "pi"

```
vector = (/1,2Pi,3Pi/)
```

- the order of defined variables is with one exception generally indifferent, but it is preferable to group similar variables
- the order becomes important only by modifying boundary conditions, if you want to modify a specific boundary by addressing its name, the related boundary type has to be defined

```
BoundaryName=inflow          ! BC_Name defined in mesh file
BoundaryType=(/2,0,0,0/)
BoundaryName=outflow         ! BC_Name defined in mesh file
BoundaryType=(/2,0,0,0/)
```

The following tables describe the main configuration options which can used in the parameter file.

3.3 Parameterfile

MPI

Variable	Default	Description
GroupSize	0	Define size of MPI subgroups, used to e.g. perform grouped IO, where group master collects and outputs data.

IO_HDF5

Variable	Default	Description
gatheredWrite	F	Set true to activate gathered HDF5 IO for parallel computations. Only local group masters will write data after gathering from local slaves.

Interpolation

Variable	Default	Description
N		Polynomial degree of computation to represent to solution

Restart

Variable	Default	Description
ResetTime	F	Override solution time to t=0 on restart.
NFVRestartSuper		Polynomial degree for equidistant supersampling of FV subcells when restartingon a different polynomial degree. Default 2*MAX(N,NRestart).

Output

Variable	Default	Description
NVisu		Polynomial degree at which solution is sampled for visualization.
NOut	-1	Polynomial degree at which solution is written. -1: NOut=N, >0: NOut
ProjectName		Name of the current simulation (mandatory).
Logging	F	Write log files containing debug output.
ErrorFiles	T	Write error files containing error output.

Output		
OutputFormat	None	File format for visualization: None, Tecplot, TecplotASCII, ParaView. Note: Tecplot output is currently unavailable due to licensing issues.
ASCIIOutputFormat	CSV	File format for ASCII files, e.g. body forces: CSV, Tecplot.
doPrintStatusLine	F	Print: percentage of time, ...
WriteStateFiles	T	Write HDF5 state files. Disable this only for debugging issues. NO SOLUTION WILL BE WRITTEN!
WriteTimeAvgFiles	T	Write HDF5 time average files. Disable this only for debugging. NO TIME AVERAGE FILES WILL BE WRITTEN!

Mesh		
Variable	**Default**	**Description**
MeshFile		(relative) path to meshfile (mandatory).
useCurveds	T	Controls usage of high-order information in mesh. Turn off to discard high-order data and treat curved meshes as linear meshes.
interpolateFromTree	T	For non-conforming meshes, built by refinement from a tree structure, the metrics can be built from the tree geometry if it is contained in the mesh. Can improve free-stream preservation.
meshScale	1.0	Scale the mesh by this factor (shrink/enlarge).
meshdeform	F	Apply simple sine-shaped deformation on cartesion mesh (for testing).
crossProductMetrics	F	Compute mesh metrics using cross product form. Caution: in this case free-stream preservation is only guaranteed for N=3*NGeo.
debugmesh	0	Output file with visualization and debug information for the mesh. 0: no visualization, 3: Paraview binary
BoundaryName		Names of boundary conditions to be set (must be present in the mesh!).For each BoundaryName a BoundaryType needs to be specified.

Mesh		
BoundaryType		Type of boundary conditions to be set. Format: (BC_TYPE,BC_STATE)
writePartitionInfo	F	Write information about MPI partitions into a file.
NGeoOverride	-1	Override switch for NGeo. Interpolate mesh to different NGeo.<1: off, >0: Interpolate

Equation of State		
Variable	**Default**	**Description**
UseNonDimensionalEqn	F	Set true to compute R and mu from bulk Mach Reynolds (nondimensional form.
BulkMach		Bulk Mach (UseNonDimensionEqn=T)
BulkReynolds		Bulk Reynolds (UseNonDimensionEqn=T)
kappa	1.4	Heat capacity ratio / isentropic exponent
R	287.058	Specific gas constant
Pr	0.72	Prandtl number
mu0	0.0	Dynamic Viscosity
Ts	110.4	Sutherland's law for variable viscosity: Ts
Tref	273.15	Sutherland's law for variable viscosity: Tref
ExpoSuth	1.5	Sutherland's law for variable viscosity: Exponent

Equation		
Variable	**Default**	**Description**
IniRefState		Refstate required for initialization.
RefState		State(s) in primitive variables (density, velx, vely, velz, pressure).
BCStateFile		File containing the reference solution on the boundary to be used as BC.

Riemann		
Variable	**Default**	**Description**
Riemann	RoeEntropyFix	Riemann solver to be used: LF, HLLC, Roe, RoeEntropyFix, HLL, HLLE, HLLEM

Riemann		
RiemannBC	Same	Riemann solver used for boundary conditions: Same, LF, Roe, RoeEntropyFix, HLL, HLLE, HLLEM

SplitDG		
Variable	**Default**	**Description**
SplitDG	PI	SplitDG formulation to be used: SD, MO, DU, KG, PI, CH

Exactfunc		
Variable	**Default**	**Description**
IniExactFunc		Exact function to be used for computing initial solution.
AdvVel		Advection velocity (v1,v2,v3) required for exactfunction CASE(2,21,4,8)
MachShock	1.5	Parameter required for CASE(10)
PreShockDens	1.0	Parameter required for CASE(10)
IniCenter		Shu Vortex CASE(7) (x,y,z)
IniAxis		Shu Vortex CASE(7) (x,y,z)
IniAmplitude	0.2	Shu Vortex CASE(7)
IniHalfwidth	0.2	Shu Vortex CASE(7)

Filter		
Variable	**Default**	**Description**
FilterType	None	Type of filter to be applied. None, CutOff, Modal, LAF
NFilter		Cut-off mode (FilterType==CutOff or LAF)
LAF_alpha		Relaxation factor for LAF, see Flad et al. JCP 2016
HestFilterParam		Parameters for Hesthaven filter (FilterType==Modal)

Overintegration		
Variable	**Default**	**Description**

Overintegration		
OverintegrationType	none	Type of overintegration. None, CutOff, ConsCutOff
NUnder		Polynomial degree to which solution is filtered (OverintegrationType == 1 or 2

Indicator		
Variable	**Default**	**Description**
IndicatorType	DG	Specify type of indicator to be used: DG, FV, Persson,Ducros, DucrosTimesJST, halfhalf, checkerboard
IndVar	1	Specify variable upon which indicator is applied, for general indicators.
IndStartTime	0.0	Specify physical time when indicator evalution starts. Before this timea high indicator value is returned from indicator calculation.(Idea: FV everywhere at begin of computation to smooth solution)
nModes	2	Number of highest modes to be checked for Persson modal indicator.
FVBoundaries	F	Use FV discretization in element that contains a side of a certain BC_TYPE
FVBoundaryType		BC_TYPE that should be discretized with FV.Set it to BC_TYPE, setting 0 will apply FV to all BC Sides

FV		
Variable	**Default**	**Description**
FV_SwitchConservative	T	Perform FV/DG switch in reference element
FV_IniSupersample	T	Supersample initial solution inside each sub-cell and take mean value as average sub-cell value.
FV_IniSharp	F	Maintain a sharp interface in the initial solution in the FV region
FV_IndUpperThreshold		Upper threshold: Element is switched from DG to FV if indicator rises above this value
FV_IndLowerThreshold		Lower threshold: Element is switched from FV to DG if indicator falls below this value

FV		
FV_toDG_indicator	F	Apply additional Persson indicator to check if DG solution after switch from FV to DG is valid.
FV_toDG_limit		Threshold for FV_toDG_indicator
FV_toDGinRK	F	Allow switching of FV elements to DG during Runge Kutta stages. This may violated the DG timestep restriction of the element.
FV_alpha_min	0.01	Lower bound for alpha (all elements below threshold are treated as pure DG)
FV_alpha_max	0.5	Maximum value for alpha
FV_alpha_ExtScale	0.5	Scaling factor for elpha if extended into neighboring elements
FV_nExtendAlpha	1	Number of times alpha should be passed to neighbor elements per timestep
FV_doExtendAlpha	F	Blending factor is prolongated into neighboring elements
FV_LimiterType	1	Type of slope limiter of second order reconstruction
swebyb		beta parameter for Sweby limiter

FV_Basis		
Variable	**Default**	**Description**
FV_CellType	equidistant	Specify type of FV cell distribution to be used: equidistant same legendre_gauss legendre_lobatto chebychev_lobatto

Sponge		
Variable	**Default**	**Description**
SpongeLayer	F	Turn on to use sponge regions for reducing reflections at boundaries.
damping	1.0	Damping factor of sponge. U_t=U_t-damping*(U-U_base) in fully damped regions.
SpongeShape		Set shape of sponge: (1) ramp : cartesian / vector-aligned, (2) cylindrical
SpongeDistance		Length of sponge ramp. The sponge will have maximum strength at the end of the ramp and after that point.

Sponge		
xStart		Coordinates of start position of sponge ramp (SpongeShape=ramp) or center (SpongeShape=cylindrical).
SpongeDir		Direction vector of the sponge ramp (SpongeShape=ramp)
SpongeRadius		Radius of the sponge zone (SpongeShape=cylindrical)
SpongeAxis		Axis vector of cylindrical sponge (SpongeShape=cylindrical)
SpongeViz	F	Turn on to write a visualization file of sponge region and strength.
SpongeBaseFlow	1	Type of baseflow to be used for sponge. (1) constant: fixed state,(2) exactfunction: exact function, (3) file: read baseflow file, (4) pruett: temporally varying, solution adaptive Pruett baseflow
SpongeRefState		Index of refstate in ini-file (SpongeBaseFlow=constant)
SpongeExactFunc		Index of exactfunction (SpongeBaseFlow=exactfunction)
SpongeBaseFlowFile		FLEXI solution (e.g. TimeAvg) file from which baseflow is read.
tempFilterWidth		Temporal filter width used to advance Pruett baseflow in time.)

TimeDisc		
Variable	**Default**	**Description**
TimeDiscMethod	CarpenterRK4-5	Specifies the type of time-discretization to be used, e.g. the name of a specific Runge-Kutta scheme. Possible values: * standardrk3-3 * carpenterrk4-5 * niegemannrk4-14 * toulorgerk4-8c * toulorgerk3-7c * toulorgerk4-8f * ketchesonrk4-20 * ketchesonrk4-18 * eulerimplicit * cranknicolson2-2 * esdirk2-3 * esdirk3-4 * esdirk4-6
TEnd		End time of the simulation (mandatory).
TStart	0.0	Start time of the simulation (optional, conflicts with restart).

TimeDisc		
CFLScale		Scaling factor for the theoretical CFL number, typical range 0.1..1.0 (mandatory)
DFLScale		Scaling factor for the theoretical DFL number, typical range 0.1..1.0 (mandatory)
dtmin	-1.0	Minimal allowed timestep (optional)
dtkill	-1.0	Kill FLEXI if dt gets below this value (optional)
maxIter	-1	Stop simulation when specified number of timesteps has been performed.
NCalcTimeStepMax	1	Compute dt at least after every Nth timestep.

Implicit		
Variable	**Default**	**Description**
adaptepsNewton	F	Adaptive Newton eps by Runge-Kutta error estimation
EpsNewton	0.1E-02	Newton tolerance, only used if adaptepsNewton=F
nNewtonIter	50	Maximum amount of Newton iterations
EisenstatWalker	F	Adaptive abort criterion for GMRES
gammaEW	0.9	Parameter for Eisenstat Walker adaptation
EpsGMRES	0.1E-02	GMRES Tolerance, only used of EisenstatWalker=F
nRestarts	10	Maximum number of GMRES Restarts
nKDim	30	Maximum number of Krylov subspaces for GMRES, after that a restart is performed
Eps_Method	2	Method of determining the step size of FD approximation of A*v in GMRES, 1: sqrt(machineAccuracy)*scaleps, 2: take norm of solution into account
scaleps	1.0	Scaling factor for step size in FD, mainly used in Eps_Method=1
FD_Order	1	Order of FD approximation (1/2)
PredictorType	0	Type of predictor to be used, 0: use current U, 1: use right hand side, 2: polynomial extrapolation, 3: dense output formula of RK scheme
PredictorOrder	0	Order of predictor to be used (PredictorType=2)

Preconditioner		
Variable	**Default**	**Description**
PrecondType	1	Preconditioner Type (0: no Preconditioner, 1: analytic, 2: finite difference, 3: compute both and compare)
PrecondIter	1	Defines how often preconditioner is built
SolveSystem	1	Solver of the preconditioned system (0: exact LU inversion, 1: inexact ILU(0) inversion, always with NoFillIn=T)
DebugMatrix	0	Write Jacobians to file for debug purposes (0: no output, 1: non-inverted matrix,2: additionally inverted matrix, 3: additionally check inversion accuracy)
HyperbolicPrecond	F	Preconditioner only for the hyperbolic flux
DoDisplayPrecond	F	Display building time of preconditioner

Analyze		
Variable	**Default**	**Description**
CalcErrorNorms	T	Set true to compute L2 and LInf error norms at analyze step.
AnalyzeToFile	F	Set true to output result of error norms to a file (CalcErrorNorms=T)
analyze_dt	0.0	Specifies time intervall at which analysis routines are called.
nWriteData	1	Intervall as multiple of analyze_dt at which HDF5 files (e.g. State,TimeAvg,Fluc) are written.
NAnalyze		Polynomial degree at which analysis is performed (e.g. for L2 errors). Default: 2*N.
AnalyzeExactFunc		Define exact function used for analyze (e.g. for computing L2 errors). Default: Same as IniExactFunc
AnalyzeRefState		Define state used for analyze (e.g. for computing L2 errors). Default: Same as IniRefState
doMeasureFlops	T	Set true to measure flop count, if compiled with PAPI.
PIDkill	-1.0	Kill FLEXI if PID gets below this value (optional)
NCalcPID	1	Compute PID after every Nth timestep.

AnalyzeEquation		
Variable	**Default**	**Description**
CalcBodyForces	F	Set true to compute body forces at walls
CalcBulkState	F	Set true to compute the flows bulk quantities
CalcMeanFlux	F	Set true to compute mean flux through boundaries
CalcWallVelocity	F	Set true to compute velocities at wall boundaries
CalcTotalStates	F	Set true to compute total states (e.g. Tt,pt)
CalcTimeAverage	F	Set true to compute time averages
WriteBodyForces	T	Set true to write bodyforces to file
WriteBulkState	T	Set true to write bulk state to file
WriteMeanFlux	T	Set true to write mean flux to file
WriteWallVelocity	T	Set true to write wall velolcities file
WriteTotalStates	T	Set true to write total states to file
VarNameAvg		Names of variables to be time-averaged
VarNameFluc		Names of variables for which Flucs (time-averaged square of the variable) should be computed. Required for computing actual fluctuations.

RecordPoints		
Variable	**Default**	**Description**
RP_inUse	F	Set true to compute solution history at points defined in recordpoints file.
RP_DefFile		File containing element-local parametric recordpoint coordinates and structure.
RP_MaxMemory	100	Maximum memory in MiB to be used for storing recordpoint state history. If memory is exceeded before regular IO level states are written to file.
RP_SamplingOffset	1	Multiple of timestep at which recordpoints are evaluated.

4 Workflow

In this chapter, the complete process of setting up a simulation in **FLEXI** is detailed.

4.1 Mesh generation using HOPR

FLEXI obtains its computational meshes solely from the high order preprocessor **HOPR** (available under GPLv3 at https://www.hopr-project.org) in HDF5 format. The design philosophy is that all tasks related to mesh organization, different input formats and the construction of high order geometrical mappings are separated from the *parallel* simulation code. These tasks are implemented most efficiently in a *serial* environment.

The employed mesh format is designed to make the parallel read-in process as simple and fast as possible. For details concerning the mesh format please refer to the **HOPR** HDF5 Curved Mesh Format documentation.

Using **HOPR**, simple, structured meshes can be directly created using an inbuilt mesh generator. More complex geometries can be treated by importing meshes generated by external mesh generators in CGNS or GMSH format. A number of strategies to create curved boundaries are also included in HOPR.

The test cases provided in Chapter 6 come with both a ready to use mesh file as well as a parameter file for **HOPR**, which can be used to generate or modify the meshes as needed.

Provided the mesh file has been set up, its location must be specified in the parameter file.

```
MeshFile=[path to mesh file.h5]
```

4.2 Solver settings

Before setting up a simulation, the code must be compiled with the desired parameters. The most important compiler options to be set are

- `FLEXI_EQNSYSNAME`, e.g. *Navier-Stokes*
- `FLEXI_NODETYPE`, the nodal collocation points used during the simulation. Available options are either GAUSS or GAUSS-LOBATTO.

All other options are set in the parameter file. The most important steps are

- **Set the polynomial degree `N`**

  Defines the polynomial degree of the solution. The order of convergence follows as $N+1$. Each grid cell contains $(N+1)^3$ collocation points to represent the solution.

- **Choose a de-aliasing approach.**

  For under-resolved Navier-Stokes simulations, e.g. in an LES setting, de-aliasing is important for numerical stability. Various choices are available and set using `OverintegrationType`.

  - `OverintegrationType=1`

    The first option is a filtering strategy. The complete operator is first evaluated at `N` (U_t^N) and then filtered to a lower effective degree `NUnder` (U_t^{Nunder}).

    To use this variant, specify `Nunder` to a value smaller than `N`.

  - `OverintegrationType=2`

    In this variant of the first option, the operator in reference space, e.g. JU_t, is first projected to the `NUnder` node set before converting it to physical space $U_t^{Nunder} = JU_t^{Nunder}/J^{Nunder}$. This implementation enforces conservation.

    To use this variant, specify `Nunder` to a value smaller than `N`.

  An alternative approach to guaranteeing non-linear stability is to use split form DG methods. This needs to be specified during compile time using the `FLEXI_SPLIT_DG` option, and the specific split flux formulation must be set using the parameter `SplitDG`.

- **Choose a Riemann solver**

  The Riemann solver defines how inter-element coupling is accomplished. The available variants are listed in Section 3.2. Use the `Riemann` and the `RiemannBC` options to specify which Riemann solver is to be used at internal interfaces and at Dirichlet boundary conditions, respectively. The default Riemann solver is "Roe with entropy fix".

- **Choose a time discretization method**

  The time discretization method is set using the option `TimeDiscMethod`. Various explicit Runge-Kutta variants are available and listed in Section 3.2. By default, the low-storage fourth order Runge-Kutta scheme by (Carpenter and Kennedy 1994) is employed.

4.3 Initial and boundary conditions

Initial and boundary conditions are controlled via the so-called `RefState` and `ExactFunction` constructs.

The `RefState` basically specifies a state vector in primitive form $(\rho, u, v, w, p)^T$. An arbitrary number of `RefStates` can be defined:

```
RefState=(/1,1,0,0,0.71428571/)
RefState=(/1,0.3,0,0,0.71428571/)
```

In this example, the first state would result in a parallel flow in x direction at $Ma = 1$, the second state at $Ma = 0.3$.

4.3.1 Initial conditions

The code contains a number of pre-defined analytic solution fields (ExactFunctions), which are invoked by specifying their respective number. For instance the initialization of a simple constant freestream is achieved by setting

```
IniExactFunc=1
```

The associated state vector to be used is determined by

```
IniRefState=1
```

This implies that the first of the two available `RefStates` is used for initialization.

Note: currently, the ExactFunctions contained in the code are not documented yet. They can be looked up in the source file `src/equations/navierstokes/equation.f90`.

4.3.2 Boundary conditions

The names of the boundaries are contained in the mesh file and can be used in the **FLEXI** parameter file to override the boundary conditions already set in the parameter file, if necessary.

FLEXI lists the boundaries and their respective boundary conditions during initialization:

```
|                Name       Type      State      Alpha
|        BC_periodicz-         1          0          3
|        BC_periodicy-         1          0          2
|        BC_periodicx+         1          0         -1
|        BC_periodicy+         1          0         -2
|        BC_periodicx-         1          0          1
|        BC_periodicz+         1          0         -3
```

Suppose, we wish to apply a Dirichlet boundary condition with `RefState` 2 at the two lateral boundaries. Therefor, we have to add the following lines to the parameter file

```
BoundaryName=BC_periodicy-
BoundaryType=(/2,2/)
BoundaryName=BC_periodicy+
BoundaryType=(/2,2/)
```

Note that the first index within brackets specifies `BC_TYPE`, while the second one specifies `BC_STATE`, in this case the number of the `RefState` to be used. In general, `BC_STATE` identifies either a `RefState`, an `ExactFunction` or remains empty, dependent on the `BC_TYPE`. Currently implemented boundary types for *Navier-Stokes* are listed in table 4.1.

Table 4.1: Boundary conditions.

Boundary Condition	BC_TYPE	BC_STATE	Comment
Periodic BC	1	-	Can only be defined in HOPR
Weak Dirichlet	2	`RefState`	
Weak Dirichlet	12	-	Like 2, but using an external state set by `BCStateFile`
Weak Dirichlet	22	`ExactFunction`	Like 2, but using an `ExactFunction`
Wall adiabatic	3	-	
Wall isothermal	4	`RefState`	Isothermal wall, temperature is specified via p and ρ contained in the `RefState`
Wall slip	9	-	Slip, symmetry or Euler wall [1]
Outflow Mach no.	23	`RefState`	
Outflow Pressure	24	`RefState`	
Outflow Subsonic	25	`RefState`	
Inflow Total pressure / Temp.	27	`RefState`	**Special Refstate:** *total* quantities $(T_t, \alpha, \beta, 0, p_t)$

4.4 Material properties

At present, the only available equation of state in the *Navier-Stokes* solver of **FLEXI** is the ideal gas. The gas constant, adiabatic exponent, Prandtl number and viscosity are specified in the parameter file using `R`, `kappa`, `Pr` and `mu0`.

[1] see (Carlson 2011) for details on the listed inflow/outflow boundary conditions.

4.5 Output time interval

Set the end time of the simulation using `TEnd` and the interval in which the solution is dumped to the hard drive with `Analyze_dt`.

Note that evaluation of body forces and other runtime analysis routines are also invoked once in every analyze interval determined by `Analyze_dt`. Set e.g. `nWriteData=10` to a value greater one to restrict the solution output to every 10th `Analyze_dt`.

Restart the simulation

The simulation may be restarted from an existing state file

```
flexi parameter.ini [restart_file.h5]
```

Note: when restarting from an earlier time (or zero), all later state files possibly contained in your directory are deleted!

