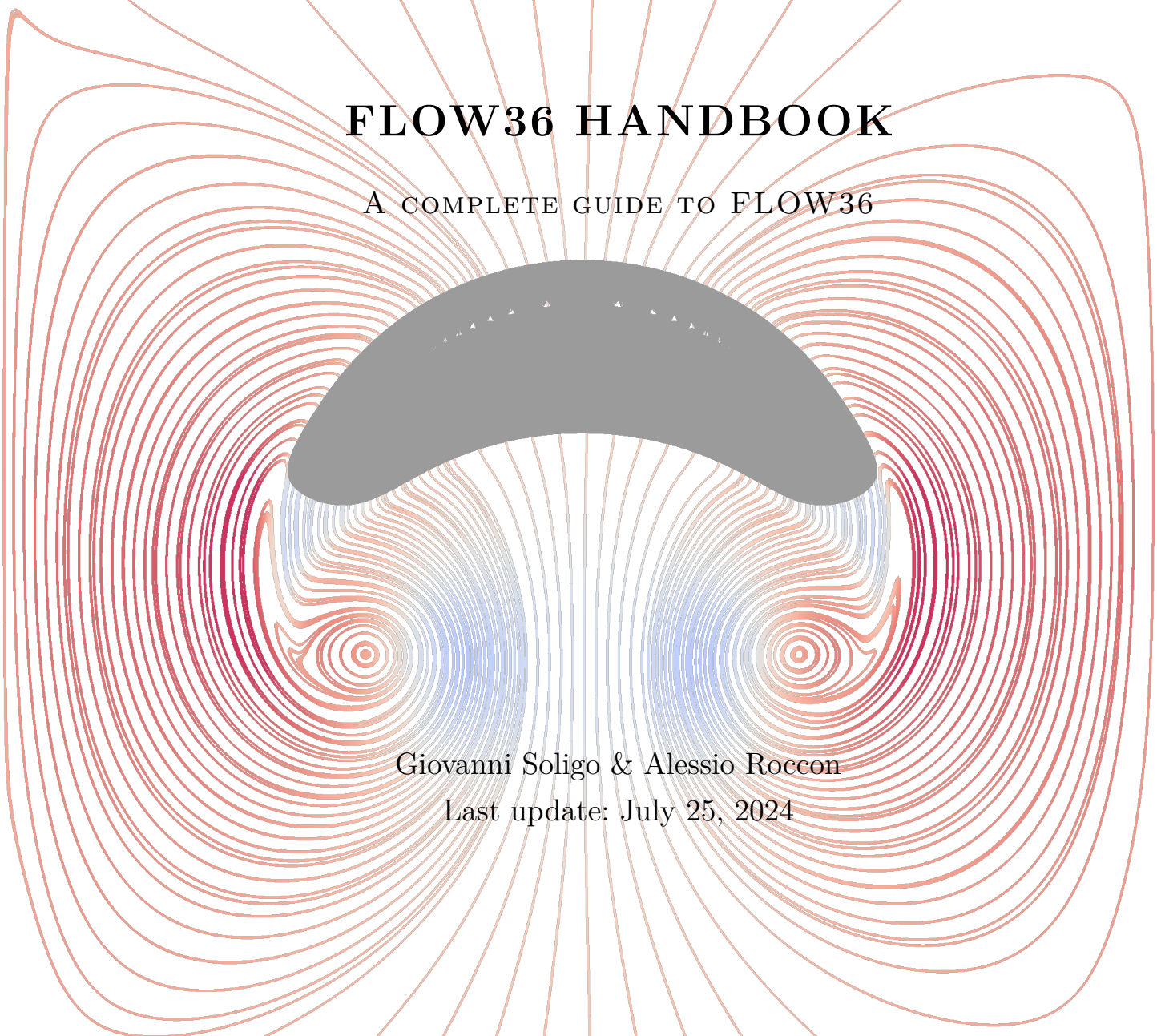
FLOW36 HANDBOOK

A complete guide to FLOW36



Giovanni Soligo & Alessio Roccon

Last update: July 25, 2024

This handbook is intended to give a general, but very detailed description of the code `FLOW36`. The code was initially written in 2017 at TU Wien by Giovanni Soligo & Alessio Roccon, as a replacement for the legacy code `FLOWSB`, in order to get better overall performance, a more readable and up to date code and improve its scalability. The code was then ported to GPU architectures during the summer 2022. The code performs Direct Numerical Simulations (DNS) in a channel geometry, using a pseudo-spectral spatial discretization. The flow solver is coupled with a phase-field method, which relies on the Cahn–Hilliard equation (phase-field method). The solver can also accounts for the presence of surfactants (in the phase-field method framework) as well as describe a passive scalar (heat transfer problems). In addition, a Lagrangian particle tracking algorithm is implemented to track the motion of inertial spherical sub-Kolmogorov particles.

This document was last updated on July 25, 2024.

Contents

Chapter 1

Getting started

The bash `compile.sh` includes all the parameters needed to run a simulation. In Section 1.3 is reported a detailed guide about this file, its structure and how to modify it.
When running the `compile.sh` in its own folder, it sets up all the files and folders needed for the simulation in the `set_run` folder. Once the `compile.sh` has finished without error, go to the `set_run` folder and launch the executable `go.sh`; at this point the simulation should start.
The main folder of the code includes the following files and subfolders:

- `initial_fields` : this folder contains the initial fields (velocity, phase-field, surfactant, temperature, particles position and velocity) used if the initial condition is set to read the initial fields (both in its serial or parallel version). These files, during the execution of `compile.sh`, are copied into the `set_run/initial_fields` folder, so they can be removed or modified even when the simulation is running.
- `Machine XYZ` : contains the `makefile` and *go.sh* needed when running on the above mentioned machine. It is copied in the main folder by the `compile.sh` script when needed. For some machines, more than one makefile (i.e. configuration) might be available depending on the compiler/libraries available and architecture (CPU, GPU). Depending on the machine, also the `openacc_flag` is set. This flag enables the GPU acceleration on the supported machines (e.g. Nvidia GPUs).
- `paraview_output_fg` : contains the code that can be used to generate Paraview compatible output file (using a rectilinear grid).
- `set_run` : contains all the files and subfolders needed for a simulation and its output.
- `source_code` : contains all the subroutine and the main file used to compile the executable file; each subroutine will be described in detail in Chapter 2.
- `stats_calc` : contains the code that can be used to extract velocity statistics from the simulation output files (mean, root mean square, skewness and flatness). These statistics can also be calculated runtime as will be seen in Section 1.3.
- `compile.sh` : this file is used to generate the simulation folder and executable. It also includes all the parameter declaration part.
- `go.sh` : its edited version is copied during `compile.sh` execution in the `set_run` folder and used to launch the simulation (directly on local machines or to be submitted to the job/load manager via batch commands).

- `input.f90` : its edited version is copied during `compile.sh` execution in the `set_run/sc_compiled` folder and it is used as an input file for the simulation parameters.

- `makefile` : called during `compile.sh` execution to create the executable of the code.

- `scaling` : contains strong and weak scaling results obtained on different machines.

- `profiling` : contains the profiling data obtained on Marconi-100 using GPU. The file can be view using Nvidia Nsight Systems.

On a local machine the code can be compiled and run just by executing the `compile.sh` script. On a cluster, since there is always a job scheduler that handles all the submitted jobs the last lines of the `compile.sh` script must be commented out (especially the call to the script `go.sh`). When running on a cluster, first run the `compile.sh` script, then move to the `set_run` directory and submit the jobscript `go.sh`. If you want to compile and run several simulation with different parameters, after the compilation, copy the folder `set_run` somewhere else and then submit the jobscribt to the job scheduler there.

1.1 Output of a simulation

Depending on the parameters choice when compiling the code, the code can give as an output different data, that will be all saved in the subfolder `results` inside the `set_run` folder.
The code will always save the initial and final velocity fields (and the phase field, surfactant, temperature, particles position and velocity if they are activated) both in physical and modal space, the x, y, z axis arrays and a time check file. This latter file will include the simulation current time, the bulk Reynolds number and, for the phase field case only, also the mean value of ϕ all over the domain and the integral of the phase $\phi = +1$ (to check the mass losses). Additional parameters can be included in this file depending on the modules activated.
In addition the output of the simulation includes:

- Flow field data (and other variables, if activated) in physical space: `[variable name]_[number of time step].dat`

- Flow field data (and other variables, if activated) in modal space: `[variable name]c_[number of time step].dat`

- Mean, root mean square, skewness and flatness for the flow field (u, v, w) (single array in the wall-normal direction). Single formatted file `stats.dat`.

- Mean pressure and root mean square of pressure fluctuations (single array in the wall-normal direction). The mean pressure value does not include the mean pressure gradient in x and y directions. The pressure solver works for a fully-developed channel flow with a non-zero mean velocity in the x direction and for a single phase flow. Single formatted file `budget.dat`; the first lines of the file explain its content.

- Energy budgets. For the pressure–strain correlation the pressure calculated above is used, so they do not consider the presence of multiple phases (the energy balance may not be zero for a multiphase flow, since some terms are missing/not properly calculated). The energy budgets are saved in the same file as the pressure (`budget.dat`).

- Streamwise and spanwise power spectra for u', v' and w' at $z^+ = 5$, $z^+ = 15$ and $z^+ = \mathrm{Re}$. Power spectra in x direction are saved in the file `power_xspectra.dat`, while those in y direction in the file `power_yspectra.dat`.

Velocity and eventually other variables are saved in a binary file, written with the same endianness and format as the MPI implementation of the machine where the simulation was run.
In future others output can be added to the code.

1.2 Post-processing

At the present time two codes are available for direct post-processing of the output data; the first one, in the folder `stats_calc` evaluate the statistics of the flow field (can be done also at run-time), while the other, `paraview_output_fg`, generates Paraview compatible output file. Both these codes use either data in physical space, either in modal space.
The statistics calculation runs alway on three MPI processes, one handle u data, another v data and the third w data. The Paraview output generation can work independently on several cores: each core takes care of writing the output at a certain time-step. Use the input file to modify some parameters of the program to better handle the output (number of variables, output grid, etc.)

1.3 The `compile.sh` file

The script `compile.sh` is divided in several parts: input parameters declaration, cleaning of `set_run` folder, copying and editing files in the `set_run` folder, compilation of the code and, only on a local machine, running the `go.sh` in the proper folder.

1.3.1 Parameters declaration

When running a simulation this is the only part that should be modified; unless needed (e.g. code modification, ...) all the rest of the script should be left untouched.
Always pay attention to the parameter type (integer, real, double, ...) when editing values.

- `machine` : declare which machine is used for the simulation (local machine, Discoverer, VSC5, Leonardo ...). According to the machine chosen, the proper modules are loaded and (on a supercomputer) the proper batch scheduler instructions will be selected in the `go.sh` script. `openacc_flag`: This flag is automatically set depending on which machine is used. This flag enables to use of GPUs on supported machines (e.g. Nvidia GPUs). For some machines, there might be two machine numbers, one with and one without GPU acceleration (e.g. M100 and Leonardo).

- `fftw_flag` : can be 0 or 1; if 0 the plans for the Fourier and Chebyshev transforms will be created using the default algorithm. On the other hand, if 1 is selected, the plan creation will take much more time, but it will choose the optimal algorithm to perform the transforms. A value equal to 1 will results in a much higher time for the `FFTW` plan creation, but it should choose the most performing algorithm according to the size of the transforms and the machine where the simulation are run. If GPU-acceleration is used, cuFFT does not have this option and the default algorithm for plan creation is employed.

- `ix` : the number of points is always a power of two: the number of points in x direction is $\texttt{NX} = 2^{\texttt{ix}}$.

- `iy` : same as `ix`, but for the y direction: the number of points in y direction is $\texttt{NY} = 2^{\texttt{iy}}$.

