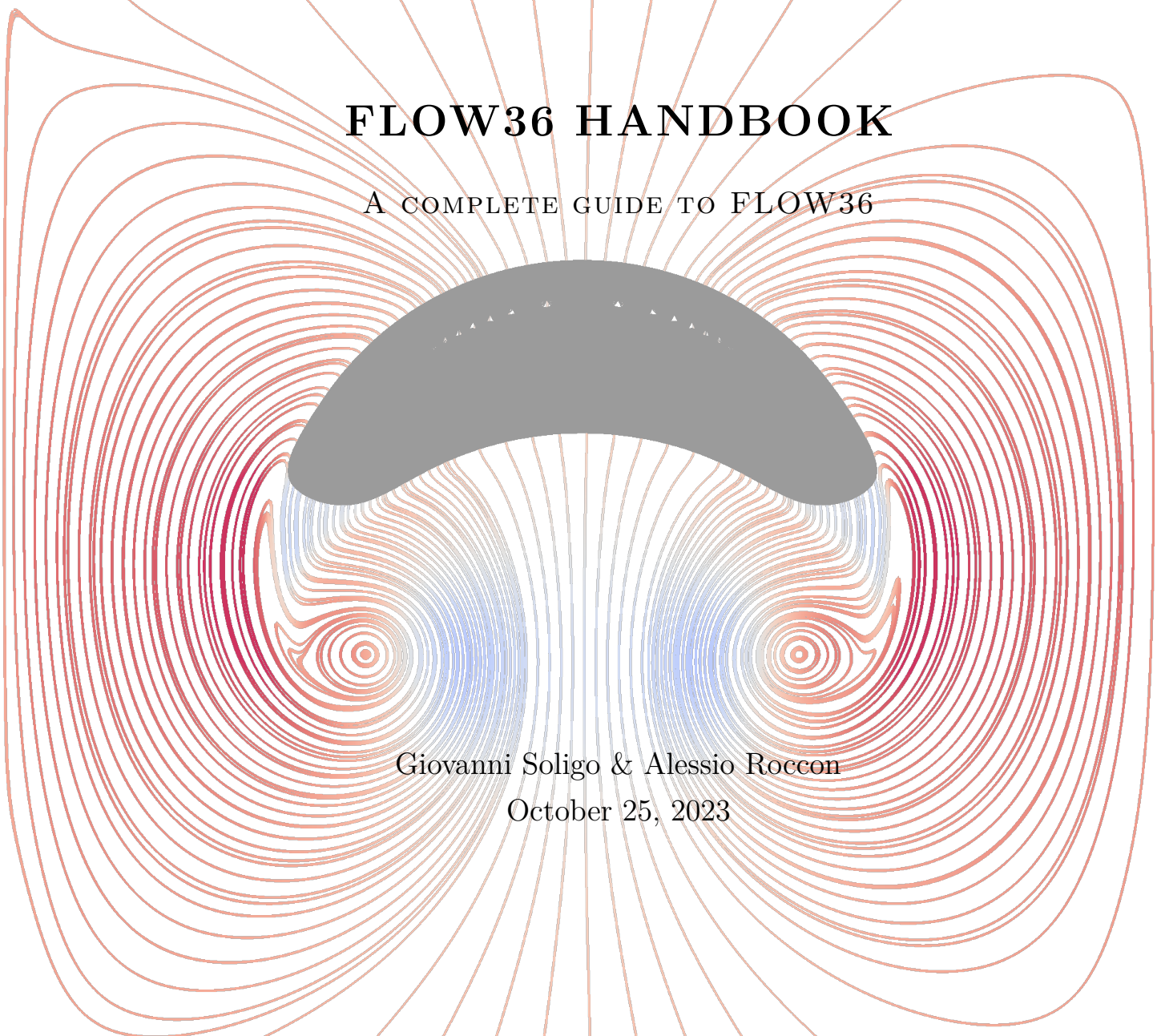
FLOW36 HANDBOOK

A COMPLETE GUIDE TO FLOW36



Giovanni Soligo & Alessio Roccon

October 25, 2023

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 performs Direct Numerical Simulations (DNS) in a channel geometry, using a pseudo-spectral spatial discretization. The flow solver is coupled with a phase variable solver, which solves the Cahn–Hilliard equation (Phase Field Model). The code was further developed during the MSCA-ITN-EID project COMETE "Next-Generation Computational Methods for Enhanced Multiphase Flow Processes" (No 813948) to simulate turbulent three-phase flows and turbulent bubbly-laden flows.

This document was last updated on October 25, 2023.

Contents

Chapter 1

Getting started

The executable `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` : contains the initial fields (velocity and phase, if dealing with a multiphase simulation) 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` : 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, if it is 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).
In addition the output of the simulation includes:

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

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 accelerationo.
- `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.

- `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 4). 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).
- `NNT` : total number of MPI processes used to run the code; equal to `NYCPU`×`NZCPU`.
- `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.
- `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:

  1. zero velocity all over the domain
  2. laminar Poiseuille flow in x direction (generated from a unitary pressure gradient)
  3. random velocity value for u, v, w

4. read input from file (parallel read)
5. read input from file (serial read, used for retro-compatibility with legacy data files)
6. ...

- `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).
- `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.
- `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 part is 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).
- `non_newtonian`: Enables the non-Newtonian-Carreau 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.
- `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).
- `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.
- `Pe` : value of Peclet number.
- `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

- `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.

- `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.
  - `part_flag`: Enables Lagrangian particle Tracking.

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 of all 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` 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` 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 differents initial condition that can be chosen.
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 mdule 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, using `include mpif.h` (deprecated) or `use mpi` (deprecated) or `use mpi f08`.

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 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).

1.4.2 Compiling for GPUs

To compile and run 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 and openMPI. Also, the only supported compiler is `nvfortran`.

1.5 Running the code

Regardless of the version compiled, to run the code the procedure 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 Troubleshotting

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 openMPI (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 suborutines 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 tranforms 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 library guide.

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. As shown in Figure 2.1, 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 4), 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 2.1.10). 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) and one for the particles (i.e. the MPI tasks taking care of tracking the particles).
At this point it comes to the plan creation for the FFTW calls: 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 $\cos\left[(k-1)\pi/(\texttt{NZ}-1)\right]$ with k spanning from 1 to `NZ`.
After the plan creation, 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).
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.

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. 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).

2.1.5 `dump_grid`

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

2.1.6 `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:

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

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

2.1.7 `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`. For the influence matrix method refer to Section 3.8.

2.1.8 `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 and to Chapter 3 for the equations solved.
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`.

2.1.8.1 `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.

2.1.8.2 `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 $(L_x/2, L_y/2, \texttt{height})$. The parameters `radius` and `height` must be specified in the `compile.sh` script.

2.1.8.3 `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`, ω_x =`wave_freq_x`, A_y =`wave_amp_y`, ω_y =`wave_freq_y` and A_r =`pert_amp` must be specified in the `compile.sh` script.

2.1.8.4 `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 $2(\texttt{radius} + 5\sqrt{2}\mathrm{Ch})$, 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.

2.1.9 `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.

2.1.10 `write_output`, `write_output_spectral` and `write_output_recovery`

These subroutines use MPI input/output 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.

2.1.11 `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).

2.1.12 `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 $\phi > 0$.

2.1.13 `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).

2.1.14 `destroy` and `destroy_phi`

These two subroutines take care of deallocating all the fields array: `destroy` deallocates all the velocity (both in physical and modal space) arrays, while `destroy_phi` deallocates all the phase field arrays.

2.1.15 `destroy_plan`

This subroutine frees the plan created for the FFTW execution.

2.2 Solver

This subroutine includes all the calls to the subroutines involved in the solution of the Navier–Stokes equations (and Cahn–Hilliard equation, if the phase field is activated). At the beginning of the subroutine the data at the current time step are provided and at the end of the subroutine these data are updated with those at the following time step.
At first the arrays of the non-linear terms for the Navier–Stokes equation `s1`, `s2` and `s3` are allocated and they are initialized to zero. After that 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 is activated, 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.
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`. Here the non-linear terms are also deallocated.
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.
At this point, 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 and the `solver` part concludes; the velocity and phase field variables have been updated with the new values.

2.2.1 `convective_ns`

This subroutine calculates the term $\left(1+(\phi+1)\frac{\alpha-1}{2}\right)\nabla(\mathbf{u}\otimes\mathbf{u})$. For the single phase case ϕ is not defined and α is equal to 1, so the first part is constant equal to 1.
The convective term is calculated as $\nabla(\mathbf{u}\otimes\mathbf{u})$ instead of $(\mathbf{u}\cdot\nabla)\mathbf{u}$ since it requires less transforms (both from physical to modal and from modal to physical) than the usual way, but here the hypothesis of $\nabla\cdot\mathbf{u}$ has been settled.

$$\nabla(\mathbf{u}\otimes\mathbf{u}) = \nabla\left(\begin{bmatrix}u\\v\\w\end{bmatrix}\begin{bmatrix}u & v & w\end{bmatrix}\right) = \begin{bmatrix}u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}+u\nabla\cdot\mathbf{u}\\ u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}+v\nabla\cdot\mathbf{u}\\ u\frac{\partial w}{\partial x}+v\frac{\partial w}{\partial y}+w\frac{\partial w}{\partial z}+w\nabla\cdot\mathbf{u}\end{bmatrix}$$

2.2.2 `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`.

2.2.3 `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.

2.2.4 `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.

2.2.5 `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.

2.2.6 `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.

2.2.7 `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.

2.2.8 `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).

2.2.9 `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.

2.2.10 `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.

2.2.11 `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.

2.2.12 `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.

2.2.13 `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.

2.2.14 `sterm_temp`

This subroutine calculates the non-linear term of the energy equation for the temperature. As the energy equation is only a simple advection diffusion equation, this terms is basically:

$$S_\theta = -\mathbf{u} \cdot \nabla\theta$$

2.2.15 `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).

2.2.16 `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.

2.2.17 `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.

2.2.18 `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.

2.2.19 `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.

2.2.20 `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.

2.2.21 `dz_fg`

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

2.2.22 `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.

2.2.23 `helmholtz_fg`

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

2.2.24 `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.

2.3 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)
  - `one_s` : transform of a unit array in modal space (allocatable)

- `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
  - `plan_y_fwd` : plan for 1D Fourier transform, y direction
  - `plan_z_fwd` : plan for 1D Chebyshev transform, z direction
  - `plan_x_bwd` : plan for 1D inverse Fourier transform, x direction

  - `plan_y_bwd` : plan for 1D inverse Fourier transform, y direction
  - `plan_z_bwd` : plan for 1D inverse Chebyshev transform, z direction

- `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)

- `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 subcommunicator, used to exchange data among MPI processes in the same $x - y$ plane
  - `col_comm` : MPI Cartesian subcommunicator, used to exchange data among MPI processes that cover the same $x - y$ region (same “column”)

Figure 2.1: Scheme of main program `FLOW_36`. xxx stands for a generic field (e.g. phase-field, surfactant field, temperature)

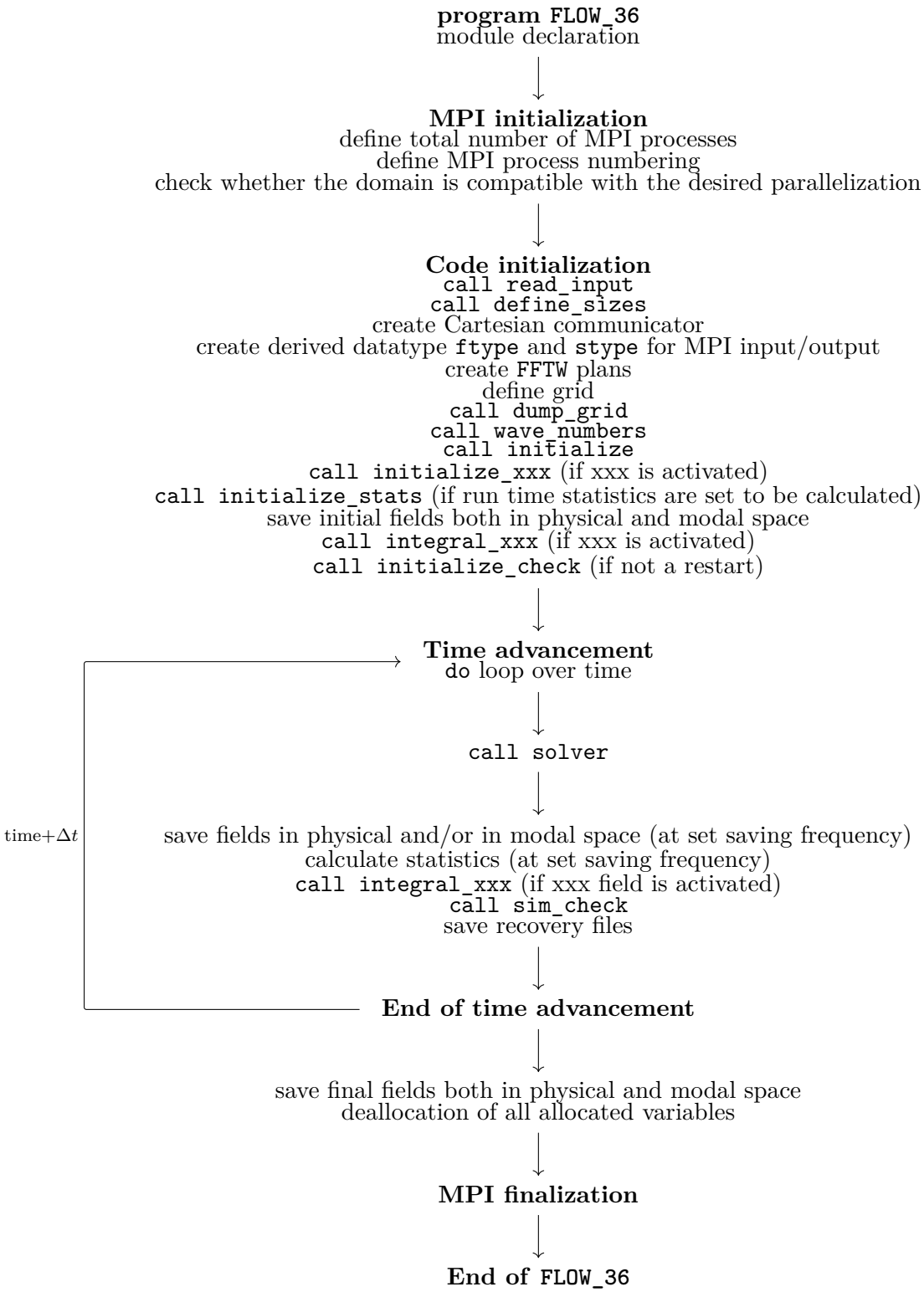
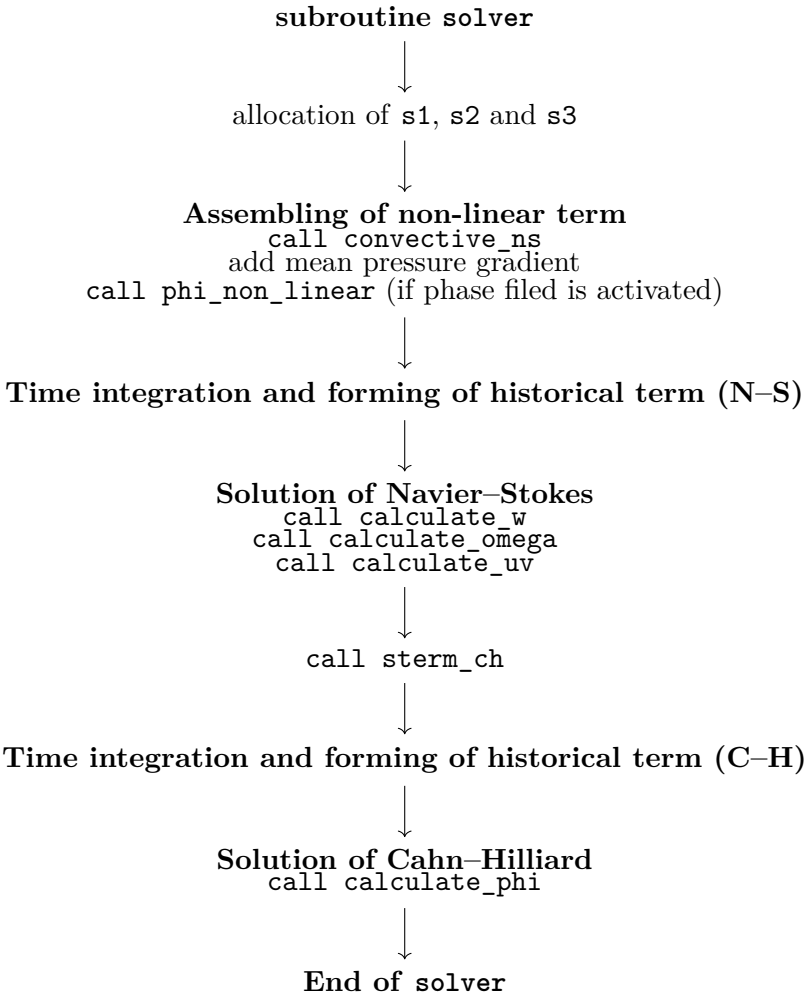