4.6 Evaluation during runtime

At every `Analyze_dt`, the following evaluations are possible:

- `CalcErrorNorms=T`: Calculate the L_2 and L_∞ error norms based on the specified `ExactFunc` as reference. This evaluation is used for e.g. convergence tests.
- `CalcBodyForces=T`: Calculate the pressure and viscous forces acting on every wall boundary condition (BC_TYPE=3,4 or 9) separately. The forces are written to .dat files.
- `CalcBulkState=T`: Calculate the bulk quantities (e.g. bulk velocity in channel flow).
- `CalcWallVelocity=T`: Due to the discontinuous solution space and the weakly enforced boundaries, the no-slip condition is not exactly fulfilled. The deviation depends mainly on the resolution in the near-wall region. Thus, this evaluation can be used as a resolution measure at the wall.

4.7 Parallel execution

The simulation code is specifically designed for (massively) parallel execution using the MPI library. For parallel runs, the code must be compiled with `LIBS_USE_MPI=ON`.

Parallel execution is then controlled using `mpirun`

```
mpirun -np [no. processors] flexi parameter.ini
```

4.7.1 Domain decomposition

The grid elements are organized along a space-filling curved, which gives a unique one-dimensional element list. In a parallel run, the mesh is simply divided into parts along the space filling curve. Thus, domain decomposition is done *fully automatic* and is not limited by e.g. an integer factor between the number of cores and elements. The only limitation is that the number of cores may not exceed the number of elements.

4.7.2 Choosing the number of cores

Parallel performance heavily depends on the number of processing cores. The performance index is defined as

(1)
$$PID = \frac{WallTime}{nCores \cdot nDOF \cdot nTimeSteps}$$

and measures the CPU time per degree of freedom and time step. During runtime, the average PID is displayed in the output

```
CALCULATION TIME PER STAGE/DOF: [ 5.59330E-07 sec ]
```

When compared to the single performance, it can be used as a parallel efficiency measure. The PID is mainly dependent on the load per core

(2)
$$Load = nDOF/nCores$$

and the polynomial degree N. Load values for optimal performance lie in the range $Load = 2000 - 5000$. A detailed parallel performance analysis at the example of a Cray XC-40 system is given in (Atak et al. 2016).

4.8 Test case environment

The test case environment can be used as to add test case-specific code for e.g. custom source terms or diagnostics to be invoked during runtime.

The compiler option `FLEXI_TESTCASE` sets the current test case. The test cases are contained in the *src/testcase/* folder.

Standardized interfaces are defined for initialization, source terms and analysis routines

- **InitTestcase**

  Read in testcase related parameters from the `parameter.ini`, initialize the corresponding data structures.

- **FinalizeTestcase**

  Deallocate test case specific data structures.

- **ExactFuncTestcase**

  Define test case specific analytic expressions for initial or boundary conditions.

- **CalcForcing**

  Impose test case specific source terms, e.g. the pressure gradient in test case `channel`.

- **AnalyzeTestCase**

  Perform test case specific diagnostics.

Currently supplied test cases are

- **default**
- **channel**: turbulent channel flow with steady pressure gradient source term
- **phill**: periodic hill flow with controlled pressure gradient source term
- **riemann2d**: a two dimensional Riemann problem
- **taylorgreenvortex**: automatic diagnostics for the Taylor-Green vortex flow

Note that the test case environment is currently only applicable to the *Navier-Stokes* equation system.

4.9 Post processing / posti_visu tool

FLEXI comes with a posti_visu tool for visualization with Paraview. The posti_visu tool takes the HDF5 files generated by FLEXI and converts them to vtu-files readable by Paraview.

```
posti_visu [posti-prm-file [flexi-prm-file]] statefile [statefiles
   ]
```

The posti_visu tool runs in parallel with activated `LIBS_USE_MPI` flag

```
mpirun -np [no. processors] posti_visu [posti-prm-file [flexi-prm-
   file]] statefile [statefiles]
```

Multiple HDF5 files can be passed to the posti_visu tool at once. The runtime parameters to be set in `parameter_postiVisu.ini` are

Table 4.2: Runtime parameters for the posti_visu tool.

NodeTypeVisu	VISU	Node type of the visualization basis: VISU,GAUSS,GAUSS-LOBATTO,CHEBYSHEV-GAUSS-LOBATTO
NVisu		Polynomial degree at which solution is sampled for visualization.
VarName		Names of variables, which should be visualized.

Some of these options are duplicates from options for **FLEXI**. You can use the same parameter file for both executables.

The default value for NodeTypeVisu uses equidistant nodes which include the boundary points of elements. The default value for NVisu is to use the same number of sampling points as collocation points on each element. For high quality visualization, it is usually adviseable to choose NVisu higher than the polynomial degree of the computation in order to keep interpolation errors small. The parameter VarName specifies the variables being visualized. It can be used multiple times for each variable. Visualization of derived quantities such as Velocities and Pressure is possible. If no VarName is specified, the five conservative variables are visualized.

The following lines can be used as an example for the parameter_postiVisu.ini file.

```
NVisu   = 10
varName = MomentumX
varName = VelocityX
varName = Density
varName = Pressure
varName = Temperature
```

5 Tools Overview

This section gives an overview over the tools contained in the **FLEXI** repository. It also provides references to the tutorials where their usage is explained. There are two different kinds of tools:

- **POSTI**-tools can be compiled together with **FLEXI** given the according `cmake` options.
- In the `tools` folder, a collection of mostly shell or python scripts can be found. They are mostly used to manage **FLEXI** runs and **FLEXI** output files.

5.1 POSTI

POSTI tools are mostly documented in section 4.9 and in the tutorials (chapter 6). Here, an overview is given together with references to the respective tutorials. A list and description of input parameters can be shown with the command

```
[posti_toolname] --help
```

for all **POSTI** tools using a parameter file.

5.1.1 Visualization

posti_visu	
Brief description	Converts **FLEXI** state files from HDF5 format to ParaView readable `.vtu` format. Spatial resolution of output as well as calculated variables and further options can be given in the parameter file.
Basic usage	`posti_visu [parameter.ini] [statefile1.h5 ...]`
Further info / usage example	4.9, 6.1.3, 6.2.3.5, 6.3.9, 6.6.2, 6.7.2

Paraview plugin	
Brief description	A ParaView reader based on `posti_visu` to load **FLEXI** state files in ParaView. Provides the interface to adjust `posti_visu` parameters in the ParaView GUI.

Paraview plugin	
Basic usage	`libVisuReader.so` is loaded as a Plugin in ParaView
Further info / usage example	No tutorials so far

5.1.2 Swap meshes

posti_swapmesh	
Brief description	Interpolates state file data from one mesh to another. Uses high-order interpolation and a Newton coordinate search algorithm. Meshes do not have to be conforming. A reference state can be given for areas in the target mesh not covered by the source mesh.
Basic usage	`posti_swapmesh [parameter.ini] [statefile.h5]`
Further info / usage example	`ini/swapmesh` folder

5.1.3 Record points

posti_preparerecordpoints	
Brief description	Enables **FLEXI** to record values at a set of physical points over time with a higher temporal sampling rate than the state file output interval. The record point coordinates and the **FLEXI** mesh are defined in the parameter file. Creates an additional `.h5` file, whose path is passed to **FLEXI** as a parameter.
Basic usage	`posti_preparerecordpoints [parameter_prepareRP.ini]`
Further info / usage example	6.3.13

posti_visualizerecordpoints	
Brief description	Performs post-processing of the `*_RP_*` files written by **FLEXI**: merges several time steps and writes output such as value over time or spectra.
Basic usage	`posti_visualizerecordpoints [parameter_visuRP.ini] [projectname_RP_*.h5]`
Further info / usage example	6.3.13

posti_evaluaterecordpoints	
Brief description	Evaluate the values at recorpoints a posteri from existing statefiles. Can be used if the recordpoints have not been set during the simulation, but will only give coarse temporal resolution.
Basic usage	`posti_evaluaterecordpoints [parameter.ini] [statefile.h5]`
Further info / usage example	No tutorials so far

5.1.4 Time averaging

posti_mergetimeaverages	
Brief description	Averages several **FLEXI** `State` or `TimeAverage` files. If `TimeAverage` files are the input, each files is weighted with its time averaging period. `State` files are all weighted equally. All HDF5 data sets are averaged. No parameter file is needed.
Basic usage	`posti_mergetimeaverages [statefile1.h5 statefile2.h5 ...]`
Further info / usage example	No tutorials so far

posti_calcfluctuations	
Brief description	Calculates Flucuations from `Mean` and `MeanSquare` as given in the (merged) `TimeAverage` files. Fluctuations are written into an additional data set in the same HDF5 file. All applicable fluctuations are calculated. No parameter file is needed.
Basic usage	`posti_calcfluctuations [statefile1.h5 statefile2.h5 ...]`
Further info / usage example	No tutorials so far

5.1.5 Test cases

posti_channel_fft	
Brief description	Calculates the mean velocity and Reynolds stress profiles of the turbulent channel flow test case by averaging both in the direction parallel to the wall and by averaging the upper and lower half of the channel. Furthermore, kinetic energy spectra dependent on the distance to the wall are computed.
Basic usage	`posti_channel_fft [parameter_channelfft.ini] [statefile1.h5 statefile2.h5 ...]`
Further info / usage example	6.9

5.2 tools folder

The scripts provided in the tools folder are generally not part of the tutorials. They are briefly described below. The path to the python files (of the form $FLEXIROOT/tools/SUBDIR/) is omitted in the following.

For most python tools, possible arguments and syntax can be shown with the -h argument:

```
python3 [toolname.py] -h
```

5.2.1 animate

The python script **animate_paraview.py** creates movies from a series of state files using PvBatch, a GUI-less interface to ParaView. You need ParaView installed on your system (details can be found in the ParaView section) and the directory containing the PvBatch executable needs to be a part of your $PATH. Before running this script, you have to visualize your **FLEXI** state file with ParaView and save the current view via Save State..., e.g. under the name pvstate.pvsm. You also need the MEncoder tool installed. The basic command to run the script is

```
python3 animate_paraview.py -l [pvstate.pvsm] -r [
   path_to_posti_paraview_plugin] [statefile1.h5 statefile2.h5 ...
   ]
```

Apart from the movie file, the script also outputs a .png-file for each HDF5 file given as input. In order to visualize a set of .vtu-files, e.g. from posti_visu output, omit the -r argument and pass .vtu-files instead of .h5-files. Further options can be shown with the -h argument.

There are further tools for image handling in this folder:

The tool **concatenatepics.py** stitches several pairs of images (e.g. creates a time series of stitched images from two time series of images). A possible command could look like this (*Further options can be shown with the -h argument*):

```
python3 concatenatepics.py -d e -p left*.png  -a right*.png
```

The tool **crop.py** crops several images to the same size. Simply pass all images as arguments:

```
python3 crop.py [image*.png]
```

The script **pics2movie.py** creates a movie from several images using the mencoder tool (which is also done as part of the animate_paraview.py script. Basic usage is again

```
python3 pics2movie.py [image*.png]
```

and further options can again be shown with the -h argument.

5.2.2 convergence_test

The python scripts `convergence` and `convergence_grid` provide automated convergence tests for p- and h-convergence, respectively. The basic command is

```
python3 convergence flexi [parameter.ini]
```

where `convergence` can be replaced by `convergence_grid` for h-convergence. Further options can again be shown with the `-h` option.

Note that for h convergence, the mesh names are hard-coded to the form `CART_HEX_PERIODIC_MORTAR_XXX_2D_mesh.h5`, where `XXX` is the number of elements in each direction, and `MORTAR` and `2D` are optional. The polynomial degree in the parameter file is *always* overwritten by that passed to the script as an optional argument, with a default value of 3, if no such argument is passed.

5.2.3 userblock

The `userblock` contains complete information about a **FLEXI** run (git branch of the repository, differences to that branch, `cmake` configuration and parameter file) and is prepended to every `.h5` state file. The parameter file is prepended in ASCII format, the rest is binary and is generated automatically during the build process with the `generateuserblock.sh` script. It can be extracted and printed using the `extract_userblock.py` script. Its basic usage is

```
python3 extract_userblock.py -XXX [statefile.h5]
```

where `-XXX` can be replaced by

- `-s` to show all available parts of the userblock (such as `CMAKE` or `GIT BRANCH`)
- `-a` to print the complete userblock
- `-p [part]` to print one of the parts listed with the `-s` command.

The second python tool in this folder is `rebuild.py`. It extracts the userblock from a state file and builds a **FLEXI** repository and binary identical to the one that the state file was created with. In order to do so, it clones a **FLEXI** git repository, checks out the given branch, applies the stored changes to the git `HEAD` and builds **FLEXI** with the stored `cmake` options. If run with the parameter file given in the `INIFILE` part of the userblock, this binary should reproduce the same results/behaviour (possible remaining sources of different output are for example differences in restart files, compilers, linked libraries or machines). The basic usage is

```
python3 rebuild.py [dir] [statefile.h5]
```

where `dir` is an empty directory that the repository is cloned into and where the `flexi` executable is built. `statefile.h5` is the state file whose userblock is used to rebuild the `flexi` executable. Help can be shown via `-h` for both userblock scripts.

5.2.4 sortfiles.sh

The `sortfiles.sh` script sorts all `.h5`-files in subfolders `State`, `BaseFlow`, `TimeAvg` and `RP`, while keeping the last time instance at the upper level. It also copies `Log.*.sdb`, `.log` and `.out` files into a `logs` subdirectory. The project name is hard-coded in the script and has to be adapted there, the directory that is to be sorted is passed as an argument.

5.2.5 getload.py

This script is specific to runs on HPC systems. It calculates a suitable number of nodes and cores to achieve

- a number of degrees of freedom per core which is close to a target
- an average number of elements per core which is just below a close integer, such that parallel efficiency is not impaired by a few cores with higher load that the others have to wait for.

No arguments are passed to this script, all input values are hard-coded and have to be adjusted in the script.

5.2.6 testcases

The python script **plotChannelFFT.py** creates plots of the mean velocity and the Reynolds stress profiles as well as the turbulent energy spectra based on the posti_channel_fft HDF5 output files. Basic usage is:

```
python3 plotChannelFFT.py -p projectname -t time
```

Further options can be shown with the `-h` argument.

6 Tutorials

This chapter will give a detailed overview of flow simulations with **FLEXI**. It assumes that you are familiar with how to set the compiler options and how to compile the code. The path to all executables is omitted. It is assumed that you either added aliases for **flexi**, **hopr** and all posti tools, or that you added the binary directories to your `$PATH` variable as described in 2.1.3.

Each tutorial is equipped with .ini files *parameter_hopr.ini*, *parameter_flexi.ini*, *parameter_postiVisu.ini* as well as mesh file **_mesh.h5* in HDF5 format (created with **HOPR**).

```
parameter_hopr.ini
parameter_flexi.ini
parameter_postiVisu.ini
mesh.h5
```

We suggest to copy each folder to a new directory, where you can run and modify the "INI-files".

6.1 Freestream

The setup considers a freestream scenario with constant pressure $p = 101325.0$ Pa, density $\rho = 1.225$ kg/m^3 and velocity vector **U** $= (1, 1, 1)^T$ m/s.

Copy the `freestream` tutorial folder

```
    cp -r $FLEXI_TUTORIALS/freestream .
    cd freestream
```

6.1.1 Mesh Generation with HOPR

The mesh files used by **FLEXI** are created by supplying an input file *parameter_hopr.ini* with the appropriate information.

```
hopr parameter_hopr.ini
```

This creates the mesh file *cartbox_mesh.h5* in HDF5 format. Alternatively, if you do not want to run **hopr**, you can also use the provided mesh.

Figure: Mesh and flow field solution with velocity vector plot of the velocity field. View in x-y-plane.

6.1.2 Flow Simulation with FLEXI

The simulation setup is defined in *parameter_flexi.ini*. The initial condition is selected via the variable vector **RefState=(/1.225,1.,1.,1.,101325./)** which represents the solution vector $(\rho, u, v, w, p)^T$. **FLEXI** allows for multiple **RefState** vectors and numerates them respectively for them to be used by different functions. In this example a single **RefState** is supplied and therefore is given the number **1**.

IniRefState = 1 : the initial condition uses **RefState 1** for the initial flow field solution.

IniExactFunc = 1 : the used exact function routine uses **RefState 1**, e.g., for the calculation of the L_2 error norm.

Constant flow properties like the gas constant are given in table 6.1 and define the gas behavior in combination with the ideal gas law $p = \rho RT$.

Table 6.1: Numerical settings

Property	Variable	Value
dynamic viscosity μ	mu0	0.000018547
ideal gas constant R	R	276
isentropic coefficient κ	kappa	1.4

6.1.3 Numerical settings

The DG solution on the mesh is represented by piecewise polynomials and the polynomial degree in this tutorial is chosen as $N = 3$.

The default settings for these properties are displayed in table 6.2.

Table 6.2: Numerical settings

Variable	Description	Value
N	polynomial degree	3
MeshFile		cartbox_mesh.h5
tend		1e-6
Analyze_dt		1e-6
nWriteData		1
CFLscale		0.99
DFLscale		0.4

The command

```
flexi parameter_flexi.ini > std.out
```

runs the code and dumps all output into the file *std.out*. If the run has completed successfully, which should take only a brief moment, the contents of the working folder should look like in figure 6.1

Two additional files have been created, which are are named **Projectname_State_Timestamp.h5**. They contain the solution vector of the conserved variables at each interpolation node at the given time, which corresponds to multiplies of **Analyze_dt**. If these files are not present, something went wrong during the execution of **FLEXI**. In that case, check the *std.out* file for an error message.

After a successful completion, the last lines in this files should look like in figure 6.2

```
total 104K
lrwxrwxrwx 1    20 Jul 22 17:05 convert -> ../build/bin/convert
-rw-r--r-- 1  2.3K Jul 22 17:06 parameter_hopr.ini
-rw-r--r-- 1  1.3K Jul 22 17:06 parameter_flexi.ini
-rw-r--r-- 1   10K Jul 22 17:06 cartbox_mesh.h5
lrwxrwxrwx 1    18 Jul 22 17:07 flexi -> ../build/bin/flexi
-rw-r--r-- 1   31K Jul 22 17:17 cartbox_State_0000000.000000000.h5
-rw-r--r-- 1   31K Jul 22 17:17 cartbox_State_0000000.000001000.h5
-rw-r--r-- 1   17K Jul 22 17:17 std.out
```

Figure 6.1: The folder contents after a successful run

```
================================================================================================================
 INITIALIZATION DONE! [    0.03 sec ]
================================================================================================================
----------------------------------------------------------------------------------------------------------------
#GridCells :    8.0000000E+00
#DOFs      :    5.1200000E+02
#Procs     :    1.0000000E+00
#DOFs/Proc :    5.1200000E+02
 WRITING INITIAL SOLUTION:
 WRITE STATE TO HDF5 FILE...DONE  [.004s]
----------------------------------------------------------------------------------------------------------------
 Errors of initial solution:
 Sim time   :    0.0000000E+00
 L_2        :    1.878429114E-16   1.878429114E-16   1.878429114E-16   1.878429114E-16   7.101037057E-11
 L_Inf      :    8.881784197E-16   8.881784197E-16   8.881784197E-16   8.881784197E-16   2.619344741E-10
----------------------------------------------------------------------------------------------------------------
Initial Timestep  :    1.0636682E-04
 CALCULATION RUNNING...
----------------------------------------------------------------------------------------------------------------
 Sys date   :    22.07.2016 17:17:58
 CALCULATION TIME PER STAGE/DOF: [ 1.62339E-06 sec ]
 Timestep   :    1.0636682E-04
#Timesteps  :    1.0000000E+00
 WRITE STATE TO HDF5 FILE...DONE  [.006s]
 Sim time   :    1.0000000E-06
 L_2        :    1.878429114E-16   4.948834301E-16   5.307730445E-16   5.692408050E-16   7.101037057E-11
 L_Inf      :    8.881784197E-16   3.996802889E-15   4.662936703E-15   5.107025913E-15   2.619344741E-10
................................................................................................................
 FLEXI RUNNING cartbox... [    0.05 sec ]
----------------------------------------------------------------------------------------------------------------

================================================================================================================
 FLEXI FINISHED! [    0.05 sec ]
================================================================================================================
```

Figure 6.2: The *std.out* file after a successful run

To visualize the solution, the *State*-files must be converted into a format suitable for **ParaView**. Issue the command

```
posti_visu parameter_postiVisu.ini parameter_flexi.ini
   cartbox_State_0000000.00000*
```

to generate the corresponding *vtu*-files, which can then be loaded into **ParaView**.

6.2 Lid-driven Cavity

This tutorial describes how to set up and run the first non-trivial flow problem. The lid-driven cavity flow is a standard test case for numerical schemes, and a number of results have been published in literature, see e.g. (Ghia 1982), (Gao 2016). This tutorial assumes that you have completed the previous tutorial, know how to edit files and postprocess the solution with your favorite visualization tool, e.g. ParaView. Also, the later parts of the tutorial assume that you have access to a computer with an MPI-based parallelization with at least 4 computing cores - otherwise, it will just take a lot longer :).

The tutorial is split into two parts: The basic part will teach you about setting up the code and running the simulations. The advanced part will build on this and give you a glimpse on how to make modification to code to accommodate more complex simulation and add features you might need. If you are just interested in running the code as is, you may skip the advanced part, or just complete parts of it.

6.2.1 Flow description

The flow under consideration is essentially incompressible and two-dimensional, but we will use the three-dimensional code for the compressible Navier-Stokes equations to solve it here. This is not the most efficient way to compute this flow, but it works well to show you how to set up and run a simulation in this tutorial. The computation will be conducted in a three-dimensional, square domain with periodic boundary conditions in the "third" direction. The wall of the cavity are modelled as isothermal walls, and a fixed flow is prescribed at the upper boundary, i.e. the lid of the domain. For the Reynolds numbers investigated here, this generates a steady, vortical flow field in the cavity.

The following picture (figure 6.3) shows the resulting velocity field and streamlines for $Re = 400$.

6.2.2 Compiler options

Make sure that **FLEXI** is compiled with the cmake options listed in the following table.

Figure 6.3: Contours of velocity magnitude for the $Re = 400$ lid-driven cavity case.