- `iz` : number of points in z direction is expressed as $\texttt{NZ} = 2^{\texttt{iz}} + 1$, since Chebyshev transforms are faster on an odd number of points.

- `exp_x` : Expansion factor along x for the variables that can be resolved on the finer grid (only the surfactant at the moment, easy to extend to other variables, must take care of the coupling).

- `exp_y` : Expansion factor along y for the variables that can be resolved on the finer grid.

- `exp_z` : Expansion factor along z for the variables that can be resolved on the finer grid.

- `NYCPU` : number of division of the domain for parallelization (y direction in physical space, z direction in modal space). In physical space each MPI process holds roughly $N_z \times N_y/N_{y,cpu} \times N_z/N_{z,cpu}$ points (for the exact method please refer to Chapter 3). In modal space each MPI process holds roughly $N_x/N_{y,cpu} \times N_y/N_{z,cpu} \times N_z$. When running 2D simulation always run on a $x - y$ plane so the value of `NYCPU` must be set to 1.

- `NZCPU` : number of division of the domain for parallelization (z direction when in physical space, y direction when in modal space).

- `multinode`: if running on a single node or many nodes, default value is 0, this parameter has an effect only when the LPT is used.

- `nodesize`: size of the node (number of MPI tasks per node), this parameter is important only when the LPT is used. It is used to split among the N nodes that solve the Eulerian fields and the N+1 node that take care of the particles.

- `restart` : if equal to 0 the simulation is a new simulation, otherwise a previous simulation is restarted. When restarting a new simulation the code will automatically set the proper initial conditions for the flow field and for the phase field (if active). All the other parameters can be modified freely. The `restart` flag determines also which files will be kept and which deleted (please refer to Section 1.3.2 for a complete description).

- `nt_restart` : time step from which restarting the simulation; the code will thus read the corresponding flow (and phase) fields.

##################################
Navier-Stokes parameters
##################################

- `incond` : defines which is the initial condition of the simulation. The complete list of initial condition are reported in the `compile.sh` script. Some examples are:
  - zero velocity all over the domain
  - laminar Poiseuille flow in x direction (generated from a unitary pressure gradient)
  - random velocity value for u, v, w
  - read input from file (parallel read)
  - read input from file (serial read, used for retro-compatibility with legacy data files)
  - shear flow along the x or y directions
  - ...
- `Re` : Re number used for the simulation.
- `Co` : Courant number threshold value, if the Courant number exceeds this value the simulation is stopped.
- `gradpx` : mean pressure gradient along x direction, defined as $\overline{\frac{\partial P}{\partial x}}$.
- `gradpy` : mean pressure gradient along y direction, defined as $\overline{\frac{\partial P}{\partial y}}$.
- `cpi_flag` : if enabled (1), simulations are performed using the constant power input framework and pressure gradient is adapted to the flow-rate so to keep constant the power injected (supported only along the x direction and for 3D domain).
- `repow`: Power Reynolds number used to compute the pressure gradient (computed on the effective viscosity.
- `lx` : size of the domain (x direction) normalized by π.
- `ly` : size of the domain (y direction) normalized by π.
- `nstart` : initial time step of the simulation.
- `nend` : final time step of the simulation.
- `dump` : saving frequency of fields (velocity and eventually phase variable) in physical space. If a value of -1 is provided no fields data will be saved during the time cycle.
- `sdump` : saving frequency of fields (velocity and eventually phase variable) in modal space. If a value of -1s is provided no fields data will be saved during the time cycle.
- `failure_dump` : saving frequency of fields (in modal space). These files are not kept and they are meant to be used only as a checkpoint if the simulation stops. The saving frequency should be higher than the normal saving frequency.

- `st_dump` : calculation and saving frequency of flow statistics at run time.
- `stat_starts` : time step from which starting the statistics calculation.
- `mean_flag` : if equal to 0 the code does not calculate the mean, root mean square, skewness and flatness of the flow field at run time, otherwise if equal to 1 it will calculate these statistics with `st_dump` frequency.
- `budget_flag` : if equal to 0 the code will skip pressure statistics and energy budgets calculation; if equal to 1 these statistics will be calculated and saved.
- `spectra_flag` : if equal to 0 the code will not calculate any velocity power spectra, otherwise if equal to 1 it will calculate them.
- `dt` : value of the time step used for time advancement.
- `bc_upb` : boundary conditions on the upper wall, if 0 applies no-slip condition, if 1 applies free-slip condition.
- `bc_lb` : boundary conditions on the lower wall, if 0 applies no-slip condition, if 1 applies free-slip condition.

##################################
Phase field parameters
##################################

- `phi_flag` : if equal to 0 the phase field part is deactivated and the Cahn–Hilliard equation will not be solved; if equal to 1 the phase field part is activated and the Cahn–Hilliard equation is solved. All the following parameters are used only when the phase field is activated.
- `phicor_flag` : Enables different phase-field formulations.
  - 0: Standard CH equation.
  - 1: Standard profile-corrected.
  - 2: Flux-corrected (A Flux-Corrected Phase-Field Method for Surface Diffusion)
  - 3: Profile-corrected turned off at the walls.
  - 4: Profile-corrected kill the gradients (filter on gradients lower than threshold $0.02 * Ch$).
  - 5: Flux-corrected kill the gradients (filter on gradients lower than threshold $0.02 * Ch$).
  - 6: Curvature-subtracted PFM (A redefined energy functional to prevent mass loss in phase-field methods).
  - 7: Conservative Allen-Cahn, Second-order phase-field model (A conservative diffuse interface method for two-phase flows with provable boundedness properties).
  - 8: Conservative Allen-Cahn, Second-order phase-field model (Accurate conservative phase-field method for simulation of two-phase flows).
- `lamcorphi`: Coefficient used to tune the profile-correction, to be set only when `phicor_flag`=1,2,3,4,5.

- `matchedrho` : if equal to zero the two phases have different densities; if equal to 1 their densities are equal. Warning: this value and the following one must be coherent, otherwise the code will stop (if `matchedrho`=0, `rhor` must be different from 1).
- `rhor` : density ratio of the phase $\phi = +1$ over the phase $\phi = -1$ (density ratio of one phase with respect to the carrier, for example density of the drop over density of the carrier fluid).
- `matchedvis` : if equal to zero the two phases have different viscosities; if equal to 1 their viscosities are equal. Warning: this value and the following one must be coherent, otherwise the code will stop (if `matchedvis`=0, `visr` must be different from 1).
- `visr` : viscosity ratio of the phase $\phi = +1$ over the phase $\phi = -1$ (viscosity ratio of one phase with respect to the carrier, for example viscosity of the drop over viscosity of the carrier fluid).
- `non_newtonian`: Enables the non-Newtonian Carreau-Yasuda model in the phase $\phi = +1$. `matchedvis` must be also set to zero
- `exp_non_new`: Exponent of the non-Newtonian model, `viscosity ratio` is used to set the infinity viscosity (see Carreau-Yasuda model)
- `We` : value of Weber number.
- `Ch` : value of Cahn number. Defines the width of the interface; always make sure that the interface contains at least three points (check with the code contained in the folder grid_check)
- `Pe` : value of Peclet number (for the phase-field variable ϕ)
- `Fr` : value of Froud number.
- `body_flag`: Introduce a body force $\propto Bd * (\phi + 1)/2$ (see below for detail on Bd.
- `Bd`: Coefficient for the body force.
- `bodydir`: Direction of the body force.
- `sgradp_flag`: Enables S-shaped pressure gradient for Taylor-Couette.
- `sgrapdir`: Set the direction of the S-shaped pressure gradient.
- `ele_flag`: Electric force at the interface
- `stuart`: Stuart number for the electric force.
- `in_condphi`
  1. only phase $\phi = -1$
  2. read input from file (parallel read)
  3. read input from file (serial read, for retro-compatibility with old legacy data files)
  4. 2D drop; accepted input values are radius and height (z coordinate)

5. 3D drop; accepted input values are radius and height (z coordinate)
6. stratified flow; accepted input values are mean height of the wave, sine wave amplitude (x, y direction), sine wave frequency (x, y direction) and random perturbation amplitude
7. 3D drop array; accepted input values are radius of the single drop, height of the drop array (z coordinate), number of drops in x direction and number of drops in y direction. The distance among two drop centers must be at least $2(\text{radius}+5\sqrt{2}\text{Ch})$, otherwise the number of drops will be reduced to fit this constraint.

- `gravdir` : define direction of gravity.
  - $+1$: positive x direction
  - -1 : negative x direction
  - $+2$: positive z direction
  - -2 : negative z direction
  - $+3$: positive y direction
  - -3 : negative y direction
- `buoyancy` : defines which gravity formulation the code will use.
  - 0 : no gravity
  - 1 : buoyancy and weight effects
  - 2 : only buoyancy effects

##################################
Surfactant parameters
##################################

- `psi_flag` : Enables solution of the CH-like equation for the surfactant (second order)
- `Pe_psi`: Surfactant Peclet number.
- `Ex`: Ex number for the surfactant.
- `Pi`: Pi number for the surfactant (diffusive term is proportional to Pi/Pe).
- `El`: Elasticity number, effect of the surfactant on surface tension.
- `in_condpsi` : defines initial condition for the surfactant.
  - 0 : Constant value.
  - 1 : Read input from file (parallel read).
  - 2 : Initialize equilibrium profile (psi_bulk).
  - 4 : Equilibrium profile multiplied with Y gradient.
  - 5 : Equilibrium profile multiplied with Z gradient.
  - 6 : Diffusion Test, angular distribution.
- `psi_mean`: Average surfactant concentration.

- psi_bulk: Bulk surfactant concentration.

############################
Energy equation parameters
############################

- temp_flag : Enables solution of the energy equation (temperature).
- Ra: Rayleigh number; for Rayleigh-Benard chose $Re = \sqrt{Ra * Pr}/4$.
- Pr: Prandtl number.
- A,B,C,D,E,F: Parameters used to setup Boundary condtions as follows: for $z = -1$, $A * T + B * dT/dZ = C$; for $z = +1$, $D * T + E * dT/dZ = F$.
- in_cond_temp : defines initial condition for the temperature field.
  - 0 : Initial constant temperature (mean value).
  - 1 : Read from data (parallel read).
  - 2 : Phase $\phi = +1$ (hot) and $\phi = -1$ (cold), only for heat transfer in multiphase turbulence.
  - temp_mean: Mean temperature for initial condition.
  - boussinesq: Activate buoyancy term in N-S.

############################
LPT parameters
############################

- part_flag: Enables Lagrangian particle Tracking.
- part_number: Number of particle for each set.
- tracer: Define if the particle is inertial (0) or is a tracer (1).
- nset: Number of sets of particles, multiple sets can be used only in the one-way coupled regime.
- stokes: Stokes number for each particle set.
- stokes_drag: Type of drag force for the particle, 0 standard Stokes drag, 1 Corrected via the Schiller-Naumann coefficient.
- part_gravity: Add the gravity term to the particle.
- twoway: One- or two-way coupled LPT, two-way is at the moment not implemented but subroutines are already present.
- part_dump: Frequency of the position, velocity, fluid velocity (at particle position) outputs for the particles.
- subiterations: Number of sub-iterations performed, must be greater than one when for $St << 1$.
- in_cond_part_pos:
  - 0 : Random position.

- 1 : Read input from file (parallel read).
- 2 : initialize random position on a $x - y$ plane at height `par_plane`
- 3 : initialize random position on N $x - y$ planes.

- `in_cond_part_vel`:
  - 0 : zero velocity.
  - 1 : read input from file (parallel read).
  - 2 : fluid velocity at particle position.
  - 3 : read input from file (parallel read).

