A user guide for nsCouette

Daniel Feldmann, Jose Manuel López,
Markus Rampp, Liang Shi & Marc Avila

2nd January 2020

This is a hands-on introduction to **nsCouette**; A highly scalable software tool to integrate the full Navier–Stokes equations for incompressible fluid flows between deferentially heated and independently rotating concentric cylinders forward in time. It is based on a pseudospectral spatial discretisation and dynamic time-stepping. It is implemented in modern **Fortran** with a hybrid **MPI-OpenMP** parallelisation scheme and thus tailored to compute turbulent flows at high Reynolds and Rayleigh numbers. With a number of preconfigured **Makefiles** for different architectures, easy installation, simple handling of input parameters, portable **HDF5** output, visualisation tools, comprehensive documentation and internal quality assurance, **nsCouette** is designed to lower the barrier to entry to numerical research in highly-turbulent fluid flows.

An additional GPU-accelerated implementation (**CUDA**) for intermediate problem sizes as well as a basic version for turbulent pipe flow (**nsPipe**) are also provided. This guide can be used to get familiar with the prerequisites, compilation, running and structure of our code before using it for your own research project.

Contents

1. For the impatient

Basically, downloading, building, and running the code goes as easy as this:

```
git clone https://github.com/dfeldmann/nsCouette
make ARCH=make.arch.gcc-linux HDF5IO=no
export OMP_NUM_THREADS=4
mpiexec -np 2 ./nsCouette.x < nsCouette.in
```

However, for further details and explanations, examples and problem solving please take the time to read this document and follow the provided tutorials. If this doesn't help, feel free to contact us. We are happy to help and we hope you enjoy using **`nsCouette`**!

2. About Taylor-Couette

The Taylor-Couette (TC) set-up is one of the most famous paradigmatic systems for wall-bounded shear flows in general and maybe the most important one if you are interested in rotating shear flows in particular. For understanding the following chapters of this documentation, it might help to acquaint yourself with a bit of crucial terminology and basic concepts of the TC system as well as with the notation we will use throughout this documentation and in the source code of **`nsCouette`**.

Here and there throughout the manual, we will mention how the code can be used to include thermal convection in the TC system (Section 6.7) or change to a straight pipe flow set-up (Section C), the other very important canonical shear flow system in cylindrical coordinates.

2.1. Governing equations

We consider a fluid with constant properties (density ρ and kinematic viscosity ν) confined between two concentric cylinders as shown in figure 1. Rotating at least one of the cylinders causes a motion of the confined fluid; A fluid-dynamical system well-known as Taylor-Couette (TC) flow. The so generated fluid motion is governed by the incompressible

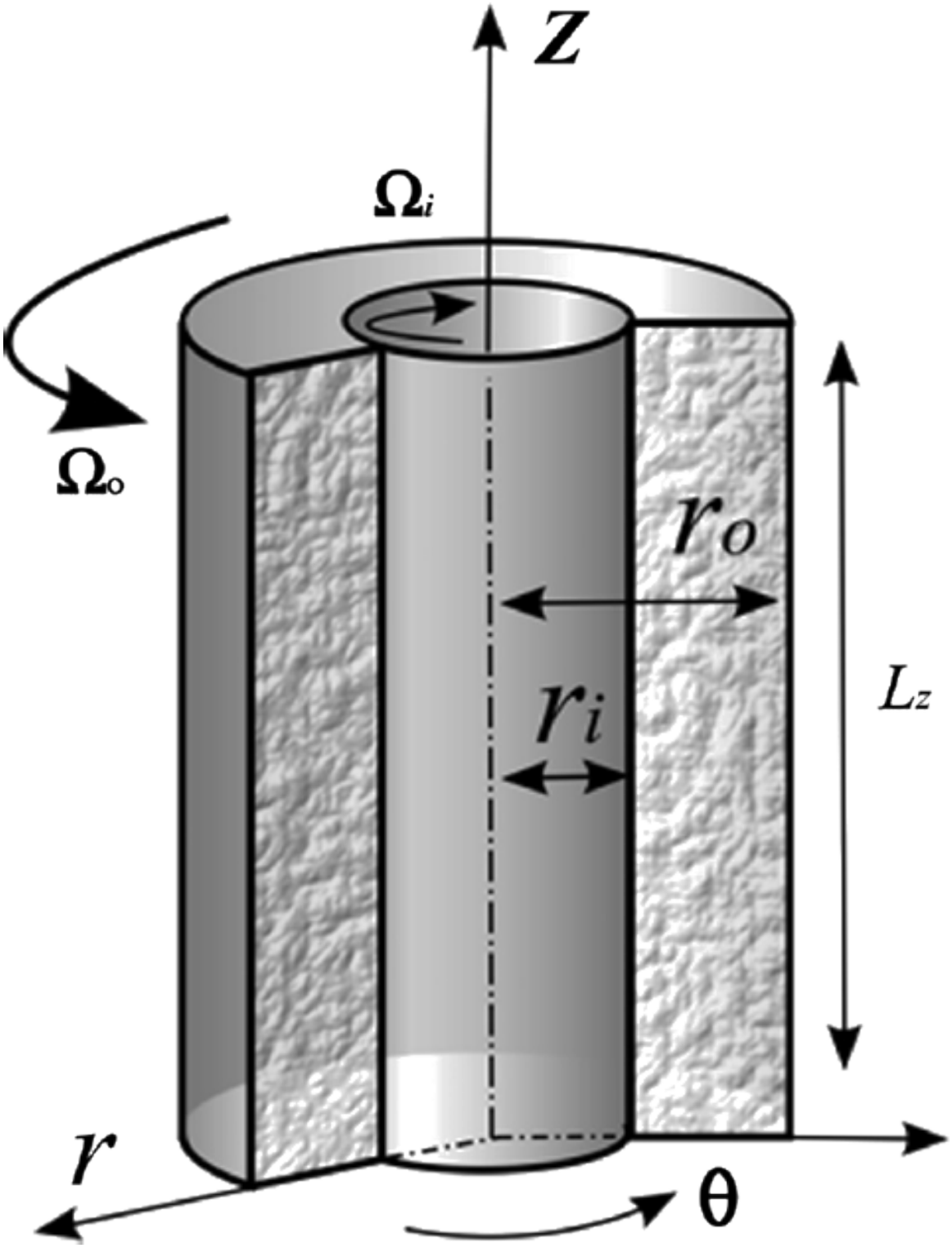


Figure 1: Schematic of the Taylor-Couette (TC) system in a cylindrical co-ordinate frame work (r, θ, z) and its relevant properties: Radius of the inner/outer cylinder wall ($r_{i,o}$), height of the computational domain (L_z), angular rotation speed of the inner/outer cylinder walls ($\Omega_{i,o}$). Sketch taken from our CAF paper [10].

Navier-Stokes equations

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \quad \text{and} \quad \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} \mathbf{u} = \frac{\boldsymbol{\nabla} p^*}{\rho} + \nu \boldsymbol{\Delta} \mathbf{u} \tag{1}$$

which describe the conservation of mass and momentum in the system. Here, t denotes the time, p^* the hydrodynamic pressure, and $\mathbf{u}$ the velocity vector, with its components u_r, u_θ and u_z pointing along the cylindrical coordinates in the radial (r), azimuthal (θ) and axial (z) direction.

For the version of **`nsCouette`** with thermal convection an additional equations for the temperature enters the stage, which couples to the momentum equation as described in section 6.7. For **`nsPipe`**, the code version to compute flow through a straight pipe geometry, the governing equations as described in detail in section C.1.

2.2. Control parameters and non-dimensional description

The TC geometry, as shown in figure 1, can be fully characterised using only two parameters. Its relative curvature is controlled by the radii ratio $\eta = r_i/r_o$ of the inner and outer cylinder, whereas its relative height is controlled by $\Gamma = L_z/d$, with $d = r_o - r_i$ being the gap-width between the inner and outer cylinder wall. With the characteristic length scale set to $d = 1$, the inner and outer radius of the domain can be calculated as

$$r_i = \frac{\eta}{1-\eta} \quad \text{and} \quad r_o = \frac{1}{1-\eta} \tag{2}$$

for any given η. For a fixed geometry, the flow between the cylinders can then be fully characterised by only two additional parameters, namely the Reynolds number

$$Re = \frac{u_{ref} \cdot d}{\nu} = \frac{2}{1+\eta} \left(Re_i - \eta Re_o \right) \tag{3}$$

and the rotation number

$$Re_\omega = \frac{2\omega_{ref} \cdot d}{u_{ref}} = (1-\eta) \frac{Re_i + Re_o}{Re_i - \eta Re_o}, \tag{4}$$

where u_{ref} is a characteristic reference velocity and ω_{ref} a characteristic angular velocity of the co-rotating reference frame, see e. g. [4] and [3]. The traditional Reynolds numbers that measure the dimensionless velocity of the inner and outer cylinders in the laboratory frame of reference, as e. g. used in the seminal experiments of Brandstäter et al. [1, 2], are given through

$$Re_i = \frac{r_i \omega_i d}{\nu} = \frac{u_i d}{\nu} \quad \text{and} \quad Re_o = \frac{r_o \omega_o d}{\nu} = \frac{u_o d}{\nu}. \tag{5}$$

Henceforth, here and in the source code, all variables will be rendered dimensionless using d, d^2/ν, and ν^2/d^2 as units for length, time, and reduced pressure ($p = p^*/\rho$), respectively. Thus, we obtain a non-dimensional form of the governing equations (1), in which Re_i and Re_o appear only through a Dirichlet boundary condition for the velocity vector

$$\mathbf{u}(r = r_{i,o}, \theta, z, t) = \begin{bmatrix} 0 & Re_{i,o} & 0 \end{bmatrix}^T \tag{6}$$

at the inner and outer wall of the cylinders. By doing this, we generate a impermeability and no-slip condition which models the solid wall. The unit for the velocity is already implicitly defined through our choice for unit length (d) and unit time (d^2/ν), and is thus given by ν/d.

For the version of **nsCouette** with thermal convection additional parameters appear in the equations to control the non-dimensional temperature gradient and the fluid properties, as described in sections 4.1.3 and 6.7. For **nsPipe**, the code version to compute flow through a straight pipe geometry, different boundary conditions, different driving mechanisms, and different control parameters appear in the equations as described in detail in section C.1.

3. Building the code

3.1. Hardware prerequisites

nsCouette is written in modern **Fortran** and, over time, has been ported to all major CPU-based high-performance computing (HPC) platforms. Amongst IBM Power, BlueGene and **x86_64** architectures – including a few generations of the prevalent Intel Xeon multi-core processors – it has also been ported to Xeon Phi (KnightsLanding), AMD EPYC (Naples, Rome) and ARMv8.1 (Marvell ThunderX2) platforms. Optimisation for the NEC SX-Aurora vector architecture is underway. The tutorials provided in section 7 of this user guide were designed to run on standard laptops and small clusters.

Additionally, we provide a GPU-accelerated (but basic) version of **nsCouette**, which is written in **C-CUDA** and runs on a single GPU. In case you are interested in using (or advancing) this version, you will need a **CUDA**-capable GPU device with compute capability 2.0 (or superior) and support for double precision arithmetic.

3.2. Software prerequisites

Building **nsCouette** requires a Linux operating system, standard compilers and only very few additional software libraries. All of them are commonly available as high-quality, open source software packages or as vendor-optimised tool chains. For the examples and tutorials (Section 7) presented in this user guide, we assume that you use a **bash** shell, although not necessarily required.

3.2.1. Compiler

You need a modern **Fortran** compiler which is **OpenMP3** compliant. Additionally, a corresponding **C** compiler is necessary. **nsCouette** has so far been tested with Intel's **ifort** and **icc** versions 12 through 15 as well as **PSXE2017** and **PSXE2018**, GNU's **GCC-4.7** to **GCC-4.9** and PGI's compilers version 14.

Optionally, you will need a suitable version of NVIDIA's **CUDA** toolkit in case you are interested in working with the subsidiary GPU-accelerated version of our code. Note, that in this case a suitable choice for the **CUDA** version highly depends on the hardware you are going to use. The GPU version of **nsCouette** has so far been tested with the **CUDA** toolkit version from 8.0 to 10.1. Free software and further information can be found here:

- https://gcc.gnu.org
- https://developer.nvidia.com/cuda-toolkit

3.2.2. Message passing interface

You need an **MPI** library which supports the thread-level **MPI_THREAD_SERIALIZED**. This means that the user code is multi-threaded, but calls to the **MPI** library are serialised. A fallback option for the minimum thread-level is implemented, which is supported by any **MPI** implementation. **nsCouette** has so far been tested with Intel's **MPI-4.1**, **5.0**, **5.1** as well as **IBM PE 1.3** and **1.5**.

Currently, the optional **CUDA** version of **nsCouette** is not parallelised to run on multiple nodes; it runs on a single GPU only. Free software and further information can be found here:

- https://www.open-mpi.org
- https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

3.2.3. Linear algebra

You need a serial **BLAS**/**LAPACK** library. **nsCouette** has so far been tested with Intel's **MKL-11.x**. The optional GPU version of **nsCouette** relies on custom **CUDA** kernels to perform linear algebra. For free software and furthe information see:

- https://en.wikipedia.org/wiki/LAPACK
- http://www.openblas.net
- http://math-atlas.sourceforge.net
- http://www.netlib.org

3.2.4. Fast Fourier transform

You need a serial but fully thread-safe **FFTW3** installation or equivalent. **nsCouette** has so far been tested with **FFTW-3.3** and also with the **FFT** implementation which comes with Intel's math kernel libraries **MKL-11.1** to **MKL-11.3**. Earlier versions of **MKL** will likely fail. The optional GPU version of **nsCouette** relies on the **cuFFT** library for computing Fourier transforms. For free software and further information see:

- http://www.fftw.org
- https://en.wikipedia.org/wiki/FFTW
- https://en.wikipedia.org/wiki/Math_Kernel_Library

3.2.5. Hierarchical data format

Optionally, you need an **MPI**-parallel **HDF5** library installation to enjoy flow field output in primitive variables (Section 4.5.3), which can be readily visualised using e. g. **ParaView** (Section 5.1) or **VisIt** (Section 5.2). **nsCouette** has already been tested with **HDF5-1.8.x** and **HDF5-1.10.0-patch1**. However, in case you don't need **HDF5** output for now or encounter difficulties in providing a suitable **HDF5** installation, you can ignore this software requirement and simply switch this feature off (**HDF5IO=no**) when you compile the code (Section 3.4.2).

The GPU version of **nsCouette** requires standard **HDF5** libraries and provides an output compatible with the **Fortran** code for visualisation. Free software and further information can be found here:

- http://www.hdfgroup.org/HDF5
- https://portal.hdfgroup.org/display/support
- https://en.wikipedia.org/wiki/Hierarchical_Data_Format

3.2.6. Software modules

If you are planing to run **nsCouette** on a common supercomputer at one of the national high-performance computing centres (e. g. in Germany LRZ, HLRS, JSC, HLRN), all of the above mentioned software packages will most likely be readily available as user-loadable environment modules. When the environment module for a particular application/version is loaded (unloaded), the user environment is changed such that this application becomes available (unavailable) on the command line without changing any environment variables manually. As a first step, you should make yourself familiar with the system you are going to work on and see what software is already available. The **module** command is a function which can take different arguments. Some useful basic combinations to start working with modules are listed in the following.

```
feldmann@fsmcluster:~/$ module list                          # list currently loaded packages
feldmann@fsmcluster:~/$ module avail                         # overview of available packages
feldmann@fsmcluster:~/$ module avail 2>&1 | grep -i intel # search for specific things
feldmann@fsmcluster:~/$ module load moduleName               # load package, default version
feldmann@fsmcluster:~/$ module switch moduleNameV2           # change to specific version
feldmann@fsmcluster:~/$ module whatis moduleName
feldmann@fsmcluster:~/$ module help moduleName
feldmann@fsmcluster:~/$ module purge                         # unload all packages
...
feldmann@fsmcluster:~/$ which mpiifort
feldmann@fsmcluster:~/$ mpiifort --version
```

3.2.7. Self-build libraries

If you are planing to run **nsCouette** on your laptop, desktop, or local institute cluster, you will most likely have to install some or all of the above mentioned software packages yourself, before you can build and run **nsCouette**. Regarding **MPI** and the required compilers, it is worth trying to install them using your favourite software package manager (e.g. **apt-get**, **rpm** and alike).

```
feldmann@darkstar:~/$ sudo apt-get install openmpi-bin libopenmpi-dev mpi-default-dev
feldmann@darkstar:~/$ sudo apt-get install gfortran gcc
feldmann@darkstar:~/$ gfortran --version
```

For the rest, it might become necessary to download the source files and build the libraries yourself. In appendix B, you will find detailed examples of how to correctly configure and build some of the required libraries (**FFTW3**, **zlib** and **HDF5**) for the use with **nsCouette** on our local **fsmcluster** at the ZARM institute. It should, however be clear, that these examples can only serve as a rough guideline to install all necessary software packages on your target Linux system.

3.3. Download the source files

Once you have checked that you have all the compilers and libraries installed on your target system, you can proceed downloading the **nsCouette** source files. Log into the machine where you want to install and run **nsCouette**. Make sure you are in your home directory and create the **nsCouette** working directory.

```
feldmann@darkstar:~/$ cd $HOME
feldmann@darkstar:~/$ mkdir nsCouette
feldmann@darkstar:~/$ cd nsCouette
```

This is the place where you should put the source files. This is also the place, where we choose to put all the case directories – one for each tutorial (Section 7) – so that we have everything together in one place. Note, however, that depending on the system you are working on, policy and quotas might require you to store your simulation data (i. e. the case directories) in designated file systems other than your home partition.

You either got the source files as a **tar** archive file from one of the authors or you can download the source files from our publicly available **github** repository. In the first case, copy the archive into the working directory you just created and unpack it.

```
feldmann@darkstar:~/nsCouette$ tar xfvz nsCouette.tar.gz
feldmann@darkstar:~/nsCouette$ ls
nsCouette nsCouette.tar.gz
```

The latter option requires a working **git** installation, which is a distributed version control system. Most of the time, **git** will either be readily available on your system or it can be installed using your favourite software package manager.

```
feldmann@darkstar:~/nsCouette$ sudo apt-get install git
...
feldmann@darkstar:~/nsCouette$ git clone https://github.com/dfeldmann/nsCouette
feldmann@darkstar:~/nsCouette$ cd nsCouette
feldmann@darkstar:~/nsCouette/nsCouette$ git checkout nsCouette-1.0
feldmann@darkstar:~/nsCouette/nsCouette$ git checkout nsCouette-gpu
feldmann@darkstar:~/nsCouette/nsCouette$ git checkout nsPipe-1.0
feldmann@darkstar:~/nsCouette/nsCouette$ git status
feldmann@darkstar:~/nsCouette/nsCouette$ git pull
```

Once you have **git** running, the **nsCouette** source file remote repository can than easily be cloned (downloaded) to your target system with one single command. This first step has to be done only once and by default you should be on the branch **nsCouette-1.0**. But you can easily switch to other branches using the **checkout** command to have access

to the pipe flow variant or the GPU-accelerated **CUDA** version. The **git** commands **status** and **pull** will help you to figure out on which branch you currently are and will get you the latest changes (e. g. bug fixes or new features) from the remote repository. A few more helpful hints on how to use **git** can be found in appendix D.

3.4. Compile the code

Once you have downloaded the source files (Section 3.3), you can proceed building the executable. Go to the top-level directory of the source files. Among other things, here you will find the following files and directories, which are important to compile **nsCouette**.

```
feldmann@darkstar:~/nsCouette/nsCouette$ ls
...
ARCH/     scripts/    Makefile    nsCouette.f90   perfdummy.f90
mod_fdInit.f90  mod_inOut.f90  mod_timeStep.f90  mod_fftw.f90    mod_hdf5io.f90
mod_myMpi.f90   mod_vars.f90   mod_getcpu.f90 mod_nonlinear.f90  mod_params.f90
...
```

The source code is organised in twelve **Fortran** files (**.f90**), as described in section 6.1 in more detail. Among other things, the directory **scripts** contains example files which will help you to easily set-up the correct environment variables which are necessary for building – and also running – the code. Have a look at them and make a copy of one of the templates which appears closest to your platform/situation and modify it accordingly using your favourite text editor (e. g. **vi**).

```
feldmann@darkstar:~/nsCouette/nsCouette/scripts$ ls
nsCouetteAtDarkstar.sh  nsCouetteAtFsm.sh  nsCouetteAtKonrad.sh  nsPipeAtKonrad.sh
submit_loadl.sh  submit_sge.sh  submit_slurm.sh
feldmann@darkstar:~/nsCouette/nsCouette/scripts$ cp n*Darkstar.sh nsCouetteAtMyPlatform.sh
feldmann@darkstar:~/nsCouette/nsCouette/scripts$ vi nsCouetteMyPlatform.sh
```

By modifying we mean, that you should load the desired modules (Section 3.2.6) and/or set all necessary library path variables correctly (Section 3.2.7). Also, you might find it convenient to create an alias in your **.bashrc** so that you can easily load the correct environment by simply typing **nsc** any time you log into your machine or open a new terminal window. Note, that you have to load this environment every time you want to (re-) compile the code and also every time you want to run the executable (Section 4).

```
feldmann@darkstar:~$ vi .bashrc
...
alias nsc="source $HOME/nsCouette/nsCouette/scripts/nsCouetteAtDarkstar.sh"
...
feldmann@darkstar:~$ source .bashrc
feldmann@darkstar:~$ nsc
```

3.4.1. Makefile settings

In general, the **Makefile** itself requires no editing by the user. It is, however, advisable to have a look at it at some point to get an idea of how **nsCouette** is structured and build. Instead of modifying the **Makefile**, all platform-specific settings should be fixed using the architecture files. You can find a variety of working examples for different systems in the respective directory. Have a look, copy the template which appears closest to your platform/situation and modify it accordingly using your favourite text editor (e. g. **vi**).

```
feldmann@darkstar:~/nsCouette/nsCouette/ARCH$ ls
make.arch.darkstar    make.arch.gcc-openblas  make.arch.Supermuc
make.arch.fsm  make.arch.Hydra    make.arch.XC30_sisu-cray
make.arch.gcc-atlas  make.arch.intel-mkl  make.arch.XC30_sisu-intel
make.arch.gcc-essl    make.arch.KNL-intel  make.arch.xl-essl
make.arch.gcc-linux  make.arch.nec-aurora
make.arch.gcc-mkl make.arch.pgi-mkl
feldmann@darkstar:~/nsCouette/nsCouette/ARCH$ cp make.arch.darkstar make.arch.myPlatform
feldmann@darkstar:~/nsCouette/nsCouette/ARCH$ vi make.arch.myPlatform
```

By modifying we mean, that you should configure the correct compiler and libraries according to your environment and also specify the desired compiler options (e. g. hardware specific optimisation flags). In case you use self-made libraries (see section 3.2.7 and appendix B), make sure to correctly set all necessary path variables (e. g. **`LD_LIBRARY_PATH`**). Please consult your local IT support in case of problems.

