A user guide for nsCouette

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

13th May 2019

This is a hands-on introduction for **nsCouette**; A highly scalable software to integrate the full Navier–Stokes equations for incompressible fluid flows between differentially heated and independently rotating concentric cylinders forward in time. **nsCouette** 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 designed to compute turbulent flows at high Reynolds and Rayleigh numbers. An additional GPU-accelerated implementation (**CUDA**) for intermediate problem sizes as well as a basic version for turbulent pipe flow (**nsPipe**) is 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.

Text width: 152.72551mm

Contents

1. 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 oneself 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`**.

1.1. 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 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.

1.2. 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

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

for any given η. For a fixed geometry, the flow between the cylinders can than 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

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

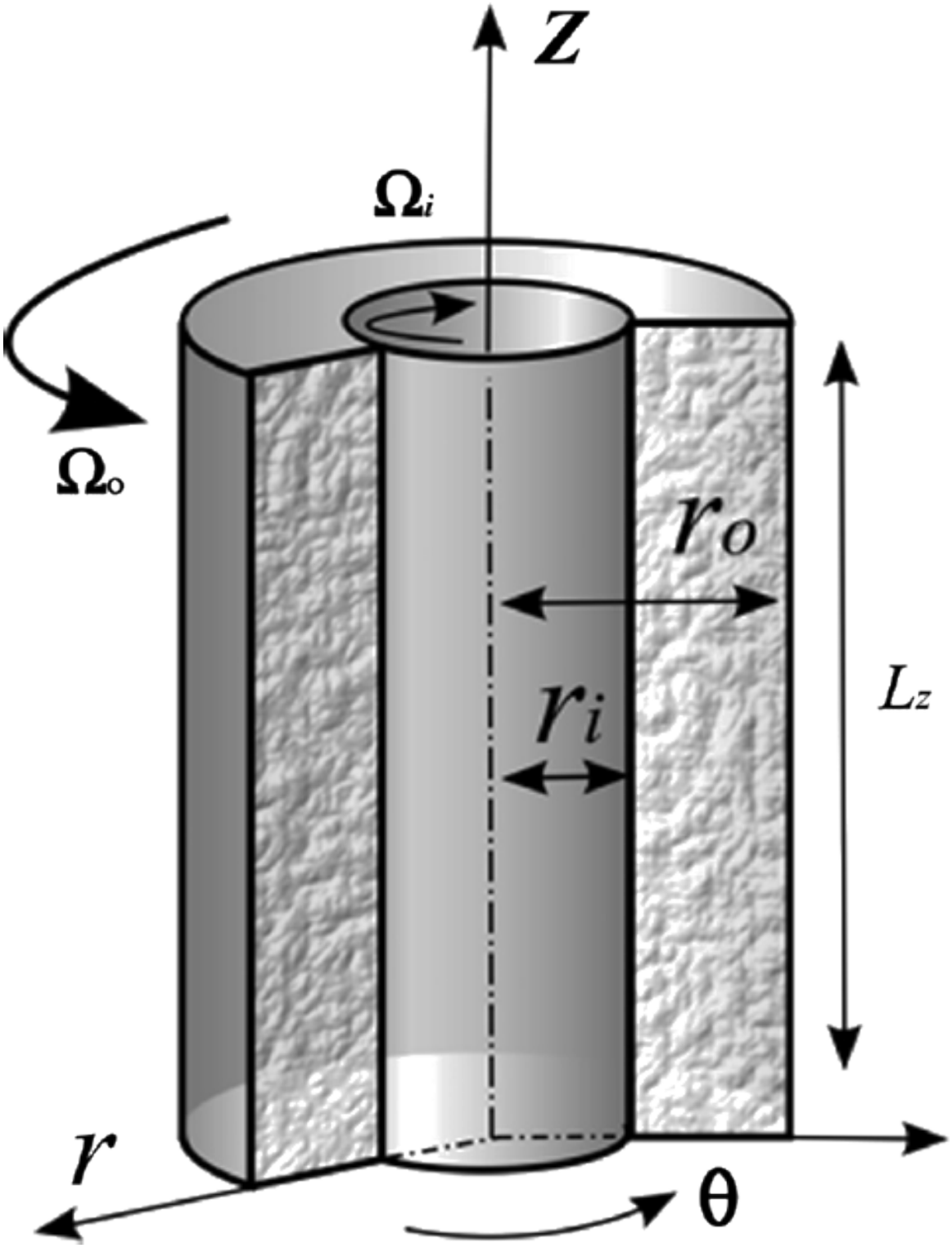


Figure 1: Schematic of the Taylor-Couette (TC) system in a cylindrical co-ordinate frame work (r, θ, z) and its relevant properties: Radius of the inner/outer cylinder wall ($r_{i,o}$), height of the computational domain (L_z), angular rotation speed of the inner/outer cylinder walls ($\Omega_{i,o}$). Sketch taken from our CAF paper [7].

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. The unit for the velocity is already implicitly defined through our choice for the unit length (d) and unit time (d^2/ν), and is thus given by ν/d.

2. Building the code

2.1. Hardware prerequisites

The code **nsCouette** is written in modern **FORTRAN** and over time has been ported to all major CPU-based high-performance computing (HPC) platforms, including IBM Power and BlueGene, as well as various **x86_64** architectures, including a few generations of Intel Xeon multi-core processors, AMD EPYC as well as Xeon Phi Knights Landing. The tutorials provided in § 6 of this user guide were designed to run on standard laptops and small Linux clusters.

2.2. 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 (§ 6) presented in this user guide, we assume that you use a **bash** shell, although not necessarily required.

2.2.1. 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 ... to ... Free software and further information can be found here:

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

2.2.2. Message passing interface

You need an **MPI** library which supports the thread-level **MPI_THREAD_SERIALIZED**, which 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**. 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

2.2.3. Linear algebra

You need a serial **BLAS**/**LAPACK** library. **nsCouette** has so far been tested with Intel's **MKL-11.x** 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

2.2.4. Fastest Fourier transform in the west

You need a serial but fully thread-safe **FFTW3** installation or equivalent. **nsCouette** has so far been tested with **FFTW-3.3** and with Intel's **MKL-11.1** to **MKL-11.3**. Earlier versions of **MKL** will likely fail. 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

2.2.5. Hierarchical data format

Optionally, you need an **MPI**-parallel **HDF5** library installation to enjoy full flow field output in primitive variables which can be readily visualised in **ParaView** (see § 4.1) or **VisIt** (see § 4.2). **nsCouette** has already been tested with **HDF5-1.8.x** and **HDF5-1.10.0-patch1**.

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

2.2.6. Software modules

If you are planing to run **nsCouette** on a supercomputer at a common national high-performance computing centre (e.g. LRZ, HLRS, JSC or HLRN in Germany), all of the above mentioned software packages will most likely be readily available as user-loadable modules. Some useful basic commands to start working with environmental modules are listed in the following.

```
feldmann@fsmcluster:~/$ module list
feldmann@fsmcluster:~/$ module avail
feldmann@fsmcluster:~/$ module load moduleName
feldmann@fsmcluster:~/$ module switch moduleName
feldmann@fsmcluster:~/$ module purge
```

2.2.7. Self-build libraries

If you are, however, 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 compilers, it is worth trying to install them using your favourite software package manager (**apt-get**, **rpm** and alike). For the rest, it might be 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 libraries (**zlib** and **HDF5**) for the use with **nsCouette** on our local **fsmcluster** at the ZARM institute. These examples can be used as a rough guideline to install all necessary software packages on your target Linux system.

2.3. Download the 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 § 6 – 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 **gitlab** 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://gitlab.mpcdf.mpg.de/mjr/nscouette
feldmann@darkstar:~/nsCouette$ cd nscouette
feldmann@darkstar:~/nsCouette/nscouette$ git checkout master
feldmann@darkstar:~/nsCouette/nscouette$ git checkout nscouette_GPU
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 to your local system. By default, you should be on the master branch, but you can easily switch to the branch **nscouette_GPU** using the **checkout** command to download the **CUDA** accelerated version instead of the regular one. The **git** commands **status** and **pull** help you to figure out on which branch you currently are and get you the latest changes (e.g. bug fixes or new features) from the remote repository, respectively. A few more helpful hints on how to use **git** can be found in appendix C.