1.3.2 Cleaning of `set_run` folder

This part of the script delete old simulation files in the `set_run` folder, to make it ready for a new run. The following files and folders are removed: `go.sh`, `sc_compiled`, `paraview_output`, `stats_calc`, `nohup.out`; if it is a new simulation also the folder `results` is removed, otherwise it is left untouched.
The restarted case will read the initial fields from the `results` folder at the beginning of the simulation.

1.3.3 Copying and editing

First, the script `go.sh` is copied in the `set_run` folder and the correct value of MPI process to be used for the run `NNT=NYCPU*NZCPU` is replaced in the copied version. Then, the input file `input.f90` is copied in the `set_run/sc_compiled` folder and all the parameter are replaced in the file with the correct vaue.
At this point all the source files of the code are copied from the folder `source_code` to the folder `set_run/sc_compiled`. Also the two folders `paraview_output_fg` and `stats_calc` are copied into the folder `set_run`.
If the phase field is activated, an input file for the phase field initial condition is created (`input_phase_field.f90`); this input file is different for the different initial condition that can be chosen. Likewise, for the surfactant, temperature and Lagrangian particles (if activated), files called `input_surfactant.f90`, `input_temperature.f90` and `input_particle.f90` are created.

Lastly, all the flags for the conditional compilation are replaced in the copied source files of the code; this way, depending on the parameters choice, the code will be compiled in different ways including or omitting some parts. This is done to avoid unneeded `if` clauses in the execution of the code as they will reduce performance (especially in `do` loops). After this step the code will be compiled and all the module files will be created in the folder `set_run/sc_compiled`, together with the executable.
If you are running on a local machine, you can leave uncommented the last three lines of the script, such that the `compile.sh` script will switch to the `set_run` folder, run the `go.sh` file and then switch back to the main folder.

1.4 Compiling the code

1.4.1 Compiling for CPUs

To compile and run on CPU-based architectures a complete MPI+FFTW setup is required (i.e. MPI and FFTW libraries installed, along with a Fortran compiler). The

invoked command for the compilation will be `mpif90` (deprecated) or `mpifort` (or similar), this is a call to the MPI compiler wrapper. MPI compiler wrappers combine the base compilers (gfortran, ifort, ftn (HPE Cray Compilers), nvfortran (Nvidia), and aocc (AMD)) with MPI and various other libraries to enable the streamlined compilation of scientific applications. A MPI parallel code CANNOT be compiled with gfortran (or other compilers) but should be compiled with the respective MPI wrapper. The invoked libraries (e.g. MPI or FFTW) should be also included among the list of the modules loaded in the header of the source code. For instance, for MPI, use `include mpif.h` (deprecated) or `use mpi` (deprecated) or `use mpi_f08`. This latter option is in theory the recommended one; however, support to this module is still partial for some compilers. For this reason, `use mpi` is employed.

Regarding the MPI libraries, a library complying with the MPI standard 3.0 is required (particles require shared memory capabilities, introduced in 3.0). Remember that in general, the library version indicated by the MPI library vendor (e.g. `openMPI 5.0`) does not correspond to the MPI library standard (indeed the MPI standard 5.0 does not exist at the moment). The code has been tested with MPI libraries from different vendors: openMPI, mpich, IBM-Spectrum (based on openMPI), Intel, Cray. Remember that openMPI and openMP are two totally different things (please do not confuse them).

Regarding the FFTW libraries, the path of the `/include` and `/lib` folders should be specified in the makefile (unless alias are created). Also for this library, the specific module should be specified in the header of the source code. The code supports the latest version of FFTW (3.x). When using Intel-based machines, the Math Kernel Library (MKL) library can be used instead of the FFTW library to perform the FFTs. This is automatically done once the respective modules have been loaded (intel-one-api). Specifically, the MKL library automatically recognizes the call to `FFTW` subroutines and replace them with call to the corresponding `MKL` library functions.

1.4.2 Compiling for GPUs

To compile and run the code on GPU-based architectures (Nvidia only), the Nvidia `hpc-sdk` toolkit is required. The toolkit includes a version of the MPI libraries (openMPI), the Nvidia compiler with support for the openACC directives (nvfortran, ex pgifort) and GPU libraries (cuFFT and many others). In some clusters, the compiler and the MPI libraries are located in two different modules and both should be loaded. This is because sometimes different MPI libraries are used in combination with the compiler). The GPU-version of the code has been tested using MPI Spectrum (Marconi-100) and openMPI (Marconi-100 and Leonardo). Also, the GPU version of the code must be compiled using `nvfortran`. This is the only compiler that supports the managed memory feature (CUDA unified memory).

1.5 Running the code

Regardless of the version compiled, the procedure to run the code is slightly different depending if one is using a cluster or a local server/machine.

On HPC clusters, once the code is compiled, a `go.sh` file is present in the set run folder (the code cannot be run directly). This file should be double checked (number of nodes, tasks, partitions, time) and then can be submitted to the load manager (usually SLURM) using the `sbatch` command. The modules you have used to compile the code

should be also loaded in the `go.sh` before the call to `mpirun` or `srun`. The load manager will decide when the job will start an on which nodes. The job status can be checked with `squeue` (see SLURM documentation for options and additional commands). The time, number of nodes and partition requested will of course influence the queue time. Please check also the machine documentation for information on the specific rules and on which storage space you should run the simulations (home, scratch, work, etc.).

On a local machine/server, the code is automatically executed by the `compile.sh` (see the last lines where the `go.sh` file is launched). On most local machines, no load manager is present and before executing the code, one should check the machine configuration (number of cores, threads and GPUs) and the actual load. This latter aspect can be checked via the `top` or `htop` commands. Do not load the machine more than 90%, all the jobs will slow-down drastically (yours as well other people jobs). Job status can be checked via the above mentioned commands and can be canceled just killing one of the MPI process with `kill PID` (where PID is the process ID that can be found executing `top`). Remember to also check the storage available, most machines are based on SSD devices and intensive use can damage them.

1.6 Troubleshooting

A list of common issues along sides with possible solutions is reported in the following list.

- `mpif90 or mpifort: command not fuound`. The invoked MPI wrapper does not exist, the MPI library has not been installed or linked (or modules not loaded). Try reinstalling the library or load the correct modules.
- `undefined reference to MPI ****`. The MPI module (header) has not been included or you are trying to use a serial compiler (gfortran, etc.) on a code that use MPI.
- `undefined reference to fft ***`. The FFTW library has not been installed (or the module loaded). Try reinstalling FFTW or loading the correct module.
- `invalid options`. The specified compiling options are not coherent with the invoked compiler (each compiler has its own set of options).
- `error on mpi get address in subroutine **2**.f90`. Known issue with open-MPI (all versions) and last version of MPICH. This issue can be readily solved including the machine you are using on the last lines modified in the `compile.sh` (just before compiling).
- `code crashes at the first time step`. Check the reading of the fields and the endianness of the initial fields. With some MPI libraries (openMPI and latest release of mpich), the MPI data type should be changed from `internal` to `native`. This change should be done in `read fields.f90` and `write output.f90`.
- `compilation error in initialize particle.f90 related to baseprt`. Known problem, please remove the subroutine from the list of the source code files (file list suource.list).

Chapter 2

Code flowchart

In this chapter a detailed overview of all the subroutines used in the code will be presented. This chapter is divided in five main sections, the first including all the subroutine of the main part of the code, the second one includes all the solver part subroutines, the third the details about physical to modal space transforms and backwards, the fourth the statistic calculation part and the latter describes all the modules included in the code. All the subroutine from a library (for example FFTW, cuFFT or MPI libraries) will not be included here; for further information and a detailed description, please refer to the respective library.

2.1 Main code

2.1.1 `FLOW_36`

This is the main program; it takes care of starting and terminating the MPI execution of the code. The first part of this script is the MPI parameter definition part: once the code is run on a determinate number of MPI processes, all the MPI processes are numbered and the total number of MPI processes is defined (the total number of MPI process is already defined when running the code, here its value is assigned to a variable). The number of MPI process in the y and z directions (`NYCPU` and `NZCPU`) and the number of grid points in the three directions are already defined in the module `commondata` as parameters, such that they are known before the input section. After the MPI initialization the code verifies that the number of MPI processes in the two directions and the grid are compatible, which means that each MPI process must always have at least one point in each direction. Due to the domain transposition and the Fourier transform the code must verify that `NYCPU` is smaller than `NX`/2 and `NY` and that `NZCPU` is smaller than `NZ` and `NY`. If this check is passed the code goes on with the execution, otherwise it will stop prompting an error message.
The input part is performed with the call to the `read_input` subroutine.
Since the domain is divided in a Cartesian-like grid (for further details, refer to Chapter 3), a MPI Cartesian communicator is defined, such that all MPI communications can be strongly simplified (especially when it comes to find to which MPI process data must be sent and from which one must be received). Here are also defined the two MPI derived datatypes used for MPI input/output operation (for further details refer to Section 2.1.14). When only Eulerian variables are solved (e.g. flow, phase-field, surfactant, temperature), the MPI communicator is unique and no splitting occurs. By opposite, when particles are tracked (Lagrangian points), two MPI communicators are created, one for

the Eulerian variables (i.e. the MPI tasks taking care of solving the Eulerian variables, `flow_comm`) and one for the particles (i.e. the MPI tasks taking care of tracking the particles, `part_comm`).
At this point it comes to the plan creation for the FFTW calls (or cuFFT): this subroutine creates the plans that will be used when performing Fourier and Chebyshev transforms.
The next step is the creation and saving of the axis arrays x, y and z. x and y (the periodic directions, discretized with a Fourier series) have constant spacing among the grid points; z axis, due to the Chebyshev discretization, uses Chebyshev nodes, defined as $\cos\left[(k-1)\pi/(\mathtt{NZ}-1)\right]$ with k spanning from 1 to `NZ`.
After the plan creation (see the corresponding subroutine also), the array containing the wave numbers are defined in the subroutine `wave_numbers`.
The velocity field initialization is performed by the subroutine `initialize`, while the subroutine `initialize_phi` initialize the phase field (if activated). Similarly, if surfactant and heat transfer module ara activated. Then the statistic calculation variables are set up by the subroutine `initialize_stats`, while the subroutine `initialize_check` initialize the simulation check parameters (bulk Reynolds number, ...). Here there is also the first call to `integral_phi`, which allows to check some useful phase field related quantities at run time, giving a general idea of the quality of the simulation.
At this point the initial fields are saved both in physical and modal space and then the time iteration loop starts. During the time advancement, first the subroutine `solver` is called; then, according to their saving frequency, statistics and fields are saved. At the end of the time step the simulation check subroutines `integral_phi` and `sim_check` are called. At the end of the time advancement the auxiliary files created for statistics calculation are deleted, the final fields are saved in physical and modal space and the allocated variables are deallocated (subroutines `destroy` and `destroy_phi`). Then the FFTW plans (`destroy_plan`), the MPI derived datatypes and the MPI Cartesian communicator are freed. The program concludes with the MPI finalization call.

In the following, there is a list of the subroutines (together with a brief explanation) of the subrouines called in the `main.f90` of FLOW36.

2.1.2 `read_input`

This subroutine reads the edited version of the file `input.f90`. If the simulation is a restart, the initial conditions on both the velocity and the phase field are forced to read from the `results` folder. This subroutine also checks that the parameters `matchedrho`, `rhor`, `matchedvis`, `visr` are coherent. At the end of the subroutine it calls the subroutine `print_start` that print to screen all the informations about the simulation when it starts.