Table 6.3: Cmake options for the cavity simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional

The standard settings are sufficient for this example. To check whether they are set, change to your `build` folder and open the cmake GUI

```
ccmake [flexi root directory]
```

If necessary, set the above options and then compile the code by issuing

```
make
```

6.2.3 Part I: Basic Tutorial - Flow at $Re = 100$

Copy the `cavity` tutorial folder

```
    cp -r $FLEXI_TUTORIALS/cavity .
```

Step into the `Basic_Re100` subfolder. The contents of the folder should look like in picture (figure 6.4). In case you do not want to generate the mesh files yourselves, the meshes have already been provided.

```
total 76
-rw-r--r-- 1                 9160 Aug 25 09:15 cavity2x2_mesh.h5
-rw-r--r-- 1                13634 Aug 25 09:15 cavity4x4_mesh.h5
-rw-r--r-- 1                33882 Aug 25 09:15 cavity8x8_mesh.h5
-rw-r--r-- 1                   65 Aug 25 09:15 parameter_flexi2vtk.ini
-rw-r--r-- 1                 3679 Aug 25 09:15 parameter_flexi.ini
-rw-r--r-- 1                 2536 Aug 25 09:15 parameter_hopr.ini
```

Figure 6.4: Files for the basic lid-driven cavity tutorial.

6.2.3.1 Mesh Generation with HOPR

The domain of interest consists of a square 2D geometry. Although the flow field is two-dimensional, we will create a three-dimensional domain here and apply periodic boundary conditions in the $z-$direction. Also, we will only use *one* element in that direction to save computational costs.

Like before, the mesh files used by **FLEXI** are created by supplying an input file *parameter_hopr.ini* with the appropriate information to **HOPR**.

```
hopr parameter_hopr.ini
```

A full tutorial on how to run **HOPR** is available at the project site HOPR-project, so we will just review the basics here. In *parameter_hopr.ini*, the following lines create a cubical domain from $[0,0,0] \rightarrow [1,1,1]$, discretized by 4×4 elements in the $x-y$ plane and 1 element in the $z-$direction.

```
Corner =(/0.,0.,0. ,,1.,0.,0. ,,1.,1.,0. ,,  0.,1.,0.,, 0.,0.,1.
    ,,1.,0.,1. ,,1.,1.,1. ,,  0.,1.,1. /)
! Corner node positions: (/ x_1,y_1,z_1, x_2,y_2,z_2,..... , x_8,
    y_8,z_8/)
nElems =(/4,4,1/)                ! number of elements in each
    direction
```

The next line maps the 6 faces of the cube to the boundary conditions following the CGNS notation, i.e. it indicates which of the boundary conditions defined below belong to which face.

```
BCIndex       =(/1,3,6,4,5,2/)         ! Indices of Boundary
    Conditions for  six Boundary Faces (z-,y-,x+,y+,x-,z+)
```

For example, the first boundary condition defined will be applied to the $z-$ face of the cube, while the third one will be applied to the $y-$ face and so on. The order $(z-, y-, x+, y+, x-, z+)$ is given by the CGNS standard and thus fixed, so if you keep this line as it is, the boundary conditions below can be defined in the (less confusing) order of $(z-, z+, y-, y+, x-, x+)$.

Figure 6.5: BC Indices for the cube, front and back view.

The boundary conditions are now assigned to faces in the following lines:

```
BoundaryName=BC_zminus              ! BC index 1
BoundaryType=(/1,0,0,1/)            ! (/ Type, curveIndex, State,
   alpha /)
BoundaryName=BC_zplus               ! BC index 2
BoundaryType=(/1,0,0,-1/)           !
vv=(/0.,0.,1./)                     !

BoundaryName=BC_wall_lower          ! BC index 3
BoundaryType=(/4,0,1,0/)
BoundaryName=BC_free                ! BC index 4
BoundaryType=(/2,0,0,0/)

BoundaryName=BC_wall_left           ! BC index 5
BoundaryType=(/4,0,1,0/)
BoundaryName=BC_wall_right          ! BC index 6
BoundaryType=(/4,0,1,0/)
```

The identifier *BoundaryName* can be chosen freely, it is good practice to start it with *BC_*. The identifier *BoundaryType* specifies what kind of boundary is to be applied to the face. For the $z-$oriented faces, periodic boundary conditions are chosen, which are denoted by type 1, the first index in the array. The second index indicates if the face is *curved* (not for this test case), and the third indicates which *reference state* will be used at the boundary. This will be discussed in more detail later, it is important to note that the state can be set here when generating the grid, but can also be overwritten in *parameter_flexi.ini* later. The fourth entry is used to match periodic boundary condition sides, i.e. these need to form a pair of the form $[a, -a]$, where a is a unique integer for each periodic pair and the sign specifies the orientation. In this example, the $z-$ and $z+$ sides are connected and the associated connection vector *vv* is in the $z-$direction. In summary, to specify a periodic side pair, the first index in *BoundaryType* must be 1, the fourth index must be a matching pair of integers and the connection vector must be specified.

In the $y-$direction, we choose an isothermal wall boundary condition at the lower side (type 4) and a Dirichlet / fixed state at the top of the cavity (type 2). Again, the corresponding states can be set here, but overwritten later.

In the $x-$direction, the left and right walls are again chosen to be isothermal wall boundary conditions.

For this basic tutorial, the simple meshes shown in picture (figure 6.7) will be used.

6.2.3.2 Preparing the Flow Simulation with FLEXI

The simulation setup is defined in *parameter_flexi.ini*. To get help on any of the parameters listed therein, you can run **FLEXI** from the command line by typing

Figure 6.6: BC Types for the lid driven cavity, front and back view.

Figure 6.7: Meshes for the basic lid-driven cavity tutorial.

```
flexi --help
```

or

```
flexi --help SECTION
```

or

```
flexi --help PARAMETER
```

The first command lists all parameters grouped thematically, however, this is not mandatory. The second command lists all parameters of a certain section and the last command only gives help about a specific option. All lines starting with a "!" are comments.

6.2.3.2.1 Output parameters

```
! ================================================ !
! OUTPUT
! ================================================ !
ProjectName    = Tutorial_Cavity_Re100
outputFormat   = 0
```

The parameter *ProjectName* is used to name all the files generated by the simulation, e.g. the state at time 0.3 (the solution vector of conserved variables at each node) is written to the file *Tutorial_Cavity_Re100_State_0000000.300000000.h5* and so on. The parameter *outputFormat* is set to 0 here, indicating that the output is *off*. Turning it on will result in an on-the-fly **visualization** output of the solution states, but this is *not* recommended as good practice - it will slow down the code considerably, in particular in parallel mode. State-files in the HDF5-format are written always independent of this parameter. The recommended way is to use the *posti_visu* tool after the simulation to generate Paraview-readable files. Note that currently, the only output format available is *vtk* (by setting outputFormat $= 3$), since Tecplot output is not available due to GPL licensing issues.

6.2.3.2.2 Interpolation / Discretization parameters

```
! ================================================ !
! INTERPOLATION
! ================================================ !
N              = 3
NAnalyze       = 10
```

The parameter *N* sets the degree of the solution polynomial, e.g. in this example, the solution is approximated by a polynomial of degree 3 in each spatial direction. This results in $(N+1)^3$ degrees of freedom for each (3D) element. In general, *N* can be chosen to be any integer greater or equal to 1, however, the discretization and the timestep calculation has not extensively been tested beyond $N \approx 23$. Usually, for a good compromise of performance and accuracy is found for $N \in [3, .., 9]$.

NAnalyze determines the polynomial degree for the analysis routines, e.g. the accuracy of the calculation of error norms or testcase specific integrals during the computation. A good rule of thumb is to set $NAnalyze = 2 \times N$.

6.2.3.2.3 Mesh parameters

```
! ================================================ !
! MESH
! ================================================ !
MeshFile       = cavity2x2_mesh.h5
useCurveds     = F

!BoundaryName=BC_wall_lower
!BoundaryType=(/4,1/)
BoundaryName=BC_wall_left
BoundaryType=(/4,1/)
BoundaryName=BC_wall_right
BoundaryType=(/4,1/)
BoundaryName=BC_free
BoundaryType=(/2,1/)
```

The parameter *MeshFile* contains the name of the **HOPR** mesh file in HDF5 format (and/or the full path to it). *UseCurveds* indicates whether the mesh is considered to be curved, i.e. if high-order mesh information should be used. Setting this to *F* can be used to discard high order information in the mesh file and treat it as a linear mesh. For the current tutorial, the meshes are linear by design.

The boundary conditions are set via the *BoundaryName* identifier, which specifies a name that must be present in the mesh file (see section on mesh generation above). Each line containing the boundary name must be followed by a line containing the *BoundaryType*. A list of types available for the Navier-Stokes equations can be found in table 4.1. For types that require additional information (like Dirichlet boundaries), the second index in *BoundaryType* refers to the *RefState* (short for reference state) which is used to determine the unknowns / quantities from the outside for this boundary condition. For example, for a Dirichlet boundary (Type 2), the full reference state is set at the boundary, so the lines

```
BoundaryName=BC_free
BoundaryType=(/2,1/)
```

indicate that the *first* reference state vector listed below is set at this boundary (the lid part of the cavity). Note that the reference vectors are always in primitive variables, i.e. $(\rho, u, v, w, p)^T$ *unless* specified otherwise. Also, for the boundaries *BC_wall_left* and *BC_wall_right*, the same reference state is chosen. These boundaries are isothermal walls, so a wall temperature needs to be specified - this is computed from the primitive variables in the associated reference state.

Note that the lines for the lower wall boundary are commented out. As has been mentioned above, the boundary conditions can be set in **HOPR** directly, and then be overwritten here.

This is just an example to show how both approaches work. Later, when running **FLEXI** with these settings, it is good practice to inspect the boundary condition information as understood by **FLEXI**. In this case, the output of **FLEXI** to the console will look like this:

```
  ----------------------------------------------------------------
  |          BoundaryName |     BC_wall_left | *CUSTOM |
  |          BoundaryType |       (/ 4, 1 /) | *CUSTOM |
  |          BoundaryName |    BC_wall_right | *CUSTOM |
  |          BoundaryType |       (/ 4, 1 /) | *CUSTOM |
  |          BoundaryName |          BC_free | *CUSTOM |
  |          BoundaryType |       (/ 2, 1 /) | *CUSTOM |
  |     Boundary in HDF file found |  BC_free
  |                            was |  2 0
  |                     is set to |  2 1
  |     Boundary in HDF file found |  BC_wall_left
  |                            was |  4 1
  |                     is set to |  4 1
  |     Boundary in HDF file found |  BC_wall_right
  |                            was |  4 1
  |                     is set to |  4 1
  ..............................................................
  BOUNDARY CONDITIONS|           Name   Type  State     Alpha
  |                  |      BC_zminus      1      0         1
  |                  |       BC_zplus      1      0        -1
  |                  |  BC_wall_lower      4      1         0
  |                  |        BC_free      2      1         0
  |                  |   BC_wall_left      4      1         0
  |                  |  BC_wall_right      4      1         0
  ..............................................................
```

6.2.3.2.4 Equation parameters In this section of the parameter file, the settings associated with the equation to be solved are set, including initial conditions and reference data for the boundary conditions. The parameter *IniExactFunc* specifies which solution or function should be used to fill the initial solution vector, i.e. it specifies what the starting flow field looks like. Setting this to 1 selects a uniform initial state in the whole domain. Note that this solution is also used to compute the errors norms and can also be time-dependent. The state itself is defined by the parameter *IniRefstate*, in this case, the second one is used.

```
! ============================================= !
! EQUATION
! ============================================= !
IniExactFunc  = 1
IniRefState   = 2
RefState      = (/1.0,1.,0.,0.,71.4285714286/)
RefState      = (/1.0,0.,0.,0.,71.4285714286/)
mu0           = 0.01
R             = 1
Pr            = 0.72
```

```
kappa           = 1.4
```

As described above, the RefStates are given in primitive variables. The second one describes a fluid at rest and is used to initialize a resting fluid in the cavity. The first state is used to determine the driving flow at the top of the cavity (and to compute the wall temperatures for the boundaries). Constant flow properties like the gas constant are given in table 6.4 and define the gas behavior in combination with the ideal gas law $p = \rho RT$. Note that the code itself does not distinguish between dimensional and non-dimensional quantities and it is the user's responsibility to set all data consistently. For anything other than an ideal gas with constant viscosity and heat conductivity, physically meaningful quantities should be set.

Table 6.4: Numerical settings

Property	Variable	Value
dynamic viscosity μ	mu0	0.1
ideal gas constant R	R	1
isentropic coefficient κ	kappa	1.4
Prandtl number Pr	Pr	0.72

From these settings, the Mach- and Reynolds number can be computed as follows, taking into account a reference cavity length of 1 and the magnitude of the driving velocity:

$$Mach = u/c = 1.0/\sqrt{\kappa\frac{p}{\rho}} = 1.0/10.0 = 0.1 \tag{6.1}$$

$$Re = \frac{uL\rho}{\mu_0} = \frac{1.0}{0.01} = 100 \tag{6.2}$$

Since we are comparing against an incompressible reference solution, setting the Mach number to 0.1 is a good compromise between accuracy and efficiency.

6.2.3.3 Temporal discretization parameters

```
! ================================================ !
! TIMEDISC
! ================================================ !
tend            = 5.0
Analyze_dt      = 0.1
nWriteData      = 1
CFLscale        = 0.9
DFLscale        = 0.4
```

The parameter *tend* determines the end time of the solution, *Analyze_dt* the interval at which the analysis routines (like error computation, checking of wall velocities etc. see below)

are called. The multiplier *nWriteData* determines the interval at which the solution state files are written to the file system, e.g. in this case *nWriteData* $\times$ *Analyze_dt* $= 0.1$ is the interval for writing to disc. The CFL and DFL numbers determine the explicit time step restriction for the advective and viscous parts. Note that these values should always be chosen to be < 1, but since the determination of the timestep includes some heuristics, both values should be chosen conservatively.

6.2.3.4 Analysis parameters

```
! ============================================================ !
! ANALYZE
! ============================================================ !
CalcErrorNorms=   T   ! Calculate error norms
CalcBodyForces=   T   ! Calculate body forces
CalcWallVelocity= T   ! Calculate wall velocities
CalcMeanFlux=     T   ! Calculate mean flux through boundaries
```

These switches trigger the output of analysis files.

Now that all the necessary preparations have been made, the simulation can be started.

6.2.3.5 Running the Simulation and Results

The command

```
flexi parameter_flexi.ini > std.out
```

runs the code and dumps all output into the file *std.out*. If you wish to run the code in parallel using MPI, the standard command is

```
mpirun -np XX flexi parameter_flexi.ini > std.out
```

where XX is an integer denoting the number of processes to be used in parallel. Note that **FLEXI** uses an element-based parallelization strategy, so the minimum load per process/core is *one* grid element, i.e. do not use more cores than cells in the grid!

If the simulation runs without error, the screen output should look similar to figure 6.8, showing information on the timestep and producing the analysis results selected via the parameter file. Once the run has completed, your working folder will contain a number of *State* files in HDF5 format and * *.dat* ASCII files for the analysis data, which can be opened in Tecplot, a spreadsheet tool etc.

Since we start the simulation from a fluid at rest, it will take some iterations / timesteps to achieve a steady state solution. One way to check if the solution has converged to a steady state is to check some characteristic quantities, e.g. the velocities at the walls, body forces etc. In figure 6.9, the temporal evolution of the velocities at the lower wall are plotted

```
-----------------------------------------------------------------------------------------------------------------------------------
 Sys date   :    24.08.2016 08:35:18
 CALCULATION TIME PER STAGE/DOF: [ 1.16975E-06 sec ]
 Timestep   :    1.2067757E-03
#Timesteps  :    4.1480000E+03
 WRITE STATE TO HDF5 FILE...DONE  [.004s]
 Sim time   :    5.0000000E+00
 L_2        :    1.289693723E-03   2.128604329E-01   1.383870394E-01   5.252758466E-14   2.108414847E-01
 L_Inf      :    2.420289377E-02   1.069901857E+00   4.860091863E-01   1.737766162E-13   4.556966951E+00
 BodyForces (Pressure, Friction) :
 BC_wall_lower   0.000000000E+00  -7.140364064E+01   0.000000000E+00  -2.981739128E-03   5.798766785E-08   3.248462435E-15
  BC_wall_left  -7.134010650E+01   0.000000000E+00   0.000000000E+00   1.634003008E-03   3.464824268E-02   4.047275268E-15
 BC_wall_right   7.150280223E+01  -1.056875702E-14  -1.323067623E-15   2.536650367E-03  -6.025536013E-02   1.401208630E-15
 Wall Velocities (mean/min/max)  :
 BC_wall_lower   3.232372981E-04   1.061193608E-05   7.034797445E-04
  BC_wall_left   9.332163661E-03   1.235531286E-05   1.700193024E-01
 BC_wall_right   1.550219379E-02   1.516219508E-05   2.161874378E-01
 MeanFlux through boundaries     :
     BC_zminus   3.913575833E-14  -1.158330336E-14  -9.262376085E-15   7.138863293E+01   9.785035970E-12
 BC_wall_lower   0.000000000E+00  -2.981739128E-03  -7.140363925E+01   3.248462435E-15   8.083132240E-04
       BC_free   4.800952919E-05  -1.649951057E-01   7.142925430E+01   1.742799326E-15  -5.107493538E-02
  BC_wall_left   0.000000000E+00  -7.133828904E+01   3.464824268E-02   4.047275268E-15   1.822428276E-02
 BC_wall_right   0.000000000E+00   7.150625517E+01  -6.025536013E-02   7.812405298E-17   3.401372014E-02
...................................................................................................................................
 FLEXI RUNNING Tutorial_Cavity_Re100_mesh8x8... [   12.77 sec ]
-----------------------------------------------------------------------------------------------------------------------------------

===================================================================================================================================
 FLEXI FINISHED! [   12.78 sec ]
-----------------------------------------------------------------------------------------------------------------------------------
```

Figure 6.8: Sample screen output of the cavity simulation.

over time for 4 different simulations, and for all cases, at $t_{end} = 5$, a sufficiently stationary solution has been achieved. Note that since the boundary conditions are applied *weakly* in a DG setting, a velocity slip at walls can occur, with its magnitude depending on the local wall resolution.

For comparison with published data, 4 simulations where run on the provided meshes. Details are listed in table 6.5.

Table 6.5: $Re = 100$ lid driven cavity: Simulations

Simulation	Mesh	N
1	cavity_2x2	3
2	cavity_4x4	3
3	cavity_8x8	3
4	cavity_2x2	7

Figure 6.9: Time evolution of wall velocity at lower wall for the $Re = 100$ lid-driven cavity case.

Figure 6.10: Contours of velocity magnitude for the $Re = 100$ lid-driven cavity case.

A contour plot of the velocity magnitude at the end time is given in figre 6.10. To generate this plot, convert the *State* files to a Paraview format by invoking

```
posti_visu parameter_postiVisu.ini parameter_flexi.ini
   your_Statefiles.h5
```

For a more quantitative comparison with published data, you can generate a plot of the $u-$velocity on the centerline ($x = 0$) of the cavity. Figure 6.11 shows the results for the 4 simulations run here, along with published data available in (Ghia 1982), (Gao 2016).

Figure 6.11: Comparison of centerline velocities for the $Re = 100$ lid-driven cavity case with published results.

For simulation 1, the agreement with literature results is fair. This is due to the coarse resolution with $2 \times (3 + 1) = 8$ degrees of freedom per x- and y-direction. Doubling the grid elements results in an improved match, doubling it again (simulation 3), the agreement with

the published data is excellent. It should be noted that the same accuracy can be achieved by increasing N and keeping the coarse grid. Simulation 3 and 4 have nearly identical results, although the number of degrees of freedom differs by a factor of 2. This is a feature of high order schemes for smooth problems.

6.2.4 Part II: Advanced Tutorial - Flow at $Re = 400$

In this part of the tutorial, we will expand upon what has been learned in the basic section above. The general setup of the computation remains the same, but the Reynolds number is increased, which will be accounted for by a new mesh and a higher resolution. Also, we will give a first glimpse at how to make your own changes to the code and show how to add a new function to be used as a custom initial or boundary condition. It is recommend that you have completed the basic tutorial, have access to at least 4 cores for the computation (or just have the patience to wait longer) and are familiar with **FORTRAN** syntax.

If not done so already, copy the `cavity` tutorial folder

```
cp -r $FLEXI_TUTORIALS/cavity .
```

and step into the `Advanced_Re400` subfolder.

6.2.4.1 Mesh Generation with HOPR

To account for the increased Reynolds number, the number of elements in the $x-y$-plane is increased to 12×12. Also, a stretching in the $y-$direction is introduced, as can be seen in picture 6.12. In *parameter_hopr.ini*, the following line facilitates an exponential stretching:

```
factor        =(/1.0,-1.2,1./)                    ! element stretching
   a constant growth factor (+/- changes direction)
```

You can generate your own mesh or re-use the provided one, labeled *cavity12x12_stretch_mesh.h5*.

