# PSI-Boil Tutorial

## Volume 1: Fundamentals

Bojan Ničeno
*Laboratory for Simulation Modelling and Computation*
*Paul Scherrer Institut, 5232 Villigen PSI, Switzerland*
bojan.niceno@psi.ch

*updated by* Yohei Sato
yohei.sato@psi.ch

May 24, 2024

# Contents

# Chapter 1

# Introduction

## 1.1 About PSI-Boil

PSI-Boil (Parallel SImulator of Boiling phenomena) is a three-dimensional, numerical solver for single- and two-phase flows with or without heat transfer. It has the capability to simulate conjugate heat transfer problems as well. The discretization of governing equations is based on the $2^{nd}$ order accurate Finite Volume (FV) method, on staggered orthogonal grids. Two-phase flows are simulated with surface tracking algorithm, based on conservative Level Set (LS) approach. Linear solvers are based on the Krylov's sub-space family of algorithms, which can be accelerated with an algebraic multigrid method.

PSI-Boil is written in `C++` programming language and uses Message Passing Interface (`MPI`) for parallelization. It compiles on most Linux-based computers[1]. The compilation procedure relies on `autotools`, as well as on the `make` utility. On most computational platforms, compilation consists of running `configure`, followed by `make`. PSI-Boil has been compiled on Linux-based PCs, clusters, as well as main-frame computers such as Cray-XT3[2]. It is also possible to compile it on Windows XP[3]. Cygwin system[4] and without `MPI` support.

The development of PSI-Boil has started in the summer of 2006, at Paul Scherrer Institute (PSI), as an integral part of the project on Multi-Scale Modeling Analysis (MSMA), which focuses on mechanistic modeling of boiling phenomena at multiple scales. Initially, it is available only to PSI personnel working on the MSMA project. But, should it prove useful, it might spread further among the academic community, presumably under a `GNU` license[5]. The main purpose of PSI-Boil is not to compete with commercial Computational Fluid Dynamics (CFD) packages, but rather to serve as a tool for academic community.

Since it's aim is not to compete with commercial CFD codes, PSI-Boil is not an integral program capable to solve a general fluid flow problem. Rather, it is a suite of objects an algorithms which are used as blocks for building programs for particular flow problems. As you will see in this tutorial, for every problem being solved, a new program[6] is written. These programs are supposed to be short and highly specialized for a particular task.

Following this rationale, PSI-Boil does not have any input files. It's main program is re-written for each problem being solved and actually serves as an input file itself.

---

[1] `www.linux.org`
[2] Cray is a registered trademark of Cray Inc. (`www.cray.com`).
[3] Windows XP is a registered trademark of Microsoft Inc. (`www.microsoft.com`).
[4] `www.cygwin.com`
[5] www.gnu.org
[6] A new `main()` function, embodied into a separate `main.cpp` file.

# Chapter 2

# Getting and Compiling the PSI-Boil Sources

## 2.1 Obtaining the sources

PSI-Boil is developed and supported for Linux operating systems. The main reasons for that are free availability of C++ compilers[1], MPI libraries[2], make facility and autotools. It can be compiled for Windows as well, but only under the Cygwin sub-system, being a Linux-like environment for Windows. Cygwin comes with all tools and libraries necessary to compile PSI-Boil, *except* MPI libraries.

### 2.1.1 Prerequisites for obtaining the PSI-Boil

PSI-Boil is stored in Github (https://github.com/Niceno/PSI-BOIL). Before you try to obtain the sources, make sure that you have:

- an account on Github,
- access to a Linux-based PC or workstation (computer, for short) *or* a Windows-based PC with Cygwin,
- git installed on your system.

To obtain the source code, type the next command:

```
> git clone https://github.com/Niceno/PSI-BOIL
```

(followed by enter) in your shell and you should get something like:

```
remote: Enumerating objects: 12801, done.
remote: Counting objects: 100% (926/926), done.
remote: Compressing objects: 100% (498/498), done.
remote: Total 12801 (delta 447), reused 856 (delta 419), pack-reused 11875
Receiving objects: 100% (12801/12801), 58.93 MiB | 21.76 MiB/s, done.
Resolving deltas: 100% (8691/8691), done.
```

Then the folder PSI-BOIL is created, which stores all the codes.

[1] www.thefreecountry.com/compilers/cpp.shtml
[2] www-unix.mcs.anl.gov/mpi/mpich1, www.open-mpi.org

### 2.1.2 Usage of git

Please refer some books or ask ChatGPT about the usage of Github. Here, several useful commands are listed.

First, change directory to the folder: PSI-BOIL.

```
> cd PSI-BOIL
```

Then, to list all the branches of PSI-BOIL in Github,

```
> git branch -a
```

To show the current branch,

```
> git branch --contains
```

To switch to a branch, named e.g. branch_to_switch,

```
> git checkout branch_to_switch
```

To create a new branch (new_branch) from a source branch (source_branch),

```
> git checkout -b new_branch old_branch
 (if you have access right)
> git push origin new_branch
```

Section 2.1 in a nutshell

- To obtain PSI-Boil sources, you need:

  - an account on Github system,
  - access to a Linux-based computer or Windows-based PC with Cygwin, either with access to PSI's afs,
  - git software (installed on almost every Linux).

-Provided all of the above is fulfilled, you get the sources with the command:

  - `git clone https://github.com/Niceno/PSI-BOIL`

## 2.2 Compiling the sources

### 2.2.1 Availability of required software

In order to compile PSI-Boil you should obtain the sources first, which is explained in 2.1. If you don't have the sources yet, please get them before proceeding with this section in the tutorial. Furthermore, you need:

- a C++ compiler and

- (hopefully) MPI compiler and libraries.

C++ compiler is available on most Linux distributions, as well as MPI. The most widely available on Linux is GNU C++ compiler. You can check if it is available on your system by running:

```
> g++
```

PSI-Boil is mostly compiled with g++, so it is a safe option to use it.

To check if you have MPI on your system, run the command:

```
> mpiexec
```

That should give you a hint whether MPI is installed, but also which *flavor* it is. That is important to know. In case you had LAM/MPI implementation, the mpiexec command will result in output like:

```
Synopsis:       mpiexec [options] <app>
                mpiexec [options] <where> <program> [<prog args>]

Description:    Start an MPI application in LAM/MPI.

Notes:
                [options]       Zero or more of the options listed below
...
```

Even if mpiexec command is available on your system, C++ compiler for MPI might not be. Try to run mpiCC or mpicxx in the shell. If none of those is available, they might be available as modules. To check which modules are available on your system, us the command:

```
> module avail
```

That should give an output like:

```
------------------------------------ Compiler -------------------------------------
OpenBLAS/0.3.23                gsl/2.7                        gtest/1.11.0-1
gtest/1.13.0-1                 openmpi/4.1.5_slurm
---------------------------------- Programming ------------------------------------
Java/15u36                     Julia/1.9.2                    Python/3.9.10
Qt/5.12.10                     TclTk/8.6.9                    Tcl/8.6.9
Tk/8.6.9                       anaconda/2019.07               autoconf/2.69
automake/1.16.1                binutils/2.37                  bison/3.8.2
clang/12.0.0_rhel7             cmake/3.26.3                   cuda/12.1.1
gcc/13.1.0                     gdb/13.1                       idl/8.5
intel/22.2                     libtool/2.4.2                  libtool/2.4.6
...
```

In the above case, two MPI implementations are available on the system as modules, namely the mpi/mpich2-1.0.4-mpd and mpi/mpich2-1.0.5. Load one of them, for example:

```
> module load gcc/13.1.0 openmpi/4.1.5_slurm
```

and try to find a compiler for MPI again.

If you do not have a C++ compiler, you can not continue. Ask your system administrator to install it for you. If you do not have a compiler for MPI you might still compile a sequential version, although it is not recommended.

### 2.2.2 Creating `Makefiles`

Let's assume that you have `C++` and `MPI` and imagine that you ran `git clone https://github.com/Niceno/PSI-BOIL` command in directory `~/Development`. In such a case, PSI-Boil will reside in:

```
~/Development/PSI-Boil
```

and let's call this directory the *root* directory, for the sake of shortness. (Just to make sure: inside the root directory, there are various sub-directories, such as: `Benchmarks`, `Src`, `Doc`, etc. and files such as: `AUTHORS`, `NEWS`, `README` ...).

The creation of `Makefile`'s for PSI-Boil is under control of `autotools`. It is beyond the scope of this tutorial to describe `autotools`, but there is a script residing in root directory, called `first`, which should be ran in order to create `Makefiles`. It can be invoked with:

```
> ./first OPEN
```

to create `Makefiles` for `openmpi` or:

```
> ./first INTEL
```

if you use `intel` compiler. If you, unfortunately, do not a `C++` compiler for `MPI`, create `Makefiles` with:

```
> ./first NOMPI
```

(The difference between these three invocations is merely a name of the `MPI C++` compiler. In case of `openmpi`, the c++ compiler is called `mpiCC`, whereas in the case of `intel`, it is called `mpiicpc`. If you have neither `openmpi` nor `intel`, you can edit the scrip `first` to add more library, or platform options.) In case if you choose the `NOMPI` option, `C++` compiler is set to `GNU`'s `g++`. That should be installed on any Linux-based computer.

No matter which of the above options you choose, the creation of `Makefile`'s creates an output on the screen similar to:

```
...
checking for mpicc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of mpicc... gcc3
checking whether we are using the GNU C++ compiler... yes
checking whether mpiCC accepts -g... yes
checking dependency style of mpiCC... gcc3
checking for a BSD-compatible install... /usr/bin/install -c
checking for ranlib... ranlib
configure: creating ./config.status
config.status: creating Makefile
config.status: creating Src/Domain/Makefile
config.status: creating Src/Global/Makefile
config.status: creating Src/Scalar/Makefile
config.status: creating Src/Vector/Makefile
config.status: creating Src/Grid/Makefile
config.status: creating Src/Parallel/Makefile
config.status: creating Src/Parallel/Out/Makefile
...
```

### 2.2.3 Compilation

`Makefile`'s for PSI-Boil, created in Sec. 2.2.2, assume that the *main* program is in the file called `main.cpp`, which resides in `Src` sub-directory of the root directory. For convenience, we will call it the *source* directory and, following examples from previous sections, it will be:

```
~/Development/PSI-Boil/Src
```

The source directory is important because compilation is performed in it. Therefore, you should now go to source directory.

There is not `main.cpp` (needed by `Makefile`'s in source directory, so you should either create it, copy it from another location, or even create a link in source directory. The last option is recommended. So, assuming you are already in the source directory, run the command:

```
> ln -i -s ../Doc/Tutorial/Volume1/Src/02-01-main.cpp main.cpp
```

When you have the desired main program in the source directory, you just run:

```
> make
```

and wait few minutes. PSI-Boil grew to be a complex and long program and compilation requires patience. That is true only for the first build, as for the sub-sequent, only the sources which have been modified are re-compiled, thus greatly reducing the compilation time.

During the compilation, the screen will be filled with messages such as:

```
Making all in Variable/Staggered/Momentum
make[1]: Entering directory `/home/niceno/Development/PSI-Boil/Src/Variable/Staggered/Mom
entum'
if mpiCC -DHAVE_CONFIG_H -I. -I. -I../../../..     -g -O2 -MT momentum.o -MD -MP -MF ".de
ps/momentum.Tpo" -c -o momentum.o momentum.cpp; \
then mv -f ".deps/momentum.Tpo" ".deps/momentum.Po"; else rm -f ".deps/momentum.Tpo"; exi
t 1; fi
if mpiCC -DHAVE_CONFIG_H -I. -I. -I../../../..     -g -O2 -MT momentum_bulk_bct.o -MD -MP
 -MF ".deps/momentum_bulk_bct.Tpo" -c -o momentum_bulk_bct.o momentum_bulk_bct.cpp; \
then mv -f ".deps/momentum_bulk_bct.Tpo" ".deps/momentum_bulk_bct.Po"; else rm -f ".deps/
momentum_bulk_bct.Tpo";exit 1; fi
if mpiCC -DHAVE_CONFIG_H -I. -I. -I../../../..     -g -O2 -MT momentum_bulk_ijk.o -MD -MP
 -MF ".deps/momentum_bulk_ijk.Tpo" -c -o momentum_bulk_ijk.o momentum_bulk_ijk.cpp; \
then mv -f ".deps/momentum_bulk_ijk.Tpo" ".deps/momentum_bulk_ijk.Po"; else rm -f ".deps/
momentum_bulk_ijk.Tpo";exit 1; fi
if mpiCC -DHAVE_CONFIG_H -I. -I. -I../../../..     -g -O2 -MT momentum_cfl_max.o -MD -MP
-MF ".deps/momentum_cfl_max.Tpo" -c -o momentum_cfl_max.o momentum_cfl_max.cpp; \
then mv -f ".deps/momentum_cfl_max.Tpo" ".deps/momentum_cfl_max.Po"; else rm -f ".deps/mo
mentum_cfl_max.Tpo"; exit 1; fi
...
```

Once the compilation is done, the executable called `Boil` will reside in the source directory.

### 2.2.4 Running the code

You can run the executable from the source directory by:

```
> ./Boil
```

Try it. It will do nothing. You should even try the parallel version:

```
> mpiexec -np 2 ./Boil
```

It also does nothing. It is no wonder, since the main program has very little action to it. It looks like:

```
1 #include "Include/psi-boil.h"
2
3 /******************************************************************************/
4 main(int argc, char * argv[]) {
5
6   boil::timer.start();
7
8   boil::timer.stop();
9 }
```

The thing which immediately reveals this is different to ordinary C++ program is in line 1. This line includes definitions of all PSI-Boil objects. One of the *global* PSI-Boil's objects, `timer`, is also used in this program[3]. It is stared in line 6 and stopped in line 8. The usage of `timer` will be explained below, with more elaborate examples. The program also reveals the *namespace* defined with PSI-Boil, that is `boil`. This namespace is used to enclose all PSI-Boil's global objects.

Section 2.2 in a nutshell

- The following software is required for compilation of PSI-Boil:
  - a C++ compiler (Gnu's g++ will do the job),
  - `autotools`,
  - `make` facility.

and is highle desirable if you have:

  - `MPI` libraries, as well.

On most Linux-based computers, all of the above software is available by default.

- To compile the PSI-Boil, you have to perform the following steps:
  - execute `first LAM/MPICH` script in root directory (`.../PSI-Boil`),
  - create `main.cpp` in source directory (`.../PSI-Boil/Src`), or link it to an already existing program,
  - run `make` in *source* directory.

[3] It should be clear that a *global* PSI-Boil object does not mean global *class*, but an instance of object created when PSI-Boil starts, accessible from any part of the code.

# Chapter 3

# PSI-Boil's Terminal Output

## 3.1 Standard terminal output

In this section, we will develop a *classical* C/C++ application, dating back to Karnaghan & Ritchie famous book on C programming language, *Hello World*. It is an application which does noting but prints the message:

```
Hello World!
```

on the terminal and yet it serves to illustrate some important features of the language. In this tutorial, however, we will not use the standard C++ language, but we will do it inside the PSI-Boil and we will use it to illustrate PSI-Boil's terminal output. We will use the program introduced in section 2.2 as a starting point. For this example, we introduce one more line which prints the desired message:

```
 1 #include "Include/psi-boil.h"
 2
 3 /*****************************************************************************/
 4 main(int argc, char * argv[]) {
 5
 6   boil::timer.start();
 7
 8   boil::aout << "Hello World!" << boil::endl;
 9
10   boil::timer.stop();
11 }
```

For your convenience, this program is already written, and to compile it, you have to link it from the *source* directory (`...PSI-Boil/Src`) with the command:

```
> ln -i -s ../Doc/Tutorial/Volume1/Src/03-01-main.cpp main.cpp
```

Once you make a link, remove the object file (`main.o`) from the source directory and re-make, *i.e.*:

```
> rm -f main.o
> make
```

This will create the executable `Boil` in the source directory. If you execute it, by issuing command:

```
> ./Boil
```

from the source directory, you will get the following output:

```
Hello World!
```

on the terminal. As you have probably guessed, the program line which creates this message is line number 8. In standard C++, one would write:

```
8   std::cout << "Hello World!" << std::endl;
```

which is, of course, still possible in PSI-Boil, but is not recommended. PSI-Boil has its own *global* objects for terminal output, suited for parallel execution and execution on high-performance computing platforms. These objects are:

- `boil::aout` - acronym for all `out`, meaning that all processors print;
- `boil::oout` - acronym for one `out`, meaning that only one processor prints;
- `boil::endl` - PSI-Boil's internal *end of line*. (This object was introduced to overcome very slow execution of output on some high-performance computational platforms. Namely, the C++ standard `std::endl` was slowing down the output on Cray-XT3 computer by two orders of magnitude. The problem disappears in newer Cray computers.)
- `boil::pid` - processor i.d. for printing. (Not an integer value holding the processor number).

Now try to execute the application on four processors. Use the command:

```
> mpiexec -np 4 ./Boil
```

As you could expect, you get the following output:

```
Hello World!
Hello World!
Hello World!
Hello World!
```

Each processor wrote its own message on the screen. You may wish to find out which processor wrote each of the messages. To find that out, you may use the `boil::pid`. Replace line 8 by:

```
8   boil::aout << boil::pid << "Hello World!" << boil::endl;
```

This program is already written and can be retrieved by:

```
> ln -i -s ../Doc/Tutorial/Volume1/Src/03-02-main.cpp main.cpp
```

After linking, remove the `main.o` and recompile the program with `make`. Re-run the program on four processors. You might get the output like this:

```
0: Hello World!
1: Hello World!
3: Hello World!
2: Hello World!
```

This time, you get processor i.d. in front of each message. You may wonder, at this point, what was the reason to introduce all these new objects just to write messages on the screen, processors i.d.'s and end of lines (`boil::endl`). Well, the reason is *encapsulation* **check**. PSI-Boil hides the details of implementation from the end-user (or developer). If `MPI` standard is replaced with something else in the future, only the PSI-Boil objects for terminal output will have to be changed (re-coded) while for the rest of the code, which was using PSI-Boil's global objects, nothing has to be changed.

Or, imagine a new hardware platform is developed in a number of years, for which the end of line (`endl`) has to be defined in a different way for efficiency. It will only have to be changed in one place, *i.e.* in the definition of `boil::endl`, while the rest of the code which uses that global object to end the line can remain unchanged.

Let's assume now you want to print a different message. For example, if you want to be sure you have reached the main body of the program, you would want to write something like:

```
8   boil::aout << "Inside the PSI-Boil!" << boil::endl;
```

This program is in (`../Doc/Tutorial/Volume1/Src/03-03-main.cpp`). Re-link it, recompile and run on one processor to get the message:

```
Inside the PSI-Boil!
```

If you run it on eight processors, with:

```
> mpiexec -np 8 ./Boil
```

you will get:

```
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
Inside the PSI-Boil!
```

Clearly, this is an overkill. Some output should be written by one processor only. To that end, the object `boil::oout` should be used. If the line 8 is replaced with:

```
8   boil::oout << "Inside the PSI-Boil!" << boil::endl;
```

(it is in `../Doc/Tutorial/Volume1/Src/03-04-main.cpp`). Link to this source, recompile and run. No matter how many processors you use, you will always get one line of output:

```
Inside the PSI-Boil!
```

Section 3.1 in a nutshell

- PSI-Boil's terminal output should be performed using the objects:

  - `boil::aout` - if all processors print,
  - `boil::oout` - if one processor prints,
  - `boil::endl` - to end the line,
  - `boil::pid` - for printing processor's i.d.

- The usage of standard `C++` output facilities, such as `std::cout`, `std::endl`, as well as `MPI`'s standard commands for getting processor i.d. is *discouraged*.

## 3.2 Development terminal output

The objects for terminal output introduced in 3.1 are most useful when the program is already well developed, and we want to enrich its execution with meaningful messages.

During the program development and debugging, however, we may wish to have more detailed messages. Not only the processor i.d. would be useful, but also the source file from which the message is written, as well as the line from which it was invoked. PSI-Boil has four macros which do just that:

- `AMS(x)` - All processors print the `MeSsage` `x`,
- `OMS(x)` - One processor prints the `MeSsage` `x`,
- `APR(x)` - All processors `PRint` the value of variable `x`,
- `OPR(x)` - One processor `PRints` the value of variable `x`,

Let's imagine we wanted to obtain the same message as in the last example of section 3.1 (`Inside the PSI-Boil!`) but in the *development*-like fashion. We should use the following program:

```
 1 #include "Include/psi-boil.h"
 2
 3 /******************************************************************************/
 4 main(int argc, char * argv[]) {
 5
 6   boil::timer.start();
 7
 8   OMS("Inside the PSI-Boil!");
 9
10   boil::timer.stop();
11 }
```

(This program resides in `../Doc/Tutorial/Volume1/Src/03-05-main.cpp`). Following procedures described in previous sections, link this program to `main.cpp` in the source directory and re-compile it. Do not forget to remove `main.o` before compilation.

Once you have the executable (always called `Boil`, residing in source directory), run it on one processor, and you will get the output:

```
FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
```

This is quite useful. It prints the message you want, but it also includes the file (`main.cpp` in this case) and line number (8 here) from which the message is printed.