2.1.3 `define_sizes`

This subroutine define the sizes of the data arrays for each MPI process, both in physical space (`fpy`, `fpz`) and in modal space (`spx`, `spy`); these sizes can differ from one MPI process to another.
In physical space each rank holds an array `NX`×`fpz`×`fpy`, while in modal space this array has size `spx`×`NZ`×`spy`. Each of these sizes is defined in the following way (here a 1D case is presented for simplicity, the extension to 3D is straightforward): N points must be divided in N_t MPI processes. The MPI processes are numbered from 0 to $N_t - 1$; if their identification number (also called rank) is lower than the remainder of the division

of N by N_t, $r =$ mod (N, N_t), then they will hold $(N - r)/N_t + 1$ points, otherwise only $(N - r)/N_t$ points.

2.1.4 `create_plan`

This subroutine creates the plans for the Fourier and Chebyshev transform performed by the code. When the FFTW library is used (CPU), the conditional compilation flag `flag_fftw` determines which algorithm will be chosen to perform the transforms. If a value of 0 is provided, the default algorithm will be chosen, otherwise if 1 is provided the creation plan will last much longer, and the best performing algorithm (for a determinate architecture, size of the arrays and memory distribution of the arrays) will be chosen. At the present time is still to be determined whether the flag 1 gives better performance than the flag 0. The FFTW library subroutines will not be reported here; if needed refer to M. Frigo and S. G. Johnson (2017). When GPUs are used, plans are created via calls to the corresponding cuFFT subroutines. For the cuFFT library, there is no option to select the FFT algorithm.

2.1.5 `create_plan_fg`

This subroutine creates the plan for FFTs (FFTW or cuFFT) performed on the more refined grid. The subfix `_fg` identify plans for the more refined grid.

2.1.6 `dump_grid`

This subroutine saves the x, y and z arrays in the folder `set_run/results`.

2.1.7 `wave_numbers`

This subroutine creates the arrays containing the wave numbers in the x and y direction.

$$k_x(i) = \frac{2\pi(i-1)}{L_x} \qquad i = 1, \ldots, \frac{N_x}{2} + 1$$

$$k_y(j) = \begin{cases} \dfrac{2\pi(j-1)}{L_y} & j = 1, \ldots, \dfrac{N_y}{2} + 1 \\ -\dfrac{2\pi(N_y - j + 1)}{L_y} & j = \dfrac{N_y}{2} + 2, \ldots, N_y \end{cases}$$

Here are also defined some useful parameters:

$$\gamma = \frac{\Delta t}{2\mathrm{Re}}$$

$$k^2(i,j) = k_x^2(i) + k_y^2(j) \qquad i = 1, \ldots, \frac{N_x}{2} + 1 \ , \quad j = 1, \ldots, N_y$$

Wave numbers for the more refined grid are also generated. If the expansion factors are equal to one, they match the one previously defined.

2.1.8 `initialize`

This subroutine first allocate the velocity arrays both in physical and modal space, then calculate the mean pressure gradient in modal space. The following step is setting the initial conditions on the velocity; at the present moment the following initial conditions are implemented.

1. Zero velocity field.
2. Laminar Poiseuille flow, generated by a unitary mean pressure gradient in x direction.
3. Random velocity field with values among $[-0.01, 0.01]$.
4. Read from an input file (either in physical space or in modal space, depending on which one is available). This is a parallel read (MPI input/output) and is the condition used for a restart.
5. Read from an old input file (for retro-compatibility with the previous code). This read is serial, every MPI process reads the whole flow field and keeps only its own part. May not work on clusters for large grids (not enough available memory in the node).

Then, depending on the user choice, the boundary conditions on the velocity and the vorticity at the upper and lower walls are set.
At the end of the subroutine the auxiliary problems for the influence matrix method are solved and included in the module `velocity`.

2.1.9 `initialize_phi`

This subroutine allocates the phase field arrays in physical and modal space at first; then the paraemter `s_coeff` is defined. This parameter is used for the splitting of the Cahn–Hilliard equation.

$$\texttt{s\_coeff} = s = \sqrt{\frac{4\mathrm{PeCh}^2}{\Delta t}}$$

For the case of non-matched densities only, additional velocity arrays are allocated; they will be used to store the velocity of the previous time step, needed for the calculation of the non linear part of the time derivative. For further details on the solution algorithm refer to Section 2.2. At this point the initial conditions on the phase field are defined:

1. $\phi = -1$ all over the domain.
2. Read from input file (parallel read). This is the condition enforced in a restarted case.
3. Serial read from input file for retro-compatibility with the previous code. This may not work on clusters for very large grids, as it may exceed the node memory as each MPI process loads the whole field.
4. Initialize 2D single drop on the $y - z$ plane (actually it is a cylinder with the axis in the x direction). This condition is employed when performing 2D simulations. The initialization is done by the subroutine `drop_2d`.
5. Initialize 3D single drop. The initialization is done by the subroutine `drop_3d`.

6. The subroutine `stratified` is called and a stratified flow is initialized.

7. Initialize a 2D array of 3D drops; done from the subroutine `drop_array`.

2.1.9.1 `drop_2d`

This subroutine reads from the `inpu_phase_field.f90` file the radius of the drop and the height of its center, and then initializes a cylinder with given radius and the axis at $y = L_y/2$ and $z =$ `height`. The parameters `radius` and `height` must be specified in the `compile.sh` script.

2.1.9.2 `drop_3d`

This subroutine reads from the `input_phase_field.f90` file the radius of the drop and the height of its center, and then initializes a sphere with given radius. The center of the sphere is at $(L_x/2, L_y/2,$ `height`$)$. The parameters `radius` and `height` must be specified in the `compile.sh` script.

2.1.9.3 `stratified`

This subroutine initializes a stratified flow with a wave perturbation in x and y directions and a random perturbation. It reads from the `input_phase_field.f90` file the wave amplitude and the wave frequency in x and y direction, the amplitude of the random perturbation and the mean height of the interface.

$$\phi(x, y, z) = -\tanh\left(\frac{k(x, y, z) - z}{\sqrt{2}\mathrm{Ch}}\right)$$

$$k(x, y, z) = h + A_x \sin(\omega_x x) + A_y \sin(\omega_y y) + A_r(2\mathrm{rand} - 1)$$

The input parameters $h =$`height`, $A_x =$`wave_amp_x`, $\omega_x =$`wave_freq_x`, $A_y =$`wave_amp_y`, $\omega_y =$`wave_freq_y` and $A_r =$`pert_amp` must be specified in the `compile.sh` script.

2.1.9.4 `drop_array`

This subroutine initializes a 2D array of 3D drops in a $x-y$ plane. The radius, the number of droplets and the height of this plane are specified in the `input_phase_field.f90` file. The distance between two drop centers must be at least $2($`radius` $+ 5\sqrt{2}\mathrm{Ch})$, otherwise the number of droplets in the direction where this condition is not met is reduced. The actual number of droplets used in the simulation is then printed at the beginning of the run.
The input parameters that must specified in the `compile.sh` script are: the radius of the droplets (`radius`), the height of the $x - y$ plane (`height`), the number of droplets in the x (`num_x`) and y (`num_y`) directions.

2.1.10 `initialize_psi`

This subroutine allocates the phase field arrays to track the surfactant concentration (second-order parameter ψ). Different initial conditions are available.

2.1.11 `initialize_theta`

This subroutine allocates the phase field arrays to track the temperature (or passive scalar). Different initial conditions are available.

2.1.12 `initialize_particle`

This subroutine allocates the arrays for tracking the particles and initialize the position and velocity of the particles. Different initial conditions are available.

2.1.13 `write_failure`

This subroutine saves in the `results/backup` folder the fields, the checkpoint iteration number and the `time_check.dat` file. These files can be used as a checkpoint to recover a stopped simulation. When updating the checkpoint, to the old checkpoint data is appended the `_old` suffix.

2.1.14 `write_output`, `write_output_spectral` and `write_output_recovery`

These subroutines use MPI I/O subroutines combined with MPI derived datatypes to write in parallel to an output file (either in physical space, either in modal space). Once opened a file and obtained its handle, each MPI process can access only to a part of the file, which is determined by the MPI derived datatype specified in the call to `mpi_file_set_view` subroutine. After writing its own part of file, each MPI process close the file and goes on with the code execution. There is no need for MPI process synchronization during this task.
The two subroutines differ only for the MPI derived datatype used: one saves the data in physical space, while the other in modal space. The default `MPI_datatype` has been recently switched to `native` for compatibility with the most recent OpenMPI releases.

2.1.15 `integral_phi`

This subroutine integrates all over the domain the positive phase field variable $\phi > 0$. This quantity is used to check the quality of the simulation by controlling the mass loss due to numerical diffusion. Reducing the Cahn number reduces the mass losses (but the interface must always be described by at least three points in each direction).

2.1.16 `integral_psi`

This subroutine integrates all over the domain of the surfactant concentration.

2.1.17 `initialize_check`

This subroutine creates the simulation check file `time_check.dat` and fill in the header of the file and the first line for the initial field.
If the phase field is deactivated, the current time (in wall units, t^+) and the bulk Reynolds number will be written to the file. If the phase field is activated it will also write the mean value of ϕ all over the domain and the value of the volume integral on $\phi > 0$.

2.1.18 `sim_check`

This subroutine opens the already created file `time_check.dat` and adds the line for the current time step. The data written to the file are the same indicated in the subroutine `initialize_check` (exception made for the file header).

2.1.19 `destroy`

This subroutine takes care of deallocating all the fields array: `destroy` deallocates all the velocity (both in physical and modal space) arrays.

2.1.20 `destroy_plan`

This subroutine frees the plan created for the FFTW/cuFFT execution. This subroutine works on both the reference grid and refined grid plans.

2.2 Solver subroutines

The subroutines reported hereunder are the ones involved in the solution of the governing equations and are called at each time step (at least). In the following, the macro-structure of the solver is first detailed and then a description of the subroutines is provided.

First, the Navier-Stokes equations are solved; the subroutine `convective_ns` is called; this subroutine calculates the non-linear term arising from the convective term in the Navier–Stokes equation and it includes also the additional term coming from the non-matched densities case. Then the mean pressure gradient in x and y direction is added to the non-linear term and, if the phase field, surfactant, temperature transport equations, the non-linear terms arising from the surface force, gravity and buoyancy force and the non-linear part of the time derivative are also added to the non-linear term of NS. After this step the non-linear term is integrated explicitly using an explicit Euler algorithm at the first time step and an Adams–Bashforth one for the second time step on. The implicit part is discretized in time with a Crank–Nicolson algorithm. The old non-linear term is then updated with the new one and it will be used again in the following time iteration. The historical term (the right hand side of the equations for velocity and vorticity) starts to get assembled: the first part is an output of the time integration subroutine, then the second and last part is given in output from the subroutine `hist_term`. Then, in order, there is the call to the subroutine `calculate_w`, `calculate_omega` and `calculate_uv` which solve the Navier–Stokes equations, obtaining the velocity (and wall-normal vorticity values). These calls conclude the Navier–Stokes solution part.

Second, If the phase field is activated, the Cahn–Hilliard equation is solved: at first the non-linear term is formed in the subroutine `sterm_ch` and then integrated in time explicitly with an explicit Euler (first time step) or an Adams–Bashforth algorithm (second time step on). The implicit part is integrated with an implicit Euler algorithm. Then the subroutine `calculate_phi` is called.

Third, If the surfactant is activated, the Cahn–Hilliard equation-like equation for the surfactant-concentration is solved: at first the non-linear term is formed in the subroutine `sterm_psi` and then integrated in time explicitly with an explicit Euler (first time step) or an Adams–Bashforth algorithm (second time step on). The implicit part is integrated with an implicit Euler algorithm.

Then, if the energy equation is solved, the transport equation for the temperatures is solved: at first the non-linear term is formed in the subroutine `sterm_`θ and then integrated in time explicitly with a Euler (first time step) or an Adams–Bashforth algorithm (second time step on). The implicit part is integrated with an implicit Crank–Nicolson algorithm.