2.4. Compile the code

Once you have downloaded the source files (§ 2.3), 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 for building **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 § 5.1 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 nsCouetteAtDarkstar.sh
    nsCouetteAtMyMachine.sh
feldmann@darkstar:~/nsCouette/nscouette/scripts$ vi nsCouetteAtMyMachine.sh
```

By modifying we mean, that you should load the correct modules (see § 2.2.6) and/or set all necessary path variables correctly in case you are using self-made libraries (see § **??**). 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 (see § 3).

```
feldmann@darkstar:~$ vi .bashrc
...
```

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

2.4.1. 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 § **??** and § B), make sure to set all necessary path variables correctly (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.

2.4.2. Compile-time options

Besides specifying the architecture file (§ 2.4.1), you can choose between a few other options to control the build process. The different options listed below can arbitrarily combined. They are meant to switch off **HDF5** flow field output (§ **??**), to include integrating the

temperature (for thermal convection and heat transfer § **??**), and to build the code in debug mode with an equation to The respective default 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>
```

`make ARCH=myplatform HDF5IO=<yes|no>`

HDF5IO is switched on by default. If HDF5IO is switched off (`make ... HDF5IO=no`) only binary MPI-IO output will be written. Checkpoint files are always written in binary MPI-IO format.

Compilation is detailed in Sect. 2.4.1. Type

`make (ARCH=...) clean`

to clean the compilation. Inside the brackets are optional.

3. Running the code

3.1. Parameter input file

Before running the program, we need to have an input file based on the following **FORTRAN** namelists

```
&parameters_grid
m_r    = 32   ! radial points  => m_r   grid points (radial)
m_th   = 16   ! azimuthal Fourier modes => 2*m_th+1 grid points (azimuthal)
m_z0   = 16   ! axial Fourier modes  => 2*m_z0+1 grid points (axial)
k_th0  = 6.0 ! azimuthal wavenumber => L_th = 2*pi/k_th0 azimuthal length of gri
k_z0   = 2.6179938779914944 ! axial wavenumber  => L_z =  2*pi/k_z0  axial lengt
eta    = 0.868d0    ! aspect ratio => r_i = eta/(1-eta),r_o = 1/(1-eta) inner/out
/

&parameters_physics
Re_i = 700d0    ! Re_i
Re_o = -700d0   ! Re_o
Gr   = 50d0 ! Gr
Pr   = 7d0   ! Pr
gap = 3.25d0    ! gap size in cm  [TE_CODE only]
gra = 980    ! 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 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 = 2000   ! output interval [steps] for coeff (dn_coeff = -1 disables c
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
print_time_screen = 100 ! output interval [steps] for timestep info to stdout
/

&parameters_control
restart = 0 ! initialization mode: 0=new run, 1=restart from checkpoint
runtime = 86400   ! maximum runtime [s] for the job
/
```

3.1.1. Boundary conditions

3.1.2. Initial conditions

To prescribe a resting fluid as initial condition, just set **`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 the wave number vector and an amplitude for each perturbation. The perturbations are designed to fulfil the no slip boundary condition at the wall and to be divergence free by construction. Figure 2 shows some useful examples of different initial conditions at $Re_i = 50$. Work in progress: Soon it will also be possible to read initial conditions from a **`coeff*`** file and add a perturbation on top by combining **`restart=1`** and **`ic_pert=T`**.

3.1.3.

3.2. On your laptop/desktop

Open a terminal. Go to your case directory. Copy the executable. And start a run by typing:

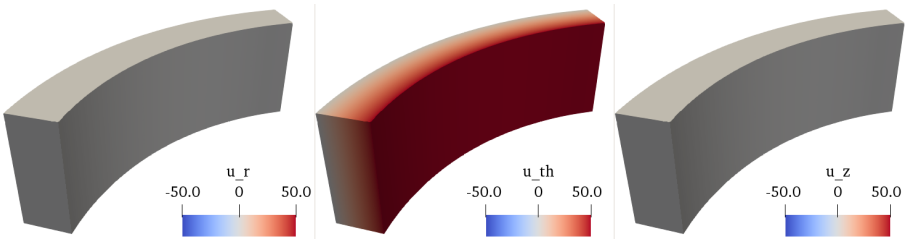
3.3. On a cluster

Go to the running directory and make sure that the following files are in the directory

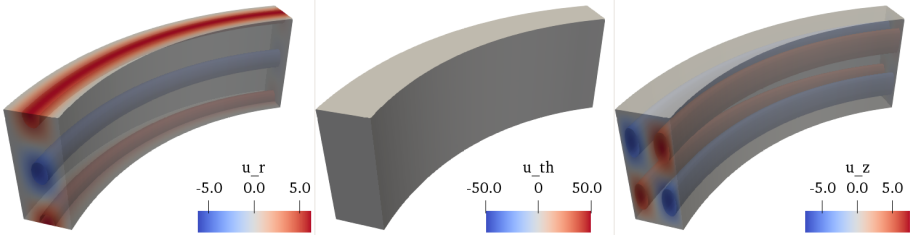
```
nsCouette.x input_nsCouette
```

Both files can be renamed. In the following the above names are assumed. Then type

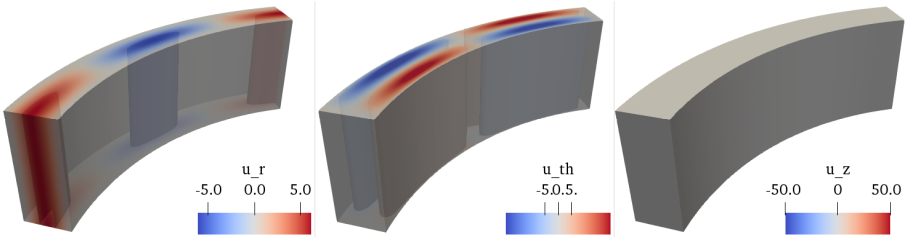
```
mpiexec -np 32 ./nsCouette.x < input_nsCouette
```



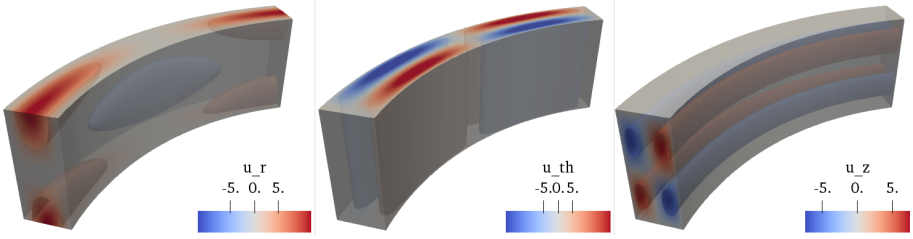
(a) Taylor-Couette analytical solution, no perturbation.



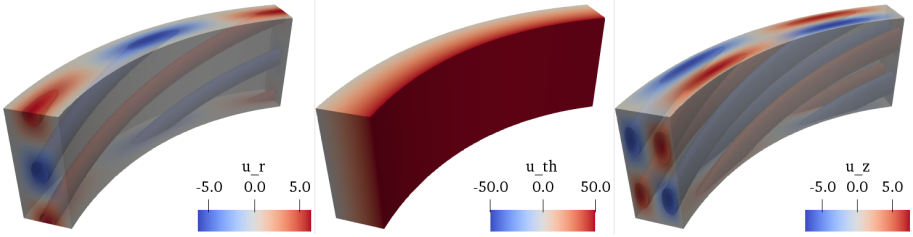
(b) Resting fluid, perturbed in mode $(l_{\theta,1}, n_{z,1}) = (0, 1)$ with amplitude $a_1 = 10^2$.