If everything is configured correctly, the executable can be (re-)build in the usual way by simply typing **`make`** and specifying the desired architecture file. It is recommended to always **`clean`** the build directory before each compilation.

```
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform clean
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform
...
feldmann@darkstar:~/nsCouette/nsCouette$ ls
ARCH/ darkstar/ myPlatform/ myPlatformGcc/ myPlatformIntel/
...
feldmann@darkstar:~/nsCouette/nsCouette$ ls darkstar/
nsCouette.x
...
```

If the code compiles correctly, a new sub directory is generated. It contains all the generated objects (**`.o`**) and modules (**`.mod`**) as well as the executable **`nsCouette.x`** itself. The subdirectory is named after the architecture file you specified (here e. g. **`ARCH=myPlatform`**). Specifying another architecture file, will generate another subdirectory with another executable. This way you are able to easily manage different builds side by side.

3.4.2. Compile-time options

Besides specifying the architecture file (Section 3.4.1), the **`make`** command also takes a few other options which help you to control the build process. The different options listed below can be arbitrarily combined. They are meant to switch off **`HDF5`** flow field output (Section 4.5.3), to include integrating the temperature for thermal convection and heat transfer (Section 6.7), and to build the code in debug mode (i. e. tightest debug settings of the compiler and no hardware specific optimisation). The respective default value is the first choice in the angled brackets.

```
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform HDF5IO=<yes|no>
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform CODE=<STD_CODE|TE_CODE>
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform DEBUG=<no|yes>
```

4. Running the code

Once you have compiled the executable (Section 3.4), you are almost ready to run a simulation. Create a case directory and make sure that you have at least the executable and a parameter input file (Section 4.1) in that case directory.

```
feldmann@darkstar:~/nsCouette/nsCouette$
feldmann@darkstar:~/nsCouette/nsCouette$ mkdir myCase01
feldmann@darkstar:~/nsCouette/nsCouette$ cp nscouette/myPlatform/nsCouette.x  myCase01/.
feldmann@darkstar:~/nsCouette/nsCouette$ cp nscouette/myPlatform/nsCouette.in myCase01/.
feldmann@darkstar:~/nsCouette/nsCouette$ cd myCase01
feldmann@darkstar:~/nsCouette/nsCouette/myCase01$ ls
nsCouette.in   nsCouette.x
feldmann@darkstar:~/nsCouette/nsCouette/myCase01$ ./nsCouette.x < nsCouette.in
```

Basically, the parameter input file is simply passed to the standard input when you call the executable. A few detailed examples are described in the tutorials (Section 7).

4.1. Parameter input file

In general, **nsCouette** has to be build only once on a system and the executable can be used for all your simulations within one research project. Most of the settings and parameters to control the simulation can be specified at run-time using a parameter input file. This file can have any name (here **nsCouette.in**) and it is simply passed to the standard input when calling the executable (here **nsCouette.x**). The parameter input file should contain all of the following **Fortran** namelists. However, the keywords within a namelist can appear in any order and individual keywords can also be omitted if not needed.

```
&parameters_grid
m_r    = 32                    ! Radial points           => m_r      grid points
m_th   = 16                    ! Azimuthal Fourier modes => 2*m_th+1 grid points
m_z0   = 16                    ! Axial Fourier modes     => 2*m_z0+1 grid points
k_th0  = 6.0                   ! Fundamental wavenumber  => L_th = 2*pi/k_th0
k_z0   = 2.6179938779914944d0 ! Fundamental wavenumber  => L_z  = 2*pi/k_z0
eta    = 0.868d0               ! Radii aspect ratio
alpha  = 0.0d0                 ! Distribution of radial points, 0=Chebyshev, 1=uniform,
    alpha<0 read from file
/

&parameters_physics
Re_i =   200.0d0           ! Inner cylinder Reynolds number
Re_o =  -200.0d0           ! Outer cylinder Reynolds number
Gr   =    50.0d0           ! Grashof number, Gr = Ra/Pr            [TE_CODE only]
Pr   =     0.71d0          ! Prandtl number                        [TE_CODE only]
gap  =     3.25d0          ! Gap size in cm                        [TE_CODE only]
gra  =   980.0d0           ! Gravitational acceleration in g/cm**3 [TE_CODE only]
nu   =     1.01d-2         ! Kinematic viscosity in cm**2 /s       [TE_CODE only]
/

&parameters_timestep
numsteps    = 10000        ! Number of timesteps to compute
variable_dt = T            ! Use a variable (T) or a constant (F) timestep size
init_dt     = 1.00d-5      ! Initial (T) or constant (F) size of timestep
maxdt       = 1.00d-2      ! Maximum allowed size of timestep (T)
Courant     = 0.25d+0      ! CFL safety factor
/

&parameters_output
fBase_ic = 'myCase01'   ! identifier for coeff_ (checkpoint) and fields_ (hdf5) files
```

```
dn_coeff = 2000          ! output interval [steps] for coeff (dn_coeff =-1 disables output)
dn_ke    = 100           ! output interval [steps] for energy
dn_vel   = 100           ! output interval [steps] for velocity
dn_Nu    = 100           ! output interval [steps] for Nusselt (torque)
dn_hdf5  = 1000          ! output interval [steps] for HDF5 output
dn_prbs   = 10           ! output interval [steps] for time series data at probe locations
prl_r(1)  = 0.20d0       ! radial probe locations (0 < r/d < 1)
prl_r(2)  = 0.50d0
prl_r(3)  = 0.80d0
prl_r(4)  = 0.20d0
prl_r(5)  = 0.50d0
prl_r(6)  = 0.80d0
prl_th(1) = 0.25d0       ! azimuthal probe locations (0 < th/L_th < 1)
prl_th(2) = 0.25d0
prl_th(3) = 0.25d0
prl_th(4) = 0.75d0
prl_th(5) = 0.75d0
prl_th(6) = 0.75d0
prl_z(1)  = 0.25d0       ! axial probe locations (0 < z/L_z < 1)
prl_z(2)  = 0.25d0
prl_z(3)  = 0.25d0
prl_z(4)  = 0.75d0
prl_z(5)  = 0.75d0
prl_z(6)  = 0.75d0
print_time_screen = 100 ! output interval [steps] for timestep info to stdout
/

&parameters_control
restart = 0              ! from scratch (0) or restrat from checkpoint file (1,2)
runtime = 86400          ! maximum wall-clock time for the job in seconds
/

&parameters_initialcondition
ic_tcbf = T ! Set Taylor-Couette base flow (T) or resting fluid (F), only when restart = 0
ic_temp = F ! Set temperature profile (T) or zero (F), only when restart = 0, only TE_CODE
ic_pert = T ! Add perturbation on top of base flow (T) or not (F), only when restart = 0
ic_p(1, :) = 4.0d-2, 0, 1 ! 1st perturbation: amplitude and wavevector (a1, k_th1, k_z1)
ic_p(2, :) = 6.0d-3, 1, 0 ! 2nd perturbation: amplitude and wavevector (a2, k_th2, k_z2)
ic_p(3, :) = 0.0d-0, 0, 0 ! 3rd perturbation: amplitude and wavevector (a3, k_th3, k_z3)
ic_p(4, :) = 0.0d-0, 0, 0 ! Add up to six user defined perturbations
/
```

4.1.1. Control the time stepper

The temporal integration scheme has been upgraded to a predictor-corrector method, see section 6.6 and Guseva et al. [5]. This enables a variable timestep size (Δt), which is dynamically controlled during runtime. This feature is of particular advantage, if the flow state is either suddenly modified (applying disturbances, changing rotation rates etc.) or naturally undergoes strongly transient dynamics.

Most aspects of the time stepper can be easily controlled on a user-level using the name list **`parameters_timestep`** in the parameter input file (see section 4.1). Here, the number of timesteps can be specified, which determines how many times the code runs through the main time integration loop. Additionally, you can choose between a user-specified constant Δt and a variable Δt. If you chose a variable timestep size by setting **`variable_dt=T`**, then Δt will be automatically updated every now and then according to the limits you can also specify in this name list — the care-free package! In case you start a simulation from scratch (**`restart=0`**) the dynamic adaptation process begins with the value you specified

at **init_dt** in the parameter file. However, in case you restart your simulation from an existing flow field file (**restart=<1|2>**), then the **init_dt** in the parameter file will be ignored and the dynamic adaptation process resumes with the last value for Δt which was stored within the velocity file. If you, on the other hand, chose a constant timestep size by setting **variable_dt=F**, then Δt will always be set to the value you specify for **init_dt** in the parameter and it will not change during the simulation. Further possibilities to control the time stepper, which however require rebuilding the code, are described in section 6.6.

Usually, you run the transient phase of a simulation with **variable_dt=T** and observe the development of Δt. As soon as you have reached a statistically steady state and want to calculate statistics or create output for movies and what not, then you resume the simulation with a constant Δt, which you set to a nice even value (**init_dt**) which fits your needs and is slightly below the minimum Δt of what you observed during the transient phase.

4.1.2. Setting the radial grid

The radial distribution of grid points can be specified at runtime by modifying the **alpha** parameter (α) included in the **parameters_grid** name list. The nodes are distributed as

$$r_j = \frac{1+\eta}{2(1-\eta)} + \frac{\operatorname{asin}\left(-\alpha\cos\left(\frac{\pi j}{m_r-1}\right)\right)}{2\operatorname{asin}(\alpha)} \quad \text{with} \quad j = 0,\ldots,m_r-1.$$

Note that α must have a value between 0 and 1. For $\alpha = 1$, the nodes are uniformly distributed, whereas for $\alpha = 0$, a Chebyshev grid with strong quadratic clustering near the walls is obtained. Radial grids whose behaviour falls between Chebyshev and equispaced grids are obtained if intermediate values of α are chosen.

Alternatively, the radial grid can be read from an external file. This option is enabled if α is set to a negative value. The file containing the radial grid point coordinated must be named **radial_distribution.in** and the points must be arranged in ascending order, i. e. from the inner to the outer cylinder.

When the simulation is started from a checkpoint file (see section 4.6), the code automatically detects if the radial grid has been modified and interpolates the solution to the new grid. This is performed by comparing the value of α in the input file (parameters of the current simulation) and the restart file (parameters of the simulation where the checkpoint file was generated). Note that if the grid of the checkpoint file is specified by an external file, this must be named **radial_distribution_old.in**. If both the new and old grids are specified in files, these must be named **radial_distribution.in** and **radial_distribution_old.in**, respectively, and the value of α in the input file must be different (i. e. a different negative number) than that in the restart file. This enables the code to identify that the grids are different and interpolation is required.

4.1.3. Setting boundary conditions

The boundary conditions (BC) in the two homogeneous directions (θ and z) are periodic; basically for all flow field variables. The BC at the solid cylinder walls can be controlled

through he name list **parameters_physics**. To model the impermeable but rotating cylinder walls of Taylor-Couette set-up, we impose Dirichlet boundary conditions for the velocity field. The two cross-stream velocity components (u_r and u_z) are set to zero everywhere at the walls. The streamwise velocity component (u_θ) is set to Re_i and Re_o everywhere at the inner and outer cylinder wall, respectively, which corresponds to the rotation speed of the respective wall, see eq. (5).

For the heat transfer version of **nsCouette**, additional Dirichlet BC are imposed for the temperature field to model the isothermal cylinder walls. The temperature difference between the inner and outer cylinder wall can be controlled by choosing the desired Rayleigh number (i. e. setting values for the Grashof and the Prandtle number) in the same name list.

4.1.4. Choosing initial conditions

Integrating the Navier-Stokes equations forward in time is a typical initial value problem. Each simulation run needs proper initial conditions for the full three-dimensional velocity field $\mathbf{u}(r, \theta, z, t_0)$ and – optionally – also for the temperature field $T(r, \theta, z, t_0)$. Basically, you have two options how to specify initial values for the flow field variables, which can be controlled using the keyword **restart** in the parameter name list **parameters_control**.

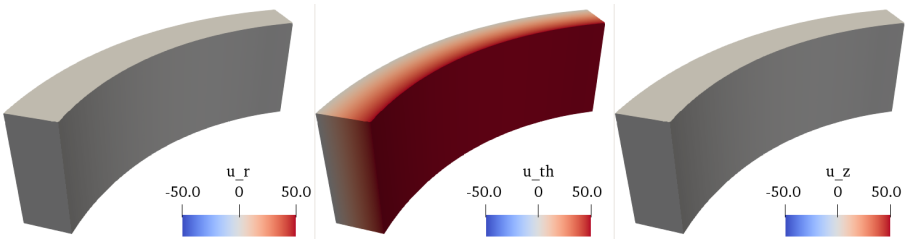
- You can choose to generate initial conditions in the code itself (**restart=0**). Starting from scratch requires no additional files and data other than the executable itself (**nsCouette.x**) and a parameter input file (**nsCouette.in**). You can run a simulation right away, as exemplarily shown in the first tutorial in section 7.2. However, there are a few different options how to generate initial conditions, as described below, and some more will be implemented in the future.

- Or you can choose to read initial conditions from an existing file you have at hand (**restart=<1|2>**). This file has to be a Fourier coefficient file in binary format (Section 4.5.2), which you might have from a similar simulation case or from a former run of the same case. See section 4.6 for details of the checkpoint-restart mechanism.

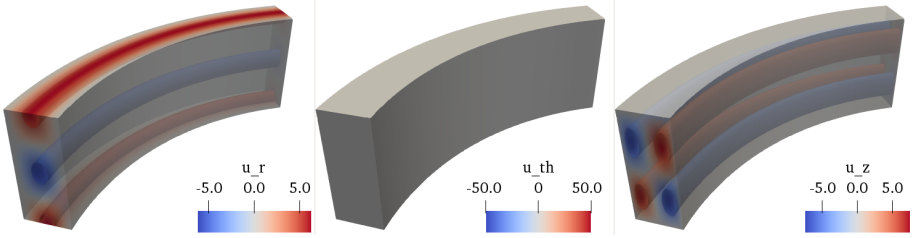
You can choose to prescribe a resting fluid as initial condition by setting **ic_tcbf=F**. To prescribe the (Taylor)-Couette analytical solution as base flow set **ic_tcbf=T**. No matter what the base flow is, you can disturb it by superimposing up to six different perturbations by setting **ic_pert=T** and than defining an amplitude and wave number vector for each perturbation. The perturbations are designed to fulfil the no slip boundary condition at the solid walls and to be divergence free by construction. Figure 2 shows some useful examples of different initial conditions at $Re_i = 50$.

4.2. On your laptop/desktop

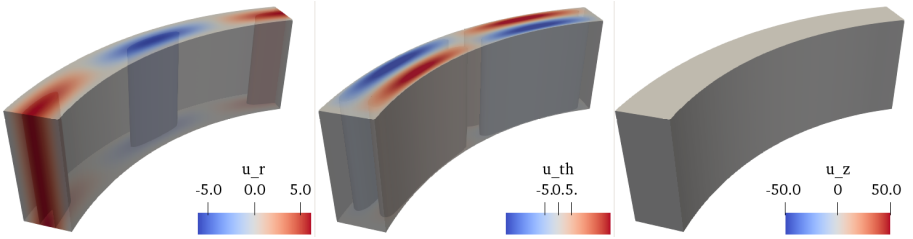
On your local machine you simple need to open a terminal and go to one of your case directories. If you do not have one, create one (Section 4). Here, we use the first tutorial case 7.2 as an example. Copy the executable and start the simulation with only one **MPI** process and only one **OpenMP** thread by simply typing:



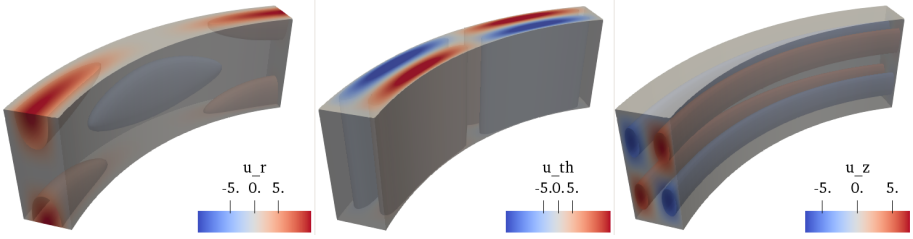
(a) Taylor-Couette analytical solution, no perturbation.



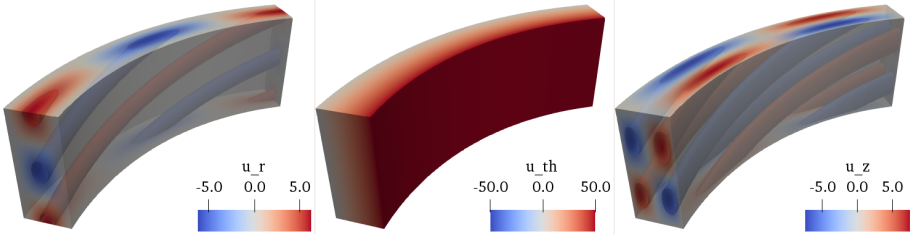
(b) Resting fluid, perturbed in mode $(l_{\theta,1}, n_{z,1}) = (0, 1)$ with amplitude $a_1 = 10^2$.



(c) Resting fluid, perturbed in mode $(l_{\theta,1}, n_{z,1}) = (1, 0)$ with amplitude $a_1 = 10^2$.



(d) Resting fluid, perturbed in modes $(0, 1)$ and $(1, 0)$ with amplitudes $a_1 = a_2 = 10^2$.



(e) Taylor-Couette base flow, perturbed in mode $(l_{\theta,1}, n_{z,1}) = (1, 1)$ with amplitude $a_1 = 10^2$.

Figure 2: Useful examples of different initial conditions at $Re_i = 50$. Shown are iso-contours (red/blue: $u_\alpha = \pm Re_i/10$) for all three velocity components.

```
feldmann@darkstar:~$ cd nsCouette/tc0040
feldmann@darkstar:~/nsCouette/tc0040$ cp ../nscouette/darkstar/nsCouette.x .
feldmann@darkstar:~/nsCouette/tc0040$ export OMP_NUM_THREADS=1
feldmann@darkstar:~/nsCouette/tc0040$ ./nsCouette.x < nsCouette.in
```

This starts a completely serial run without no parallelisation at all. To run **nsCouette** with more than one **MPI** task, here e. g. four, type

```
feldmann@darkstar:~/nsCouette/tc0040$ export OMP_NUM_THREADS=1
feldmann@darkstar:~/nsCouette/tc0040$ mpiexec -np 4 ./nsCouette.x < nsCouette.in
```

This starts a parallel simulation purely based on **MPI** without any **OpenMP** parallelisation involved. For making use of the hybrid **MPI-OpenMP** parallelisation strategy, the global variable **OMP_NUM_THREADS** has to be set to a positive value other than unity. See section 4.4 for a detailed description of how to control the hybrid scheme.

Note, that the number of **MPI** tasks can be freely selected at runtime with the only restriction that it must evenly divide the number of radial grid points ($N_r \equiv$ **m_r**), which are selected in the **nsCouette.in** input file, see Section 4.1. There is no such restriction regarding the total number of Fourier modes (**m_f**), see section 6.4 and appendix A). In these cases, where the number of chosen **MPI** tasks does not evenly divide **m_f**, a few dummy modes are automatically added. This causes a small imbalance and waste of computational resources, which is reported at the beginning of the run, but gives enormous flexibility in choosing spatial discretisation and degree of parallelisation.

4.3. On a cluster

Example batch submission scripts for the Sun Grid Engine (SGE) and SLURM with Intel MPI, and IBM LoadLeveller with IBM MPI are provided in the subdirectory **scripts**. Note that these scripts can serve only as a rough guideline. They worked on a number of HPC systems but most probably require some adaptation to a specific batch environment.

In this example, a number of **numsteps=10000** timesteps would be attempted to run, but the code would stop after a maximum **runtime=86400** seconds (24 h) of elapsed wall-clock time depending on what limit is reached first. We have implemented an intrinsic safety margin corresponding to the average number of seconds it takes to perform two timesteps.

If **OpenMP** is enabled – this is the default behaviour when compiling the code – environmental variables must be used to control the number of threads you would like to use and other related stuff. For example, specify

```
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
export OMP_SCHEDULE=static
export OMP_STACKSIZE=256M
```

if you want to use four **OpenMP** threads. The second line maps the threads to (physical) cores. The fourth line sets the stack size; experiment with larger or smaller values if unexpected segmentation faults (**SEGFAULT**) occur or if the memory per core is scarce, respectively. These are just basic usage guidelines for **MPI** and **OpenMP**. For an advanced use of **MPI** and **OpenMP** (e. g. using **bash** scripts) or in case of problems, please consult your local IT support.