Finally, if also the Lagrangian particle tracking is enabled, particles are also time advanced. Depending on the type of particle considered, first the velocity at the particle position is obtained using the subroutine `lagran4`, then the particle position and velocity

(for inertial particles only) at the new time step are obtained using an explicit Euler scheme.

The subroutines reported in the following (transforms excluded) are called at least once during the time advancement of the governing equations. A brief description for each subroutine is also provided.

2.2.1 `phi_non_linear`

This subroutine calculates all the non-linear terms of the Navier–Stokes equation which arises from the presence of the phase field.
The first term calculated is the surface force, then the non-linear part of the viscous term, the gravity and buoyancy term and the non-linear part of the time derivative.
The non linear-part of the viscous term is non-zero only for the non-matched viscosities case, while the gravity and buoyancy term and the non-linear part of the time derivative are non-zero only for the non-matched densities case.
At the end of the subroutine these non-linear contributions are added to the non-linear terms `s1`, `s2` and `s3`.

2.2.2 `euler`, `adams_bashforth`

This subroutine performs the explicit time integration of the non-linear terms, using an explicit Euler algorithm at the first time step and an Adams–Bashforth one from the second time step on. `euler` and `adams_bashforth` are used for the Navier–Stokes non-linear terms.

2.2.3 `euler_phi` and `adams_bashforth_phi`

This subroutine performs the explicit time integration of the non-linear terms of the Cahn-Hilliard equation, using an explicit Euler algorithm at the first time step and an Adams–Bashforth one from the second.

2.2.4 `euler_psi` and `adams_bashforth_psi`

This subroutine performs the explicit time integration of the non-linear terms of the CH-like equation for the surfactant, using an explicit Euler algorithm at the first time step and an Adams–Bashforth one from the second.

2.2.5 `euler_theta` and `adams_bashforth_theta`

This subroutine performs the explicit time integration of the non-linear terms of the energy equation equation, using an explicit Euler algorithm at the first time step and an Adams–Bashforth one from the second.

2.2.6 `hist_term`

This subroutine assembles the right hand side of the fourth order equation for the wall-normal velocity and for the second order equation for the wall-normal vorticity.

2.2.7 `hist_term_temp`

This subroutine assembles the right hand side of the energy equation. Note: This procedure can be simplified and made directly in the solver (like for the surfactant, and the second-order phase-field method).

2.2.8 `calculate_w`

This subroutine solves the equation for the wall-normal component of the velocity. The fourth order equation is split in two second order Helmholtz like equations; due to the lack of boundary conditions on one of the two equations, the influence matrix method is used here.
In this subroutine all the quantities needed to solve the Helmholtz problem are set up.

2.2.9 `calculate_omega`

This subroutine sets up all the quantities needed to solve the second order Helmholtz equation for the wall-normal vorticity. At the end of the subroutine the new wall-normal vorticity is updated with the new value.

2.2.10 `calculate_uv`

This subroutine calculates the streamwise and spanwise velocity components starting from the wall-normal velocity and the wall-normal vorticity. Here the continuity equation and the definition of wall normal vorticity are used.

2.2.11 `sterm_ch`

This subroutine calculates the non-linear term of the Cahn–Hilliard equation; this term includes the convective term and part of the laplacian of the chemical potential.

$$S_\phi = -\mathbf{u} \cdot \nabla\phi + \frac{1}{\mathrm{Pe}}\nabla^2\phi^3 - \frac{s+1}{\mathrm{Pe}}\nabla^2\phi$$

The s coefficient allows for the inclusion of part of the diffusive-like term in the non-linear term (to improve numerical stability) and it is defined as:

$$s = \sqrt{\frac{4\mathrm{PeCh}^2}{\Delta t}}$$

The term mentioned above is the standard term computed using the standard phase-field method equation (CH), i.e. using `phicor_flag`=0. If `phicor_flag` is not equal 0, the computed non-linear terms are different and depend on the formulation considered. Also the splitting coefficient might be different (recomputed), as some terms (diffusive) are included in the linear part for stability reasons. For sake of brevity (otherwise this becomes the bible and not an handbook), the formulation of these terms is not reported but can be easily obtained from the subroutine.

2.2.12 `sterm_psi`

This subroutine calculates the non-linear term of the CH-like for the surfactant. This term is computed in a non-intuitive way for performance. In particular, each term is not computed directly, but instead the term is first decomposed in three different pieces using vectorial identity. Finally, the advection term is then added.

2.2.13 `sterm_temp`

This subroutine calculates the non-linear term of the energy equation for the temperature.

2.2.14 `calculate_phi`

This subroutine first splits the fourth order equation for the phase variable in two second order Helmholtz equations and then solves these Helmholtz equations. Here there is no need for the influence matrix method, as enough boundary conditions are provided for both Helmholtz equations. This subroutine is used when the Cahn-Hilliard equation is used (`phicor_flag`=1 to 6).

2.2.15 `calculate_phi_ac`

This subroutine solves the second order equation for the phase variable when the conservative Allen-Cahn equation is employed (`phicor_flag`=7) . Here there is no need for the influence matrix method, as enough boundary conditions are provided for the Helmholtz equation.

2.2.16 `calculate_psi`

This subroutine solves the second order equation for the CH-like equation for the surfactant. Here there is no need for the influence matrix method, as enough boundary conditions are provided for the Helmholtz equation.

2.2.17 `calculate_theta`

This subroutine solves the second order equation for the energy transport equation. This is basically the advection-diffusion equation of a scalar. Here there is no need for the influence matrix method, as enough boundary conditions are provided for the Helmholtz equation.

2.2.18 `courant_check`

This subroutine calculates the maximum Courant number on the whole domain; if the calculated Courant exceeds the Courant limit, the simulation is stopped.

2.2.19 `dz` and `dz_red`

These subroutines calculate the wall-normal derivative in the modal space. The only difference between the two subroutines resides in the passed array dimensions.
Since there are dependencies between the input and output arrays, the arrays passed to the subroutine must not be the same, otherwise the output array will be wrongly calculated.

2.2.20 `dz_fg`

This subroutine computes the wall-normal derivative in the modal space of a variable defined in the fine grid space.

2.2.21 `helmholtz`, `helmholtz_red` and `helmholtz_rred`

All these subroutines solve the given Helmholtz problem; they differ only for the passed array dimensions. The input of the subroutine is the right hand side of the equation and, at the end of the subroutine, is replaced by the solution of the Helmholtz problem. The subroutine asssembles the matrices that will be passed to the Gauss solver. The coefficient matrix holds in the first two lines the boundary conditions at the upper and lower wall, then the rest of the matrix is a tridiagonal-like matrix that allows for a fast and efficient Gauss solver.

2.2.22 `helmholtz_fg`

This subroutine solves the Helmholtz problem for a variable defined in the fine grid space.

2.2.23 `gauss_solver`, `gauss_solver_red` and `gauss_solver_rred`

These subroutines perform the Gauss back-substitution algorithm on the passed set of equations (matrix form) and return the solution. They only differ for the size of the arrays passed.

2.2.24 `lagrangian_tracker`

This subroutine advance the position and velocity of the Lagrangian particles. This operation requires of the subroutines reported below to evaluate the velocity at the particle position and the forces acting on the particle.

2.2.25 `lagran4`

This subroutine evaluates the velocity at the particle position. A 4-th order Lagrangian interpolation is used to compute the velocity at the particle position.

2.2.26 `calculate_forces`

This subroutine evaluates the forces acting on the particles. It computes the right hand side of the governing equation for the particle velocity.

2.3 Transforms

The two most important subroutine used to shuttle data from physical to modal space and backward are `phys_to_spectral` and `spectral_to_phys`. These subroutines accept as arguments the input array in physical [modal] space, the output array in modal [physical] space and a flag for the dealiasing of the output array. The dealiasing follows the 2/3 rule, which means that only the 2/3 of the lower modes will be retained.

2.3.1 `phys_to_spectral`

At the beginning of this subroutine each MPI process holds an array containing all the data in the x direction and only part of the data in the y and z directions; the array is in physical space. At first the subroutine `fftx_fwd` performs a 1D Fourier transform in the x direction. Since all data in a certain direction are needed when performing a

transform, the subroutine `yz2z` takes care of exchanging data among the various MPI processes, such that, after this subroutine each MPI process holds all the data in the y direction and only part of the data in the x and z directions. Now, the subroutine `ffty_fwd` performs a 1D transform in the y direction. At this point another subroutine `xz2xy` exchange data among the various MPI processes, such that each MPI process holds all the data in the z direction. Finally the subroutine `dctz_fwd` performs a discrete Chebyshev transform in the z direction.
At the end of the subroutine, the array is in modal space.

2.3.2 `phys_to_spectral_fg`

Same as `phys_to_spectral` but for a variable defined in the fine grid space.

2.3.3 `fftx_fwd`

This subroutine performs a discrete Fourier transforms in the x direction on the input array. It is a real-to-complex transform: the input is a real array, while the output is a complex array. The output array is defined as a real in the code but the last index of the array determines whether it is the real or the imaginary part (1 corresponds to the real part, 2 to the imaginary part). According to the dealiasing flag, this subroutine can perform dealiasing in the x direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the transform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler.

2.3.4 `fftx_fwd_fg`

Fine grid version.

2.3.5 `yz2xz`

This subroutine exchange data among MPI processes such that at the beginning of the subroutine each MPI process holds all data in the x direction and only part in the y and z directions and at the end of the subroutine each MPI process holds all data in the y direction and only part in the x and z directions. The MPI communications among the various MPI processes are easily handled by exploiting the Cartesian topology defined for the MPI processes.

2.3.6 `yz2xz_fg`

Fine grid version.

2.3.7 `ffty_fwd`

This subroutine performs a discrete Fourier transforms in the y direction. Both input and output arrays are complex, the last index of the array determines the real (1) and imaginary (2) part. According to the dealiasing flag, this subroutine can perform dealiasing in the y direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the transform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler.

2.3.8 `xz2xy`

This subroutine takes care of exchanging data among MPI processes, such that at the beginning of the subroutine each MPI process hold all data in the y direction and only part in the x and z direction, while at the end of the subroutine each MPI process holds all data in the z direction and part of them in the x and y directions. As before, a Cartesian topology is exploited during MPI communications.

2.3.9 `xz2xy_fg`

Fine grid version.

2.3.10 `dctz_fwd`

This subroutine performs a discrete Chebyshev tranforms in the wall-normal direction; it work on complex valued arrays and their storage in memory is the same as stated in Section 2.3.4. According to the dealiasing flag, this subroutine can perform dealiasing in the z direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the transform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler.

A few notes for the GPU version only: this is a real-to-real transform that is not directly supported by cuFFT. However, it it possible to use FFT routines to perform DCT: first the array is made even symmetric and then the real and complex part of the array undergo a classic FFT transform, only the real part of the output is kept (fun fact, using this trick, DCT forward and backward can be done with the very same code). In addition, as transforms on the GPUs are very fast, contrary to the CPU implementation where DCT is performed row by row, here DCT is performed in a batched mode. This however requires transposition of the input array so that the input satisfies the advanced data layout of FFTW. Even with transposition (back and forth), DCT performed in this way is faster than row by row (or slice by slice).

2.3.11 `dctz_fwd_fg`

Fine grid version.

2.3.12 `spectral_to_phys`

This subroutine shuttle the data array from modal space to physical space. At the beginning of the subroutine each rank holds all the data in the z direction and only a part in the x and y directions. The subroutine `dctz_bwd` perform an inverse Chebyshev transform in the z direction; then the subroutine `xy2xz` exchange data among MPI processes, such that, after the subroutine execution, each MPI process holds all the data in the y direction and only a part in the other two dimensions. Then the subroutine `ffty_bwd` performs a 1D inverse Fourier transform on the data and the subroutine `xz2yz` exchange data among MPI processes in such a way that each MPI process holds all data in the x direction. After the call to `fftx_bwd`, which performs an inverse Fourier transform on the data, each MPI process holds the data in physical space.