6.2.4.2 Adding a custom initial or boundary function

Before running the simulation, we will include a custom function to be used as boundary condition for the top boundary, i.e. the velocity driving the flow in the cavity. In this, we follow the suggestions from (Gao 2016), where the $u(x)$ velocity at the lid is given as

$$u(x) = \begin{cases} c_1x^4 + c_2x^3 + c_3x^2 + c_4x, & \text{if } 0 \leq x < 0.2 \\ d_1x^4 + d_2x^3 + d_3x^2 + d_4x + d_5, & \text{if } 0.8 < x \leq 1.0 \\ 1, & \text{otherwise} \end{cases} \tag{6.3}$$

Figure 6.12: Stretched mesh for the $Re = 400$ lid-driven cavity case.

with

$$
\begin{aligned}
[c_1, c_2, c_3, c_4] =& 1000 \times [4.9333, -1.4267, 0.1297, -0.0033] \\
[d_1, d_2, d_3, d_4, d_5] =& 10000 \times [0.4933, -1.8307, 2.5450, 1 - .5709, 0.3633]
\end{aligned}
\tag{6.4}
$$

This assumes the top bondary to be from $0 \ldots 1$, as is the case in our domain. To add this new function to **FLEXI**, locate the file "$FLEXI/src/equations/navierstokes/idealgas/exactfunc.f90" and open it in the text editor of your choice. Locate the *SUBROUTINE ExactFunc*, which provides functions to the boundary, initial condition and analysis routines of the code. The header of the routine you are looking for is shown in figure 6.13.

```
!==================================================================================================================
!> Specifies all the initial conditions. The state in conservative variables is returned.
!> t is the actual time
!> dt is only needed to compute the time dependent boundary values for the RK scheme
!> for each function resu and the first and second time derivative resu_t and resu_tt have to be defined (is trivial for consta
!==================================================================================================================
SUBROUTINE ExactFunc(ExactFunction,tIn,x,resu)
! MODULES
USE MOD_Preproc       ,ONLY: PP_PI
USE MOD_Globals       ,ONLY: Abort,CROSS
USE MOD_Equation_Vars,ONLY: Kappa,sKappaM1,KappaM1,AdvVel,RefStateCons,RefStatePrim,IniRefState
USE MOD_Equation_Vars,ONLY: IniCenter,IniHalfwidth,IniAmplitude,IniAxis
USE MOD_Equation_Vars,ONLY: P_Parameter,U_Parameter
USE MOD_Timedisc_Vars,ONLY: fullBoundaryOrder,CurrentStage,dt,RKb,RKc,t
USE MOD_TestCase,     ONLY: ExactFuncTestcase
USE MOD_EOS,          ONLY: PrimToCons,ConsToPrim
IMPLICIT NONE
!------------------------------------------------------------------------------------------------------------------
! INPUT/OUTPUT VARIABLES
INTEGER,INTENT(IN)              :: ExactFunction          !< determines the exact function
REAL,INTENT(IN)                 :: x(3)                   !< physical coordinates
REAL,INTENT(IN)                 :: tIn                    !< solution time (Runge-Kutta stage)
REAL,INTENT(OUT)                :: Resu(5)                !< state in conservative variables
!------------------------------------------------------------------------------------------------------------------
! LOCAL VARIABLES
REAL                            :: tEval
REAL                            :: Resu_t(5),Resu_tt(5),ov ! state in conservative variables
REAL                            :: Frequency,Amplitude
REAL                            :: Omega
REAL                            :: Vel(3),Cent(3),a
REAL                            :: Prim(5)
REAL                            :: r_len
REAL                            :: Resul(5),Resur(5)
REAL                            :: random
REAL                            :: du, dTemp, RT, r2                ! aux var for SHU VORTEX,isentropic vortex case 12
REAL                            :: pi_loc,phi,radius                ! needed for cylinder potential flow
REAL                            :: h,sRT,pexit,pentry               ! needed for Couette-Poiseuille
REAL                            :: vec_c(1:5), vec_d(1:5), x_p(1:5)  ! needed for lid driven cavity flow, case 9
INTEGER                         :: i
!==================================================================================================================
```

Figure 6.13: Header for the subroutine *ExactFunc*

To add equation 6.3 to the code, add a new *CASE* to the routine, in which you define the state vector for the primitive variables, and then convert them to conservative ones (see e.g. case 8 for how this is done). You might also need to introduce some new local variables for this routine. To check if your changes are syntactically correct, compile the code with your changes by invoking *make* from your build directory. If the compilation process was not successful, check the output on the screen that will give you some hints on what might be wrong.

To benefit from this tutorial, it is recommend that you do try to complete this programming task. For reference, figure 6.14 shows a sample piece of code with a correct implementation as *CASE 9*. From now on, we will refer to *CASE 9* as the case number in question, yours might be different of course.

```
CASE(9) !lid driven cavity flow from Gao, Hesthaven, Warburton
        !"Absorbing layers for weakly compressible flows", to appear, JSC, 2016
        ! Special "regularized" driven cavity BC to prevent singularities at corners
        ! top BC assumed to be in x-direction from 0..1
  Prim = RefStatePrim(IniRefState,:)
  DO i=1,5
   x_p(i)=x(1)**(i-1)
  END DO
  vec_c(1:5)=1000*(/0.0,-0.0033,0.1297,-1.4267,4.9333/)
  vec_d(1:5)=10000*(/0.3633,-1.5709,2.5450,-1.8307,0.4933/)

  IF (x(1).LT.0.2) THEN
    prim(2)=DOT_PRODUCT(vec_c,x_p)
  ELSEIF (x(1).LE.0.8) THEN
    prim(2)=1.0
  ELSE
    prim(2)=DOT_PRODUCT(vec_d,x_p)
  ENDIF
  CALL PrimToCons(prim,Resu)
```

Figure 6.14: Sample code for the custom equation from (Gao 2016).

Note that in the sample code, we do not specify the full primitive state vector within the routine, but re-use the *IniRefState* and just overwrite the u-velocity. This is a matter of choice, but it allows to set the Mach number by setting the *IniRefState* accordingly.

6.2.4.3 Preparing the Flow Simulation with FLEXI

To setup the simulation, you can either modify the *parameter_flexi.ini* files from the basic tutorial or use the provided ones. We will conduct 2 simulations: one with the constant driving flow boundary conditions as before, and one with the new custom equation specified as equation *CASE 9*. Two parameter files for these cases are provided, the one with the custom boundary condition named *parameter_flexi_custombc.ini*. For both cases, the changes in the parameterfile are:

```
MeshFile       = cavity12x12_stretch_mesh.h5
...

mu0            = 0.0025
                 ...

tend           = 100.0
```

The mesh file needs to be adjusted as well as the viscosity to set the Reynolds number. Also, the end time is set to 100 to account for the longer accomondation period the simulation will need to achieve a steady state. Setting it to 100 is very conservative, as we will see later, feel free to lower it as you see fit.

For the custom boundary condition case, the parameter file needs to be adjusted to address the new boundary settings. According to table 4.1, Dirichlet boundary conditions with a

specified reference equation (instead of a reference *state*) are of type 22. The second index in the entry refers thus to the equation (*CASE 9*) we have programmed above.

```
BoundaryName=BC_free
BoundaryType=(/22,9/)
```

Now we are ready to run both simulations and compare the results.

6.2.4.4 Running the Simulation and Results

The command

```
flexi parameter_flexi.ini > std.out
```

runs the code and dumps all output into the file *std.out*. If you wish to run the code in parallel using MPI, the standard command is

```
mpirun -np XX flexi parameter_flexi.ini > std.out
```

Change the parameter file to *parameter_flexi_custombc.ini* for the second run. Note that you can adust the end time if you wish. From figure 6.15, which shows the evolution of the mean velocity at the lower wall, the flow reaches steady state after about $t = 35$. The following figures show the flow field and comparison of the centerline velocities with published results.

Figure 6.15: Evolution of wall velocities at the lower wall for $Re = 400$ lid driven cavity simulations.

Y
X
Z
VelocityMagnitude: 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1

Figure: Steady state solution of velocity magnitude of $Re = 400$ lid driven cavity. Left: Constant boundary condition, Right: Custom boundary condition.

6.3 Flow around a NACA 0012 airfoil

In this tutorial, the simulation around a NACA 0012 airfoil at $Re = 5000$ and $Ma = 0.4$ is considered. First, we explain how to set the main flow parameters. We then describe the evaluation of lift and drag and visualization of the flow field. Finally, we show how to use the *sponge zone* to remove artificial reflections from the outflow boundary, so that a clean acoustic field is retained.

Copy the `naca0012` tutorial folder to your desired working directory.

```
cp -r $FLEXI_TUTORIALS/naca0012 .
```

Figure 6.16: Evolution of wall velocities at the lower wall for $Re = 400$ lid driven cavity simulations.

Figure 6.17: Contours of velocity magnitude for the flow around a NACA 0012 airfoil.

6.3.1 Compiler options

Make sure that **FLEXI** is compiled with the cmake options listed in the following table.

Table 6.6: Cmake options for the NACA 0012 simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional

The standard settings are sufficient for this example. To check whether they are set, change to your `build` folder and open the cmake GUI

```
ccmake [flexi root directory]
```

If necessary, set the above options and then compile the code by issuing

```
make
```

6.3.2 Mesh Generation with HOPR

The mesh file used by **FLEXI** is created from the external linear mesh `NACA0012_652.cgns` and a 3rd order boundary description `NACA0012_652_splitNg2.cgns` using **HOPR**

```
hopr parameter_hopr.ini
```

This creates the mesh file *NACA0012_652_Ng2_mesh.h5* in HDF5 format. If **HOPR** is not available, the mesh file is supplied in this tutorial.

6.3.3 Flow Simulation with FLEXI

The simulation setup is defined in `parameter_flexi.ini`. The initial condition is selected via the variable vector `RefState=(/1.,0.990268069,0.139173101,0.,4.4642857/)` which represents the vector of primitive solution variables $(\rho, u, v, w, p)^T$. The chosen velocity vector $(u, v)^T$ yields an angle of attack of $\alpha = 8°$ and a velocity magnitude of 1. The first supplied `RefState` is given the number 1, the second is given the number 2 and so on.

`IniRefState = 1` : the initial condition uses `RefState 1` for the initial flow field solution.

`IniExactFunc = 1` : exact function routine for initialization, case 1 initializes a freestream state based on `IniRefState`.

Material properties are given in table 6.7. Based on the ideal gas law, we get

$$Ma = 1/\sqrt{\kappa p/\rho} = 0.4$$

Note that in this non-dimensional setup the mesh is scaled such that the chord length is unity, i.e. $C = 1$. Then to arrive at $Re = \rho u C/\mu = 5000$, the viscosity is set to

$$\mu = \rho u C/Re = 1/Re = 0.0002$$

Table 6.7: Material properties set in the parameter file

Property	Variable	Value
dynamic viscosity μ	mu0	0.0002
ideal gas constant R	R	4.4642857
Prandtl number	Pr	0.72
isentropic coefficient κ	kappa	1.4

6.3.4 Numerical settings

The DG solution on the mesh is represented by piecewise polynomials and the polynomial degree in this tutorial is chosen as $N = 3$.

The main code settings are displayed in table 6.8.

Table 6.8: Numerical settings

Variable	Description	Value
N	Polynomial degree	3
MeshFile	Mesh file to be used	NACA0012_652_Ng2_mesh.h5
tend	end time of the simulation	10
Analyze_dt	time interval for analysis	0.01
nWriteData	dump solution every n'th Analyze_dt	10
CFLscale		0.9
DFLscale		0.9

6.3.5 Boundary conditions

The boundary conditions were already set in the mesh file by **hopr**. Thus, the simulation runs without specifying the boundary conditions in the **FLEXI** parameter file. The freestream boundaries of the mesh are Dirichlet boundaries using the same state as the initialization, e.g. `IniRefState` by default.

The boundary types and states used by the simulations can always be checked in the initial console output of **FLEXI**. In order to run a quick test run, first set

```
maxiter=1
```

in your parameter file to enforce the simulation to stop after one time step. Then run the code

```
flexi parameter_flexi.ini
```

The code will finish after one time step. The initialization output contains the boundary conditions

```
          Name       Type      State      Alpha
        BC_wall          3          0          0
      BC_inflow          2          0          0
     BC_outflow          2          0          0
      BC_zminus          1          0          1
       BC_zplus          1          0         -1
```

Type 2 identifies the weak Dirichlet boundary condition, type 3 the adiabatic wall. The boundary conditions in z direction are not relevant for this 2D example and are realized as periodic boundaries.

Suppose we want to change the wall boundary condition from adiabatic to isothermal.

- This is done by first defining a second RefState. Add to the parameter file the second of the following lines

  ```
  RefState=(/1.,0.990268069,0.139173101,0.,4.4642857/)
  RefState=(/1.,0.990268069,0.139173101,0.,4.9107143/)
  ```

  Note that in the second RefState the pressure was increased by 10%. From the ideal gas law, we then know that the temperature also is raised by 10%, since the density has been kept constant. Also note that the isothermal wall ignores the velocity components but only uses the thermodynamic quantities.

- Second, overwrite the initial wall boundary condition

  ```
  BoundaryName=BC_wall
  BoundaryType=(/4,2/)
  ```

Here, 4 implies the use of the isothermal boundary condition and 2 indicates the RefState to be used for the wall. Note that the temperature is set indirectly by the RefState given in terms of ρ and p.

To check the newly set boundary condition, run the code once more for one iteration. The output should now look like this

```
      Boundary in HDF file found |  BC_wall
                              was |  3 0
                        is set to |  4 2
.....................................................
         Name       Type      State      Alpha
       BC_wall          4          2          0
     BC_inflow          2          0          0
    BC_outflow          2          0          0
     BC_zminus          1          0          1
      BC_zplus          1          0         -1
```

Now the wall boundary condition is isothermal using the temperature defined by the second `RefState` listed in your parameter file. A complete overview over the boundary conditions and how to use them is given in Section 4.3.2.

Before starting your simulation, remember to disable the `maxiter` line again by setting it to -1.

6.3.6 Running the code

We proceed by running the code in parallel. For example using 4 processors, use the following command

```
mpirun -np 4 flexi parameter_flexi.ini
```

On a 2012 laptop with core i5 processor, this simulation takes about 40 minutes.

6.3.7 Evaluation of the lift and drag forces

The forces acting on the airfoil are one of the main desired output quantities from the simulation. They are calculated on the fly during runtime. The associated flags in the parameter file are

```
CalcBodyForces=T
WriteBodyForces=T
```

The first line activates the calculation of the forces at each `Analyze_dt`, the second line enforces output of the forces to a `.dat` file.

The body forces are a good measure for convergence. In the context of time-dependent flows this determines whether the solution has reached a quasi steady state. The figure 6.18 shows the x and y components of the force acting on the airfoil until `TEnd=10`.

Figure 6.18: Resulting forces on the airfoil up to $t = 10$.

The lift and drag coefficients can be easily calculated by rotating these forces from the computational reference frame to the one of the freestream.

From the forces, it is clear that the steady state has not yet been reached and the simulation must be run further. Before we proceed with the simulation, we will nonetheless examine the preliminary results to check the quality of the simulation.

6.3.8 Wall velocities

Due to the weak coupling between the grid cells and to boundaries, boundary conditions are enforced weakly, e.g. by applying a specific flux. This adds largely to the stability of the scheme. However, as a result the no-slip condition at the wall is not exactly fulfilled by the numerical solution. Rather, it is approximated as far as the resolution allows.

Evaluation of the velocity vector near the wall helps quantifying this error, which can be seen as a quality measure for the near wall resolution. In the parameter file, the computation and output to a `.dat` file of the average and extreme values of the wall velocity at every `Analyze_dt` are activated by setting

```
CalcWallVelocity=T
WriteWallVelocity=T
```

During the computation, we get output like the following:

```
 Wall Velocities (mean/min/max)  :
       BC_wall   2.737875251E-02   3.639096145E-04   6.228999317E
   -01
```

In our case, the wall velocity is on average at about 3% of the freestream velocity, reaching a peak of 60%. This peak typically occurs at the quasi-singularity at the trailing edge. To decrease this deviation from the theoretical no-slip condition, either the wall-normal mesh size must be decreased or the polynomial degree increased.

It is important to note that both of the above measures will, besides increasing the number of degrees of freedom, *decrease the time step*, which directly affects the computational time. Thus, it is important to achieve an acceptable trade-off between the acceptable error and the computational time.

In this tutorial, the observed slip velocity is deemed uncritical and we proceed with the same resolution.

6.3.9 Visualization

To visualize the solution, the *State*-files must be converted into a format suitable for **Paraview**. Issue the command

```
mpirun -np 4 posti_visu parameter_postiVisu.ini
   parameter_flexi_navierstokes.ini
   NACA0012_Re5000_AoA8_State_0000000.0*
```

to generate the corresponding *vtu*-files using supersampling with $(NVisu + 1)^3$ points in each cell. The *vtu* file can then be loaded into **Paraview**. Figure 6.19 shows a visualization of the density at $t = 10$. The levels of $0.99 < \rho < 1.01$ are chosen to reveal the acoustic radiation from the airfoil.

Figure 6.19: Contours of density with $0.99 < \rho < 1.01$ at $t = 10$ for the flow around a NACA 0012 airfoil.

The large scale vortex shedding of the wake due to the high angle of attack is clearly visible. Acoustic radiation from the airfoil can also be observed.

Now, a problem becomes apparent: the vortex street propagating towards the outflow boundary results in a second, artificial acoustic source at the outflow boundary. This is one of the fundamental problems in direct aeroacoustic computations. Before we proceed with the simulation, we will now make use of the sponge zone functionality of **FLEXI** to remove this artificial source.

6.3.10 Remove outflow reflections using the sponge zone

The sponge zone introduces a dissipative source term to the discrete operator, which is only active in a user-specified region, typically upstream of the outflow boundary. We use the sponge zone to dampen the vortices convected downstream before they hit the outflow boundary. See (Flad et al. 2014) for the background of our sponge zone implementation.

In order to activate the sponge zone, set

```
SpongeLayer=T
```

in the parameter file.

Most of the other parameters are already set for the current simulation:

Table 6.9: Sponge settings

Variable	Description	Value
SpongeShape	Set shape of sponge: 1: cartesian	1
damping	Damping factor of sponge	1.
xStart	Coordinates of start position of sponge ramp (for SpongeShape=1)	(/2.0,0,0/)
SpongeDistance	Set shape of sponge: 1: Cartesian sponge region.	3.0
SpongeDir	Direction vector of the sponge ramp (for SpongeShape=1)	(/1,0,0/)
SpongeBaseFlow	Type of baseflow to be used for sponge 4: moving average (Pruett baseflow)	4
tempFilterWidth	Temporal filter width used to advance Pruett baseflow in time.	2.0
SpongeViz	Write a visualization file of the	T

The source term is of the form

$$\tilde{U}_t = U_t - d\sigma(\vec{x})\,(U - U_B) \tag{6.5}$$

First, `damping` determines the strength of the source term, i.e. d in the above equation. It is dependent on the mean convection velocity, the desired amount of amplitude reduction and the thickness of the sponge zone. Typically, some trial and error is necessary to obtain an appropriate value. In non-dimensional calculations, i.e. velocity and length scale are of $\mathcal{O}(1)$, $d = 0.1...2$.

Ramping of the source term from 0 is necessary to avoid reflections at the sponge interface. If such reflections occur, it is necessary to choose a thicker sponge ramp so that the source term is ramped up less steeply. We choose a parallel ramp by setting `SpongeShape=1`. The ramp's starting position, thickness and direction are set by `xStart`, `SpongeDistance` and `SpongeDir`, respectively. These parameters govern the shape function $\sigma(\vec{x})$ which smoothly ramps the source term from 0 to 1.

It is important to emphasize that the regions where the source term is active should not be interpreted physically. They should be considered a part of the boundary condition. Likewise, they must be placed sufficiently far downstream of the airfoil such that they do not influence the near field solution. With the chosen settings, the sponge zone starts one chord behind the airfoil and is ramped up to 1 at the outflow boundary, located 4 chords behind the airfoil.

In order to visualize the ramping function $d\sigma(\vec{x})$, set `SpongeViz=T`. For that purpose, change the project name to e.g. "test" and set `maxiter=1` to immediately stop the simulation after one iteration. This will output a .vtu file ready for visualization. After that, don't forget to change back the project name and reset `maxiter=-1`.

Now, we have to choose the desired baseflow (U_B). For the current configuration, the moving average (`SpongeBaseFlow=4`) is appropriate. It produces a mean field slowly progressing in time, which adapts to the potential flow around the airfoil. The parameter `tempFilterWidth` determines the effective time window for the moving average. It should be chosen somewhat larger than the largest time scales to be damped. Its value of 2.0 here is chosen based on the frequency of the oscillations in the body forces seen in Figure 6.18.

The moving average base flow needs an initial value. One option is to provide an initial flow field from a file using the `SpongeBaseflowFile` parameter. If this parameter is not set, the code will initialize the base flow with the same values as the solution itself. Thus, in the case of a fresh computation, the base flow will be initialized with the `IniExactFunc` used to initialize the solution. In the case of a restart, like in the present tutorial, the base flow will be initialized with the state file provided for the restart of the simulation.

Note that using the moving average baseflow, the code will dump additional **baseflow*.h5* files to the hard drive, which are necessary to restart the simulation with the corresponding base flow from a given time. If these files are available and exhibit the same project name as the current simulation, the code automatically detects the right base flow file to initialize the base flow at the restart time. However, if in that situation `SpongeBaseflowFile` is still provided in the parameter file, the moving average will be *restarted using the flow field provided in the parameter file*, which may not be desirable.

6.3.11 Restarting the simulation

If the simulation run is interrupted or if you decide to run the simulation further than `TEnd=10`, **FLEXI** can easily be restarted. In the current setting, the code dumps the solution in 0.1 time units intervals. Since we want to progress our simulation further, set `TEnd=25` in the parameter file. Since we have now turned on the sponge zone, it is also advisable to modify the project name, i.e.

```
ProjectName=NACA0012_Re5000_AoA8_SP
```

If the state files are in the current folder, restart the simulation as follows

```
mpirun -np 4 flexi parameter_flexi.ini
   NACA0012_Re5000_AoA8_State_0000010.000000000.h5
```

Note that it is also possible to change the polynomial degree of the simulation during restart. This can be used to generate a rough initial solution at low N and then switch to a high quality approximation at higher N later. The code will automatically project the solution onto the new polynomial basis at startup. Restart is however only possible using the *same mesh file*.

Once the simulation is done, visualize the results again following the instructions in Section 6.3.9.

The following two figures 6.20 and 6.21 show the resulting instantaneous density contours with (Figure 6.20, $t = 25$) and without sponge zone (Figure 6.21, $t = 20$). Clearly, the source at the outflow is damped to uncritical levels. Fine-tuning of the sponge parameters may improve the results.

Another look at the body forces indicates that a quasi-steady state has been reached.

6.3.12 Two-dimensional computation

The laminar flow around this airfoil is in nature two-dimensional. Nonetheless, we ran this simulation using a three-dimensional code by imposing periodic boundary conditions and only one mesh element in the spanwise direction. This means we

- Compute one unecessary variable (momentum in spanwise direction),
- Compute three-dimensional fluxes for all variables,
- Compute one unecessary gradient,
- Use several degrees of freedom in the spanwise direction due to the high order ansatz in each element.

To circumvent this, **FLEXI** provides the option to perform truely two-dimensional calculations. To enable this, we need to set the flag FLEXI_2D to ON during configuration. **FLEXI** will then perform two-dimensional computations, but you will need to provide a mesh that

Figure 6.20: Contours of density with $0.99 < \rho < 1.01$ at $t = 25$ for the flow around a NACA 0012 airfoil with moving average sponge zone.

Figure 6.21: Contours of density with $0.99 < \rho < 1.01$ at $t = 20$ for the flow around a NACA 0012 airfoil without sponge zone.

Figure 6.22: Resulting forces on the airfoil up to $t = 25$.

consists of only one element in the third dimension. Only then will you be able to start the computation. Since this is already the case for the mesh used in this tutorial, we can start a two-dimensional computation right away after we compile our code with FLEXI_2D set to on. So switch to your build directory, set the CMAKE option FLEXI_2D to on, configure and compile your code once more. Then switch back to the tutorial folder.