(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.

to run the program with 32 cores/MPI tasks.

Note that the number of MPI tasks selected at runtime must evenly divide the parameter `m_r` in the input file. Starting with `nsCouette 1.0` this restriction does *not* apply to `m_f` anymore.

In this example setting a number of numsteps=10000 time steps would be attempted to run, but the code would stop after a maximum runtime = 86400 seconds (24h) of ("wall-clock") time. We have implemented an intrinsic safety margin corresponding to the average number of seconds it takes to 2 timesteps.

If OpenMP is enabled (this is the default), set the environmental variable to specify the number of threads you would like to use. 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 4 OpenMP threads. The second line maps threads to (physical) cores. The fourth line sets the stacksize, experiment with larger or smaller values if unexpected SEGFAULTs occur or if the memory per core is scarce, respectively. These are just basic usage of MPI and OpenMP. For an advanced use of MPI and OpenMP (*e.g.*, `bash` script) or in case of problems, please consult your local IT support.

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.

Notes on runtime settings for hybrid MPI/OpenMP The most critical performance issue when running the code in hybrid mode (i.e. OpenMP enabled) can be an improper pinning of the MPI tasks and OpenMP threads to the CPU cores. We suggest basic settings above based on the `OMP_PLACES` environment variable from the OpenMP standard, but different compilers/runtimes and node architectures (NUMA, Hyperthreading, ...) might require specific settings. The actual mapping can be checked by examining the standard output of **`nsCouette`**, which, for an example with 16 nodes, 32 MPI tasks (2 tasks per node) and 12 OpenMP threads per task should read like this (the individual lines appear in random order):

```
TASKS:  32 THREADS:12 THIS:   0  Host:nid01226   Cores:  0  1  2  3  4  5  6  7  8  9
TASKS:  32 THREADS:12 THIS:   1  Host:nid01226   Cores: 12 13 14 15 16 17 18 19 20 21
TASKS:  32 THREADS:12 THIS:   2  Host:nid01227   Cores:  0  1  2  3  4  5  6  7  8  9
TASKS:  32 THREADS:12 THIS:   3  Host:nid01227   Cores: 12 13 14 15 16 17 18 19 20 21
TASKS:  32 THREADS:12 THIS:   4  Host:nid01228   Cores:  0  1  2  3  4  5  6  7  8  9
TASKS:  32 THREADS:12 THIS:   5  Host:nid01228   Cores: 12 13 14 15 16 17 18 19 20 21
[...]
```

Note, 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` (open source, see http://code.google.com/p/likwid/) 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* (hyperthreading is not beneficial for **`nsCouette`**) of the CPU, divided by the number of MPI tasks which we map to this CPU. On contemporary HPC clusters with 2 Intel or AMD CPUs per node, we typically use 2 MPI tasks per node and, for example set:

- `OMP_NUM_THREADS=8` (8-core CPUs, Intel Xeon E5-2670, *SandyBridge*)
- `OMP_NUM_THREADS=12` (16-core CPUs, Intel Xeon E5-2698v3, *Haswell*)
- `OMP_NUM_THREADS=20` (20-core CPUs, Intel Xeon Gold-6148, *Skylake*)
- ...

Note, however, that the OpenMP parallelism of the part of the code, where the radial derivatives of the velocity are Fourier-transformed to compute the nonlinear terms (performace label 'fft' in mod_nonlinear.f90), is limited by `3*mp_r` (or `4*mp_r`, if the `make CODE=TE_CODE` option is used), where $\mathrm{mp_r}$ is the number of radial points per MPI task. 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 4 or 8 MPI tasks per node on large multicore 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.

3.4. Output

3.4.1. Time series data

If the program runs correctly, it generates the following files once it starts running

```
ke_mode  ke_th  ke_total  ke_z
parameter  torque  vel_mid
```

The input and output files and their formats are specified in the source file `mod_inOut.f90`. These files are time series of different quantities, saved in columns every `dn_*` time steps as specified in the `input_nsCouette` file. The first column is time, whereas the other columns are the relevant physical quantities. You can plot them using `xmgrace` or `gnuplot`, or others.

3.4.2. Velocity field Fourier coefficients

The binary coefficient files will be generated every `dn_coeff` timestep, according to the setting of the parameter in the input file. Since in the example, we output the coefficient files every 2000 time steps within 10000 total time steps, there will 5 coefficient files in total. The name of the files will be like

```
coeff_DNS1.00002000
coeff_DNS1.00004000
...
coeff_DNS1.00010000
```

These files are the Fourier coefficients of the velocity field at one time step ("snapshot") and can also be used as checkpoints to continue a simulation (cf. Sect. 3.5). They can not be visualized straightforwardly but by some user-written post-processing utilities (*e.g.*, using the library `plplot`).

3.4.3. Physical space flow field data

However, if you set `HDF5IO=yes` at the compilation time, the program will output `hdf5`-formatted files and corresponding XDMF metadata files, as follows

```
fields_DNS1_00001000.h5   fields_DNS1_00001000.xmf
fields_DNS1_00002000.h5   fields_DNS1_00002000.xmf
...
fields_DNS1_00010000.h5   fields_DNS1_00010000.xmf
```

The `HDF5` format can be visualized with a lot of visualization softwares, such as **`VisIt`**, **`ParaView`**, etc.

3.5. Checkpoint-restart mechanism

On large HPC systems, the runtime of a single batch job is usually limited to something like 24 h. In order to enable long-running simulations, **`nsCouette`** implements a semi-automatic checkpoint-restart mechanism. Regardless of what you specify for **`dn_coeff`** in the parameter input file (see § 3.1), a Fourier coefficient file (**`coeff_*`**) 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 input file 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 (which corresponds the default setting `restart=0`). The initial time for the new simulation run is that of the initial condition and the time series data generated is appended to the existing output files.

A third initialization option (`restart=2`) is also available, which allows to start the simulation from a checkpoint file, but setting the time to zero and creating new output files. The grid size of the continued simulation can be modified by adapting the corresponding parameters in the `input_nsCouette` file, but m_r must remain to be divisible by the number of processors. Writing of checkpoints can be disabled by setting `dn_coeff = -1` in the input file. 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 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.

The checkpoint-restart mechanism allows submitting a chain of (many) individual jobs to a batch system at once and thus facilitates handling of long-running simulations with **`nsCouette`**. Basic functionalities of the batch system (e.g. options `-hold_jid` for SGE

or `--dependency` for SLURM) can be used to express the chain-type dependency of the individual jobs.

4. Post-processing

4.1. Visualisation with `ParaView`

To visualise and analyse instantaneous flow field data with the open source software tool **`ParaView`**, see fig. 3, just go through the introductory steps below and follow our tutorials in § 6. The tutorials also provide a few ready-to-use state files (**`*.pvsm`**) to start with.