Figure 2.2: Scheme of subroutine `solver`



Chapter 3

Numerical method

3.1 Problem description

The code is meant to solve a turbulent channel flow, either a closed channel case, either an open channel case; the basic formulation includes the incompressible Navier–Stokes equations, that can be coupled in the full formulation with the Cahn–Hilliard equation. In Figure 3.1 the geometry of the channel is presented together with the reference frame used. The domain size is $n\pi h \times m\pi h \times 2h$ ($x \times y \times z$).

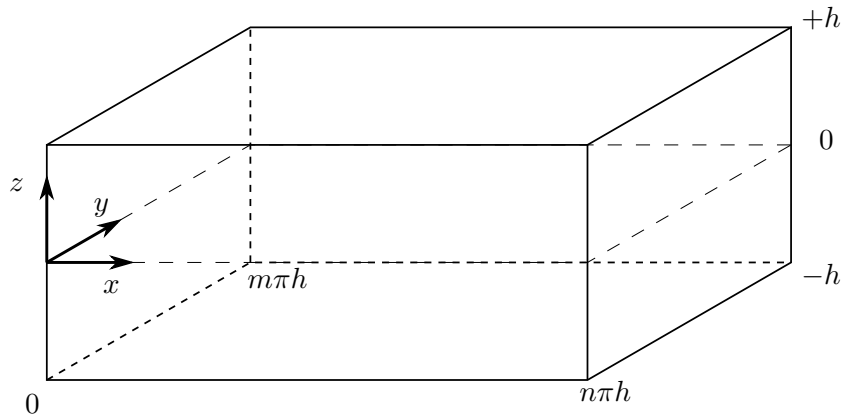


Figure 3.1: Sketch of the domain

The x and y directions are periodic directions, while $z = +h$ and $z = -h$ can be either a solid boundary, either a free-slip boundary, depending on the choice of the boundary conditions.

3.2 Single phase flow

For the single phase case (phase field deactivated) the incompressible Navier–Stokes equations are solved. The equations are made non-dimensional using the shear velocity u_τ and the channel half height h. The Reynolds number is defined as:

$$\mathrm{Re}_\tau = \frac{\rho h u_\tau}{\mu}$$

where ρ is the fluid density and μ the dynamic viscosity.
The dimensionless incompressible Navier–Stokes equations reads:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \dfrac{1}{\mathrm{Re}_\tau} \bar{\nabla}^2 \mathbf{u} \end{cases} \tag{3.1}$$

The pressure gradient is then split in two part, a mean pressure gradient and a fluctuating part:

$$\nabla p = \Pi + \nabla p'$$

This splitting is done in order to remove the pressure as a problem unknown: the code solves the Navier–Stokes equation using the velocity–vorticity formulation, which means that the four variables are the three velocity components and the wall-normal component of the vorticity.
The velocity–vorticity formulation solves a second order equation for the transport of vorticity and a fourth order equation for the wall-normal velocity; streamwise and spanwise velocity components are calculated from the continuity equation and the definition of the wall-normal vorticity. Before starting to derive all the equations, the Navier–Stokes equation will be rewritten in a more compact form:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \dfrac{\partial \mathbf{u}}{\partial t} = \mathbf{S} - \nabla p' + \dfrac{1}{\mathrm{Re}_\tau} \bar{\nabla}^2 \mathbf{u} \end{cases}$$

The $\mathbf{S}$ term is the non-linear term of the Navier–Stokes equations (in the code its components are `s1`, `s2` and `s3`) and for a single phase flow is defined as:

$$\mathbf{S} = -\mathbf{u} \cdot \nabla \mathbf{u} - \Pi$$

The second oreder equation for the transport of the vorticity is obtained by taking the curl of the Navier–Stokes equations; the fourth order equation for the transport of wall-normal velocity is obtained by taking again the curl of the Navier–Stokes equations (basically, taking two times the curl of the Navier–Stokes equations). The full set of equations includes (in order): the second order equation for vorticity, the fourth order equation for wall-normal velocity, the continuity equation and the definition of wall-normal vorticity.

$$\begin{cases} \dfrac{\partial \omega}{\partial t} = \nabla \times \mathbf{S} + \dfrac{1}{\mathrm{Re}_\tau} \bar{\nabla}^2 \omega & \text{only } z \text{ component} \\ \dfrac{\partial (\bar{\nabla}^2 \mathbf{u})}{\partial t} = \bar{\nabla}^2 \mathbf{S} - \nabla(\nabla \cdot \mathbf{S}) + \dfrac{1}{\mathrm{Re}_\tau} \bar{\nabla}^4 \mathbf{u} & \text{only } z \text{ component} \\ \dfrac{\partial u}{\partial x} + \dfrac{\partial v}{\partial y} + \dfrac{\partial w}{\partial z} = 0 \\ \omega_z = \dfrac{\partial v}{\partial x} - \dfrac{\partial u}{\partial y} \end{cases} \tag{3.2}$$

The system of equations 3.2 is discretized in the code using a pseudospectral method and solved in modal space; here discrete Fourier transforms are used in the periodic directions (x and y), while discrete Chebyshev transforms in the wall-normal direction

(z). A detailed explanation of the pseudospectral discretization will be given in Section 3.5, here only the results will be used.
The discretized system of equation in modal space reads:

$$\begin{cases} \dfrac{\partial \hat{\omega}_z}{\partial t} = \mathrm{i}k_x\hat{S}_2 - \mathrm{i}k_y\hat{S}_1 + \dfrac{1}{\mathrm{Re}_\tau}\left(\dfrac{\partial^2\hat{\omega}_z}{\partial z^2} - k^2\hat{\omega}_z\right) \\ \dfrac{\partial}{\partial t}\left(\dfrac{\partial^2\hat{w}}{\partial z^2} - k^2\hat{w}\right) = -k^2\hat{S}_3 - \mathrm{i}k_x\dfrac{\partial\hat{S}_1}{\partial z} - \mathrm{i}k_y\dfrac{\partial\hat{S}_2}{\partial z} + \dfrac{1}{\mathrm{Re}_\tau}\left(k^4\hat{w} + \dfrac{\partial^4\hat{w}}{\partial z^4} - 2k^2\dfrac{\partial^2\hat{w}}{\partial z^2}\right) \\ \mathrm{i}k_x\hat{u} + \mathrm{i}k_y\hat{v} + \dfrac{\partial\hat{w}}{\partial z} = 0 \\ \hat{\omega}_z = \mathrm{i}k_x\hat{v} - \mathrm{i}k_y\hat{u} \end{cases} \tag{3.3}$$

The coefficient k^2 is given by $k^2 = k_x^2 + k_y^2$.
The system of equations 3.3 is then discretized in time using a Crank–Nicolson scheme for the implicit part and an Adams–Bashforth for the explicit one (complete details on the time integration are given in Section 3.6).
A more compact notation can be introduced with the historical terms H_i^n with $i = 1, 2, 3$ and H^n.

$$H_i^n = \Delta t\left(\frac{3}{2}\hat{S}_i^n - \frac{1}{2}\hat{S}_i^{n-1} + \frac{1}{2\mathrm{Re}_\tau}\frac{\partial^2\hat{u}_i^n}{\partial z^2} + \left(\frac{1}{\Delta t} - \frac{k^2}{2\mathrm{Re}_\tau}\right)\hat{u}_i^n\right) \qquad \text{for } i = 1, 2, 3$$

$$H^n = \frac{\partial}{\partial z}\left(\mathrm{i}k_xH_1^n + \mathrm{i}k_yH_2^n\right) + k^2H_3^n$$

The superscript n denotes the current time step, $n+1$ is the time step for which the unknowns have to be calculated.
Once defined the historical terms, the time and space discretized system of equation can be rewritten in a more compact form, highlighting the Helmholtz problems:

$$\begin{cases} \left(\dfrac{\partial^2}{\partial z^2} - \beta^2\right)\left(\dfrac{\partial^2}{\partial z^2} - k^2\right)\hat{w}^{n+1} = \dfrac{H^n}{\gamma} \\ \left(\dfrac{\partial^2}{\partial z^2} - \beta^2\right)\hat{\omega}_z^{n+1} = -\dfrac{\mathrm{i}k_xH_2^n - \mathrm{i}k_yH_1^n}{\gamma} \\ \mathrm{i}k_x\hat{u} + \mathrm{i}k_y\hat{v} + \dfrac{\partial\hat{w}}{\partial z} = 0 \\ \hat{\omega}_z = \mathrm{i}k_x\hat{v} - \mathrm{i}k_y\hat{u} \end{cases} \tag{3.4}$$

The coefficients γ and β^2 are defined as:

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

$$\beta^2 = \frac{1 + \gamma k^2}{\gamma}$$

The values of $\hat{w}$ and $\hat{\omega}_z$ can be obtained using the Chebyshev–Tau method and the influence matrix method. For the details on the resolution of these two variables, please refer to Sections 3.7 and 3.8.

Once the values of $\hat{w}^{n+1}$ and $\hat{\omega}_z^{n+1}$ have been calculated, the continuity equation and the definition of wall-normal vorticity can be used to obtain the values of $\hat{u}^{n+1}$ and $\hat{v}^{n+1}$.

$$\begin{bmatrix} -\mathrm{i}k_y & \mathrm{i}k_x \\ \mathrm{i}k_x & \mathrm{i}k_y \end{bmatrix} \begin{bmatrix} \hat{u}^{n+1} \\ \hat{v}^{n+1} \end{bmatrix} = \begin{bmatrix} \hat{\omega}_z^{n+1} \\ -\dfrac{\partial \hat{w}^{n+1}}{\partial z} \end{bmatrix}$$

This system of equation can be solved unless the determinant of the coefficient matrix is zero. In that case a different path must be sought.
Starting from the fourth order equation for the velocity (vectorial equation) for the x and y components and with the hypothesis of $k_x = 0$ and $k_y = 0$ we obtain two fourth order equations for $\hat{u}$ and $\hat{v}$:

$$\begin{cases} \dfrac{\partial^2 \hat{u}}{\partial z^2} - \dfrac{\hat{u}}{\gamma} = -\dfrac{H_1}{\gamma} \\ \dfrac{\partial^2 \hat{v}}{\partial z^2} - \dfrac{\hat{v}}{\gamma} = -\dfrac{H_2}{\gamma} \end{cases} \tag{3.5}$$

Solving these two Helmholtz equations gives the solution for $\hat{u}$ and $\hat{w}$ for $k_x = k_y = 0$.

3.3 Phase Field Model

If the phase field is activated in the code, an additional phase is included in the computations. This multiphase system is solved using the Phase Field Model. The interface between the two phases is a diffuse interface: all the variables varies smoothly across the interface, following an hyperbolic tangent profile. This approach is the opposite of the sharp interface one, where the interface is seen as a discontinuity and jump conditions across the interface are imposed on the variables. With the diffuse interface approach there is no need to introduce any jump condition across the interface, as the interface and the variables across the interface are resolved.
With the introduction of the Phase Field Model an additional variable is introduced: the phase field ϕ. The advection of this new variable is calculated by the Cahn–Hilliard equation.
In this code the modified H model is implemented; this model allows the use of phases with different densities and viscosities, but still keeping the hypothesis of divergency free flow field. This hypothesis is exact in the bulk phases, but it is not correct at the interface; on the other hand the interface is a very limited portion of the whole domain. Another hypothesis of the modified H model is to neglect the density gradients; as before, this hypothesis is exact in the bulk phases. The variable density and viscosity are a function of the phase field variable ϕ.
The introduction of the Phase Field Model introduces some new terms in the Navier–Stokes equations:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \rho \dfrac{\partial \mathbf{u}}{\partial t} + \rho \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \dfrac{1}{\mathrm{Re}_\tau} \left(\mu (\nabla \mathbf{u} + \nabla \mathbf{u}^T) \right) + \dfrac{1}{\mathrm{Fr}^2} \rho \mathbf{g} + \dfrac{3}{\sqrt{8}} \dfrac{1}{\mathrm{WeCh}} \kappa \nabla \phi \end{cases} \tag{3.6}$$

κ is the chemical potential and is defined as $\kappa = \phi^3 - \phi - \mathrm{Ch}^2 \nabla^2 \phi$, Fr is the Froud number, $\mathbf{g}$ is the gravity versor, We is the Weber number and Ch is the Cahn number.