While developing parallel programs, it is very useful to know which processor prints the message as well. This is achieved with macro `AMS`. To try it, change the line 8 to:

```
8   AMS("Inside the PSI-Boil!");
```

(This program is in `../Doc/Tutorial/Volume1/Src/03-06-main.cpp`). If you re-compile the program and run it on one processor, you will get:

```
PROC: 0, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
```

The same as above, but including the processor number which printed the message. As you may expect, of you run the program on eight processors, you will get:

```
PROC: 0, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 1, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 2, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 3, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 4, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 5, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 6, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
PROC: 7, FILE: main.cpp, LINE: 8, "Inside the PSI-Boil!"
```

Needless to say, these kind of messages are important when checking if all processors reached certain portion of the code.

While macro's `AMS` and `OMS` are useful for printing only messages, the remaining two (`APR` and `OPR`) print values of variables passed to them. Take the following program for example (`../Doc/Tutorial/Volume1/Src/03-07-main.cpp`):

```
 1 #include "Include/psi-boil.h"
 2
 3 /******************************************************************************/
 4 main(int argc, char * argv[]) {
 5
 6   boil::timer.start();
 7
 8   int a;
 9
10   APR(a);
11
12   boil::timer.stop();
13 }
```

In line 8, new variable `a` is introduced and *no* value is assigned to it. It's value is printed, in development-like manner, from line 10. Since the variable is not initialized, the output is in effect system, compiler, vendor (you name it) dependent, but on my system it looks like:

```
PROC: 0, FILE: main.cpp, LINE: 10, a = 2326516
PROC: 1, FILE: main.cpp, LINE: 10, a = 10477556
PROC: 2, FILE: main.cpp, LINE: 10, a = 10477556
PROC: 3, FILE: main.cpp, LINE: 10, a = 10477556
PROC: 4, FILE: main.cpp, LINE: 10, a = 10477556
```

```
PROC: 5, FILE: main.cpp, LINE: 10, a = 10477556
PROC: 6, FILE: main.cpp, LINE: 10, a = 10477556
PROC: 7, FILE: main.cpp, LINE: 10, a = 10477556
```

Note that variable `a` has different values on different processors. This example illustrates two important points:

- it is very dangerous to use uninitialized variables,
- for parallel programs it is particularly dangerous, since uninitialized variable might have different values on different processors[1].

The usage of the remaining macro (`OPR`) is straightforward and is not covered in this tutorial.

Section 3.2 in a nutshell

- PSI-Boil macro's for terminal output useful during the program development are:
  - `AMS(x)` - all processors print the message `x`,
  - `OMS(x)` - one processor prints the message `x`,
  - `APR(x)` - all processors print the value of variable `x`,
  - `OPR(x)` - one processor prints the value of variable `x`,
- Furthermore, it is worth stressing that:
  - it is very dangerous to use uninitialized variables,
  - which is particularly pronounced for parallel programs.

[1] An object-oriented (OO) enthusiast might argue at this point that one should not use simple types such as integers, but everything should be defined as an object since then the object's constructor would be responsible for variable initialization. Although it is true, I felt that defining everything as object, even integers and floating point numbers, could lead to excessive invocation of their constructors (and destructors) inside various PSI-Boil's loops and therefore render an inefficient program. Furthermore, these objects would presumably hinder compiler optimizations in numerically intensive parts of the code (vector-dot and matrix-vector products). Therefore, such an OO-purism has no place in numerical simulations.

# Chapter 4

# Timing PSI-Boil

Numerical solution of three-dimensional unsteady flow phenomena is very CPU-time demanding. Therefore, both developers and users of CFD programs are constantly concerned about the efficiency of their programs. From user's point of view, the total time needed to resolve a certain problem is most important, while the developer might be interested in more detailed information, on efficiency of certain parts of the code, comparison of different algorithms, suitability for different computer architectures and alike.

PSI-Boil provides means to measure both the overall execution time of the program, and to fine measure parts of the algorithms, whether it means whole subroutines, or certain programming constructs. These time-measurements are performed using the PSI-Boil's global object `boil::timer`, briefly introduced in section 2.2.

## 4.1 Global timing

The examples considered so far executed almost instantly and it does not make sense to measure their performance. Here is a program example which has a cognizable CPU-time of execution (`04-01-main.cpp`)[1]:

```
 1 #include "Include/psi-boil.h"
 2
 3 const int N = 256;
 4
 5 /******************************************************************************/
 6 main(int argc, char * argv[]) {
 7
 8   boil::timer.start();
 9
10   real a;
11
12   boil::oout << "Running ..." << boil::endl;
13
14   for(int i=0; i<N; i++)
15     for(int j=0; j<N; j++)
16       for(int k=0; k<N; k++) {
17         a = acos(-1.0);
```

[1] From this point on, the example program names will be given without their location, because it is assumed they all reside in directory: `PSI-Boil/Doc/Tutorial/Volume1/Src`. Furthermore, the procedures for linking them into source directory and compiling them will not be repeated.

```
18       }
29
20   boil::timer.stop();
21   boil::timer.report();
22 }
```

The central part of this program is nested loop in lines 14–16 which computes $acos(-1.0)$. Clearly, this program will calculate *acos* (a relatively time-consuming operation) for $256^3$ times. The commands `boil::timer.start();` and `boil::timer.stop();` (lines 8 and 20 respectively) start and stop the global timer. In addition to these, we also use `boil::timer.report();`, which will report (print on the terminal) the time needed to execute this program. Compile and run this program, to get an output such as this[2]:

```
Running ...
+========================================
| Total execution time: 5.14 [s]
+----------------------------------------
```

As you can see PSI-Boil reports on the total CPU-time needed for program execution. You only had to use commands:

- `boil::timer.start();` - as the first,
- `boil::timer.stop();` - as penultimate, and
- `boil::timer.report();` - as the last statement in the program.

If you call `boil::timer.stop();` before `boil::timer.start();`, or `boil::timer.report();` before any of the other two, error messages will be printed on the terminal.

Although the main topic of this section is timing of PSI-Boil programs, two more issues deserve attention at this point:

- Line 10: `real a;` - declaration of floating point number which is not one of the standard `C++` data types (neither `float` nor `double`). It is essentially a macro, defined in: `Global/global_precision.h` as one of the standard floating point types. The reason for introducing it, is to make precision of PSI-Boil independent of compiler options, which may be tedious to remember for different compilers or platforms. The usage of `MPI` libraries (which have their own definitions of single and double precision numbers) makes things even less transparent. With macro for `real`, things are quite straightforward: if you want the entire PSI-Boil to run in single precision, define `real` as `float`, if you want it in double precision, just define `real` as `double`. Use of standard declarations is *strongly discouraged*, since it would lead to inconsistencies between different parts of the program.
- Line 3 defines an integer constant which sets the limits for loops in lines 14–16. That is a good programming practice. If you kept number 256 in the loops itself, every time you want to change the problem size, you would have to change all three lines, not to mention that in a bigger program you might forget what all these numbers mean. Such *ghost* numbers greatly reduce the modifiability of the code and should be avoided by any means.

[2] The actual time in seconds will differ on your computer, of course

Section 4.1 in a nutshell

- To measure the total CPU-time used by PSI-Boil use commands:
  - `boil::timer.start();` - as the first,
  - `boil::timer.stop();` - as the penultimate, and
  - `boil::timer.report();` - as the last statement in the program.

- Floating point numbers are defined by macro `real`, which is defined to be on of the standard C++ data types `float` or `double`.

- Use of *ghost* numbers should be avoided.

## 4.2 Local timing

In the section 4.1 we measured the total time needed for program execution. Imagine you want to compare two algorithms, one using the *acos* function, and the other using *cos* (04-02-main.cpp)[3]:

```
 1 #include "Include/psi-boil.h"
 2
 3 const int N = 256;
 4
 5 /******************************************************************************/
 6 main(int argc, char * argv[]) {
 7
 8   boil::timer.start();
 9
10   real a;
11   real b;
12
13   boil::oout << "Running acos ..." << boil::endl;
14
15   /* algorithm using "acos" */
16   for(int i=0; i<N; i++)
17     for(int j=0; j<N; j++)
18       for(int k=0; k<N; k++) {
19         a = acos(-1.0));
20       }
21
22   boil::oout << "Running cos ..." << boil::endl;
23
24   /* algorithm using "cos" */
25   for(int i=0; i<N; i++)
26     for(int j=0; j<N; j++)
27       for(int k=0; k<N; k++) {
28         a = cos(-1.0);
29       }
30
31   boil::timer.stop();
32   boil::timer.report();
```

[3]These algorithms obviously serve no real purpose but are used solely to illustrate timing features

```
33 }
```

If you compile and run it, you will get the following output:

```
Running acos ...
Running cos ...
+=========================================
| Total execution time: 7.21 [s]
+-----------------------------------------
```

The execution time is measured as above, but the information on the total time is not quite useful now. You would like to see how much time each of the algorithms (*acos* and *cos*) consumes. Object `boil::timer` can be used for that purpose too. To achieve that, you must enclose the parts of the code you would like to measure with `boil::timer.start()` and `boil::timer.stop()`, but with the *names* of the parts of the algorithm sent as parameters. To illustrate it, lines 15–29, should be replaced by the following lines (`04-03-main.cpp`):

```
15   /* algorithm using "acos" */
16   boil::timer.start("acos algorithm");
17   for(int i=0; i<N; i++)
18     for(int j=0; j<N; j++)
19       for(int k=0; k<N; k++) {
20         a = acos(-1.0);
21       }
22   boil::timer.stop("acos algorithm");
23
24   boil::oout << "Running cos ..." << boil::endl;
25
26   /* algorithm using "cos" */
27   boil::timer.start("cos algorithm");
28   for(int i=0; i<N; i++)
29     for(int j=0; j<N; j++)
30       for(int k=0; k<N; k++) {
31         a = cos(-1.0);
32       }
33   boil::timer.stop("cos algorithm");
```

The lines which are different to previous program (`04-02-main.cpp`) are in lines: 16, 22, 27 and 33. Lines 16 and 22 enclose the *acos* algorithm and send the name `"acos algorithm"` as parameter, while the lines 27 and 33 enclose the *cos* algorithm and send the name `"cos algorithm"` as parameter. These names are arbitrary, but must be *unique*. Furthermore, each `boil::timer.start(char * name);` must have a corresponding `boil::timer.stop(char * name);`. If you compile and run this program, you will get the following output:

```
Running acos ...
Running cos ...
+=========================================
| Total execution time: 7.05 [s]
+-----------------------------------------
| Time spent in acos algorithm: 5.91 [s]    (83.8298%)
| Time spent in cos algorithm : 1.14 [s]    (16.1702%)
| Time spent elsewhere: 0 [s]    (0%)
+-----------------------------------------
```

This output quite apparently shows how much time was spent in each part of the code that you wanted to measure. The names you have assigned in calls to `boil::timer.start("name")` and `boil::timer.stop("name");` (`"acos algorithm"` and `"cos algorithm"`) appear in the output,

showing how much time was spend in each of them and what percentage of total CPU-time they consume. This information is a clear indication to parts of the code which need optimization for speed. In other words, PSI-Boil has built-in *profiling* capability. All these features work for parallel version as well. The information which parallel version would print on the terminal is the average it spent over all processors. If you wanted to have separate information for each processor, you should use `boil::timer.report_separate();` instead of `boil::timer.report();`.

It was mentioned above that each call to `boil::timer.start(char * name);` requires a call to `boil::timer.stop(char * name);`. It does not have to mean that number of `boil::timer.start(char * name);` is always equal to number of `boil::timer.stop(char * name);` in a program unit. Imagine you are timing a subroutine which has two possible exits, typical for iterative algorithms[4]:

```
 1 void iterative(const int n, const real res) {
 2
 3   boil::timer.start("iterative");
 4
 5   /* start the iterative procedure */
 6   for(int it=0; it<n; it++) {
 7      ...
 8      ...
 9      ...
10      if(residual < res) {
11        boil::oout << "algorithm converged!" << boil::endl;
12        boil::timer.stop("iterative");
13        return;
14      }
15   }
16
17   boil::oout << "algorithm failed to converge!" << boil::end;
18   boil::timer.stop("iterative");
19   return;
20 }
```

In this example, the whole subroutine `iterative(const int, const real)` is measured. Timing starts in line 3, but it can end in two ways. If the desired criteria is met inside the iterative procedure (lines 6–15), you exit from the subroutine in line 13. If the iterative procedure fails to meet the desired criteria, you exit from the subroutine in line 20. A call to `boil::timer.stop(char * name);` must be present for each of these cases. In the above example it is. In lines 12 and 18.

Section 4.2 in a nutshell

- To measure the CPU-time spent in one part of the code, enclose this part with commands:

  - `boil::timer.start(char * name);`
  - `boil::timer.stop(char * name);`

  where `name` is a *unique* name, and it will appear in the output of the final call to `boil::timer.report()`.

- Each call to `boil::timer.start(char * name);` *must* have a corresponding call to `boil::timer.stop(char * name);`.

[4] This is just an illustrative example, not a real program which could be compiled inside the PSI-Boil. It's source is therefore not available in a separate file

# Chapter 5

# Creating Computational Domains

PSI-Boil currently supports only three-dimensional Cartesian grids. Three-dimensional grids are represented with the class `Domain`. This class holds all the geometric data needed for discretization of governing equations, and is built from three one-dimensional grids; each one defining discretization in one coordinate direction. These, one-dimensional grids, used for building three-dimensional domains are defined by class `Grid1D`.

In this chapter, you will learn how to create one-dimensional grids with the class `Grid1D`, and how to use them to define three-dimensional domains.

Furthermore, you will learn about the so-called *ravioli* classes in PSI-Boil, and use two of them: `Range` and `Periodic`.

## 5.1 One-dimensional grids

One-dimensional grids (`Grid1D`) are building blocks for three-dimensional computational domains (`Domain`). A one-dimensional grid is a set of points defined along a line segment, parallel to a particular coordinate direction. These points, usually referred to as *nodes*, divide the line segment in a number of smaller sub-segments, also called *cells*. In the cell-centered finite volume method, used in PSI-Boil for discretization of governing equations, the unknowns are placed in centers of the cells.

### 5.1.1 Uniform grid

Imagine you want to create a one-dimensional grid, along the coordinate axis $\xi$, having length $L = 2.0$, starting at $\xi = -0.25L$, ending at $\xi = 0.75L$, divided in 16 cells. The cells have equal size, i.e. the grid is *uniform*. Such a grid can be created in PSI-Boil with the following program (`05-01-main.cpp`):

```
1 #include "Include/psi-boil.h"
2
3 const real L = 2.0;
4 const int  N = 16;
5
6 /******************************************************************************/
7 main(int argc, char * argv[]) {
8
9   boil::timer.start();
```

```
10
11   /* non-periodic grid */
12   Grid1D grid( Range<real>(-0.25*L, 0.75*L), N, Periodic::no());
13
14   // grid.print();
15   // grid.plot("grid.eps");
16
17   boil::timer.stop();
18   boil::timer.report();
19 }
```

The grid is represented by the object `grid`, and is defined in line 12. Program line 12 calls a constructor for grid, passing three parameters:

- `Range<real>(-0.25*L, 0.75*L)` - start and end $\xi$ coordinate of the grid,
- `N` - number of grid cells (*not* nodes),
- `Periodic::no()` - indicator that the grid is non-periodic.

The first parameter sends the start and end $\xi$ coordinate, but they are enclosed inside another object of type `Range`. As it's name implies, `Range` is a class which defines certain range, whether a range of integer or real numbers[1]. Enclosing the starting and ending coordinate inside one object has been done to group the similar items together (in one parameter) and, closely connected with that, reduction of total number of parameters.

Second parameter, `N`, requires little explanation. It is an integer which specifies number of cells in the grid.

Third parameter is a switch, telling `Grid1D`'s constructor whether the grid is periodic or not. It is passed with the special object of type `Periodic`, which has only two member functions: `yes()` and `no()`. Their usage is quite intuitive: If a grid is periodic, one would send `Periodic::yes()` as a parameter, and if it is not, one sends `Periodic::no()`.

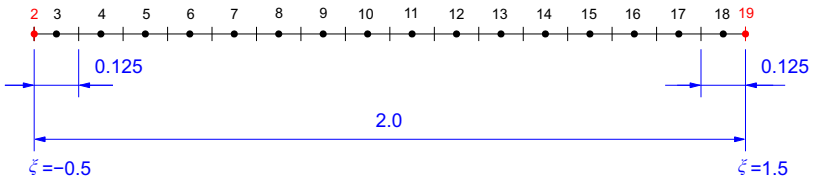



Figure 5.1: Uniform one-dimensional grid: • domain cells, + domain nodes, • boundary cells, + boundary nodes. Coordinates and dimensions are in blue.

The grid created by this program is shown in Fig. 5.1. First thing worth nothing is that there are 16 cells inside the domain (shown by black dots and numbered from 3-18). But, in addition to cells in the domain, PSI-Boil creates additional cells at the boundaries (red dots, numbered 2 and 19). These *boundary* cells hold the values of boundary conditions or serve as *buffer* cells for parallel version. Note that the cell index starts with 3 instead of 1! This is due to the strategy in parallel computation, i.e. there are three layers of buffer cells, which will be explained in the next section. In addition, note that, for the case of non-periodic grid, boundary cells coincide with boundary nodes.

[1] Actually it could be defined for any types of objects which can be related with greater (>), smaller (<) or equal.

### 5.1.2 Periodic grid

If you want to create a periodic, line 12 from the above program should be changed to (`05-02-main.cpp`):

```
11   /* periodic grid */
12   Grid1D grid( Range<real>(-0.25*L, 0.75*L), N, Periodic::yes());
```

This program, when compiled and ran, creates the grid shown in Fig. 5.2. Dimensions of the domain cells are the same as in previous example, but boundary cells do *not* coincide any longer with boundary nodes. Boundary cell 2 is now a copy of domain cell 18, shifted by $L$ in negative $\xi$ direction, while cell 19 is a copy of cell 3, shifted by $L$ in positive $\xi$ direction. In the same way, cell 0 is a copy of cell 16, cell 1 is cell 17, cell 20 is cell 4, and cell 21 is cell 5.

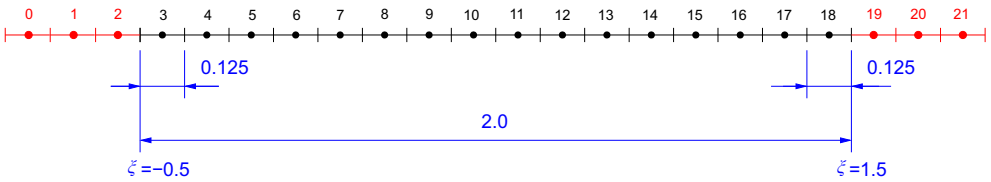



Figure 5.2: Periodic grid: • domain cells, + domain nodes, • boundary cells, + boundary nodes. Coordinates and dimensions are in blue.

No matter if the grid is periodic or not, PSI-Boil adds additional cells on the boundaries. The number of cells passed by the user, however, is equal to the number of *computational* cells.

### 5.1.3 Stretched grid

The grids created in Sec˙ 5.1.1 and 5.1.2 were both uniform. Many problems require grid to be stretched towards the regions where important phenomena is occurring at smaller scales to resolve it more accurately. To create such *stretched* grids we need to pass the desired cell dimension at the beginning and the end of the grid to `Grid1D`'s constructor. It is illustrated by the following program (`05-03-main.cpp`):

```
 1 #include "Include/psi-boil.h"
 2
 3 const real L  = 2.0;
 4 const real D1 = 0.002;
 5 const real D2 = 0.04;
 6 const int  N  = 32;
 7
 8 /******************************************************************************/
 9 main(int argc, char * argv[]) {
10
11   boil::timer.start();
12
13   /* stretched grid */
14   Grid1D grid( Range<real>(-0.25*L, 0.75*L),
15                Range<real>(D1, D2),
16                N,
17                Periodic::no());
18
19   grid.plot("grid.eps");
20
```

```
21   boil::timer.stop();
22   boil::timer.report();
23 }
```

Stretched grid is created by calling `Grid1D` constructor in lines 14–17. Four parameters are passed to this constructor and they represent:

- `Range<real>(-0.25*L, 0.75*L)` - start and end $\xi$ coordinate of the grid,
- `Range<real>(D1, D2)` - start and end cell size of the grid,
- `N` - number of grid cells (*not* nodes),
- `Periodic::no()` - indicator that the grid is non-periodic.

First, third and fourth parameter are the same as before, while second parameter passes the desired start and end cell size. Grid created by the program (`05-03-main.cpp`) is illustrated in Fig. 5.3. As you can see, the grid is stretched towards both walls, with stretching being stronger towards the beginning of the domain.

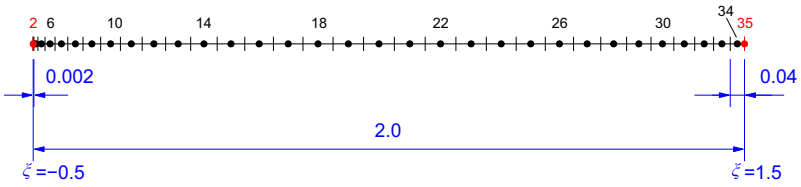



Figure 5.3: Stretched grid: • domain cells, + domain nodes, • boundary cells, + boundary nodes. Coordinates and dimensions are in blue.

## 5.2 Ravioli classes

Small objects, such as that of `Range` type, are in PSI-Boil referred to as *ravioli* objects. They are very small and have limited functionality (small number of member functions), but primarily serve for improved code readability, safer argument passing and to reduce chances for making coding errors.