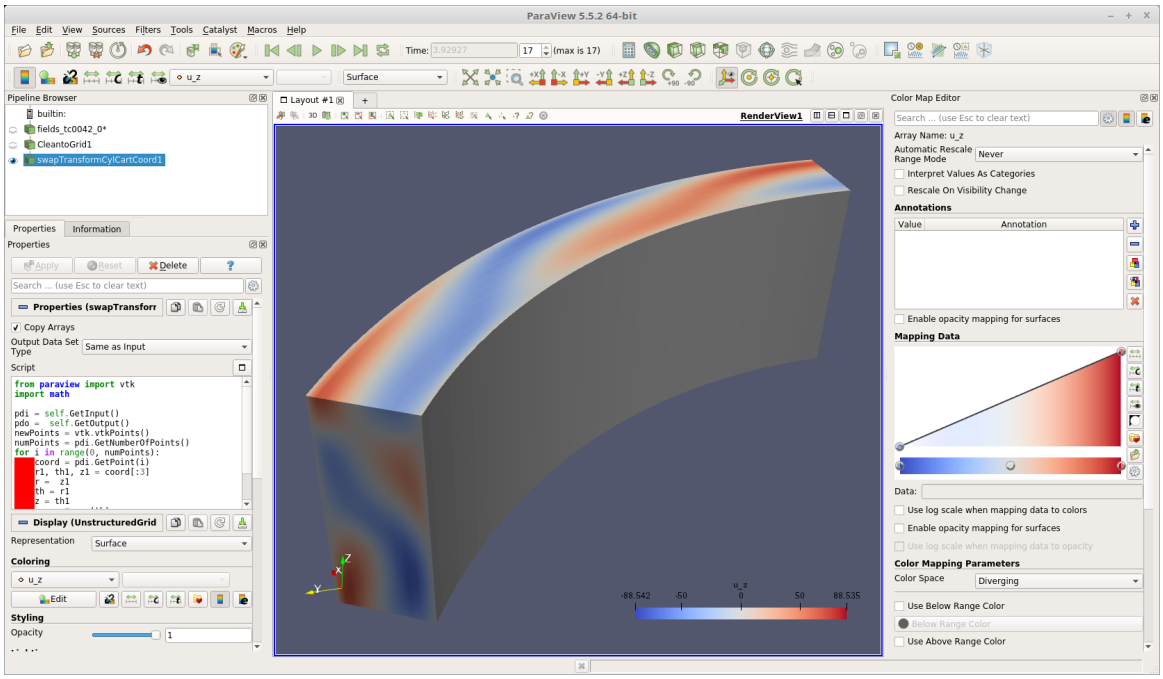


Figure 3: Screenshot of the **`ParaView`** GUI and the first basic steps to visualise and analyse the **`HDF5`** flow field output from **`nsCouette`**. Follow our tutorials in § 6 for further details.

Note, however, that this is just a collection of very basic quick-start recipes. For more information, please consult the official documentation [6] and the pertinent online discussion forums.

Preparation

- Download a recent pre-compiled version from the official web page `www.paraview.org` 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 costume-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 § **??**. To add the costume 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/visualization**
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 § 6.4)

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

  and start **ParaView**.

- Once the **ParaView** main window has appeared (something like in fig. 3), 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

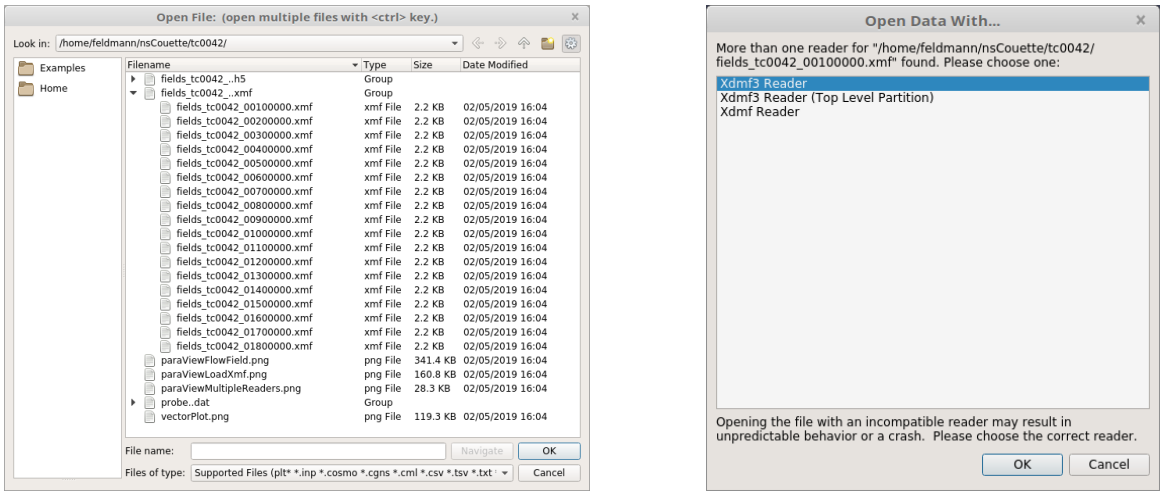


Figure 4: Screenshot of the file open dialogue window (left) where you can select the **xmf** files, which act as a reader or interface through which **ParaView** access the flow field data contained in the **HDF5** files. If there is more than one **xmf** reader, chose the first on in the appearing dialogue window (right).

and the corresponding **xmf** files as shown in figure 4. The **xmf** files are simple text files, which contain meta data about the content of the **HDF5** container. The **xmf** 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 **xmf** 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 **xmf** file. Note, however, that loading a large number of snapshots might take very long for large simulations!

- If a new dialogue window appears (fig. 4 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 3.

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 3, 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 10, 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 10, use the **Glyph** filter as exemplarily shown in the **ParaView** state file **vectorPlot.pvsm** which comes with the second tutorial described in § **??**.

4.2. Visualisation using VisIt

As an alternative to **ParaView** (§ 4.1), you can also use the open source software **VisIt** (fig. 5), 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 how to

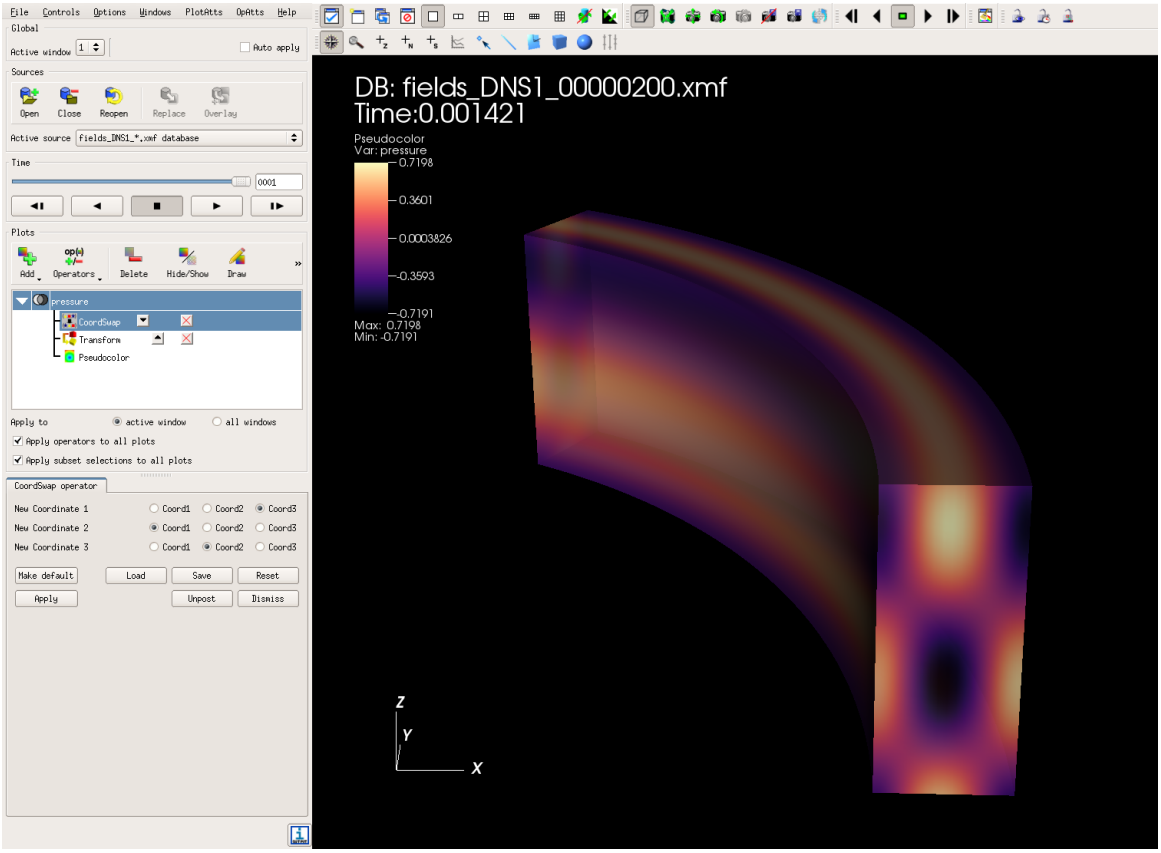


Figure 5: A screenshot of the **GUI** of **VisIt** showing a basic example of a pseudo-colour representation of the instantaneous pressure field in a small computational domain similar to the set-up used in the first tutorial in § 6.2.