This quantities are defined as follows:

$$\mathrm{Fr} = \frac{u_\tau}{\sqrt{gh}}$$

$$\mathrm{We} = \frac{\widetilde{\rho} u_\tau^2 h}{\widetilde{\sigma}}$$

$$\mathrm{Ch} = \frac{\varepsilon}{h}$$

$\widetilde{\sigma}$ is the dimensional surface tension, ε is a measure of the interface thickness and h is the channel half height.
Density and viscosity are made dimensionless using the values of the phase with $\phi = -1$.

$$\rho = 1 + \frac{\alpha - 1}{2}(\phi + 1)$$

$$\mu = 1 + \frac{\beta - 1}{2}(\phi + 1)$$

α and β are the density and the viscosity ratio, respectively.

$$\alpha = \frac{\rho_{\phi=+1}}{\rho_{\phi=-1}} = \frac{\rho_+}{\rho_-}$$

$$\beta = \frac{\mu_{\phi=+1}}{\mu_{\phi=-1}} = \frac{\mu_+}{\mu_-}$$

It can be seen that ρ and μ can be split easily in a constant part and in a ϕ dependent part.
With the introduction of phase field and this definition for the density and the viscosity, the Navier–Stokes solving procedure is unchanged: the only change is in the non-linear term, to which other parts are added. The new Navier–Stokes non-linear term thus results in:

$$\begin{aligned}\mathbf{S} = &- \frac{\alpha - 1}{2}(\phi + 1)\frac{\partial \mathbf{u}}{\partial t} - \left(1 + \frac{\alpha - 1}{2}(\phi + 1)\right)\mathbf{u} \cdot \nabla \mathbf{u} - \Pi + \\ &+ \frac{1}{\mathrm{Re}_\tau}\nabla \cdot \left(\frac{\beta - 1}{2}(\phi + 1)(\nabla \mathbf{u} + \nabla \mathbf{u}^T)\right) + \\ &+ \frac{1}{\mathrm{Fr}^2}\left(1 + \frac{\alpha - 1}{2}(\phi + 1)\right)\mathbf{g} + \frac{3}{\sqrt{8}}\frac{1}{\mathrm{WeCh}}\kappa \nabla \phi\end{aligned}$$

For the the matched densities case α is equal to 1, so the time derivative term, the gravity and buoyancy terms and part of the convective terms vanish; for the matched viscosities case β is equal to 1 and the viscous term in $\mathbf{S}$ vanishes.
Thus, for the case of two phases with matched densities and viscosities the only added term to the original Navier–Stokes non-linear term is the surface force term.
Once intrduced these new terms in the non-linear part of the equation, the solution algorithm is the same for the single phase case.

3.3.1 Cahn–Hilliard equation

The Cahn–Hilliard equation describes the advection of the phase field variable ϕ in the domain; the interface is advected by the flow field $\mathbf{u}$ and the shape of the interface is kept by the chemical potential κ.

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \frac{1}{\mathrm{Pe}}\nabla(M \nabla \kappa) \tag{3.7}$$

Here M is the dimensionless mobility coefficient; in the code M is assumed constant and equal to one (once made dimensionless):

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \frac{1}{\mathrm{Pe}} \nabla^2 \kappa$$

The Peclet number is defined as:

$$\mathrm{Pe} = \frac{u_\tau h}{\widetilde{M} \widetilde{\kappa}}$$

where $\widetilde{M}$ is the dimensional mobility coefficient.
Inserting the expression for the chemical potential in the Cahn–Hilliard equation, a fourth order equation for the phase variable is obtained:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \frac{1}{\mathrm{Pe}} (\nabla^2 \phi^3 - \nabla^2 \phi - \mathrm{Ch}^2 \nabla^4 \phi)$$

To increase the stability of the numerical solution, the laplacian term is split in two parts, one that will be dealt with implicitly and the other explicitly:

$$-\nabla^2 \phi = s \nabla^2 \phi - (1 + s) \nabla^2 \phi$$

The s coefficient is defined as:

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

At this point the Cahn–Hilliard non-linear term can be highlighted:

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

Thus, the Cahn–Hilliard equation in compact form reads:

$$\frac{\partial \phi}{\partial t} = S_\phi + \frac{s}{\mathrm{Pe}} \nabla^2 \phi - \frac{\mathrm{Ch}^2}{\mathrm{Pe}} \nabla^4 \phi$$

After the spatial discretization, the equation in modal space is:

$$\frac{\partial \hat{\phi}}{\partial t} = \hat{S}_\phi + \left(\frac{\partial^2}{\partial z^2} - k^2 \right) \left[\frac{s}{\mathrm{Pe}} - \frac{\mathrm{Ch}^2}{\mathrm{Pe}} \left(\frac{\partial^2}{\partial z^2} - k^2 \right) \right] \hat{\phi}$$

For the time discretization an implicit Euler method is used for the implicit part, while for the explicit part an Adams–Bashforth method is used, except for the first time step, where an explicit Euler algorithm is used.
The following fourth order equation for the phase field is then obtained:

$$\left(\frac{\partial^2}{\partial z^2} - k^2 - \frac{s}{2\mathrm{Ch}^2} \right) \left(\frac{\partial^2}{\partial z^2} - k^2 - \frac{s}{2\mathrm{Ch}^2} \right) \hat{\phi} = \frac{H_\phi^n}{\gamma} \tag{3.8}$$

The historical term H_ϕ^n is defined as:

$$H_\phi^n = \frac{\Delta t}{2} (3 \hat{S}^n - \hat{S}^{n-1}) + \hat{\phi}^n$$

Equation3.8 can be split in two Helmholtz equation, each of them has its own known boundary conditions, as the boundary conditions for the phase field are on the first and third derivative.

$$\begin{cases} \left(\dfrac{\partial^2}{\partial z^2} - k^2 - \dfrac{s}{2\mathrm{Ch}^2}\right)\hat{\phi}^{n+1} = \theta \\ \left.\dfrac{\partial \hat{\phi}^{n+1}}{\partial z}\right|_{z=\pm 1} = 0 \end{cases} \qquad \begin{cases} \left(\dfrac{\partial^2}{\partial z^2} - k^2 - \dfrac{s}{2\mathrm{Ch}^2}\right)\theta = \dfrac{H_\phi^n}{\gamma} \\ \left.\dfrac{\partial \theta}{\partial z}\right|_{z=\pm 1} = 0 \end{cases} \tag{3.9}$$

First, the Helmholtz equation for the auxiliary variable θ is solved, then the value of the phase field is obtained from the other Helmholtz problem.

3.4 Boundary conditions

The boundary conditions on the phase field do not depend on the type of boundary (open or closed boundary) and they are:

$$\begin{cases} \left.\dfrac{\partial \phi}{\partial z}\right|_{z_b} = 0 \\ \left.\dfrac{\partial^3 \phi}{\partial z^3}\right|_{z_b} = 0 \end{cases}$$

On the other hand, the boundary conditions on the velocity and on the vorticity depend on the type of boundaries: In the code two different cases can be chosen: two solid boundaries (closed channel case) or one solid boundary and one open boundary (open channel case).

3.4.1 No–slip condition

The no–slip condition is enforced whenever there is a solid wall: in this case $u = v = w = 0$ for each x, y at the boundary. This way the continuity equation yelds to:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = \frac{\partial w}{\partial z} = 0$$

From the vorticity definition it results:

$$\omega_z = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = 0$$

3.4.2 Free–slip condition

For the free–slip condition $w = 0$ at the boundary, while the derivatives of u and v along z are zero (free–slip means that there is no shear stress at the boundary).

The derivative in the wall-normal direction of the continuity equation yelds:

$$\frac{\partial^2 u}{\partial x \partial z} + \frac{\partial^2 v}{\partial y \partial z} + \frac{\partial^2 w}{\partial z^2} = \frac{\partial}{\partial x}\frac{\partial u}{\partial z} + \frac{\partial}{\partial y}\frac{\partial v}{\partial z} + \frac{\partial^2 w}{\partial z^2} = \frac{\partial^2 w}{\partial z^2} = 0$$

Taking the derivative of the vorticity equation along the z direction yelds:

$$\frac{\partial \omega_z}{\partial z} = \frac{\partial^2 v}{\partial x \partial z} - \frac{\partial^2 u}{\partial y \partial z} = \frac{\partial}{\partial x}\frac{\partial v}{\partial z} - \frac{\partial}{\partial y}\frac{\partial u}{\partial z} = 0$$

3.4.3 Recap of the boundary conditions for the Navier–Stokes equation

Table 3.1: Boundary conditions

type	**velocity**	**vorticity**
no–slip	$w(z_b) = 0\ ,\ \left.\frac{\partial w}{\partial z}\right\|_{z_b} = 0$	$\omega_z(z_b) = 0$
free–slip	$w(z_b) = 0\ ,\ \left.\frac{\partial^2 w}{\partial z^2}\right\|_{z_b} = 0$	$\left.\frac{\partial \omega_z}{\partial z}\right\|_{z_b} = 0$

3.4.4 Closed channel and open channel

The code uses the non-dimensionaliztion for the closed channel case: the shear velocity are different for the open channel and the closed channel cases. The shear velocity is defined as:

$$u_c = u_\tau = \sqrt{\frac{\tau_w}{\rho}}$$

where τ_w is the shear stress at the wall. Its value can be easily obtained from a force balance and depends on the geometry studied:

- Open channel:

$$\tau_w L_x L_y = \Delta\bar{p} L_y 2h$$

$$\tau_w = 2\frac{\Delta\bar{p}}{L_x}h$$

- Closed channel:

$$2\tau_w L_x L_y = \Delta\bar{p} L_y 2h$$

$$\tau_w = \frac{\Delta\bar{p}}{L_x}h$$

$\Delta\bar{p}$ is the time and space averaged pressure gradient in the flow direction x. This mean component is Π as seen in the previous sections, since we apply the following splitting to the pressure term:

$$\nabla p = \nabla\bar{p} + \nabla p' = \Pi + \nabla p'$$

This way the values of the wall shear stress can be rewritten as $\tau_w = 2\Pi h$ for the open channel case and as $\tau_w = \Pi h$ for the closed channel case.
Thus, for the open channel, the shear velocity is:

$$u_\tau^{oc} = \sqrt{\frac{2\Pi h}{\rho}} = \sqrt{2} u_\tau^{cc} \tag{3.10}$$

while for the closed channel it is:

$$u_\tau^{cc} = \sqrt{\frac{\Pi h}{\rho}} \tag{3.11}$$

Since the definition of the shear Reynolds number is unique in the code and is:

$$\mathrm{Re}_\tau = \mathrm{Re}_\tau^{cc} = \frac{\rho u_\tau^{cc} h}{\mu}$$

when performing an open channel simulation the input parameter in the `compile.sh` script is the closed channel shear Reynolds number. The actual shear Reynolds number for the open channel case is:

$$\mathrm{Re}_\tau^{oc} = \frac{\rho u_\tau^{oc} 2h}{\mu} = 2\sqrt{2}\frac{\rho u_\tau^{cc} h}{\mu} = 2\sqrt{2}\mathrm{Re}_\tau^{cc}$$

Usually, for the open channel case, the Reynolds number is defined on the channel height, not on the half channel height as done for the closed channel case.

3.5 Pseudospectral spatial discretization

The grid is uniform in the x and y directions, while for the z direction the Chebyshev Gauss-Lobatto points are used. The grid points are thus defined as follows:

$$\begin{cases} x_i = \dfrac{i-1}{N_x - 1} L_x & i = 1, \ldots, N_x \\ y_j = \dfrac{j-1}{N_y - 1} L_y & j = 1, \ldots, N_y \\ z_k = \cos\left(\dfrac{(k-1)\pi}{N_z - 1}\right) & k = 1, \ldots, N_z \end{cases}$$

The use of Fourier discretization in the x and y directions implicitly forces a periodic boundary condition on the corresponding boundaries. Since in the wall-normal direction a periodic boundary condition can not be applied, Chebyshev polynomials are used to discretize variables in that direction.
For the Fourier transforms in the two directions two sets of wave numbers can be defined, k_x for the x direction and k_y for the y direction. These wave numbers are directly used in the transforms.

$$k_x(i) = \frac{2(i-1)\pi}{L_x} \quad \text{with } i = 1, \ldots, N_x/2 + 1$$

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

A generic variable $f(x, y, z, t)$ in physical space can be represented in the modal space as a function of the wave numbers and of the Chebyshev polynomials (truncated series). The coefficient $\hat{f}(k_x, k_y, k, t)$ represents the Fourier coefficient, while T_k is the k^{th} Chebyshev polynomial.

$$f(x, y, z, t) = \sum_{i=0}^{N_x/2} \sum_{j=-N_y/2+1}^{N_y/2} \sum_{k=0}^{N_z-1} \hat{f}(k_x, k_y, k, t) T_k(z) e^{\mathrm{i}(k_x x + k_y y)}$$

The adoption of a modal representation of the variables allows the exact calculation of the derivatives in the three directions. For the Fourier directions the derivative can be easily calculated by multiplying the variable for the imaginary unit i times the corresponding direction's wave numbers:

$$\frac{\partial f(x, y, z, t)}{\partial x} = \sum_{i=0}^{N_x/2} \sum_{j=-N_y/2+1}^{N_y/2} \sum_{k=0}^{N_z-1} \hat{f}(k_x, k_y, k, t) T_k(z) \mathrm{i} k_x e^{\mathrm{i}(k_x x + k_y y)}$$

$$\frac{\partial f(x, y, z, t)}{\partial y} = \sum_{i=0}^{N_x/2} \sum_{j=-N_y/2+1}^{N_y/2} \sum_{k=0}^{N_z-1} \hat{f}(k_x, k_y, k, t) T_k(z) \mathrm{i} k_y e^{\mathrm{i}(k_x x + k_y y)}$$

For the wall-normal derivatives the derivative is not so immediate: to obtain the exact value the recursive relationship on the Chebyshev polynomials and their derivatives for the Chebyshev Gauss-Lobatto points must be exploited.
First of all, the Chebyshev polynomials are defined as:

$$\begin{aligned} T_0(z) &= 1 \\ T_1(z) &= z \\ T_n(z) &= 2zT_{n-1}(z) - T_{n-2}(z) \end{aligned}$$

Their derivatives are recursively defined as:

$$\begin{aligned} \frac{\partial T_0(z)}{\partial z} &= 0 \\ \frac{\partial T_1(z)}{\partial z} &= 1 \\ \frac{\partial T_n(z)}{\partial z} &= \frac{\partial T_{n-2}}{\partial z} + 2nT_{n-1} \end{aligned}$$

This way all spatial derivatives are exact and can be directly taken in modal space; this does not mean that there is no discretization error, which is actually introduced when truncating the infinite Fourier and Chebyshev series to a finite sum of interpolating functions.