Take class `Periodic`, for example. If it was not defined, one would have to send some other data types, such as integers, for example. Then, an integer would be sent to `Grid1D`'s constructor, having value 0 if non-periodic, and 1 if periodic. Or, maybe, -1 if periodic and 0 if non-periodic. Such parameters are difficult to understand, and would always have to read `Grid1D`'s documentation to recall which number means what (Did we agree 0 was periodic, or non-periodic?). Furthermore, type checking is not ensured with some simple types, integers and Boolean in particular. If one sends Boolean variable as a parameter where integer should have been sent, compiler would not notice it, resulting in hard-to-find bugs. With ravioli parameter, compiler recognizes, during compilation time, any inconsistencies in parameter types.

Most frequently used ravioli classes are all declared in directory: `PSI-Boil/Src/Ravioli`. Those used only inside one other class[2].

More details about each of the ravioli classes can be obtained either from it's Doxygen documentation (check the directory: `PSI-Boil/Doc/Dox`), or from it's header file. If you find the additional

[2]For example: type of preconditioner is defined as a ravioli class and is used only inside the preconditioner class. It would not make sense to place a ravioli class with such a limited application into `PSI-Boil/Src/Ravioli` directory.

information insufficient, or even missing, feel free to contact the author. The same goes for all other classes, of course.

Sections 5.1 and 5.2 in a nutshell

- PSI-Boil supports three-dimensional Cartesian grids, represent with class `Domain`.

- `Domain` is built from three one-dimensional grids (`Grid1D`), defining resolution in $x$, $y$ and $z$ directions.

- To create a 1D grid, use the constructor:

  - `Grid1D(Range<real>(xi_s, xi_e), N, Periodic);`

for uniform grids, or:

  - `Grid1D(Range<real>(xi_s, xi_e), Range<real>(D_s, D_e), N, Periodic);`

for non-uniform grids.

- PSI-Boil uses quite a lot of small objects, called ravioli, for improved code readability, safer argument passing and reduced chance of coding errors. Most of them are declared in sub-directory `Ravioli`.

## 5.3 Three-dimensional domains

As stated before, PSI-Boil supports three-dimensional Cartesian computational domains, defined by objects of type `Domain`. It uses three one-dimensional grids `Grid1D` to define resolution in each of the coordinate direction ($x$, $y$ and $z$).

The following program (`05-04-main.cpp`) creates a cubical domain, having dimension $1 \times 1 \times 1$, with uniform cell distribution in $x$ and $y$ direction, while stretched towards both ends in $z$ direction:

```
 1 #include "Include/psi-boil.h"
 2
 3 const real L = 3.2;
 4 const real D = 0.01;
 5 const int  N = 32;
 6
 7 /******************************************************************************/
 8 main(int argc, char * argv[]) {
 9
10   boil::timer.start();
11
12   /* uniform grid */
13   Grid1D g_uni( Range<real>(-0.5*L, 0.5*L), N, Periodic::no() );
14
15   /* stretched grid */
16   Grid1D g_str( Range<real>(0,L), Range<real>(D,D), N*2, Periodic::no());
17
18   /* create domain */
19   Domain dom(g_uni, g_uni, g_str);
20
21   /* plot the domain */
22   boil::plot = new PlotTEC();
```

```
23   boil::plot->plot(dom, "dom");
24
25   boil::timer.stop();
26   boil::timer.report();
27 }
```

Line 13 creates `g_uni`, a uniform grid with 32 cells, ranging from $\xi = -1.6$ to $\xi = 1.6$. Line 16 creates `g_str`, a grid ranging from $\xi = 0$ to $\xi = 3.2$ which is stretched towards both ends. Note that the size of cells next to the boundaries of `g_str` is $\Delta\xi = 0.01$.

Command which creates the computational domain is in line 19. This line creates an object of type `Domain`, called `dom`. It's constructor accepts three arguments: grid in $x$, $y$ and $z$ direction. In this particular case, we use grid `g_uni` for distribution in $x$ and $y$, and grid `g_str` to define distribution in $z$ direction.

Domain is plotted in lines 21–23. Plotting is performed with the global PSI-Boil object `boil::plot`. This global object can be created in two ways, depending on the format of output files you would like to create. If you would like to plot your grids (and later results) in Tecplot[3] format, create the `boil::plot` as in line 22. If you prefer output files in General Mesh Viewer[4] (GMV) format, change the line 22 to:

```
22   boil::plot = new PlotGMV();
```

When running the program `05-04-main.cpp`, you will get the following output:

```
Domain level 4 created !
Domain level 3 created !
Domain level 2 created !
Domain level 1 created !
# Plotting: dom_p000.dat
+========================================
| Total execution time: 0.18 [s]
+----------------------------------------
| Time spent in plotting: 0.18 [s]    (100%)
| Time spent elsewhere: 0 [s]    (0%)
+----------------------------------------
```

The output informs you that it has created four grid levels in addition to the one specified by the user. The level specified by the user is always level 0, while the coarser levels have larger numbers. Thus, for this particular case, level 0 will have resolution of: $32 \times 32 \times 64$, level 1: $16 \times 16 \times 32$, level 2: $8 \times 8 \times 16$, level 3: $4 \times 4 \times 8$ and, finally, level 4 will have the coarsest resolution of only $4 \times 4 \times 4$ cells.

The grid on the finest level (level 0) is plotted from line 23, using the global object `boil::plot` and its member function `boil::plot->plot()`. As the arguments, you send the `Domain` to be plotted (in this case it is `dom`) and the name you want to associate with it (`"dom"`). The plotting function (`boil::plot->plot()`) names the output file as: `dom_p000.dat`, where `_p000` is processor number and `.dat` is extension recognized by Tecplot. The grid on the finest level, visualized by Tecplot is shown in Fig. 5.4.

Note that program reports on the time spent in plotting functions with:

```
| Time spent in plotting: 0.18 [s]    (100%)
```

That is because all the plotting functions are embedded in *local* timing routines, such as the ones introduced in the section 4.2.

[3] Tecplot is a registered trademark of Tecplot Inc. (`www.tecplot.com`)

[4] `www-xdiv.lanl.gov/XCM/gmv/GMVHome.html`

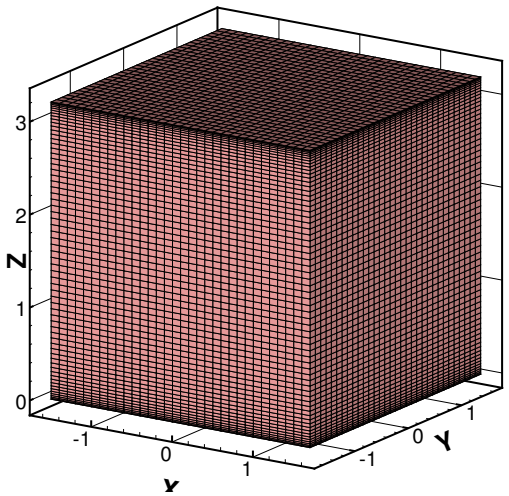



Figure 5.4: Stretched computational domain.

You have to stretch the grid towards the walls very often, but sometimes you might also want to stretch in the interior - particuluarly if there are obstacles (or inner walls) inside. A way to achieve such a stretching is to combine two stretched grids together. The way to do it is demostrated below. The lines of the `05-04-main.cpp` code are replaced with the following (to get the program `05-05-main.cpp`):

```
15   /* stretched grid */
16   Grid1D g_str( Range<real>(0,0.5*L), Range<real>(D,D), N, Periodic::no());
17
18   Grid1D g_str_2(g_str, g_str, Periodic::no());
19
20   /* create domain */
21   Domain dom(g_uni, g_uni, g_str_2);
```

`Grid1D g_str` is still created in line 16, but this time is half as short and it has half as many cells as in the previous case. Next, additional grid is created in line 18, where two `g_str`'s are added one after another. The `Grid1D` constructor in line 18 takes two `Grid1D`'s as parameters, leaves the absolute coordinates of the first intact, but disregards the absolute coordinates of the second. It shifts and attaches the grid sent as second parameter to the grid sent as first parameter[5]. Finally, `Domain dom` is created in line 21 using the new, double-stretched grid to set the resolution in $z$ direction, and is plotted in Fig. 5.5.

[5] If it was not shifting the second, they would collapse into one grid for this case.

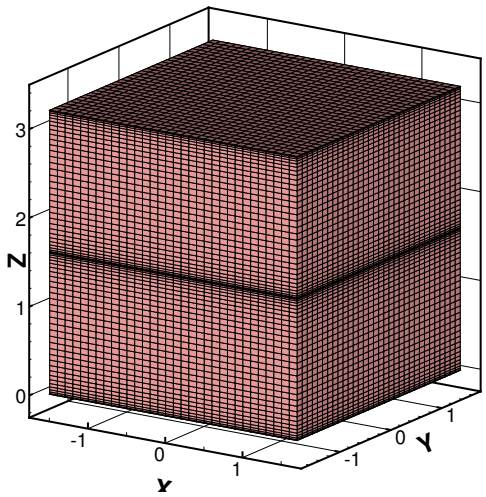



Figure 5.5: Double-stretched computational domain.

Section 5.3 in a nutshell

- Three-dimensional Cartesian grids, represent with class `Domain`, are created from three one-dimensional grids (`Grid1D`), with the constructor:

  - `Domain(Grid1D & gx, Grid1D & gy, Grid1D & gz);`

where `gx`, `gy` and `gz` represent grid resolutions in $x$, $y$ and $z$ coordinate directions.

- In order to plot domains, global object `boil::plot` must be defined before it is used.

- `boil::plot` can be defined as:

  - `boil::plot = new PlotGMV();` for plotting with GMV,

  - `boil::plot = new PlotTEC();` for plotting with Tecplot.

- Grids (`Grid1D`) can also be constructed by appending one after another, using the constructor: `Grid1D(Grid1D &, Grid1D &, Periodic)`.

- `Domains` can be plotted explicitly, using the command:

  - `boil::plot->plot(Domain & dom, "file_name");`

## 5.4 Immersed Bodies

One of the distinct features of PSI-Boil is it's ability to solve problems in complex computational domains using an immersed boundary method (IBM). For such cases, computational domain is divided into a *fluid* part, where all governing equations for heat transfer and fluid flow are solved, and the *solid* part, where no momentum equations are solved. However, in case of conjugate heat transfer problems, heat transfer equations is solved in the solid part as well.

In PSI-Boil solid parts of the computational domain are defined as *immersed boundaries*, represented by the class `Body`. These immersed bodies (IB's) are defined by a computer aided design (CAD) files in ASCII stereolitography (STL) format. The creation of CAD files is performed by a third party program. At PSI we use AC3D[6], but any other CAD software package able to export ASCII STL file format could be used. This tutorial does not cover the usage of AC3D, nor any other CAD packages. It is important, however, to mention the convention adopted in PSI-Boil concerning IB's: *normals on the body surface point to the fluid part of the domain.*

The usage of IB is illustrated in the following example. Consider a case of a domain with dimensions: $4 \times 1 \times 1$ in $x$, $y$ and $z$ direction respectively, with a step-like obstacle with dimension $0.5 \times 1.0 \times 0.5$ placed at the bottom wall (minimum $z$) of the domain. Immersed body is created in AC3D and stored in the file named `step.stl`. The dimensions of the IB are set to be slightly larger in CAD program to avoid ambiguity when cutting cells. In effect, the actual dimensions of the IB are $0.501 \times 1.001 \times 0.501$. This file should be stored in the running directory. The program which uses the STL file to create a mesh for a domain with a step-like obstacle is (`05-06-main.cpp`):

```
 1 #include "Include/psi-boil.h"
 2
 3 const real LX = 4.0;
 4 const real LY = 1.0;
 5 const int  NX =  64;
 6 const int  NY =  16;
 7
 8 /******************************************************************************/
 9 main(int argc, char * argv[]) {
10
11   boil::timer.start();
12
13   /* plot in Tecplot format */
14   boil::plot = new PlotTEC();
15
16   /* grids */
17   Grid1D g_x( Range<real>(0,LX), NX, Periodic::no() );
18   Grid1D g_y( Range<real>(0,LY), NY, Periodic::no() );
19
20   /* create immersed body */
21   Body step("05-06-step.stl");
22
23   /* plot step before immersion */
24   boil::plot->plot(step, "step-before");
25
26   /* create domain */
27   Domain dom(g_x, g_y, g_y, &step);
28
29   /* plot step after immersion */
30   boil::plot->plot(step, "step-after");
31
32   /* plot domain after immersion */
33   boil::plot->plot(dom, "dom-after");
34
35   boil::timer.stop();
36   boil::timer.report();
37 }
```

Program lines from 1–18 should be clear at this point. `Body` is created in line 21. It is defined with the STL file name containing IB definition. As soon as the `Body` is created, it is plotted from line 24. This plotting is not necessary, of course, but can be useful for checking.

[6] `http://www.inivis.com/`

Problem domain is defined in line 27. The novelty in this constructor is the final argument, a reference to `Body` (an IB). `Domain`'s constructor will cut the computational cells with IB, but also change the IB itself. Line 30 plots the IB after cutting and line 33 the final computational domain. If you compile and run this program, it will create the following output:

```
# Plotting: step-before_p000.dat
Domain level 4 created !
Domain level 3 created !
Domain level 2 created !
Domain level 1 created !
# Plotting: sca_p000.dat
# Plotting: vec_p000.dat
# Plotting: step-after_p000.dat
# Plotting: dom-after_p000.dat
+========================================
| Total execution time: 0.29 [s]
+----------------------------------------
| Time spent in plotting    : 0.24 [s]    (82.7586%)
| Time spent in bounding box: 0 [s]    (0%)
| Time spent in cell cutting: 0.02 [s]    (6.89655%)
| Time spent in flood fill  : 0.01 [s]    (3.44828%)
| Time spent elsewhere      : 0.02 [s]    (6.89655%)
+----------------------------------------
```

File containing IB before immersing (cutting) is called `step-before_p000.dat`, while the IB after immersing is stored in file `step-after_p000.dat`. They are plotted in Figs. 5.6 and 5.7, respectively.

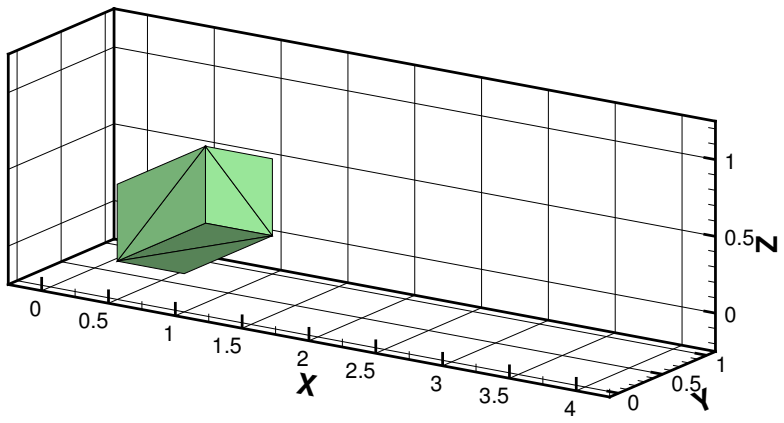



Figure 5.6: IB plotted from line 24, *i.e.* before immersing it into computational domain. At this point it is the same as it was defined in a CAD program.

In addition to the IB and domain files described above, PSI-Boil creates two additional files: `sca_p000.dat` and `vec_p000.dat`. These files hold scalar and vector respectively, representing ratio of the computational cell volume immersed in fluid. So, for cells inside the fluid part of the domain, it's value is 1, in the solid part it is 0, and for cells which are cut by immersed body, it's value is between 0 and 1. `sca_p000.dat` holds these values for scalar variables, while `vec_p000.dat` holds vector values. It is a good practice to visualize these fields, just to visually check if IB is properly handled by PSI-Boil. Visualization of `sca_p000.dat` is given in Fig. 5.9.

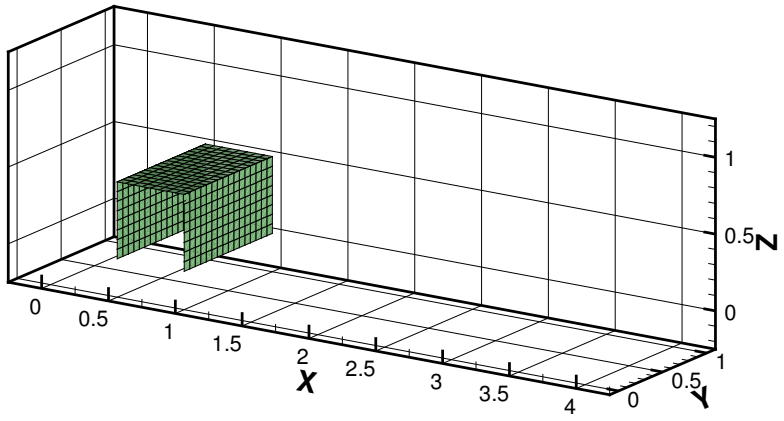



Figure 5.7: IB plotted from line 30, *i.e.* after immersing it into computational domain. At this point, the topology of the IB has changed; the original triangles are replaced by cell cuts.

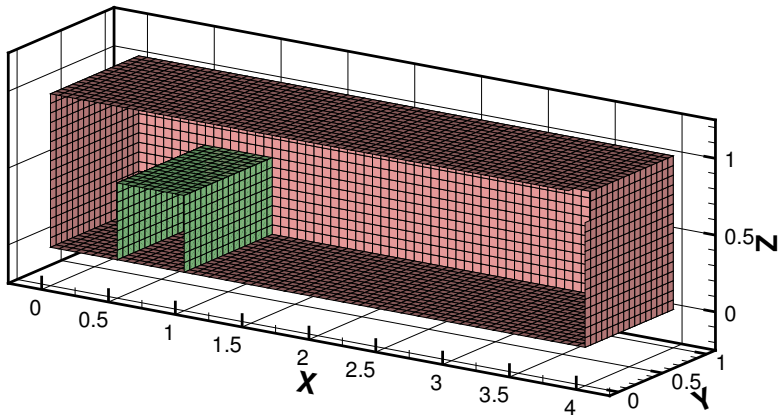



Figure 5.8: Computational domain with IB. Domain is colored in pink and obstacle in green. The front face is removed from the figure for the sake of clarity.

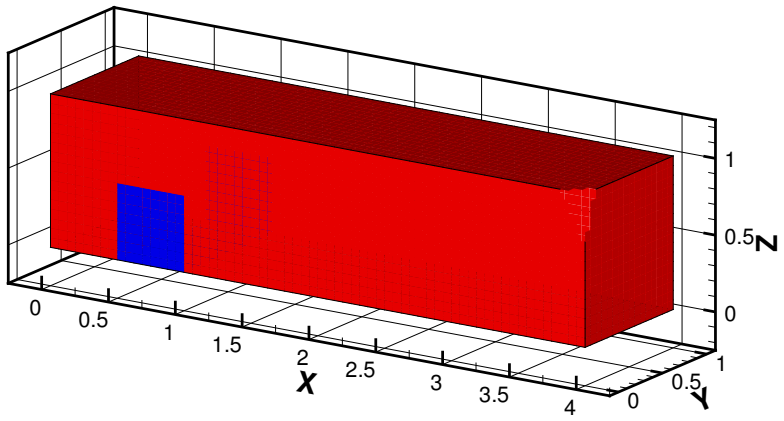



Figure 5.9: Scalar field representing volume fraction of the cell immersed in fluid.

Section 5.4 in a nutshell

- To deal with problems in complex geometries, PSI-Boil divides computational `Domain` into *fluid* and *solid* part.

- Solid part is represented with an object of type `Body`, while fluid is everywhere else.

- By convention, the CAD file representing the `Body` has surface normals oriented toward the fluid part of the computational domain.

- `Body` is an object created from a 3D CAD file as following:

  - `Body body(const std::string name);`

where `name` is the name of the CAD file in ASCII STL format.

- `Body` is inserted into a `Domain`, using the constructor:

  - `Domain(Grid1D &, Grid1D &, Grid1D &, Body *);`

`Body` is sent as a pointer to `Domain` because it changes during the process of computational cell cutting.

## 5.5 Domain decomposition

Run the previous program (`05-06-main.cpp`) in parallel on two processors. As a reminder, you do it with:

```
> mpiexec -np 2 ./Boil
```

The program will create the following output:

```
# Plotting: step-before_p000.dat
# Plotting: step-before_p001.dat
Domain level 3 created !
Domain level 2 created !
Domain level 1 created !
# Plotting: sca_p000.dat
# Plotting: vec_p000.dat
# Plotting: sca_p001.dat
# Plotting: vec_p001.dat
# Plotting: step-after_p000.dat
# Plotting: dom-after_p000.dat
# Plotting: step-after_p001.dat
# Plotting: dom-after_p001.dat
+=======================================
| Total execution time: 0.175 [s]
+---------------------------------------
| Time spent in plotting    : 0.13 [s]    (74.2857%)
| Time spent in bounding box: 0 [s]    (0%)
| Time spent in cell cutting: 0.015 [s]    (8.57143%)
| Time spent in flood fill  : 0.02 [s]    (11.4286%)
```

```
| Time spent elsewhere      : 0.01 [s]    (5.71429%)
+-----------------------------------------
```

(Remember that domains are plotted implicitly, without direct invocation). You get two domains, one for each processor. Namely, the grids are stored in `dom-after_p000.dat` for processor 0 and `dom-after_p001.dat` for processor 1. These two grids are shown in Fig. 5.10.