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;

5. About the code

In this section, the code is further documented on a programming level; i.e. structure, data types, variables, parallelisation etc. Additionally, some details on selected methodological aspects are outlined.

5.1. Structure of the source code

The source code comprises only a few thousand lines of code, which are structured in a minimum set of twelve basic source files – including a **Makefile**. The different parts of the code are thematically structured into separate files and **FORTRAN** modules according to the following summary.

```
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
mod_hdf5io.f90   # HDF5 output of primitive flow field variables
mod_getcpu.f90   # Mapping of the CPU. This module cannot be included.
...
```

5.2. Programme flow chart

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

1. Initialisation phase: **MPI**, read parameter file (§ 3.1), set initial conditions (§ 3.1.2)
2. Time-stepping: linear sub-step computation and nonlinear pseudo-computation)
3. Finalising phase: **MPI**, checkpoint output (§ **??**)

5.3. Constants & variables

All constants are specified in `mod_params.f90`, while all variables are given in `mod_vars.f90`. The data type are mostly specified with `KIND=4 or 8`, indicating single or double precision. The derived data types in `mod_vars.f90` are `phys` for variables in physical field, `spec` for variables in Fourier space and `vec_mpi` for vector variables in each MPI subdomain. The type `phys` and `spec` contains all physically relevant quantities, $\mathbf{u}$ and p. The array of type `vec_mpi` is 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 the Fig.2 in our CAF paper [7].

5.4. Temporal discretisation

Since the publication of our CAF paper [7], 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 time step 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 [8]. 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

$$\nabla P_1^{q+1} = \nabla \cdot (Lu^q + N^q)$$
$$\left(\frac{1}{\delta t} - c\nabla^2\right) u_1^{q+1} = N^q + \left(\frac{1}{\delta t} - (1-c)\nabla^2\right) u^q - \nabla P_1^{q+1}$$

- 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

$$\nabla P_{j+1}^{q+1} = \nabla \cdot \left(Lu^q + N_j^{q+1}\right)$$
$$\left(\frac{1}{\delta t} - c\nabla^2\right) u_{j+1}^{q+1} = cN_j^{q+1} + (1-c)\, N^q + \left(\frac{1}{\delta t} - (1-c)\,\nabla^2\right) u^q - \nabla P_{j+1}^{q+1}$$

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 § **??**

- The coefficient c (d_implicit) defines the implicitness of the method and can be specified in mod_params.f90. Note that the temporal scheme is second-order 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

$$\delta t = C\min(\nabla/|v|), \tag{7}$$

where C is the Courant number

5.5. Features for thermal convection

The user can switch to **nsCouette** with temperature by setting CODE ?= TE_CODE in the Makefile. Alternatively, it can also be specifed when the code is built up

```
make ARCH=myplatform HDF5IO=<yes|no> CODE=TE_CODE
```

Note that the version without temperature (STD_CODE) is set by default.
In the version of **nsCouette** provided the flow is stratified radially by heating the inner cylinder and cooling the outer cylinder, i.e. a negative temperature gradient. The sense of the temperature gradient can be easily modified by inverting the boundary conditions for the temperature in mod_timeStep.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
```

and modifiying the temperature and axial velocity of the base flow accordingly in the subroutine base_flow contained in mod_InOut.f90.
The temperature is coupled to the Navier-Stokes equations by considering a Boussinesq-like approximation that includes centrifugal buoyancy effects. The dimensionless governing equations and Boussinesq-like appproximation implemented can be found in Journal of Fluid Mechanics,Volume 73, December 2013, pp. 56-77.
The code produces two additional time series outputs. The `Nusselt` file contains time series for the Nusselt number at the inner and outer cylinders, whereas the file **Temp_energy** includes the temporal variation...

Time series of the temperature at the some user specified probe locations do now appear in the files **probes*.dat**.

6. Tutorials

6.1. 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 the version **gnuplot 5.0**, which you can check by typing

```
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 get **ParaView**
- Or — if you prefer — get **VisIt**
- A proper LaTeXinstallation might also be useful e.g. to compile this manual or to convert the ***.eps** output from **gnuplot** to proper ***.pdf** figures.

6.2. 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

```
cd ~/nsCouette
cp -r ../nscouette/tutorials/tc0040 .
cd tc0040
vi nsCouette.in
```

and brows the parameter input file (**nsCouette.in**) using your favourite text editor (e.g. **vi**, **kate**, **gedit** and such). 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
/
```

while all the other parameters in this **namelist** are irrelevant for our first tutorial. They will be discussed later in § **??** when we turn to thermal convection. 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, as can be seen here

```
&parameters_grid
m_r  = 16 ! M radial 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. Also have a look at [7] around page 3 for further details, definitions and notations regarding the spatial discretisation scheme of **nsCouette**.

We want the simulation to run for 60 000 time steps; and the size of the computational time step Δt should be dynamically adapted to automatically guarantee numerical stability during integration. Both can be controlled using the parameter keywords

```
&parameters_timestep
numsteps = 60000 ! Number of time steps
variable_dt = T  ! Use variable (T) or fixed (F) time step size
/
```

in the relevant **namelist**. Since integrating the Navier-Stokes equations forward in time is a typical initial value problem, one last but very important issue to discuss before we finally run our first DNS, is choosing the initial conditions 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 input file

```
&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)
ic_pert = T ! Add perturbation on top of base flow (T) or not (F)
ic_p(1, :) = 1.0d-0, 0, 1 ! 1st perturbation: amplitude and wavevector
ic_p(2, :) = 1.0d-0, 1, 0 ! 2nd perturbation: amplitude and wavevector
/
```

we choose to start our simulation from scratch and set a resting fluid ($u_r = u_\theta = u_z = 0$) as initial state ($t = 0$) for our simulation. 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. Up to $l = 6$ independent perturbations can be superimposed by specifying the tuple $(a_l, k_{\theta,l}, k_{z,l})$, representing the respective perturbation amplitude and wave vector. When thinking of a real world laboratory set-up, perturbations could model residual motions from filling the tank, mechanical vibrations, small density or temperature gradients and such.

If you have already compiled the code as described in § 2, you now simply have to copy the executable to your case directory

```
cp ../nscouette/darkstar/nsCouette.x .
./nsCouette.x < nsCouette.in
```

and start the simulation by prompting the executable and passing the parameter file to the standard input. See also § 3 for further details and different ways to start a simulation. Now **nsCouette** runs on one core 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 31.63 s). The few last lines of the log should look something like this:

```
  step= 59993  dt=   1.8704145521610013E-005
  step= 59994  dt=   1.8704145521610013E-005
  step= 59995  dt=   1.8704145521610013E-005
  step= 59996  dt=   1.8704145521610013E-005
  step= 59997  dt=   1.8704145521610013E-005
  step= 59998  dt=   1.8704145521610013E-005
  step= 59999  dt=   1.8704145521610013E-005
 written hdf5/xdmf files to disk: fields_tc0040_00060000.{h5,xmf}
  step= 60000  dt=   1.8704145521610013E-005
 written coeff file to disk: coeff_tc0040.00060000
 written coeff file to disk: coeff_tc0040.00060000
T0: number of timesteps  : 60000
T0: total time incl I/O [s]:  31.63
T0:  av time per timestep excl I/O [s]:   0.00
ALL: min/max av time per timestep [s]: 0.0005 0.0005
feldmann@darkstar:~/nsCouette/tc0040$
```

The log tells us, that **nsCouette** wrote instantaneous state files (**fields*.h5**, **fields*.xmf**, and **coeff***) to disk, as described in § 3.4. Listing the content of our case directory by typing

```
ls

coeff_tc0040.00020000   coeff_tc0040.00060000   ke_mode  probe01.dat probe05.dat
coeff_tc0040.00020000.info coeff_tc0040.00060000.info ke_th probe02.dat probe06.dat
coeff_tc0040.00040000   fields_tc0040_00060000.h5  ke_total probe03.dat torque
coeff_tc0040.00040000.info fields_tc0040_00060000.xmf ke_z  probe04.dat
```

reveals, that next to the three Fourier coefficient files and the one physical flow field file there are also some time series data files containing temporal information about the kinetic energy (**ke_***), the velocity at six individual probe locations (**probe*.dat**), and the Nusselt number (**torque**).