4.4. Control the hybrid parallelisation scheme

The hybrid parallelisation scheme of **nsCouette** relaxes the limit imposed by the one-dimensional (1d) slab decomposition on the maximum number of processor cores and maps naturally to the prevailing multi-node, multi-core HPC architectures (See section 6.8). Specifically, the flexibility to chose an appropriate combination for the number of **MPI** tasks per node ($N_{T/N}$) and the number of **OpenMP** threads per **MPI** task ($N_{T/T}$) over time has proven key for achieving good performance across a rather wide range of node architectures. Figure 3 illustrates the performance of **nsCouette** on different architectures ranging from 20 to 64 physical cores per compute node. Note, that $N_{T/N}$ is typically some-

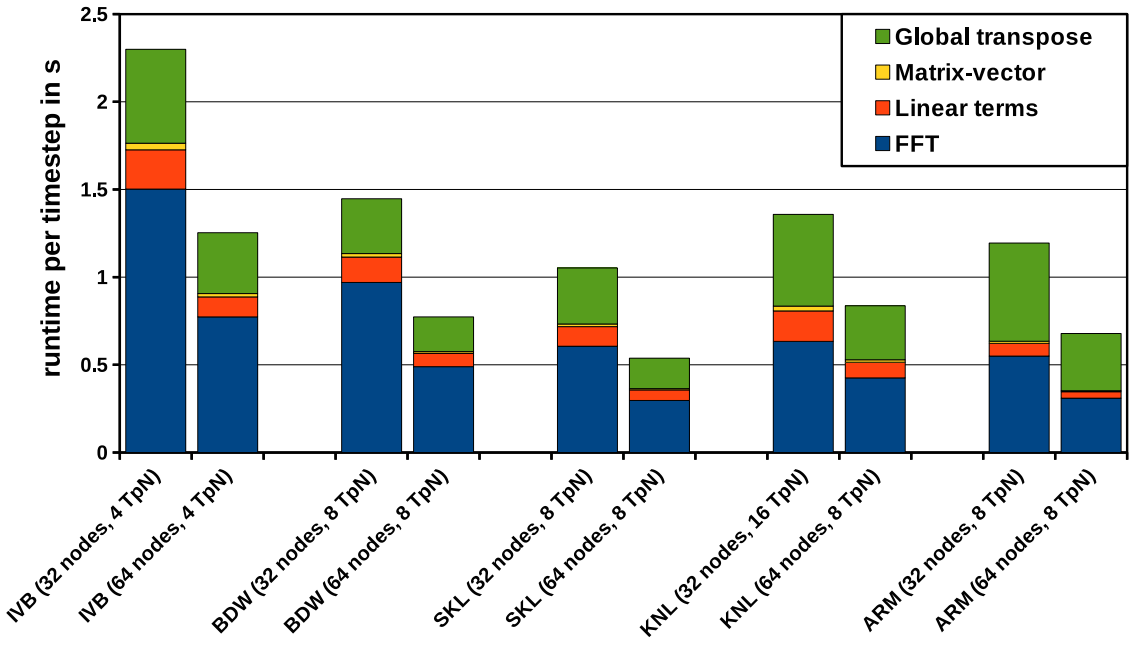


Figure 3: Runtime per timestep and breakdown into the main algorithmic components (different colours) of a typical **nsCouette** run ($N_r = 512$ and 513×1025 Fourier modes) computed on 32 and 64 dual-socket nodes of various HPC clusters, using a platform-specific number of **MPI** tasks/node (TpN). IVB: Intel Xeon E5-2680v2 (IvyBridge), 20 cores/node. BDW: Intel Xeon E5-2698v4 (Broadwell), 40 cores/node. SKL: Intel Xeon 6148 (Skylake), 40 cores/node. KNL: Xeon Phi 7230 (KnightsLanding), 64 cores/node. ARM: Marvell ThunderX2 ARM v8.1, 64 cores/node. The IVB and BDW clusters employ a Mellanox InfiniBand FDR network (56 $^{\mathrm{Gbit}}/_{\mathrm{s}}$), whereas SKL and KNL use Intel OmniPath (100 $^{\mathrm{Gbit}}/_{\mathrm{s}}$). The ARM cluster is interconnected with Cray Aries (80 $^{\mathrm{Gbit}}/_{\mathrm{s}}$). **nsCouette** was built using platform-optimised software tool chains (i. e. compilers and libraries) but no platform-specific Optimisation of the source code was performed. Corresponding architecture files (**nsCouette/ARCH**) are shipped with the code (Section 3.4.1).

thing like four or eight and the product ($N_{T/N} \times N_{T/T}$) is supposed to be equal the total number of physical cores per node. The most critical performance issue when running the code in hybrid mode (i. e. with **OpenMP** enabled), can be an improper pinning of the **MPI** tasks and **OpenMP** threads to the CPU cores. We suggest the basic settings above based on the **OMP_PLACES** environment variable from the **OpenMP** standard. However, different compilers and node architectures (NUMA, Hyper-threading, ...) might require specific settings. The actual mapping can be checked by examining the standard output

of **nsCouette**. Consider the following example. You are working on a machine with 16 dual-socket compute nodes. Each compute node has two CPUs with 12 physical cores each. So there are 24 physical cores per node and 384 physical cores in total. Then it would be a good first try to run **nsCouette** with 32 **MPI** tasks – i. e. two tasks per node and one task per CPU, respectively – and twelve **OpenMP** threads per **MPI** task and CPU, respectively. For this example, the standard output of **nsCouette** should look something like this:

```
feldmann@darkstar:~/nsCouette/tc0040$ export OMP_NUM_THREADS=12
feldmann@darkstar:~/nsCouette/tc0040$ mpiexec -np 32 ./nsCouette.x < nsCouette.in
...
TASKS: 32 THREADS:12 THIS: 0 Host:nid01226 Cores:  0  1  2  3  4  5  6  7  8  9 10 11
TASKS: 32 THREADS:12 THIS: 1 Host:nid01226 Cores: 12 13 14 15 16 17 18 19 20 21 22 23
TASKS: 32 THREADS:12 THIS: 2 Host:nid01227 Cores:  0  1  2  3  4  5  6  7  8  9 10 11
TASKS: 32 THREADS:12 THIS: 3 Host:nid01227 Cores: 12 13 14 15 16 17 18 19 20 21 22 23
TASKS: 32 THREADS:12 THIS: 4 Host:nid01228 Cores:  0  1  2  3  4  5  6  7  8  9 10 11
TASKS: 32 THREADS:12 THIS: 5 Host:nid01228 Cores: 12 13 14 15 16 17 18 19 20 21 22 23
...
```

It is important to note, that the individual lines can appear in random order and, that the numbering scheme for the cores is due to some low-level settings in the operating system and hence is highly machine specific. Tools like **cpuinfo** (comes with Intel **MPI**) or **likwid-topology** can be used to check the numbering scheme of a specific computer.

It is recommended to map at least one **MPI** task to each NUMA domain (CPU socket) and specify **OMP_NUM_THREADS** to the number of physical cores per node divided by the number of **MPI** tasks you want map to each compute node. Hyper-threading is not beneficial for **nsCouette**. On contemporary HPC clusters with two Intel or AMD CPUs per node, we typically use two **MPI** tasks per compute node and set:

```
export OMP_NUM_THREADS=8  # 1 node = 2 x ( 8-core CPU, Intel Xeon E5-2670, SandyBridge)
export OMP_NUM_THREADS=16 # 1 node = 2 x (16-core CPU, Intel Xeon E5-2698v3, Haswell)
export OMP_NUM_THREADS=20 # 1 node = 2 x (20-core CPU, Intel Xeon Gold-6148, Skylake)
...
```

Note, however, that in the part of the code, where the radial derivatives of the velocities are Fourier-transformed to compute the nonlinear terms, the **OpenMP**-parallelism is limited by **3*mp_r**. In case of thermal convection (Section 6.7), it is limited by **4*mp_r**, because of the additional array for the temperature field. Recall, that **mp_r** is the number of radial grid points per **MPI** task. For the interested reader, this part of the code can be identified by the performance **fft** in the **mod_nonlinear.f90** source file. Thus, if the maximum number of **MPI** tasks is chosen for a given number of radial points (and hence **mp_r=1**), it is advisable to use four or eight **MPI** tasks per node on large multi-core CPUs, and decrease **OMP_NUM_THREADS** accordingly, in order not to waste resources. A forthcoming release aims at alleviating this efficiency bottleneck by supporting threaded FFTs.

4.5. Output

4.5.1. Time series data

If the program runs correctly, it immediately generates a set of data files, which contain time series data of different quantities. These files are updated continuously during the

entire simulation run and the sampling frequency for the time series output can be independently specified using the keywords **dn_ke**, **dn_prbs** etc. in the parameter input file, as detailed in section 4.1.

```
feldmann@darkstar:~/nsCouette/tc0040$ ls
ke_mode  ke_total  probe01.dat  probe03.dat  probe05.dat  torque
ke_th    ke_z      probe02.dat  probe04.dat  probe06.dat  Nusselt
...
```

The first data column always contains the physical time in viscous units (d^2/ν). See section 2.2 for more details on non-dimensionalisation. The other columns are the relevant physical quantities like modal kinetic energy, integral torque, as well as pressure, velocity and temperature at particular probe locations in the flow field. The probe locations can be specified using the respective keywords in the namelist **parameters_output**, as detailed in section 4.1. Since these files are simple text files, the containing time series data can be easily read, plotted and processed using any common editor, command line tool and plotting programme (e. g. **gnuplot** or **xmgrace**). The tutorials in section 7 provide further details and examples of how to visualise and interpret the respective time series.

4.5.2. Fourier space flow field coefficients

As described in detail in section 6.4, **nsCouette** is based on a spectral discretisation using a Fourier-Galerkin expansion in two of the three spatial directions. Therefore, the entire information about an instantaneous flow field snapshot is stored and treated in terms of spectral coefficients. Input and output of the spectral coefficients is documented here in this section, whereas optional output in primitive variables ($\mathbf{u}, p, T$) is discussed in section 4.5.3.

The spectral coefficients are dumped to individual files (**coeff_**) for each time step at a user-specified output interval. For a negative interval (**dn_coeff=-1**), continuous output is disabled. However, in any case, one coefficient file is written after the very last timestep of a correctly terminated simulation run. The coefficient files, also called checkpoint files, are always written in binary **MPI-IO** format. They are meant to be read by **nsCouette** itself or by derived post-processing tools using the parallel I/O infrastructure of **nsCouette**. A small human-readable text file (**coeff_*.info**) containing all relevant metadata is written automatically with each coefficient file.

The binary coefficient files will be generated every now and then, according to the setting of the **dn_coeff** parameter in the input file (**nsCouette.in**), see section 4.1. For example, if you specify to output a coefficient file every 2000 timesteps and run the simulation in total for 11 000 timesteps starting from scratch at timestep zero, in the end there will be six coefficient files in total.

```
feldmann@darkstar:~/nsCouette/myCase01$ ls coeff*
coeff_myCase01.00002000     coeff_myCase01.00002000.info
coeff_myCase01.00004000     coeff_myCase01.00004000.info
coeff_myCase01.00006000     coeff_myCase01.00006000.info
coeff_myCase01.00008000     coeff_myCase01.00008000.info
coeff_myCase01.00010000     coeff_myCase01.00010000.info
coeff_myCase01.00011000     coeff_myCase01.00011000.info
```

These files contain the Fourier coefficients representing the velocity field and optionally the temperature field at one particular timestep; also called snapshots. Based on these

coefficient files, **nsCouette** implements an easy-to-use checkpoint-restart mechanism for handling long-running simulations. How the **coeff** files are used as a checkpoint to continue a simulation and how the restart mechanism is controlled, is detailed in section 4.6.

The coefficient files can also easily be used for post-processing with tools the user has to implement or derive from main code himself. One example of such a post-processing tool is provided in our repository.

```
feldmann@darkstar:~/nsCouette/nsCouette/postproc$ ls *.f90
mod_preAnalysis.f90
waveSpeed.f90
```

Reading the **coeff** files with external tools (e. g. **Python** or **PLplot**) is of course possible but, however, not straightforward and has to be implemented by the user himself. To enable easy and straightforward visualisations of the flow field, **nsCouette** also implements optional **HDF5** output, which is described in the following section.

4.5.3. Physical space flow field data

The primitive variables – namely the velocity (u_r, u_θ, u_z), the pressure (p) and optionally the temperature (T) – can be written in **HDF5** format to individual files for each timestep at a user-specified output interval. The additional **HDF5** output is optional and completely independent of the output of spectral coefficients (Section 4.5.2). All relevant metadata for each flow field snapshot are written to small **xdmf** files in order to facilitate easy analysis with common visualisation tools like **ParaView** and **VisIt**.

The **HDF5** functionality is switched on by default and can be disabled at compile time (**make HDF5IO=<yes|no>**), as detail in section 3.4.2. In that case, only spectral snapshots in binary format (Section 4.5.2) will be written. However, if the **HDF5** functionality is enabled, **nsCouette** will generate **fields*.h5** files every now and then, according to the setting of the **dn_hdf5** parameter in the input file (**nsCouette.in**), see section 4.1. For example, if you specify to output snapshots every 2000 timesteps and run the simulation in total for 11 000 timesteps starting from scratch at timestep zero, in the end there will be five physical flow field files in total.

```
feldmann@darkstar:~/nsCouette/myCase01$ ls fields*
fields_myCase01_00002000.h5       fields_myCase01_00002000.xmf
fields_myCase01_00004000.h5       fields_myCase01_00004000.xmf
fields_myCase01_00006000.h5       fields_myCase01_00006000.xmf
fields_myCase01_00008000.h5       fields_myCase01_00008000.xmf
fields_myCase01_00010000.h5       fields_myCase01_00010000.xmf
```

The **HDF5** format can be readily read, manipulated and visualised with a lot of different software tools (e. g. **Python**, **PLplot** and many more). Here, we discuss and shortly introduce two common open-source solutions; i. e. **ParaView** (Section 5.1) and **VisIt** (Section 5.2). Both tools allow loading sequences of **xdmf** files produced by **nsCouette**, which enables the user to interactively perform comprehensive visual and quantitative analysis of the flow field and to easily generate videos and animations. Sample scripts based on the **Python** interface of **VisIt**, as well as a custom-made **ParaView** filter for handling the cylindrical coordinate system are distributed with **nsCouette**. **VisIt** has a built-in operator for handling cylindrical coordinates. A detailed visualisation tutorial is included in section **??**.

4.6. Checkpoint-restart mechanism

On large HPC systems, the runtime of a single batch job is usually limited to something like 12 h or 24 h. In order to enable long-running simulations, **nsCouette** implements a semi-automatic checkpoint-restart mechanism, where the initial conditions for the next run will be read from a flow field snapshot, which was written at the end of the preceding run. Regardless of what you specify for **dn_coeff** in the parameter input file (Section 4.1), a Fourier coefficient file (Section 4.5.2) is written as a checkpoint at the very end of any correctly terminated simulation run. Additionally, a small text file named **restart** is created, which is required to automatically continue the simulation in a next run. It contains the same information as the corresponding ***.info** file.

By setting **restart=1** in the parameter input file (Section 4.1) the simulation takes the information from the **restart** file and uses the specified checkpoint file as the new initial condition, rather than starting a new run from scratch, which corresponds the default behaviour (**restart=0**). The initial time for the new simulation run is that of the initial condition and the generated time series data (Section 4.5.1) is appended to the existing output files.

Hence, the checkpoint-restart mechanism of **nsCouette** allows submitting a chain of (many) individual jobs to a batch system at once and thus facilitates handling of long-running simulations. Basic functionalities of the batch system (e. g. options like **-hold_jid** for **SGE** or **-dependency** for **slurm**) can be used to express the chain-type dependency of the individual jobs.

It is important to note, that the **restart** file is only written if a run has finished successfully. If, on the contrary, the run terminates prematurely (e. g. due to a hardware issue, or a numerical or code-specific problem), the last valid coefficient file can serve as a checkpoint for restarting the run. In such a case, the **restart** file has to be created manually, simply by copying the corresponding **coeff_*.info** file.

A third initialisation option (**restart=2**) is also available, which allows to start the simulation from a checkpoint file, but resetting the physical time and timestep counter to zero and creating new/fresh output files for the time series data. This might be useful if you want to change, for example, resolution within one simulation case or if you want to create a new simulation case with different control parameters based on initial conditions you created in a former simulation case.

In any case the grid size of the continued simulation can be freely modified by adapting the corresponding parameters in the **nsCouette.in** file, but the number of radial grid points (N_r, **m_r**) must, of course, remain to be divisible by the number of processors (See also section 6.8).

5. Post-processing

5.1. Visualisation with ParaView

To visualise and analyse instantaneous flow field data with the open source software tool **ParaView** (see figure 4) just go through the introductory steps below and follow our tutorials in section 7. The tutorials also provide a few ready-to-use state files (***.pvsm**) to start with. Note, however, that this is just a collection of very basic quick-start recipes. For

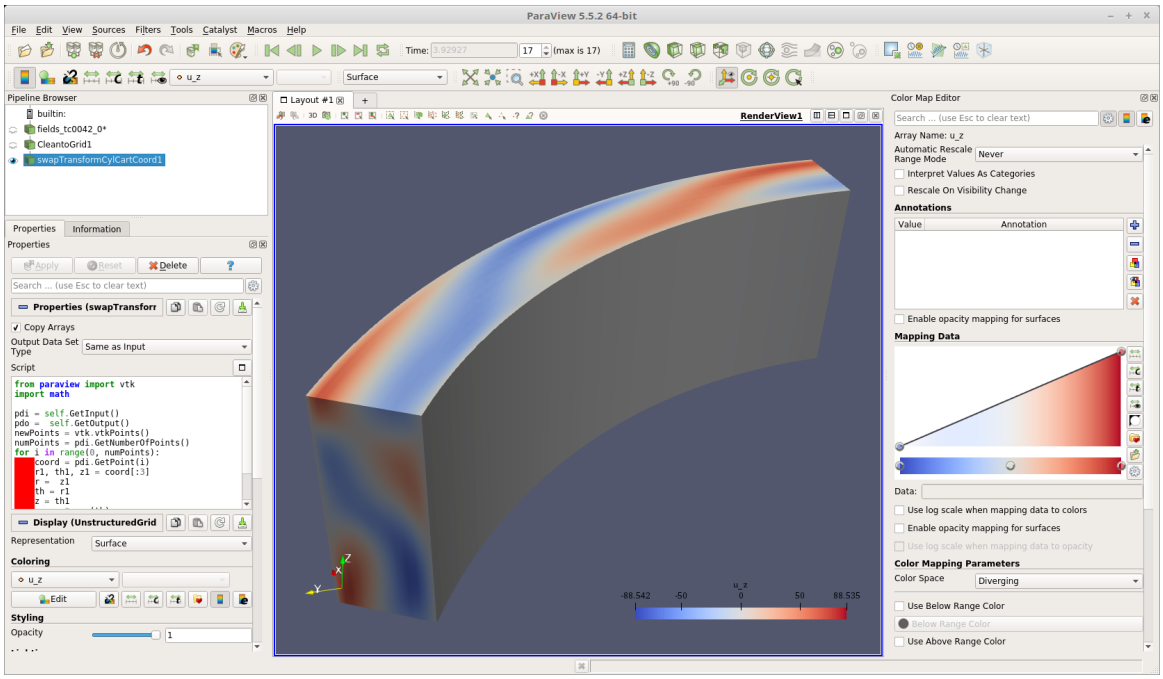


Figure 4: Screenshot of the **ParaView** GUI and the first basic steps to visualise and analyse the **HDF5** flow field output from **nsCouette**. Follow our tutorials in section 7 for further details.

more information, please consult the official documentation [9] and the pertinent online discussion forums.

Preparation

- First of all, download a recent pre-compiled version from the official web page and follow the instructions to install and run it. Or download the source files and build it yourself.
- Once you have **ParaView** running, you need to load a custom-made **Python**-filter, which comes with **nsCouette**. This filter can be used to perform a coordinate transformation for every state file you load into **ParaView**. This is a crucial step for correct three-dimensional rendering of the flow field, since **ParaView** only knows Cartesian coordinates (x, y, z), whereas the flow field data in the **HDF5** output files is defined in a cylindrical coordinate system (r, z, θ), see section 4.5.3. To add the custom filter to the list of known filters do the following:
  1. Launch **ParaView**
  2. Chose from the menu **Tools** → **Manage custom filters**
  3. Press **Import** in the upcoming window
  4. Browse to **$HOME/nsCouette/nsCouette/visualisation**
  5. Select the file **swapTransformCylCartCoord.cpd**
  6. Press **close**

Now the filter is known to **ParaView** and can be used any time you open this installation. To perform a coordinate transformation choose **Filters** → **Alphabetical** → **swapTransformCylCartCoord** from the menu.

Loading flow field data

- Once **ParaView** is installed and the transformation filter has been successfully added, you can begin loading flow field data. You should go to the case directory where you have stored the **HDF5** output files (here one of the first tutorial cases section 7.4)

```
feldmann@darkstar:~/$ cd nsCouette/tc0042
feldmann@darkstar:~/nsCouette/tc0042$ paraview &
```

  and start **ParaView**.

- Once the **ParaView** main window has appeared (something like in fig. 4), chose from the menu **File** → **Open** or click on the yellow folder icon in the top left corner to open the file selection dialogue. There you will see a list of the available **HDF5** files

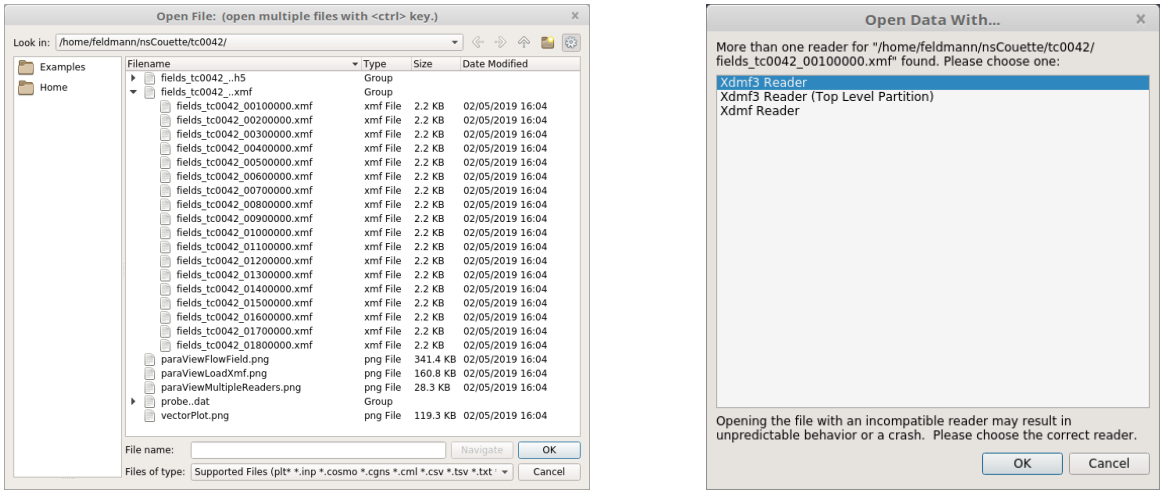


Figure 5: Screenshot of the file open dialogue window (left) where you can select the **xdmf** files, which act as a reader or interface through which **ParaView** access the flow field data contained in the **HDF5** files. If there is more than one **xdmf** reader, chose the first on in the appearing dialogue window (right).

  and the corresponding **xdmf** files as shown in figure 5. The **xdmf** files are simple text files, which contain meta data about the content of the **HDF5** container, see also section 4.5.3. The **xdmf** files act as a reader or interface through which **ParaView** access the data in the **HDF5** container. Select one single or a group of several **xdmf** files and press **OK**. Note, that all files with the same base name are automatically grouped. So if there are many flow field files in this directory, you can easily load the entire time series of snapshots by selecting the group instead of a single **xdmf** file. Note, however, that loading a large number of snapshots might take very long for large simulations!

