A user guide for nsCouette

Liang Shi, Markus Rampp, Jose M. Lopez, Björn Hof & Marc Avila

October 9, 2018

The code `nsCouette` is written in Fortran90 and designed to simulate incompressible Taylor-Couette flows with infinitely long cylinders (periodic in the axial direction). It has recently been extended to solve temperature stratified Taylor–Couette flows (see § 4 for details). For the methodology, implementation and validation, please refer to our paper [Computer & Fluids 106 (2015) 1-11, hereafter referred to as CAF paper]. This guide can be used to get familiar with the prerequisites, compilation, running and the structure of the code before using it.

1 Building and running the code

Let's get off the ground by running an example of wavy-vortex flow (Figure 5 in CAF paper).

1.1 Prerequisites

1) Compiler: a modern Fortran90 compiler which is OpenMP 3 compliant and additionally a corresponding C-compiler (tested with Intel ifort/icc 12...15, GCC 4.7...4.9, PGI 14)

   for free software see: https://gcc.gnu.org/

2) MPI: an MPI (Message Passing Interface) library which supports thread-level `MPI_THREAD_SERIALIZED`. A fallback for the minimum thread-level which is supported by any MPI implementation is implemented. (tested with Intel MPI 4.1,5.0,5.1, IBM PE 1.3,1.5)

   for free software see: http://www.open-mpi.de/

3) BLAS/LAPACK: a serial BLAS/LAPACK library (tested with Intel MKL 11.x)

   for free software see: http://www.openblas.net/,

   http://math-atlas.sourceforge.net/,

   http://www.netlib.org/

4) FFTW: a serial (but fully thread-safe!) FFTW3 installation or equivalent (tested with FFTW 3.3, MKL 11.[1-3], earlier MKL versions will likely fail)

   for free software see: http://www.fftw.org

5) HDF5 (optional): an MPI-parallel HDF5 library installation (tested with HDF5 1.8.x)

   for free software see: http://www.hdfgroup.org/HDF5/

1.2 How to compile the code

In the top-level directory, there are the following files

```
ARCH/            doc/
mod_fdInit.f90   mod_inOut.f90   mod_timeStep.f90
getcpu.c         mod_fftw.f90    mod_myMpi.f90      mod_vars.f90
input_nsCouette  mod_getcpu.f90  mod_nonlinear.f90  nsCouette.f90
Makefile         mod_hdf5io.f90  mod_params.f90     perfdummy.f90
   ...
```

Before compiling the program, check and change a few things first.

1.2.1 Makefile settings

1. `Makefile` needs no editing by the user
2. platform-specific settings are specified in `ARCH/make.arch.<architecture>`. Some working templates like `make.arch.intel-mkl`, `make.arch.gcc-mkl`, `make.arch.pgi-mkl`, `make.arch.gcc-linux` are available in the `ARCH` subdirectory
3. copy the template which appears closest to the target platform, e.g.

   `cp ARCH/make.arch.intel-mkl ARCH/make.arch.myplatform`

   then edit `ARCH/make.arch.myplatform`, and build the code with:

   `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.

1.2.2 Setup

Starting with `nsCouette 1.0` the file `mod_params.f90` needs no more editing and all settings can be made in the input file (see below). For reference we print here `mod_params.f90` from pre-1.0 versions in order to document the geometrical and numerical parameters that were used for the example of figure 5 in our CAF paper.

```
INTEGER(KIND=4),PRIVATE   :: ir
!-----------------------------------Mathematics 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 parametres
INTEGER(KIND=4),PARAMETER :: m_r  = 32            ! Maximum spectral mode (> n_s-1)
INTEGER(KIND=4),PARAMETER :: m_th = 16
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 ! 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     ! points in fourier dirs
  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          ! 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-distributed nodes
  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


!-----------------------------------Miscellaneous-------------------
  REAL(KIND=8), PARAMETER :: d_implicit = 0.51d0 !implicitness
  REAL(KIND=8), PARAMETER :: tolerance_dterr = 5d-5

!----------------------defaults for runtime parameters (can be set in input file)

  REAL(KIND=8)   :: Courant = 0.25d0
  INTEGER(KIND=4) :: print_time_screen = 250
  LOGICAL        :: variable_dt= .true.
  REAL(kind=8) :: maxdt = 0.01d0

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

!----------------------defaults for runtime parameters (can be set in input file)
  REAL(KIND=8)    :: dt = 1.0d-7          ! Time step (default value)
```

```
INTEGER(KIND=4) :: numsteps = 100000    ! Number of steps (default value)
INTEGER(KIND=4) :: dn_coeff = 5000      ! coeff per dn_coeff steps (default value)
INTEGER(KIND=4) :: dn_ke = 100          ! energy per dn_ke steps (default value)
INTEGER(KIND=4) :: dn_vel = 100         ! velocity per dn_vel steps (default value)
INTEGER(KIND=4) :: dn_Nu = 100          ! Nusselt per dn_Nu steps (default value)
INTEGER(KIND=4) :: dn_hdf5 = 1000       ! HDF5 output per dn_hdf5 steps (default value)
```

The variables are compatible with the symbols in our CAF paper.