Let's have a look at the torque data first, which is an integral system quantity. By typing something like

```
head -n 5 torque
  1.1100000000000E-004  5.7398470062833E+001  3.2245110074931E-012
  2.2200000000000E-004  3.5662943342695E+001 -1.1665951907500E-010
  3.3300000000000E-004  2.6254965224438E+001 -2.7358305823309E-010
  4.4400000000000E-004  2.1766851281236E+001 -2.4695918809171E-010
  5.5500000000000E-004  1.9296460086638E+001  7.9074775088634E-011
```

or

```
tail -n 5 torque
  1.1190369601483E+000  1.0000254300784E+000  9.9996854691748E-001
  1.1192240016035E+000  1.0000253831213E+000  9.9996860501287E-001
  1.1194110430587E+000  1.0000253362363E+000  9.9996866300095E-001
  1.1195980845140E+000  1.0000252894380E+000  9.9996872088193E-001
  1.1197851259692E+000  1.0000252427244E+000  9.9996877865598E-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 at $t = 0$ we have specified, up to the very end of this simulation run at around $t \approx 1$. 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. 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 and the Nusselt number is defined as the actual torque normalised by the torque of the laminar flow state (See §**??** and 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 6 by simply typing

```
gnuplot torque.gpl
okular torque.pdf &
```

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

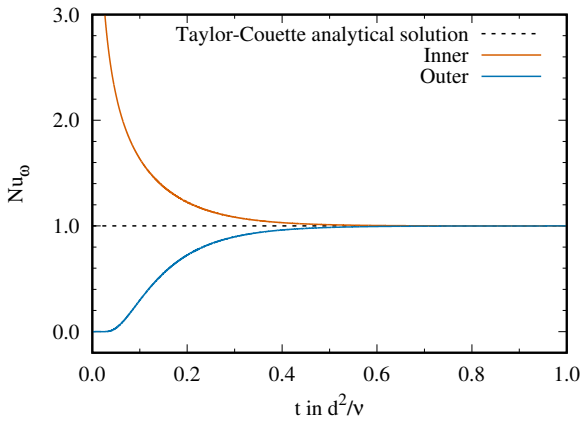


Figure 6: Temporal evolution of the torque at the inner and outer cylinder wall in our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. The torque is expressed in a non-dimensional way using a type of a Nusselt number, which relates the actual torque to the torque of the laminar Taylor-Couette analytical solution.

tonically 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 and take effect everywhere in the flow domain.

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

```
gnuplot probes.gpl
okular probes.pdf &
```

we generate a plot as shown in figure 7. We can easily see that the streamwise (azimuthal)

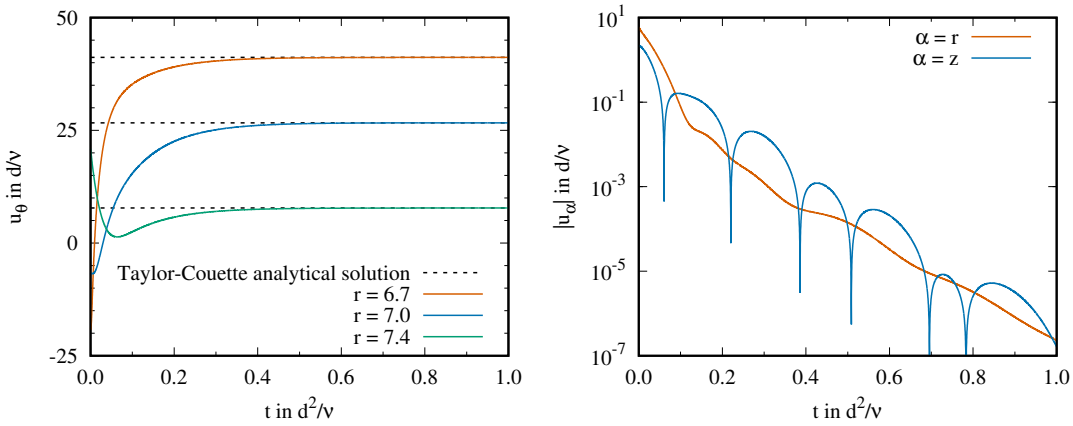


Figure 7: Temporal evolution of the velocity components at individual probe locations in our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. The streamwise velocity component (u_θ) is shown at three different radial locations ($r_i \leqslant r \leqslant r_o$) and converges everywhere to 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 ($r = 7.0$).

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 not survive in the linearly sable regime and the analytical solution for laminar Taylor-Couette flow is nicely reproduced! 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 [7] on page 5 and in [3] around page 424. The particular radial probe locations for which the analytical solution is needed to produce figure 7, can be found in the header of the **`probe0*.dat`** files by typing

```
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-005 6.7327503775987E-003  2.3139225064339E-001 -5.1548270462764E-002
2.1090000000000E-004 7.5859964050830E-003  3.1463670574244E-001 -5.1479743198774E-002
3.2190000000000E-004 8.2214409624340E-003  2.3121209404937E-001 -5.1579311466462E-002
4.3290000000000E-004 8.5819398548364E-003  1.0309282864180E-001 -5.1781194327691E-002
5.4390000000000E-004 8.7948219702162E-003 -2.3699549021819E-002 -5.2001676668548E-002
```

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

```
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 (see § 3.5) or rerun the simulation.

Since we have only looked at particular points in the flow field so far, figure 8 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 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 respective **`namelist`** in the parameter file. The colour-coding in figure 8 represents the only non-zero velocity component

$$\mathbf{u} = \begin{bmatrix} u_r \\ u_\theta \\ u_z \end{bmatrix} = \begin{bmatrix} 0 \\ f(r) \\ 0 \end{bmatrix}, \tag{8}$$

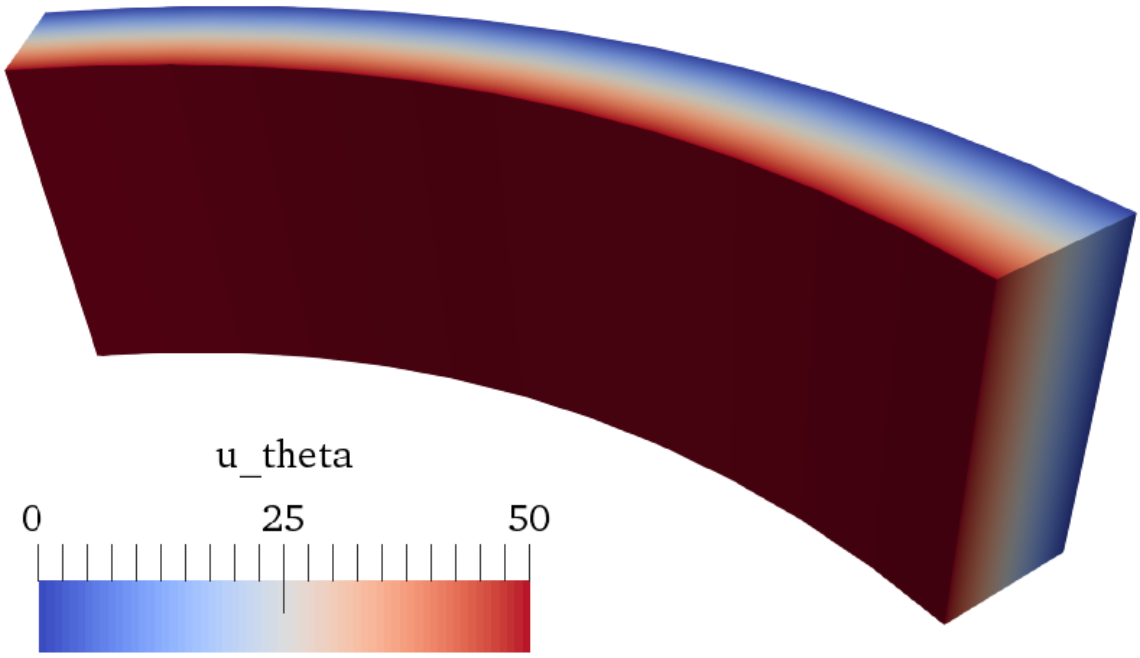


Figure 8: The depicted volume represents the entire computational domain ($(6.576 \leqslant r \leqslant 7.576) \times (0 \leqslant \theta \leqslant {}^{2\pi}/_{6}) \times (0 \leqslant z \leqslant 2.4)$) we have chosen for our first DNS; i.e. only one sixth of a full cylinder with a very small axial height. 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.