- If a new dialogue window appears (fig. 5 right) after selecting snapshots and pressing **OK**, then select the option **Xdmf3 Reader**; but not **Xdmf3 Reader (Top Level Partition)** or anything else. Otherwise ignore this step.
- Now press **Apply** in the main **ParaView** window to actually load the selected snapshots. Note, that, since **ParaView** is designed to deal with large data sets, no action or data manipulation takes actually place until you finally hit the **Apply** button.
- Note, that data extraction, manipulation and plotting in **ParaView** is accomplished through so-called filters. You find a complete list of all available filters by choosing **Filters** → **Alphabetical** from the menu. The pipe line browser on the left shows all loaded data sets and all filters applied to it as individual objects or instants in an hierarchical order. Always make sure that you select/highlight the correct instant to which you want to apply the next filter operation.
- Once the data set has been loaded, you should first apply the following two filters in exactly this order:
  1. Select **Filters** → **Alphabetical** → **Clean to Grid** from the menu and than hit the **Apply** button.
  2. Select **Filters** → **Alphabetical** → **swapTransformCylCartCoord** from the menu and than hit the **Apply** button to perform the transformation from cylindrical to Cartesian coordinates.
- Now recenter the view by clicking the **Zoom to data** button to obtain something similar to what is shown figure 4.

Visualise flow field data

- Choose the quantity you are interested in. By default, this might be the pressure p. To get something similar as shown in figure 4, select the e. g. the axial velocity component u_z.
- If you have loaded more than one snapshot, you can simply hit the green **Play** button to show a movie representation of how the flow field changes with time. Note, that this process might be very slow in case of large simulations.
- To extract iso-surfaces for different velocity components, as shown for example in figure 12, select the instant named **swapTransformCylCartCoord** in the pipe line browser, choose **Filters** → **Alphabetical** → **Contour** from the menu and hit **Apply**. Select the velocity component you are interested in and modify the the contour levels in the properties panel directly below the pipe line browser.
- To create a vector representation of the cross-stream velocity components as for example in figure 12, use the **Glyph** filter as exemplarily shown in the **ParaView** state file **vectorPlot.pvsm** which comes with the second tutorial described in section 7.3.

5.2. Visualisation using `VisIt`

As an alternative to **`ParaView`** (section 5.1), you can also use the open source software **`VisIt`** (fig. 6), to visualise and analyse the instantaneous flow field snapshots generated with **`nsCouette`**. The following basic steps will give you a good starting point to learn

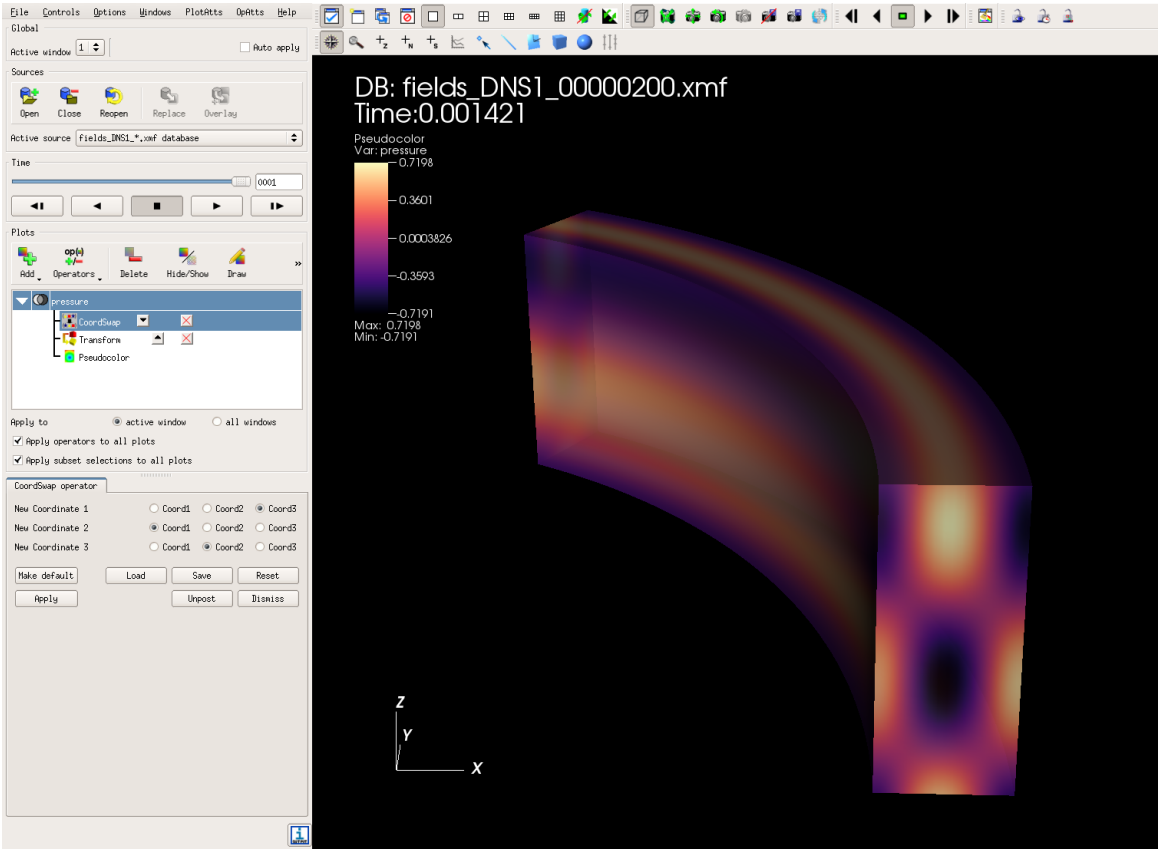


Figure 6: A screenshot of the **`GUI`** of **`VisIt`** showing a basic example of a pseudo-colour representation of the instantaneous pressure field in a small computational domain similar to the set-up used in the first tutorial in section 7.2.

how to read the **`HDF5`** files and how to render the containing flow field data.

- Download and install the software from the official webpage.
- Move all the **`.h5`** and **`.xmf`** files to one directory (**`DIR_HDF5`**) of your choice.
- Go there and start **`VisIt`**. A GUI window will pop up to select/open files.
- Enter the path to **`DIR_HDF5`**.
- Select the **`.xmf`** files and press the **`OK`** button.
- Two GUI windows will appear: The **`VisIt`** main window (control panel) and the Vis window (white blank).

- Press the **Add** button in the middle of the main window. From the pull-down menu select the type of you want to plot, *e.g.*, "contour" or "pseudocolor";
- **Important step:** Add two operators one after another on the plot by clicking on the "Operators" next to the "Add" button. Select in the pulldown menu "CoordinateSwap" to change our file coordinates (θ, z, r) to (r, θ, z) and "Transform" to change our coordinate system from cylindrical to cartesian (cf. Fig. **??**);
- Click on "OpAtts" on the top toolbars in the Main window and change the operator attributes according to the above specifications.
- Click on "Draw" and you have the plot in the right Vis window (for large file, it takes time!);
- Play with the plot and carefully observe the change after each operation;

6. About the code

This section provides further documentation of **nsCouette**; partly on a programming level and also from a software engineering point of view. The functionalities are summarised and the code structure, data types, variables, parallelisation strategy etc. are outlined. This part of the user guide is meant to be complementary to the information given our technical papers []. Additionally, some details on selected methodological aspects are outlined. Further technical documentation of the source code – auto-generated with the **FORD** tool – can be found here.

6.1. Structure of the source code

The source code comprises only a few thousand lines of code, which are structured in a minimum set of roughly a dozen basic source files – including a **Makefile**. The different parts of the code are more or less thematically structured into separate files and **Fortran** modules according to the following list. Interested developers can rely on a comprehensive, auto-generated, interactively browsable online **FORD** documentation, which includes dependency and call graphs.

```
feldmann@darkstar:~/nsCouette/nsCouette$ ls
...
Makefile           # Script file to compile the code
nsCouette.f90      # Main routine: Initialisation, time-stepping, finalisation
mod_params.f90     # All parameters and constants
mod_vars.f90       # Definition of all (derived) variables
mod_myMpi.f90      # MPI-related subroutines: windows, derived data types, etc.
mod_fftw.f90       # Subroutines related to fast Fourier transforms
mod_fdInit.f90     # Finite differences matrices for 1st and 2nd radial derivatives
mod_nonlinear.f90  # Subroutine to compute the nonlinear advection term (pseudo-spectral)
mod_timeStep.f90   # The predictor-corrector time-stepper
mod_inOut.f90      # Messy collection of input, output, base flow, initial conditions etc.
mod_hdf5io.f90     # HDF5 output of primitive flow field variables
mod_getcpu.f90     # Mapping of the CPU. This module cannot be included.
...
```

6.2. Programme flow chart

Basically speaking, the programme flow of **nsCouette** is as follows.

1. Initialisation phase: **MPI**, read parameter file (section 4.1), set initial conditions (section 4.1.4)
2. Time-stepping: linear sub-step computation and nonlinear pseudo-computation)
3. Finalising phase: **MPI**, checkpoint output (section **??**)

6.3. Constants, variables & data types

All (numerical) constants used in **nsCouette** are specified in **mod_params.f90**, while all variables are given in **mod_vars.f90**. The data types used for the numerical constants and variables are mostly specified with **kind=4** or **kind=8**, indicating single or double precision, respectively. The derived data types in **mod_vars.f90** are called

- **phys** for variables in physical space
- **spec** for variables in Fourier (spectral) space
- **vec_mpi** for vector variables in each **MPI** sub-domain

The types **phys** and **spec** contain all physically relevant quantities, such as the velocity vector ($\mathbf{u}$), the pressure (p), and, optionally, the temperature (T) field. Arrays of type **vec_mpi** are mainly used in the node-independent calculation and the data is arranged such that **mp_f** is a map of the couple (**m_th**,**m_z**/**np**). The distribution of the whole array (**m_th**,**m_z**) into each **MPI** task is according to figure 2 in our CAF paper [10]. See also section 6.8 for more details on the parallelisation strategy.

6.4. Spatial discretisation

Due to the cylindrical nature of the problem, the governing equations (see section 2.1) are handled in a polar co-ordinate system (r, θ, z), as sketched in figure 1. The spatial discretisation of the flow variables is obtained using finite differences (FD) in the wall-normal or radial direction r and using a Fourier–Galerkin ansatz in the two homogeneous directions θ and z. The FD discretisation in r is discussed in more detail in section 4.1.2. The Fourier–Galerkin expansion in the azimuthal and axial direction for the velocity vector ($\mathbf{u}$), the pressure (p) and, optionally, the temperature (T) is given by

$$\mathbf{u}(r,\theta,z) = \sum_{l=-L}^{L} \sum_{n=-N}^{N} \hat{\mathbf{u}}(r,n,l) \cdot e^{i(nk_\theta\theta + lk_z z)}, \tag{7}$$

$$p(r,\theta,z) = \sum_{l=-L}^{L} \sum_{n=-N}^{N} \hat{p}(r,n,l) \cdot e^{i(nk_\theta\theta + lk_z z)}, \tag{8}$$

$$T(r,\theta,z) = \sum_{l=-L}^{L} \sum_{n=-N}^{N} \hat{T}(r,n,l) \cdot e^{i(nk_\theta\theta + lk_z z)}. \tag{9}$$

Here, k_z is the minimum (fundamental) axial wavenumber and therefore fixes the axial length $\Gamma = 2\pi/k_z$ of the computational domain. This is basically the non-dimensional height of the TC cylinder in terms of the gap-width d, as shown in figure 1 and explained in section 2.2. In case of **nsPipe** (see appendix C for details), this corresponds to length of the computational pipe domain. Similarly, $L_\theta = 2\pi/k_\theta$ is the azimuthal arc degree. The natural periodic boundary condition in θ – i. e. a full cylindrical domain – is obtained by setting $k_\theta = 1$. Setting $k_\theta = 4$, for instance, corresponds to one quarter of the annulus, c.f. figure 10. This imposes a fourfold azimuthal symmetry and generates a solution in a symmetry subspace.

The hat symbol (e. g. $\hat{\mathbf{u}}$) in eq. (9) denotes quantities in Fourier space (data type see section 6.3) and the tuple (N, L) therefore determines the spectral numerical resolution in θ and z. The fundamental wavenumbers ($k_\theta \equiv$ **k_th0**) and ($k_z \equiv$ **k_z0**), as well as the number of Fourier modes ($N \equiv$ **m_th**) and ($L \equiv$ **m_z0**), can be specified by the user in the **nsCouette.in** input file, as described in section 4.1.

The coefficients $\hat{\mathbf{u}}(r, l, n)$, $\hat{\mathbf{p}}(r, l, n)$ and $\hat{\mathbf{T}}(r, l, n)$ are complex numbers known as spectral coefficients. They contain the entire instantaneous flow field information and are written to the binary **coeff_** files, as described in section 4.5.2.

Because the physical flow field $(\mathbf{u}, p, T)$ is real valued, the θ-coefficients – coming from a real-to-complex Fourier transform – are complex valued but complex conjugate symmetric with respect to the zero mode. The z-coefficients – coming from a complex-to-complex Fourier transform – do in general not exhibit any symmetry. in the beginning of the azimuthal symmetry of the TC setup, spectral coefficients corresponding to negative and positive azimuthal modes are complex conjugated. Therefore, only half of the θ modes need to be stored and solved for, whereas all z modes need to be considered. To reduce the size of the arrays and the number of computations, the equations are solved only for the positive azimuthal modes and the symmetry property is enforced through the FFT in the computations of the nonlinear terms. The total number of spectral coefficients used in a simulation with **nsCouette** is hence given by $m_f = (m_{th} + 1) \times 2m_{z0}$.

6.5. Accuracy of the radial derivatives

In **nsCouette** the radial derivatives are approximated by high order finite differences. The accuracy of this approximation is determined by the stencil length, i. e. the number of consecutive nodes that are used in the approximation. The stencil length in **nsCouette** corresponds to the parameter **n_s**, which can be found in the module **mod_params.f90**. Hence, by setting the value of **n_s** at compile time, the user can modify the order of accuracy of the finite difference scheme. The only restriction is that **n_s** $\geqslant 5$. The default implementation of **nsCouette** is **n_s** $= 9$.

Note that to keep the matrices banded the stencil length decreases towards the boundaries (e. g. one sided finite difference approximations are used at the cylinders). However, for radial grid distributions with $\alpha \leqslant 0.5$, the global order is not affected by this reduction due to the clustering of nodes near the cylinders (see figure 3 in [10]).

As a final remark, we would like to stress that increasing the order of the finite difference scheme does not impact on the code's performance. This is because the computational cost of evaluating the radial derivatives (basically matrix-vector multiplications) is practically negligible as compared with the cost of the fast Fourier transforms performed during the

computation of the nonlinear terms.

6.6. Temporal discretisation

Since the publication of our CAF paper [10], quite some changes have been implemented to **nsCouette** to further improve its usability and performance. Notable among these is the implementation of a new time-stepper. The governing equations are now integrated forward in time using a predictor-corrector algorithm, which allows to significantly increase the computational timestep size Δt in the simulation. The ideas for this time-stepper were borrowed from the **openpipeflow** project; an open source code for pipe flow simulations developed by Ashley Willis [11]. A comprehensive description can be found on Ashley's web page, while the basic steps of our implementation are outlined in the following.

- In the predictor step the equations are solved explicitly to obtain a rough approximation of the velocity field u_1^{q+1}. If the matrices containing the linear and non-linear terms are denoted as L and N respectively, the predictor step can be written as

$$\begin{aligned} \boldsymbol{\nabla} P_1^{q+1} &= \boldsymbol{\nabla} \cdot \left(L u^q + N^q \right) \\ \left(\frac{1}{\delta t} - c \boldsymbol{\nabla}^2 \right) u_1^{q+1} &= N^q + \left(\frac{1}{\delta t} - (1-c) \boldsymbol{\nabla}^2 \right) u^q - \boldsymbol{\nabla} P_1^{q+1} \end{aligned}$$

- The velocity computed in the predictor step (u_1^{q+1}) is then refined in multiple corrector steps, until a certain tolerance is reached. The iteration for the corrector step is given by

$$\begin{aligned} \boldsymbol{\nabla} P_{j+1}^{q+1} &= \boldsymbol{\nabla} \cdot \left(L u^q + N_j^{q+1} \right) \\ \left(\frac{1}{\delta t} - c \boldsymbol{\nabla}^2 \right) u_{j+1}^{q+1} &= c N_j^{q+1} + (1-c)\, N^q + \left(\frac{1}{\delta t} - (1-c)\, \boldsymbol{\nabla}^2 \right) u^q - \boldsymbol{\nabla} P_{j+1}^{q+1} \end{aligned}$$

  and the tolerance is set to **tolerance_dterr = 5E-5** by default. If necessary, it can easily by changed by modifying the file **mod_param.f90**, see section **??**

- The coefficient c defines the implicitness of the temporal integration scheme. It is set to **d_implicit = 0.5** by default and can easily be changed at compile time by modifying the file **mod_params.f90**. Note, that the temporal scheme is of second-order accuracy if $c = 0.5$.

- The boundary conditions are imposed through influence matrices which are computed at a pre-processing stage.

- The user can choose between fix or variable time-step size. In the latter case Δt is computed as

$$\Delta t = C \min \left(\boldsymbol{\nabla} / |v| \right), \tag{10}$$

  where C is the Courant or CFL number.

6.7. Feature for thermal convection

You can easily switch to the optional temperature version of **nsCouette** by simply setting the external variable **CODE=TE_CODE** when building the code. Note, that the standard version of **nsCouette** – i. e. without the additional temperature equation – is set by default (**CODE=STD_CODE**), see also section 3.4.2.

```
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform clean
feldmann@darkstar:~/nsCouette/nsCouette$ make ARCH=myPlatform HDF5IO=no CODE=TE_CODE
```

The equation for the temperature (T) is coupled to the Navier–Stokes equations by considering a Boussinesq-like approximation that includes centrifugal buoyancy effects. Details about the dimensionless governing equations and the Boussinesq-like approximation are documented in Lopez et al. [7]. Now, the code produces additional output. For details also see section 4.5. The file **Nusselt** contains time series of the normalised heat transfer at the inner and outer cylinders walls and the file **Temp_energy** contains time series of the modal energy related to the temperature field. Time series of the temperature at the some user specified probe locations do now appear in the files **probes*.dat**. The snapshot flow field files (**coeff_*** and **fields*.h5**) now also contain full information about the instantaneous temperature field.

In the provided version of **nsCouette**, the flow is stratified radially by heating the inner cylinder and cooling the outer cylinder, i. e. a negative radial temperature gradient $\partial T/\partial r$. A descriptive example is given in the tutorial 7.5.

6.7.1. How to modify the source code

The sense of the temperature gradient can be easily modified by inverting the boundary conditions for the temperature in the file **mod_timeStep.f90** and modifying the temperature and axial velocity of the base flow accordingly in the **subroutine base_flow** contained in the file **mod_InOut.f90**.

```
! Boundary conditions for mode 0 (prescribed temperature at the cylinders)
if (abs(fk_mp%th(1,k)) <= epsilon .and. abs(fk_mp%z(1,k)) <= epsilon) then
 rhs_p(1)   = dcmplx( 0.5d0, 0d0)
 rhs_p(m_r) = dcmplx(-0.5d0, 0d0)
end if
```

6.8. Parallelisation scheme

Data distribution across **MPI** tasks is accomplished using a one-dimensional (1d) slab decomposition of the computational domain. This technique follows naturally and straightforwardly from the Fourier–Galerkin ansatz we use for spatial discretisation, see section 6.4: when the variables are expanded in Fourier series, equations 1 are separated into a system of m_f independent linear equations, i. e. one equation for each spectral coefficient. These equations are evenly distributed across the number of **MPI** tasks, denoted in the code as **np**. Each discretised variable in **nsCouette** is stored as a two-dimensional (2d) array, (m_r, m_f), with the second dimension identifying the spectral coefficients and the first dimension its radial dependence. When the 1d slab decomposition is applied, the data is

partitioned along the second dimension of the array, $mp\_f = m\_f/np$, so that each **MPI** task contains a data chunk of size (m_r, mp_f).

Since the computation of the nonlinear terms is carried out using a pseudospectral technique, spectral data must be transformed into data in physical space. These transformations are performed using fast Fourier transforms (FFT), which require each **MPI** task to contain all spectral coefficients. To satisfy this requirement, while keeping a 1d slab domain decomposition, the data, during the calculation of the nonlinear terms, is divided along the radial direction and organised in 2d arrays of size $(m\_f, mp\_r = m\_r/np)$. Such modification of the storage arrangement is efficiently performed by using global data transpositions based on **MPI_Alltoall** and task-local transposes.

Note that both the number of radial modes (m_r) and Fourier modes (m_f) must be divisible by the number of **MPI** tasks (np), which imposes a restriction in the selection of the numerical resolution. **nsCouette** implements a simple technique that relaxes partially such restriction by allowing a free selection of the number of Fourier modes. If m_f is not divisible by np, m_f is automatically increased to satisfy the divisibility condition and the additional spectral coefficients are set to zero. Hence, the only restriction for the selection of modes in **nsCouette** is that m_r must be divisible by np.

Since in turbulent simulations the number of radial modes is typically much smaller than the total number of Fourier modes ($m\_r << m\_f$), the number of **MPI** tasks that can be used in our 1d slab decomposition is limited by the value of m_r. Consequently, if only this paralellisation strategy were applied, the achievable parallel speedup with respect to a serial code version would be at most m_r. In order to increase the parallel scalability beyond this limit, an additional **OpenMP** parallelisation layer has been introduced. In the parts of the code where the solution of the linear equations is computed, **OpenMP** threads allow to parallelise over the mp_f spectral coefficients contained within each **MPI** task. That is, if **n_threads** is the number of **OpenMP** threads within each **MPI** task, each thread will solve $mp\_f/n\_threads$ linear equations to obtain the spectral coefficients. In the part where the non-linear terms are computed, some coarse-grain parallelism (e. g. the scalar state variables and the components of the velocity vector can be Fourier-transformed independently of each other and some of these computations can be overlapped with the global transpositions) can be exploited with the **OpenMP** threads.