Compilation is detailed in Sect. 1.2.1. Type

```
make (ARCH=...) clean
```

to clean the compilation. Inside the brackets are optional. Before each compilation, it is recommanded to `make clean`.

If everything is configurated correctly (including some environmental variables like `LD_LIBRARY_PATH` in some clusters. Please consult your local IT support in case of `LD_LIBRARY_PATH` problems), the compilation generates a new subdirectory (named after the ARCH= setting), including all the `.o` and `.mod` files as well as the executable file `nsCouette.x`.

1.3 Running the code

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
k_z0  = 2.6179938779914944 ! axial wavenumber       => L_z =  2*pi/k_z0  axial length of grid
eta   = 0.868d0           ! aspect ratio => r_i = eta/(1-eta),r_o = 1/(1-eta) inner/outer rad
/

&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 ouput)
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
/
```

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
```

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 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
[...]
```

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 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.

1.4 Output & visualisation

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.

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. 1.5). They can not be visualized straightforwardly but by some user-written post-processing utilities (*e.g.*, using the library `plplot`). 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.

1.4.1 Visualization with VisIt

To visualize with the open source software `VisIt`, do the following steps:

- Install the software from the webpage `https://wci.llnl.gov/simulation/computer-codes/visit`;
- Move the files `.h5` and `.xmf` to one directory `DIR_HDF5`;
- Open `VisIt` and a GUI window `File open` will pop up.
- Enter the path where the `.h5` and `.xmf` files are located in the `path` line; You can see the directories and files in the `Directories` and `Files` windows;
- Select the `.xmf` files and click on "OK" button;
- Then you have two GUI windows: VisIt Main window (the "control panel" one) and Vis window (the white blank one);
- Click on "Add" button in the middle of the Main window and you have a pull-down menu. Select the type 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. 1);