2.3.13 `spectral_to_phys_fg`

Fine grid version.

2.3.14 `dctz_bwd`

This subroutine performs an inverse discrete Chebyshev transforms in the wall-normal directions; it works on complex valued arrays and their storage in memory is the same as stated in Section 2.3.4. According to the dealiasing flag, this subroutine can perform dealiasing in the x direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the transform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler. As for the `dctz_fwd`, the GPU implementation of the DCT transform is slightly different as the real-to-real transform is not directly supported by cuFFT (see above for details).

2.3.15 `dctz_bwd_fg`

Fine grid version.

2.3.16 `xy2xz`

This subroutine exchange data among the MPI processes, such that in input each MPI process holds all data in the z direction and only part in the x and y directions and in output each MPI process holds all data in the y directions and only part in the x and z directions. The MPI communications are easily handled by using a Cartesian topology.

2.3.17 `xy2xz_fg`

Fine grid version.

2.3.18 `ffty_bwd`

This subroutine performs an inverse discrete Fourier transfoms in the y direction. According to the dealiasing flag, this subroutine can perform dealiasing in the y direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the transform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler.

2.3.19 `ffty_bwd_fg`

Fine grid version.

2.3.20 `xz2yz`

This subroutine exchange data among the MPI processes, such that in input each MPI process holds all data in the y direction and only part in the x and z directions while in output each MPI process holds all data in the x directions and only part in the y and z directions. The MPI communication are easily handled by using a Cartesian topology.

2.3.21 `xz2yz_fg`

Fine grid version.

2.3.22 `fftx_bwd`

This subroutine performs an inverse discrete Fourier transforms in the x direction on the input array. It is a complex-to-real transform: the input is a complex array, while the output is a real array. According to the dealiasing flag, this subroutine can perform dealiasing in the z direction. Depending on the value of the `openacc_flag`, FFTW (CPU) or cuFFT (GPU) libraries are used to perform the trasnform. All transform subroutines are inside modules (since May 2022) for visibility and compatibility with Nvidia Fortran compiler.

2.3.23 `fftx_bwd_fg`

Fine grid version.

2.4 Statistic calculation

This section is addressed to the run time statistics calculation; at the present moment the code can calculate mean, root mean square, skewness and flatness of the velocities, mean and root mean square for the pressure (without considering the mean pressure gradient), energy budget for a single phase channel flow (mean flow in the x direction) and the power spectra of the velocity fluctutations at $z^+ = 5$, $z^+ = 15$ and $z^+ = \mathrm{Re}$.

2.4.1 `initialize_stats`

This subroutine initializes the statistic calculation; if the simulation is not a restart it initializes the counter `flowiter` to 0 and creates the files where statistics are saved. On the other hand, if the simulation is restarted, it reads the flowiter value from the files where statistic are saved and exit the subroutine.
For the new simulation case, if the time step from where the statistics calculation starts is the initial time step the statistics are calculated, otherwise the statistics are initialized to zero and the counter is reduced by one.

2.4.2 `del_old_stats`

At the end of the time advancement cycle this subroutine deletes the old statistics file, denoted by the suffix `_old.dat` in the results folder. This file are kept so that, if the simulation crashes when writing new statistics to the corresponding file, there is a backup and the simulation can be restarted from the previous time step available.

2.4.3 `statistics`

This subroutine is called during the time advancement cycle to calculate statistics at run time. At first the counter `flowiter` is incremented by one, then the previous statistics files are read and renamed adding the suffix `_old`. Then the statistics at the current time step are read and a weighted time average with the old statistics is performed. At the end of the subroutine the new statistics are written to a file.
Flow statistics (mean, root mean square, skewness and flatness) are written to `stats.dat`, pressure mean and root mean square and the energy budget are written to `budget.dat`, the streamwise power spectra are written to `power_xspectra.dat` and the spanwise power spectra to `power_yspectra.dat`.

2.4.4 `mean_calc`

This subroutine calculates the mean, root mean square, skewness and flatness of the flow field. The results are gathered to the MPI process with number 0, which also takes care of reading and writing to file.

2.4.5 `budget_calc`

This subroutine calculate the mean and the root mean square of the pressure and the energy budgets. Pressure is calculated with the hypothesis of single phase flow, while energy budgets are calculated for a fully developed flow with mean flow only in the x direction and they do not consider the presence of another phase (for a two or more phases flow the pressure will be wrongly calculated and some energy budget terms, like the surface force, would be missing from the total energy budget). Here also, all the data are gathered to the MPI process 0 which writes them to a file.

2.4.6 `sterm_pressure`

This subroutine calculates the non-linear term of the Navier–Stokes equation. These terms are calculated by scratch since the saved non-linear terms in the module `sterms` and the velocity data are shifted by one time step (Δt).

2.4.7 `power_spectra`

This subroutine calculates the streamwise and spanwise power spectra at three differents z locations (z^+ =5, 15 and Re). All the power spectra data are gathered to the MPI process 0.

2.5 Modules

Here all the modules included in the code with the variables there defined will be introduced.

- `commondata`
  - `nx` : number of grid points in x direction (passed as a parameter)
  - `ny` : number of grid points in y direction (passed as a parameter)
  - `nz` : number of grid points in z direction (passed as a parameter)
  - `nycpu` : number of MPI processes in which the y direction is divided (physical space, passed as a parameter)
  - `nzcpu` : number of MPI processes in which the z direction is divided (physical space, passed as a parameter)
  - `rank` : number of the MPI process (each MPI process has its own number, spanning from 0 to `ntask`-1)
  - `ntask` : total number of MPI processes
  - `ierr` : mandatory argument for all MPI subroutine calls (up to `use mpi`, in `use mpi_f08` it is an optional argument)
  - `cart_comm` : Cartesian communicator for the MPI Cartesian topology
  - `xl` : x length of the domain

  - `yl` : y length of the domain
  - `folder` : folder where results are saved (passed as a parameter)

- `grid`
  - `x` : array containing the x axis
  - `y` : array containing the y axis
  - `z` : array containing the z axis

- `velocity`
  - `u` : array containing u velocity (physical space, allocatable)
  - `v` : array containing v velocity (physical space, allocatable)
  - `w` : array containing w velocity (physical space, allocatable)
  - `uc` : array containing u velocity (modal space, allocatable)
  - `vc` : array containing v velocity (modal space, allocatable)
  - `wc` : array containing w velocity (modal space, allocatable)
  - `wa2` : array containing the first auxiliary Helmholtz problem for the influence matrix method (allocatable)
  - `wa3` : array containing the second auxiliary Helmholtz problem for the influence matrix method (allocatable)
  - `sgradpx` : array containing the mean pressure gradient in the x direction (modal space, allocatable)
  - `sgradpy` : array containing the mean pressure gradient in the y direction (modal space, allocatable)

- `wavenumber`
  - `kx` : array containing the x wavenumbers
  - `ky` : array containing the y wavenumbers
  - `k2` : array containing $k^2(i,j) = k_x^2(i) + k_y^2(j)$

- `sim_par`
  - `pi` : value of π (parameter)
  - `Re` : Reynolds number
  - `dt` : time step
  - `gradpx` : mean pressure gradient in x direction (physical space, scalar)
  - `gradpy` : mean pressure gradient in y direction (physical space, scalar)
  - `Co` : limit Courant number
  - `gamma` : $\Delta t/(2\mathrm{Re})$
  - `p_u`, `q_u`, `r_u` : coefficients for the boundary conditions on u, written as

$$pu(\pm 1) + q \left.\frac{\partial u}{\partial z}\right|_{z=\pm 1} = r$$

    Used for the case where $k_x = k_y = k^2 = 0$ and it is not possible to calculate u and v as usual, and a Helmholtz equation must be solved for u.

- `p_v`, `q_v`, `r_v` : coefficients for the boundary conditions on v, written as

$$pv(\pm 1) + q \left.\frac{\partial v}{\partial z}\right|_{z=\pm 1} = r$$

  Used for the case where $k_x = k_y = k^2 = 0$ and it is not possible to calculate u and v as usual, and a Helmholtz equation must be solved for v.
- `p_w`, `q_w`, `r_w` : coefficients for the boundary conditions on w, written as

$$pw(\pm 1) + q \left.\frac{\partial w}{\partial z}\right|_{z=\pm 1} = r$$

- `p_o`, `q_o`, `r_o` : coefficients for the boundary conditions on ω_z, written as

$$p\omega_z(\pm 1) + q \left.\frac{\partial \omega_z}{\partial z}\right|_{z=\pm 1} = r$$

- `zp` : z location of boundaries: $[-1, +1]$
- `bc_up` : boundary condition at the upper boundary
- `bc_low` : boundary condition at the lower boundary
- `restart` : restart flag
- `nt_restart` : time step from which the simulation is restarted
- `in_cond` : initial condition for the flow field
- `nstart` : starting time step
- `nend` : final time step
- `ndump` : saving frequency of solution in physical space
- `sdump` : saving frequency of solution in modal space

- `phase_field`
  - `phi_flag` : flag for the activation/deactivation of the phase field calculations
  - `in_cond_phi` : initial condition for the phase variable
  - `b_type` : flag used to switch from:
    1. no gravity case
    2. gravity and buoyancy case
    3. buoyancy case
  - `rhor` : density ratio
  - `visr` : viscosity ratio
  - `We` : Weber number
  - `Ch` : Cahn number
  - `Pe` : Peclet number
  - `Fr` : Froud number
  - `grav` : gravity versor
  - `s_coeff` : coefficient used for the splitting of the Cahn–Hilliard equation, $s = \sqrt{\frac{4\mathrm{PeCh}^2}{\Delta t}}$

  - `phi` : phase field variable (physical space, allocatable)
  - `phic` : phase field variable (modal space, allocatable)
  - `phi_fg` : phase field variable in fine grid (physical space, allocatable)
  - `phic_fg` : phase field variable in fine grid (modal space, allocatable)

- `surfactant`
  - `phi_flag` : flag for the activation/deactivation of the surfactant calculations
  - `in_cond_phi` : initial condition for the surfactant
  - `Ex` : Ex parameter
  - `Pe_psi` : Peclet number of the surfactant
  - `P_i` : Pi parameter
  - `El` : Elasticity number
  - `psi` : surfactant concentration field variable (physical space, allocatable)
  - `psic` : surfactant concentration field variable (modal space, allocatable)
  - `psi_fg` : surfactant concentration field variable in fine grid (physical space, allocatable)
  - `psic_fg` : surfactant concentration field variable in fine grid (modal space, allocatable)

- `temperature`
  - `phi_flag` : flag for the activation/deactivation of the energy equation
  - `in_cond_theta` : initial condition for the temperature field
  - `Ra` : Rayleigh number
  - `Pr` : Prandtl number
  - `p_theta` : p value for the boundary condition on temperature
  - `q_theta` : q value for the boundary condition on temperature
  - `r_theta` : r value for the boundary condition on temperature
  - `theta` : temperature variable (physical space, allocatable)
  - `thetac` : temperature variable (modal space, allocatable)

- `particle`
  - `part_flag` : flag for the activation/deactivation of the LPT
  - `part_number`: number of particles
  - `in_cond_part_pos` : initial condition for the particle position
  - `in_cond_part_vel` : initial condition for the particle velocity
  - `part_dump`: frequency for saving particles position and velocity
  - `nset`: number of particle sets
  - `subiterations`: number of sub-iterations
  - `part_index`: Id of the particle
  - `dt_part`: dt for the particles
  - `stokes`: stokes number