A sketch illustrating the data distribution for this hybrid strategy is shown in figure 7. The variables here are discretised using $m\_r = 8$ radial modes and $m\_f = 64$ spectral coefficients, and the simulation is run on a machine with 64 cores. Data chunks resulting from the **MPI** slab decomposition are indicated by the letter P followed by a number. Note that the number of **MPI** tasks has been set to eight (P0...P7), which is the maximum possible value in this example ($np = m\_r$). There are also four **OpenMP** threads per **MPI** task which are indicated by the letter T followed by a number (T0...T3). The subgroups of data that are handled by each thread are shown in different colours. With the choice of $np = 8$, $mp\_f = 64/8 = 8$ linear equations are solved within each **MPI** task (**mod_timestep.f90**), and only one radial point, $mp\_r = 8/8 = 1$, is contained within each **MPI** task in the transposed layout (**mod_nonlinear.f90**). By using four **OpenMP** threads per **MPI** task in this example, an additional factor of four in concurrency and hence a total theoretical parallel speedup of 32 is gained for the linear terms. In the transposed layout, by contrast, since a single radial point is computed by each **MPI** task, the theoretical speedup would be limited by eight. Nevertheless, as indicated above, additional concurrency for the four

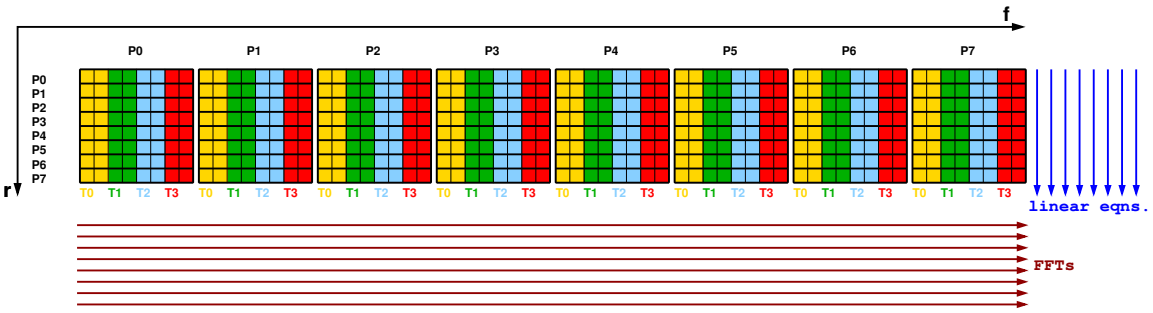


Figure 7: Sketch of the one-dimensional domain decomposition of the two-dimensional numerical grid of eight radial (r) points times 64 Fourier (f) modes. Like in the code a linearised index is used to label the Fourier modes in the z and θ directions.

OpenMP threads per **MPI** task is exposed by:

1. Computing the Fourier transforms and derivative computations for each scalar variable and the individual components of vectors, respectively, independently in parallel.

2. Handling the global **MPI** transposition for some of the variables while others are still being computed.

Finally, we would like to note that an alternative strategy to the use of **OpenMP** is to perform a 2d **MPI** domain decomposition. This strategy has been used in a number of similar codes for the simulation of fluid flows (e. g. Ashley Willis' **openpipeflow** [11], John Gibson's **channelflow.org**, Simpson, et cetera). We conducted a comparative study of the performance and scalability of two pipe flow codes with similar algorithmic formulation but one implementing a 2d **MPI** domain decomposition [11] and the other a hybrid **MPI**-**OpenMP** strategy (**nsPipe**). We observed that both strategies may be beneficial depending on the type of problem under consideration. For example, for big setups (i. e. large m_r and large m_f) typical for high Reynolds number turbulence, our hybrid strategy exhibited a much better performance and scalability. However, in problems where extreme long pipes are considered (i. e. large m_f) at rather low Reynolds (i. e. small m_r), a 2d **MPI** decomposition in r and z turned out to be more efficient than our hybrid strategy. Overall, when confronted with a 2d **MPI**-only domain decomposition, we believe that our hybrid parallelisation scheme is a good compromise between simplicity of the implementation and achievable parallel scalability, which fits perfectly with the technological trend of multicore processors with ever increasing core counts and stagnating per-core performances.

7. Tutorials

7.1. Prerequisites

- Get a recent **gnuplot** installation to quickly inspect the time series output interactively and to easily produce some decent line plots using the ***.gpl** script files, which come with this tutorials. The scripts provided here were generated and tested using version **5.0**.

```
feldmann@darkstar:~/nsCouette$ gnuplot --version
gnuplot 5.0 patchlevel 3
```

- For more advanced plotting and data analysis a recent **Python** installation can be quite helpful. Some of this tutorials come with a few basic ***.py** script files which can be used as a starting point for your further work.
- For three-dimensional flow field visualisation we recommend you to install **ParaView** (Section 5.1) or **VisIt** (Section 5.2).
- A proper LaTeX installation might also be useful e. g. to compile this manual or to convert the ***.eps** output from **gnuplot** to proper ***.pdf** figures.

7.2. Laminar Taylor-Couette flow in the stable regime

To start off, we will first run a simulation of Taylor-Couette flow in the stable regime, i. e. at a very low Reynolds number. This allows us to perform our first DNS on a regular laptop in less than a minute. Moreover, we can readily compare the results to what we expect from theory. Go to your working directory, copy the first Taylor-Couette tutorial case from the repository and inspect the parameter input file (**nsCouette.in**) using your favourite text editor (e. g. **vi**, **emacs**, **kate**, **gedit** and such).

```
feldmann@darkstar:~$ cd ~/nsCouette
feldmann@darkstar:~/nsCouette$ cp -r ../nscouette/tutorials/tc0040 .
feldmann@darkstar:~/nsCouette$ cd tc0040
feldmann@darkstar:~/nsCouette/tc0040$ vi nsCouette.in
```

We choose to keep the outer cylinder wall stationary by setting $\mathrm{Re}_o = 0$ and further choose a very low rotation speed of the inner cylinder by setting $\mathrm{Re}_i = 50$. The corresponding part in the input file **nsCouette.in** looks like this.

```
&parameters_physics
Re_i = 50.0d0    ! inner cylinder Reynolds number
Re_o =  0.0d0    ! outer cylinder Reynolds number
/
```

All the other parameters in this namelist (**parameter_physics**) are irrelevant for our first tutorials. They will be discussed later in section 7.5, when we turn to thermal convection. See also section 4.1 and A for an exhaustive description of the parameter input file. Due to this very small Reynolds number, a very coarse spatial resolution should be sufficient to produce accurate results. Here, we choose $M = 16$ radial grid points in physical space and $N = L = 4$ Fourier modes in azimuthal (θ) and axial (z) direction, respectively. The corresponding entries in the namelist **paramters_grid** are shown here.

```
&parameters_grid
m_r  = 16            ! M radial grid points
m_th =  4            ! N azimuthal Fourier modes
m_z0 =  4            ! L axial Fourier modes
/
```

This choice corresponds to a total of $n_r \times n_\theta \times n_z = M \times (2N+1) \times (2L+1) = 16 \times 9 \times 9$ significant physical grid points, while the nonlinear terms are actually evaluated on

$M \times (3N + 1) \times (3L + 1) = 16 \times 13 \times 13$ physical grid points for dealiasing purposes. For further details, definitions and notations regarding the spatial discretisation scheme of **nsCouette**, you might want to have a look at our CAF paper [10] around page 3.

We want the simulation to run for 8000 timesteps. And the size of the computational timestep Δt should be dynamically adapted to automatically guarantee numerical stability during time integration. Both can be controlled using the relevant keywords in the namelist **parameters_timestep**.

```
&parameters_timestep
numsteps    = 8000       ! Number of computational timesteps
variable_dt = T          ! Use variable (T) or fixed (F) timestep size
/
```

Integrating the Navier-Stokes equations forward in time is a typical initial value problem. Therefore, one remaining issue to discuss – before we finally run our first DNS – is choosing proper initial conditions (Section 4.1.4) to start from. Since we do not have any suitable flow field data at hand, we choose the most simple – and compared to a real world laboratory set-up also the most similar – condition to start with. As can be seen in the parameter input file, we choose to start our simulation from scratch and prescribe a zero-velocity field ($u_r = u_\theta = u_z = 0$) as initial flow state ($t = 0$), just like the resting fluid in the experimental set-up just before you switch on the cylinder rotation.

```
&parameters_control
restart = 0           ! Start from scratch (0) or restart from checkpoint (1,2)
/

&parameters_initialcondition
ic_tcbf = F ! Set Taylor-Couette base flow (T) or resting fluid (F), only when restart = 0
ic_pert = T ! Add perturbation on top of base flow (T) or not (F), only when restart = 0
ic_p(1, :) = 4.0d-2, 0, 1 ! 1st perturbation: amplitude and wavevector (a1, k_th1, k_z1)
ic_p(2, :) = 6.0d-3, 1, 0 ! 2nd perturbation: amplitude and wavevector (a2, k_th2, k_z2)
/
```

Additionally, the initial zero-velocity flow field is superimposed by a finite amplitude perturbation to test whether the code actually captures the well-known stable behaviour of the Taylor-Couette system for such a small Reynolds number. When thinking of a real world laboratory set-up, perturbations could model residual motions from filling the tank, mechanical vibrations from the drive, small density or temperature gradients and such. Up to $l = 6$ independent perturbations can be prescribed by specifying the tuple $(a_l, k_{\theta,l}, k_{z,l})$, representing the respective perturbation amplitude and wave vector.

If you have already compiled the code (Section 3), you now simply have to copy the executable to your case directory and start the simulation by prompting the executable and passing the parameter input file to the standard input.

```
feldmann@darkstar:~/nsCouette/tc0040$ cp ../nscouette/darkstar/nsCouette.x .
feldmann@darkstar:~/nsCouette/tc0040$ ./nsCouette.x < nsCouette.in
```

Further details and different ways to start a simulation can be found in section 4. Now **nsCouette** runs on one single **MPI** task and throws its log output directly to the terminal. You can watch the progress of the simulation step by step, which should usually finish in less than a minute (Here after 4.15 s). The few first and last lines of the log should look something like this.

```
starting NSCouette (a88e2a11) using 1 MPI tasks, 8 OpenMP threads/task
TASKS: 1 THREADS: 8 THIS: 0 Host:darkstar Cores: 0 0 0 0 0 0 0 0
  step=   100  dt=   1.1100000000000001E-004
  step=   200  dt=   1.2321000000000003E-004
  step=   300  dt=   1.3676310000000004E-004
  ...
  step=  7700  dt=   1.8704145521610010E-004
  step=  7800  dt=   1.8704145521610010E-004
  step=  7900  dt=   1.8704145521610010E-004
 written hdf5/xdmf files to disk: fields_tc0040_00008000.{h5,xmf}
  step=  8000  dt=   1.8704145521610010E-004
 written coeff file to disk: coeff_tc0040.00008000
 written coeff file to disk: coeff_tc0040.00008000
--------------------------------------------------------------------------------
Total number of computed timesteps: 00008000
Total elapsed WCT since start up:  4.15s
Average elapsed WCT per timestep w/o coeff io (min, mean, max): 0.0005s, 0.0005s, 0.0005s
feldmann@darkstar:~/nsCouette/tc0040$
```

The log tells us, that at some particular timesteps, **nsCouette** wrote instantaneous state files (**coeff*** and **fields***) to disk, as described in section 4.5. Listing the content of our case directory by typing

```
feldmann@darkstar:~/nsCouette/tc0040$ ls
coeff_tc0040.00002000       coeff_tc0040.00008000          ke_total     probe05.dat
coeff_tc0040.00002000.info  coeff_tc0040.00008000.info  ke_z         probe06.dat
coeff_tc0040.00004000       fields_tc0040_00008000.h5   probe01.dat  torque
coeff_tc0040.00004000.info  fields_tc0040_00008000.xmf  probe02.dat
coeff_tc0040.00006000       ke_mode                                  probe03.dat
coeff_tc0040.00006000.info  ke_th                                    probe04.dat
```

reveals, that next to the four Fourier coefficient files (Section 4.5.2) and the one physical flow field file (Section 4.5.3) there are also some time series data files. They contain temporal information about the kinetic energy (**ke_***), the velocity at six individual probe locations (**probe0*.dat**), and the torque Nusselt number (**torque**). Let's have a look at the torque data first, which is an integral system quantity. By typing something like

```
feldmann@darkstar:~/nsCouette/tc0040$ head -n 5 torque
1.1100000000000E-003     1.4343271052595E+001     1.4059654954111E-009   ...
2.2200000000000E-003     1.0042605705170E+001    -4.1595016094334E-009   ...
3.3300000000000E-003     8.1720439469609E+000     2.6556245818254E-008   ...
4.4400000000000E-003     7.0899641989767E+000    -3.7680014324448E-009   ...
5.5500000000000E-003     6.3560048662009E+000    -6.2320501650089E-008   ...
feldmann@darkstar:~/nsCouette/tc0040$ tail -n 5 torque
1.4642139302435E+000     1.0000008374531E+000     9.9999896083642E-001   ...
1.4660843447956E+000     1.0000008220272E+000     9.9999897984950E-001   ...
1.4679547593478E+000     1.0000008069813E+000     9.9999899851429E-001   ...
1.4698251738999E+000     1.0000007921586E+000     9.9999901683719E-001   ...
1.4716955884521E+000     1.0000007776289E+000     9.9999903482444E-001   ...
```

the first/last five lines of this simple time series text file are dumped to the terminal. The first column represents the advancing physical time ranging from the initial state ($t = 0$) we have specified, up to the very end ($t \approx 1.5$) of this simulation run. The next two columns represent the instantaneous non-dimensional torque $Nu_{\omega,i}$ and $Nu_{\omega,o}$ integrated over the inner and outer cylinder wall, respectively. Data columns four and five (not shown here), represent the friction Reynolds number (Re_τ) at the inner and outer cylinder wall. We can already see here, that $Nu_{\omega,o} = 0$ in the initially resting flow field, since the outer wall as well as the adjacent fluid are at rest. Since the inner cylinder wall starts to move at $t = 0$ at a constant speed while the adjacent fluid have been initialised at rest, the torque

$Nu_{\omega,i}$ exerted to the fluid by the inner cylinder wall, abruptly takes rather high values. At the end of the simulation, both torque values have reached a value of one. This makes total sense, since the flow state at this low Reynolds number is dominated by viscosity and should therefore recover the analytical solution for laminar Taylor-Couette flow. The torque Nusselt number is defined as the actual torque normalised by the torque of the laminar flow state (See e. g. [3] around page 426 for further details). Therefore, our torque results compare favourably with theoretical predictions.

More can be seen by making a simple time series plot using e. g. **`gnuplot`**. In this tutorial case directory you can find a ready-to-use script to generate the plot shown in figure 8 by simply typing

```
feldmann@darkstar:~/nsCouette/tc0040$ gnuplot torque.gpl
feldmann@darkstar:~/nsCouette/tc0040$ okular torque.pdf &
feldmann@darkstar:~/nsCouette/tc0040$ ls torque*
torque  torque.eps  torque.gpl  torque.pdf
```

and opening the generated **`*.pdf`** or **`*.eps`** figure using your favourite document viewer (e. g. **`okular`**, **`evince`**, **`acroread`** and alike). We see that the inner and outer torque val-

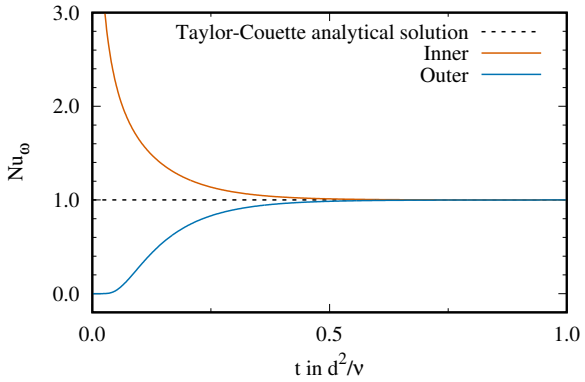


Figure 8: Temporal evolution of the torque at the inner and outer cylinder wall in our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. The torque is expressed in a non-dimensional way using a type of a Nusselt number, which relates the actual torque to the torque of the laminar Taylor-Couette analytical solution.

ues both monotonically converge to the theoretically predicted value of one after roughly $\mathcal{O}(1)$ viscous time units. This again makes total sense, since this is approximately the time span it should take for the presence of the driving inner wall to propagated to the opposing wall (at a distance $d = 1$) solely by the action of viscosity.

Next, we take a look at the time series output of the velocity vector at some particular probe locations in the flow field, which will give us more local insights into the flow, where the torque provided us with rather global informations. By simply typing

```
feldmann@darkstar:~/nsCouette/tc0040$ gnuplot probes.gpl
feldmann@darkstar:~/nsCouette/tc0040$ okular probes.pdf &
```

we generate a plot as shown in figure 9. We can easily see that the streamwise (azimuthal) velocity component (u_θ) converges nicely to the theoretically predicted values everywhere in the flow domain ($r_i \leqslant r \leqslant r_o$). As expected, both cross-stream velocity components ($|u_r|$ and $|u_z|$) decay to zero. The initial finite amplitude perturbations which we imposed do

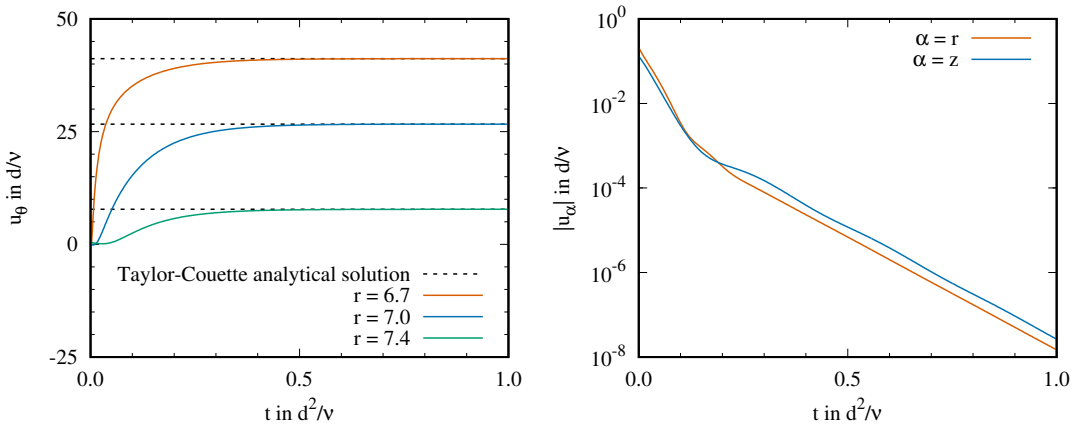


Figure 9: Temporal evolution of the velocity components at individual probe locations in our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. The streamwise velocity component (u_θ) is shown at three different radial locations ($r_i \leqslant r \leqslant r_o$) and converges everywhere to the theoretically predicted values (left). The absolute value of the two cross-stream components ($|u_r|$ and $|u_z|$) is shown exemplarily at one radial location ($r = 7.0$).

not survive in the linearly sable regime and the analytical solution for laminar Taylor-Couette flow is nicely reproduced/recovered. Information about the analytical Taylor-Couette solution can be found in the **`probes.gpl`** file by typing e. g.

```
sed -n '18, 33 p' probes.gpl

# Taylor-Couette set-up
Re_i = 50.0 # inner cylinder Reynolds number
Re_o =  0.0 # outer cylinder Reynolds number
eta = 0.868 # raddii ratio

# Analyitcal Taylor-Couette solution
nutc = 1.0 # Taylor-Couette analytical torque (Nusselt number)
c1 = (Re_o - eta * Re_i) / (1 + eta)
c2 = (eta * (Re_i - eta*Re_o)) / ((1 - eta) * (1 - eta**2.0))
r1 = 6.741192272578147 # radial probe location from file header
r2 = 7.023493344123748 # radial probe location from file header
r3 = 7.410322878937004 # radial probe location from file header
utc1 = c1*r1 + c2/r1  # Taylor-Couette analytical velocity
utc2 = c1*r2 + c2/r2  # Taylor-Couette analytical velocity
utc3 = c1*r3 + c2/r3  # Taylor-Couette analytical velocity
```

or e. g. in [10] on page 5 and in [3] around page 424. The particular radial probe locations for which the analytical solution is needed to produce figure 9, can be found in the header of the **`probe0*.dat`** files by typing

```
feldmann@darkstar:~/nsCouette/tc0040$ head -n 10 probe01.dat
# Time series data from probe 01 on rank 00000
# Radial location n_r = 00005 n_rp = 0005 r =  6.741192272578147E+00
# Azimuthal location n_th = 00003 th =  2.617993877991494E-01
# Axial location n_z = 00003 z =  6.000000000000010E-01
# Time, u_r, u_th, u_z
9.9900000000000E-004  4.9755347543047E-002 -5.0697872196875E-001  -3.9262289916098E-001
2.1090000000000E-003  4.9251475018859E-002  2.0445331579843E-003  -3.7513595580142E-001
3.2190000000000E-003  4.8415330734597E-002  1.4934842744334E+000  -3.5808738661681E-001
4.3290000000000E-003  4.7373835221887E-002  3.3163726265500E+000  -3.4181516483879E-001
5.4390000000000E-003  4.6167965547437E-002  5.1781059897116E+000  -3.2637614051804E-001
```

In case you are interested in time series data at different probe locations, you can modify the coordinates and the sampling rate in the parameter file using your favourite text editor

```
feldmann@darkstar:~/nsCouette/tc0040$ vi nsCouette.in
&parameters_output
dn_prbs   = 10  ! output interval [steps] for time series data at probe locations
prl_r(1)  = 0.20d0 ! radial probe locations (0 < r/d < 1)
prl_r(2)  = 0.50d0
prl_r(3)  = 0.80d0
prl_r(4)  = 0.20d0
prl_r(5)  = 0.50d0
prl_r(6)  = 0.80d0
prl_th(1) = 0.25d0 ! azimuthal probe locations (0 < th/L_th < 1)
prl_th(2) = 0.25d0
prl_th(3) = 0.25d0
prl_th(4) = 0.75d0
prl_th(5) = 0.75d0
prl_th(6) = 0.75d0
prl_z(1)  = 0.25d0 ! axial probe locations (0 < z/L_z < 1)
prl_z(2)  = 0.25d0
prl_z(3)  = 0.25d0
prl_z(4)  = 0.75d0
prl_z(5)  = 0.75d0
prl_z(6)  = 0.75d0
/
```

and restart (Section 4.6) or rerun the simulation.