- 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;

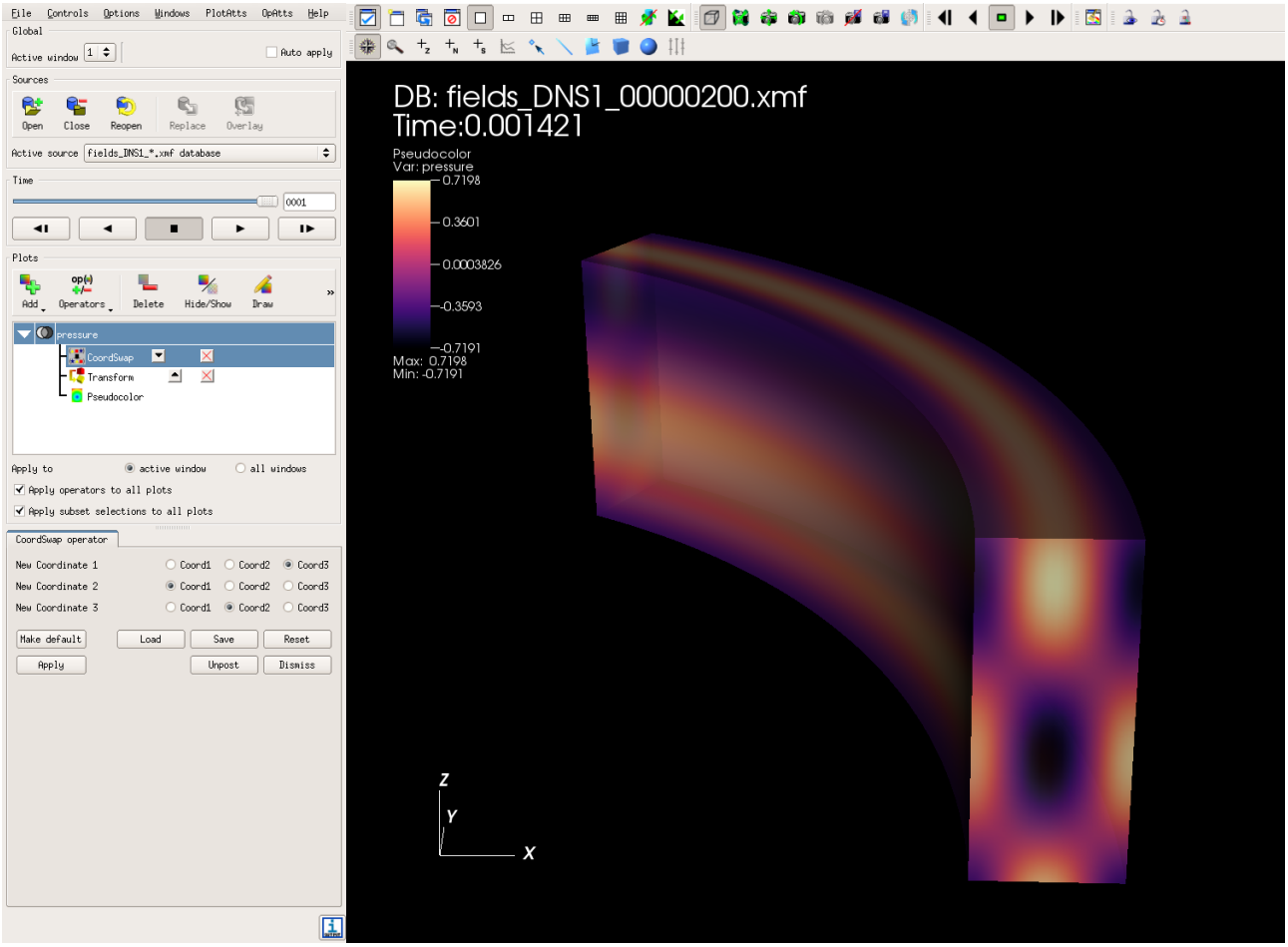

Figure 1. Screenshot of the VisIt GUI

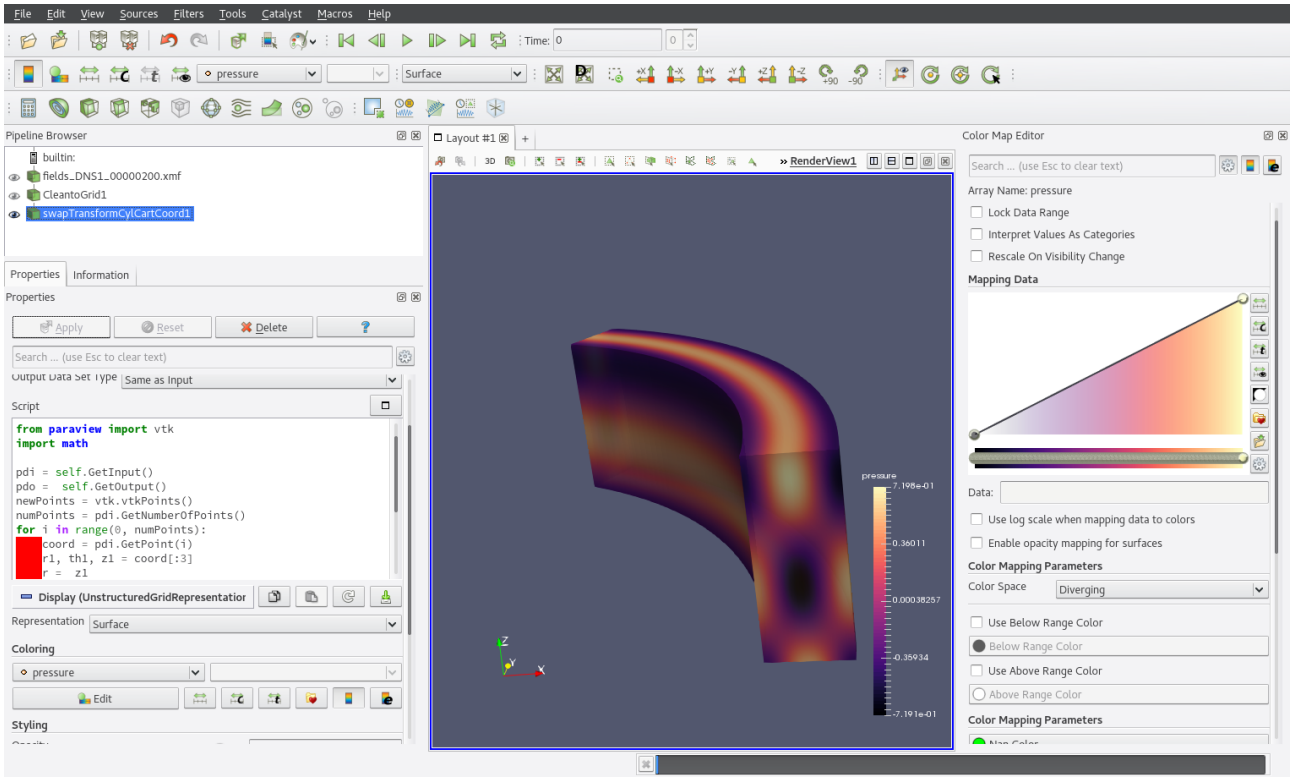

Figure 2. Screenshot of the ParaView GUI

1.4.2 Visualization with ParaView

To visualize with the open source software `ParaView`, do the following steps:

- install ParaView from the package manager or download and install from `https://www.paraview.org`
- to load the filter for performing the transformation of cordinates $(\theta, z, r) \to (r, \theta, z)$ in paraview do: Tools → Manage custom filters → Import → (select the `swapTransformCylCartCoord.cpd` file in subdirectory `visualization/`)
- to apply the filter do:
  1. Filters → Alphabetical → Clean to Grid
  2. Filters → Alphabetical → swapTransformCylCartCoord

When applying the filters make sure that they are in correct order, as shown in the panel at the upper left in the paraview GUI (cf. Fig. 2)

Note that this is just a collection of very basic quick-start recipes. For more information, please consult the documentation of the corresponding software packages.

1.5 Continuing a run: checkpoint-restart mechanism

On large HPC clusters the runtime of a batch job on HPC machines is usually limited, typically to something like 24 hours. In order to enable long-running simulations, nsCouette implements a semi-automatic checkpoint-restart mechanism. At the end of a run, a coefficient file is written as

a checkpoint, together with a small text file named `restart` which contains additional information that is required to continue a run. 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 is that of the initial condition and the 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.

2 About the code

In this section, the code will be further documented in the programming level, especially on the data type and structure.

2.1 The structure of the code

The code contains a minimum set of 12 files including `Makefile` and about a few thousand code lines. Basically speaking, the code has an initialisation phase (MPI, initial conditions or read-in), time-step computation (linear substep computation and nonlinear pseudo-computation), finalizing phase (output and `MPI_finalize`). The contents of each file are summarized as follows:

```
Makefile            The script file to compile the code
nsCouette.f90       Main routine including initialization and finalization
mod_params.f90      All constants of the code
mod_vars.f90        Definition of all (derived) variables
mod_myMpi.f90       MPI-related subroutines: windows, derived MPI data type,etc
mod_fftw.f90        FFTW-related subroutines
mod_fdInit.f90      Subroutines for the differentiation matrices
                    (1st and 2nd radial derivatives)
mod_nonlinear.f90   Subroutine to compute the nonlinear advection term
                    (pseudo-spectral)
mod_timeStep.f90    Subroutines to timestep the equations
mod_inOut.f90       Read-in, input, output subroutines, also base flow function
mod_hdf5io.f90      HDF5 output subroutines
mod_getcpu.f90      Mapping of the cpu. This module can not be included.
```

2.2 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 sub-domain. The type `phys` and `spec` contains all physically relevant quantities, **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.

3 Time-stepper: predictor-corrector method

Since the publication of CAF some changes have been implemented in the code to further improve its performance. Notable among these is the implementation of a new time-stepper that allows to significantly increase the timestep size in the computations. The idea for this time-stepper was borrowed from the openpipeflow project, an open source code to compute pipe flow, to whose documentation the reader is referred for a comprehensive description. The most salient novelties are summarized in what follows

- The equations are integrated by using a predictor-corrector algorithm.

- 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

$$\begin{aligned} \nabla P_1^{q+1} &= \nabla \cdot (Lu^q + N^q), \\ (\frac{1}{\delta t} - c\nabla^2)u_1^{q+1} &= N^q + (\frac{1}{\delta t} - (1-c)\nabla^2)u^q - \nabla P_1^{q+1}, \end{aligned} \tag{1}$$

- The velocity computed in the predictor step u_1^{q+1} is then refined in the corrector step up to a certain user-specified tolerance (tolerance_dterr). The iterations for the corrector step are given by

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

- 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

$$\delta t = Cmin(\nabla/|v|), \tag{3}$$

  where C is the Courant number

4 Specificities of nsCouette with temperature

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 outputs. The `Nusselt` file contains time series for the Nusselt number at the inner and outer cylinders, whereas the file `temp_mid` includes the temporal variation of the temperature at the same location where the `vel_mid` file is generated.

References

- Liang Shi, Markus Rampp, Björn Hof and Marc Avila, A hybrid MPI-OpenMP parallel implementation for pseudospectral simulations with application to Taylor-Couette flow. Computers & Fluids 106 (2015) 1-11. (arXiv:1311.2481)
- Jose M. Lopez, Francisco Marques and Marc Avila, The Boussinesq approximation in rapidly rotating flows. Journal of Fluid Mechanics,Volume 73, December 2013, pp. 56-77.