You can now immediately run the code again using the same parameter file, but your new executable. All the options are the same for two-dimensional and three-dimensional computations. For compatibility reasons, you will have to specify vectorial parameters using three dimensions (e.g. the momentum in the `RefState`), but the third dimension will be ignored. Pay attention to how much faster your code will run when using the two-dimensional version.

By default the *State*-files will be written like three-dimensional data by extruding the solution in the third dimension so the files will be compatible with all already existing post-processing tools. You can set the option `Output2D` to TRUE if you want to write two-dimensional files, thus reducing the size of the output considerably. Also keep in mind that no matter if two- or three-dimensional, the number of dimensions of arrays will always be the same (one dimension simply beeing of size 1 for two-dimensional computations) and also the number of variables will be the same. This means we will still have a momentum in the third dimension, which is simply set to zero (no calculations will be done for this momentum).

6.3.13 Using record points

Let's assume we are interested in the temporal evolution of the flow variables in the vicinity of the profile's upper and lower surfaces. Using the *State*-files, we will always have the whole flow field. If we want to have a very fine temporal resolution or our field is very large, the cost of writing and storing these files can get prohibitive. As a remedy, **FLEXI** provides an option to sample specific points in the flow field at a very fine temporal resolution (down to a single time step) and only output those specific points. We call these points *record points*.

If record points should be used, three distinctive steps have to be taken:

- Create the record points themselves
- Run a simulation with active record points
- Visualize the results

The first and third step are performed with pre- and postprocessing tools respectively, while a simulation using record points simply requires to set the specific options in **FLEXI**.

6.3.13.1 Prepare the record points

The coordinates of the record points are defined using a **POSTI** tool called *preparerecordpoints*, which is build when the respective CMake option is set. The tool takes a single parameter file as an input, so it can be run like

```
posti_preparerecordpoints parameter_recordpoints.ini
```

A sample parameter file is included in the NACA0012 tutorial folder. You need to specify the mesh that should be used to create the record points and a project name. The options *NSuper* and *maxTolerance* control the algorithm that tries to find the record points in the mesh - a good default for *NSuper* is to set it to at least twice of the polynomial degree of your mesh. The main part of the parameter file is the actual definition of the record points. They are created in so called sets, and each set is part of a named group. In the example, we have two sets - one for the suction side and one for the pressure side of the airfoil - each assigned to a separate group. There are several different types of sets available, which can be used to create single points, lines or planes of record points. For the NACA example, we are using the *boundary layer plane* type of set. This special type of plane is defined by a spline which is projected onto the nearest boundary. Along this projected line, a user-specified number of points is distributed in an equidistant way. The plane is then created by following the normal vector of the boundary at each of the points for a specified distance. Along the normal direction, the record points can be stretched to cluster them near a wall.

The definition of one of the groups looks like this:

```
GroupName           = suctionSide
BLPlane_GroupID     = 1
BLPlane_nRP         = (/20,30/)
BLPlane_nCP         = 2
BLPlane_CP          = (/0.9,0.014,0.5/)
BLPlane_height      = 0.05
BLPlane_CP          = (/0.999,0.001,0.5/)
BLPlane_height      = 0.05
BLPlane_fac         = 1.04
```

The first parameter creates a group which is used for identification purposes - each set needs to be assigned to a group. Then, the new set is defined by setting the respective *GroupID* option, in this case *BLPlane_GroupID*. The ID is set to 1 to assign it to the first group. Following this general definitions, the actual parameters of the set follow. For the boundary layer plane, we first define the number of the points used in wall-tangential and wall-normal direction using the *BLPlane_nRP* option. The next six options define the spline that is projected onto the boundary. First we set the number of control points in the boundary, then define the coordinates of each of the points and the height in the wall-normal direction at this point. The last option defines the stretching that is used in the wall-normal direction.

To find out how the other types of sets are defined, have a look at the descriptions of their parameters.

If you now run the preparerecordpoints tool using the syntax given above, the algorithm will first calculate the physical coordinates of the recordpoints following your definitions. Then, it will attempt to identify the element in the mesh that contains each record point and the coordinates in the reference element therein. This is done by inverting the mapping from reference to physical space using Newton's method. This information is needed to later interpolate the polynomial solution to exactly the positions of the record points. The results are then written to a file called 'ProjectName_RPSet.h5'. If you set the *doVisuRP* option to true, you will also get a visualization of the record points that can be viewed in ParaView.

6.3.13.2 Using record points

To actually use the recordpoints during a simulation, you simply need to set a few options in the **FLEXI** parameter file. The import ones are the following:

```
RP_inUse              =                   T
RP_DefFile            = NACA0012_RPSet.h5
RP_SamplingOffset     =                   1
```

With the first option, the general usage of the record points system can be turned on or off. The *RP_DefFile* option must then be set to the name of the file that has been created using the preparerecordpoints tool. You can specify that the solution should be sampled at every *RP_SamplingOffset* timestep.

If you now run the simulation, besides the *State*-files additional *RP*-files will be created, containing the value of the conservative variables at each record point at every sampling time step.

6.3.13.3 Visualizing the solution at the record points

The visualizerecordpoints tool is used to post-process the recordpoints data. It can be used to merge several of the *RPSet.h5* files together and create a longer timeseries. Besides simply visualizing the timeseries, there are a multitude of post-processing options available. You can use the time series to determine the mean and fluctuating part of the solution or perform spectral analysis using FFTs. For boundary layer planes, different types of turbulent quantities like the skin friction can be directly computed.

As an example, we want to calculate the mean flow $\bar{U}$ and the temporal fluctutations U' at our two record points planes. Again, a sample parameter file for the tool is given in the tutorial folder. The options set are

```
ProjectName           = NACA0012
GroupName             = suctionSide
GroupName             = pressureSide
RP_DefFile            = NACA0012_RPSet.h5
OutputTimeAverage     =       T
```

```
doFluctuations        =        T
```

where besides setting a name for the project and giving the path to the file that contains the record point definition, we define the name of the groups we want to visualize. They must be the ones we defined when using the preparerecordpoints tool. Regarding actual evaluation, we want to calculate the temporal average and the fluctuations, simply calculated as $U' = U - \bar{U}$. We did not specify any variables that we want to visualize, in that case all conservative variables that are stored in the *RPSet.h5* files will be used. If you want to visualize a specific or a derived variable (e.g. the pressure), it can simply be set in the parameter file like this:

```
VarName               = Pressure
```

You can run the tool using

```
posti_visualizerecordpoints parameter_visualizeRecordpoints.ini
   NACA0012_Re5000_AoA8_RP_*
```

This will take all the time samples recorded during our simulation as input. For each of the planes, a separate *.vts* file contaning the temporal average will be written. The fluctuations are time-resolved data, and due to limitations in the VTK file format every time step has to be written to a single file. To avoid creating a lot of files in the working directory, those will be stored in a subfolder called *timeseries*. In the working directory, *.pvd* files for the fluctuations are created. These files can be opened with ParaView and will contain the complete time series with the correct time value.

6.4 Convergence Test

In this tutorial the order of convergence for **FLEXI** is computed. The procedure is fully scripted, such that in the end a number of runs have been performed on a variation of grids or a variation of polynomial degrees and the order of convergence is computed automatically. A plot of the corresponding L2 error norms is produced and copied into the directory where the convergence test is executed from. Note, the script is written in *Python 2.7*.

The convergence test is separated into two parts, first an inviscid test where the order of convergence for the advective part, i.e. the Euler equation without viscous fluxes, is computed. Then a viscous convergence test calculates the order of convergence with consideration of the viscosity.

In this tutorial, the following topics are dealt with:

- Calling **FLEXI** using a Python script
- Compute the order of convergence
- Switch from a Navier-Stokes simulation to an Euler simulation
- Treatment of source terms in **FLEXI**

- Execution of **FLEXI** with Mortar meshes

6.4.1 Manufactured Solution

The basic principle of computing the order of convergence of **FLEXI** is to apply a benchmark test where an exact analytical solution is known. We apply here the method of manufactured solutions. A detailed description is found in Roache (Roache 2001). In general, a smooth function is proposed as solution to the equation system. Since the proposed function is generally not a solution to the equation system, a source term is calculated to force the corresponding solution. Therefore, the function is derived analytically and inserted into the equation system, cf. for the continuity equation

$$\rho_t + (\rho u)_x = Q(x,t) \quad \text{with} \quad \rho = A + \sin(B(x,t)) \quad \text{and} \quad u = \text{const.} \tag{6.6}$$

The source term has to be added accordingly within the time integration loop of the flow solver. In **FLEXI**, sine waves in the density are advected on a constant velocity field. For the different equation systems, i.e. Euler or Navier-Stokes, different source terms have to be considered.

6.4.2 Inviscid Convergence Test

Copy the *convtest* tutorial folder

```
cp -r $FLEXI_TUTORIALS/convtest .
cd convtest
```

6.4.2.1 Compiler Options

Since this test case is valid for the Euler equations only, the code has to be compiled without parabolic fluxes. Hence, make sure to compile **FLEXI** with the cmake options in the following table

Table 6.10: Cmake options for the convergence test simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	OFF	
LIBS_USE_MPI	ON	optional

6.4.2.2 Mesh Generation with HOPR

The mesh files for this tutorial can be found in the tutorial directory. Parameter files are provided to produce the meshes using **HOPR**. With *parameter_hopr.ini* conform meshes are created and the file *parameter_mortar_hopr.ini* generates non-conform meshes.

6.4.2.3 Manufactured Solution For The Inviscid Case

The manufactured solution is

$$\rho = A * (1 + B * \sin(\Omega * |\underline{x} - \underline{v}t|)) \quad \text{with} \quad A, B, \Omega, \underline{v} = const. \tag{6.7}$$

The advantage is that for the Euler equation, the resulting source term is zero, $Q(x,t) \equiv 0$.

As mesh, a Cartesian box is used with periodic boundaries. The mesh and the corresponding solution are shown in following figure:

Figure: Mesh and flow field solution of the density. View in x-y-plane.

To investigate the order of convergence of a given polynomial degree N, the mesh resolution has to increase. We provide meshes with 1, 2, 4 and 8 elements in each spatial direction. They are provided in the tutorial directory with an according parameter file for the preprocessing tool **HOPR**.

6.4.2.4 Flow Simulation with FLEXI

The inviscid convergence test is run from the parameter file *parameter_convtest_flexi.ini*. Essentially, any valid parameter file can be used, as a manufactured solution is simulated. This allows to test the various methods and features of the code and investigate their order of convergence. However, for this tutorial we restrict the parameter file to a very simple baseline test case. The following entries have to be made

IniExactFunc = 2 : The manufactured solution is applied by means of the exact function. This function is used to initialize **FLEXI** and can in general be used for Dirichlet boundary conditions. Here, we apply only periodic boundary conditions.

N_Analyze = 10 : The number of interpolation nodes for the analyze routines, needed for calculating the error norms. We suggest at least $N_Analyze = 2N$.

AdvVel = (/0.3,0.,0./) : The constant velocity vector $\underline{v}$

CalcErrorNorms = T : Flag to turn on the calculation of L2 and L_inf error norms

ProjectName = ConvTest : Project name used to name the output files of **FLEXI** and the convergence test scripts.

6.4.2.5 Numerical settings

The default settings for the time integration are displayed in table 6.11.

Table 6.11: Numerical settings for time integration

Variable	Description	Value
tend		0.5
Analyze_dt		0.5
nWriteData		1
CFLscale		0.9
DFLscale		0.9

The remaining numerical settings necessary, e.g. the polynomial degree and the mesh filename are set via the script file. The script can be found in the directory

```
$FLEXIROOT/tools/convergence_test
```

Two scripts are provided, the file *convergence_grid* calculates the order of grid convergence for a given polynomial degree N with increasing mesh resolution. The file *convergence* calculates spectral convergence on a given mesh with increasing polynomial degree. Both scripts are written in Python 2.7. In the first case of grid convergence, the polynomial degree and the set of meshes can be adjusted. Here we choose a polynomial degree of 3, i.e. the

theoretical order of convergence is $N + 1 = 4$. The spectral convergence is calculated for polynomials of degree $N \in [1, 10]$ on a mesh with 4 elements in each spatial direction.

The command

```
$FLEXIROOT/tools/convergence_test/convergence_grid $FLEXIDIR/bin/
   flexi parameter_convtest_flexi.ini --gnuplot
```

runs the code. The standard output of **FLEXI** is written into the logfile *ConvTest.log*. An ASCII file *ConvTest_convfile_grid.csv* is written that includes all L2 and L_inf error norms for the state vector U for all meshes and the corresponding orders of convergence. Furthermore, a PDF file *ConvTest_convtest_grid.pdf* is generated that plots the L2 error of the momentum in x-direction against the number of elements of the meshes. Another curve represents the theoretical order of convergence for the chosen polynomial degree.

Spectral convergence can be investigated using the command

```
$FLEXIROOT/tools/convergence_test/convergence $FLEXIDIR/bin/flexi
   parameter_convtest_flexi.ini --gnuplot
```

Corresponding files are produced, where *_grid* is replaced by *_N*. Figure 6.23 shows the result for grid convergence.

Figure 6.23: Plot of grid convergence

Figure 6.24 shows the result for spectral convergence.

Figure 6.24: Plot of spectral convergence

6.4.3 Viscous Convergence Test

The second convergence test includes the viscous terms. Make sure, **FLEXI** is compiled with the setting listed in the table 6.12 to include parabolic fluxes.

Table 6.12: Cmake options for the convergence test simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional

For this case, another manufactured solution is chosen

$$\rho = 2 + A * \sin(\Omega * |\underline{x}| - v\pi t) \quad \text{with} \quad A, \Omega, v = const. \tag{6.8}$$

The same function is applied to the momentum in all spatial directions. The mass specific total energy in this case is $\rho e = \rho\rho$. This manufactured solution has a non-zero source term. In **FLEXI**, this source term is added in the routine *CalcSource* in the file

```
$FLEXIROOT/src/equations/navierstokes/idealgas/exactfunc.f90
```

Note that this manufacture can also be solved without considering the viscous terms. In this case the source term does not vanish.

The parameter file for this case is *parameter_convtestvisc_flexi.ini*. Some modifications have to be made, compared to the inviscid convergence test. The correct exact function for this test case is *IniExactFunc = 4*. Note, that a viscosity has to be prescribed. Our default value for this case is $Mu0 = 0.03$.

Execution of the convergence tests is analogously to the inviscid case, e.g.

```
$FLEXIROOT/tools/convergence_test/convergence_grid $FLEXIDIR/bin/
   flexi parameter_convtestvisc_flexi.ini --gnuplot
```

6.4.4 Execution of the Convergence Test with non-conform meshes

FLEXI is capable of using non-conform meshes with mortar interfaces. To illustrate the execution of **FLEXI** with these meshes, four mortar meshes are provided for the convergene test. Their file names include the term *MORTAR*. To use the convergence test script, just open the script file at *$FLEXIROOT/tools/convergence_test/convergence_grid* and exchange the file names of the meshes. The script can then be run similar as described above.

6.4.5 Execution of Convergence Test Outside of Tutorials

The convergence test scripts are provided in the directory

```
   $FLEXIROOT/tools/convergence_test
```

including the Python script to execute **FLEXI**

```
   $FLEXIROOT/tools/convergence_test/execute_flexi.py
```

6.5 Taylor Green Vortex

This tutorial describes how to set up and run the basic test case for turbulent flows, the Taylor-Green-Vortex (TGV) - see e.g. (Gassner and Beck 2013). We will learn how to avoid catastrophic failure of the code due to non-linear instabilities. This is done by using polynomial de-aliasing. In a second step we add the sub grid scale model of Smagorinsky. The tutorial assumes that you are familiar with the general **FLEXI** and **HOPR** work flow (please finish the previous tutorials first if this sounds strange to you).

6.5.1 Flow description

The initial condition to the (TGV) is a sinus distribution in the u and v velocity components. This leads to rapid production of turbulent structures, after a short initial laminar phase. While the test case is incompressible in principle, we solve it here in a compressible setting. The chosen Mach number with respect to the highest velocity in the field is 0.1. The Reynolds number of the flow is defined as $1/\nu$. The domain is set up as a triple periodic box with edge length 2π.

6.5.2 Compiler options

Make sure that **FLEXI** is compiled with the cmake options listed in the following table.

Table 6.13: CMake options for the cavity simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	

Option	Value	Comment
LIBS_USE_MPI	ON	optional
FLEXI_EDDYVISCOSITY	ON	optional
FLEXI_TESTCASE	taylorgreenvortex	optional

For others you may keep the default values. Compile the code.

6.5.2.1 Mesh Generation with HOPR

We use a mesh with 4 cells per direction for the tutorial. In case you want to generate other meshes the parameter file for **HOPR** is included in the tutorial directory (parameter_hopr.ini), the default mesh is included. Using 4 cells with a polynomial degree of $N = 7$, means we use the typical large eddy setup of 32 DOF per direction.

6.5.3 Tutorial - Flow at $Re = 1600$

Copy the `tgv` tutorial folder to your working directory

```
cp -r $FLEXI_TUTORIALS/taylorgreenvortex .
```

Step into the folder. In case you do not want to generate the mesh files yourselves, the meshes have already been provided.

6.5.3.1 Preparing the Flow Simulation with FLEXI

The simulation setup is defined in *parameter_flexi.ini*. To get help on any of the parameters listed therein, you can run **FLEXI** from the command line by typing

```
flexi --help
```

The parameters in the file are grouped thematically, however, this is not mandatory. All lines starting with a "!" are comments.

6.5.3.1.1 Output The test case has its own analyze output (PROJECTNAME_TGVAnalysis.csv), that we will use. We don't look at flow visualization in this tutorial. Besides other interesting quantities, the file contains the incompressible dissipation rate. This is the resolved dissipation of the gradient field, computed as the integral over the domain of the strain rate tensor norm $S_{ij}S_{ij}$, times viscosity times 2. It is stored in the second column of the file. We will use this quantity in the tutorial to verify your results.

6.5.3.1.2 Interpolation / Discretization parameters

```
! ================================================ !
! INTERPOLATION
! ================================================ !
N               = 7
```

The parameter *N* sets the degree of the solution polynomial, e.g. in this example, the solution is approximated by a polynomial of degree 7 in each spatial direction. This results in $(N+1)^3$ degrees of freedom for each (3D) element. In general, *N* can be chosen to be any integer greater or equal to 1, however, the discretization and the timestep calculation has not extensively been tested beyond $N \approx 23$. Usually, a good compromise of performance and accuracy is found for $N \in [3, .., 9]$.

To apply polynomial de-aliasing there are the following options:

```
! ================================================ !
! OVERINTEGRATION (ADVECTION PART ONLY)
! ================================================ !
OverintegrationType=0  ! 0:off 1:cut-off filter
                       ! 2: conservative cut-off
NUnder          = 7    ! specifies effective polydeg
                       ! (modes > NUnder are thrown away)
                       ! for types 1 and 2
```

FLEXI has three ways of doing polynomial de-aliasing. Mode 0: don't do it. Mode 1: a filter is applied to the time-update $(J * U_t)$. The filter is formulated as a Galerkin projection of degree N to $NUnder$, the effective resolution is thus $NUnder$. Mode 2: in principle identical to Mode 1, but takes into account non-linear metric terms. For the linear mesh of this tutorial the result is identical, while Mode 2 is slightly more computational expensive, so we omit it.

For **FLEXI** we can run under-resolved computations without sub grid scale model. The only artifical dissipation is then provided by the Riemann solver used for the inter-cell fluxes. You can change the Riemann solver to see the effect with the following parameters:

```
! ================================================ !
! Riemann
! ================================================ !
Riemann=  RoeEntropyFix ! Riemann solver to be used:
                        ! LF, HLLC, Roe,
                        ! RoeEntropyFix, HLL, HLLE, HLLEM
```

To add Smagorinsky's model set the following parameter to 1, here CS is the Smagorinsky constant usually chosen around 0.1 for isotropic turbulence (such as TGV).

```
! ================================================ !
! LES MODEL
! ================================================ !
eddyViscType = 0       ! Choose LES model, 1:Smagorinsky
```

```
    CS = 0.02                ! Smagorinsky constant
    PrSGS = 0.6              ! turbulent Prandtl number
```

6.5.3.2 Running the Simulation and Results

The command

```
flexi parameter_flexi.ini > std.out
```

runs the code and dumps all output into the file *std.out*. If you wish to run the code in parallel using MPI, the standard command is

```
mpirun -np XX flexi parameter_flexi.ini > std.out
```

where XX is an integer denoting the number of processes to be used in parallel. Note that **FLEXI** uses an element-based parallelization strategy, so the minimum load per process/core is *one* grid element, i.e. do not use more cores than cells in the grid! **FLEXI** writes a TGV analyze file (PROJECTNAME_TGVAnalysis.csv). You can use yor favorite plotting programm to visualize the ASCII data. Using gnuplot you can create plots with the following syntax:

```
gnuplot
set key autotitle columnhead
set datafile separator ","
plot "PROJECTNAME_TGVAnalysis.csv" u 1:2 w l
```

Where you replace PROJECTNAME with the projectname you defined in the parameter_flexi.ini file.

6.5.3.3 Part I: crashing simulation

First we run **FLEXI** without overintegration/de-aliasing. We will find that the code crashes, once scale production becomes relevant. You can compare your result to the plot in the tutorial folder

6.5.3.4 Part II: Overintegration

We now use overintegration by changing the respective settings in the parameter_flexi.ini file as described above. For `Overintegration==1` set $N = 11$ and $NUnder = 7$. You can compare your result to the plot below.

6.5.3.5 Part III: Explicit LES model

To see the effect of adding explicit eddy viscosity we activate the LES model (Smagorinsky) as described above. To obtain the reference result of the following plot set $CS = 0.1$. Don't forget to switch overintegration of again and set polynomial degree to $N = 7$. Feel free to play around with the constant, have fun!

6.6 SOD Shock tube

The Sod shock tube example (Sod 1978) is one of the most basic test cases to investigate the shock capturing capabilities of a CFD code. A initial discontinuity is located in the middle of the one dimensional domain $[0, 1]$. The left and right states are given by $\rho = 1, v = 0, p = 1$ and $\rho = 0.125, v = 0, p = 0.1$. These states are already set as `RefState` in the the *parameter_flexi.ini* file.

Copy the `sod` tutorial folder

```
cp -r $FLEXI_TUTORIALS/sod .
```

6.6.1 Mesh Generation with HOPR

The mesh files used by **FLEXI** are created by supplying an input file *parameter_hopr.ini* with the appropriate information.

```
hopr parameter_hopr.ini
```

This creates the mesh file *SOD_100_mesh.h5* in HDF5 format.