3.6 Time discretization

The time integration algorithm follows an implicit/explicit (IMEX) scheme. The non-linear term are discretized in time using either an explicit Euler scheme (for the first time step only) either an Adams–Bashforth scheme (from the second time step on). This is valid both for the Navier–Stokes and for the Cahn–Hilliard time discretization

of the (non-linear) explicit terms.

$$\begin{cases} \dfrac{u_{n+1}-u_n}{\Delta t} = F_n & \text{Explicit Euler} \\ \dfrac{u_{n+1}-u_n}{\Delta t} = \dfrac{3F_n - F_{n-1}}{2} & \text{Adams–Bashforth} \end{cases}$$

For the implicit part two different algorithms are used: for the Navier–Stokes equations the Crank–Nicolson algorithm is used, while for the Cahn–Hilliard equation the more dissipative impicit Euler is used.

$$\begin{cases} \dfrac{u_{n+1}-u_n}{\Delta t} = F_{n+1} & \text{Implicit Euler} \\ \dfrac{u_{n+1}-u_n}{\Delta t} = \dfrac{F_{n+1}+F_n}{2} & \text{Crank–Nicolson} \end{cases}$$

3.6.1 Dealiasing

Being a pseudospectral code, the products between variables are evaluated in physical space. To avoid the arising of aliasing when transforming variables back to modal space, dealiasing must be performed on the variables.
In the code the dealiasing is performed directly by the subroutines `phys_to_spectral` and `spectral_to_phys` by the flag `aliasing`. If this flag is set equal to one, then dealiasing is performed; dealiasing occurs at the end of the subroutines that perform Fourier or Chebyshev transforms and at the beginning of the subroutines the perform inverse Fourier and Chebyshev transforms.
The code follows the 2/3 rule (see Canuto et al. (2006) for a detailed reference) for dealiasing. According to the 2/3 rule, all the modes with $|k|$ greater then 2/3 of N (where k is the mode number and N is the total number of modes) are set to zero.
Due to the particular ordering of the modes of the FFTW library, three different dealiasing procedures are applied in the code:

- **1D Fourier transform,** x: this is a real-to-complex transform, so the negative frequencies are the complex conjugate of the positive ones, thus they do not need to be saved. If N_x is the number of points in the x direction, the output of the Fourier transform will be $N_x/2+1$ complex numbers (the first and the last have zero imaginary part). The modes are stored in memory as: $0, 1, 2, \ldots, N_x/2$ (due to Fortran array indexing, in the code they are shifted by one, so the corresponding array indexes are $1, 2, 3, \ldots, N_x/2+1$). The index one corresponds to the zeroth mode (mean mode).
  When applying dealiasing the modes from $2/3N_x/2$ to $N_x/2$ are set to zero.

- **1D Fourier transform,** y: this is a complex-to-complex transform. If N_y is the number of points in the y direction, the modes are stored as: $0, 1, 2, \ldots, N_y/2, -N_y/2+1, -N_y/2+2, \ldots, -1$, so first are stored the positive frequencies and then the negative ones in backward order. The corresponding array indexing is $1, 2, 3, \ldots, N_y/2+1, N_y/2+2, N_y/2+3, \ldots, N_y$. The first mode corresponds to the mean mode in the y direction.
  When applying dealiasing the modes from $2/3N_y/2$ to $N_y/2$ and from $-N_y/2+1$ to -$2/3N_y/2$ are set to zero.

- **1D Chebyshev transform,** z: this transform is formally a complex-to-complex transform, even though it can be considered as two real-to-real transforms. The

real part of the array is transformed in a real array and the imaginary part in a real array. The modes are saved as $0, 1, 2, \ldots, N_z - 1$ (where N_z is the number of points in the z direction). The corresponding array indexing is $1, 2, 3, \ldots, N_z$. When applying dealiasing the modes from $2/3N_z$ to N_z are set to zero.

3.7 Chebyshev–Tau method

3.7.1 General method

The most general form of the Chebyshev–Tau method is presented here by applying it to the Burger equation. In the following equation the Burger equation is reported in strong form.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0 \qquad \forall t > 0 \tag{3.12}$$

Strong form means that the equation must be verified at each point of the domain Ω. A solution in $(-1, 1)$ that verifies the following boundary conditions is sought:

$$\begin{cases} u(-1, t) = u_L(t) \\ u(1, t) = u_R(t) \end{cases}$$

The Chebyshev–Tau method solves the equation in weak form, which means that the integral over the domain of the equation multiplied by a test function v must be zero for each test function.

$$\int_\Omega \frac{\partial u}{\partial t} v dx + \int_\Omega u\frac{\partial u}{\partial x} v dx - \int_\Omega \nu\frac{\partial^2 u}{\partial x^2} v dx = 0 \qquad \forall v \in X, \forall t > 0 \tag{3.13}$$

The weak form is called also integral form.
The discrete solution u^N is:

$$u^N(x, t) = \sum_{k=0}^{N} \hat{u}_k(t) T_k(x)$$

We define now the following sets: the set $\mathbb{P}_N$ is the set of all the polynomials of degree lower or equal to N, the set X_N which is a subset of $\mathbb{P}_N$ and the set Y_N which is a subset of $\mathbb{P}_{N-2}$.
The Chebyshev–Tau method enforces the equation in weak form using the test function of Y_N, $N - 1$ test function (polynomials of degree from 0 to $N - 2$). Since the discrete solution han $N + 1$ coefficients we enforce the two boundary conditions to obtain the other two missing equations.
This way we have $N + 1$ equations for $N + 1$ unknown coefficients $\hat{u}_k$.

$$\int_{-1}^{1} \left(\frac{\partial u_N}{\partial t} + u_N\frac{\partial u_N}{\partial x} - \nu\frac{\partial^2 u_N}{\partial x^2} \right) T_k(x) \frac{1}{\sqrt{1 - x^2}} dx = 0 \qquad \forall\, k = 0, ..., N - 2$$

$$\begin{cases} u_N(-1, t) = u_L(t) \\ u_N(1, t) = u_R(t) \end{cases} \tag{3.14}$$

The weight $w(x) = \frac{1}{\sqrt{1-x^2}}$ is required for the orthogonality condition of Chebyshev polynomials:

$$\int_{-1}^{1} T_k(x) T_j(x) w(x) dx = \begin{cases} 0 & \text{if } j \neq k \\ \pi & \text{if } j = k = 0 \\ \frac{\pi}{2} & \text{if } j = k \neq 0 \end{cases}$$

3.7.2 Application to the 2^{nd} order equation for vorticity

The second order equation for vorticity reads:

$$\frac{\partial^2 \hat{\omega}_z^{n+1}}{\partial z^2} - \beta^2 \hat{\omega}_z^{n+1} = F^n$$

with boundary conditions in physical space:

$$\begin{cases} p_1 \omega_z(x,y,-1) + q_1 \frac{\partial \omega_z(x,y,-1)}{\partial z} = r_1(x,y) \\ p_2 \omega_z(x,y,1) + q_2 \frac{\partial \omega_z(x,y,1)}{\partial z} = r_2(x,y) \end{cases}$$

In modal space these boundary conditions are:

$$\begin{cases} p_1 \hat{\omega}_z(k_1,k_2,-1) + q_1 \frac{\partial \hat{\omega}_z(k_1,k_2,-1)}{\partial z} = \hat{r}_1(k_1,k_2) \\ p_2 \hat{\omega}_z(k_1,k_2,1) + q_2 \frac{\partial \hat{\omega}_z(k_1,k_2,1)}{\partial z} = \hat{r}_2(k_1,k_2) \end{cases}$$

To apply the Chebyshev–Tau method a one-dimensional second order equation with mixed boundary conditions is needed; in this case these hypothesis are verified.
The two functions $\hat{\omega}_z$ and F can be written as a Chebyshev truncated serie:

$$\hat{\omega}_z^{n+1} = \sum_{n=0}^{N} a_n T_n(z)$$

$$F^n = \sum_{n=0}^{N} b_n T_n(z)$$

The second order equation is then integrated in z twice; the following property of Chebyshev polynomials can be exploited:

$$\int_{-1}^{z} \sum_{n=0}^{N} a_n T_n(s) ds = \sum_{n=1}^{N+1} l_n T_n(z)$$

l_n can thus be expressed as a function of a_n:

$$\begin{cases} l_{N+1} = \frac{a_N}{2(N+1)} \\ l_N = \frac{a_N}{2N} \\ l_n = \frac{1}{2N}(a_{n-1} - a_{n+1}) \quad \text{for } n = 1, ..., N-1 \end{cases}$$

After integrating twice the vorticity transport equation, the resulting equation reads:

$$\sum_{n=0}^{N} a_n T_n(z) - \beta^2 \sum_{n=2}^{N+2} m_n T_n(z) = \sum_{n=2}^{N+2} f_n T_n(z) + A T_1(z) + B T_0(z)$$

Where

$$A = \frac{\partial \hat{\omega}_z(k_1,k_2,-1)}{\partial z}$$

and

$$B = \frac{\partial \hat{\omega}_z(k_1,k_2,-1)}{\partial z} + \hat{\omega}_z(k_1,k_2,-1)$$

s_n is now defined as:

$$s_n = a_n - \beta^2 m_n - f_n$$

This way, the equation can be rewritten as (f_n comes from the double integration of the F^n term, thus it depends on the b_n):

$$(a_0 - B)T_0(z) + (a_1 - A)T_1(z) + \sum_{n=2}^{N} s_n T_n(z) - \sum_{n=N+1}^{N+2} \left(\beta^2 m_n - f_n\right) T_n(z) = 0$$

The Chebyshev–Tau method is now applied, using $N-1$ Chebyshev polynomials as test function; in particular the test functions used are $T_n(z)w(z)$ with $n = 2, ..., N$. This choice implies that all the s_n must be zero for $n = 2, ..., N$; this results comes from the orthogonality of the Chebyshev polynomials. Thanks to the integration property s_n can be expressed as a linear combination of the a_n, while f_n as a linear combination of the b_n.
The remaining two missing equations are obtained from the boundary conditions:

$$p_1 \sum_{n=0}^{N} a_n T_n(-1) + q_1 \sum_{n=0}^{N} a_n \left.\frac{\partial T_n}{\partial z}\right|_{z=-1} = r_1$$
$$p_2 \sum_{n=0}^{N} a_n T_n(1) + q_2 \sum_{n=0}^{N} a_n \left.\frac{\partial T_n}{\partial z}\right|_{z=1} = r_2$$

The boundary conditions are then rewritten in a more compact form gathering the unknowns:

$$\sum_{n=0}^{N} d_n a_n = r_1$$
$$\sum_{n=0}^{N} e_n a_n = r_2$$

This way a $N+1$ linear equations system with $N+1$ unknowns is obtained:

$$\begin{bmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & ... & d_n \\ e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & ... & e_n \\ s_1 & 0 & v_1 & 0 & t_1 & 0 & 0 & 0 & ... & 0 \\ 0 & s_2 & 0 & v_2 & 0 & t_2 & 0 & 0 & ... & 0 \\ 0 & 0 & s_3 & 0 & v_3 & 0 & t_3 & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & ... & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & ... & v_N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_N \end{bmatrix}$$

The matrix coefficients are:

$$\begin{cases} s_{n-2} = -k^2 n & n = 3, \ldots, N+1 \\ v_{n-2} = 4n(n-1)(n-2) + 2(n-1)k^2 & n = 3, \ldots, N+1 \\ t_{n-2} = -k^2(n-2) & n = 3, \ldots, N-1 \\ g_{n-2} = nb_{n-2} - 2(n-1)b_n + (n+2)b_{n+2} & n = 3, \ldots, N-1 \\ g_{N-1} = (N-1)b_{N-3} - 2(N-2)b_{N-1} & \\ g_N = Nb_{N-2} - 2(N-1)b_N & \end{cases}$$

The coefficient matrix obtained has the first two rows full, then from the third to the $N+1$ row is a tridiagonal matrix; the system can be easily solved using a Gauss elimination algorithm.

3.7.3 Application to the 4^{th} order equation for velocity

The velocity transport equation is a fourth-order equation, so the Chebishev–Tau algorithm cannot be applied directly: first this equation must be split in two Helmholtz equations.
The auxiliary variable θ is thus defined as:

$$\theta^{n+1} = \frac{\partial^2 \hat{w}^{n+1}}{\partial z^2} - k^2 \hat{w}^{n+1}$$

This way two Helmholtz for $\hat{w}$ and θ are obtained:

$$\begin{cases} \dfrac{\partial^2 \omega^{n+1}}{\partial z^2} - \beta^2 \theta^{n+1} = F^n \\ \dfrac{\partial^2 \hat{w}^{n+1}}{\partial z^2} - k^2 \hat{w}^{n+1} = \theta^{n+1} \end{cases}$$

The Helmholtz problem for $\hat{w}$ has known boundary conditions in physical space but there are no *a priori* known boundary conditions for the auxiliary variable θ, as they are a function of the unknown velocity. The influence matrix method must thus be applied to solve these two Helmholtz problems; the detailed procedure used to calculate $\hat{w}$ is reported in Section 3.8.2.
It must be noticed that when $k^2 = 0$ the Helmholtz equations for the wall-normal velocity degenerates; in that case the solution for $k^2 = 0$ is always zero. This comes from the point that $k^2 = 0$ implies $k_x = 0$ and $k_y = 0$, which is the mean mode of the wall-normal velocity in the x and y directions, which is zero for each z value.