Since we have only looked at particular points in the flow field so far, figure 10 shows a three-dimensional view on the entire computational domain ($(r_i \leqslant r \leqslant r_o) \times (0 \leqslant \theta \leqslant L_\theta) \times (0 \leqslant z \leqslant L_z)$) we have chosen for our first test case. To keep the simulation simple and quick, we restrict ourselves to only a segment of a full real-world Taylor-Couette cylinder set-up. Here we chose one sixth of the full azimuth ($L_\theta = {}^{2\pi}/_{k_{\theta,0}} = {}^{2\pi}/_{6}$) and only a very small height of ($L_z = {}^{2\pi}/_{k_{z,0}} = {}^{2\pi}/_{2.618} = 2.4$), as can be specified by setting

```
&parameters_grid
k_th0 = 6.0d0                ! azimuthal fundamental wavenumber
k_z0  = 2.61799387799149d0   ! axial fundamental wavenumber
eta   = 0.868d0              ! inner to outer radii aspect ratio
\
```

in the namelist **parameters_grid** in the parameter input file. The colour-coding in figure 10 represents the only non-zero velocity component

$$\mathbf{u} = \begin{bmatrix} u_r \\ u_\theta \\ u_z \end{bmatrix} = \begin{bmatrix} 0 \\ f(r) \\ 0 \end{bmatrix}, \tag{11}$$

which is obviously invariant with respect to the azimuth θ and the axial location z. It only varies with respect to the wall-normal location r and decreases monotonically from the inner cylinder wall at $r = r_i = 6.576$, which is driven at a speed of ${}^{50d}/_{\nu}$, down to zero at the outer cylinder wall ($r = r_o = 7.576$), which is kept stationary. This pseudo-colour representation of the flow field can be easily reproduced using **ParaView** and loading the state file **laminarTaylorCouette.pvsm** which comes with this tutorial. Further details on how to visualise flow fields can be found in section 5.1 or section 5.2, in case you prefer the software **VisIt**.

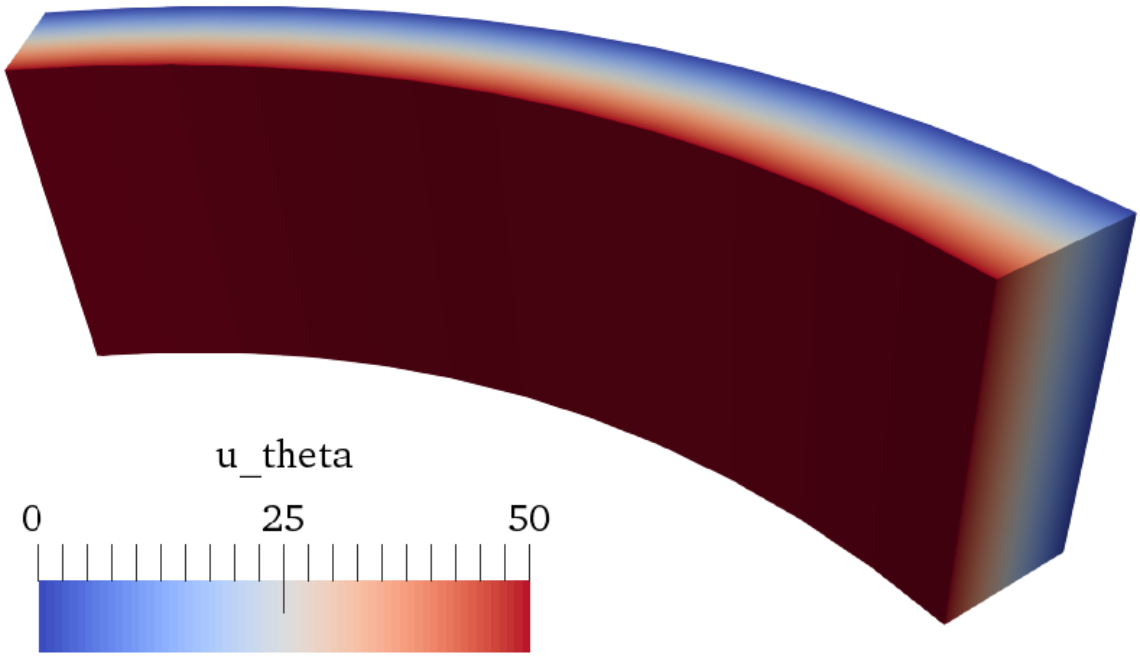


Figure 10: The depicted volume represents the entire computational domain ($(6.576 \leqslant r \leqslant 7.576) \times (0 \leqslant \theta \leqslant {}^{2\pi}/_{6}) \times (0 \leqslant z \leqslant 2.4)$) we have chosen for our first DNS; i. e. only one sixth of a full cylinder with a very small axial height. The colour-coding represents the only non-zero velocity component (u_θ), which is obviously invariant with respect to θ and z and only varies with respect to the wall-normal location r according to the analytical solution for laminar Taylor-Couette flow.

Last but not least we want to have a look at the time series data for the kinetic energy. The two files **`ke_th`** and **`ke_z`** provide integral kinetic energies for each discrete Fourier mode (n_θ and l_z, respectively) at each timestep. So in our case, both files contain six columns of time series data: the first one being the physical time t and the next five being the integral kinetic energy in mode number $0 \leqslant n_\theta \leqslant N = 4$ and $0 \leqslant l_z \leqslant L = 4$, respectively. The kinetic energy per mode is an integral quantity in the sense, that it is integrated in both cases over the radial direction (r) and also over either the axial (z) or the azimuthal (θ) direction, respectively, depending on which quantity you are looking at. By simply typing

```
feldmann@darkstar:~/nsCouette/tc0040$ gnuplot keThZ.gpl
feldmann@darkstar:~/nsCouette/tc0040$ okular keThZ.pdf &
```

we generate a plot as shown in figure 11. Since there is no relevant energy content in any of the modes larger than the zero mode (which represents the mean value in the respective direction), there is no variation of the flow field neither in θ nor in z direction. And there is obviously also no periodic or chaotic variation with respect to time at this low Reynolds number. Actually, the kinetic energy in all modes decays to practically zero. The amount of energy introduced to the initial conditions by the finite amplitude perturbations is monotonically damped out by the dominating effect of viscosity and it never increases again. You can easily double-check this for larger times by simply restarting (Section 4.6) the simulation and let it run for another 8000 or more timesteps. Although the dynamics at this

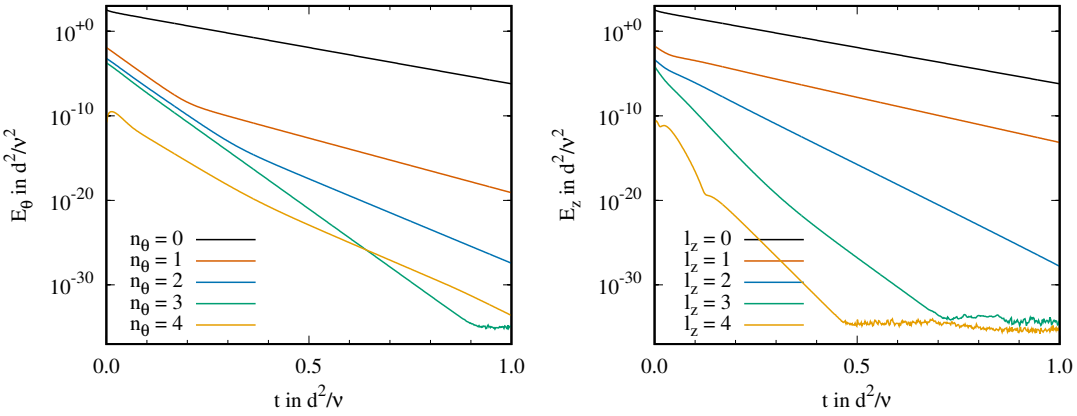


Figure 11: Temporal evolution of the azimuthally (left) and axially (right) dependent kinetic energy contained in each discrete azimuthal (n_θ) and axial (l_z) mode for our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. See also page 3 in [10] for definitions and details on the spatial discretisation scheme of **`nsCouette`**. As expected, the non-zero kinetic energy introduced by the finite amplitude perturbation in the initial velocity field decays monotonically to practically zero in all modes and never increases again.

low Reynolds number is rather boring and also trivial as well as expected, these per mode energies are a very convenient and important measure to track and analyse the dynamical behaviour of the Taylor-Couette system in general. However, have in mind, that for more interesting (i. e. increasing) Reynolds numbers – and thus finer spatial resolution – these files progressively become huge and are not so easy to handle anymore. One option could be to increase the sampling frequency for these specific output. This can be done using the respective keyword in the namelist **`parameters_output`** in the parameter input file.

```
feldmann@darkstar:~/nsCouette/tc0040$ vi nsCouette.in
&parameters_output
dn_ke = 1000    ! timestep output interval for modal kinetic energy
/
```

Additionally, the highest wave numbers represent the spatial resolution of the simulation, see section 6.4. Plots like these in figures 11, 15 and 18 become very handy as a first indicator for the quality of the spatial resolution. To resolve all relevant flow scales, the trailing coefficients of the spectral expansions should be smaller than the leading coefficients by at least four or five orders of magnitude. Here, the highest wave numbers only show numerical noise at a very low level ($\mathcal{O}\left(10^{-30}\right)$) but no physically relevant energy content. Hence, the spatial resolution in θ in z direction can be considered as sufficient.

7.3. Taylor-vortex flow

Above a certain Reynolds number – slightly depending on the exact geometry and boundary conditions – the Taylor-Couette flow becomes unstable and gives rise to Taylor vortices. In a next step, we want to increase the Reynolds number to the unstable regime and see whether **`nsCouette`** reproduces this well known Taylor-vortex state. Go back to your working directory, copy the second Taylor-Couette tutorial case from the repository and inspect the parameter file (**`nsCouette.in`**) using your favourite text editor.

```
feldmann@darkstar:~$ cd ~/nsCouette
feldmann@darkstar:~/nsCouette$ cp -r ../nscouette/tutorials/tc0041 .
feldmann@darkstar:~/nsCouette$ cd tc0041
feldmann@darkstar:~/nsCouette/tc0041$ vi nsCouette.in
```

We triple the rotation speed of the inner cylinder by changing the respective keywords in the namelist **parameters_physics**. Since we now expect a slightly more complex flow field, we also have to increase the spatial resolution to account for the spatial velocity gradients related to the pronounced vortex in the flow field.

```
&parameters_physics
Re_i = 150.0d0 ! inner cylinder Reynolds number
Re_o =   0.0d0 ! outer cylinder Reynolds number
/
&parameters_grid
m_r  = 24   ! M radial points
m_th =  4   ! N azimuthal Fourier modes
m_z0 =  8   ! L axial Fourier modes
/
```

This choice corresponds to a total of $n_r \times n_\theta \times n_z = M \times (2N+1) \times (2L+1) = 24 \times 9 \times 17$ significant physical grid points. The finer spatial resolution naturally demands a smaller computational timestep Δt. Since Δt will be adapted automatically to ensure numerical stability during integration, we simply have to increase the total number of timesteps in namelist **parameters_timestep**, to end up at roughly the same physical time t as in our first tutorial case (Section 7.2).

```
&parameters_timestep
numsteps    = 22000   ! Number of computational timesteps
variable_dt = T       ! Use variable (T) or fixed (F) timestep size
/
```

Both changes will surely increase the overall computational time necessary for our second DNS. To save resources and to keep the tutorials small, we now choose start our second simulation from a flow state similar to the one we ended up with in our first tutorial (Section 7.2). We do this by prescribing the analytical Taylor-Couette velocity profile as initial condition and again add a finite amplitude perturbation on top of it.

```
feldmann@darkstar:~/nsCouette/tc0041$ vi nsCouette.in
&parameters_initialcondition
ic_tcbf = T ! Set Taylor-Couette base flow (T) or resting fluid (F), only when restart = 0
ic_pert = T ! Add perturbation on top of base flow (T) or not (F), only when restart = 0
ic_p(1, :) = 4.0d-2, 0, 1 ! 1st perturbation: amplitude and wavevector (a1, k_th1, k_z1)
ic_p(2, :) = 6.0d-3, 1, 0 ! 2nd perturbation: amplitude and wavevector (a2, k_th2, k_z2)
/
```

To run this tutorial, you now simply have to copy the same executable as before (Section 7.2) to the new case directory and start the simulation by prompting the executable and passing the new parameter input file to the standard input. Further details and different ways to start a simulation can be found in section 4.

```
feldmann@darkstar:~/nsCouette/tc0041$ cp ../nscouette/darkstar/nsCouette.x .
feldmann@darkstar:~/nsCouette/tc0041$ ./nsCouette.x < nsCouette.in
...
  step=       19000  dt=   6.6666666666666670E-005
  step=       20000  dt=   6.6666666666666670E-005
  step=       21000  dt=   6.6666666666666670E-005
```

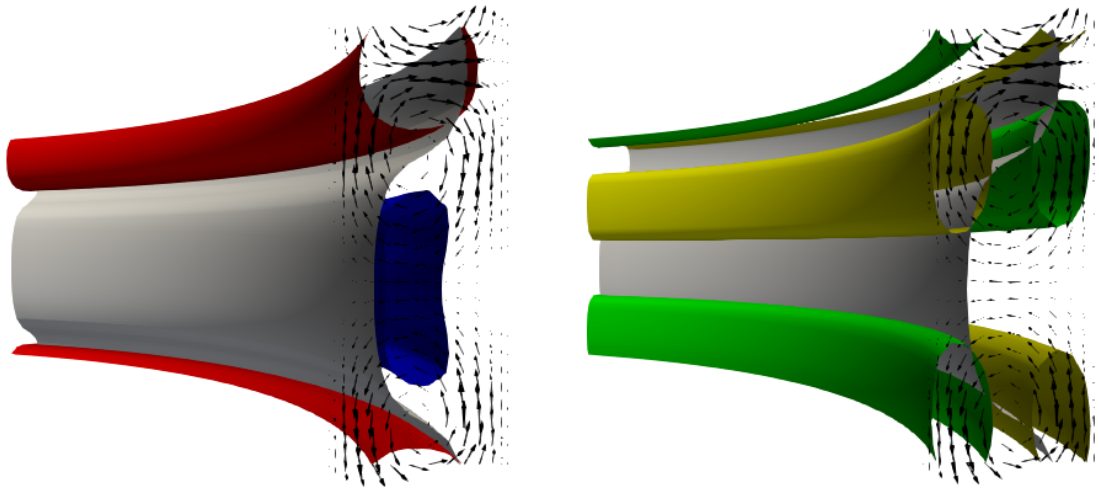


Figure 12: Three-dimensional flow field visualisation using **ParaView**. Shown are iso-contours (red/blue: $u_r = \pm 4$, grey: $u_\theta = 75$, yellow/green: $u_z = \pm 4$) and a vector representation of the cross-stream velocity components. Stationary Taylor-vortex flow state at $Re_i = 150$.

```
 written hdf5/xdmf files to disk: fields_tc0041_00022000.{h5,xmf}
  step=       22000  dt=   6.6666666666666670E-005
 written coeff file to disk: coeff_tc0041.00022000
 written coeff file to disk: coeff_tc0041.00022000
--------------------------------------------------------------------------------
Total number of timesteps: 00022000
Total elapsed WCT:    26.96s
Elapsed WCT per timestep w/o coeff io (min, max, average):   0.0012s,    0.0012s,
    0.0012s
```

Now **nsCouette** runs on one single **MPI** task and throws its log output directly to the terminal. You can watch the progress of the simulation step by step, which should usually finish in less than a minute (Here after 26.96 s).

Let's continue with first having a look at the final flow state. Figure 12 shows a three-dimensional representation of the instantaneous flow field at the last timestep from our simulation run. As already mentioned above, the higher Reynolds number now gives rise to a spatially more complex state with a secondary flow ($u_{r,z} \neq 0$) forming toroidal vortices. As shown in figure 12, alternating regions of positive (red) and negative (blue) u_r push high-speed fluid from the rotating inner cylinder towards the stationary outer cylinder; And Vice versa for low-speed fluid. This can be easily seen by the bent iso-surface (grey) representing the average streamwise velocity ($u_\theta = {}^{(Re_i + Re_o)}/_2$) between the cylinders. To fulfil mass conservation, these regions are flanked by four alternating regions of axial up (green) and down (yellow) wash. This well-known phenomenon is a so-called Taylor-Vortex (TV), which is more thoroughly discernible through the vector representation of the two cross-stream velocity components u_r and u_z, which are also included in figure 12. You can easily create such a plot with **ParaView** by loading the corresponding ***.h5** file (Section 4.5.3), performing a coordinate transformation (Section 5.1) and playing around with the **contour**, **slice** and **glyph** filters. Loading the provided state file

(**taylorVortexIsoContourVectorPlot.pvsm**) into **ParaView** might also be helpful to learn creating such plots.

The Taylor-Vortex state – which is reached after a certain transient phase – is invariant in θ (axisymmetric) and also invariant in t (stationary). The first one can be easily seen from the discussed flow field representation shown in figure 12. The latter one can be easily seen from the time series of u_θ shown in figure 13. The streamwise velocity component recorded at three different radial locations ($r_i \leqslant r \leqslant r_o$) converges everywhere to a constant value, which is, however, different from the one theoretically predicted for the simpler Taylor-Couette flow state. Note, that this time we started our simulation from the analytical Taylor-Couette flow field and not from a resting fluid.

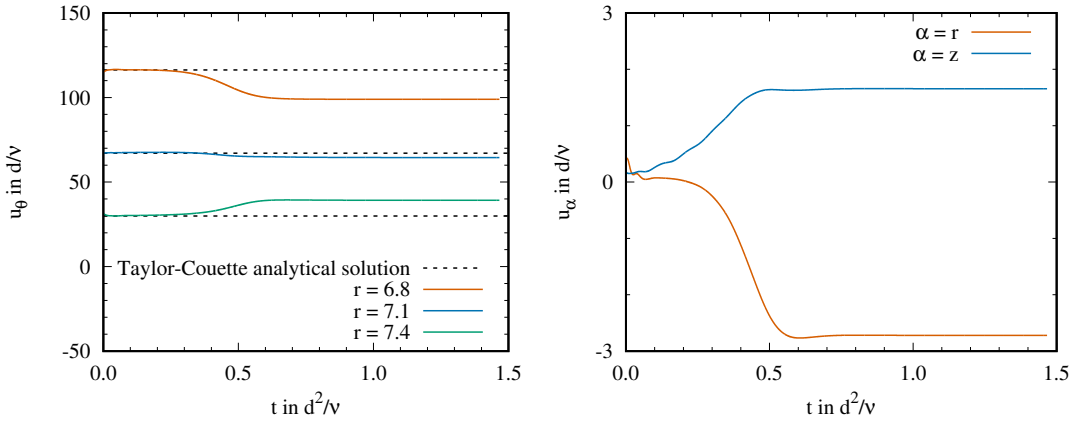


Figure 13: Temporal evolution of the velocity components at individual probe locations in our second Taylor-Couette tutorial **tc0041** with $Re_i = 150$. The streamwise velocity component (u_θ) is shown at three different radial locations ($r_i \leqslant r \leqslant r_o$) and converges everywhere to a constant value different from the theoretically predicted values for the simple Taylor-Couette flow state. The values of the two cross-stream components (u_r and u_z) also converges to constant but non-zero value, as shown exemplarily at one radial location ($r = 7.0$).

The values of the two cross-stream components (u_r and u_z) also converge to constant but non-zero values after a certain transient phase, see figure 13. Due to the presence of a Taylor-Vortex pair, stronger velocity gradients occur near the walls. These stronger gradients lead to higher shear-stresses and thus to higher torque values compared to pure Taylor-Couette flow state, as can be seen in figure 14. Since momentum and energy must be conserved, a minimal requirement for sufficient spatial resolution is that the torque at the inner cylinder matches that of the outer cylinder. The time series provided by **nsCouette** and plots like figures 14 allow to easily monitor the evolution of these quantities and hence check this resolution requirement. It can be easily seen, that both torque values match very well. For a more quantitative analysis, an additionally python script is also provided (**nsCouette/postprocessing/python**) to manipulate (plot, average, compare, etc) the torque time series. See also the thermal convection tutorials 7.5. The plots shown in figures 13, 14, 15 can be easily reproduced with the ready-to-use **gnuplot** scripts provided in this tutorial.

```
feldmann@darkstar:~/nsCouette/tc0041$ gnuplot probes.gpl
feldmann@darkstar:~/nsCouette/tc0041$ gnuplot torque.gpl
feldmann@darkstar:~/nsCouette/tc0041$ gnuplot keThZ.gpl
```

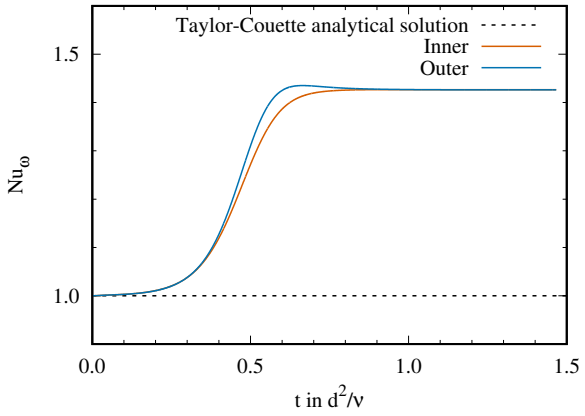


Figure 14: Temporal evolution of the torque at the inner and outer cylinder wall in our second Taylor-Couette tutorial **`tc0041`** with $Re_i = 150$. The torque is expressed in a non-dimensional way using a type of a Nusselt number, which relates the actual torque to the torque of the laminar Taylor-Couette analytical solution.

```
feldmann@darkstar:~/nsCouette/tc0041$ okular probes.pdf torque.pdf keThZ.pdf &
```

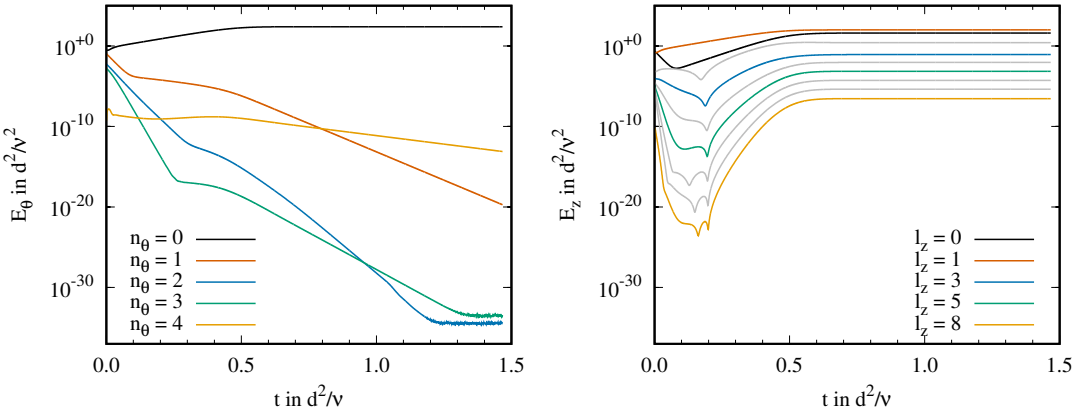


Figure 15: Temporal evolution of the azimuthally (left) and axially (right) dependent kinetic energy contained in each discrete azimuthal (n_θ) and axial (l_z) mode for our second Taylor-Couette tutorial **`tc0041`** with $Re_i = 150$.