6.6.2 Flow Simulation with FLEXI

This example requires the Finite Volume shock capturing and the Euler equations. In this tutorial we will investigate the shock capturing based on switching the DG representation to FV sub-cells. Therefore set the following options in cmake:

```
FLEXI_FV           SWITCH
FLEXI_PARABOLIC    OFF
```

and recompile the **FLEXI** code. The simulation setup is defined in *parameter_flexi.ini* and includes options for the Finite Volume shock capturing.

Option	Value	Description
IndicatorType	Persson	
IndVar	1	first conservative (density) used for indicator evaluation
IndStartTime	0.001	until this time FV is used in the whole domain
FV_LimiterType	MinMod	
FV_IndUpperThreshold	-3.	upper threshold (if IndValue above this value, switch to FV)
FV_IndLowerThreshold	-4.	lower threshold (if IndValue below this value, switch to DG)

Explanation of the Finite Volume specific options:

- `IndicatorType`: specifies the indicator function that is used to detect DG elements that contain a discontinuity. The `Persson` indicator (Persson and Peraire 2006) is an element local indicator, which compares the different modes of the DG polynomial. If the amount of solution in the highest mode compared to the amount in the lower modes is high the DG polynomial may oscillate. All indicator functions return a high value for trouble elements and a low value for smooth elements.
- `IndVar`: Variable that is used to evaluate the indicator function. Here the density is used. In general the pressure (6) is a good choice.
- `IndStartTime`: Until this time the actual indicator function is overwritten by a very high value to force the use of FV elements in the hole domain. This is necessary if initial discontinuities (like in this example) are placed perfectly at the element boundaries. In this case an element local indicator (like Persson) can not detect the discontinuity.
- `FV_LimiterType`: Limiter of the second order reconstruction.
- `FV_IndUpperThreshold` and `FV_IndLowerThreshold`: These two threshold values are used to decide in which elements the DG method and where the FV sub-cell scheme should be used. In general a single threshold could be sufficient. If the indicator value

raises above this threshold the element used the FV scheme, and if the value is below the DG scheme is used. Practice shows that this could lead to an ongoing switching between the two schemes. Therefore a DG element is switched only to FV if the indicator value gets greater than the upper threshold, but it does not switch back to DG immediately if the value falls below this threshold. This only happens if the indicator value falls below the lower threshold.

The command

```
flexi parameter_flexi.ini
```

runs the code and generates 5 state files **sod_State_TIMESTAMP.h5** for $t = 0.0, 0.05, 0.10, 0.15, 0.20$. To visualize the solution, the *State*-files must be converted into a format suitable for **ParaView**. Execute the command

```
posti_visu parameter_postiVisu.ini parameter_flexi.ini
   sod_State_0000000.*
```

to generate the corresponding *vtu*- and *vtm*-files, which can then be loaded into **ParaView**. There are two types of *vtu*-files, which contain either the DG or the FV part of the solution. The *vtm*-files combine the DG and FV *vtu*-file of every timestamp. Load the *vtm*-files into **ParaView**.

Since this example is a 1D test case use the `Plot Over Line`-Filter in **ParaView** and select the `X Axis` as line. The result should look like in figure 6.25.

6.7 Double Mach Reflection

The Double Mach Reflection is a classical test case to investigate the abilities of a numerical scheme to represent shock and contact discontinuities. It was invented by Woodward and Colella (Woodward and Colella 1984). A Mach 10 oblique shock wave hits a reflecting wall and the initial conditions are given by the Rankine-Hugoniot conditions

$$(\rho, v_1, v_2, p) = \begin{cases} (8.0, 8.25 \cdot cos(30°), -8.25 \cdot sin(30°), 116.5) & x < x_0 + \sqrt{\frac{1}{3}}y \\ (1.4, 0.0, 0.0, 1.0) & x \geq x_0 + \sqrt{\frac{1}{3}}y \end{cases}, \tag{6.9}$$

where $x_0 = \frac{1}{6}$ is the start of the wall and the computational domain is $\Omega = [0, 4] \times [0, 1]$, which is discretized by an equidistant cartesian mesh.

Copy the `dmr` tutorial folder

```
    cp -r $FLEXI_TUTORIALS/dmr .
```

Figure 6.25: Solution of the Sod shock tube example at $t = 0.2$.

6.7.1 Mesh Generation with HOPR

The mesh files used by **FLEXI** are created by supplying an input file *parameter_hopr.ini* with the appropriate information.

```
hopr parameter_hopr.ini
```

This creates the mesh file *DMR_mesh.h5* in HDF5 format.

6.7.2 Flow Simulation with FLEXI

This example requires the finite volume (FV) shock capturing. Two variants of the shock capturing are implemented in **FLEXI**, which are both based on the FV sub-cell technique. For this, the DG element is subdivided into finite volume sub-cells, where each sub-cell corresponds to one degree of freedom of the initial DG element. This allows to keep the total number of degrees of freedom in the simulation constant and to switch between the DG and the FV operator for each individual element. The first approach switches DG elements into the FV sub-cell representation such that each element is either represented by its DG solution and DG discretization operator or the element is represented by FV sub-cells, i.e. a piecewise constant representation, as detailed in (Sonntag and Munz 2017). The alternative is to blend the FV operator to the DG operator, while the solution itself always remains a DG polynomial. Please see (Hennemann et al. 2021) for more details. In the following, both approaches will be applied to simulate the Double Mach Reflection.

6.7.2.1 Using the Finite Volume Switching

First, set the option `FLEXI_FV=SWITCH` in the cmake configuration to enable the switching-based shock capturing. Additionally you should switch `FLEXI_PARABOLIC` off and recompile the **FLEXI** code. The simulation setup is defined in *parameter_flexi_switch.ini* and includes options for the shock capturing via switching.

Option	Value	Description
FV_LimiterType	MinMod	
IndicatorType	Jameson	
IndVar	6	Pressure is used for indicator evaluation
FV_IndUpperThreshold	0.010	upper threshold (if IndValue above this value, switch to FV)
FV_IndLowerThreshold	0.005	lower threshold (if IndValue below this value, switch to DG)
FV_toDG_indicator	T	enable an additional Persson indicator to check switch from FV to DG
FV_toDG_limit	-5.5	threshold for additional Persson indicator

Option	Value	Description
FV_IniSupersample	T	supersample initial solution for every FV sub-cell, since DG polynomial oscillates for initial solution

Explanation of the Finite Volume specific options (read also the explanations for the Sod shock tube in chapter 6.6.2):

- `IndicatorType`: The `Jameson` indicator is used to detect troubled cells, that should use the FV operator. This indicator is an adaption of the switching function of the Jameson-Schmidt-Turkel scheme (Jameson, Schmidt, and Turkel 1981) to Finite Volume sub-cells. In contrast to the Persson indicator this indicator is not element local and therefore more robust for travelling discontinuities.
- `FV_toDG_indicator`: if set to `T` (true) an additional Persson indicator for the switch from FV to DG is used. A FV element designated for the switch to DG is converted to DG and the Persson indicator is evaluated for this DG polynomial to test if the polynomial is oscillating. Only if this test is passed the element becomes a DG element. Otherwise it remains a FV element. This helps especially for all indicator functions acting on the FV sub-cells. Here the Jameson indicator only acts on the direct adjacent DOFs of a specific DOF, which does not allow to capture all high frequencies of a polynomial.
- `FV_toDG_limit`: Threshold for the additional Persson indicator. An element can only switch back to DG from FV if the indicator value on the DG representation is below this threshold.
- `FV_IniSupersample`: If this option is `F` (false) the solution is initialized as DG polynomials for all elements. The indicator function is evaluated to find all troubled elements, which are than converted to FV elements. This causes major trouble if discontinuities lay inside DG elements which leads to heavy oscillating polynomials. Converting these oscillations to FV may produce invalid solutions (i.e. negative density). Switching this option on (`T`) enables a super sampling of the initial solution for every FV sub-cell, which removes the mentioned problems with oscillating polynomials. The mean value of every FV sub-cell is computed by evaluating the initial solution in $(N+1)^3$ equidistant points inside the sub-cell and then taking the arithmetic mean value.

The command

```
flexi parameter_flexi_switch.ini
```

runs the code and generates 11 state files **dmr_SWITCH_State_TIMESTAMP.h5** for $t = 0.0, 0.02, \ldots, 0.20$. To visualize the solution, the *State*-files must be converted into a format suitable for **ParaView**. Execute the command

```
posti_visu parameter_postiVisu.ini parameter_flexi_switch.ini
   dmr_SWITCH_State_0000000.0*
```

to generate the corresponding *vtu*- and *vtm*-files, which can then be loaded into **ParaView**. There are two types of *vtu*-files, which contain either the DG or the FV part of the solution. The *vtm*-files combine the DG and FV *vtu*-file of every timestamp. Load the *vtm*-files into **ParaView**.

The result at $t = 0.2$ should look like in figure 6.26.

Figure 6.26: Distribution of DG and FV elements (top) and density (bottom) of Double Mach Reflection at $t = 0.2$.

6.7.2.2 Using the Finite Volume Blending

Next, we will investigate the blending approach. For this, set the option `FLEXI_FV=BLEND` in the cmake configuration to enable the blending-based shock capturing. Moreover, the FV blending requires to set `FLEXI_NODETYPE=GAUSS-LOBATTO`. Also enable the split-form DG with `FLEXI_SPLIT_DG=ON` and recompile **FLEXI**. More details on the split formulation are given later in Section 6.9. In this tutorial we use an entropy-stable split formulation to ensure the stability of the FV blending approach. For the FV sub-cell blending, the elements are not switched completely to the FV operator, but instead the DG operator R_{DG} and FV operator R_{FV} are blended as

$$R = \alpha R_{FV} + (1 - \alpha) R_{DG}$$

with the blending coefficient α. Instead of switching between a DG and a FV discretization, the blending allows a continuous transition between the DG and FV operators. The blending factor is computed based on the indicator proposed by (Hennemann et al. 2021), which is parameter-free and does not require any parameters to be tuned by the user.

There are some specific options to for the FV blending that can be set in the parameter file.

Option	Value	Description
FV_LimiterType	MinMod	
FV_alpha_min	0.01	All elements with a blending factor 'alpha' below this threshold are treated as pure DG elements. This improves the computational efficiency, since the FV discretization operator does not have to be computed for these element.
FV_alpha_max	0.5	Maximum blending factor 'alpha'. All blending coefficients exceeding this value are clipped to FV_alpha_max
FV_doExtendAlpha	T	If true, the blending factors are propagated to the neighboring elements
FV_alpha_ExtScale	0.5	Scaling factor by which the blending factor is scaled when propagated to its neighbor. Has to be between 0 and 1
FV_nExtendAlpha	1	Number of times this propagation of the blending should be performed. Higher values correspond to a wider sphere of influence

Similarly to the switching-based approach, the command

```
flexi parameter_flexi_blend.ini
```

runs the code and generates 11 state files **dmr_BLEND_State_TIMESTAMP.h5** for $t = 0.0, 0.02, \ldots, 0.20$. To visualize the solution, execute the command

```
posti_visu parameter_postiVisu.ini parameter_flexi_blend.ini
   dmr_BLEND_State_0000000.0*
```

to generate the corresponding *vtu*- and *vtm*-files, which can then be loaded into **ParaView**.

The result at $t = 0.2$ should look like in figure 6.27.

6.8 Linear Scalar Advection-Diffusion Equation

Besides the Navier-Stokes equations, **FLEXI** provides another equation system, the three-dimensional linear scalar advection-diffusion (LinAdvDiff for short) equation:

$$\frac{\partial \Phi}{\partial t} + \nabla \cdot (\mathbf{u}\Phi) = d\nabla^2\Phi,$$

where a scalar solution Φ is advected with the constant (three-dimensional) velocity $\mathbf{u}$ and is subjected to diffusion with a constant scalar diffusion coefficient d.

Figure 6.27: Distribution of the blending factor between the DG and FV operators `FV_alpha` (top) and density (bottom) of Double Mach Reflection at $t = 0.2$.

The LinAdvDiff equation is useful to test and develop features in a simple and computationally cheap environment as well as to investigate basic properties of the DG operator, as we will do in this tutorial.

6.8.1 Theoretical Background

The dispersion and dissipation properties of the DGSEM operator can be analysed (Gassner and Kopriva 2011) by looking at the evolution of solutions to the one-dimensional linear scalar advection equation (without physical dissipation, $d = 0$) when a wave with a specific wavenumber is initialised. You sould have a look in the paper if you are interested in the details of the analysis, here only a short summary is presented.

Under the mentioned simplifications, the equation to consider simply reads

$$\frac{\partial \Phi}{\partial t} + u \frac{\partial \Phi}{\partial x} = 0$$

and we consider a wave-like analytical solution on an infinite domain

$$\Phi(x, t) = e^{i(kx - \omega t)}$$

with the constant scalar transport velocity u, the angular frequency $\omega = ku$ and the wavenumber k.

We now assume to have a uniform mesh with a mesh size of Δx and seek numerical solutions of the form

$$\mathbf{\Phi}^l = \hat{\mathbf{\Phi}} e^{i(kl\Delta x - \omega t)}$$

where $\mathbf{\Phi}^l$ is a vector containing the degrees of freedom in cell l and $\hat{\mathbf{\Phi}}$ is a complex amplitude vector, both of size $N+1$. If we look at the matrix notation of the semi-discrete system (meaning only spatial discretization) in a single element, we can formulate an algebraic eigenvalue problem

$$\underline{\underline{A}}\hat{\mathbf{\Phi}} = \Omega\hat{\mathbf{\Phi}},$$

with $\Omega = \frac{\omega\Delta x}{a}$. The matrix $\underline{\underline{A}}$ contains the spatial discretization and depends on the non-dimensional wavenumber $K = k\Delta x$.

By looking at the solutions of this eigenvalue problem, one can obtain relations for the dissipation and dispersion behavior inherent of the (spatial) numerical scheme for different wavenumbers. In figure 6.28 we plot these relationships depending on the polynomial degree N using DGSEM with Gauss nodes for the so called physical mode. This mode is associated with the eigenvalue that follows the exact dispersion relation for the largest range of wavenumbers and also has the biggest influence on the overall numerical solution, at least for rather well-resolved waves.

Figure 6.28: Dispersion and dissipation relationship for $N = 1 - 10$ over the modified wavenumber K^*. For the dispersion, the dashed line gives the exact relation.

The quantities are normalized by the number of grid points, $K^* = \frac{K}{N+1}$ and $\Omega^* = \frac{\Omega}{N+1}$ to give a fair comparison between different polynomial degrees. This means that if $K^* = \pi$, we have two points per wavelength which is the theoretical minimum to resolve a wave given by the Nyquist-theorem while a normalized wavenumber of 0 indicates a constant solution.

As we can see, the dissipation properties of higher-order approximations are significantly improved compared to low-order schemes. They are able to preserve waves with higher frequencies without dissipating them. We also observe a sharp increase of the dissipation error in the high modes associated with higher-order approximations, which is one of the reasons the timestep is scaled down for larger N.

We will now try to show these properties by conducting some numerical experiments using the linear scalar advection equation.

6.8.2 Compiling FLEXI with LinAdvDiff

If you want to use the LinAdvDiff equations, you need to specify the equation system during the configuration. We are going to create a second `build`-folder to keep any existing **FLEXI** binaries that might have been compiled using the Navier-Stokes equation system.

```
cd $FLEXIROOT
mkdir buildLinAdv && cd buildLinAdv
cmake -DFLEXI_EQNSYSNAME=linearscalaradvection ../
make
```

Since we don't want to have diffusion in this tutorial we can either turn of the parabolic terms in general during configuration by using the following `cmake` command instead of the previous one:

```
cmake -DFLEXI_EQNSYSNAME=linearscalaradvection -DFLEXI_PARABOLIC=
   OFF ../
```

or simply specify a diffusion coefficient of 0 in the parameter file.

Of course you can also use `ccmake ../` and set the parameters in the interface. The ones necessary are listed in the table below.

Option	Value	Comment
FLEXI_EQYNSYSNAME	linearscalaradvection	
FLEXI_PARABOLIC	OFF	optional

Note: The path to the flexi binaries is now different than before. We therefore type it out in all commands in this tutorial. You can, of course, set a different alias for the path to this executable in your `~/.bashrc`.

6.8.3 Setup

We want to run numerical experiments and solve the one-dimensional linear scalar advection equation. As our initialization condition we choose a single wave with a specific angular frequency and observe the behaviour of this wave depending on the normalised non-dimensional wavenumber K^* and the polynomial degree N. Since the analysis in the previous section has been done on a infinite domain, we are going to make our computational domain large enough to neglect influences from the boundaries. An alternative would be to use periodic boundary conditions, but this would limit our choice of wavelengths since they have to fit in the domain and could also negatively influence the stability of the time discretization. When we conduct the experiments, we are going to focus our attention to a single element in the middle of the domain and observe the evolution of the wave in there.

6.8.3.1 Mesh

For this tutorial we are going to create a (quasi-)one-dimensional equidistant grid. We choose $\Delta x = 2$ which is equal to the reference element. We are going to make the computational domain rather long, so the boundary conditions will have no influence on our solution. Recall that for LinAdv information propagates with velocity u, so if our boundaries are a distance of L away from the cell we are looking at, we can compute solutions up to $t = uL$ without having any influence of the boundary conditions.

In our case we are creating a cartesian grid with $x \in [-61, 61]$ and 61 cells to reach our desired Δx. We will later focus on the single element in $x \in [-1, 1]$. To achieve a (quasi-)one-dimensional simulation, we choose periodic boundary conditions in $y-$ and $z-$direction. The boundary conditions in the x-direction will not matter, so we simply use Dirichlet-type BCs (BC type 2) with the analytical wave function as the boundary state. We specify this by setting the BC state to zero, which means that the initialization function will be used instead of a seperate function.

We provide a parameter file for **HOPR** to generate this mesh. It is located in the subdirectory `mesh` within the directory that belongs to this tutorial. Either run `hopr` with this parameter file

```
cd $FLEXIROOT/tutorials/linadv/mesh
hopr parameter_hopr.ini
```

or use the mesh file `CART_1D_mesh.h5` which is provided in this directory.

6.8.3.2 FLEXI parameters

A `parameter_flexi.ini` is provided in the directory `$FLEXIROOT/tutorials/linadv/`. We are going to discuss the parameters that are specific to the LinAdvDiff equation system and

the ones needed for this tutorial. In the section `EQUATION` in the parameter file you will find the options that directly influence the behaviour of the equation:

```
! ============================================================ !
! EQUATION
! ============================================================ !
AdvVel          = (/1.,0.,0./)
DiffC           = 0.
```

The parameter `AdvVel` sets the advection velocity in all three space directions. Since we are simulating a one-dimensional case, we only set the first entry to something other than zero. Here we choose one, although the special initialization that we will be using will set the advection velocity to this value no matter what we put in here. Since we don't want any physical diffusion, we set the diffusion coefficient `DiffC` to zero. If you did compile without the `FLEXI_PARABOLIC` option set to `ON`, you don't need to set this.

We also need to set the initial condition and we choose a special one here to match the initial conditions of the analysis we want to compare against. Recall that the exact solution was

$$\Phi(x,t) = e^{i(kx-\omega t)},$$

where the amplitude of the wave is given by the real part of this complex expression. This real part is then given by

$$A(x,t) = cos(kx - \omega t).$$

This function is implemented as `ExactFunc 6` in the LinAdvDiff equation system, so we are specifying this in the parameter file:

```
IniExactFunc   = 6
```

What is left is to specify the angular frequency ω, since the wavenumber is given by $k = \frac{\omega}{u}$. The angular frequency can be set during runtime using the parameter

```
OmegaRef       = 2.
```

We will adopt this parameter when we run our simulations.

To reduce the possibility that the error introduced by the time discretization will alter our results (we are focused here on the behaviour of the spacial discretization), we set our CFL number to a rather small value:

```
CFLscale       = 0.1
```

Since this is a really small and fast computation, we are also going to use visualization routines during runtime, so we don't need to convert the state files using the `posti_visu` tool. To do this, we set the following parameters in our parameter file:

```
outputFormat   = 3
...
NVisu          = 30
```

which will enable the `.vtu` output. Whenever the analyze routines are called, there will also be a call to the visualization routines. Since we did not set a `Analyze_dt` in the parameter file, the analyse routines will only be called at the beginning and the end of the simulation. The parameter `NVisu` sets the polynomial degree of the output basis.

Remark: There is also a parameter that allows us to only call the visualize routines every n-th call of the analyze functions. This parameter is called `nWriteData`. So if you set

```
nWriteData   = 10
```

in the parameter file, the visualization file will only be written every 10th analyze routine call. The default for this value is simply `1`.

6.8.4 Simulation and results

We are going to start with the simulation of a well-resolved wave and a moderate polynomial degree of $N = 4$ which is already set in the parameter file. For a modified wavenumer of $K^* = \frac{1}{4}\pi$, the dissipation and dispersion relations tell us that we can expect very little errors. To get the desired modified wavenumber, we need to calculate the corresponding angular frequency. Using the relations given above, the angular frequency can be computed as

$$\omega = \frac{K^*(N+1)}{\Delta x} u.$$

This means we need to set the following frequency in the parameter file:

```
OmegaRef       = 1.96349540849
```

All other options are already set correctly. The simulation will be performed up to a time of $t = 5$. We can now run the simulation using

```
cd $FLEXIROOT/tutorials/linadv
$FLEXIROOT/buildLinAdv/bin/flexi parameter_flexi.ini
```

Even on a single processor this calculation should only take a few seconds.

Now we can have a look at the result by opening the file `LinAdvCosineWave_Solution_0000005.000000000.vtu` with ParaView. We are only interested in the central element in $x \in [-1, 1]$, so we can clip everything else. Also since we are performing a one-dimensional simulation, we can extract the solution along the x-axis and create a simple line-plot to look at the results. To see how dispersion and dissipation introduced by the numerics influence our results, we will also plot the analytical solution to the wave transportation problem, given by the equation

above. The result can be seen in figure 6.29. As was expected, only very little deviation can be found from the analytical solution, since we are considering a very well resolved wave with a low value for K^*.

Figure 6.29: Result for $N = 4$ and a well resolved wave with $K^* = \frac{1}{4}\pi$ at $t = 5$. Comparison against exact wave transport.