3.8 Influence matrix

3.8.1 General method

The influence matrix method is employed in differential problems where one of the boundary conditions depends on an unknown function.
Be Ω a generic domain and $\Gamma = \partial\Omega$ its boundary; we define then three linear differential operators $\mathbb{F}$, $\mathbb{G}$ and $\mathbb{H}$.
The following problem $[P]$ has boundary conditions for f which depends on the unknown function h:

$$[P] \begin{cases} \mathbb{F}[f(x)] = g(x) & \text{in } \Omega \\ \mathbb{G}[h(x)] = f(x) & \text{in } \Omega \\ f(x) = \mathbb{H}[h(x)] & \text{in } \Gamma \\ h(x) = h_\Gamma(x) & \text{in } \Gamma \end{cases} \tag{3.15}$$

f and h are the two unknown functions, g is a known function and h_Γ is a known boundary condition for h.
Using the influence matrix method we can evaluate the boundary conditions for f so that we can resolve the problem $[P]$.
Let's introduce another problem $[\widetilde{P}]$; the only difference from the formulation of $[P]$ is the boundary condition on $\widetilde{f}$. $\widetilde{f}_\Gamma(x)$ is an arbitrary distribution on Γ.

$$[\widetilde{P}] \begin{cases} \mathbb{F}[\widetilde{f}(x)] = g(x) & \text{in } \Omega \\ \mathbb{G}[\widetilde{h}(x)] = \widetilde{f}(x) & \text{in } \Omega \\ \widetilde{f}(x) = \widetilde{f}_\Gamma(x) & \text{in } \Gamma \\ \widetilde{h}(x) = h_\Gamma(x) & \text{in } \Gamma \end{cases} \tag{3.16}$$

The problem $[\widetilde{P}]$ has an unique solution $(\widetilde{f}, \widetilde{h})$, which does not necessarily verify the boundary condition $\widetilde{f}(x) = \mathbb{H}[\widetilde{h}(x)]$ in Γ.
We define now two functions, $\bar{f}$ and $\bar{h}$ and the problem $[\bar{P}]$ as the difference between the problems $[P]$ and $[\widetilde{P}]$.

$$\bar{f} = f - \widetilde{f}$$
$$\bar{h} = h - \widetilde{h}$$

$$[P] - [\widetilde{P}] = [\bar{P}] \begin{cases} \mathbb{F}[\bar{f}(x)] = 0 & \text{in } \Omega \\ \mathbb{G}[\bar{h}(x)] = \bar{f}(x) & \text{in } \Omega \\ \bar{f}(x) = \mathbb{H}[\bar{h}(x)] + \mathbb{H}[\widetilde{h}(x)] - \widetilde{f}_\Gamma(x) & \text{in } \Gamma \\ \bar{h}(x) = 0 & \text{in } \Gamma \end{cases} \tag{3.17}$$

We can split problem $[\bar{P}]$ in N sub-problems $[\bar{P}_k]$ with $k = 1, ..., N$ and each one of these sub-problems has its own solution $(\bar{f}_k, \bar{h}_k)$.
This way we have:

$$\bar{f}(x) = \sum_{k=1}^{N} \lambda_k \bar{f}_k(x)$$
$$\bar{h}(x) = \sum_{k=1}^{N} \lambda_k \bar{h}_k(x)$$

The generic problem $[\bar{P}_k]$ is (in the following the boundary has been discretized in N points $x_l = 1, ..., N$):

$$[\bar{P}_k] \begin{cases} \mathbb{F}[\bar{f}_k(x_l)] = 0 & \text{in } \Omega \\ \mathbb{G}[\bar{h}_k(x_l)] = \bar{f}_k(x_l) & \text{in } \Omega \\ \bar{f}_k(x_l) = \delta_{kl} & \text{in } \Gamma \\ \bar{h}_k(x_l) = 0 & \text{in } \Gamma \end{cases} \tag{3.18}$$

The λ_k coefficients are unknown and they can be calculated from the third equation of the problem $[\bar{P}]$ evaluated at each point x_l, substituting in $\bar{f}$ and $\bar{h}$ the sub-problems solutions.

$$\sum_{k=1}^{N} \lambda_k \bar{f}_k(x) - \mathbb{H}\left[\sum_{k=1}^{N} \lambda_k \bar{h}_k(x)\right] = -\widetilde{f}(x) + \mathbb{H}[\widetilde{h}(x)]$$

Since $\mathbb{H}$ is a linear operator we can write:

$$\sum_{k=1}^{N} \lambda_k \left(\bar{f}_k(x) - \mathbb{H}[\bar{h}_k(x)]\right) = -\widetilde{f}(x) + \mathbb{H}[\widetilde{h}(x)]$$

This equation holds for each x_l with $l = 1, ..., N$, so we have a linear equations system:

$$\begin{bmatrix} \bar{f}_1(x_1) - \mathbb{H}[\bar{h}_1(x_1)] & ... & \bar{f}_N(x_1) - \mathbb{H}[\bar{h}_N(x_1)] \\ \bar{f}_1(x_2) - \mathbb{H}[\bar{h}_1(x_2)] & ... & \bar{f}_N(x_2) - \mathbb{H}[\bar{h}_N(x_2)] \\ \vdots & & \vdots \\ \bar{f}_1(x_N) - \mathbb{H}[\bar{h}_1(x_N)] & ... & \bar{f}_N(x_N) - \mathbb{H}[\bar{h}_N(x_N)] \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} -\widetilde{f}(x_1) + \mathbb{H}[\widetilde{h}(x_1)] \\ -\widetilde{f}(x_2) + \mathbb{H}[\widetilde{h}(x_2)] \\ \vdots \\ -\widetilde{f}(x_N) + \mathbb{H}[\widetilde{h}(x_N)] \end{bmatrix} \tag{3.19}$$

Using the definitions of $\bar{f}$ and $\bar{h}$ we have:

$$f(x) = \widetilde{f}(x) + \bar{f}(x) = \widetilde{f}(x) + \sum_{k=1}^{N} \lambda_k \bar{f}_k(x)$$
$$h(x) = \widetilde{h}(x) + \bar{h}(x) = \widetilde{h}(x) + \sum_{k=1}^{N} \lambda_k \bar{h}_k(x)$$

Using problems $[\bar{P}]$ and $[\widetilde{P}]$ we have:

$$f(x) = \widetilde{f}(x) + \bar{f}(x) = \mathbb{G}[\widetilde{h}(x)] + \sum_{k=1}^{N} \lambda_k \mathbb{G}[\bar{h}_k(x)]$$

3.8.2 Application to the 4^{th} order equation for $\hat{w}$

The influence matrix method is needed as the boundary conditions on one of the two Helmholtz problems (obtained from the splitting of the fourth order equation for the wall-normal velocity) are missing. The original fourth order equation reads:

$$\left(\frac{\partial^2}{\partial z^2} - k^2\right)\left(\frac{\partial^2}{\partial z^2} - \beta^2\right)\hat{w} = H \tag{3.20}$$

The $\hat{}$ denotes quantities in modal space; for ease of notation in this section the $\hat{}$ notation will be dropped.
For the closed channel case the boundary conditions are on the velocity value and on its first derivative at the wall:

$$\begin{cases} w(z = \pm 1) = 0 \\ \left.\dfrac{\partial w}{\partial z}\right|_{z=\pm 1} = 0 \end{cases} \tag{3.21}$$

For the open channel case the boundary conditions on the open boundary are on the velocity value and on its wall-normal second derivative; in this case there are the boundary conditions for the auxiliary problem, but only at one boundary. For the other boundary, the closed boundary, the boundary conditions for the auxiliary problem are still missing.
Equation 3.20 can be splitted in two Helmholtz equations (the previous section notation is kept: Ω denotes the domain, $\Gamma = \delta\Omega$ is the border of the domain):

$$\begin{cases} \left(\dfrac{\partial^2}{\partial z^2} - k^2\right) w = \psi & \text{in } \Omega \\ w = w_\gamma & \text{in } \Gamma \end{cases}$$
$$\begin{cases} \left(\dfrac{\partial^2}{\partial z^2} - \beta^2\right) \psi = H & \text{in } \Omega \\ \psi = f(w) & \text{in } \Gamma \end{cases} \tag{3.22}$$

The boundary conditions on the ψ problem are unknown, as they are a function of the wall-normal velocity w. As seen for the general case, these Helmholtz problems can be split in a subproblem that do not necessarily verifies the boundary conditions and two other subproblems that verify the boundary conditions on one of the two boundaries (one subproblem satisfies the boundary condition at $z = +1$, while the other at $z = -1$).

$$\begin{cases} w = w_1 + Aw_2 + Bw_3 \\ \psi = \psi_1 + A\psi_2 + B\psi_3 \end{cases} \tag{3.23}$$

Problem with subscript 1 has a unique solution that does not necessarily verify the boundary conditions on ψ, problem with subscript 2 verifies the boundary conditions

on ψ at $z = -1$, while problem with subscript 3 verifies the boundary conditions on ψ at $z = +1$. The three subproblems are thus:

$$[P_1] = \begin{cases} \dfrac{\partial^2 w_1}{\partial z^2} - k^2 w_1 = \psi_1 & \text{in } \Omega \\ w_1 = w_\gamma & \text{in } \Gamma \\ \dfrac{\partial^2 \psi_1}{\partial z^2} - \beta^2 \psi_1 = H & \text{in } \Omega \\ \psi_1 = \psi_\gamma & \text{in } \Gamma \end{cases}$$

$$[P_2] = \begin{cases} \dfrac{\partial^2 w_2}{\partial z^2} - k^2 w_2 = \psi_2 & \text{in } \Omega \\ w_2 = 0 & \text{in } \Gamma \\ \dfrac{\partial^2 \psi_2}{\partial z^2} - \beta^2 \psi_2 = 0 & \text{in } \Omega \\ \psi_2(-1) = 1 \quad \psi_2(+1) = 0 \end{cases} \tag{3.24}$$

$$[P_3] = \begin{cases} \dfrac{\partial^2 w_3}{\partial z^2} - k^2 w_3 = \psi_3 & \text{in } \Omega \\ w_3 = 0 & \text{in } \Gamma \\ \dfrac{\partial^2 \psi_3}{\partial z^2} - \beta^2 \psi_3 = 0 & \text{in } \Omega \\ \psi_3(-1) = 0 \quad \psi_3(+1) = 1 \end{cases}$$

The boundary condition on ψ_1 is arbitrary; in the code $\psi_1 = 0$ in Γ was selected. The problems $[P_2]$ and $[P_3]$ are not time dependent, so they are solved at the beginning of the simulation and stored in the arrays `wa2` and `wa3` (only the auxiliary solution for w is kept); the problem $[P_1]$ is time dependent (the H term is time dependent), so it is calculated during the time cycle.
Once splitted the original Helmholtz problems in three subproblems, the boundary conditions on w are applied:

$$\begin{cases} p_1 w(-1) + q_1 \left.\dfrac{\partial w}{\partial z}\right|_{z=-1} = r_1 \\ p_2 w(+1) + q_2 \left.\dfrac{\partial w}{\partial z}\right|_{z=+1} = r_2 \end{cases}$$

The variable w can be split in a linear combination of w_1, w_2 and w_3:

$$\begin{cases} p_1 \left(w_1(-1) + A w_2(-1) + B w_3(-1)\right) + q_1 \left(\dfrac{\partial w_1(-1)}{\partial z} + A \dfrac{\partial w_2(-1)}{\partial z} + B \dfrac{\partial w_3(-1)}{\partial z}\right) = r_1 \\ p_2 \left(w_1(+1) + A w_2(+1) + B w_3(+1)\right) + q_2 \left(\dfrac{\partial w_1(+1)}{\partial z} + A \dfrac{\partial w_2(+1)}{\partial z} + B \dfrac{\partial w_3(+1)}{\partial z}\right) = r_2 \end{cases}$$

Since the three subproblems have an unique solution and verify the Chebyshev–Tau method hypothesis, they can be solved using the Chebyshev–Tau method and then the values of the coefficient A and B can be calculated:

$$\begin{bmatrix} p_1 w_2(-1) + q_1 \dfrac{\partial w_2(-1)}{\partial z} & p_1 w_3(-1) + q_1 \dfrac{\partial w_3(-1)}{\partial z} \\ p_2 w_2(+1) + q_2 \dfrac{\partial w_2(+1)}{\partial z} & p_1 w_3(+1) + q_1 \dfrac{\partial w_3(+1)}{\partial z} \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} r_1 - p_1 w_1(-1) - q_1 \dfrac{\partial w_1(-1)}{\partial z} \\ r_2 - p_2 w_1(+1) - q_2 \dfrac{\partial w_1(+1)}{\partial z} \end{bmatrix}$$

To solve for w_i with $i = 1, 2, 3$, first the Helmholtz problem for ψ_i is solved, then the right hand side of the Helmoltz equations for w_i is known.
Since the code works in modal space, w_1 is a complex valued function, while w_2 and w_3 are real valued fuction; thus A and B must be complex valued coefficients.

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}}_{\mathbb{R}} \underbrace{\begin{bmatrix} A \\ B \end{bmatrix}}_{\mathbb{C}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}_{\mathbb{C}}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \mathbb{R}\mathrm{e}(A) \\ \mathbb{R}\mathrm{e}(B) \end{bmatrix} = \begin{bmatrix} \mathbb{R}\mathrm{e}(b_1) \\ \mathbb{R}\mathrm{e}(b_2) \end{bmatrix}$$
$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \mathbb{I}\mathrm{m}(A) \\ \mathbb{I}\mathrm{m}(B) \end{bmatrix} = \begin{bmatrix} \mathbb{I}\mathrm{m}(b_1) \\ \mathbb{I}\mathrm{m}(b_2) \end{bmatrix} \tag{3.25}$$

Once A and B have been calculated, the value of w can be obtained as:

$$\begin{cases} \mathbb{R}\mathrm{e}(w) = \mathbb{R}\mathrm{e}(w_1) + \mathbb{R}\mathrm{e}(A)w_2 + \mathbb{R}\mathrm{e}(B)w_3 \\ \mathbb{I}\mathrm{m}(w) = \mathbb{I}\mathrm{m}(w_1) + \mathbb{I}\mathrm{m}(A)w_2 + \mathbb{I}\mathrm{m}(B)w_3 \end{cases} \tag{3.26}$$

3.9 Wall units and outer units

In the code two different dimensionless categories are used: outer units and wall units. The first ones are denoted using the superscript $-$, while the second ones using the superscript $+$.
Outer units are obtained making physical units dimensionless with the channel half-height h and the shear velocity: u_τ

$$\mathbf{x}^- = \frac{\mathbf{x}}{h} \qquad \mathbf{u}^- = \frac{\mathbf{u}}{u_\tau} \qquad t^- = \frac{t u_\tau}{h}$$

On the other hand wall units are obtained nondimensionalizing physical units with the shear velocity u_τ and the kinematic viscosity ν (typical turbulence related quantities):

$$\mathbf{x}^+ = \frac{\mathbf{x} u_\tau}{\nu} \qquad \mathbf{u}^+ = \frac{\mathbf{u}}{u_\tau} \qquad t^+ = \frac{t u_\tau^2}{\nu}$$

So, the dimensionless velocity is the same in both wall units and outer units, while for the spatial coordinate we have:

$$\mathbf{x}^+ = \frac{\mathbf{x} u_\tau}{\nu} = \frac{\mathbf{x}}{h} \frac{h u_\tau}{\nu} = \mathbf{x}^- \mathrm{Re}_\tau$$