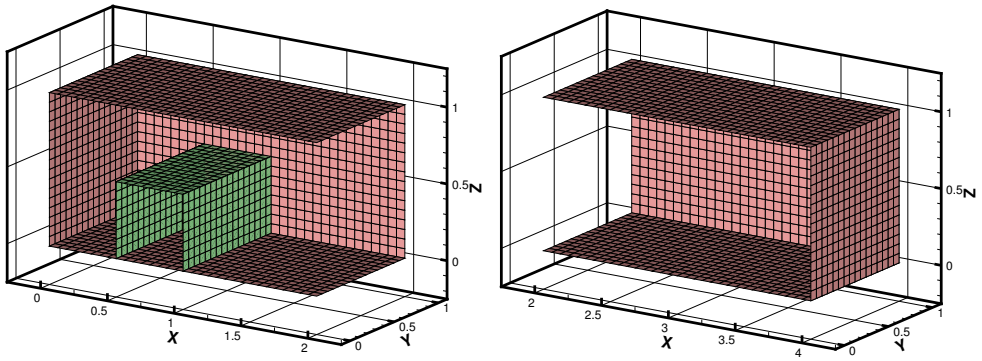



Figure 5.10: Computational domain decomposed in two sub-domains. The front face, as well as the face between the sub-domains are removed from the figure for the sake of clarity.

While it is undoubtedly interesting to see the decomposed computational domain, it is of little practical value. Much more often one wants to see the entire computational domain, together with solutions of governing equations on it. PSI-Boil, if ran in parallel, plots as many domains and result files as there were processors used. That is the simplest, but also the most efficient way to parallelize file output. To connect the files, we use an external program, called `Connect`[7]

`Connect` is a part of PSI-Boil package, and resides in directory: `PSI-Boil/Src/Connect`, hereafter referred to as *connect* directory. If you followed the instructions for creating `Makefiles` (Sec. 2.2.2), there should already be a `Makefile` in connect directory. Go there and run `make`. After a short compilation, an executable with the name `Connect` is created. Go back to source and run:

```
> ./Connect/Connect
```

It will print short message with outlining program usage, namely:

```
Usage:
Connect/Connect <base_name> <ext> <number_of_processor> [time_step/level]
```

The first item being printed is the `Connect` program name with full path. It is followed by command line arguments, which are:

- `base_name` - name assigned in call to plotting functions In the present case, it is `domain`, implicitly defined by PSI-Boil.
- `ext` - file extension, determining the file type being connected. It may be either `gmv` for GMV, or `dat` for Tecplot.
- `number_of_processor` - clearly number of processors over which the domain is distributed. For this case, it is 2.

[7] It was possible to create PSI-Boil in such a way that it creates a single result file, no matter how many processors have been used. But, that would put additional burden on parallel efficiency of PSI-Boil, only to make post-processing more convenient for the user. It does not make sense to sacrifice efficiency for convenience, particularly for an academic code such as PSI-Boil, which is not meant to beat commercial CFD packages in user-friendliness, or convenience.

- `time_step/level` - the final argument which specifies the time step, or grid level. This argument is optional, but for the present case, grid level has to be specified.

So, to connect the domain you got after the run on two processors, run:

```
> ./Connect/Connect dat dom 2 0
```

and, after messages such as these:

```
reading: dom-after_p000.dat
reading: dom-after_p001.dat
number of variables (excluding coordinates): 0
number of variables (excluding coordinates): 0
creating: dom-after.dat
body 0
present at 0
nodes 1664
cells 832
bodies finished
```

It will create the file `dom-after.dat` which holds the single grid (not distributed).

Section 5.5 in a nutshell

- When ran in parallel, PSI-Boil automatically decomposes the the computational domain.

- PSI-Boil performs separate plotting, i.e. each processors plots its own sub-domain.

- Solid part is represented with `Obstacle`s, while fluid is everywhere else.

- Sub-domains are connected into a single file, using the program `Connect`, residing in directory `PSI-Boil/Src/Connect`.

- `Connect` is not build by default. To build it, run `make` in it's directory.

# Chapter 6

# Scalar and Vector Fields

Once computational domains are defined (Sec. 5) the *fields* can be defined as well. Fields are unknown functions solved in a numerical simulation, such as: temperature, pressure, species concentration, velocity fields, and their source terms, as well as physical properties, or anything else which is defined as a variable function over a computational domain.

As PSI-Boil is based on *staggered* FV method, scalar fields are computed at cell centers, while vector fields are at the staggered grid, placed on the grid cell faces. It is therefore clear that two classes are defined in PSI-Boil to represent fields: `Scalar` which holds the scalar unknown (temperature, pressure, etc.) and `Vector` which holds velocity components.

In this chapter, you will learn how to define each of these types of fields. The most essential thing for each field is the domain for which it is defined. Field constructors are therefore quite simple, the only argument they need is a reference to `Domain`.

## 6.1 Scalars

Example program which defines scalar field `p`, assigns initial values to it, and plots it, is given below (`06-01-main.cpp`):

```
 1 #include "Include/psi-boil.h"
 2
 3 const real L =  6.2831853071796;
 4 const int  N = 64;
 5
 6 /******************************************************************************/
 7 main(int argc, char * argv[]) {
 8
 9   boil::timer.start();
10
11   /* plot in Tecplot format */
12   boil::plot = new PlotTEC();
13
14   /* grid in "x", "y" and "z" direction */
15   Grid1D g( Range<real>(0,L), N, Periodic::yes() );
16
17   /* computational domain */
18   Domain d(g, g, g);
19
20   /* define unknowns */
```

```
21   Scalar p(d); // pressure
22
23   /* assign initial values to pressure */
24   for(int i=1; i<p.ni(); i++)
25     for(int j=1; j<p.nj(); j++)
26       for(int k=1; k<p.nk(); k++)
27         p[i][j][k] = (1.0/16.0 )             *
28                      (2.0 + cos(2.0*p.zc(k))) *
29                      (cos(2.0*p.xc(i)) + cos(2.0*p.yc(j)));
30
31   /* plot scalar */
32   boil::plot->plot(p, "pressure", 0);
33
34   boil::timer.stop();
35   boil::timer.report();
36 }
```

This program actually initializes `p` according to:

$$p = \frac{1}{16}(2 + cos(2z))(cos(2x) + cos(2y)) \tag{6.1}$$

The program should be clear until line 21. Here is something new, field `p`, of type `Scalar`, is defined from domain `d`.

Since a `Domain` in PSI-Boil is a *structured* entity, it has clearly defined extensions in $x$, $y$ and $z$ coordinate directions[1]. `Domain` has build in functions which give the resolution in each direction, and these are: `Domain::ni()`, `Domain::nj()` and `Domain::nk()`. For this case, user specified resolution 64 in each coordinate direction (lines 4 and 15). But, as it was explained in Sec. 5.1, PSI-Boil adds two additional cells at the boundary of each coordinate direction, meaning that `Domain::ni()`, `Domain::nj()` and `Domain::nk()` hold the value of 66.

`Scalar`, being defined for a `Domain`, assumes the same structure as the `Domain`, and its values can be accessed with the usual `C++` syntax for three-dimensional arrays:: `p[i][j][k];` Furthermore, `Scalar` has analogue member functions: `Scalar::ni()`, `Scalar::nj()` and `Scalar::nk()` which define it's resolution[2]. Hence, to *browse* through all the values of `Scalar p` over `Domain d`, we use the triple loop in lines 24–26, using `Scalar`'s member functions `ni()`, `nj()` and `nk()` to define ranges. This loop will browse from 1, which is the first non-buffer cell in each direction, to 64, the last non-buffer cell in each coordinate direction. By doing so, we browse exactly through 64 cells in each direction, which is equal to the resolution specified by the user (line 4).

This may quite look nice at the first glance, but is actually a *very dangerous* practice. Imagine `Domain` (which also reflects the `Scalar` changes internal structure in the future, and, instead of one, adds two layer of buffer cells at each end. All the loops in the program, having the form as the one given in lines 24–26 would fail! That should not come as a surprise, because the 1's that are used to specify the loop ranges in lines 24–26, are in essence, *ghost* numbers. To avoid this dangerous practice, `Scalar` provides additional member functions: `Scalar::si()` and `Scalar::ei()` which mark the start and end of `Scalar` cell range in $i$ direction (Analogue functions exist for $j$ and $k$ directions, of course.) Therefore, much safer variant of the loop, resilient to internal changes in the objects `Domain` and `Scalar` would be:

[1] Often denoted as $i$, $j$ and $k$.

[2] The values obtained by `Scalars` `ni()`, `nj()` and `nk()` are the same as those obtained from the `Domain` for which the scalar is defined.

```
24   for(int i=p.si(); i<=p.ei(); i++)
25     for(int j=p.sj(); j<=p.ej(); j++)
26       for(int k=p.sk(); k<=p.ek(); k++)
```

If the inner structure of `Scalar` changes, it would reflected in its `si()`, `ei()`, etc. and loops as this one would still work properly.

Since PSI-Boil is a three-dimensional program, triple loops like these occur very frequently, which makes programming a tedious, lengthy and, let's face it, an error-prone task. Try to make a *small* error in line 25. Re-write it as:

```
25     for(int j=p.sj(); j<=p.ej(); i++)
```

re-compile and re-run. You get a segmentation fault, only because you mistyped `j++` as `i++`, an error which is easy to make, particularly if you have plenty of triple loops all around the code.

To avoid these sorts of errors, a special macro has been defined (`Scalar/scalar_browsing.h`), which replaces the triple loop with a single line (take the program `06-02-main.cpp`):

```
24   for_vijk(p,i,j,k)
```

which takes as parameters field over which you want to browse (`p`), as well as counters in $i$, $j$ and $k$ direction (`i`, `j` and `k`). This macro reduces the size of the program and in that way reduces the chance to make a typing error.

One more detail remains to be explained at this point. To access cell-center coordinates, `Scalars`'s member functions: `xc(int)`, `yc(int)` and `zc(int)` should be used. In addition to these, there is a number of other `Scalars`'s member functions to access geometrical data, such as cell dimension (`dxc()`, `dyc()` and `dzc()`), distance to neighboring cell centers (`dxe()`, `dxw()` ...). All relevant information about these functions is in the Doxygen documentation (`PSI-Boil/Doc/Dox`).

The pressure field prescribed by program (`06-02-main.cpp`) is shown in Fig. 6.1.

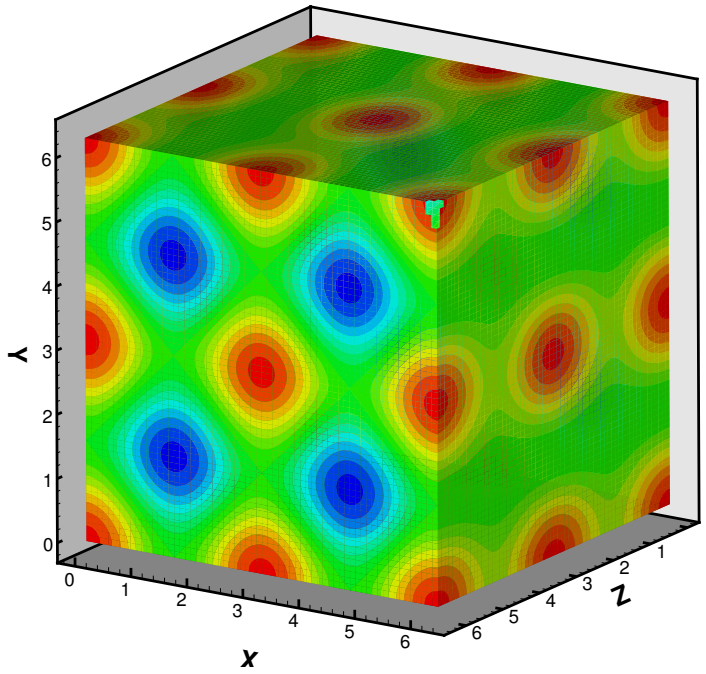



Figure 6.1: Pressure field prescribed by Eq. 6.1

Section 6.1 in a nutshell

- Scalar fields are defined with PSI-Boil class `Scalar`.

- `Scalar` is defined for a `Domain`, with the constructor:

  - `Scalar(Domain &);`

- `Scalar`'s values are accessed with the usual `C++` syntax:

  - `p[i][j][k];`

- To browse through all the `Scalar` values, use the following macro:

  - `for_vijk(Scalar &, int i, int j, int k);`

Similar macros are defined in directory: `PSI-Boil/Src/Field/Scalar/scalar_browsing`.

- `Scalar` has a number of member functions to access `Domains` geometrical quantities.

## 6.2 Vectors

Example program which defines vector field `uvw`, assigns initial values to it, and plots it, is given below:

```
...
20   /* define unknowns */
21   Vector uvw(d); // velocity
22
23   /* assign initial values to velocity in "i" direction */
24   Comp m = Comp::u();
25   for_vmijk(uvw,m,i,j,k)
26     uvw[m][i][j][k] =  sin(uvw.xc(m,i)) * cos(uvw.yc(m,j)) * cos(uvw.zc(m,k));
27
28   /* assign initial values to velocity in "j" direction */
29   Comp m = Comp::v();
30   for_vmijk(uvw,m,i,j,k)
31     uvw[m][i][j][k] = -cos(uvw.xc(m,i)) * sin(uvw.yc(m,j)) * cos(uvw.zc(m,k));
32
33   /* plot vector */
34   boil::plot->plot(uvw, "velocity", 0);
...
```

Only the lines which are different to program `06-01-main.cpp` are given. This program is stored as `06-03-main.cpp`. Velocity vector `uvw` is created from computational domain only in line 21, as an object of type `Vector`.

Vector field `uvw` stores three Cartesian velocity components: $u$, $v$ and $w$. The program actually initializes `uvw` to:

$$u = sin(x)cos(y)cos(z) \tag{6.2}$$

$$v = -cos(x)sin(y)cos(z) \tag{6.3}$$

$$w = 0 \tag{6.4}$$

Observe in lines 26 and 31 that `Vector` fields are accessed like the four-dimensional matrices in C++: `uvw[m][i][j][k];`, where `m` is a component counter (`Comp::u()` for $u$, `Comp::v()` for $v$ and `Comp::w()` for $w$), and `i`, `j` and `k` are positions in three-dimensional structured domain.

For shorter and safer browsing through vectors, a number of macros have been defined in `Vector/vector_browsing.h`, in a similar way it was done for scalar. Two examples are given in above listing, in lines 25 and 30. The macro:

```
30   for_vmijk(uvw,m,i,j,k)
```

takes as parameters the `Vector` field (`uvw`), the component for which you want to browse (`m`), and counters in $i$, $j$ and $k$ direction (`i`, `j` and `k`.

One might wonder at this point why do we have to send vector component (`m`) to macro for browsing, isn't the grid defined by `Vector`'s member functions `si()`, `ei()`, `sj()`, .... Well, it must not be forgotten that vectors are defined on a staggered grid, which means their resolution is different from `Scalar`'s, and, even more, it is different for each vector component.

Make a small test. Insert the following lines to program `06-03-main.cpp`:

```
33   /* resolution of "u" velocity component */
34   OPR(uvw.ni( Comp::u() ));
35   OPR(uvw.nj( Comp::u() ));
36   OPR(uvw.nk( Comp::u() ));
37   /* resolution of "v" velocity component */
38   OPR(uvw.ni( Comp::v() ));
39   OPR(uvw.nj( Comp::v() ));
40   OPR(uvw.nk( Comp::v() ));
41   /* resolution of "w" velocity component */
42   OPR(uvw.ni( Comp::w() ));
43   OPR(uvw.nj( Comp::w() ));
44   OPR(uvw.nk( Comp::w() ));
```

Re-compile and re-run to get:

```
FILE: main.cpp, LINE: 34, uvw.ni( Comp::u() ) = 67
FILE: main.cpp, LINE: 35, uvw.nj( Comp::u() ) = 66
FILE: main.cpp, LINE: 36, uvw.nk( Comp::u() ) = 66
FILE: main.cpp, LINE: 38, uvw.ni( Comp::v() ) = 66
FILE: main.cpp, LINE: 39, uvw.nj( Comp::v() ) = 67
FILE: main.cpp, LINE: 40, uvw.nk( Comp::v() ) = 66
FILE: main.cpp, LINE: 42, uvw.ni( Comp::w() ) = 66
FILE: main.cpp, LINE: 43, uvw.nj( Comp::w() ) = 66
FILE: main.cpp, LINE: 44, uvw.nk( Comp::w() ) = 67
```

Apparently, resolution is different for each vector component. The reason is easy to understand. As the grid in shifted[3] in $i$ direction, additional cells are created on the plane of faces at the beginning of the domain. The same holds for shifts in $j$ and $k$ direction, with increase in according directions. The increase of number of cells due to staggering is illustrated in Fig. 6.2. For the same reason, many `Vector`'s member functions related to geometrical quantities (such as `dxc()`, `dyc()`, `dzc()`, `dxw()`, `dxe()`, `dV()`, ...) also depend on `Vector`'s component.

Velocity field prescribed by program (`06-03-main.cpp`) is shown in Fig. 6.3.

[3] staggered

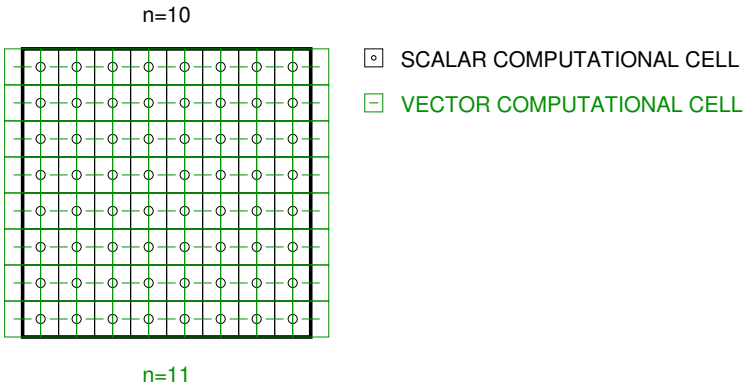



Figure 6.2: Number of cell rows for a `Vector` field is greater by one than than for `Scalar` field, in the direction of staggering.

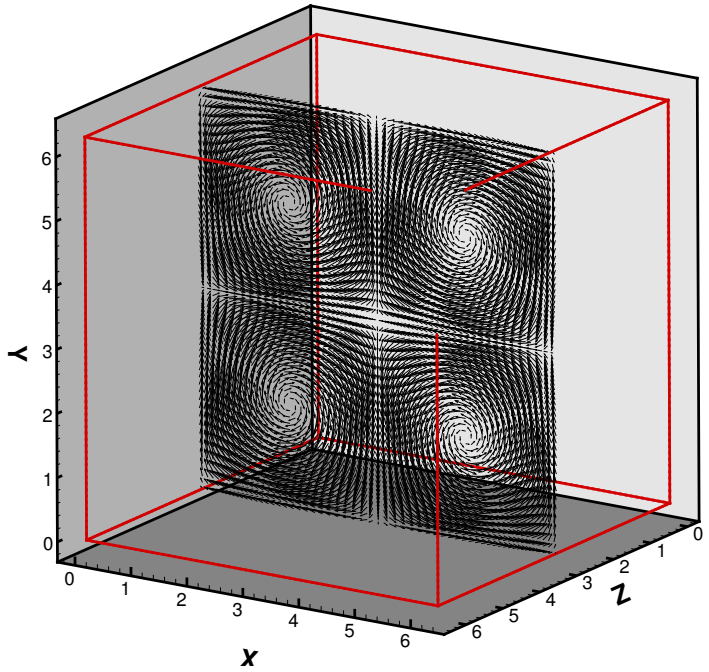



Figure 6.3: Pressure field prescribed by Eq. 6.4

Section 6.2 in a nutshell

- Vector fields are defined with PSI-Boil class `Vector`.

- `Vector` is defined for a `Domain`, with the constructor:

  - `Vector(Domain &);`

- `Vector`'s values are accessed with the usual `C++` syntax:

  - `p[m][i][j][k];`

where `m` is a vector component ranging from 0-3 (for $i$, $j$ and $k$).

- To browse through all the `Vector` values of component `m`, use the following macro:

  - `for_vmijk(Vector &, int m, int i, int j, int k);`

Similar macros are defined in directory: `PSI-Boil/Src/Vector/vector_browsing`.

- `Vector` has a number of member functions to access `Domain`s geometrical quantities. Most of them depend on the `Vector`'s component

# Chapter 7

# Transport Equation Solvers

Scalar and vector fields introduced in Sec. 5 can be used to hold transported variables, such as species, enthalpy and momentum. In PSI-Boil, these fields are embedded into objects of type `Scalar` and `Vector`, but these objects do not have the possibility to discretize and solve transport equations.

A set of objects which discretizes transport equations is defined and has a type `Equation`[1]. `Equation` class spawns two *children*, namely the `Centered` for transported scalar fields, and `Staggered` for transported vector fields. `Centered` class spawns more children, each specialized for a particular transport equation. `Staggered` class, on the other hand, has only one child, representing the equation for momentum conservation.

In this chapter, all transport equations implemented in PSI-Boil are summarized, and enthalpy conservation equation is solved for a simple problem of steady heat conduction. Although the equation solved is simple, it introduces quite a few new PSI-Boil features, such as linear system solvers, definition of materials, etc.

## 7.1 Transport equations in PSI-Boil

In this section, transport equations currently implemented in PSI-Boil are briefly outlined. They all written in *conservative* form because it is closer to physical laws we are trying to simulate and because the FV method employed by PSI-Boil discretizes transport equations in the same form. Furthermore, physical units for each equation are given, since it serves as a good check whether the implementation (discretization) of each member in the equation is valid.