Of course it is much more interesting to look at the behaviour of waves that are not well resolved and how the polynomial degree influences the behaviour of these waves. To do this, we are going to perform four simulations with polynomial degrees ranging from 2 to 11 and set the normalised wavenumber to $K^* = 1.6$. Table 6.18 will give you an overview of the polynomial degrees and the angular frequencies needed to achieve a constant K^*.

Table 6.18: Angular frequencies needed to get $K^* = 1.6$.

N	OmegaRef
2	2.4
4	4.0
6	5.6
11	9.6

Now run these four simulations by adjusting the values for `N` and `omegaRef` in the parameter file before each run. Remember to give each of them a unique `ProjectName` or else the results will be overwritten each time. Also make sure that `NVisu` is set to at least three times the value of N to get a meaningful visualization.

We now compare the results as they can be found in the next figure. Again we plot the solution at $t = 5$ for the considered polynomial degrees against the analytical solution. Since we compare using a constant normalized wavenumber, the actual frequency of the wave is increased with the polynomial degree. If we take a look at the result for $N = 2$ (which should not be considered high-order), we observe both a significant amount of dissipation (a drop in amplitude by about 85%) as well as severe dispersion, e.g. a change in angular frequency and phase angle.

For a polynomial degree of $N = 4$ (at the lower end of what can be considered high-order), we already observe some improvements. The amplitude only drops by around 65%, although the phase shift is still clearly visible. This trend continues with higher polynomial degrees. For the highest value of N tested here, we observe only small changes in phase angle or amplitude.

These results show how higher-order schemes are able to capture waves with fewer points per wavelength than the lower-order approximations. This is one of the central aspects why we use such schemes.

Figure: Result for a under-resolved wave with $K^* = 1.6$ at $t = 5$ for $N = 2$ (top left), $N = 4$ (top right), $N = 6$ (bottom left) and $N = 11$ (bottom right).

6.9 Plane Turbulent Channel Flow

This tutorial describes how to set up and run the Plane-Turbulent-Channel-Flow test case. We will learn how to use the split form DG method to guarantee non-linear stability of the turbulent channel flow. In a second step, we add the sub grid scale model of Smagorinsky combined with Van Driest type damping to run stable wall-bounded turbulent flows with explicit small scale dissipation. The tutorial assumes that you are familiar with the general **FLEXI** and **HOPR** work flow (please finish the previous tutorials first if this sounds strange to you).

6.9.1 Flow description

The flow is calculated in a plane channel with half-height $\delta = 1$, streamwise (x coordinate) length 2π and span (z coordinate) width π with periodic boundaries in the x- and z-directions as well as no-slip walls at the top and the bottom of the domain. As initial conditions an analytical mean turbulent velocity profile is used superimposed with sinus perturbations in the u, v and w velocity components and a constant density of $\rho = 1$. The superimposed pertubations lead to rapid production of turbulent flow structures. Since the wall friction would slow down the flow over time, a constant pressure source term imposing a pressure gradient $\frac{dp}{dx} = -1$ is added as a volume source. While the test case is incompressible in principle, we solve it here in a compressible setting. The chosen Mach number with respect to the bulk velocity in the field is $Ma = 0.1$ according to the Moser channel test case. In this setting, the wall friction velocity τ will always be equal to 1. We can the define a Reynolds number based on the channel half-height and the wall friction velocity as $Re_\tau = 1/\nu$.

6.9.2 Compiler options

Make sure that **FLEXI** is compiled with the CMake options listed in the following table.

Table 6.19: CMake options for the plane turbulent channel flow test case simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional
FLEXI_EDDYVISCOSITY	ON	optional
FLEXI_NODETYPE	GAUSS-LOBATTO	
FLEXI_SPLIT_DG	ON	
FLEXI_TESTCASE	channel	
POSTI	ON	

Option	Value	Comment
POSTI_CHANNEL_FFT	ON	

For all other CMake options you may keep the default values. Compile the code.

6.9.2.1 Mesh Generation with HOPR

We use a cartesian mesh with 4 cells per direction for the tutorial. The mesh is stretched in the wall-normal direction to accommodate for the straining of the vortexes close to the wall. In case you want to generate other meshes the parameter file for **HOPR** is included in the tutorial directory (*parameter_hopr.ini*), the default mesh is included. Using 4 cells with a polynomial degree of $N = 5$, means we use a large eddy simulation setup of 24 DOFs per direction.

6.9.3 Tutorial - Flow at $Re_\tau = 180$

Copy the `plane_turbulent_channel_flow` tutorial folder to your working directory.

```
cp -r $FLEXI_TUTORIALS/plane_turbulent_channel_flow .
```

Step into the folder. In case you do not want to generate the mesh files yourself, a default mesh has already been provided.

6.9.3.1 Preparing the Flow Simulation with FLEXI

The simulation setup is already defined in *parameter_flexi.ini*.

6.9.3.1.1 Output In this tutorial we don't look at the flow visualization of the instantaneous state files. Here, we will rather post process consecutive, instantaneous state files with the `posti_channel_fft` tool. As an output, we receive mean velocity and Reynolds stress profiles as well as turbulent energy spectra at different locations normal to the channel wall.

6.9.3.1.2 Interpolation / Discretization parameters In this tutorial we use the split form DG method to guarantee non-linear stability of the turbulent channel flow simulation. As already specified in the CMake options table 6.19, the `FLEXI_SPLIT_DG` option has to be switched ON in combination with the `FLEXI_NODETYPE GAUSS-LOBATTO`. **FLEXI** has several split flux formulations implemented. Therefore, a specific split flux formulation has to be set in the *parameter_flexi.ini* file. In this tutorial the pre-defined split flux formulation by Pirozzoli is used, which results in a kinetic energy preserving DG scheme.

```
! ================================================ !
! SplitDG
! ================================================ !
SplitDG        = PI      ! SplitDG formulation to be used: SD, MO,
   DU, KG, PI
```

To switch on Smagorinsky's model set the eddyViscType to 1 in the *paramerter_flexi.ini* file. In addition, the following parameters have to be set. CS is the Smagorinsky constant usually chosen around 0.11 for wall bounded turbulent flows and the turbulent Prandtl number is commonly set to 0.6. To ensure the correct behaviour of the eddy viscosity provided by Smagorinsky's model when approaching a wall, Van Driest type damping has to be switched on.

```
! ================================================ !
! LES MODEL
! ================================================ !
eddyViscType = 0        ! Choose LES model, 1:Smagorinsky
VanDriest = T           ! Van Driest damping for LES viscosity (
   channel flow only)
CS = 0.11               ! Smagorinsky constant
PrSGS = 0.6             ! turbulent Prandtl number
```

6.9.3.2 Running the Simulation and Results

Now run the simulation, either using

```
flexi parameter_flexi.ini
```

or

```
mpirun -np XX flexi parameter_flexi.ini
```

when you want to use more than one processor. Once the simulation finished state files can be post processed by the `posti_channel_fft` tool which was build by the `POSTI_CHANNEL_FFT` CMake option. To run the postprocessing, the standard command is

```
posti_channel_fft parameter_channel_fft.ini [State1 State2 ...]
```

where the *parameter_channel_fft.ini* file is given in the tutorial folder and the amount of statefiles is specified by the user. In this tutorial we use all state files with a timestamp between $t = 10.0$ and $t = 15.0$. As an output you receive three different files. One containing the mean velocity profiles as well as the Reynolds stress profiles and the other two files contain turbulent erergy spectra. To visualize those files you can run the python script `plotChannelFFT.py` in the `tools/testcases` folder with the following command in your simulation directory

```
python $FLEXIROOT/tools/testcases/plotChannelFFT.py -p
   $PROJECTNAME -t $POSTITIME
```

where `$PROJECTNAME` specifies the project name specified in the *parameter_flexi.ini* file and `$POSTITIME` the timestamp of your output files from the `posti_channel_fft` tool.

6.9.3.3 Part I: SplitDG iLES

First, we run **FLEXI** without Smagorinsky's model which we call an implicit LES (iLES), as no explicit sub-grid scale dissipation model is added. The resulting mean velocity and Reynolds stress profiles as well as turbulent energy spectra close to the centre of the channel are given in Figure 6.30.

Figure 6.30: Mean velocity and Reynolds stress profiles (left) as well as turbulent energy spectra close to the centre of the channel (right} of an implicit LES at $Re_\tau = 180$.

6.9.3.4 Part II: SplitDG with explicit LES model

In a second step, we run **FLEXI** with Smagorinsky's model and Van Driest damping which needs to be switched on in the parameter file as described above. The resulting mean velocity and Reynolds stress profiles as well as turbulent energy spectra close to the centre of the channel are given in Figure 6.31. In comparison to the previous simulation you might recognize the effect of the explicit damping on the Reynolds stress profile $\overline{u'u'}$ close to the maximum, most. To further study the influence of Smagorinsky's model play around with the spatial resolution both in terms of grid resolution as well as the polynomial degree N. You can also increase the Reynolds number to $Re_\tau = 395$ or $Re_\tau = 590$ and compare the results to DNS results from Moser et al. (Moser, Kim, and Mansour 1999).

Figure 6.31: Mean velocity and Reynolds stress profiles (left) as well as turbulent energy spectra close to the centre of the channel (right} of a LES with Smagorinsky's model and vanDriest damping at $Re_\tau = 180$.

6.9.4 Performance improvements

FLEXI comes with some advanced optimizations in order to increase its computational efficiency for compute-intesive simulations. These appear once the flag `FLEXI_PERFORMANCE=ON` is set. The first option is `FLEXI_PERFORMANCE_OPTLIFT`, which can be activated to optimize the computation of the parabolic terms of the applied equation system. However, POSTI is not available if this option is enabled. The option `FLEXI_PERFORMANCE_PGO=ON` allows to enable profile-guided optimization (PGO). For PGO, the executable is first instrumented with some profiling tools by the compiler and then executed on a simple test case. The generated profiling data can be used by the compiler to identify bottlenecks and hotspots in the code that it cannot spot from the static source code by itself. The executable is thus compiled a second time using the gathered profiling data to perform these additional optimizations. In FLEXI, this two-step compilation works as follows. First, FLEXI is compiled with the following options.

Table 6.20: CMake options for the plane turbulent channel flow test case with additional performance flags enabled.

Option	Value	Comment
CMAKE_BUILD_TYPE	Profile/Release	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional
FLEXI_EDDYVISCOSITY	ON	optional
FLEXI_NODETYPE	GAUSS-LOBATTO	
FLEXI_SPLIT_DG	ON	

Option	Value	Comment
FLEXI_TESTCASE	channel	
FLEXI_PERFORMANCE	ON	
FLEXI_PERFORMANCE_OPTLIFT	ON	
FLEXI_PERFORMANCE_PGO	ON	
POSTI	OFF	

For the first step, FLEXI havs to be compiled with `CMAKE_BUILD_TYPE=Profile` in order to activate the profiling. Then, FLEXI has to be executed on a simple test case like the freestream tutorial by following the instructions from Section 6.1. Finally, FLEXI is compiled a second time, but this time with the build type set to `CMAKE_BUILD_TYPE=Release` in order to incorporate the generated profiling data into the compilation process. Now, FLEXI can be executed as usual and should show a considerable performance improvement in comparison to the previous simulations. Please be aware, that PGO is currently only supported for the GNU compiler and that this two-step compilation process has to be performed each time the compile options, i.e. the FLEXI executable, are changed.

6.10 Flow around a cylinder

In this tutorial, the simulation around a two-dimensional circular cylinder at $Re_D = 200$ and $Ma = 0.2$ is considered. The goal of this tutorial is to introduce the usage of a new functionality of the **posti_visualizerecordpoints** tool and the new tool **posti_dmd**.

6.10.1 Compiler options

Make sure that **FLEXI** is compiled with the CMake options listed in the following table.

Table 6.21: Cmake options for the cylinder simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_2D	ON	
FLEXI_EQYNSYSNAME	navierstokes	
FLEXI_PARABOLIC	ON	
LIBS_USE_MPI	ON	optional
POSTI_RP_VISUALIZE	ON	
POSTI_DMD	ON	

To check whether they are set, change to your `build` folder and open the CMake GUI

```
ccmake [flexi root directory]
```

If necessary, set the above options and then compile the code by issuing

```
make
```

6.10.2 Mesh Generation with HOPR

The mesh file used by **FLEXI** is created by **HOPR**

```
./hopr parameter_hopr.ini
```

This creates the mesh file *Cylinder_Re200_mesh.h5* in HDF5 format. If **HOPR** is not available, the mesh file is supplied in this tutorial.

6.10.3 Flow Simulation with FLEXI

The simulation setup is defined in `parameter_flexi.ini`. The initial condition is selected via the variable vector `RefState=(/1.,1.0,0.,0.,17.857/)` which represents the vector of primitive solution variables $(\rho, u, v, w, p)^T$.

Material properties are given in table 6.22. Based on the ideal gas law, we get

$$Ma = 1/\sqrt{\kappa p/\rho} = 0.2$$

Note that in this non-dimensional setup the mesh is scaled such that the reference length is unity, i.e. $D = 1$. Then to arrive at $Re = \rho u D/\mu = 200$, the viscosity is set to

$$\mu = \rho u D/Re = 1/Re = 0.005$$

Table 6.22: Material properties set in the parameter file

Property	Variable	Value
dynamic viscosity μ	mu0	0.005
ideal gas constant R	R	17.857
Prandtl number	Pr	0.72
isentropic coefficient κ	kappa	1.4

6.10.4 Numerical settings

The DG solution on the mesh is represented by piecewise polynomials and the polynomial degree in this tutorial is chosen as $N = 4$.

The main code settings are shown in table 6.23.

Table 6.23: Numerical settings

Variable	Description	Value
N	Polynomial degree	4
MeshFile	Mesh file to be used	Cylinder_Re200_mesh.h5
tend	end time of the simulation	300
Analyze_dt	time interval for analysis	0.01
nWriteData	dump solution every n'th Analyze_dt	500
CFLscale		0.9
DFLscale		0.9

6.10.5 Boundary conditions

The boundary conditions were already set in the mesh file by **HOPR**. Thus, the simulation runs without specifying the boundary conditions in the **FLEXI** parameter file. The freestream boundaries of the mesh are Dirichlet boundaries using the same state as the initialization, the wall is modeled as an adiabatic wall. The boundary conditions in z direction are not relevant for this 2D example, but would be realized as periodic boundaries for a 3D simulation. All boundary conditions used are listed below.

```
        Name       Type      State      Alpha
   BC_cylinder        3          0          0
   BC_farfield        2          0          0
```

6.10.6 Running the code

We proceed by running the code in parallel. For example, using 4 processors, use the following command

```
mpirun -np 4 flexi parameter_flexi.ini
```

The simulation runs for 300 convective time units to achieve periodic vortex shedding, thus the simulation can take up to one to two hours.

6.10.7 Evaluation of Strouhal number

The Strouhal number (which is a non-dimensional frequency, $Sr = \frac{f \cdot D}{u}$, describing the oscillatory motion of the flow) is estimated using the forces acting on the cylinder induced by the vortex shedding. The forces are calculated on the fly during runtime. The associated flags in the parameter file are

```
CalcBodyForces=T
WriteBodyForces=T
```

The first line activates the calculation of the forces at each `Analyze_dt`, the second line enforces output of the forces to a file. In figure 6.32 the force in y-direction is plotted. By measuring the time from peak to peak over several periods the Strouhal number can be estimated to 0.1959 which is close to the expected value from literature.

Figure 6.32: Resulting forces on the airfoil up to $t = 10$.

6.10.8 Evaluation of the separation angle

The mean separation angle is evaluated using the **record points**-tool as introduced in 6.3.13. The simulation setup already contains the **record points** set and the record points are written during the simulation. The **record points** set contains a plane within the boundary layer of the upper cylinder side. This time we want to use the **Plane_doBLProps** functionality within the **posti_visualizerecordpoints** tool. With this tool we want to analyze the boundary layer properties such as the wall friction to estimate the separation point. The parameter needed are already set in the parameter_visualizeRecordpoints.ini file.

You can run the tool using

```
posti_visualizerecordpoints parameter_visualizeRecordpoints.ini
   Cylinder_Re200_RP_*
```

After executing the tool, you will get a file named **Cylinder_RP_BLProps_upperSide_BLPla000001.vts** which can be visualized with ParaView. The Data we want to visualize is one dimensional, so you won't be able to see the data in the render view. To visualize it you need the apply the plot over time filter. ParaView should automatically apply the correct range to plot on. By plotting **tau_w** over the circumference you can estimate the separation angle to $113\ deg$ (the intersection with the $y = 0$ line).

6.10.9 Dynamic Mode Decomposition

In this part of the tutorial we want to introduce the posti tool **posti_dmd**. The dynamic mode decomposition is an algorithm divide a temporal series into a set of modes which are associated with a frequency and grow/decay rate. With this tool we are also capable to determine the Strouhal frequency. The dynamic mode decomposition (DMD) is implemented according to Schmid et al. (Schmid, Meyer, and Pust 2009).

To use this tool, we need a higher temporal resolution of the written state files. Thus, we change the time **tend** to 310 and **nWriteData** to 1. We restart the simulation from the latest state file:

```
flexi parameter_flexi.ini Cylinder_Re200_State_0000300.000000000.
   h5
```

To execute the DMD on the density run the following command:

```
posti_dmd parameter_dmd.ini Cylinder_Re200_State_00003*
```

Depending on the available memory you might have to decrease the number of input state files. After execution you will see two additional files **Cylinder_Re200_DMD_0000300.000000000.h5** and **Cylinder_Re200_DMD_Spec_0000300.000000000.dat**. The first file contains the field representation of the different modes and the second file contains the ritz spectrum of the modes.

To visualize the field run the following command:

```
posti_visu parameter_postivisu.ini Cylinder_Re200_DMD_0000300
   .000000000.h5
```

The new file **Cylinder_Re200_Solution_0000300.000000000.vtu** now contains five modes to visualize. Figure 6.33 shows the steady, the global, the first and the second harmonic mode. The global mode is the mode of the considered Strouhal number.

With the python script *plot_RitzSpectrum.py* the Ritz spectrum of the DMD can be plotted. The script is placed in the tools folder of **FLEXI**. To plot the spectrum execute:

Figure 6.33: DMD modes of the density field. Top left steady mode, top right global mode, bottom left first harmonic, bottom right second harmonic.

```
python plot_RitzSpectrum.py -d Cylinder_Re200_DMD_Spec_0000300
    .000000000.dat
```

The result is a Ritz spectrum as shown in fig 6.34. On the *x-axis* the frequency of the modes and on the *y-axis* the growth/decay factor is plotted, whereat modes with $\omega_r < 0$ are damped. The modes placed directly on the x-axis are the already discussed modes, from left to right the global, the first, the second harmonic mode and so on. The color and size of the plotted modes represent the Euclidian norm of the mode which can be interpreted as an energy norm of the mode.

Figure 6.34: Ritz spectrum.

6.11 Flow over a flat plate with RANS equations

FLEXI is mainly designed for scale-resolving simulations of turbulent flow, i.e. direct numerical simulation (DNS) or large eddy simulation (LES). Unfortunately, the required resolution of the grids in both space and time for those simulation approaches exceeds the available computational power by several orders of magnitude for many applications that are relevant to e.g. engineering applications.

An alternative for those situations is to solve the so-called Reynolds Averaged Navier-Stokes or RANS equations. The RANS equations are derived by applying a temporal filter to the Navier-Stokes equations, which must be wide enough to filter out any fluctuating components of the solution variables. One is thus left with a set of equations that describe a stationary solution. Formally, those filtered equations will look like the original Navier-Stokes equations, except for closure terms that appear due the filtering of the non-linear fluxes. Those closure terms depend on the unfiltered variables, and thus need to be modelled. For RANS, the closure terms must contain the effects of all scales except for the stationary flow field. The

modelling of those is a challenge, and many different approaches have developed over the years. In **FLEXI**, a rather simple but widely used model after Spalart and Allmars (SA model, see (Allmaras and Johnson 2012)) is implemented. It belongs to the class of one-equation models, and solves a single additional equation for a turbulent viscosity $\tilde{\mu}$. All effects of the unresolved scales are thus modeled by assuming an increased viscosity of $\mu + \mu_{turb}(\tilde{\mu})$.

In this tutorial, we will simulate the flow over a flat plate using the RANS equations. For the presented setup, the Reynolds number at the streamwise position $x = 1m$ along the plate will be equal to $Re_x = \frac{\rho_\infty u_\infty x}{\mu} = 5 \cdot 10^6$. The mach number is set to $Ma = 0.2$.

First, we explain how the parameters for the RANS equations have to be set. Next, it is shown how implicit time discretization can help to save computational time.

Copy the `naca0012` tutorial folder to your desired working directory.

```
cp -r $FLEXI_TUTORIALS/flatPlate .
```

6.11.1 Compiler options

Make sure that **FLEXI** is compiled with the cmake options listed in the following table. Note that since only a stationary flow over the plate is considered, the example is purely two-dimensional, which additionally saves computational resources.

Table 6.24: CMake options for the RANS flat plate simulation.

Option	Value	Comment
CMAKE_BUILD_TYPE	Release	
FLEXI_2D	ON	
FLEXI_EQYNSYSNAME	rans_sa	
FLEXI_FV	OFF	
FLEXI_LIFTING	br1	
FLEXI_PARABOLIC	ON	
FLEXI_MPI	ON	optional
FLEXI_NODETYPE	GAUSS	

```
ccmake [flexi root directory]
```

Set the above options and then compile the code by issuing

```
make
```

6.11.2 Flow Simulation with FLEXI

The simulation setup is defined in `parameter.ini`.

`IniRefState = 1` : the initial condition uses `RefState 1` for the initial flow field solution.

`IniExactFunc = 1` : exact function routine for initialization, case 1 initializes a freestream state based on `IniRefState`.

Since we solve an additional equation for the SA model, the `RefState` will also have an additional entry compared to the Navier-Stokes simulations. In the sixth entry, a value for $\tilde{\nu} = \frac{\tilde{\mu}}{\rho}$ must be set. For fully turbulent simulations, a value of $\tilde{\mu} = \frac{\mu}{10}$ is appropriate.

Material properties are given in table 6.25.

Table 6.25: Material properties set in the parameter file

Property	Variable	Value
dynamic viscosity μ	mu0	0.0000171
isentropic coefficient κ	kappa	1.4

6.11.3 Numerical settings

The DG solution on the mesh is represented by piecewise polynomials and the polynomial degree in this tutorial is chosen as $N = 4$.

The main code settings are displayed in table 6.26.