We have the same result for time:

$$t^+ = \frac{t u_\tau^2}{\nu} = \frac{t u_\tau}{h} \frac{h u_\tau}{\nu} = t^- \mathrm{Re}_\tau$$

Chapter 4

MPI Parallelization and GPUs

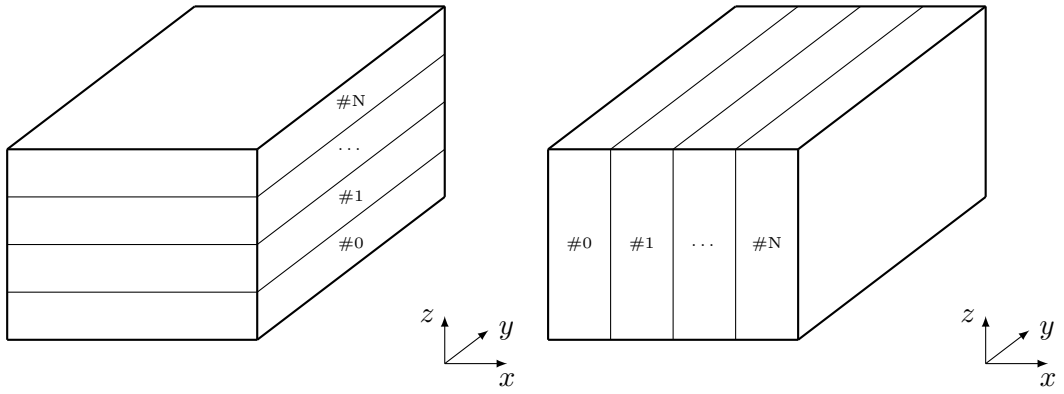
In this chapter, the parallelization, two possible domain decomposition algorithms and the acceleration strategy will be presented: the slab decomposition and the pencil decomposition. Section 4.1 presents the general parallelization strategy. Sections 4.2 and 4.3 will be dedicated to the detailed explanation of these parallelization strategies, focusing also on their strengths and limitations. Section 4.4 will present a benchmark between these strategies, together with some scalability results. Finally, section 4.5 will discuss the acceleration strategy for GPU-use and section 4.6 will discuss possible performance improvement and profiling techniques.

4.1 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.

4.2 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 4.1 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.



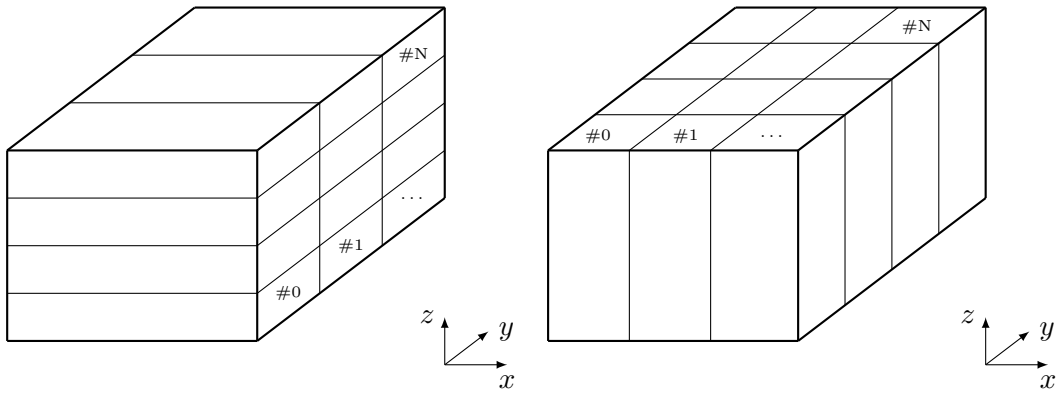
(a) Slab decomposition, physical space (b) Slab decomposition, modal space

Figure 4.1: 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.

4.3 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,



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

Figure 4.2: Pencil decomposition, MPI processes numbering

according to Figure 4.2, 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 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.

4.4 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 4.3 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.

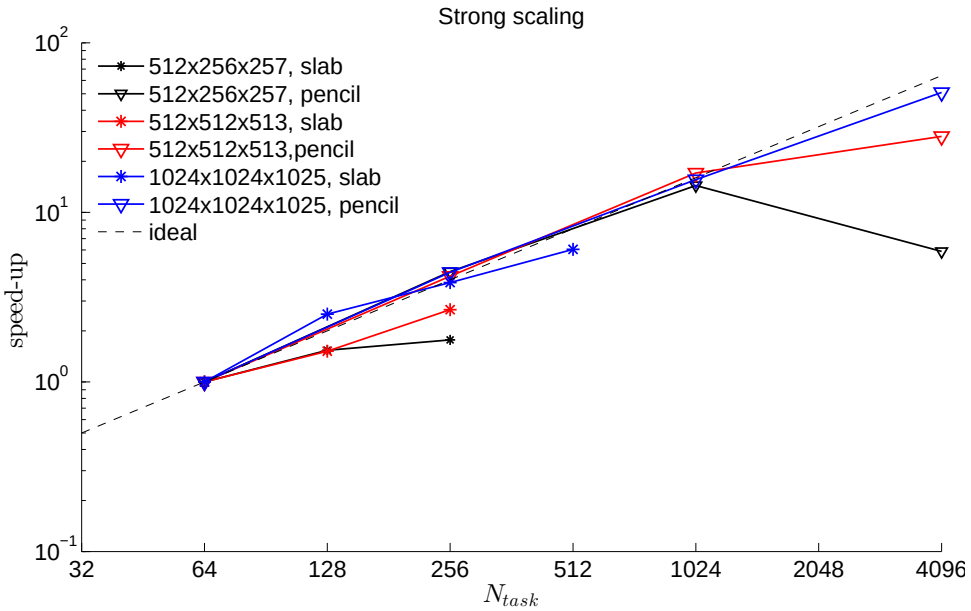


Figure 4.3: Strong scalability

Table 4.1: 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

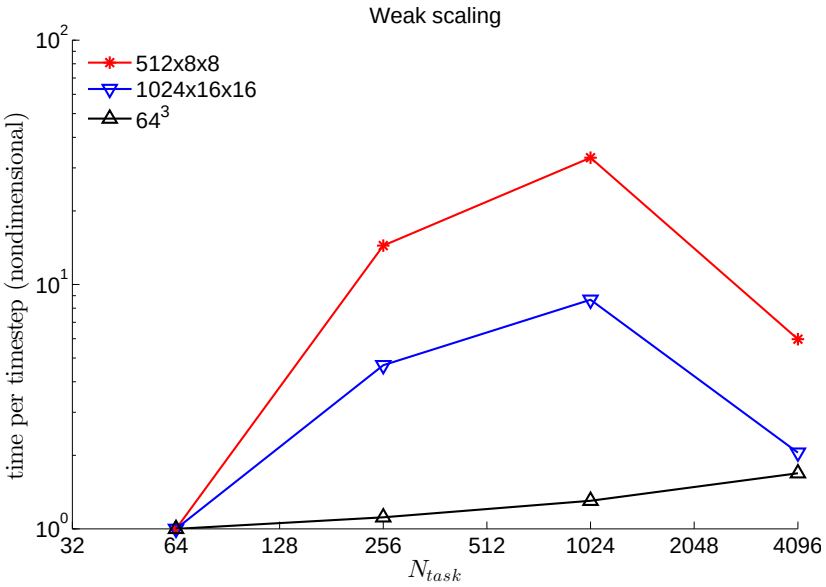


Figure 4.4: Weak scalability

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 4.2: 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

4.5 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).

4.6 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`.

Chapter 5

Code validation

5.1 Single phase validation

The code was validated running a simulation at $\mathrm{Re}_\tau = 300$ starting from a fully developed turbulent channel flow; all the results were gathered on a timespan of 1 500 t^+. The results were then compared with the DNS database of N. Kasagi (2017) at the same shear Reynolds number. Here is presented the outcome of this comparison; the data from N. Kasagi (2017) are denoted as tht-lab in the following pictures.
The velocity skewness was also compared with the old code results (`FlOWSB`), showing a perfect agreement.