### 7.1.1 Species conservation equation

Species conservation equations, defined with the class `Concentration`, reads:

$$\int_V \frac{\partial \rho\alpha}{\partial t} dV + \int_S \rho \mathbf{u} \alpha \, d\mathbf{S} = \int_S \gamma \nabla \alpha \, d\mathbf{S} + \dot{M} \quad [\frac{kg}{s}] \tag{7.1}$$

where $\rho$ $[\frac{kg}{m^3}]$ is the mass density, $\alpha$ [1] is the species concentration, $t$ $[s]$ is time, $\mathbf{u}$ $[\frac{m}{s}]$ is velocity field, $\gamma$ $[\frac{kg}{ms}]$, and $\dot{M}$ $[\frac{kg}{s}]$ is the species source. The equation is written in integral form, i.e. it is defined for a general volume $V$ $[m^3]$, enclosed by surface $S$ $[m^2]$.

[1] Since it represents a transport equation

Class `Concentration` is derived from `Centered`, which is a derivation of `Equation`. `Concentration` is defined in `PSI-Boil/Src/Equation/Centered/Concentration`.

### 7.1.2 Enthalpy conservation equation

Enthalpy conservation equations, defined with the class `Enthalpy`, reads:

$$\int_V \frac{\partial \rho C_p T}{\partial t} dV + \int_S \rho C_p \mathbf{u} T \, d\mathbf{S} = \int_S \lambda \nabla T \, d\mathbf{S} + \dot{Q} \quad [\frac{J}{s} = W] \tag{7.2}$$

where $C_p$ $[\frac{J}{kg\,K}]$ is thermal capacity, $T$ $[K]$ is the temperature, $\lambda$ $[\frac{W}{m\,K}]$ the heat conductivity and $\dot{Q}$ $[\frac{J}{s}]$ is the heat source.

`Enthalpy` is defined in `PSI-Boil/Src/Equation/Centered/Enthalpy` and is derived from `Centered` and `Equation`.

### 7.1.3 Phase indicator equation

Phase indicator function ($\Phi$), defined in the class `LevelSet`, is used in conjunction with conservative Level-Set (LS) method implemented in PSI-Boil. Essentially, it is purely hyperbolic conservation equation for $\Phi$:

$$\int_V \frac{\partial \Phi}{\partial t} dV + \int_S \mathbf{u} \Phi \, d\mathbf{S} = 0 \quad [\frac{m^3}{s}] \tag{7.3}$$

Class `LevelSet` is defined in `PSI-Boil/Src/Equation/Centered/LevelSet` and has the same ancestry as `Enthalpy` and `Concentration`. `LevelSet` may be a bit of a misnomer, so it might change it's name in the future.

### 7.1.4 Pressure-Poisson equation

Not really a transported quantity, but result of the time discretization of momentum conservation equations, pressure-Poisson equation is defined as:

$$\int_S \frac{\nabla p'}{\rho} \, d\mathbf{S} = \frac{1}{\Delta t} \int_S \mathbf{u} \, d\mathbf{S} \quad [\frac{m^3}{s^2}], \tag{7.4}$$

where $p$ $[\frac{kg}{m\,s^2} = Pa]$ is the pressure. As all the other cell-centered `Equations`, `Pressure` is derived from `Centered` and `Equation`. It's definition can be found in: `PSI-Boil/Src/Equation/Centered/Pressure`.

### 7.1.5 Momentum conservation equation

The only class belonging to `Staggered` branch of `Equations`, is the `Momentum`, which discretizes the momentum conservation equation:

$$\int_V \frac{\partial \rho \mathbf{u}}{\partial t} dV + \int_S \rho \mathbf{u}\mathbf{u} \, dS = \int_S \mu \nabla \mathbf{u} \, dS - \int_V \nabla p \, dV + \mathbf{F} \quad [\frac{kg\,m}{s^2} = N] \tag{7.5}$$

Here, $\mu$ $[\frac{kg}{m\,s}]$ is dynamic viscosity.

`Momentum` is the only class derived from `Staggered`, which is derived from `Equation`. Class `Momentum` is defined in: `PSI-Boil/Src/Equation/Staggered/Momentum`.

Section 7.1 in a nutshell

- Transport equations in PSI-Boil are defined as the class `Equation` serving as a parent to:
  - `Centered` for cell-centered variables, spawning to
    - `Concentration` - species conservation equation,
    - `Enthalpy` - enthalpy conservation equation,
    - `LevelSet` - phase indicator equation for LS method,
    - `Pressure` - pressure-Poisson equation,
  - and `Staggered` for face-centered (staggered) variables, used only as a parent to
    - `Momentum` - momentum conservation equation.

## 7.2 Simulation of heat conduction

Having briefly introduced transport `Equations` in PSI-Boil we move forward to apply one of them to solve a simple steady heat conduction problem, governed by the equation:

$$\int_S \lambda \nabla T \, d\mathbf{S} = 0 \quad [W] \tag{7.6}$$

It is clearly a special case of Eq. 7.2, without unsteady, convective and source terms.

Let's consider a cubical computational domain, having dimensions $1 \times 1 \times 1$, and let's impose the following boundary conditions:

- $T = 300\ [K]$ at $x = 0$
- $T = 400\ [K]$ at $x = 1$
- $\frac{\partial T}{\partial n} = 0$ everywhere else

The program which discretizes and solves this problem follows (`07-01-main.cpp`):

```
 1 #include "Include/psi-boil.h"
 2
 3 const real L =  1.0;
 4 const int  N = 64;
 5
 6 /******************************************************************************/
 7 main(int argc, char * argv[]) {
 8
 9   boil::timer.start();
10
11   Grid1D g( Range<real>(0,L), N, Periodic::no());
12   Domain d(g, g, g);
13
14   Scalar t(d), q(d);                          /* temperature and its source */
15   Vector uvw(d);                              /* velocity field */
```

```
16
17   t.bc().add( BndCnd( Dir::imin(), BndType::dirichlet(), 300.0 ) ); /* b.c. */
18   t.bc().add( BndCnd( Dir::imax(), BndType::dirichlet(), 400.0 ) ); /* b.c. */
19   t.bc().add( BndCnd( Dir::jmin(), BndType::neumann() ) );          /* b.c. */
20   t.bc().add( BndCnd( Dir::jmax(), BndType::neumann() ) );          /* b.c. */
21   t.bc().add( BndCnd( Dir::kmin(), BndType::neumann() ) );          /* b.c. */
22   t.bc().add( BndCnd( Dir::kmax(), BndType::neumann() ) );          /* b.c. */
23
24   Matter solid(d);                                  /* matter */
25
26   Krylov * solver = new CG(d, Prec::di());          /* linear solver */
27
28   Times time;                                       /* simulation time */
29
30   Enthalpy enth(t, q, uvw, time, solver, &solid);   /* enthalpy conservation
31                                                        equation              */
32   enth.diffusion_set(TimeScheme::backward_euler()); /* time stepping scheme */
33
34   AC multigrid( &enth );                            /* AMG solver for enth. */
35
36   t = 350.0;                                        /* initial "guess" */
37
38   multigrid.vcycle(ResRat(1e-4));                   /* solve linear system */
39
40   boil::plot = new PlotTEC();
41   boil::plot->plot(t, "t");
42
43   boil::timer.stop();
44   boil::timer.report();
45 }
```

Although the problem being solved is simple, the program `07-01-main.cpp` introduces many new concepts and each of them is presented in a separate subsection.

### 7.2.1 Boundary conditions

Boundary conditions are defined for fields, i.e. for `Scalar`s and for `Vector`s. A class `BndCnd`, defined in `Src/Boundary` holds the boundary conditions for various fields. `BndCnd` can be defined in several ways (you can find them all in `Src/Boundary/bndcnd.h`, but essentially, it is defined by *direction*, boundary condition *type* and *value*.

#### 7.2.1.1 Direction

First parameter to define a boundary condition is *direction*. Direction can denote any of the logical six boundaries of computational `Domain`, meaning: $i_{min}$ and $i_{max}$ (sometimes also referred to as *west* and *east*), $j_{min}$ and $j_{max}$ (or *south* and *north*), and $k_{min}$ and $k_{max}$ (or *bottom* and *top*). These directions are represented with ravioli object `Dir`, which can assume one of the following values[2]:

- `Dir::imin();` - west,
- `Dir::imax();` - east,
- `Dir::jmin();` - south,

[2] Actually, `Dir` is used as if it was a constant defining direction. But a constant which is easy to understand and for which compiler performs checking, when passed as an argument

- `Dir::jmax();` - north,
- `Dir::kmin();` - bottom and
- `Dir::kmax();` - north boundary of the `Domain`.

`Dir` is a ravioli object and is defined in `Src/Ravioli`.

#### 7.2.1.2 Boundary condition type

Boundary condition type is stored in the ravioli class `BndType` which, since used only with `BndCnd` is not defined in `Src/Ravioli` directory, but together with `BndCnd`, in `Src/Boundary`. `BndType`, can assume one of the following values:

- `BndType::undefined();`
- `BndType::dirichlet();`
- `BndType::neumann();`
- `BndType::periodic();`
- `BndType::inlet();`
- `BndType::outlet();`
- `BndType::wall();`
- `BndType::symmetry();`

which are self-explanatory. (**Check later if all `BndTypes` are applicable to all `Equations`**)

#### 7.2.1.3 Boundary condition value

This argument is optional. Namely, for some boundary conditions, such as symmetry and periodic, it does not have to be specified. If value is not for other `BndTypes`, it is assumed to be equal to zero. Value can be specified as a `real` number, but also analytically, as a function of $x$, $y$ and $z$ coordinates. That will be covered in later sections.

#### 7.2.1.4 Final form of boundary condition definition

For example, to define a *Dirichlet* boundary condition at the *west* boundary of the `Domain` (at $i = i_{min}$), with the value of 300, we use the line:

```
BndCnd( Dir::imin(), BndType::dirichlet(), 300.0 );
```

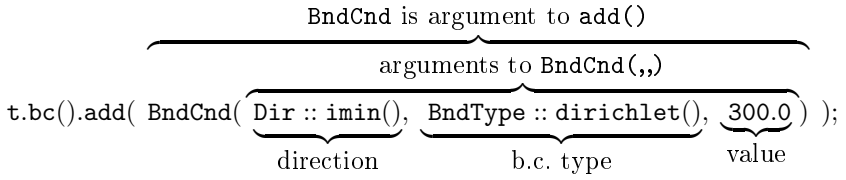
To define a *Neumann* condition at the *north* boundary ($j = j_{max}$), we do not have to specify the value, so the syntax is:

```
BndCnd( Dir::jmax(), BndType::neumann() );
```

These definitions are pointless if not assigned to fields representing transported quantities. To achieve that, we use the member function:

```
Scalar::bc().add( BndCnd & );
```

Program lines 17–22 assign various boundary conditions to `Scalar t`. The lines look cramped a bit, but that is because boundary condition object (`BndCnd`) is defined at the place it is also passed as argument to `Scalar::bc().add()`. To illustrate it further, program line 17, with all the parts clearly denoted is shown below:




### 7.2.2 Matter

There is a new type of object defined in line 24, called `Matter`. For every `Equation` solved by PSI-Boil, `Matter` must be defined, bringing with it a suite of physical properties featured in governing equations presented in Sec. 7.1. For this example, governed by the equation for steady conduction (Eq. 7.6 no property needs to be adjusted. Furthermore, `Matter` can be defined as a *mixture*, but that will not be covered in this section. Therefore, we simply define an object of type `Matter` now.

### 7.2.3 Krylov

Next novelty is the object of type `Krylov`, which is a linear solver used to solve systems of discretized equations, created in line 26. Currently, there are three solvers from Krylov sub-space family defined: Conjugate Gradient (implemented as `CG`), Bi-Conjugate Gradient (as `BiCG`) and Conjugate Gradient Squared (implemented as class `CGS`). `Krylov` is a parent class to these three solvers and as such, it can *point* to each of them[3]. A particular solver can be selected from one of the following constructors:

- `Krylov * solver = new CG (d, Prec::di());` for CG solver
- `Krylov * solver = new BiCG(d, Prec::di());` for BiCG solver
- `Krylov * solver = new CGS (d, Prec::di());` for CGS solver

Constructor to each of the solvers takes a `Domain` as first argument and type of *preconditioner* as a second. Type of preconditioner is defined by a ravioli object `Preconditioner` and it can assume one of the following values[4]:

- `Preconditioner::di()` for diagonal,
- `Preconditioner::ic()` for Incomplete Cholesky, and
- `Preconditioner::no()` for no preconditioning.

### 7.2.4 Times

Object of type `Times` is defined in line 28. It defines the simulation time, i.e. number of time steps which have to be performed, and a value for the time step. In present case, which is steady[5], nothing needs to be specified. `Times` is a ravioli type object, defined in directory `Src/Ravioli`. It is also useful in time-integration loops for unsteady problems, as will be shown below.

[3] Remember that pointer to a parent class is also a pointer to all of it's children.
[4] This ravioli object is not defined in directory `Src/Ravioli`, because it is used only with `Krylov` type solvers.
[5] Steady cases are an exception for kind of problems PSI-Boil aims at

### 7.2.5 Transport equation

Once we have field for transported quantity (here the temperature, `Scalar t`), field for it's source (here `Scalar q`), convective velocity field (`Vector uvw`), simulation time (`Times time`) and a linear solver (`Krylov * solver`), we can define a transport equation with the constructor:

```
30   Enthalpy enth(t, q, uvw, time, solver, &solid);
```

Convective velocity field, source and time, *must* be provided, no matter if problem involves transport by velocity, has any external sources, or is it unsteady. It may look as an overhead, but keep in mind that *vast* majority of problems analyzed with PSI-Boil are *unsteady*, involve fluid flow and feature external *sources* on transported variables. Actually, it would be an overhead to write additional constructors for these special cases which occur very rarely. The form of constructor shown above is *the same* for all transport `Equation`s implemented in PSI-Boil. Constructor for `Equation`, such as the one in line 30, also discretizes the governing transport equation, for this case, it is the enthalpy conservation defined with Eq. 7.2.

### 7.2.6 Time-stepping scheme

For each transport equation, time stepping scheme can be set for *diffusive* and *convective* terms. If nothing is specified, PSI-Boil uses default schemes, which are:

- *Adams-Bashforth* for convection, and
- *Crank-Nicolson* for diffusive terms.

For steady cases[6], time stepping scheme should be set to *steady*. In this case, there is no convection, so setting the time-stepping scheme for convection is pointless. For diffusion, it is set with the line:

```
32   enth.diffusion_set(TimeScheme::steady());
```

This setting uses `Equation`'s member function `diffusion_set` and ravioli object `TimeScheme`, which can have one of the following values:

- `TimeScheme::backward_euler();`
- `TimeScheme::crank_nicolson();`
- `TimeScheme::adams_bashforth();`

### 7.2.7 Additive correction multigrid solver

This case, since steady, results in relatively poorly conditioned linear system of discretized equations. Therefore, it is sound to use a multigrid procedure to accelerate it's convergence. PSI-Boil has only one multigrid algorithm, based on Additive Correction (AC) scheme (**add reference**). It is defined in class `AC`. Algebraic multigrid is based on coarsening discretized system of equations and solving these coarser (smaller) systems faster, leading to faster reduction of residuals on finer (bigger) levels as well. `AC` is defined in line 34:

```
34   AC multigrid( &enth );
```

[6] Again: which is an exception

It takes the reference to `Equation` as a parameter. Remember that `Equation` was already defined with a linear solver (`solver`), which will be used by `AC` to *smooth* each refinement level.

The solution of the liner system by `AC` is invoked by:

```
38   multigrid.vcycle(ResRat(1e-4));
```

which starts a "V" multigrid cycle and runs until residuals fall four levels of magnitude.

This program creates the following output:

```
Domain level 4 created !
Domain level 3 created !
Domain level 2 created !
Domain level 1 created !
FILE: centered_coarsen.cpp, LINE: 6, coarsenig
Centered level 32 x 32 x 32 created !
Centered level 16 x 16 x 16 created !
Centered level 8 x 8 x 8 created !
Centered level 4 x 4 x 4 created !
Number of cycling levels: 5
FILE: centered_update_rhs.cpp, LINE: 14, conv_ts.N() = 0
Initial  res = 2.95794
Cycle 1; res = 0.062026
Cycle 2; res = 0.00803034
Cycle 3; res = 0.000562558
Cycle 4; res = 0.000247838
Converged after 4 cycles!
# Plotting: t_p000.dat
+========================================
| Total execution time: 6.21 [s]
+----------------------------------------
| Time spent in enthalpy discretize: 0.03 [s]    (0.462963%)
| Time spent in vcycle             : 5.12 [s]    (79.0123%)
| Time spent in coarsening         : 0.04 [s]    (0.617284%)
| Time spent in plotting           : 1.21 [s]    (18.6728%)
| Time spent elsewhere: 0.08 [s]    (1.23457%)
+----------------------------------------
```

The messages in the first five lines should be familiar. They are printed during the `Domain` definition phase. Lines beginning with `"Centered level ..."` are messages from the `AC` multigrid solver, printed while it creates coarser levels. For this case, number of cycling levels, including the finest one (defined by the user) is five. Lines beginning with `"Cycle ..."` print the residuals from each "V" cycle performed in multigrid solver. It is generally an *excellent* sign if residuals fall by an order of magnitude in each cycle. More often they fall by a factor of five in each cycle. The output finishes with CPU-time statistics. In `main.cpp` we have defined only global timers (lines 9, 43 and 44), but certain PSI-Boil subroutines have built-in local timers. The output we got for this program is typical. Very little time was spent in discretization of governing equations (0.49 %), approximatelly the same in `Domain` coarsening (0.61 %), but vaste majority of CPU time is spend in V-cycle (79 %). For this simulation, particularly because it is steady, a lot of time is spent in plotting as well (18.7 %). Only (1.23 %) is spent in the parts of the code which are not measured, meaning that time monitors are properly placed.

This data provides useful profiling information. It is clear that any *tuning* of the code in the parts of the code which discretize governing equations, will not bring the overall CPU-time down considerably.

### 7.2.8 Final solution

The final solution obtained by the program (stored in file: `t_p000.dat` is given in Fig. 7.1. Clearly, the solution is linear, ranging from $T = 300$ $[K]$ to $T = 400$ $[K]$.

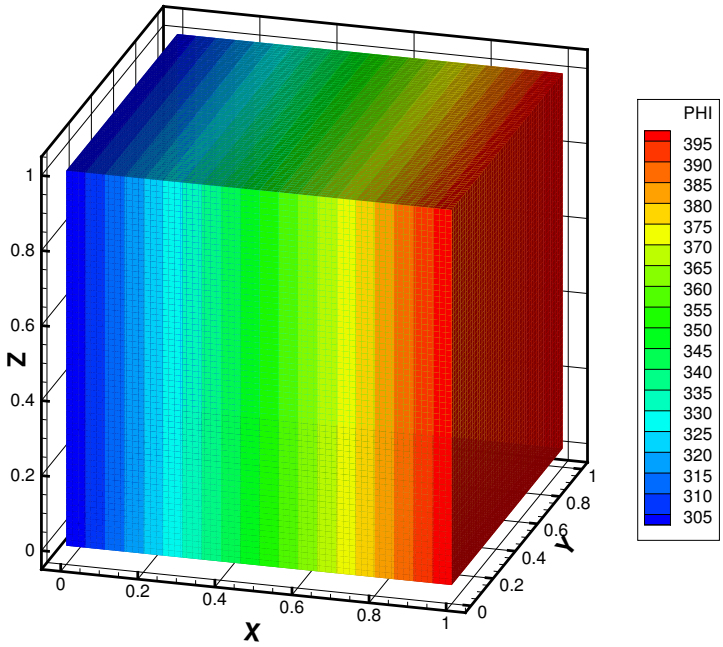



Figure 7.1: Temperature field

### 7.2.9 Exercise

Note that `AC` multigrid solver is optional. Try to modify the program lines 34–38 to:

```
34// AC multigrid( &enth );                          /* AMG solver for enth. */
35
36   t = 350.0;                                      /* initial "guess" */
37
38   enth.solve(ResRat(0.001), "enthalpy");
```

With this modification, you do not create `AC` multigrid solver (line 34 is a comment) and you directly call Krylov solver to *solve* the discretized equations for you. Compile and run this program. Visualize the results and compare with the ones shown in Fig. 7.1. What do you see? Can you

comment on that?

Section 7.2 in a nutshell

- To solve a transport equation in PSI-Boil, the following objects have to be used (in addition to `boil::timer`, `Grid1D` and `Domain` introduced before):
  - `Matter` - to define the physical properties of the matter (substance) under consideration,
  - `Krylov` - a member of the Krylov's subspace family of solvers for solving the discretized equations,
  - `Times` - object representing simulation time,
  - Field variables represented by `Scalar` and `Vector` objects,
  - `BndType`, `BndCnd` and `Dir` to define boundary conditions,
  - Governing `Equation` - such as enthalpy, species, momentum conservation,
  - (optional) Setting the time stepping scheme with the `TimeScheme` object.
  - (optional) `AC` multigrid solver.