  - `dens_part`: density ratio of the particle
  - `d_par`: diameter of the particle
  - `xp`: particle position
  - `up`: particle velocity
  - `uf`: streamwise velocity at particle position
  - `vf`: spanwise velocity at particle position
  - `wf`: wall-normal velocity at particle position
  - `Tf`: temperature at particle position
  - `fb_x`: x-component of the force acting on the particle
  - `fb_y`: y-component of the force acting on the particle
  - `fb_z`: z-component of the force acting on the particle
  - `window_*`: MPI windows used to access the Eulerian field in a shared-memory fashion

- `velocity_old`
  - `ucp` : u velocity at previous time step (modal space, allocatable)
  - `vcp` : v velocity at previous time step (modal space, allocatable)
  - `wcp` : w velocity at previous time step (modal space, allocatable)

- `mpiIO`
  - `ftype` : derived datatype, used for MPI input/output operations in physical space
  - `stype` : derived datatype, used for MPI input/output operations in modal space

- `par_size`
  - `fpy` : y size of the field array in physical space (for parallelization)
  - `fpz` : z size of the field array in physical space (for parallelization)
  - `spx` : x size of the field array in modal space (for parallelization)
  - `spy` : y size of the field array in modal space (for parallelization)
  - `cstart` : 3 element array containing the triplet of the lowest indexes in the global indexing system for arrays in physical space
  - `fstart` : 3 element array containing the triplet of the lowest indexes in the global indexing system for arrays in modal space

- `fftw3`
  - `plan_x_fwd` : plan for 1D Fourier transform, x direction via FFTW
  - `plan_y_fwd` : plan for 1D Fourier transform, y direction via FFTW
  - `plan_z_fwd` : plan for 1D Chebyshev transform, z direction via FFTW
  - `plan_x_bwd` : plan for 1D inverse Fourier transform, x direction via FFTW
  - `plan_y_bwd` : plan for 1D inverse Fourier transform, y direction via FFTW
  - `plan_z_bwd` : plan for 1D inverse Chebyshev transform, z direction via FFTW

  - `plan_x_fwd_fg` : plan for 1D Fourier transform, x direction (fine grid) via FFTW
  - `plan_y_fwd_fg` : plan for 1D Fourier transform, y direction (fine grid) via FFTW
  - `plan_z_fwd_fg` : plan for 1D Chebyshev transform, z direction (fine grid) via FFTW
  - `plan_x_bwd_fg` : plan for 1D inverse Fourier transform, x direction (fine grid) via FFTW
  - `plan_y_bwd_fg` : plan for 1D inverse Fourier transform, y direction (fine grid) via FFTW
  - `plan_z_bwd_fg` : plan for 1D inverse Chebyshev transform, z direction (fine grid) via FFTW

- `cufftplans`
  - `cudaplan_x_fwd` : plan for 1D Fourier transform, x direction via cuFFT
  - `cudaplan_y_fwd` : plan for 1D Fourier transform, y direction via cuFFT
  - `cudaplan_z_fwd` : plan for 1D Chebyshev transform, z direction via cuFFT
  - `cudaplan_x_bwd` : plan for 1D inverse Fourier transform, x direction via cuFFT
  - `cudaplan_y_bwd` : plan for 1D inverse Fourier transform, y direction via cuFFT
  - `cudaplan_z_bwd` : plan for 1D inverse Chebyshev transform, z direction via cuFFT
  - `cudaplan_x_fwd_fg` : plan for 1D Fourier transform, x direction (fine grid) via cuFFT
  - `cudaplan_y_fwd_fg` : plan for 1D Fourier transform, y direction (fine grid) via cuFFT
  - `cudaplan_z_fwd_fg` : plan for 1D Chebyshev transform, z direction (fine grid) via cuFFT
  - `cudaplan_x_bwd_fg` : plan for 1D inverse Fourier transform, x direction (fine grid) via cuFFT
  - `cudaplan_y_bwd_fg` : plan for 1D inverse Fourier transform, y direction (fine grid) via cuFFT
  - `cudaplan_z_bwd_fg` : plan for 1D inverse Chebyshev transform, z direction (fine grid) via cuFFT
  - `gerr`: integer to check the errore on cuFFT calls

- `sterms`
  - `s1_o` : non-linear term, Navier–Stokes x component, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)
  - `s2_o` : non-linear term, Navier–Stokes y component, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)

  - `s3_o` : non-linear term, Navier–Stokes z component, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)
  - `sphi_o` : non-linear term of Cahn–Hilliard equation, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)
  - `spsi_o` : non-linear term of Cahn–Hilliard-like equation for the surfactant, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)
  - `stheta_o` : non-linear term of the energy equation, used to avoid to calculate again the old non-linear term at the following time step (modal space, allocatable)

- `stats`
  - `flowiter` : counter, keeps track of the number of field used for time averages
  - `stat_dump` : frequency of statistics calculation and saving
  - `stat_start` : time step from where start the statistics calculation
  - `plane_comm` : MPI Cartesian sub-communicator, used to exchange data among MPI processes in the same $x - y$ plane
  - `col_comm` : MPI Cartesian sub-communicator, used to exchange data among MPI processes that cover the same $x - y$ region (same "column")

- `nvtx`
  - Used only for profiling when GPUs are used, by default is commented, it requires the Nvidia compiler and `-nvtxtools`.

Chapter 3

MPI Parallelization and GPUs

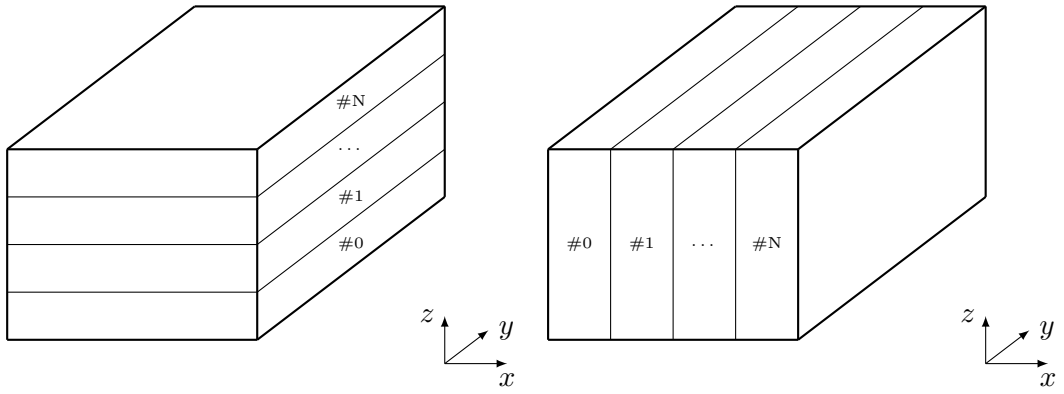
In this chapter, the parallelization, two possible domain decomposition algorithms and the acceleration strategy will be presented: the slab decomposition and the pencil decomposition. Section 3.1 presents the general parallelization strategy. Sections 3.2 and 3.3 will be dedicated to the detailed explanation of these parallelization strategies, focusing also on their strengths and limitations. Section 3.4 will present a benchmark between these strategies, together with some scalability results. Finally, section 3.5 will discuss the acceleration strategy for GPU-use and section 3.6 will discuss possible performance improvement and profiling techniques.

3.1 Parallelization

For the solution of Eulerian fields (flow field, phase-field, surfactant concentration temperature fields), the computational domain is splitted into pencil. Each MPI is assigned to a chunk of nodes (see details in the two sections below). MPI transpositions are required every time non-linear terms are computed (i.e. full forward or backward transforms). When also the Lagrangian particles is enabled (particles, fibers), the situation is different and depends on the number of nodes considered. If running on a single-node, all MPI tasks perform the computations of the Eulerian fields as well as the tracking of the Lagrangian points/entities. If running multi-node with M nodes, $M-1$ nodes solve the Eulerian fields while the node$M$ take care of the Lagrangian tracking using the MPI-shared memory feature.

3.2 Slab decomposition

The slab decomposition is the simpler case of domain decomposition presented here. The domain is divided in so called slabs which contain all the data in two directions and only a part of the data in the remaining direction.
Since here a pseudospectral method is used and when performing transforms each MPI process must hold all the data in the transform direction, during the passage from physical to modal space (and backwards) MPI processes must exchange data. As shown in Figure 3.1 in physical space each MPI process holds all the data in a $x-y$ plane and only a part of data in the z direction, while in modal space each MPI process hold all data in a $y-z$ plane and only a part of data in the x direction.
When in physical space 1D Fourier transforms are performed in the x and y directions; then all MPI processes exchange data to transpose the slabs from the $x-y$ plane to the $y-z$ plane and a 1D Chebyshev transform is performed in the wall-normal direction.



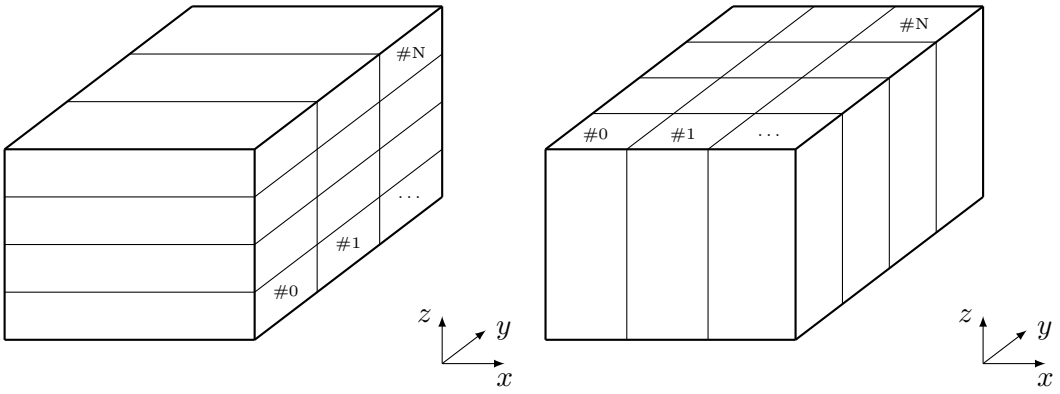
(a) Slab decomposition, physical space (b) Slab decomposition, modal space

Figure 3.1: Slab decomposition, MPI processes numbering

At this point the data are in modal space.
The slab decomposition requires one series of MPI communication among all MPI processes for each passagge for physical [modal] space to modal [physical] space. The way the domain is divided among all processes restricts the maximum number of MPI processes that can actually be used: using a N^3 domain limits the maximum number of processes to roughly N. In fact each MPI process must hold at least a plane of data ($N \times N \times 1$). This limit thus depends on the choice of the grid.
Using the slab decomposition method allows to reduce the number of MPI communications but poses a strong limit on the maximum number of MPI processes.

3.3 Pencil decomposition

The pencil decomposition strongly increases the limit on the maximum number of MPI processes that can be used at the cost of an increased number of MPI communication. With this strategy each MPI process holds all the data in one direction and part of the data in the other two directions. When passing from physical space to modal space, according to Figure 3.2, first a 1D Fourier transform is performed in the x direction, then there is a series of MPI communications such that each rank, after that, holds all the data in the y direction. Then, a 1D Fourier transform is performed in the y direction; again there is a series of MPI communication such that each MPI process then holds all data in the z direction. Finally a 1D Chebyshev transform is performed in the z direction. In order to pass from modal space to physical space, all the previous steps must be done in reverse order.
The pencil decomposition strategy thus strongly increases the maximum number of MPI processes that can be used: a domain with N^3 point can be parallelized up to roughly N^2 MPI processes (each MPI process hold $N \times 1 \times 1$ points with the highest number of MPI processes). The cost for this increasing in the maximum number of MPI processes is a higher number of MPI communication: now there are two series of MPI communications.
In the pencil domain decomposition case a doubly periodic Cartesian topology can be used to find out in a much easier way which are the MPI processes involved in the communications. For example, in the first MPI communication series only the MPI



(a) Pencil decomposition, physical space (b) Pencil decomposition, modal space

Figure 3.2: Pencil decomposition, MPI processes numbering

processes in the same $x-y$ plane exchange data, while in the second MPI communication series only the MPI processes belonging to the same $y-z$ plane exchange data.