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 $50\,{}^{d}/_{\nu}$, 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 § 4.1 or § 4.2, in case you prefer the software **`VisIt`**.

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 time step. 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

```
gnuplot keThZ.gpl
okular keThZ.pdf &
```

we generate a plot as shown in figure 9. Since there is no relevant energy content in any of

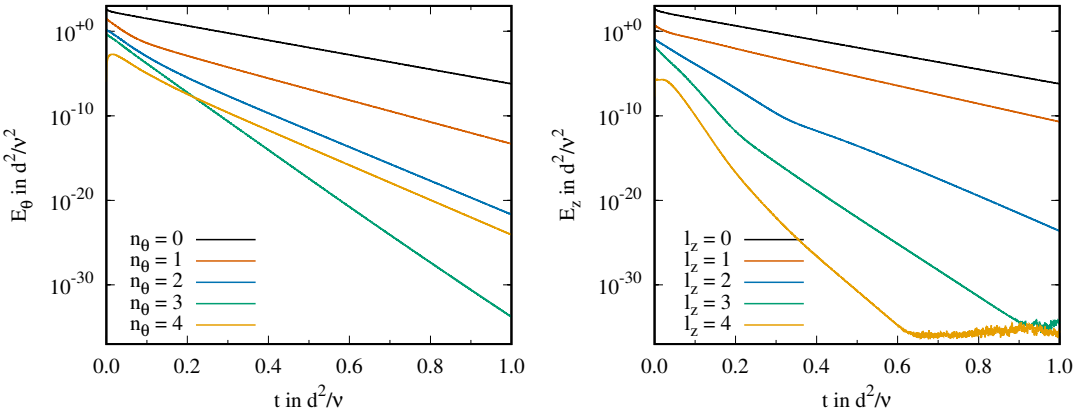


Figure 9: Temporal evolution of the azimuthally (left) and axially (right) dependent kinetic energy contained in each discrete azimuthal (n_θ) and axial (l_z) mode for our first Taylor-Couette tutorial **`tc0040`** with $Re_i = 50$. See also page 3 in [7] for definitions and details on the spatial discretisation 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.

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 (see §**??**) the simulation and let it run for another 60 000 or more time steps. Although the dynamics at this 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 by setting . . . to larger values. . .)

Additionally, the highest wave numbers represent the spatial resolution of the simulation. They show numerical noise, no relevant energy content. Good indicator, that the spatial resolution is sufficiently fine. . .

6.3. Laminar Taylor-vortex flow in the unstable regime

In a next step we want to increase the Reynolds number to the unstable regime and see whether **`nsCouette`** reproduces the well known Taylor-vortex state. Go back to your working directory, copy the second Taylor-Couette tutorial case from the repository

```
cd ~/nsCouette
cp -r ../nscouette/tutorials/tc0041 .
cd tc0041
vi nsCouette.in
```

and inspect the parameter file (**nsCouette.in**) using your favourite text editor. We triplicate the rotation speed of the inner cylinder by setting

```
&parameters_physics
Re_i = 150.0d0 ! inner cylinder Reynolds number
Re_o =   0.0d0 ! outer cylinder Reynolds number
/
```

in the respective **namelist**. Since we now expect a slightly more complex flow field, we also have to increase the spatial resolution slightly by setting

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

to account for the spatial velocity gradients related to the pronounced vortex in the flow field. This choice corresponds to a total of $n_r \times n_\theta \times n_z = M \times (2N+1) \times (2L+1) = 16 \times 17 \times 17$ significant physical grid points. The finer spatial resolution naturally demands a smaller computational time step Δt. Since Δt will be adapted automatically to ensure numerical stability during integration, we simply have to increase the total number of time steps by setting

```
&parameters_timestep
numsteps = 1000000 ! Number of time steps
variable_dt = T ! Use variable (T) or fixed (F) time step size
/
```

in the relevant **namelist**, to end up at roughly the same physical time t as in our first test case. Both changes will surely increase the overall computational time necessary for our second DNS. To save resources and to keep the tutorials quick, we now chose the analytical Taylor-Couette velocity profile as initial condition by setting

6.4. Laminar wavy vortex flow in the unstable regime

6.5. 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 [**?**]. 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 [5]. 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 [5]. 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.

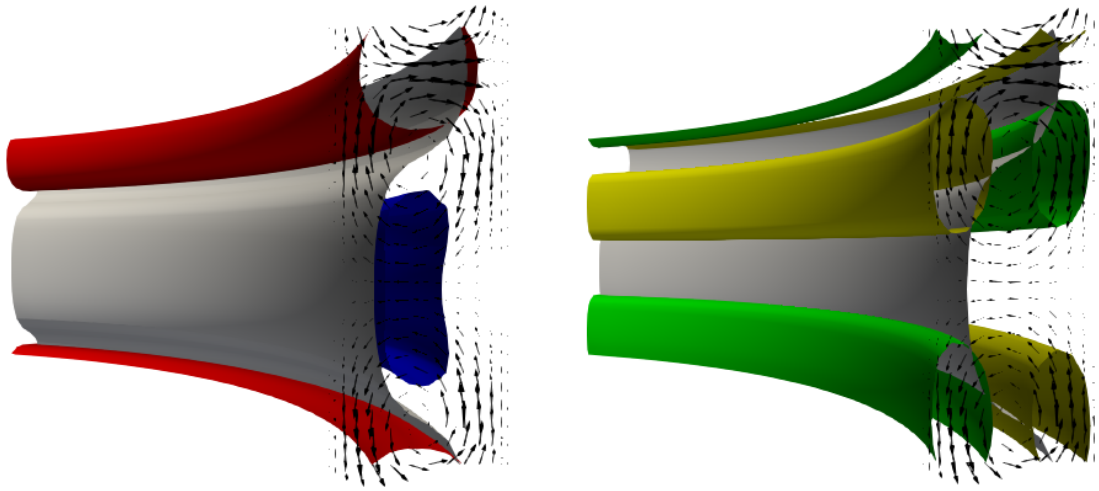


Figure 10: Three-dimensional flow field visualisation using **ParaView**. Shown are iso-contours (red/blue: $u_r = \pm 4$, grey: $u_\theta = 75$, yellow/green: $u_z = \pm 4$) and a vector representation of the cross-stream velocity components. Stationary Taylor-vortex flow state at $Re_i = 150$.