Table 6.26: Numerical settings

Variable	Description	Value
N	Polynomial degree	4
MeshFile	Mesh file to be used	meshes/CART_HEX2D_FlatPlateC_mesh.h5
tend	end time of the simulation	0.25
Analyze_dt	time interval for analysis	0.001
nWriteData	dump solution every n'th Analyze_dt	50
CFLscale		0.9
DFLscale		0.9

6.11.4 Mesh and boundary conditions

The mesh itself is of a simple rectangular shape. On the left site and on the top, the free-stream value given by the initialization is prescribed directly. On the right, a pressure-based outflow condition is used. Since the flow is stationary, no additional treatment of the outflow is necessary. On the bottom of the domain, for $0.2m$ a symmetry condition is applied, since we don't want to directly start with the flat plate at the inflow. After $x = 0.2m$, an adiabatic wall begins. The mesh is stretched, such that the nodes in wall-normal direction are clustered towards the wall. The gradients in (even the stationary) turbulent boundary layer are very large, such that we need a lot of resolution in that direction. The grid is also stretched in the wall-parallel direction, and nodes are clustered at the leading edge of the plate. Here, larger gradients are expected since the boundary layer is developing very rapidly at the start. At later x-positions, we can use very large cells in the wall-parallel direction, since not a lot of change is going on there.

Several versions of the mesh are included in the subfolder `mesh`. They only differ in the amount of cells used in the wall-normal direction. The main influence is the size of the cell closest to the wall, which needs to resolve the largest gradient. Mesh *A* is the finest, with a cell size of $y^+ \approx 4$, expressed in wall-units in the middle of the flat plate. This does not yet take into account that there are multiple nodes inside of a cell for DG methods! Mesh *C*, which is used in the default setup, in combination with the polynomial degree of $N = 4$ leads to $y^+ \approx 4$ for the first **node**. Though this is much larger than the usual requirement of $y^+ \approx 1$, due to the high order nature of the simulation this is enough to gain results that are in reasonable agreement with experimental and other numerical reference solutions.

6.11.5 Wall distance

The additional equation for the SA model uses the distance from the closest solid wall as an input. Since this is a static quantity, we can compute this in a pre-processing step and then simply read that information if we start our actual simulation. For the default setup, the file containing the information about the wall distance is already included, in the file `meshes/CART_HEX2D_FlatPlateC_walldistance.h5`. If you want to use other meshes or a different polynomial degree, you will need to generate the file yourself. For this purpose, the pre-processing tool `walldistance` is included in **FLEXI**. You need to change some compile options so it will be available, which are listed in the following table.

Table 6.27: CMake options for the walldistance tool.

Option	Value	Comment
FLEXI_BUILD_POSTI	ON	
POSTI_BUILD_WALLDISTANCE	ON	

Once the tool is build, you can run it by issuing a command like

```
posti_walldistance walldistance.ini
```

The tool only takes a parameter file as input. A sample `.ini` can be found in the `meshes` subfolder, you need to specify the required polynomial degree and the mesh to use. The program will then use a gradient descent technique to find the distance to the closest wall for each solution point. Since for Gauss-Lobatto nodes the boundaries and thus walls are included in the set of nodes, we would have points where the distance is zero. Unfortunately, the SA equation divide by that distance, which does not work. Thus, only use the RANS equation system with Gauss nodes!

6.11.6 Running the code

We proceed by running the code in parallel. For example using 4 processors, use the following command

```
mpirun -np 4 flexi parameter.ini
```

On a 2017 laptop with core i5 processor, this simulation takes about 20 hours until the default end-time is reached. Note that the time here has no physical meaning, since we are interested in the stationary solution. The analyze routines will report the remaining residual for each quantity (L_2 norm of the spatial DG operator). These are indicators for how much the solution is still changing. We call our solution converged if a certain threshold for the residuals is reached, or if they do not get smaller anymore.

6.11.7 Implicit time discretization

This simulation setup is now used to illustrate how to use implicit time discretization instead of explicit methods. Since the mesh contains very small cells to resolve the gradient at the wall, the explicit time step is really small. Also, we don't require a time-accurate solution of the problem, instead we want to converge to a stationary solution as fast as possible. This makes our simulation perfect for implicit time discretization. In FLEXI several explicit and implicit Runge-Kutta methods are implemented. To get a list of the available options run the command

```
flexi --help
```

Here is a list how the result could look like

```
TimeDiscMethod        =      CarpenterRK4-5 ! Specifies the type
   of time-discretization to be used, e.g. the name of a specific
   Runge-Kutta scheme. Possible values:
                                       !   * standardrk3-3
                                       !   * carpenterrk4-5
                                       !   * niegemannrk4-14
```

```
                                        !   * toulorgerk4-8c
                                        !   * toulorgerk3-7c
                                        !   * toulorgerk4-8f
                                        !   * ketchesonrk4-20
                                        !   * ketchesonrk4-18
                                        !   * eulerimplicit
                                        !   * cranknicolson2-2
                                        !   * esdirk2-3
                                        !   * esdirk3-4
                                        !   * esdirk4-6
```

The list starts with explicit methods which are followed by the implicit ones starting from `eulerimplicit`. Implicit time discretization methods allow for a higher CFL number than the explicit ones as they are typically unconditionally stable. The drawback of implicit methods is that they require the solution of large non-linear equation systems in each timestep. In FLEXI the arising non-linear equation system is solved with Newton's method. This necessitates a linear solver for each Newton step. For this the iterative GMRES method is used. Iterative methods such as Newton's method and GMRES require the definition of convergence criteria and iteration parameters. Below, they are shortly summarized:

```
!==========================================
! Implicit
!==========================================
adaptepsNewton        =                   F ! Adaptive Newton
   convergence criterion by Runge-Kutta error estimation
EpsNewton             =             0.1E-02 ! Newton tolerance,
   only used if adaptepsNewton=F
nNewtonIter           =                  50 ! Maximum amount of
   Newton iterations

EisenstatWalker       =                   F ! Adaptive abort
   criterion for GMRES
gammaEW               =                 0.9 ! Parameter for
   Eisenstat Walker adaptation
EpsGMRES              =             0.1E-02 ! GMRES Tolerance,
   only used if EisenstatWalker=F
nRestarts             =                  10 ! Maximum number of
   GMRES restarts
nKDim                 =                  30 ! Maximum number of
   Krylov subspaces for GMRES, after that a restart is performed
```

Note that the adaptive Newton tolerance is only available for `esdirk2-3`,`esdirk3-4` and `esdirk4-6`. When this option is true, Newton's convergence criterion is smaller the smaller the timestep is, hence this option has to be used with care. An adaptive GRMES convergence criterion means that a coarser tolerance is chosen for the first Newton steps, which is decreased for increasing Newton steps. If Newton's method or GMRES do not converge during calculation the abort criteria and the maximum iterations have to be adjusted in the parameter file.

To accelerate both, Newton's method and GMRES, different options are included in FLEXI. We start with Newton's method: There are different options to choose as the convergence property of Newton's method depends highly on the start value. Note that the method 3 (dense output extrapolation) is only available for `esdirk3-4` and `esdirk4-6`.

```
!==========================================
! Implicit
!==========================================
PredictorType          =                    0 ! Type of predictor to
    be used, 0: use current U, 1: use right hand side, 2:
   polynomial extrapolation, 3: dense output formula of RK scheme
PredictorOrder         =                    0 ! Order of predictor
   to be used (PredictorType=2)
```

Typically a predictor is beneficial for 'small' CFL numbers as here the extrapolation gives good results. For 'larger' timesteps, a predictor which does not use the current U can increase the required amount of iterations.

The solution of the linear system which arises from Newton's method requires the inversion of the large system matrix. As FLEXI is designed for large scale parallel simulations, it is not desirable to build up this large matrix and multiply it with the state vector but to approximate it via a finite difference. The parameters for the finite difference are summarized below.

```
!==========================================
! Implicit
!==========================================
FD_Order               =                    2 ! Order of FD
   approximation (1/2)
Eps_Method             =                    2 ! Method to determine
   the step size of the FD approximation of A*v in GMRES, 1: sqrt(
   machineAccuracy)*scaleps, 2: take norm of solution into account
scaleps                =                  1.0 ! Scaling factor for
   step size in FD, mainly used in Eps_Method=1
```

In general, the order of the finite difference approximation can be kept at 1. But in flows with very low Mach numbers or when very little change in a stationary solution is happening, the degradation of the accuracy of the finite difference by machine precision can become an issue. In such cases the order of the finite difference can be increased, as it is done in this tutorial. The method of how to calculate the step size of the finite difference sometimes has to be adjusted when using nondimensional equations where e.g. the Mach or Reynolds number explicitly occur.

To accelerate the convergence of GMRES a preconditioner can be applied. The preconditioner approximates the inverse of the system matrix (Jacobian). Again, several options are available:

```
!==========================================
! Preconditioner
!==========================================
```

```
PrecondType            =                    1 ! Preconditioner Type
   (0: no Preconditioner, 1: analytic, 2: finite difference, 3:
   compute both and compare)
PrecondIter            =                    1 ! Defines how often
   preconditioner is built
SolveSystem            =                    1 ! Solver of the
   preconditioned system (0: exact LU inversion, 1: inexact ILU(0)
    inversion, always with NoFillIn=T)
DebugMatrix            =                    0 ! Write Jacobians to
   file for debug purposes (0: no output, 1: non-inverted matrix
   ,2: additionally inverted matrix, 3: additionally check
   inversion accuracy)
HyperbolicPrecond      =                    F ! Preconditioner only
   for the hyperbolic flux
NoFillIn               =                    F ! Precond for
   parabolic system forced to have the same sparsity as the Euler
   Precond
DoDisplayPrecond       =                    F ! Display building
   time of preconditioner
```

The parameters `PrecondType`,`DebugMatrix` and `DoDisplayPrecond`only have to be changed if analysis concerning the preconditioner shall be conducted. In an application case those parameters should remain unchanged. An important parameter is `SolveSystem`: Using the exact LU inversion is sometimes necessary to obtain convergence of the linear solver. Nevertheless the application of the inexact ILU(0) is typically faster, especially for higher orders and higher dimensions (3d).

6.11.8 Runing the simulation with implicit time discretization

The implicit method FLEXI uses (Newton-GMRES) is well suited for unsteady calculations as here the initial guess for Newton's method is relatively good. Starting a simulation from an unphysical homogeneous state causes very poor convergence properties of Newton's method. Hence, it is beneficial to start the current simulation with an explicit time discretization method and than do a restart after several timesteps have been calculated. For the current simulation we suggest to simulate until `tend=0.02` with an explicit scheme and than use this state for a restart with `eulerimplicit`. The CFL number can then be massively increased, e.g. to 100. The following plot shows how the residuals of the different conservative variables then decrease over time. The residuals are the L_2-Norm of the spatial DG operator at a stage of the time-discretization method. They decrease by several orders of magnitude before reaching a plateau. Here, the Newton algorithm will detect that nearly no change is happening in the iterative procedure, and no iterations will be performed any more. A stationary solution has been reached.

Figure 6.35: Residuals over time for the implicit part of the flat plate computation

7 Unit tests

Unit tests are used to test individual key units of the source code. Currently these key routines include:

- Calculation of node positions and integration weights.
- Calculation of Vandermonde matrices.
- Calculation of derivative matrices.
- Algorithms to interpolate data from one set of nodes to another.
- Algorithm to prolong volume data to sides.
- Algorithm to perform a surface integral.
- Functionality of Read-In tools.

7.1 Integration of unit test with CTest

These unit tests are integrated into the **FLEXI** build process using the CTest tool. Usually CTest will be run every time you build **FLEXI** and give you an overview on the exit status of each test that looks something like this:

```
Run unit tests
Test project /home/FLEXI/build
    Start 1: NodesAndWeights
1/6 Test #1: NodesAndWeights ..............   Passed    0.01 sec
    Start 2: Vandermonde
2/6 Test #2: Vandermonde ..................   Passed    0.01 sec
    Start 3: DerivativeMatrix
3/6 Test #3: DerivativeMatrix .............   Passed    0.00 sec
    Start 4: ChangeBasis
4/6 Test #4: ChangeBasis ..................   Passed    0.00 sec
    Start 5: SurfInt
5/6 Test #5: SurfInt ......................   Passed    0.00 sec
    Start 6: ProlongToFace
6/6 Test #6: ProlongToFace ................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 6

Total Test time (real) =   0.05 sec
```

To manually run the tests after a build use the CTest command

```
ctest
```

in your build directory. The manual page of CTest can give you an overview of all available options.

If you don't want to run the test after each build there is a CMake option called FLEXI_UNITTESTS that can be used to turn the tests on and off. This is an advanced option that CCMake will only show if you enter the advanced mode by pressing the t key.

7.2 Implementation of unit tests

All unit tests are implemented in FORTRAN and can be found in the subdirectory unitTests in your **FLEXI** directory alongside a seperate CMakeLists.txt and some binary input and reference files.

7.2.1 CMakeLists.txt

The CMakeLists.txt defines a custom function called add_unit_test which can be used in the CMakeLists.txt to add a single test to the CTest tool. The syntax is

```
add_unit_test(NAME SOURCEFILE.F90)
```

All tests are defined using this function. At the end of the CMakeLists.txt a custom target all_tests is defined which includes all unit tests and will run the ctest command after it has been build.

The whole CMakeLists.txt content is included in the main CMakeLists.txt if the option FLEXI_UNITTESTS is set to ON (default) by CMake.

7.2.2 General unit test structure

The general structure of the unit tests is the same in all cases. They are implemented as FORTRAN programs. The unit test will call a function or subroutine from the **FLEXI** framework with input either set in the program itself or read from a binary file. The output of this call will then be compared to some precomputed reference results (also stored as binary files) with a certain tolerance to account for differences in e.g. compiler versions and system architecture. If the results are within the given tolerance, the test will be passed, otherwise it will fail by returning a value other than 0.

The programs usually also contain a command line option that can be uses to generate the reference solution from a code version that is known to work correctly.

Have a look at the source code of one of the already implemented unit tests if you want to have a more detailed idea about how to implement your own tests.

7.2.3 Generation of reference mesh data

Some of the unit tests require parts of the mesh data structure to be able to call the funtions to be tested. For this purpose, a curved single element is created and all the mesh data stored as a binary file called `UnittestElementData.bin`. This binary file can then be read during runtime by the unit test programs.

To generate the curved single element mesh, run **HOPR** with the parameter file provided in the `unitTest` subdirectory of **FLEXI**. To generate the binary file, run **FLEXI** with the following command line argument and the parameter file provided in the `unitTest` subdirectory:

```
flexi --generateUnittestReferenceData parameter.ini
```

8 Installation guidelines

This chapter contains guidelines to install the code from Github on specific systems.

8.1 Cloning and compiling on machines at the HLRS

Unfortunately, the GitHub server is not available on machines at the HLRS, such as the Hazelhen, due to restricted internet access. The workaround is to use ssh tunnels to access the GitHub repositories. Note that the reomte repositories hosted at teh GitLab at the Institute of Aerodynamics and Gasdynamics (IAG), no ssh tunnel is required and cloning works straight forwardly.

The following instructions to access the GitHub repositories on HLRS machines is taken from the HLRS wickie page, see https://wickie.hlrs.de/platforms/index.php/Secure_Shell_ssh#Git.

8.1.1 HTTPS

Unfortunately, just using a SSH tunnel as with the SSH and git protocols is not sufficient in this case. Instead, one has to connect via an additional SOCKS proxy on a machine that has unlimited access to the internet, e.g. your local machine.

In order to do so, establish a proxy by using a special feature of OpenSSH:

```
ssh -N -D 1080 localhost
```

This will establish some kind of a "loopback" SSH connection from your local machine to itself which will not execute any command (-N) but act as an SOCKS proxy on port 1080 (-D 1080).

On a second shell, now login to the desired HWW-system and forward a port on the remote machine (e.g. 7777) to the port on your local machine where the newly established SOCKS proxy is listening on (1080):

```
ssh -R 7777:localhost:1080 <system-name>.hww.de
```

By doing so, you have a SOCKS proxy listening on port 7777 of the HWW-system. Hence you can use this proxy for accessing remote git repositories. Unfortunately, the default versions of git installed on the HWW-systems are not capable of doing this. You hence have to load an appropriate version first:

```
module load tools/git
```

In order to use the proxy, you can now add "-c https.proxy=‘socks5://localhost:7777’ " to your git commands, e.g.:

```
git -c https.proxy='socks5://localhost:7777' clone https://
github.com/flexi-framework/flexi.git
```

In order to avoid typing this in every git call, you can also set the respective port to be used whenever git talks to a remote repository via HTTPS by

```
git config --global https.proxy 'socks5://localhost:7777'
```

Unfortunately, to connect with GitHub for pulling or pushing, the connection to Hazelhen has to be done via the ssh tunnel.

References

Allmaras, Steven R, and Forrester T Johnson. 2012. "Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model." In *Seventh International Conference on Computational Fluid Dynamics (Iccfd7)*, 1–11.

Atak, Muhammed, Andrea Beck, Thomas Bolemann, David Flad, Hannes Frank, and Claus-Dieter Munz. 2016. "High Fidelity Scale-Resolving Computational Fluid Dynamics Using the High Order Discontinuous Galerkin Spectral Element Method." In *High Performance Computing in Science and Engineering' 15*, 511–30. Springer.

Bassi, F., and S. Rebay. 1997. "A High-Order Accurate Discontinuous Finite Element Method for the Numerical Solution of the Compressible Navier-Stokes Equations." *J. Comput. Phys.* 131: 267–79.

Bassi, F., S. Rebay, G. Mariotti, S. Pedinotti, and M. Savini. 1997. "A High-Order Accurate Discontinuous Finite Element Method Fir Inviscid an Viscous Turbomachinery Flows." Edited by R. Decuypere and G. Dibelius. Technologisch Instituut, Antwerpen, Belgium.

Carlson, Jan-Reneé. 2011. "Inflow/Outflow Boundary Conditions with Application to Fun3d."

Carpenter, M., and C. Kennedy. 1994. "Fourth-Order 2N-Storage Runge-Kutta Schemes." NASA TM 109111.

Flad, David, Hannes Frank, Andrea D Beck, and Claus-Dieter Munz. 2014. "A Discontinuous Galerkin Spectral Element Method for the Direct Numerical Simulation of Aeroacoustics." In *20th Aiaa/Ceas Aeroacoustics Conference*, 2740.

Gao, J. S.; Warburton, Z.; Hesthaven. 2016. "Efficient Absorbing Layers for Weakly Compressible Flows." *Submitted to Journal of Scientific Computing*.

Gassner, Gregor J, and Andrea D Beck. 2013. "On the Accuracy of High-Order Discretizations for Underresolved Turbulence Simulations." *Theoretical and Computational Fluid Dynamics* 27 (3-4): 221–37.

Gassner, Gregor J., Andrew R. Winters, and David A. Kopriva. 2016. "Split form nodal discontinuous Galerkin schemes with summation-by-parts property for the compressible Euler equations." *J. Comput. Phys.* 327 (December): 39–66. https://doi.org/10.1016/j.jcp.2016.09.013.

Gassner, Gregor, and David A Kopriva. 2011. "A Comparison of the Dispersion and Dissipation Errors of Gauss and Gauss-Lobatto Discontinuous Galerkin Spectral Element Methods." *SIAM Journal on Scientific Computing* 33 (5): 2560–79.

Ghia, K. N.; Shin, U.; Ghia. 1982. "High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method." *Journal of Computational Physics* 48: 387–411.

Hennemann, Sebastian, Andrés M Rueda-Ramírez, Florian J Hindenlang, and Gregor J Gassner. 2021. "A Provably Entropy Stable Subcell Shock Capturing Approach for High Order Split Form Dg for the Compressible Euler Equations." *Journal of Computational Physics* 426: 109935.

Hindenlang, Florian, Gregor J Gassner, Christoph Altmann, Andrea Beck, Marc Staudenmaier, and Claus-Dieter Munz. 2012. "Explicit Discontinuous Galerkin Methods for Unsteady Problems." *Computers & Fluids* 61: 86–93.

Jameson, A., Wolfgang Schmidt, and Eli Turkel. 1981. "Numerical solution of the Euler equations by finite volume methods using Runge Kutta time stepping schemes." In *14th Fluid and Plasma Dynamics Conference*. Fluid Dynamics and Co-Located Conferences. American Institute of Aeronautics; Astronautics. https://doi.org/10.2514/6.1981-1259.

Kopriva, David A., and Gregor Gassner. 2010. "On the Quadrature and Weak Form Choices in Collocation Type Discontinuous Galerkin Spectral Element Methods." *Journal of Scientific Computing* 44 (2): 136–55.

Kopriva, David A, Stephen L Woodruff, and M Yousuff Hussaini. 2002. "Computation of Electromagnetic Scattering with a Non-Conforming Discontinuous Spectral Element Method." *International Journal for Numerical Methods in Engineering* 53 (1): 105–22.

Moser, Robert D, John Kim, and Nagi N Mansour. 1999. "Direct Numerical Simulation of Turbulent Channel Flow up to Re $\tau=$ 590." *Physics of Fluids* 11 (4): 943–45.

Niegemann, Jens, Richard Diehl, and Kurt Busch. 2012. "Efficient Low-Storage Runge–Kutta Schemes with Optimized Stability Regions." *Journal of Computational Physics* 231 (2): 364–72.

Persson, Per-Olof, and Jaime Peraire. 2006. "Sub-Cell Shock Capturing for Discontinuous Galerkin Methods." In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics; Astronautics. https://doi.org/10.2514/6.2006-112.

Roache, Patrick J. 2001. "Code Verification by the Method of Manufactured Solutions." *Journal of Fluids Engineering* 124 (1): 4–10.

Schmid, Peter J, Knud Erik Meyer, and Oliver Pust. 2009. "Dynamic Mode Decomposition and Proper Orthogonal Decomposition of Flow in a Lid-Driven Cylindrical Cavity." In *8th International Symposium on Particle Image Velocimetry*, 25–28.

Sod, Gary A. 1978. "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws." *Journal of Computational Physics* 27 (1): 1–31. https://doi.org/10.1016/0021-9991(78)90023-2.

Sonntag, Matthias, and Claus-Dieter Munz. 2017. "Efficient Parallelization of a Shock Capturing for Discontinuous Galerkin Methods Using Finite Volume Sub-Cells." *Journal of Scientific Computing* 70 (3): 1262–89.

Woodward, Paul, and Phillip Colella. 1984. "The numerical simulation of two-dimensional fluid flow with strong shocks." *Journal of Computational Physics* 54 (1): 115–73. https://doi.org/10.1016/0021-9991(84)90142-6.