## 7.3 Heat conduction with source term

In this section, we move a step forward, and solve a steady heat conduction problem, but with the addition of source term. This problem is governed by the equation:

$$\int_S \lambda \nabla T \, d\mathbf{S} = \int_V \dot{q} \, dV \quad [W] \tag{7.7}$$

Let's consider the same computational domain as in Sec. 7.2 with following boundary conditions:

- $T = 300\ [K]$ at $x = 0$ and $x = 1$,
- $\frac{\partial T}{\partial n} = 0$ everywhere else.

and let's set the internal heat source to $\dot{q} = 100.0\ [\frac{W}{m^3}]$. It is straightforward to work out the analytical solution to this problem:

$$T(x) = \frac{\dot{q}}{2\lambda}(x^2 - x) + 300 \quad [K] \tag{7.8}$$

It is a parabolic temperature profile, reaching the maximum of $T = 312.5\ [K]$ at $x = 0.5$.

The program which discretizes and solves this problem (`07-03-main.cpp`) is created by slight modifications of the pure heat conduction program (`07-01-main.cpp`) and here we outline only the lines which differ between these two programs:

```
...
18   t.bc().add( BndCnd( Dir::imax(), BndType::dirichlet(), 300.0 ) ); /* b.c. */
...
37   t = 350.0;                                          /* initial "guess" */
38
39   for_vijk(q,i,j,k)                                   /* fill the source term */
```

```
40      q[i][j][k] = 100.0 * q.dV(i,j,k);
41
42    multigrid.vcycle(ResRat(1e-4));                    /* solve linear system */
...
```

The change in line 18 is apparent: the boundary temperature at $i = i_{max}$ to 300 $[K]$ Source term for Eq. 7.7 is defined in lines 38 and 39. The macro `for_vijk(q,i,j,k)` in line 38, should be familiar - it browses through all internal cells of `Scalar q`, but line 39 deserves special attention. Since PSI-Boil discretizes equations in their *integral* form (see Sec. 7.1), the source term (i.e. the right hand side) of the discretized equation must also be in integral form. That is why we multiply prescribed value of internal heat source (100 $[\frac{W}{m^3}]$) with cell volume (`q.dV(i,j,k)`) in each cell. We actually apply mid-point integration rule to have the source term in right dimensions, which is, for case of enthalpy equation a $[W]$.

The solution obtained by the program `07-03-main.cpp` is given in Fig. 7.2. The temperature profile is parabolic in $x$ direction, and reaches maximum of $T = 312$ $[K]$ at $x = 0.5$, as predicted by analytical solution in Eq. 7.8.

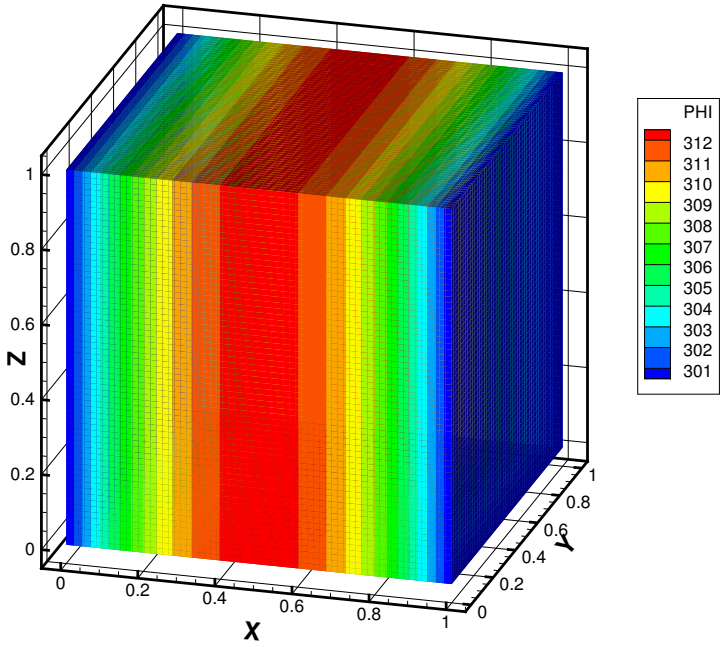



Figure 7.2: Temperature field solution for steady conduction with heat source.

### 7.3.1 Changing the physical properties

The solution of steady heat conduction without source terms, did not depend on physical properties of the material under considerations. But, for steady heat transfer with source term (Eq. 7.7) solution depends on thermal conductivity $\lambda$. We did not specify anything for it in the program `07-03-main.cpp`, meaning that PSI-Boil used the default values, which is 1. To set it to different value, say $\lambda = 2.0$, the following command should be used:

```
solid.lambda(2.0);
```

after `Matter` has been defined and before the definition of the governing `Equation`. Modify the program `07-03-main.cpp` to define $\lambda = 2.0$. Re-compile and re-run. Numerical solution should reach maximum $T = 325$ $[K]$.

Other `Mater`'s member function, which can be used to change physical properties are:

- `Matter::rho (real value);` - density $\rho$ $[\frac{kg}{m^3}]$
- `Matter::mu (real value);` - dynamic viscosity $\mu$ $[\frac{kg}{m\,s}]$
- `Matter::cp (real value);` - thermal capacity $C_p$ $[\frac{J}{kg\,K}]$
- `Matter::lambda(real value);` - thermal conductivity $\lambda$ $[\frac{W}{m\,K}]$
- `Matter::gamma (real value);` - specified diffusivity $\gamma$ $[\frac{kg}{ms}]$
- `Matter::sigma (real value);` - surface tension $\sigma$ $[\frac{N}{m} = \frac{kg}{s^2}]$

Section 7.3 in a nutshell

- When specifying a source term, make sure it has right dimensional units.

- Keep in mind that PSI-Boil solves transport equations in *integral* form.

- It is almost sure that you will have to integrate the prescribed source term with the mid-point rule (multiplying it by cell volume)

- Use `for_vijk(Scalar,int,int,int)` macro to browse through source field and member function `Scalar::dV(i,j,k)` to access cell volume.

- Physical properties can be set by `Matter`'s member functions.

# Chapter 8

# Structure of a Typical PSI-Boil Program

In Sec. 7.2 and 7.3 PSI-Boil programs were used to solve transport equations for steady heat conduction. Although these problems are simple, the programs which were developed (`07-01-main.cpp` and `07-03-main.cpp`) featured a general structure PSI-Boil programs assumes, if used for simulating general transport phenomena.

Since this structure is typical also for simulating much more complex phenomena, it is outlined here as a reference point for programs developed in later chapters. Figure 8.1 shows the typical structure of a PSI-Boil program. The program flows from top to bottom and is divided into six logical units (numbers 1-6 on the left of Fig. 8.1) in addition to header (H) and footer (F). The structures on the left side of Fig. 8.1 illustrates the actions performed, and most of them are connected by a straight line with PSI-Boil's classes (or global objects) which perform or define them. There are *dependencies* between the classes. For example, `Domain` created from three `Grid1D`'s (Sec. 5.3), meaning that `Domain` depends on `Grid1D`. These dependencies govern the order in which PSI-Boil's objects are created. There is some flexibility in the order objects are used in the general structure. But, thick dashed lines should not be crossed, except for objects with dashed frame. For example, simulation time (`Times`) is now in unit 3, but since it does not depend on any other object, it could have been defined right after the header. That is illustrated with the small dashed arrow on the left, and with the small letter H next to it. In the same way, variable initialization, now performed in unit 5, could have been done right after the variable definition, as denoted by the small arrow and number 2. On the other hand, multigrid solver `AC` from unit 5, can *not* be defined before the governing equations in unit 4 and therefore can *not* cross the dashed line.

Header (H) starts global object `boil::timer`, while footer (F) stops it and prints the information on the CPU time on the terminal. Header and footer must be present in every PSI-Boil program, because many object use local timers to measure the performance of their member functions.

Quite naturally, right after the header, first unit starts which defines geometry (enclosed in an orange frame on Fig.8.1). Definition of geometry means defining grids (`Grid1D`), IB's (`Body`) and domains (`Domain`).

Once the computational domain is defined, we can proceed with definition of scalar and vector fields (`Scalar` and `Vector`) featured in transport equations being solved. As soon as their are defined, boundary conditions should be imposed on them as well.

Third unit is a bit heterogeneous. It defines physical properties (`Matter`) which depends on `Scalar`

in case of multiphase flows[1]. Simulation time is also placed in this unit, although it could have been defined right after the header. We prefer to put it here for convenience, just to be closer to governing equations unit which uses it as a parameter. Finally, solvers `Krylov` are quite rigidly in this unit. They must be defined after field variables and before governing equations.

Unit number four is dedicated to definition of governing equations. All the equations are defined here (usually, that means: `Momentum`, `Pressure`, `Enthalpy` ...) as well as particular details about spatial and temporal discretization. As far as spatial discretization is concerned, various convection schemes can be defined (see the global class `Limiter` and associated ravioli class `ConvScheme` in the directory: `Src/Global`). This feature is yet to be explored in the chapters below. Temporal discretization schemes have been briefly touched upon in Sec. 7.2.6.

The following unit, number five, is actually a preparation for the time-loop. As first, a multigrid solver (`AC`) has to defined and it depends on the governing equations. `AC` is almost invariantly defined for pressure, while the other variables, particularly when discretized for unsteady simulation, have well-conditioned linear systems and `Krylov` solvers are usually efficient enough. Variables and source terms should be initialized before the time loop, as well as plotting format. Both of these last two object could have been defined much sooner. Variables could have been initialized right after their definition, while the plotting format could have been chosen right after the header (indicated with small arrows).

Undoubtedly, the most important unit of the PSI-Boil is unit six, the *Time loop*. It can be as simple as a single call to linear solver[2] or as complex as time loop embedding another iterative procedure to couple various equations. This unit will vary the most between different programs and different classes of problems in particular. Therefore, it is beyond the scope of this chapter to explain it any further.

General structure of a PSI-Boil program, resembles two things: an *input file* for a CFD package defining the problem to be solved (defining geometry, boundary conditions, material properties, choosing plotting format, ...), but also a flow diagram of a CFD program. In essence, it serves as both. As stated in Sec. 1.1, each problem solved with PSI-Boil requires a separate main program (stored in `main.cpp`), which used PSI-Boil's objects to solve a particular program. Each program defines geometry, boundary condition, physical properties (units 1-4 in Fig. 8.1), typical for an input file, but also defines the algorithm (units 5 and 6 in Fig. 8.1), as the flow diagram does.

Chapter 8 in a nutshell

- Although PSI-Boil should be regarded as a collection of objects which facilitates the building of programs for numerical analysis of transport phenomena, each such program has a typical structure, represented in this chapter in form of a diagram.

- The typical structure has six distinct units, in addition to header and footer.

- This structure is flexible to some extent, but dependencies between the units introduce a certain order in which PSI-Boil objects are laid out.

- PSI-Boil's main program (the one which is different for each problem solved) can be regarded as an input file for simulation and a flow diagram of the applied algorithm.

[1] That will be covered in later chapters.
[2] For example, line 38 in `07-01-main.cpp` is a complete *Time loop*.

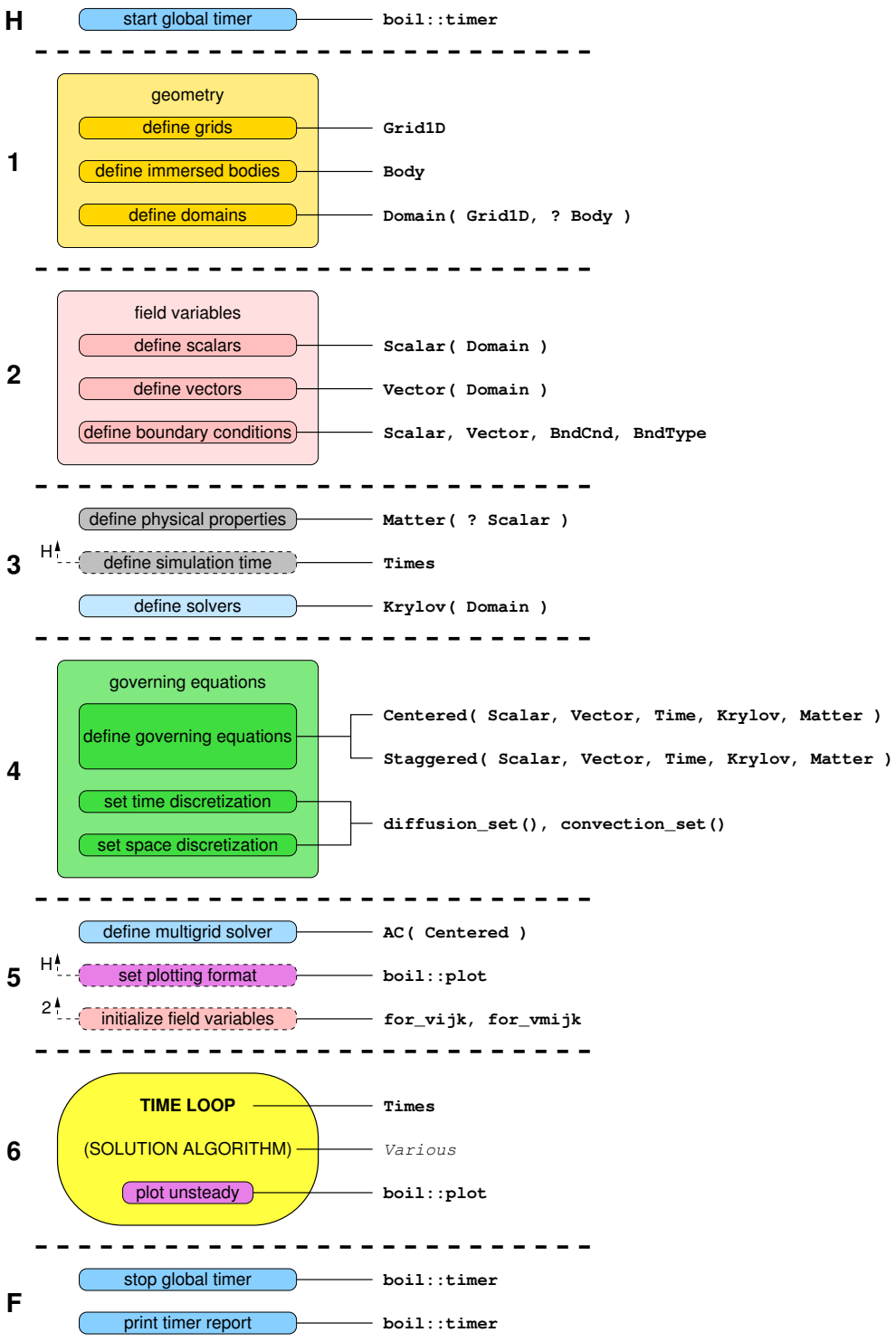



Figure 8.1: General structure of a PSI-Boil program.

# Chapter 9

# Single-phase Flows

This chapter deals with the solution of transport equation for incompressible fluid flow. The equation which describes this phenomenon is momentum conservation equation, given by 7.5, repeated here for convenience:

$$\int_V \frac{\partial \rho \mathbf{u}}{\partial t} dV + \int_S \rho \mathbf{u}\mathbf{u}\, dS = \int_S \mu \nabla \mathbf{u}\, dS - \int_V \nabla p\, dV + \mathbf{F} \quad [N] \tag{9.1}$$

This form, however, poses a problem. We have one three equations (one for each velocity component), but for unknowns: velocity components and pressure.

For incompressible flow problems, pressure has always been sought through additional constraint, mass conservation equation:

$$\int_V \frac{\partial \rho}{\partial t} dV = -\int_S \rho \mathbf{u} d\mathbf{S} \quad [\frac{kg}{s}] \tag{9.2}$$

Momentum and mass conservation have been linked in various ways, providing different algorithms for computation of incompressible flows. The most widely used in applied CFD are SIMPLE, SIMPLEC, SIMPLER and even PISO (**references**). For highly unsteady flow, at simulation of which PSI-Boil aims, the most efficient is the *fractional step method*, sometimes referred to as the *projection* method. It combines momentum conservation equation (7.5) with mass conservation equation (9.2), at intermediate time between two simulated time steps, to give additional equation for pressure (already introduced as 7.4):

$$\int_S \frac{\nabla p}{\rho}\, d\mathbf{S} = \frac{1}{\Delta t} \int_S \mathbf{u}^\star\, d\mathbf{S} \quad [\frac{m^3}{s^2}], \tag{9.3}$$

often referred to as pressure-Poisson equation.

The fractional step method, which is used by PSI-Boil, is now outlined. The momentum conservation equation (9.1) must first be discretized in time, part by part. The most usual approach is to use Crank-Nicolson scheme for diffusion (to avoid very low time steps needed by the forward Euler scheme) and to use Adams-Bashforth for convection, for the sake of simplicity. Inertial terms are discretized with backward Euler scheme. The time-discretized momentum equation then looks like:

$$\mathbf{I}^\star - \mathbf{I}^{n-1} + \frac{3}{2}\mathbf{C}^{n-1} - \frac{1}{2}\mathbf{C}^{n-2} = \frac{1}{2}\mathbf{D}^\star + \frac{1}{2}\mathbf{D}^{n-1} + \mathbf{F}^{n-1} \tag{9.4}$$

Here, $\star$ denotes the tentative (intermediate) time step, while $\mathbf{I}^\star = \frac{1}{\Delta t}\int_V \rho \mathbf{u}^\star dV$ and $\mathbf{D}^\star = \int_S \mu \nabla \mathbf{u}^\star\, d\mathbf{S}$ denote inertial and diffusive term in the tentative time step. Old (known) values of velocity (at time step $n-1$) define terms: $\mathbf{I}^{n-1} = \frac{1}{\Delta t}\int_V \rho \mathbf{u}^{n-1} dV$, and $\mathbf{D}^{n-1} = \int_S \mu \nabla \mathbf{u}^{n-1}\, d\mathbf{S}$, as well as convective term at old time step: $\mathbf{C}^{n-1} = \int_S \rho \mathbf{u}^{n-1}\mathbf{u}^{n-1}\, d\mathbf{S}$. Clearly, $\mathbf{C}^{n-2} = \int_S \rho \mathbf{u}^{n-2}\mathbf{u}^{n-2}\, d\mathbf{S}$.

Equation 9.4 is discretized in space giving a linear system of equations for intermediate velocity ($\mathbf{u}^\star$). Once this system is solved gives velocity field which does not, generally, satisfies mass conservation equation. It is used to compute pressure from 9.3. The computed pressure is used to *project* velocity into a divergence free velocity field:

$$\mathbf{u}^n = \mathbf{u}^\star - \frac{1}{\rho}\nabla p\,\Delta t \tag{9.5}$$

which represents velocity at new time step.

## 9.1 Flow over a cylinder

In this section, a flow over a cylinder is solved with PSI-Boil. The flow is two-dimensional and isothermal and occurs at relatively low Reynold number ($Re$). This problem will tech you how to define boundary conditions for some parts of the boundary and is also a first fluid flow solver in this tutorial.

The geometry and boundary conditions are shown in Fig. 9.1, and dimensions are specified as following: $L = 2.2$, $H = 0.41$, and $W = H/2$. Cylinder diameter is set to $D = 0.1$. Distance of cylinder axis to lower corner of the inlet is $h = 0.2$, meaning the cylinder is not placed symmetrically in the problem domain. It is intentionally so, to promote earlier transition to a vortex shedding regime.

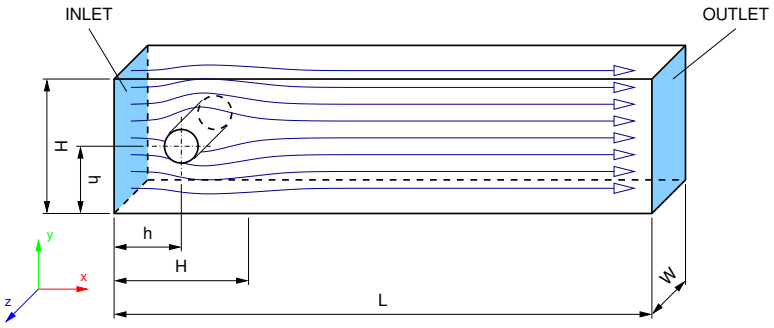



Figure 9.1: Geometry, boundary conditions and basic topology for the flow over a cylinder.

Parabolic velocity profile is specified at the inlet according to the equation:

$$u = 4 \cdot U_m \cdot y \cdot (H - y)/H^2 \tag{9.6}$$

where $U_m$ is maximum inlet velocity equal to 1.5.

The program which solves this problem is stored under name `09-01-main.cpp` and will be outlined in the rest of this chapter. However, different program units, as outlined in Ch. 8, will be introduced in separate sub-sections. For the sake of shortness, some program comments will be skipped.

Domain dimensions, as well as resolutions, are defined as constants outside the body of the main function as:

```
3 /* dimensions */
4 const real H  = 0.41;
5 const real L  = 2.2;
6
7 /* resolutions */
8 const int NY =  64;
```

```
 9 const int NX1 = NY;
10 const int NX2 = NY;
11 const int NZ  = NY/8;
```

The grid will be uniform in $y$ and $z$ (normal and span-wise), but not in $x$ (stream-wise) direction. That is why we introduce two variables for resolution in $x$, namely `NX1` and `NX2`.

### 9.1.1 Domain

IB, in this case cylinder, is defined in line 23:

```
23   Body cyl("09-01-cylinder.stl");
```

Clearly, cylinder is defined in STL format in file `09-01-cylinder.stl`, which must be present in the running directory.