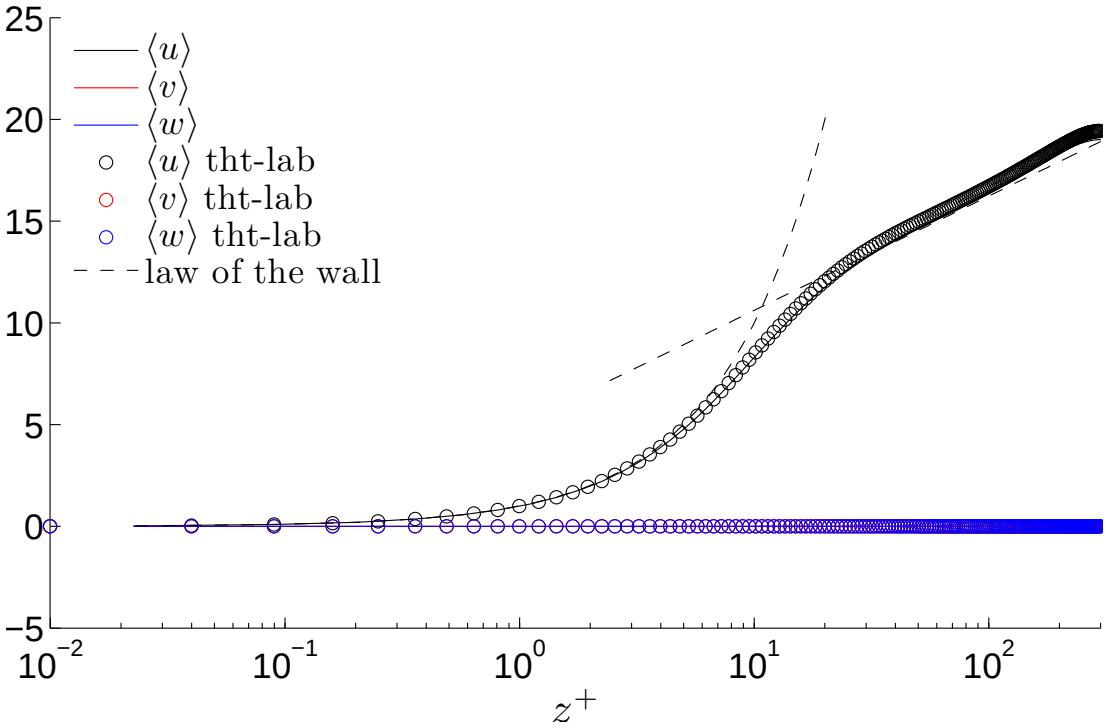


Figure 5.1: Mean velocity

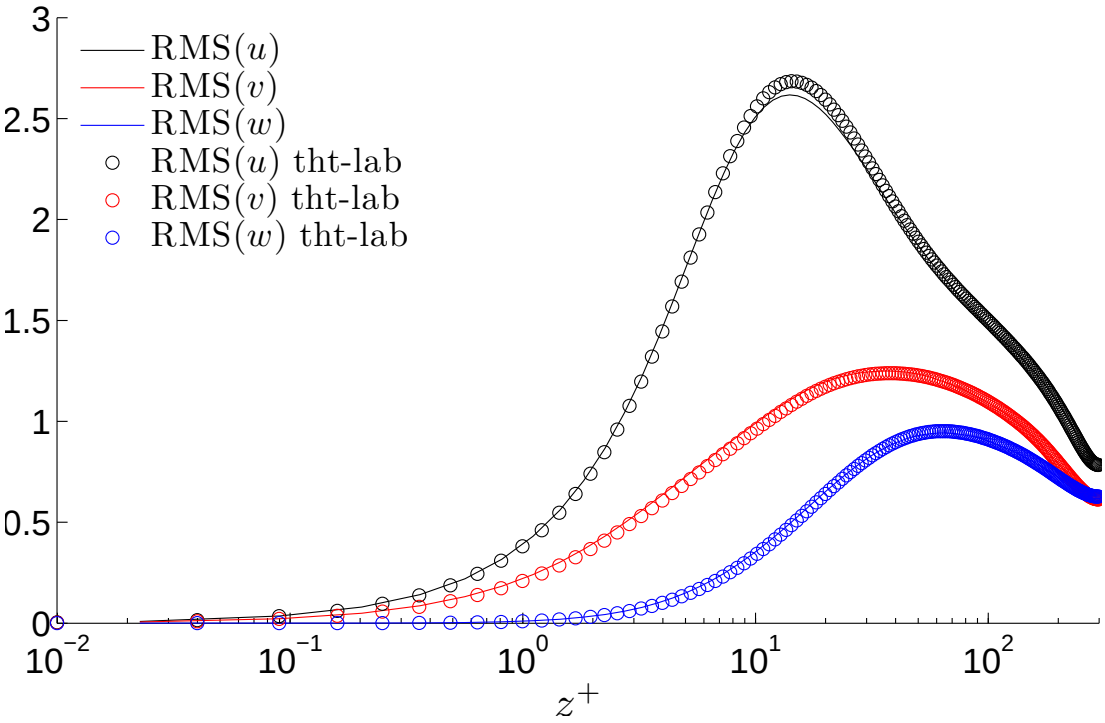


Figure 5.2: Velocity root mean square

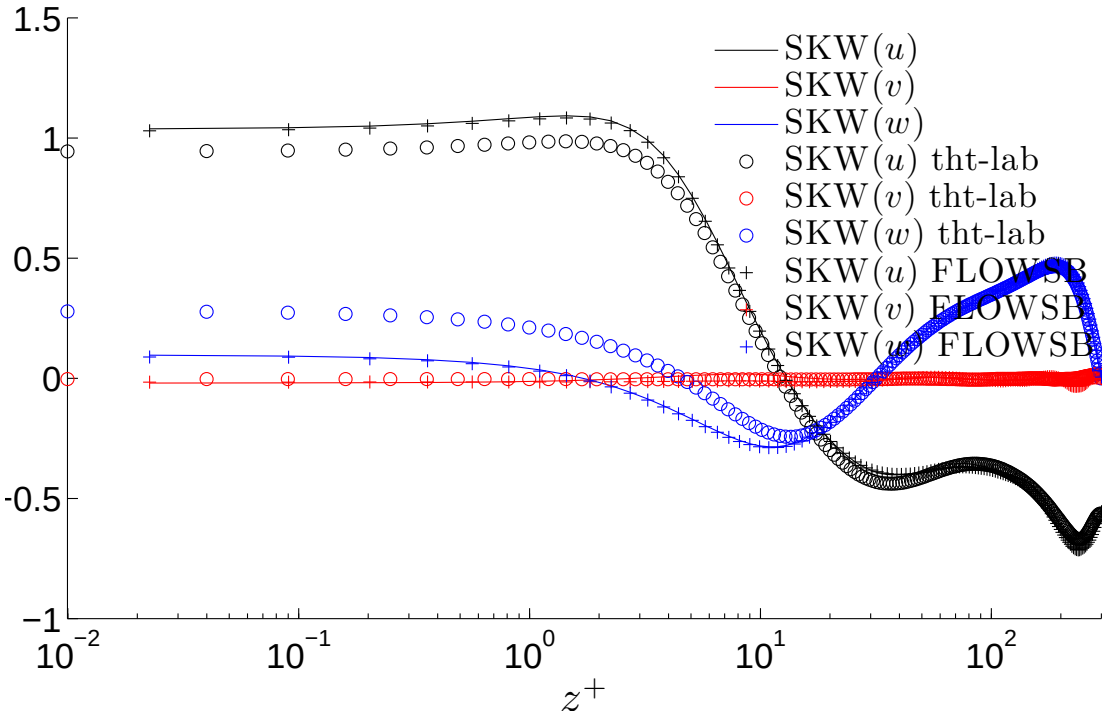


Figure 5.3: Velocity skewness

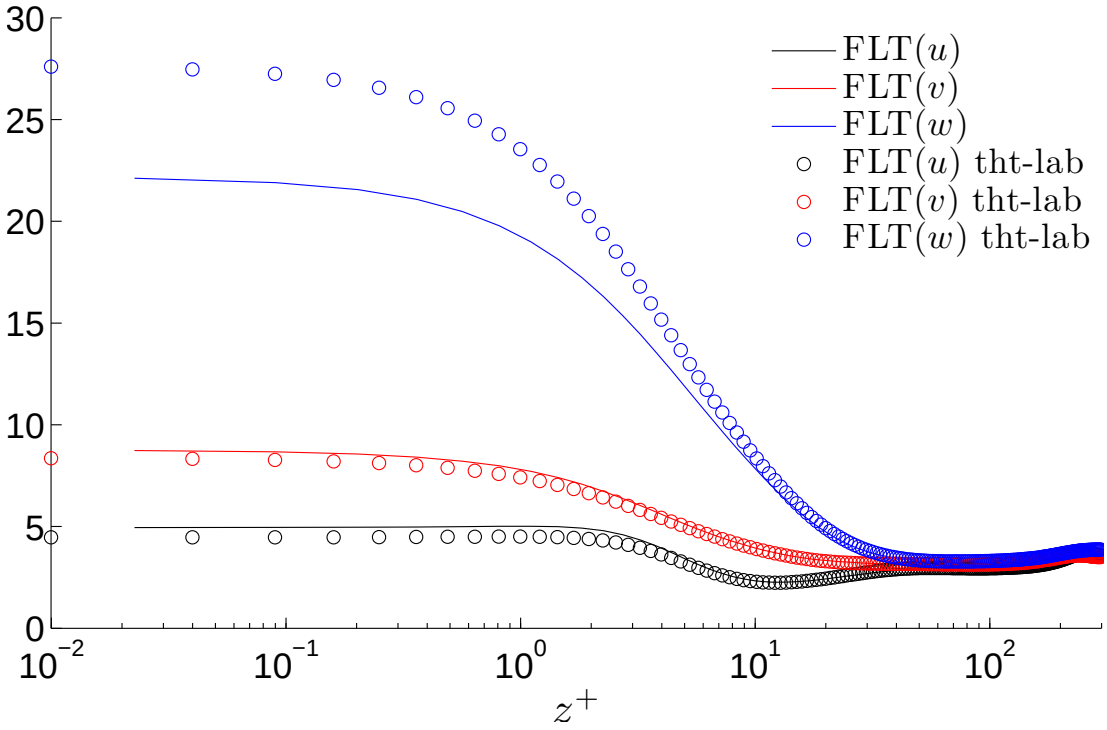


Figure 5.4: Velocity flatness

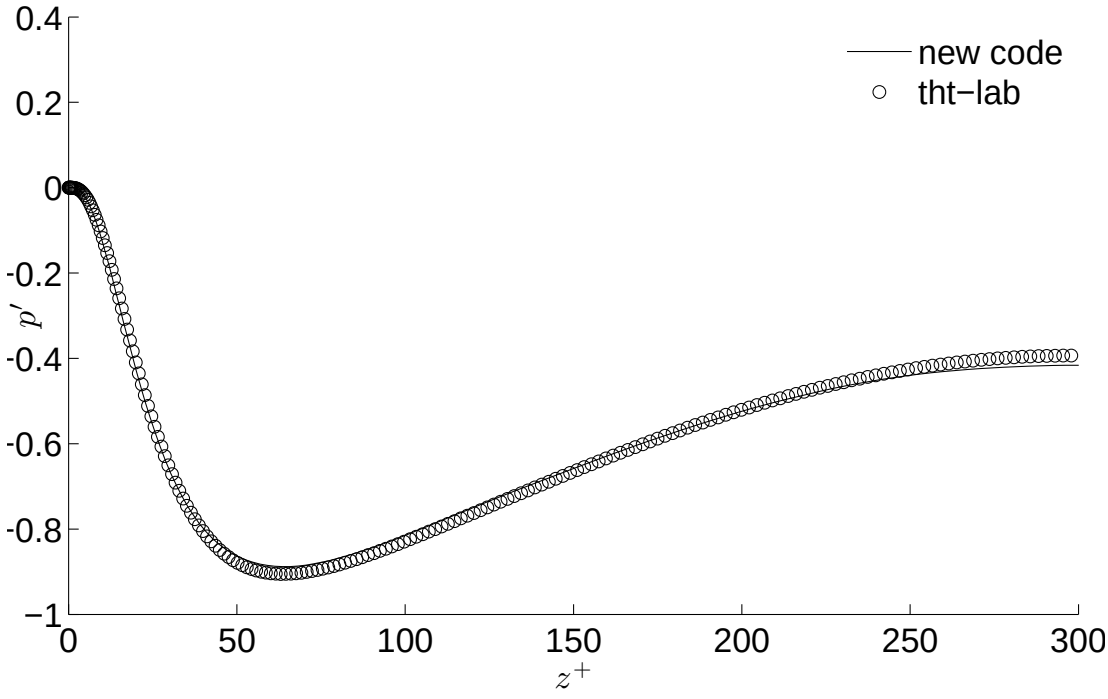


Figure 5.5: Mean pressure

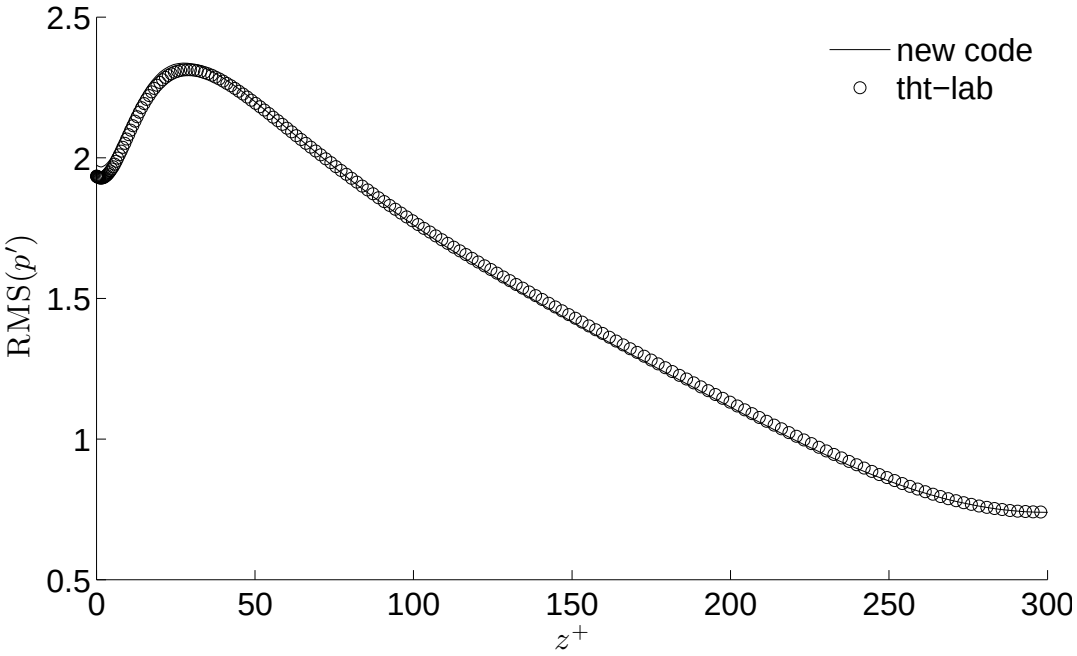


Figure 5.6: Pressure root mean square

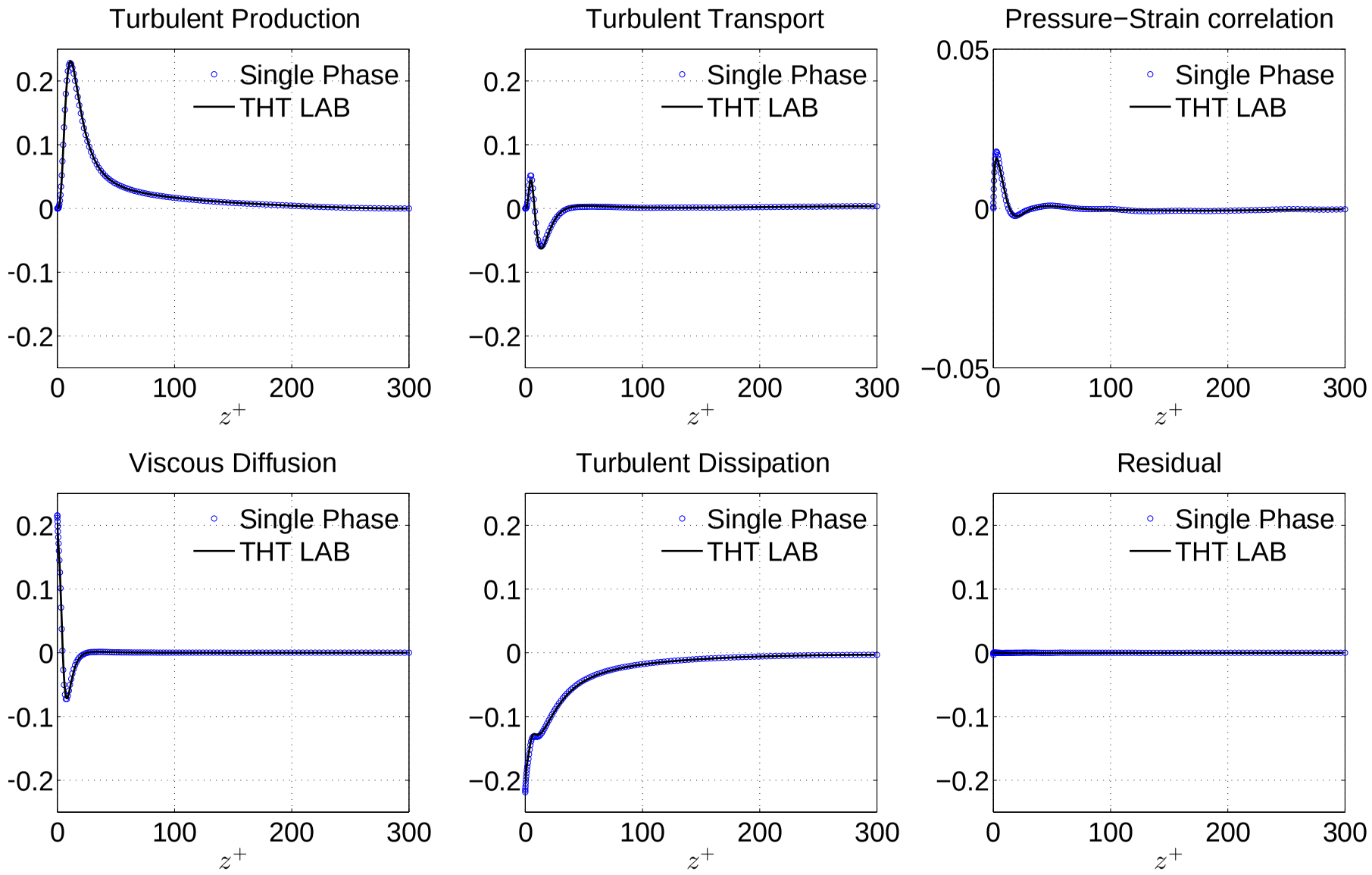


Figure 5.7: Energy budget

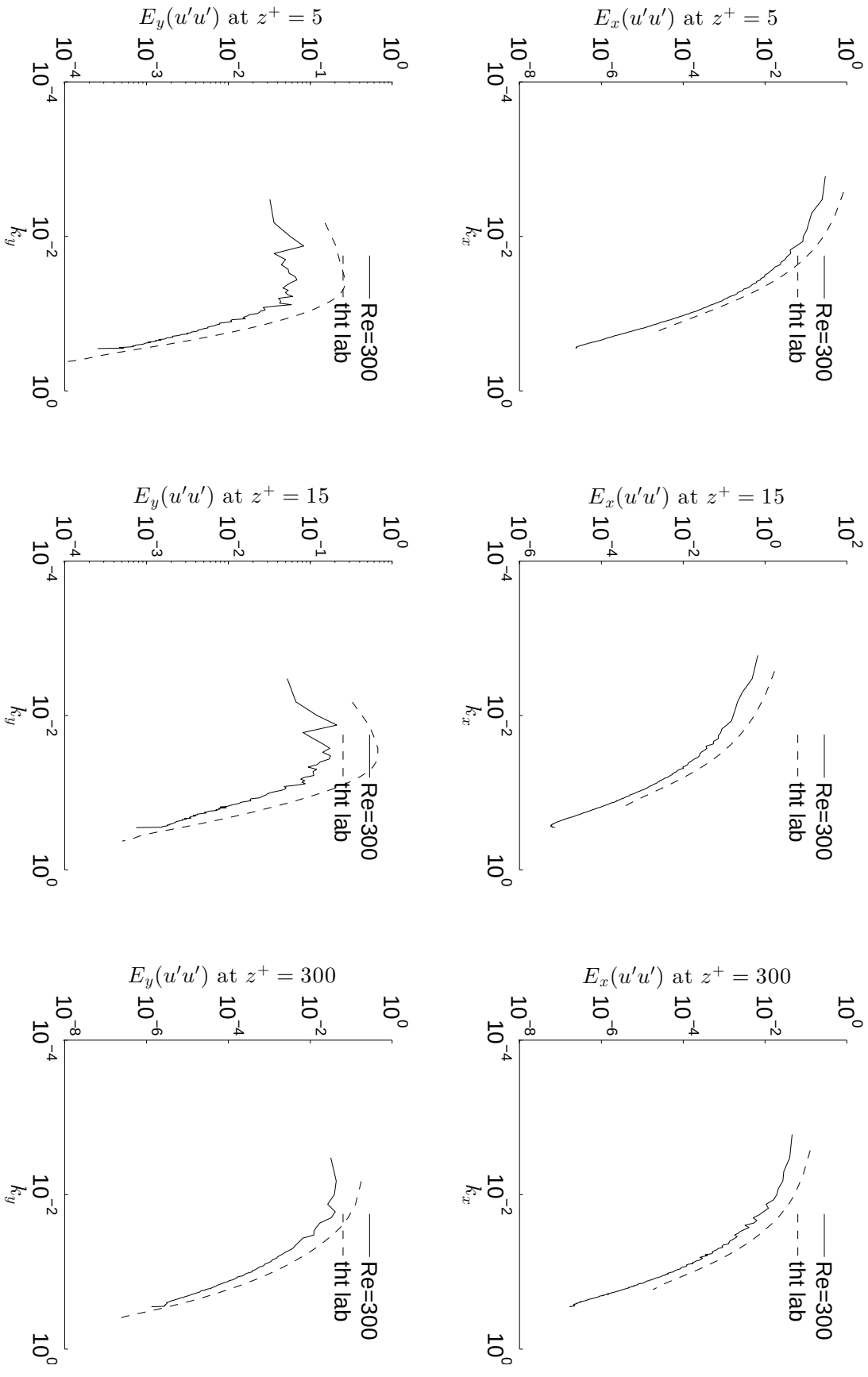


Figure 5.8: u' power spectra

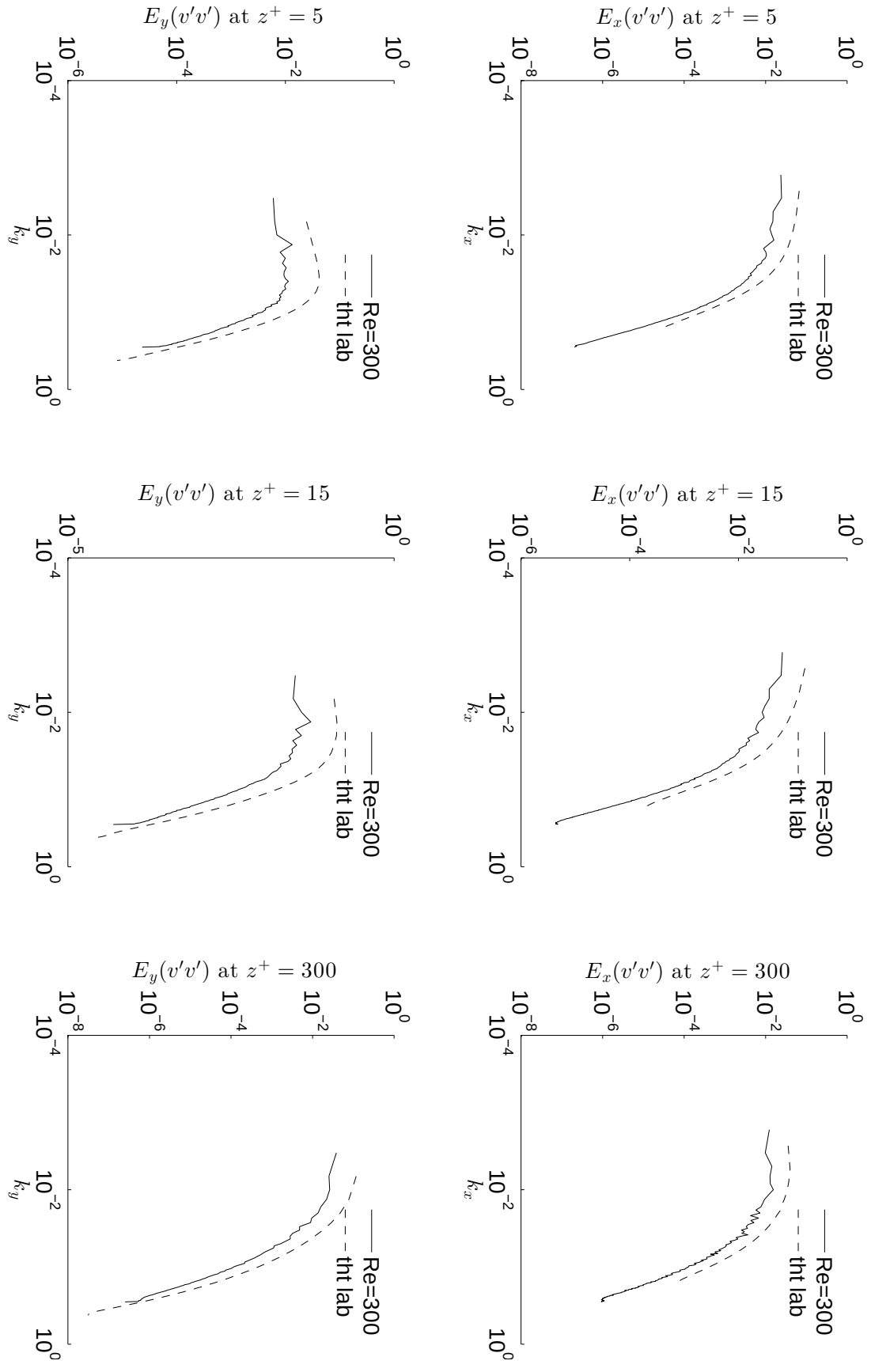


Figure 5.9: v' power spectra

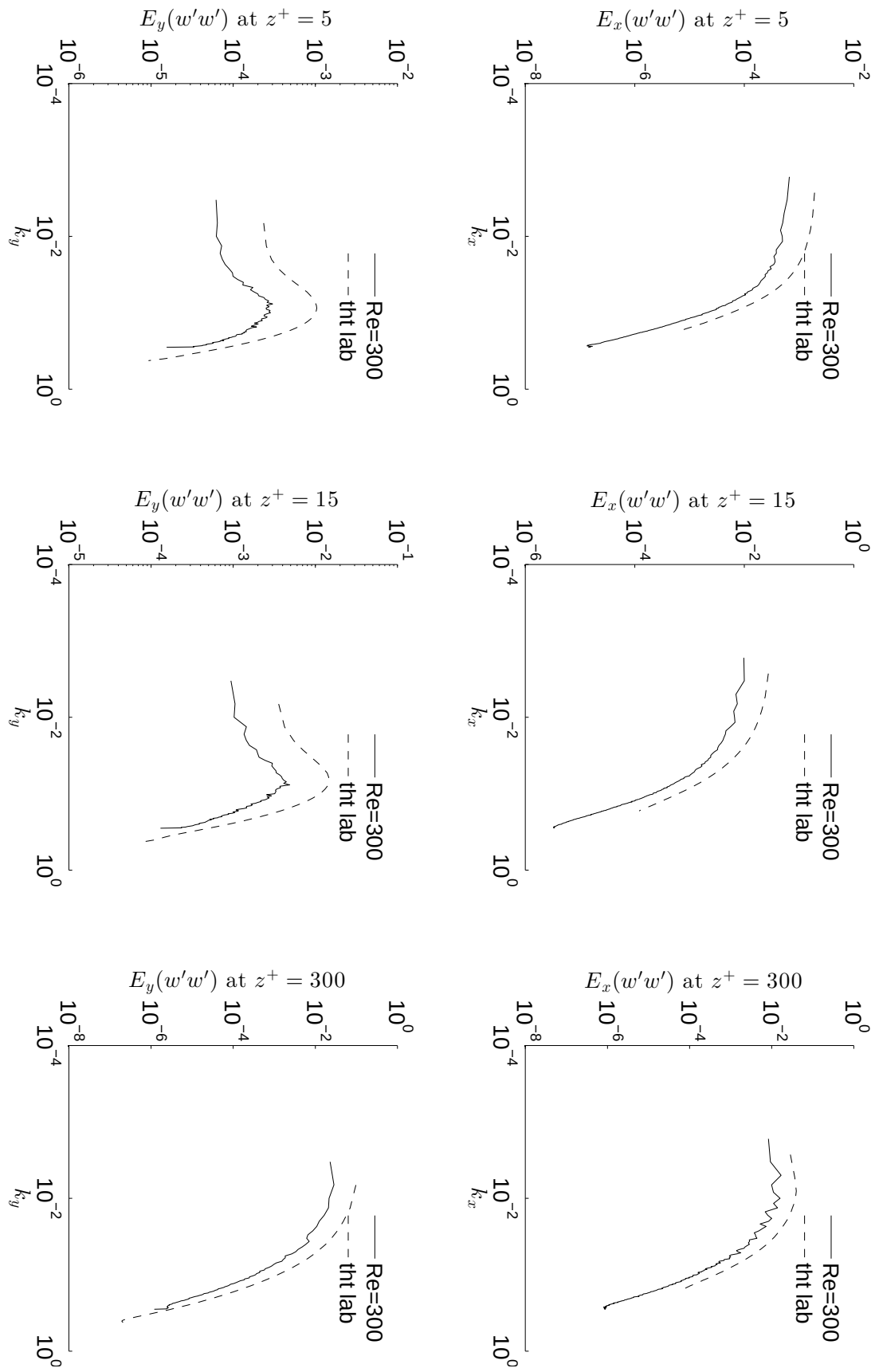


Figure 5.10: w' power spectra

5.2 Phase field validation