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 § 2. 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 14 by simply typing

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

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. Got 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 to 4000 and that we have also increased the axial resolution, since we expect a slightly more complex flow

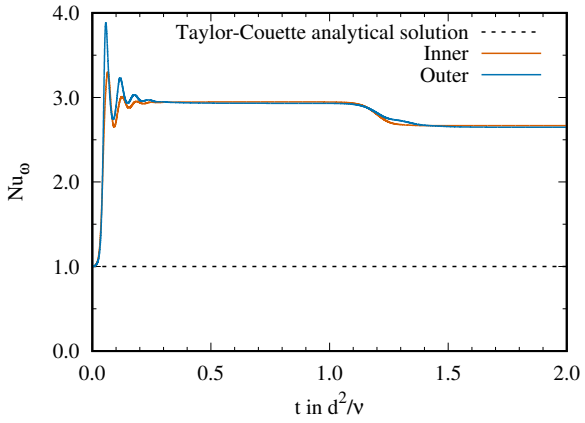


Figure 11: Temporal evolution of the torque at the inner and outer cylinder wall in our third Taylor-Couette tutorial **`tc0042`** with $Re_i = 458.1$. The torque is expressed in a non-dimensional way using a type of a Nusselt number, which relates the actual torque to the torque of the laminar Taylor-Couette analytical solution.

state. Moreover, we now want to run the simulation for another 400 000 time steps (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 time steps
/
&parameters_output
...
fBase_ic = 'tc0075' ! identifier for coeff_ (checkpoint) and fields_ (hdf5) files
/
&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

```
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
```

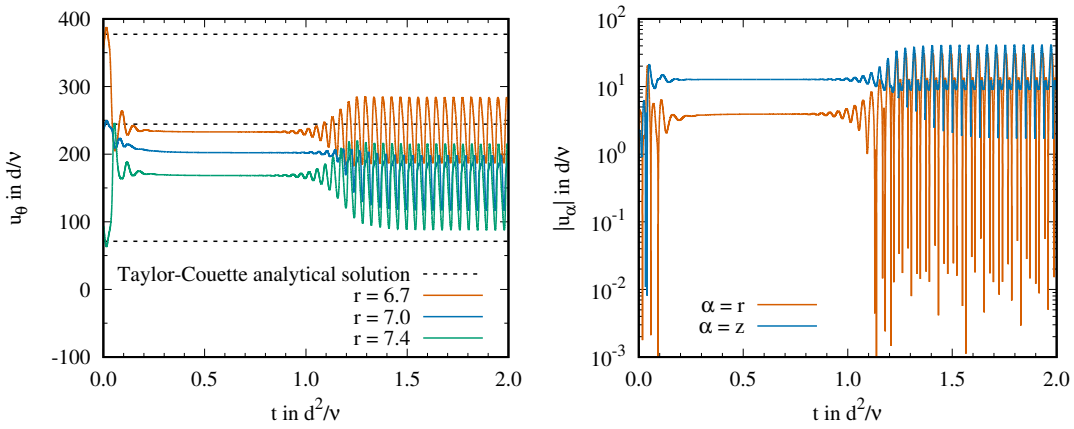


Figure 12: Temporal evolution of the velocity components at individual probe locations in our third Taylor-Couette tutorial **`tc0042`** with $\mathrm{Re}_i = 458.1$. The streamwise velocity component (u_θ) is shown at three different radial locations ($r_i \leqslant r \leqslant r_o$) and converges everywhere 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 ($r = 7.0$).

```
/
&parameters_info
...
/
```

using your favourite text editor.

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 § 3.1. 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 § 5.5 and in the tutorials § 6.5. 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 § 5.4.

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 [7]. 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
```

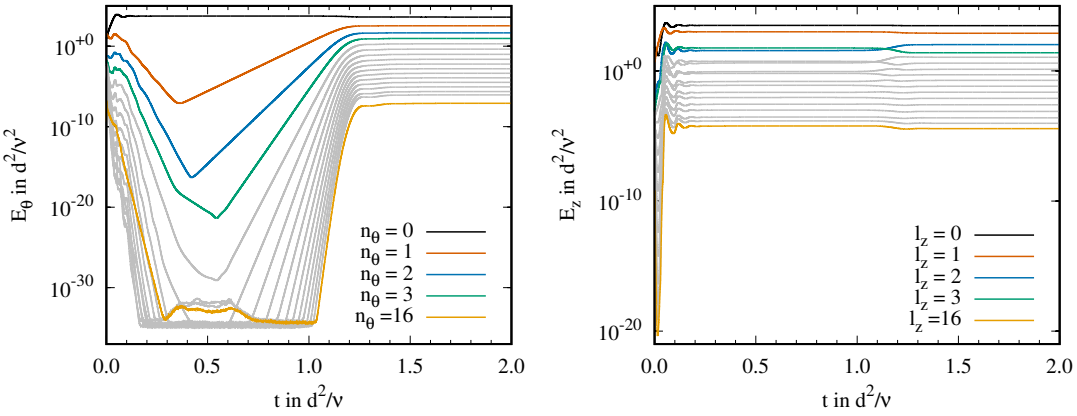


Figure 13: Temporal evolution of the azimuthally (left) and axially (right) dependent kinetic energy contained in each discrete azimuthal (n_θ) and axial (l_z) mode for our third Taylor-Couette tutorial **`tc0042`** with $Re_i = 458.1$. See also page 3 in [7] for definitions and details on the spatial 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.

```
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
```

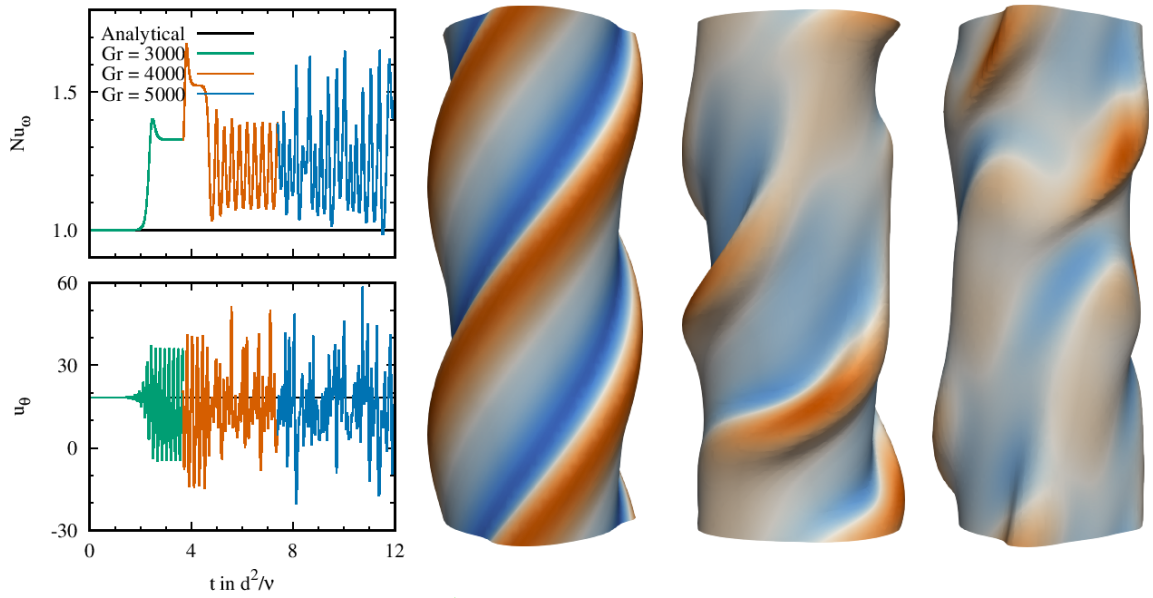


Figure 14: Different flow states for increasing Grass number when restarted the simulation from a former run. Temporal evolution of the torque at the inner cylinder wall and the streamwise (azimuthal) velocity component at one mid-gap probe location. Iso-surfaces of the mean Temperature (T = 0) colour-coded by the inwards (blue) and outwards (r) directed wall-normal (radial) velocity component.

```
!--- 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 time steps
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 § 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, e.g. ~/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, e.g. 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. E.g. 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 § B.1

```
#
# Download and unpack to some directory of your choice, e.g. ~/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, e.g. 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. E.g. 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. 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] 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.

[6] K. Moreland. The ParaView Tutorial, 2018.

[7] 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.

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