Grids are defined with the following lines:

```
28   Grid1D gz ( Range<real>(-H/4, H/4), NZ,  Periodic::yes());
29   Grid1D gy ( Range<real>( 0.0, H  ), NY,  Periodic::no());
30   Grid1D gx1( Range<real>( 0.0, H  ), NX1, Periodic::no());
31   const real dx = H/(real)NY;
32   Grid1D gx2( Range<real>( H,   L  ), Range<real>(dx, 8.0*dx), NX2, Periodic::no());
33   Grid1D gx( gx1, gx2, Periodic::no());
```

No stretching is used in normal and span-wise ($y$ and $z$) direction. Span-wise direction is defined to be periodic. Grid in stream-wise ($x$) direction is created from two parts: a *uniform* part, spanning from $x = 0$ to $x = H$, created in line 30 as `gx1` and a *non-uniform* one, spanning from $x = H$ to $x = L$, created in line 32 as `gx2`. The size of first cell in `gx2` is set to be `dx`, which corresponds to cell size in `gx1`. Final grid in $x$ direction is created in line 33 as a union of `gx1` and gx2.

Once the IB and grids are defined, domain is created in line 38 with:

```
38   Domain d(gx, gy, gz, &cyl);
```

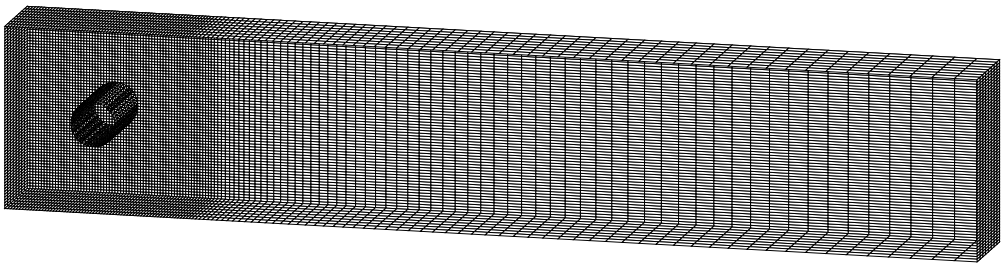


Figure 9.2: Computational grid for the flow around a cylinder.

### 9.1.2 Field variables

Now we have reached the second logical unit of the program, which defines field variables. For this case, we have velocity and momentum forces as `Vector`'s and pressure and it's source as `Scalars`, defined in lines 40 and 41:

```
43   Vector uvw(d), xyz(d); /* velocity and its source */
44   Scalar p  (d), f  (d); /* pressure and its source */
```

This is followed by the definition of boundary conditions. Since `Vectors` have three components, boundary conditions have to be assigned through a loop browsing through them, as it is done in lines 49–57:

```
49   for_m(m) {
50     uvw.bc(m).add( BndCnd( Dir::imin(), BndType::inlet(),
51                    "4.0*1.5*y*(0.41-y)/0.41^2", "0.0", "0.0") );
52     uvw.bc(m).add( BndCnd( Dir::imax(), BndType::outlet() ) );
53     uvw.bc(m).add( BndCnd( Dir::jmin(), BndType::wall() ) );
54     uvw.bc(m).add( BndCnd( Dir::jmax(), BndType::wall() ) );
55     uvw.bc(m).add( BndCnd( Dir::kmin(), BndType::periodic() ) );
56     uvw.bc(m).add( BndCnd( Dir::kmax(), BndType::periodic() ) );
57   }
```

Browsing is performed with macro `for_m`, defined in `Field/Vector/vector_browsing.h`. Variable representing vector component is defined as `m` throughout PSI-Boil. It is a ravioli object of type `Comp`. Member function for defining boundary conditions for `Vectors` defined from the one for scalars (introduced in Sec. 7.2.1) because it also has to provide the component for which this boundary condition applies, i.e. the syntax of the member function is:

```
Vector::bc(Comp m).add( BndCnd & );
```

In lines 50–51 inlet is defined (`BndType::inlet()`) at inlet plane (`Dir::imin()`). For velocity inlet, three values must be specified, one for each component. They are defined in line 55 as: `"4.0*1.5*y*(0.41-y)/0.41^2"`, `"0.0"` and `"0.0"`. PSI-Boil supports analytically prescribed boundary conditions, as long they are functions of $x$, $y$ and $z$. For velocity outlet, as well as for wall and periodic conditions, no values need to be specified, as shown in lines 52–56.

Boundary conditions for the pressure, when solved with the projection method, should be $\frac{\partial p}{\partial n} = 0$ on all boundaries where velocities are known. In this case, this are all, except the ones at periodic ones. All boundary conditions for pressure are defined in lines 62–67:

```
59   p.bc().add( BndCnd( Dir::imin(), BndType::neumann() ) );
60   p.bc().add( BndCnd( Dir::imax(), BndType::neumann() ) );
61   p.bc().add( BndCnd( Dir::jmin(), BndType::neumann() ) );
62   p.bc().add( BndCnd( Dir::jmax(), BndType::neumann() ) );
63   p.bc().add( BndCnd( Dir::kmin(), BndType::periodic() ) );
64   p.bc().add( BndCnd( Dir::kmax(), BndType::periodic() ) );
```

Note that boundary conditions have been defined for both variables at all boundary planes. No patch remained undefined. That is a *rule*. PSI-Boil *does not* support *default* boundary conditions for any variable.

### 9.1.3 Physical properties, time and solver

We have reached the third unit (according to standard PSI-Boil program layout outlined in Fig. 8.1, which defines physical properties (`Matter`), simulation time (`Times`) and linear solver (`Krylov`). In this program, the third unit looks like:

```
69   Matter fluid(d);
70
71   fluid.mu(0.001);
72
73   Times time(10000, dx/8.0);
74
75   Krylov * solver = new CG(d, Prec::di());
```

New substance is created with the name `fluid` in line 69, and its dynamic viscosity has been set to 0.001 in line 71. Simulation time is represented with object `time`, created with number of time steps equal to 10000 and time step equal to $\Delta t = dx/8$ The value of time step is calculated from the requirement that Courant-Friedrich-Levy (CFL) number is around $\frac{1}{4}$ during the simulation[1]. Time variable may be defined in other ways. Instead of number of time steps and the time step value, we could have defined total simulation time and number of time steps. The details of these, and other possible definitions, take a look at `Src/Ravioli/times.h`.

### 9.1.4 Transport equations

Two transport equations are defined in the fourth unit, one for momentum transport (9.1) and one for pressure-Poisson (9.3):

```
80   Pressure pr(p, f, uvw, time, solver, &fluid);
81   Momentum ns( uvw, xyz, time, solver, &fluid);
```

Attributes sent to `Pressure` constructor are the pressure `p` and its source `f`, used for storing the right hand side of Eq. 9.3, which is computed from velocity `uvw`, sent as a third parameter. `Momentum` constructor takes velocity field `uvw` and its forces `xyz` as parameters. Both `Pressure` and `Momentum` constructor need variable defining simulation time, solver and substance, represented in lines 80 and 81 as objects `time`, `solver` and `fluid`.

### 9.1.5 Preparation for the time-loop

The fifth unit contains only one line:

```
83   AC multigrid( &pr );
```

which creates a multigrid solver for the `Pressure p`. When solving transient transport equations, it is sound to create a multigrid solver for pressure only. Transported variables (momentum, enthalpy, ...) give well-conditioned systems which can be solved with `Krylov` solver only. Even more, multigrid solver *can not* be defined for momentum equations, since they change their resolution due to staggered arrangement of variables. No variable needs initialization for this problem.

This program features modest $Re$, but for this geometry, it should yield and oscillatory solution, *i.e.* it should lead to vortex shedding. Therefore, it would be convenient if PSI-Boil was printing the value of certain variable at a point in the domain during the simulation to check whether an oscillatory solution has been reached, or to check the frequency of oscillations, etc. There is a class which does just that. It is called `Location` and is defined in `Src/Monitor/Location`. Here, we define a `Location` as:

```
85   Location loc("monitor", d, NX1, NY/2, NZ/2);
```

It creates `Location loc`, called `"monitor"`, in the `Domain d`, at logical coordinates ($i$, $j$ and $k$) equal to `NX1`, `NY/2`, and `NZ/2`. That `Location` will be placed midway between the walls in the region where uniform (`gx1`) and non-uniform (`gx2`) grids meet. It's usage is explained below.

### 9.1.6 The time-loop

Finally, we reach the time loop of the program, reading:

[1] Keep in mind that maximum value of inlet velocity is 1.5 and that $\Delta x = dx$

```
 90   for(time.start(); time.end(); time.increase()) {
 91
 92     boil::oout << "##################" << boil::endl;
 93     boil::oout << "#                  " << boil::endl;
 94     boil::oout << "# TIME:      " << time.current_time() << boil::endl;
 95     boil::oout << "#                  " << boil::endl;
 96     boil::oout << "# TIME STEP: " << time.current_step() << boil::endl;
 97     boil::oout << "#                  " << boil::endl;
 98     boil::oout << "##################" << boil::endl;
 99
100     ns.cfl_max();
101
102     ns.new_time_step();
103
104     ns.solve(ResRat(1e-2));
105
106     p = 0.0;
107
108     multigrid.vcycle(ResRat(1e-2));
109
110     ns.project(p);
111
112     loc.print(uvw, Comp::v());
113
114     if( time.current_step() % 100 == 0)
115       boil::plot->plot(uvw,  p, "uvw,p",  time.current_step());
116   }
```

This loop embodies an implementation of fractional step algorithm in PSI-Boil. In line 90 you can see how can variable `time` be used to cycle through time. It's member functions `Times::start()`, `Times::end()` and `Times::increase()` serve as parts of C++'s `for` loop. The advantage is that this loop remains the same, no matter how the variable `time` was defined. Two additional `Times`' member functions are used to plot current simulation time and time step in lines 94 and 96.

Line 100 calls `Momentum`'s member function `Momentum::cfl_max` which computes and prints the value of CFL number. Is it not necessary to call it, but it may serve as a good indication why a certain simulation crashed[2]. If CFL is too low, on the other hand, we are wasting a lot of computational time. As a *rule of thumb*, optimum CFL is in the range from $0.35 - 0.4$.

Preparation for the new time step (which involves computation of terms with superscript $n-1$ and $n-2$ in Eq. 9.4) are performed at line 102. That is followed by it's solution at line 104. At that point, we have tentative velocity ($\mathbf{u}^\star$).

In line 108, PSI-Boil uses the tentative velocity field to compute the right hand side of Eq.9.3 and solve it to get the new pressure field. For unsteady simulations, convergence of the multigrid solution for the pressure usually improves if pressure is initialized to zero. It is performed in line 106.

This pressure field is used to *project* tentative velocity ($\mathbf{u}^\star$) into a new ($n$) divergence-free field ($\mathbf{u}^n$). This closes the fractional step algorithm for one time step, and the new one can start.

Line 112 shows how can an object of type `Location` be conveniently used to monitor the computed values. It prints the value of the argument (here `Vector` and it's component) at position defined in line 91, during the construction of `Location`.

The programs plots the results (for velocities and pressure) every $100^{th}$ from lines 114 and 115.

[2] If CFL numbers gradually increase, and gradually reach values higher than 0.5, simulation will be inaccurate and will probably crash due to a too high time step.

### 9.1.7 Running the program

Compile and run this program. Run it with:

```
> ./Boil > out &
```

to store the terminal output to file "out". This simulation might take few hours to finish. If you would like to check the progress during the run, issue the command:

```
> tail -100f out
```

As an example, output for time step 20 looks like:

```
##################
#
# TIME:      0.0152148
#
# TIME STEP: 20
#
##################
cfl max = 0.230888 in direction u  at: 0.192187 0.265859 -0.0384375
u, residual = 6.87374e-07, ratio = 0.000281587
v, residual = 4.13061e-07, ratio = 0.000270693
w, residual = 2.03872e-08, ratio = 0.000250742
FILE: momentum_scale_out.cpp, LINE: 24, volf_in = 0.0840603
FILE: momentum_scale_out.cpp, LINE: 77, ratio = 1.00061
@get_src; err sou = 0.471205 -0.0399352
Initial  res = 0.00387017
Cycle 1; res = 0.000186533
Cycle 2; res = 8.59826e-05
Cycle 3; res = 2.87255e-05
Converged in 3 cycles!
monitor: -0.00159419
```

The lines beginning with a # only mark the beginning of a time step. CFL is printed right after that. The following two lines show the result of solving of momentum equations. `residual` is the residual of the solving procedure, and `ratio` shows the level of reduction of residuals in the solvers. (For example, for `w`, residuals were reduced by roughly 4000 times). The following two lines (beginning with `FILE`), are actually the development lines (see Sec. 3.2) which print the inlet bulk velocity and the ratio between inlet and outlet bulk velocities. The former depends on inlet boundary condition, while the latter should always be close to unity. Line beginning with `@get_src` prints the square and absolute integral of mass error created by tentative velocity field. While the square of the error may have any value, absolute integral should be as close to zero as possible.

The line which follows shows the residual history of the multigrid algorithm. The final line is the output created by program line 112, i.e. by `Location`. It can be used to plot the history of $v$ velocity component. Here is a simple way how to do it. Once the simulation is finished, run the command:

```
> cat out | grep monitor > monitor.dat
```

That will create the file "monitor.dat" which looks like:

```
1 Location monitor at x = 0.406797, y = 0.201797, z = -0.0128125 created.
```

```
2 monitor: -0.00246605
3 monitor: -0.002181
4 monitor: -0.00406565
5 monitor: -0.00401776
6 monitor: -0.00400525
7 monitor: -0.00399137
8 monitor: -0.00392429
...
```

Erase the first line, and all occurrences of: `"monitor:"` to get a file like:

```
1  -0.00246605
2  -0.002181
3  -0.00406565
4  -0.00401776
5  -0.00400525
6  -0.00399137
7  -0.00392429
...
```

You could have also created such a file using three UNIX commands `cat`, `grep` and `awk` in a single line:

```
cat out | grep monitor | awk '{print $2}' > monitor.dat
```

Plot this line using `grace` or `gnuplot` to see the time-history. It is plotted in Fig. 9.3. You can clearly see the history of $v$ velocity component at $x = 0.41$, $y = 0.2$ and $z = 0.0$. Obviously the flow has reached oscillatory regime.

Figure 9.3: Time-history of $u$ velocity component at $x = 0.41$, $y = 0.2$ and $z = 0.0$.

### 9.1.8 Results and footer output

Just before the end of the time loop, there are lines for creating results:

```
114     if( time.current_step() % 100 == 0)
115       boil::plot->plot(uvw,  p, "uvw,p",  time.current_step());
```

These lines save results for velocity and pressure every 100 time steps in files called: `uvw,p_p000_0100.dat`, `uvw,p_p000_0200.dat`, `uvw,p_p000_0300.dat`, etc. Visualization of pressure and velocity field for the final time step is given in Fig. 9.4 and Fig. 9.5 respectivelly..

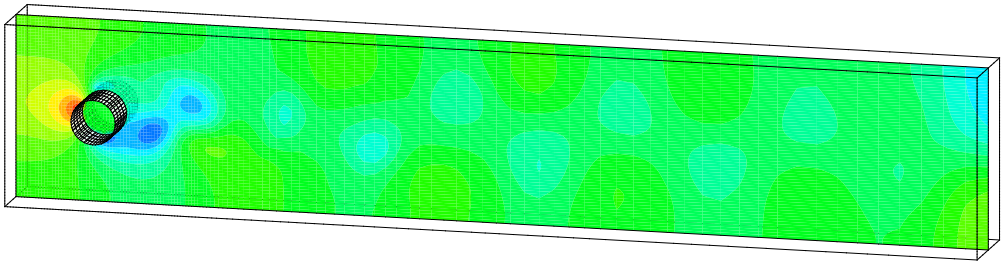


Figure 9.4: Pressure field at the end of simulation.

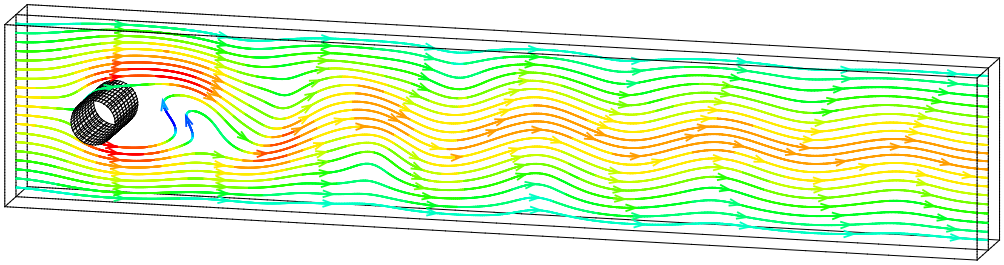


Figure 9.5: Stream-lines colored with stream-wise velocity component at the end of simulation.

At the end of simulation, you will get the following information in the file `out`, to which the terminal output is redirected:

```
+=========================================
| Total execution time: 7857.54 [s]
+-----------------------------------------
| Time spent in bounding box       : 0.01 [s]    (0.00127267%)
| Time spent in cell cutting       : 0.04 [s]    (0.00509068%)
| Time spent in flood fill         : 0.05 [s]    (0.00636335%)
| Time spent in plotting           : 59.22 [s]    (0.75342%)
| Time spent in pressure discretize: 0.01 [s]    (0.00127267%)
| Time spent in momentum discretize: 0.07 [s]    (0.00890869%)
| Time spent in momentum solver    : 428.55 [s]    (5.45339%)
| Time spent in vcycle             : 5885.13 [s]    (74.8979%)
| Time spent in coarsening         : 6.9 [s]    (0.878142%)
| Time spent elsewhere             : 1413.92 [s]    (17.9943%)
+-----------------------------------------
```

As stated above, quite a few PSI-Boil objects have built-in local timers. Here you see a typical situation: pressure solution, since the system is poorly conditioned, takes almost 75% of CPU-time. Momentum equations, thanks to unsteady term which conditions the system well, takes

slightly more than 5%, a staggering difference.

Section 9.1 in a nutshell

- When prescribing boundary condition for velocity, it must be done for *each* component.

- PSI-Boil supports analytically prescribed values for boundary conditions as long as they are functions of coordinates $x$, $y$ and $z$.

- Periodic boundary condition *must* be imposed on the variable, regardless of the fact periodicity has been defined for the grid.

- Boundary conditions should be prescribed for *all* boundaries.

- There is *no* default boundary condition in PSI-Boil.

- `Momentum`'s member function `Momentum::cfl_max` can be used to check whether the time step is appropriate for the simulation.

- If CFL exceeds the value of 0.5, the simulation will likely become unstable.

- As a *rule of thumb*, it is good to chose a time step which gives CFL around 0.35.

- Object of type `Location` can be used to monitor time-histories of the solution at specified locations inside the computational `Domain`.

## 9.2 Thermally-driven cavity flow

This case was chosen to demonstrate the coupling of enthalpy and momentum equations. The problem is illustrated in Fig. 9.6. It is a cavity with a square-shaped cross-section, with differentially heated side walls. Top and bottom walls are insulated. Buoyancy forces give rise to circular motion of the fluid inside the cavity.

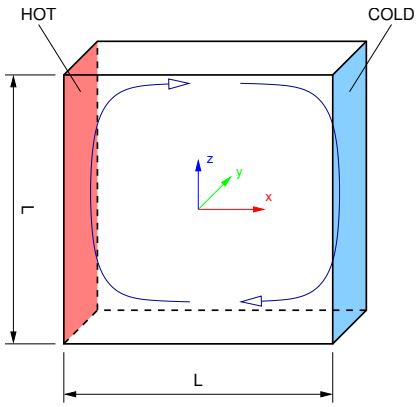



Figure 9.6: Geometry, boundary conditions and basic topology for the thermally-driven cavity flow.

The program which solves this problem is stored as: `09-02-main.cpp`. The features which are new for this problem are described in detail, while the concepts which have already been explained

above (grid generation, boundary conditions, etc.) are merely mentioned.

The governing equations in non-dimensional form read:

$$\int_{V^\star} \frac{\partial T^\star}{\partial t^\star} dV^\star + \int_{S^\star} \mathbf{u}^\star T^\star \, d\mathbf{S}^\star = \int_{S^\star} \nabla T^\star \, d\mathbf{S}^\star \tag{9.7}$$

$$\int_{V^\star} \frac{\partial \mathbf{u}^\star}{\partial t^\star} dV^\star + \int_{S^\star} \mathbf{u}^\star \mathbf{u}^\star \, dS^\star = Pr \int_{S^\star} \nabla \mathbf{u} \, dS^\star - \int_{V^\star} \nabla p^\star \, dV^\star + RaPr \int_{V^\star} \mathbf{d}\theta \, dV^\star \tag{9.8}$$

if the scales for length, velocity, time and pressure are $L$, $\alpha/L$, $L^2/\alpha$ and $\alpha^2\rho/L^2$ respectively. Here $\alpha = \lambda/(\rho C_p)$ is thermal diffusivity coefficient. Non-dimensional temperature is defined as $T^\star = (T - T_C)/(T_H - T_C)$ This problem can be fully characterized with two non-dimensional numbers: the Rayleigh ($Ra = \rho g \beta \Delta T L^3/\mu\alpha$) and the Prandtl ($Pr = \mu/\alpha\rho$) number. These two numbers are defined as a constants at the beginning of the program:

```
10 const real Pr = 0.71;
11 const real Ra = 1.0e+5;
```

The problem is two-dimensional, but PSI-Boil can not handle purely two-dimensional grids. Two-dimensionality can be *mimicked* by imposing the periodicity in the homogeneous direction. This is achieved by the following part of the code:

```
 4 const real LX =   1.0;
 5 const real LY =   0.125;
 6
 7 const int NX = 64;
 8 const int NY =  4;
...
18   /*----------+
19   |  grid(s)  |
20   +----------*/
21   Grid1D gx( Range<real>( -0.5*LX, 0.5*LX ), NX, Periodic::no());
22   Grid1D gy( Range<real>( 0, LY ),           NY, Periodic::yes());
...
27   Domain d(gx, gy, gx);
```

As defined in Fig. 9.6, homogeneous direction is $y$. The resolution in $y$ direction is only four cells, a *minimum* resolution which can be defined in PSI-Boil.