7.4. Wavy vortex flow

Figrues 16 to 18 show the results for a wavy vortex flow state simulation, which can be reproduced analogue to the detailed description in tutorial 7.2 and 7.3. More tutorials will follow in the future!

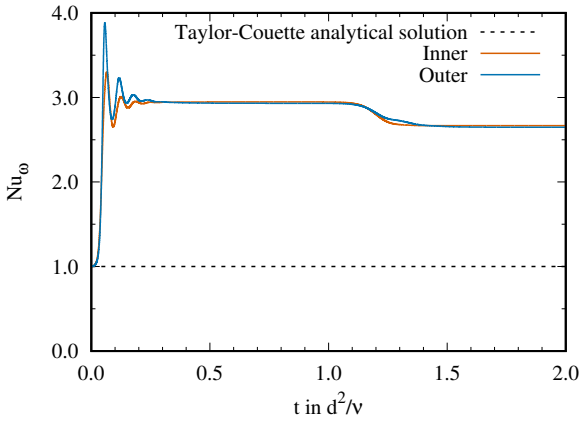


Figure 16: Temporal evolution of the torque at the inner and outer cylinder wall in our third Taylor-Couette tutorial **tc0042** with $Re_i = 458.1$. The torque is expressed in a non-dimensional way using a type of a Nusselt number, which relates the actual torque to the torque of the laminar Taylor-Couette analytical solution.

7.5. Thermal convection

The flow of air between a hot rotating cylinder and a cooled stationary cylindrical enclosure is a simple model to investigate heat transfer in rotating machines [6]. At low rotation rates and small temperature differences, the heat transfer is purely conductive and the flow has only an azimuthal component. In this simple case, the governing equations admit a simple analytic solution, termed basic state, whose temperature and velocity profiles depend only on the radial coordinate, see e. g. equations (2) to (4) in [8]. Heat transfer can be enhanced by either increasing the speed of the inner cylinder (forced convection), or by increasing the temperature difference (natural convection). In both cases, the basic state exhibits a sequence of distinct instabilities leading to turbulent heat transfer [8]. A measure of the efficiency is given by the Nusselt number Nu_i, which is the ratio of total heat transfer at the inner cylinder, normalised by the heat transfer of the purely conductive basic state at the same temperature difference.

Here, we want to reproduce this behaviour in a series of three short DNS runs. First, you have to build the temperature version (**TE_CODE**) of **nsCouette**, as described in section 3. Once this is done, go to your working directory, copy the first thermal convection tutorial case from the repository

```
cd ~/nsCouette
cp -r ../nscouette/tutorials/tc0073 .
cd tc0073
vi nsCouette.in
```

and inspect the parameter file (**nsCouette.in**) using your favourite text editor.

In this tutorial case directory you can find two ready-to-use scripts to generate the time series plots shown in figure **??** by simply typing

```
gnuplot probeCompare.gpl
gnuplot torqueCompare.gpl
okular probeComapre torqueCompare.pdf &
```

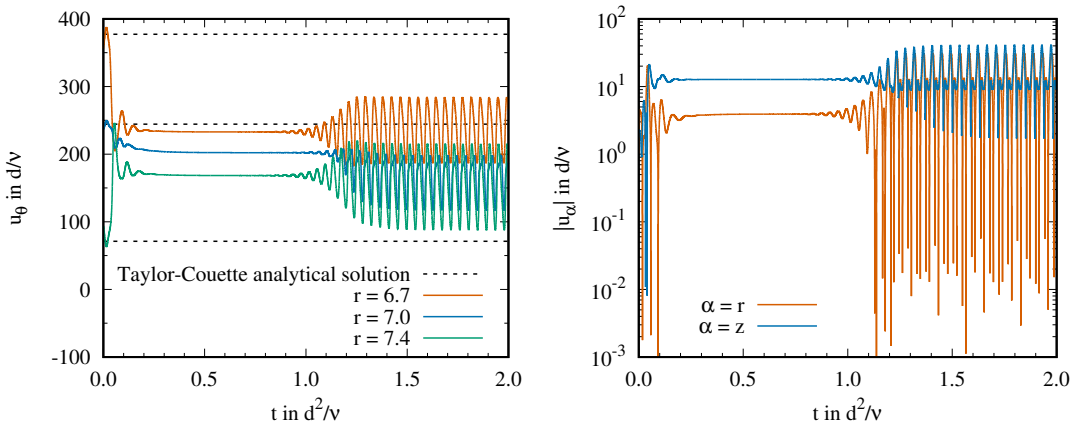


Figure 17: Temporal evolution of the velocity components at individual probe locations in our third Taylor-Couette tutorial **tc0042** with $Re_i = 458.1$. The streamwise velocity component (u_θ) is shown at three different radial locations ($r_i \leqslant r \leqslant r_o$) and converges everywhere to the theoretically predicted values (left). The absolute value of the two cross-stream components ($|u_r|$ and $|u_z|$) is shown exemplarily at one radial location ($r = 7.0$).

and opening the generated ***.pdf** figures using your favourite document viewer (e. g. **okular**, **evince**, **acroread** and alike).

Now we want to increase the effect of thermal convection by increasing the temperature difference between the inner and outer cylinder wall. In **nsCouette** this can be done by specifying a larger Grashof number. Go back to your working directory, copy the next convection tutorial case and inspect the input file

```
cd ~/nsCouette
cp -r ../nscouette/tutorials/tc0075 .
cd tc0075
vi nsCouette.in
```

using your favourite text editor. Note, that we now have increased Gr to 4000 and that we have also increased the axial resolution, since we expect a slightly more complex flow state. Moreover, we now want to run the simulation for another 400 000 timesteps (instead of 200 000 as before) to end up with roughly the same physical time span, since we now expect a slightly smaller Δt due to larger velocities and a finer grid resolution:

```
vi nsCouette.in
&parameters_grid
...
m_z0 = 32  ! L axial Fourier modes => 2*m_z0+1 grid points (axial)
/
&parameters_physics
...
Gr = 4000.0d0 ! Grashof number Gr = Ra/Pr [TE_CODE only]
/
&parameters_timestep
...
numsteps = 400000   ! number of computational timesteps
/
&parameters_output
...
fBase_ic = 'tc0075' ! identifier for coeff_ (checkpoint) and fields_ (hdf5) files
```

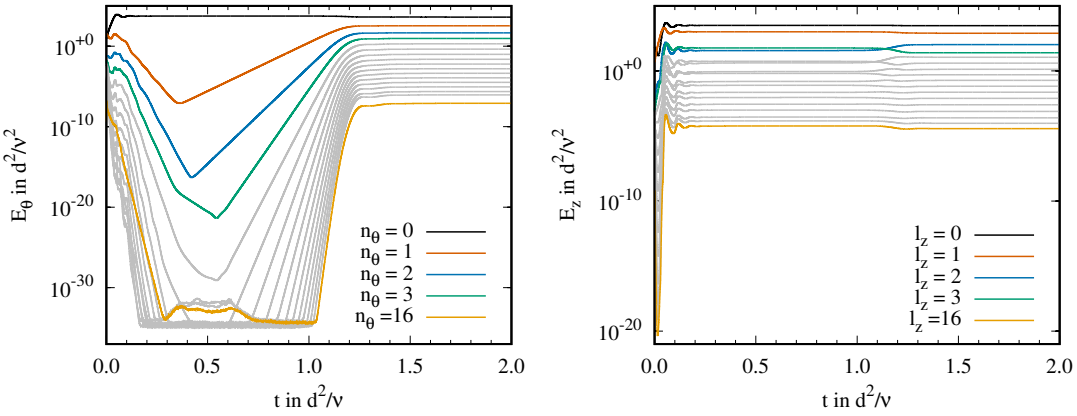


Figure 18: Temporal evolution of the azimuthally (left) and axially (right) dependent kinetic energy contained in each discrete azimuthal (n_θ) and axial (l_z) mode for our third Taylor-Couette tutorial **`tc0042`** with $Re_i = 458.1$. See also page 3 in [10] for definitions and details on the spatial discretisation scheme of **`nsCouette`**. As expected, the non-zero kinetic energy introduced by the finite amplitude perturbation in the initial velocity field decays monotonically to practically zero in all modes and never increases again.

```
/
&parameters_control
...
restart = 1    ! initial conditions, start from: 0=scratch, 1,2=checkpoint
/
```

To specify the initial conditions, create a symbolic link to the last flow field snapshot from the former DNS at the lower Gr

```
ln -s ../tc0073/coeff_tc0073.00200000 coeff_tc0075.00200000
cp ../tc0073/coeff_tc0073.00200000.info restart
```

and also create a file **`restart`** from the corresponding **`*.info`** file. The file **`restart`** has to be slightly modified to account for the new case/base name

```
vi restart
&parameters_restart
...
fbase_ic = tc0075
/
&parameters_info
...
/
```

using your favourite text editor.

Figure 19 summarises the results of three DNS with increasing temperature difference at a fixed rotation rate, using **`nsCouette`**. Scripts (**`gnuplot`**) and state files (**`ParaView`**) to create the shown plots are provided in the tutorial case directories.

The first DNS was initialized by applying small single harmonic disturbances to the basic state at $Re_i = 50$ and $Ra = 2130$. Fig. 19a shows that initially $Nu_i \approx 1$, corresponding to purely conductive heat transfer. However, after roughly two viscous time units, a sharp increase of Nu_i is observed, indicating that the basic state has become unstable. This is confirmed by the time-series of u_θ at a fixed mid-gap probe location in the computational

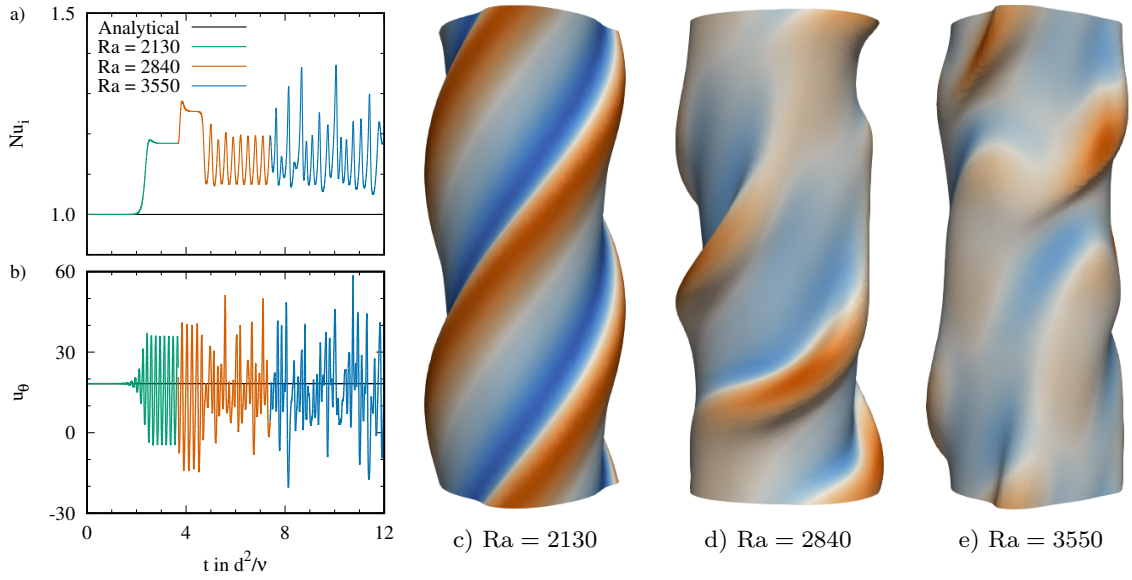


Figure 19: Different flow states for increasing Grass number when restarted the simulation from a former run. Temporal evolution of the torque at the inner cylinder wall and the streamwise (azimuthal) velocity component at one mid-gap probe location. Iso-surfaces of the mean Temperature ($T = 0$) colour-coded by the inwards (blue) and outwards (red) directed wall-normal (radial) velocity component.

domain, see Fig. 19b. The final state of this run ($t \approx 3.5 d^2/\nu$) is visualized in Fig. 19c, which shows a three-dimensional rendering of a $T = 0$ iso-surface generated with **`ParaView`**. The temperature surface is color-coded by inward/outward (blue/red) facing values of the wall-normal velocity component u_r. By saving several snapshots, a movie can be easily produced, which reveals a spiral flow pattern rotating at a constant speed like a barber pole. This explains why the u_θ signal reaches a state where it is periodic in time: the spiral pattern passes repeatedly through the probe location at which the velocity is recorded without changing its shape. This also explains why the integral heat flux (Nu_i) remains constant. The second and third DNS were initialized with the final state of the former runs and by increasing the Rayleigh number to $Ra = 2840$ and 3550, respectively. The time series and the final states of these runs are also shown in Fig. 19. They reveal that the flow state undergoes a sequence of transitions to different flow states with increasing spatio-temporal complexity as Ra increases.

As already mentioned in the tutorials before (e.g. 7.3), the time series output of the modal kinetic energy (figures 11, 15, and 18) can be used to check the spatial resolution of a simulation. Another minimal/necessary requirement for sufficient spatial resolution is that the torque at the inner cylinder matches that of the outer cylinder. The same holds for the Nusselt number. This is because momentum and energy must be conserved in the system. The time series provided by **`nsCouette`** allow to easily monitor the evolution of these quantities and hence check this resolution requirement. For a quantitative analysis, a python script is provided average, plot and compare the torque time series, see figure 20.

```
feldmann@fsmcluster:~/nsCouette/tc0076$ python torque.py
Read time series data from file torque
Data from t = 7.3975195320694 to t = 12.97130416704
```

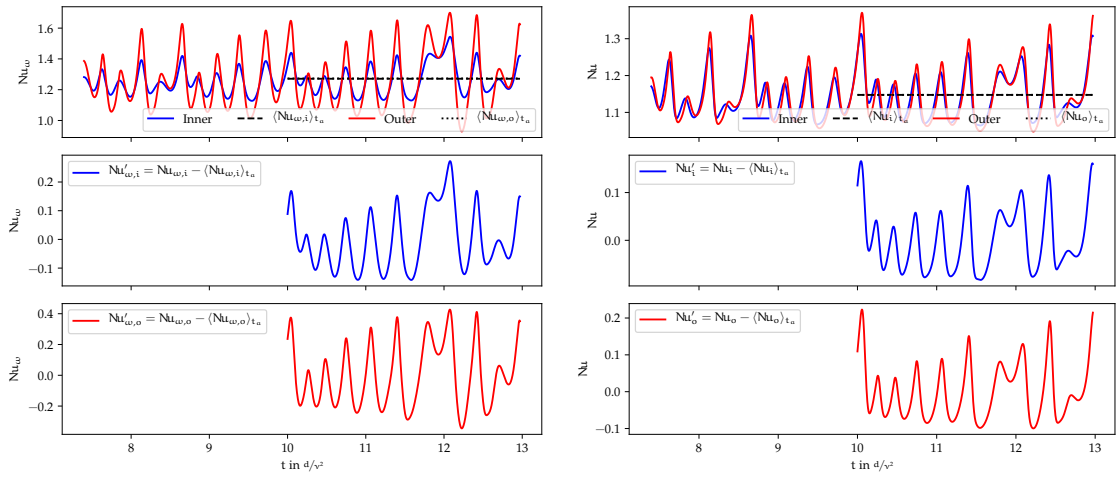


Figure 20: Temporal evolution of the torque Nusselt number and the Nusselt number at the inner and outer cylinder. The black dashed lines represent time averaged values at the inner and outer cylinder walls. The length of the black dashed lines represent the averaging interval. Python scripts to reproduce such plots are shipped with the code.

```
Temporal averaging from ta0 = 10.000006327682  to ta1 = 12.97130416704
Averaged inner torque <ti>_ta = 1.2710938766304294
Averaged outer torque <to>_ta = 1.2731075611260696
Absolute torque difference  o-i    =  0.002013684495640211
Relative torque difference (o-i)/o =  0.0015842138276822885
Written file torque.pdf
feldmann@fsmcluster:~/nsCouette/tc0076$ python Nusselt.py
Read time series data from file Nusselt
Data from t = 7.3975195320694 to t = 12.97130416704
Temporal averaging from ta0 = 10.000006327682  to ta1 = 12.97130416704
Averaged inner torque <ti>_ta = 1.1474836522546679
Averaged outer torque <to>_ta = 1.1476308708970335
Absolute torque difference  o-i    =  0.00014721864236566518
Relative torque difference (o-i)/o =  0.00012829694094238135
Written file Nusselt.pdf
feldmann@fsmcluster:~/nsCouette/tc0076$
```

8. Frequently asked questions (FAQ)

1. Q: How do I compile the code for multiple processors?

   A: There is no special compile-time-option (Section 3.4.2) to generate such things like a serial or parallel executable. The number of processors can freely be selected at *runtime*, as described in the section 4.

2. Q: Why?

   A: Because!

A. List of parameters

Starting with **nsCouette** version **1.0**, the file **mod_params.f90** needs no more editing before building the code. Instead, most of the settings and parameters can be specified at run-time as described in section 4.1. So in general, **nsCouette** has to be build only once on a system. The same executable can be used for all your simulations within one project, as long as you don't want to change something in the code or want to switch to e. g. simulations with thermal convection. The latter one is described in section 6.7 and in the tutorials section 7.5. Another example for when you need to modify source files and rebuild the code, is changing the parameters of the time stepper, which is detailed in section 6.6.

For reference and completeness, we here print the **mod_params.f90** file from a pre-**1.0** code version; also in order to document the geometrical and numerical parameters that were used for the example of Figure 5 in our CAF paper [10]. The variable names in the source code are more or less compatible with the symbols and notation used in our CAF paper.

```
INTEGER(KIND=4),PRIVATE   :: ir
!--- Mathematical constants
REAL(KIND=8)   ,PARAMETER :: epsilon = 1D-10  ! numbers below it = 0
REAL(KIND=8)   ,PARAMETER :: PI = ACOS(-1d0)  ! pi = 3.1415926...
COMPLEX(KIND=8),PARAMETER :: ii = DCMPLX(0,1) ! Complex i = sqrt(-1)

!--- Spectral parameters
INTEGER(KIND=4),PARAMETER :: m_r  = 32  ! Maximum spectral mode (> n_s-1)
INTEGER(KIND=4),PARAMETER :: m_th = 16  ! Number of Fourier modes
INTEGER(KIND=4),PARAMETER :: m_z0 = 16
INTEGER(KIND=4),PARAMETER :: m_z = 2*m_z0
INTEGER(KIND=4),PARAMETER :: m_f  = (m_th+1)*m_z ! Total number of Fourier modes
REAL(KIND=8)   ,PARAMETER :: k_th0= 6.D0   ! Minimum azimuthal wavenumber
REAL(KIND=8)   ,PARAMETER :: k_z0 = 2*PI/2.4  ! Minimum axial wavenumber

!--- Physical parameters
INTEGER(KIND=4),PARAMETER :: n_r  = m_r ! Number of grid points
INTEGER(KIND=4),PARAMETER :: n_th = 2*m_th
INTEGER(KIND=4),PARAMETER :: n_z  = m_z
INTEGER(KIND=4),PARAMETER :: n_f  = n_th*n_z  ! Number of points in Fourier directions
REAL(KIND=8)   ,PARAMETER :: len_r  = 1d0  ! Physical domain size
REAL(KIND=8)   ,PARAMETER :: len_th = 2*PI/k_th0
REAL(KIND=8)   ,PARAMETER :: len_z  = 2*PI/k_z0
REAL(KIND=8)   ,PARAMETER :: eta = 8.68d-1 ! Radii ratio r_i/r_o
REAL(KIND=8)   ,PARAMETER :: r_i = eta/(1-eta)   ! Inner radius
REAL(KIND=8)   ,PARAMETER :: r_o = 1/(1-eta)  ! Outer radius
REAL(KIND=8)   ,PARAMETER :: r(n_r) = (/(((r_i+r_o)/2 &
- COS(PI*ir/(n_r-1))/2), ir=0,(n_r-1))/) ! Chebyshev distribution for radial points
REAL(KIND=8)   ,PARAMETER :: th(n_th) = (/(ir*len_th/n_th,ir=0,n_th-1)/)
REAL(KIND=8)   ,PARAMETER :: z(n_z) = (/(ir*len_z/n_z,ir=0,n_z-1)/)
REAL(KIND=8)   ,PARAMETER :: gap = 3.25d0  ! gap size in cm
REAL(KIND=8)   ,PARAMETER :: gra = 980  ! gravitational acceleration in g/cm**3
REAL(KIND=8)   ,PARAMETER :: nu = 1.01d-2  ! kinematic viscosity in cm**2 /s

!--- Time stepper
REAL(KIND=8), PARAMETER :: d_implicit = 0.51d0  ! implicitness
REAL(KIND=8), PARAMETER :: tolerance_dterr = 5.0d-5   ! tolerance for corrector step

!--- MPI & FFTW parameters
INTEGER(KIND=4), PARAMETER :: root = 0   ! Root processor
INTEGER(KIND=4), PARAMETER :: fftw_nthreads = 1
LOGICAL  , PARAMETER :: ifpad = .TRUE.   ! If apply '3/2' dealiasing
```

```
!--- Finite differences parameters
INTEGER(KIND=4),PARAMETER :: n_s = 9 ! Leading length of stencil

!--- Defaults for runtime parameters
REAL(KIND=8) :: Courant = 0.25d0
INTEGER(KIND=4) :: print_time_screen = 250
LOGICAL   :: variable_dt= .true.
REAL(kind=8) :: maxdt = 0.01d0
INTEGER(KIND=4) :: numsteps = 100000 ! Number of timesteps
INTEGER(KIND=4) :: dn_coeff = 5000   ! Output coefficients every nth step
INTEGER(KIND=4) :: dn_ke = 100 ! Output energy every nth step
INTEGER(KIND=4) :: dn_vel = 100   ! Output velocity every nth step
INTEGER(KIND=4) :: dn_Nu = 100 ! Output Nusselt every nth step
INTEGER(KIND=4) :: dn_hdf5 = 1000 ! Output HDF5 every nth step
```