3.4 Domain decomposition strategies benchmark

Some test cases were run on the Marconi A1 Broadwell partition to verify the scalability of the code. The Marconi A1 partition machine has 1 512 nodes, each one with 36 cores/node, for a total of 54 432 cores. Each node is made up of two socket; each one is an Intel Xeon E5-2697 v4 @2.3 GHz. The total amount of RAM memory available per node is 128 GB/node, but it is suggested to use up to ∼120 GB/node.
Two kind of scalability tests were run: a strong scalability analysis and a weak scalability one. In the strong scalability case the overall domain size is kept constant, while the number of MPI processes is increased. As the number of MPI processes increases, the load on each core is reduced (lower number of points per core). In the weak scalability case the load (number of points per core) is kept constant while the overall number of points is increased according to the increase in the number of MPI processes.
For the strong scalability case three different grids were tested both for slab and pencil domain decomposition: $512 \times 256 \times 257$, $512 \times 512 \times 513$ and $1024 \times 1024 \times 1025$. As commonly found in literature the speed-up obtained is normalized by the speed-up obtained on the lowest number of MPI processes used; in this case the lowest number was 64 MPI processes. The ideal behaviour is linear in the total number of MPI processes; as can be seen from Figure 3.3 the slab decomposition runs deviate quite early from the ideal case, while the pencil cases keep an optimal speed-up at least up to 1 024 MPI processes. At this point only the larger grid does not show worsening in the performance, while the smaller grids show a larger deviation from the ideal case. Due to the limit on the maximum number of cores that can be requested on the Marconi A1 partition (around 6 000), we could not verify the scalability on a higher number of MPI processes.

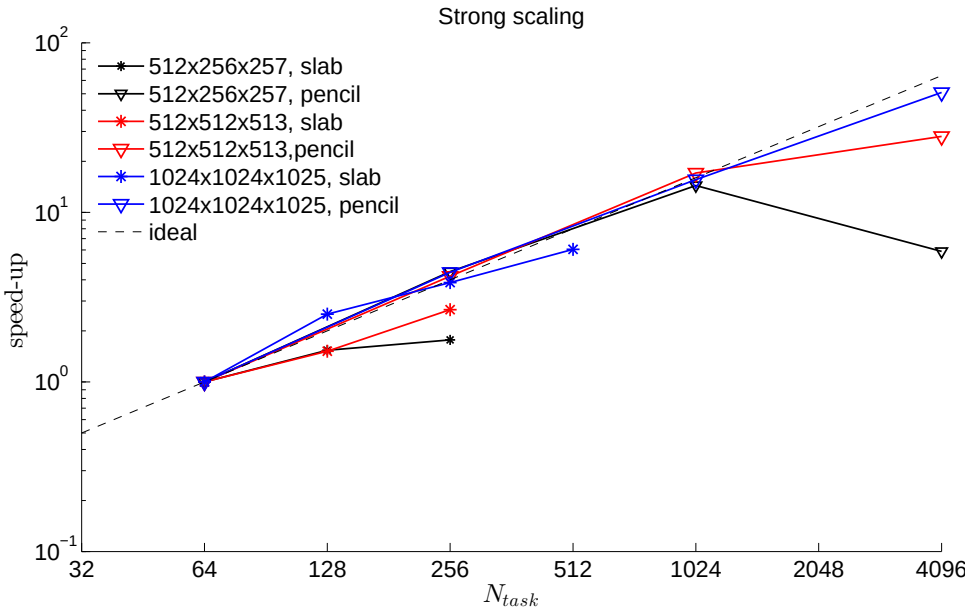


Figure 3.3: Strong scalability

Table 3.1: Strong scaling speed-up calculated with respect to 64 MPI processes case

$N_{y,cpu}$	$N_{z,cpu}$	grid	time per time step [s]	speed-up	ideal speed-up
1	64	$512 \times 256 \times 257$	1.22	1	1
1	128	$512 \times 256 \times 257$	0.79	1.54	2
1	256	$512 \times 256 \times 257$	0.69	1.77	4
8	8	$512 \times 256 \times 257$	1.44	1	1
16	16	$512 \times 256 \times 257$	0.32	4.5	4
32	32	$512 \times 256 \times 257$	0.10	14.4	16
64	64	$512 \times 256 \times 257$	0.25	5.76	64
1	64	$512 \times 512 \times 513$	4.82	1	1
1	128	$512 \times 512 \times 513$	3.18	1.52	2
1	256	$512 \times 512 \times 513$	1.81	2.66	4
8	8	$512 \times 512 \times 513$	6.64	1	1
16	16	$512 \times 512 \times 513$	1.58	4.20	4
32	32	$512 \times 512 \times 513$	0.39	17.03	16
64	64	$512 \times 512 \times 513$	0.24	27.67	64
1	64	$1024 \times 1024 \times 1025$	55.72	1	1
1	128	$1024 \times 1024 \times 1025$	22.20	2.51	2
1	256	$1024 \times 1024 \times 1025$	14.46	3.85	4
1	512	$1024 \times 1024 \times 1025$	9.20	6.06	8
8	8	$1024 \times 1024 \times 1025$	62.40	1	1
16	16	$1024 \times 1024 \times 1025$	14.10	4.43	4
32	32	$1024 \times 1024 \times 1025$	4.00	15.6	16
64	64	$1024 \times 1024 \times 1025$	1.23	50.73	64

For the weak scalability case three different runs were run: in the first two cases denoted by $512 \times 8 \times 8$ and $1024 \times 16 \times 16$ the aspect ratio of the arrays was kept constant as the number of MPI processes increased. In the other case, denoted by 64^3

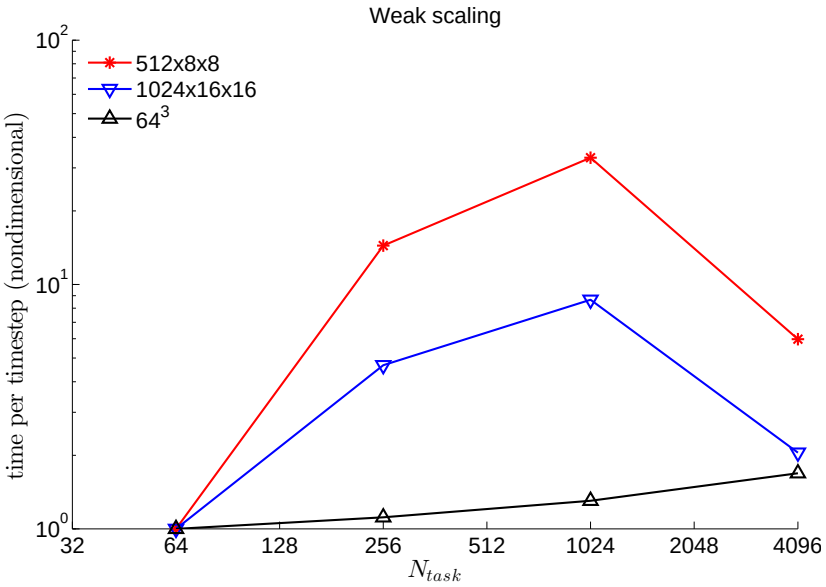


Figure 3.4: Weak scalability

only the total number of points per MPI process was kept constant. This allowed for a better data layout in memory, which gave much better results, only matched by the others two cases when the global grid approached a unitary aspect ratio in the three dimensions.

Table 3.2: Weak scaling time per time step normalized with respect to 64 MPI processes case

$N_{y,cpu}$	$N_{z,cpu}$	load/core	grid	t/t_{step} [s]	t/t_{step} normalized
8	8	$512 \times 8 \times 8$	$512 \times 64 \times 65$	0.06	1
16	16	$512 \times 8 \times 8$	$512 \times 128 \times 129$	0.81	14.46
32	32	$512 \times 8 \times 8$	$512 \times 256 \times 257$	1.85	33.04
64	64	$512 \times 8 \times 8$	$512 \times 512 \times 513$	0.34	6.07
8	8	$1024 \times 16 \times 16$	$1024 \times 128 \times 129$	0.54	1
16	16	$1024 \times 16 \times 16$	$1024 \times 256 \times 257$	2.50	4.66
32	32	$1024 \times 16 \times 16$	$1024 \times 512 \times 513$	4.64	8.66
64	64	$1024 \times 16 \times 16$	$1024 \times 1024 \times 1025$	1.10	2.05
8	8	64^3	$256 \times 256 \times 257$	0.65	1
16	16	64^3	$512 \times 512 \times 257$	0.73	1.12
32	32	64^3	$512 \times 512 \times 1025$	0.85	1.30
64	64	64^3	$1024 \times 1024 \times 1025$	1.10	1.69

3.5 GPU-Acceleration

When using a GPU-accelerated cluster, the parallelization backbone of the code still relies on a MPI approach; the overall workload is divided among the different MPI tasks using a 2D domain decomposition. On top of the MPI parallelization scheme, CUDA Fortran instructions and OpenACC directives are used to accelerate the code execution. Each MPI task is assigned to a specific GPU and thus to a specific pencil of the domain. All the computationally intensive operations are performed on the GPUs. Specifically, the

Nvidia cuFFT libraries are used to perform all the transforms (Fourier and Chebyshev) and the entire solver can be efficiently executed on the GPU thanks to the fine-grain parallelism offered by the numerical scheme (series of 1D independent problems along the wall-normal direction). To limit as much as possible Device to Host (D2H) and Host to Device (H2D) communications, which may hamper code performance, the required MPI communications are performed exploiting the CUDA-awareness capabilities of the MPI libraries when available (e.g., when using MPI Spectrum, OpenMPI, MPICH, etc.). In this way, GPUDirect RDMA technologies can be used to avoid costly D2H and H2D synchronizations. The only library required by the GPU version of the code FLOW36 is the Nvidia cuFFT library, which is used to perform all the transforms (Fourier and Chebyshev transforms). This library is part of the standard software stack of the targeted machine (present inside the Nvidia hpc-sdk together with the CUDA-aware version of the MPI libraries). Additional details on the bottlenecks of the GPU version and further possible optimizations can found in the git repository of the code.

Please refer to github for a detailed report on the GPU-acceleration and its details (see the ISSUES section). Ask AR for details on the GPU version, bottlenecks and details of what can and cannot be done. The git version is a vanilla implementation using openACC (to mantain the code portable and keep the same structure). More performance can be achieved with simple modifications (not implemented at the moment to not break the portability).

3.6 Profiling

The CPU version of the code can be profiled with any of the tools available (TAU-C, Intel vTune, AMDuProf, etc.). AMDuProf is required when using AMD architectures (AMD Rome, Milan, Epic). The GPU version of the code can be profiled using the tools provided by Nvidia. For better tracking of the different activities, you can enable the NVTX module available at the end of module.f90. This module should be decommented (as it is not supported in other compilers). Naturally, NVTX is supported only by the nvfortran (ex-PGI compiler) and results can be visualized using Nvidia Nsight system. Support to the NVTX instructions should be enabled via specific compiler options. For details on NVTX module see: `https://github.com/maxcuda/NVTX_example`.

Bibliography

C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang. *Spectral Methods*. Springer, Berlin, Germany, 2006.

M. Frigo and S. G. Johnson. FFTW, 2017. URL `http://www.fftw.org/`.

N. Kasagi. DNS database, 2017. URL `http://thtlab.jp/`.

A. Prosperetti. Motion of two superposed viscous fluids. *Phys. Fluids*, 24(7):1217–1223, 1981.

List of Figures

List of Tables