The phase field was validated using the undamped analytical solution from Prosperetti (1981) for capillary waves at the interface between two superposed fluids (stable configuration).

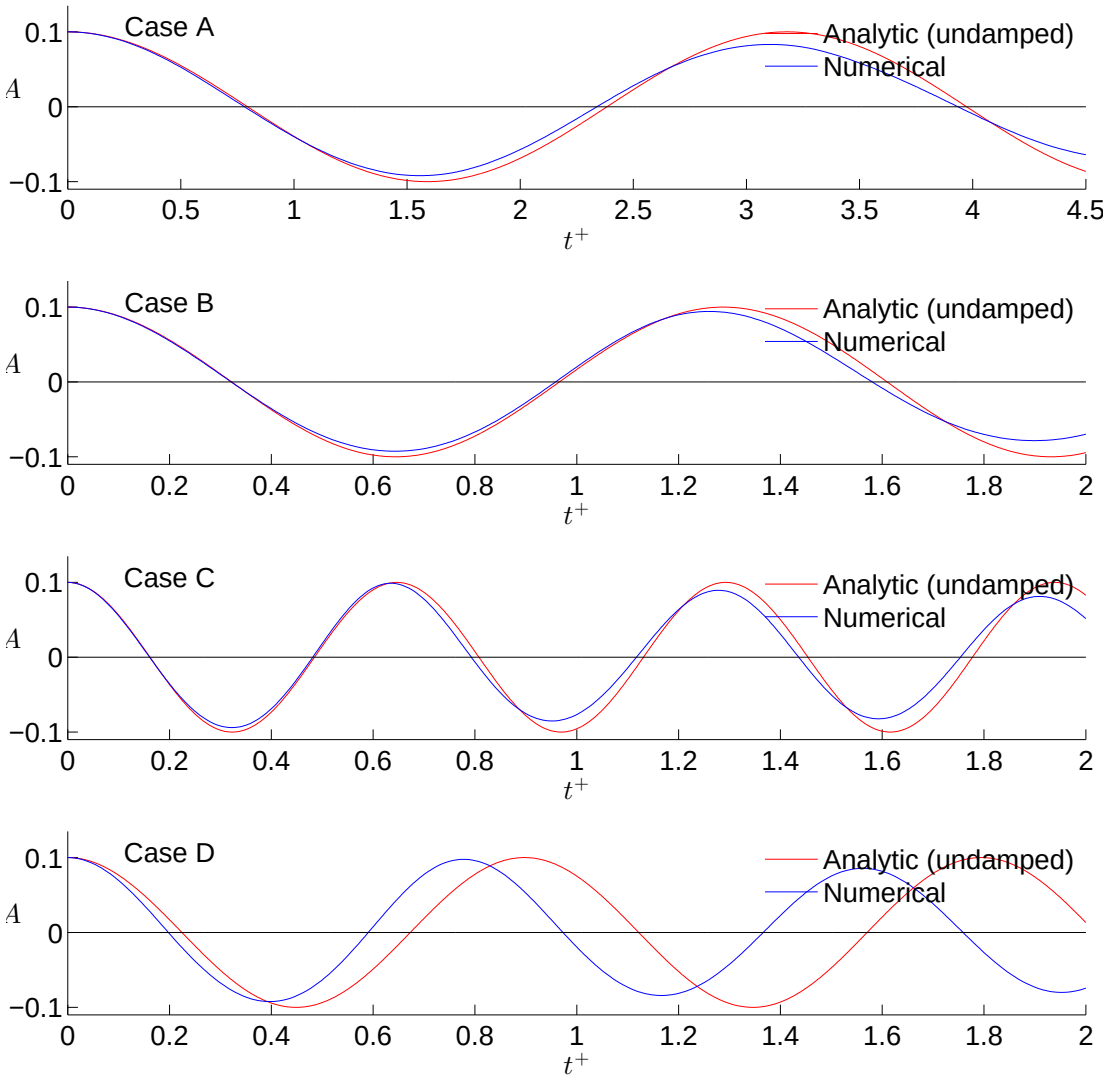


Figure 5.11: Capillary waves, comparison of numerical solution (damped) with Prosperetti analytic solution (undamped). **Case A** : $\rho_R = 0.9$, Fr $= 0.1$, $k_y = 1$. **Case B** : $\rho_R = 0.5$, Fr $= 0.1$, $k_y = 1$. **Case C** : $\rho_R = 0.5$, Fr $= 0.05$, $k_y = 1$. **Case D** : $\rho_R = 0.5$, Fr $= 0.1$, $k_y = 2$.

Table 5.1: Case run for phase field validation

Case	ω_0 (analytic)	ω (numerical)	$\Delta\omega = \frac{\omega_0 - \omega}{\omega_0} \cdot 100$ [%]	Density ratio	Fr
Case A	1.976789	1.993080	-0.82	0.9	0.1
Case B	4.879132	4.986655	-2.20	0.5	0.1
Case C	9.721965	9.933890	-2.18	0.5	0.05
Case D	7.001862	8.107336	-15.79	0.5	0.1

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