B. Configure and build additional software manually

B.1. Build `zlib`

This is a detailed description of how to manually build **`zlib`** libraries on our local **`fsmcluster`** at the ZARM institute. This is required for installing **`HDF5`**, as described in section B.2

```
# Brief comments on how to install zlib locally on fsmcluster@zamr, which is
# needed for HDF5, which is needed for NetCDF with netcdf-4 capabilities.
#
# Daniel Feldmann, 18th January 2017
#
# Download and unpack to some directory of your choice, \eg ~/zlib/build
tar xfvz zlib-1.2.11.tar.gz
cd zlib-1.2.11/
#
#
# Load Intel compiler suite available on fsmclsuter
module purge
module load Intel/PSXE2017
source /home/centos/Intel/PSXE2017/bin/ifortvars.sh intel64
source /home/centos/Intel/PSXE2017/bin/compilervars.sh intel64
source /home/centos/Intel/PSXE2017/bin/iccvars.sh intel64
#
# Check compiler version
module list
mpiicc --version
#
# Use Intel C compiler for the following steps, whatever icc or mpiicc, is not
# important as far as I know...
# export CC=icc
export CC=mpiicc
export CFLAGS="-O3 -xHost -ip -mcmodel=medium"
#
# Configure zlib to be installed into some place of you wish, \eg as follows
mkdir ~/zlib/zlib-1.2.11
./configure --prefix=/home/feldmann/zlib/zlib-1.2.11
#
# Build, test and install zlib via
make
make check
make install
#
# Do not forget to unset environment varibles, which might mess up future builds
unset CC
```

```
unset CFLAGS
#
# You may need to add the path to the newly installed library to the
# LD_LIBRARY_PATH environment variable if that lib directory is not searched by
# default. \eg put the following line to your ~/.bashrc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/feldmann/zlib/zlib-1.2.11/lib
#
# Done!
```

This is a detailed description of how to manually build **zlib** libraries on my local desktop machine, which might be useful as a rough guide line. This is required for installing **HDF5**, as described in section B.2

```
# Brief comments on how to install zlib locally on darkstar@zarm, which is
# needed for HDF5, which is needed for NetCDF with netcdf-4 capabilities.
#
# Daniel Feldmann, 6th January 2017
#
tar xfvz zlib-1.2.11.tar.gz
cd zlib-1.2.11/
#
#
# Install openmpi and GNU C compiler if not already installed, both easily
# available via the package manager
sudo apt-get install openmpi-bin openmpi-common libopenmpi1.10 libopenmpi-dev mpi-default-
    bin mpi-default-dev
sudo apt-get install gcc gfortran g++
#
# Check compiler version
which mpicc
mpicc --version
#
# Use GNU C compiler for the following steps, whatever gcc or mpicc, is not
# important as far as I know...
# export CC=gcc
export CC=mpicc
export CFLAGS='-O3 -mcmodel=medium'
#
# Configure zlib to be installed into some place you wish, \eg as follows
mkdir ~/zlib/zlib-1.2.11
./configure --prefix=~/zlib/zlib-1.2.11
#
# Build, test and install zlib via
make
make check
make install
#
# Do not forget to unset environment varibles, which might mess up subsequent builds
unset CC
unset CFLAGS
#
# You may need to add the path to the newly installed library to the
# LD_LIBRARY_PATH environment variable if that lib directory is not searched by
# default. \eg put the following line to your ~/.bashrc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/feldmann/zlib/zlib-1.2.11/lib
#
# Done!
```

B.2. Build HDF5

This is a detailed description of how to manually build **MPI**-parallel **HDF5** libraries with **Fortran** interfaces on our local **fsmcluster** at the ZARM institute. This requires a **zlib**

installation, as described in section B.1

```
#
# Download and unpack to some directory of your choice, \eg ~/hdf5/build/.
tar xvf hdf5-1.10.0-patch1.tar
cd hdf5-1.10.0-patch1
#
# Load Intel compiler suite
module purge
module load Intel/PSXE2017
source /home/centos/Intel/PSXE2017/bin/ifortvars.sh intel64
source /home/centos/Intel/PSXE2017/bin/compilervars.sh intel64
source /home/centos/Intel/PSXE2017/bin/iccvars.sh intel64
#
# Check compiler version
module list
mpiicc --version
mpiifort --version
#
# Define parallel Intel compilers to be used
export CC=mpiicc
export CCP="mpiicc -E"
export FC=mpiifort
#
# Set compiler flags for larger file size support and also optimisation level 3
export CFLAGS="-O3 -mcmodel=medium -xHost -ip"
export FCFLAGS="-O3 -mcmodel=medium -xHost -ip"
#
# Specify path to locally installed zlib library
export LIBS="-lz"
export LDFLAGS="-L/home/feldmann/zlib/zlib-1.2.11/lib"
export CPPFLAGS="-I/home/feldmann/zlib/zlib-1.2.11/include/"
#
# Configure HDF5 to be build for parallel use with fortran interface, as well as
# the use of the local installation of zlib and to be installed into some
# place you wish, \eg as follows
mkdir ~/hdf5/hdf5-1.10.0-patch1
./configure --enable-parallel --enable-fortran --with-zlib=/home/feldmann/zlib/zlib-1.2.11
     --prefix=/home/feldmann/hdf5/hdf5-1.10.0-patch1
#
# Build, test and install HDF5 as follows, while building and testing take quite
# some time... (enough for lunch and/or coffee)
make
make check
make install
#
# During building a lot of warnings come up like 'non-pointer conversion from
# "int" to "char" may lose significant bits' Absolutely no idea if this might
# be a problem...
#
# Do not forget to unset environment varibles, what might mess up future builds
unset CC CPP FC
unset CFLAGS FCFLAGS CPPFLAGS LDFLAGS LIBS
#
# You may need to add the path to the newly installed library to the
# LD_LIBRARY_PATH environment variable if that lib directory is not searched by
# default. \eg put the following line to your ~/.bashrc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/feldmann/hdf5/hdf5-1.10.0-patch1/lib
#
# Done!
```

This is a detailed description of how to manually build **MPI**-parallel **HDF5** libraries with **Fortran** interfaces on my local desktop computer using **gcc**. This requires a **zlib** installation, as described in section B.1

```
 Brief comments on how to install HDF5 locally for parallel use and fortran
# interface for the use with openpipeflow on darkstar@zarm.uni-bremen.de
# HDF5 is also required for NetCDF with netcdf-4 capabilities.
#
# Daniel Feldmann, 6th February 2017
#
# 1. Download and unpack to some directory of your choice, \eg ~/hdf5/build/.
tar xvf hdf5-1.10.0-patch1.tar
cd hdf5-1.10.0-patch1
#
# 2. Make sure zlib version 1.2.5 or later is installed (do not confuse with
# libzc library)
ls /*/*zlib* /*/*/*zlib*
echo $LD_LIBRARY_PATH | grep --color=auto zlib
#
# 3. Check GNU C and fortran compiler versions
which mpicc
mpicc --version
which mpif90
mpif90 --version
#
# 4. Set GNU C and fortran compiler, parallel (openmpi) versions
export CC=mpicc
export FC=mpif90
#
# 5. Set compiler flags for larger file size support and also optimisation
# level 3 for the fortran compiler
export CFLAGS="-mcmodel=medium"
export FCFLAGS="-mcmodel=medium -O3"
#
# 6. Configure HDF5 to be build for parallel use with fortran interface, as well
# as the use of the local installation of zlib and to be installed into some
# place you wish, \eg as follows
mkdir ~/hdf5/hdf5-1.10.0-patch1
./configure --enable-parallel --enable-fortran --with-zlib=/home/feldmann/zlib/zlib-1.2.11
     --prefix=/home/feldmann/hdf5/hdf5-1.10.0-patch1
#
# 7. Build, test and install HDF5 as follows, while building and testing take quite
# some time... (enough for lunch and/or coffee). Note that the parallel tests
# will only succede on a parallel file system
make
make check
make install
#
# During building a lot of warnings come up like 'non-pointer conversion from
# "int" to "char" may lose significant bits' Absolutely no idea if this might
# be a problem...
#
# 8. Do not forget to unset environment varibles, what might mess up future
# builds
unset CC FC
unset CFLAGS FCFLAGS
#
# 9. You may need to add the path to the newly installed library to the
# LD_LIBRARY_PATH environment variable if that lib directory is not searched by
# default. \eg put the following line to your ~/.bashrc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/feldmann/hdf5/hdf5-1.10.0-patch1/lib
# Done!
```

C. Code variant for pipe flow (**nsPipe**)

The code **nsPipe** merges the numerical formulation of **openpipeflow** [11], an open source code to simulate pipe flow, with the hybrid parallel strategy of **nsCouette**, creating a highly scalable solver to simulate pipe flow at large values of the Reynolds number. The source code structure of **nsPipe** was conceived to follow that of **nsCouette** as closely as possible. Both codes share the same hardware and software requirements (see sections 3.1 and 3.2). Likewise, **nsPipe** can be compiled following the steps described in section 3. Note, however, that the executable obtained after compilation is in this case named **nsPipeFlow.x**. Also from a user perspective, **nsPipe** works in the same way as **nsCouette**, and so the reader is encouraged to read in detail all sections in this user guide and, in particular, follow the tutorials in section 7, before using **nsPipe**.

Here, we will only briefly discuss those details that are specific to **nsPipe**, with emphasis placed on I/O parameters or operations that differ with respect to **nsCouette**.

C.1. Governing equations and parameters

We consider a fluid flowing through a straight circular pipe of constant cross section. Fluid motion is governed by the Navier-Stokes and continuity equations in cylindrical coordinates (eq. 12). These are rendered dimensionaless by choosing the pipe radius (R), the centreline velocity of the laminar profile (u_c) and the dynamic pressure ρu_c^2 as characteristic length, velocity and pressure scales, respectively.

$$\boldsymbol{\nabla}\cdot\mathbf{u}=0 \quad \text{and} \quad \frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u}=\frac{\boldsymbol{\nabla}p^*}{\rho}+\frac{1}{Re}\boldsymbol{\Delta}\mathbf{u}+F\hat{e}_z \tag{12}$$

The control parameter here is the dimensionless Reynolds number, $Re=\frac{u_c R}{\nu}$, which quantifies the ratio of inertial to viscous forces. If Re is small ($\sim O(100)$) velocity fluctuations are damped by the viscous forces and the flow remains laminar. Unlike certain regimes of Taylor-Couette flow, where turbulence is achieved after the flow passes through a sequence of increasingly complex states, turbulence in pipes arises suddenly. I. e. the transition is subcritical. First observations of turbulence in pipe flow can be made at $Re\approx 1850$ where localized turbulent patches known as puffs appear transiently. Other complex spatio-temporal dynamics such as puff splitting and turbulence spreading ("slugs") are observed as Re increases during the so called transitional regime ($2050 \lesssim Re \lesssim 2800$). Persistent space filling turbulence occurs eventually at $Re \gtrsim 3000$.

The term $F\hat{e}_z$ in eq. (12) denotes the volumetric force driving the fluid motion. There are two ways to drive motion in pipes.

1. Pressure driven flow. A constant pressure gradient between inlet and outlet is imposed. Under this condition the wall shear stress is fixed but the bulk (average) velocity and therefore the Reynolds number will change throuhout the simulation.

2. Flow rate driven flow. A constant flow rate is imposed, i. e. the bulk velocity is constant. Under this condition the Reynolds number is constant but the wall shear stress will fluctuate.

Both approaches are implemented in **nsPipe**.

C.2. Boundary conditions

Equations (12) require of boundary conditions to be solved. In **nsPipe**, periodic boundary conditions are imposed for both velocity and pressure in the streamwise and azimuthal directions. In the radial direction boundary conditions have to be imposed at the pole ($r = 0$) and at the pipe wall ($r = R$). Since the equations in cylindrical coordinates are singular at the pole this must be excluded from the radial grid. The boundary closure is carried out by enforcing the proper symmetry of each azimuthal Fourier mode at $r = 0$. That is, if for a given azimuthal Fourier mode n, a variable X (pressure or velocity) is an even function in r, $X(r) = X(-r)$, its radial derivative must be zero at the pole, $\partial_r X = 0$. However, if X is an odd function in r, $X(r) = -X(-r)$, the value of the variable at the pole must be zero, $X = 0$. The pressure and the streamwise velocity will be even functions if n is even and odd functions otherwise. In contrast, the radial and azimuthal velocity components will be odd functions if n is even and viceversa. At the pipe wall, no-slip boundary conditions are imposed for the velocity field and a homogeneous Neumann boundary condition is used for the pressure,

$$\mathbf{u}(r = R, \theta, z, t) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T, \quad \text{and} \quad \nabla p|_{wall} = 0 \tag{13}$$

We recall here that our solver does not require solving the pressure formally as the velocity is later corrected to satisfy both no-slip and divergence free boundary condition using an influence matrix technique.

C.3. Input file

As in **nsCouette** the simulation control parameters are defined in an input file named **nsPipe.in**, which is read from the standard input in the initial stage of the simulation. See also sections 4.1 for further details. This feature allows users to modify the control parameters without the need of recompiling the code. An example of an input file for **nsPipe** is given below.

```
&parameters_grid
m_r   = 32                ! radial points           => m_r      grid points (radial)
m_th  = 4                 ! azimuthal Fourier modes => 2*m_th+1 grid points (azimuthal)
m_z0  = 4                 ! axial Fourier modes     => 2*m_z0+1 grid points (axial)
k_th0 = 1.0d0             ! azimuthal wavenumber    => L_th = 2*pi/k_th0 azimuthal length
    of grid
k_z0  = 0.628318531d0     ! axial wavenumber        => L_z =  2*pi/k_z0  axial length of
    grid
/

&parameters_physics
Re = 100d0               ! Re
const_flux = T           ! T: constant flow rate; F: constant pressure gradient
/

&parameters_timestep
numsteps     = 20000     ! number of steps
init_dt      = 1.0d-4    ! initial size of timestep
variable_dt = T          ! use a variable (=T) or fixed (=F) timestep
maxdt        = 0.01      ! maximum size of timestep
Courant      = 0.25      ! CFL safety factor
/
```

```
&parameters_timestep
numsteps     = 20000      ! number of steps
init_dt      = 1.0d-4     ! initial size of timestep
variable_dt = T           ! use a variable (=T) or fixed (=F) timestep
maxdt        = 0.01       ! maximum size of timestep
Courant      = 0.25       ! CFL safety factor
/


&parameters_timestep
numsteps     = 20000      ! number of steps
init_dt      = 1.0d-4     ! initial size of timestep
variable_dt = T           ! use a variable (=T) or fixed (=F) timestep
maxdt        = 0.01       ! maximum size of timestep
Courant      = 0.25       ! CFL safety factor
/

&parameters_output
fBase_ic = 'DNS1'         ! identifier for coeff_ (checkpoint) and fields_ (hdf5) files
dn_coeff = 5000           ! output interval [steps] for coeff (dn_coeff = -1 disables ouput)
dn_ke    = 100            ! output interval [steps] for energy
dn_friction    = 100      ! output interval [steps] for friction parameters
dn_hdf5  = 1000           ! output interval [steps] for HDF5 output
print_time_screen = 100 ! output interval [steps] for timestep info to stdout
/

&parameters_control
restart = 0               ! initialisation mode: 0=new run, 1=restart from checkpoint (keep
    time), 2=restart from checkpoint (set time=0 and create new output files)
runtime = 86400           ! maximum runtime [s] for the job
/
```

Note that there are several differences with respect to the input file in **nsCouette** (section 4.1). Since the physical mechanisms underlying the flow dynamics differ between Taylor-Couette and pipe flows, the *parameter_physics* namelist contain different parameters. In **nsPipe** only two physical parameters are needed to set up a simulation: the Reynolds number (**Re**) and the force driving the fluid motion (**const_flux**). The latter is a logical variable. If it is set as true (T), motion is driven by a constant mass flow rate. Otherwise, the flow is pressure driven. A new parameter **dn_friction** appears in the *parameters_output* list. It indicates the frequency at which the output file named **friction** is updated. This file contains information about the bulk velocity or wall shear stress (depending on whether pressure driven or constant mass flow rate are chosen), centreline velocity of the mean velocity profile, friction velocity and friction factor. In addition to the **friction** file, a file containing the mean velocity profile (i. e. azimuthal and axially averaged streamwise velocity) is also produced every **dn_coeff** steps. Note that the **dn_probes** parameters, used in **nsCouette** to specify those physical locations at which time series of velocity (or temperature) are recorded, are absent in the current version of **nsPipe**. This functionality will however be added in future releases.

C.4. Initial condition

As explained in section 4.1.4 there exist three options to start the simulation and these can be specified in the input file using the **restart** parameter. When the initilization option **restart=0** is chosen, the simulation is started from the base flow (the classical Hagen-Poiseuille profile), which is perturbed during the initialisation phase. The disturbances

added to the base flow are implemented in the subroutine **pulse_init** contained in the module **mod_InOut**. By default, **nsPipe** implements a pair of streamwise localized rolls ($v = A(g + rg')\cos(\theta)e^{-w\sin^2(\pi z/L_z)}$ and $u = Ag\sin(\theta)e^{-w\sin^2(\pi z/L_z)}$, where u and v are the radial and azimuthal velocity components, L_z is the pipe axial length, r,θ and z are the three spatial directions in cylindrical coordinates, w fixes the axial length of the perturbation, $g = (1-r^2)^2$, and A is the amplitude of the disturbance (set to 0.1 by default). This perturbation is known to produce puffs at the transitional regime and turbulence at higher Reynolds numbers. Nevertheless, the user may need to modify A and w depending on the values of Re and L_z considered in the simulations. Unlike **nsCouette**, the current implementation of **nsPipe** does not allow to define these parameters in the input file. Hence, changes need to be done in the source code (**mod_InOut.f90**) and the code must be subsequently compiled.

D. Useful notes to start working with **git**

Say you work on a branch called **myBranch** to implement your own stuff to **nsCouette**. While you are working, the **master** branch or any other branch has been changed by you or by any other developer. Say a bug was fixed in another part of the code. Now, you want to incorporate this bug fix to your current status of **myBranch**. One option to do this would be:

```
feldmann@darkstar:~/nsCouette/nsCouette git checkout myBranch   # gets you on your branch
feldmann@darkstar:~/nsCouette/nsCouette git fetch origin  # gets you up to date
feldmann@darkstar:~/nsCouette/nsCouette git merge origin/master
```

The **fetch** command can be done at any point before the merge, i. e. , you can swap the order of **fetch** and **checkout**, because **fetch** just goes over to the named remote (here **origin**) and says to it: "gimme everything you have that I don't", i. e. , all commits on all branches. They get copied to your repository, but named **origin/branch** for any branch named **branch** on the remote.

At this point you can use any viewer (e. g. **git log**) to see "what they have" that you don't, and vice versa. Sometimes this is only useful for Warm Fuzzy Feelings ("ah, yes, that is in fact what I want") and sometimes it is useful for changing strategies entirely ("whoa, I don't want THAT stuff yet").

Finally, the **merge** command takes the given commit, which you can name as **origin/master**, and does whatever it takes to bring in that commit and its ancestors, to whatever branch you are on when you run the **merge**. You can insert **-no-ff** or **-ff-only** to prevent a fast-forward, or merge only if the result is a fast-forward, if you like.

References

[1] A. Brandstäter, J. Swift, H. L. Swinney, A. Wolf, J. D. Farmer, E. Jen, and P. J. Crutchfield. Low-dimensional chaos in a hydrodynamic system. *Physical Review Letters*, 51(16):1442–1445, 1983.

[2] A. Brandstäter and H. L. Swinney. Strange attractors in weakly turbulent Couette-Taylor flow. *Physical Review A*, 35(5):2207–2220, 3 1987.

[3] H. J. Brauckmann, M. Salewski, and B. Eckhardt. Momentum transport in Taylor-Couette flow with vanishing curvature. *Journal of Fluid Mechanics*, 790:419–452, 2016.

[4] B. Dubrulle, O. Dauchot, F. Daviaud, P.-Y. Longaretti, D. Richard, and J.-P. Zahn. Stability and turbulent transport in Taylor–Couette flow from analysis of experimental data. *Physics of Fluids*, 17(9):095103, 9 2005.

[5] A. Guseva, A. P. Willis, R. Hollerbach, and M. Avila. Transition to magnetorotational turbulence in Taylor–Couette flow with imposed azimuthal magnetic field. *New Journal of Physics*, 17(9):093018, 9 2015.

[6] D. A. Howey, P. R. N. Childs, and A. S. Holmes. Air-Gap Convection in Rotating Electrical Machines. *IEEE Transactions on Industrial Electronics*, 59(3):1367–1375, 3 2012.

[7] J. M. Lopez, F. Marques, and M. Avila. The Boussinesq approximation in rapidly rotating flows. *Journal of Fluid Mechanics*, 737:56–77, 12 2013.

[8] J. M. Lopez, F. Marques, and M. Avila. Conductive and convective heat transfer in fluid flows between differentially heated and rotating cylinders. *International Journal of Heat and Mass Transfer*, 90:959–967, 11 2015.

[9] K. Moreland. The ParaView Tutorial. *Technical Report SAND 2018-11803 TR*, Version 5., 2018.

[10] L. Shi, M. Rampp, B. Hof, and M. Avila. A hybrid MPI-OpenMP parallel implementation for pseudospectral simulations with application to Taylor–Couette flow. *Computers & Fluids*, 106:1–11, 1 2015.

[11] A. P. Willis. The Openpipeflow Navier–Stokes solver. *SoftwareX*, 6:124–127, 2017.