For this case, we need variables for temperature, enthalpy source, momentum and its force, pressure and its source. They are defined with:

```
32   Vector uvw(d), xyz(d); // vel
33   Scalar p  (d), f  (d); // p.
34   Scalar t  (d), g  (d); // t.
```

Boundary conditions are defined in lines 39–60, and need no further explanation. It is maybe worth reminding that periodic boundary conditions must be set to variables, notwithstanding they were defined for grids.

Boundary condition section is followed by the definition of physical properties. This is a particular case, solved in non-dimensional form (Eq. 9.7 and 9.8), so the only "property" which has to be changed is "dynamic viscosity":

```
65   Matter fluid(d);
66   fluid.mu( Pr );
```

Setting the number of time steps and selection of Krylov solver are done in lines 68 and 70, and need no further explanation. Pressure-Poisson equation, as well as transport equations are defined in lines 75–77:

```
75   Pressure pr  ( p,   f,   uvw, time, solver, &fluid);
76   Momentum ns  ( uvw, xyz,      time, solver, &fluid);
77   Enthalpy enth( t,   g,   uvw, time, solver, &fluid);
```

Just before the time loop, multigrid solver is defined in line 79 and a monitoring location (called `m0`) in line 81:

```
79   AC multigrid( &pr );
80
81   Location m0("m0", d, NX/2, NY/2, NX/4);
```

Finally, we reach the time loop, which, in essence, is an *evolution* of the one presented in Sec. 9.1:

```
83   for(time.start(); time.end(); time.increase()) {
...
93     enth.new_time_step();
94     enth.solve(ResRat(0.001));
95
96     ns.cfl_max();
97     ns.new_time_step();
98
99     Comp m = Comp::w();
100    for_vmijk(xyz,m,i,j,k)
101      xyz[m][i][j][k] = Pr*Ra * 0.5*(t[i][j][k]+t[i][j][k-1]) * xyz.dV(m,i,j,k);
102
103    ns.solve(ResRat(0.001));
104
105    multigrid.vcycle(ResRat(0.001));
106
107    ns.project(p);
108
109    pr.update_rhs();
110
111    m0.print(uvw,Comp::u());
112  }
```

Each time step starts with computation of enthalpy. That is possible, because convective terms are discretized with Adams-Bashforth time stepping scheme, meaning that it uses velocity and temperature field in old time step $(n-1)$ and the time step before old $(n-2)$. Right-hand side of enthalpy equation is assembled in line 93, and the linear solver is invoked from line 94.

That is followed by computation of momentum equations. Right hand side terms depending on old time steps are computed in line 97. For this case, however, that is not all. Temperature field is coupled with momentum equations via the Boussinesq approximation defined in lines 99–101. It is worth noting that macro `for_vmijk` has been used again, with parameter `m` specifying the direction in which the buoyancy acts. It is worth noting that even for this case, the right hand side is integrated over cell using mid-point rule, i.e.: multiplying the source with cell volume. Cell volume is stored in `Vector`'s member function `Vector::dV(m,i,j,k)` which, unlike the `Scalar`'s variant takes one parameter more, for vector component. The reason for that lies in the fact that cells of a staggered generally have different volumes than centered cells, as illustrated in Fig. 9.7.

Figure 9.7 also makes it clear why temperature for the source term in program line 101 is taken as an arithmetic average between `t[i][j][k]` and `t[i][j][k-1]`.

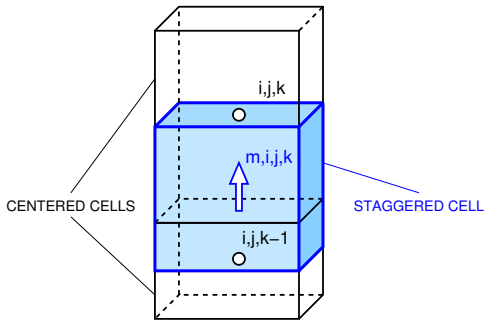



Figure 9.7: Volume of a staggered (`Vector`'s) cell is different from volume of a centered (`Scalar`'s) cell. `Scalar::dV(i,j,k)` $\neq$ `Vector::dV(m,i,j,k)`

Lines 103-107 are the same as in Sec. 9.1 and need no explanation. A novelty is in line 109, which calls for an explicit update[3] of the right-hand side of pressure-Poisson equation. It is a trick to get the mass (im)balance after the projection of velocities into a divergence-free field. To illustrate it, it is worth taking a look at the part of the output of this program:

```
@get_src; err sou = 1.59193 4.2526e-12
Initial  res = 4.68899e-08
Cycle 1; res = 2.01202e-08
Cycle 2; res = 1.61724e-09
Cycle 3; res = 2.19523e-10
Cycle 4; res = 3.98007e-11
Converged after 4 cycles!
@get_src; err sou = 4.37667e-12 6.17362e-14
```

Compare two lines beginning with `@get_src`. Te first one is result of implicit call from line 105, while the last is result of explicit call from line 109. Comparing the first number of the two (`1.59193` and `4.37667e-12`), gives an indication of the success of projection step. In this case, it is very successful, since projection step reduces mass error by many orders of magnitude.

Program line 111 produces time history of $u$ velocity component, plotted in Fig. 9.8. It is quite convincing that simulation has reached the steady state.

Figure 9.8: Time-history of $u$ velocity component at $x = -0.01$, $y = -0.05$ and $z = -0.26$.

[3] It is called implicitly from `multigrid.vcycle(ResRat(0.001));`.

### 9.2.1 Results and footer output

Results, plotted from lines 114–116 (not shown here) are displayed in Fig. 9.9.

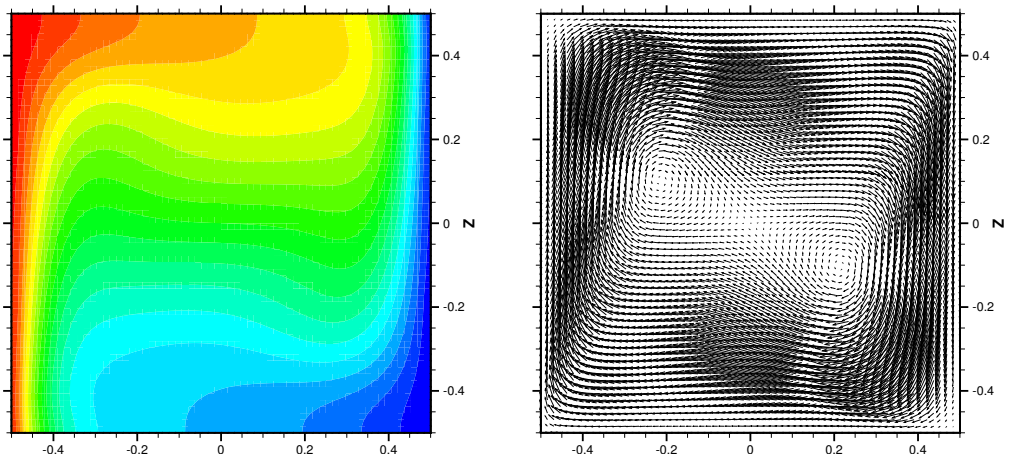



Figure 9.9: Non-dimensional temperature (left) and velocity field (right) for the thermally driven cavity at $Pr = 0.71$ and $Ra = 10^5$.

Imagine you wanted to extract some profiles from the results, say $u$ velocity profile at $x = 0$ and $w$ velocity profile at $z = 0$. These profiles might be extracted from Tecplot, which was used to create plots in Fig. 9.9, but that will not give the values computed by PSI-Boil, but rather interpolations of these results performed internally by Tecplot.

PSI-Boil offers a class which can extract profiles from solution, in much the same way certain locations in the computational `Domain` are monitored with the object of type `Location`. The class which extract profiles is called `Rack`, defined in `Src/Monitor/Rack`, and its usage is demonstrated in lines:

```
118   Rack    r0("u-comp", d, NX/2+1, NY/2, Range<int>(1,NX));
119   Rack    r2("w-comp", d, Range<int>(1,NX), NY/2, NX/2+1);
120   r0.print(uvw,Comp::u());
121   r2.print(uvw,Comp::w());
```

Line 118 creates a measuring `Rack` named `"u-comp"` in `Domain d`, at logical coordinates: ($i$ and $j$) equal to `NX/2+1` and `NY/2` while ranging in $k$ direction from `1` to `NX`. This `Rack` extracts $u$ velocity component in line 120. Line 119 creates a `Rack` oriented in $i$ direction while line 121 extract $w$ velocity profile from it. These profiles are printed on terminal[4] and this printing appears as:

```
Rack:u-comp
0.0078125 0.046875 -0.492188 -3.7066
0.0078125 0.046875 -0.476562 -10.3498
0.0078125 0.046875 -0.460938 -16.309
0.0078125 0.046875 -0.445312 -21.5107
0.0078125 0.046875 -0.429688 -25.8977
...
```

which list $x, y, z$ coordinates, followed by value of $u$ velocity component. These lists should be extracted from and saved in a separate file. A plot of these two profiles is shown in Fig. 9.10.

The profiles in Fig. 9.10 give quantitative data, but velocity components are not usually reported for this case. It is the Nusselt number ($Nu = \partial T^\star / \partial n^\star$) which is the target benchmark datum. As

[4] or in a file to which the terminal output was redirected with `./Boil > out &`

Figure 9.10: Non-dimensional $u$ velocity distribution along the line $x = 0$ (left) and $w$ velocity distribution along the line $z = 0$.

with velocity profiles, it is quite dangerous to extract this value from post-processing tool, since it is based on internal approximations done in the tool. The value of $Nu$ which is really computed in PSI-Boil *must* be extracted from the program itself. For this case, it is done with the following piece of coding:

```
123   const int ii = t.si();    /* i inside the domain */
124   const int iw = ii-1;      /* i in the wall */
125   const int j  = t.ej()/2; /* j in the middle */
126   boil::oout << " Nusselt number " << boil::endl;
127   for_vk(t,k) {
128     real nu = (t[iw][j][k] - t[ii][j][k]) / t.dxw(ii);
129     boil::oout << t.zc(k) << " " << nu << boil::endl;
130   }
```

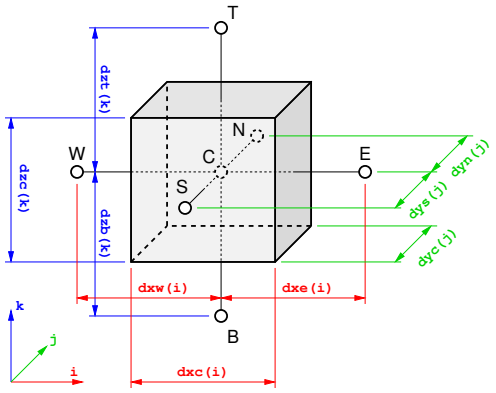



Figure 9.11: Cell dimensions.

Lines 123–125 define constants to avoid *ghost numbers*. Line 126 just sends a message to the terminal. Loop spanning from line 127 to 129 computes the $Nu$ and prints it to the terminal, together with $z$ coordinate. Here `t[iw][j][k]` is temperature at the wall, `t[ii][j][k]` is the temperature in the first computational cell inside the domain, and `t.dxw(ii)` is the distance between the cell centre at $i$ to the cell centre at $i-1$, which is in em this case at the wall. Distance `dxw(i)` as well as other dimensions for a general cell (not the one at the wall) are illustrated

in Fig. 9.11. Nusselt number for the heated wall ($x = -0.5$) is plotted in Fig. 9.12

Figure 9.12: Non-dimensional $u$ velocity distribution along the line $x = 0$ (left) and $w$ velocity distribution along the line $z = 0$.

Section 9.2 in a nutshell

- If solving a two-dimensional problem with PSI-Boil specify one coordinate direction as *homogeneous* by making it very thin compared to the others and by specifying periodic boundary conditions for it.

- Minimum resolution in PSI-Boil is four cells.

- When specifying forces for `Momentum` equations, integrate them over *staggered* volumes (`Vector::dV(m,i,j,k)`).

- You can use and additional call to `Pressure::update_rhs` after the projection of velocity, to check the mass conservation reduction level.

- Extracting profiles from post-processing packages is strongly discouraged, since they extract *interpolated* solutions, not the *computed* ones.

- Profiles which are really computed in PSI-Boil can be extracted with objects of class `Rack`.

- Usage of `Rack` is the same as that of `Location`, except that, instead of three coordinates, one sends two coordinates and one `Range` to it's constructor.

- You can access all relevant cell dimensions using `Scalars` member functions `dxc`, `dyc`, `dzc`, `dxw`, `dxe`, `dys`, `dyn`, `dzb` and `dzt`.

## 9.3 Flow around a cube matrix

In this section a flow around the cube matrix is solved using Direct Numerical Simulation (DNS). It is a DNS in sense that not sub-grid-scale (SGS) model is used, rather in the sense that it resolves *all* scales of motion. Cube matrix is placed at the bottom of the plane channel. The flow

is assumed to be fully developed, so it only suffices to simulate one flow segment (flow around one cube) with periodic boundary conditions applied in stream-wise ($x$) and span-wise ($y$) direction. Problem domain is illustrated in Fig. 9.13. The dimensions of the problem domain are as follows: $D = 6\,[cm]$, $H = 5.1\,[cm]$ and $h = 1.5\,[cm]$. Working fluid is air, having density $\rho = 1.205\,[kg/m^3]$ and kinematic viscosity $\nu = 1.511 \times 10^{-5}$.

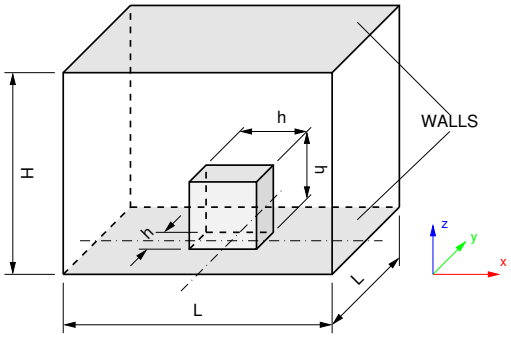



Figure 9.13: Problem domain for the flow around a cube matrix.

Flow rate for this case is specified with stream-wise bulk velocity equal to $u_b = 3.86\,[m/s]$. Reynolds number based on channel height is $Re \approx 13000$. At this $Re$, the flow is turbulent. This case, in fact, is the first simulation of a turbulent flow in this tutorial. PSI-Boil is a tool which simulates turbulent flows using Large Scale Simulation (LSS) approaches, namely the DNS and LES[5]

Simulation of turbulent flows with LES/DNS consists of two steps:

- Unsteady flow simulation
- Computation of flow statistics

First step generates a sequence of unsteady flow realizations stored on a disk, while the second step reads the flow realizations from disk and computes necessary statistics (Reynolds stresses, turbulent kinetic energy, correlation, turbulent spectra, etc.). Having PSI-Boil's philosophy in mind[6] two different programs will be outlined in this section: one for unsteady flow simulation `09-03-main.cpp` and the other for computation of flow statistics `09-03-stat.cpp`. Obviously these two codes share some object definitions. The shared objects are stored in `C++` header file `09-03-common.h`.

### 9.3.1 Unsteady flow simulation

The program which conducts the unsteady flow simulation is very similar to program introduced in Sec. 9.1. It also follows closely the typical PSI-Boil program layout introduced in Chap. 8.

To compile it, you do not only need a link to *main* program in *source* directory (`PSI-Boil/Src`), but also to the include file (`09-03-common.h`). You can get the main program and the include file, provided you are in source directory, with:

```
> ln -i -s ../Doc/Tutorial/Volume1/Src/09-03-main.cpp main.cpp
> ln -i -s ../Doc/Tutorial/Volume1/Src/09-03-common.h .
```

[5] In more general sense, LSS includes also simulation of turbulent flows with interface tracking, but as it is still an active area of research, nomenclature is not yet fully established.

[6] It is not an integral package suited for all flow situations one can encounter, but rather a suite of objects which facilitates building of different algorithms for flow simulation.

The include file holds grids and domain definition, and is included in the main program with the:

```
13   #include "09-03-common.h"
```

There is nothing new in these definitions, so there is no need to explain them in more detail here. `Times` object is defined in line:

```
16   Times time(100000, 0.00002); /* ndt, dt */
```

For an LSS, we do not expect to get a steady solution to the problem. We are aware (and hopeful) that simulation will yield an unsteady solution and we want to perform enough time steps to get a relevant statistical sample for later computation of flow statistics. Here we set the number of time step to 100000. Time step which gives the CFL number in the stable range $(0.3 - 0.4)$ is 0.00002. The physical time of this simulation is thus $2\,[s]$.

The program defines variables (velocity, pressure and their sources) in lines 21–22. Boundary conditions for variables are set in lines 24–38. That is followed by the definition of materials (lines 43–45), solver (line 47), transport equations (lines 52 and 53). As usually, multigrid solver is defined for pressure equation (in line 59). For this case, the flow is not initialized, but the forces in momentum equations are (lines 61 and 62).

The time loop, which spans from lines 67–105, has all the steps introduced before, say in Sec. 9.1, but it has a small section which re-computes the pressure drop at each time step to keep the bulk velocity constant, and a section which periodically saves the data for later computations of flow statistics.

Mass flow in computational domain is kept constant using the second Newton's law:

$$F = m \cdot a \quad [N] \tag{9.9}$$

If applied to a computational domain, acceleration $a$ is the rate of change of bulk velocity. Force $F$, on the other hand is integrated pressure drop in the domain. Hence, second Newton's law, can be written as:

$$\frac{\partial p^n}{\partial x} V = m \cdot \frac{u_b^n - u_b^{n-1}}{\Delta t} \quad [N] \tag{9.10}$$

We want the bulk velocity at new time step ($u_b^n$) to be equal to the prescribed (desired) bulk velocity ($u_{des}$). Furthermore, acknowledging that mass $m$ divided by volume $V$ is density, the expression for pressure drop which gives the desired bulk velocity in the new time step is:

$$\frac{\partial p^n}{\partial x} = \rho \cdot \frac{u_{des} - u_b^{n-1}}{\Delta t} \tag{9.11}$$

The implementation of this expression is given in lines 91–96.

```
91     real b_new = ns.bulk(Comp::u(), LX*0.333);
92
93     real p_drop = fluid.rho() * (b_des - b_new) / time.dt();
94
95     Comp m = Comp::u();
96     for_vmijk(xyz,m,i,j,k) xyz[m][i][j][k] = p_drop * uvw.dV(m,i,j,k);
```

Line 91 computes bulk velocity in $x$ direction using `Momentum`'s member function `bulk`. As the parameter, this function takes the desired component (`Comp`) and the $x$ coordinate at which the bulk velocity is computed[7]. Line 93 is the implementation of Eq. 9.11, while lines 95 and 96 insert new pressure drop to the momentum equation.

Periodic savings of turbulent velocity fields are performed in line:

[7] Although it should be the same at any position.

```
102     if( time.current_step() % 50 == 0 ) uvw.save("uvw", time.current_step());
```

The line 102 checks the reminder of division of the current time step with 50, and if it is zero, velocity is saved in the binary format, using the `Vector`'s member function `save`. This member function is available for `Scalar`s as well. The `save` functions just dumps the memory occupied by a certain field variable into a binary file. As such, it can not be visualized with a post-processing tool. The `save` command saves the files with extension `.bck` (to remind of *backup*). As with other PSI-Boil output function, even here each processors writes it's own file but they can *not* be connected into a single one using the `Connect` program.

Lines 103 and 104:

```
103     if( time.current_step() % 5000 == 0) /* 5000 */
104       boil::plot->plot(uvw,p,"uvw,p", time.current_step());
```

store variables (velocity and pressure) for post-processing each 5000 time steps. Examples of these unsteady flow fields (realizations) are given in Fig. 9.14.

### 9.3.2 Computation of flow statistics

Program for computation of flow statistics (`09-03-stat.cpp`) does not follow the typical PSI-Boil's program structure. Therefore, it is given and explained here in full detail.

```
 1 #include "Include/psi-boil.h"
 2
 3 #include <vector>
 4
 5 /******************************************************************************/
 6 main(int argc, char * argv[]) {
 7
 8   boil::timer.start();
 9
10   /*------------------------------+
11   |  grids, obstacles and domain  |
12   +------------------------------*/
13   #include "09-03-common.h"
14
15   /*------------------+
16   |  define unknowns  |
17   +------------------*/
18   Vector uvw(d);   // velocity
19   Scalar ut(d);  Scalar vt(d);  Scalar wt(d);     // temporary u, v, w
20   Scalar u (d);  Scalar v (d);  Scalar w (d);     // averaged  u, v, w
21   Scalar uu(d);  Scalar vv(d);  Scalar ww(d);     // averaged  uu,vv,ww
22   Scalar uv(d);  Scalar uw(d);  Scalar vw(d);     // averaged  uv,uw,vw
23
24   u  = 0; v  = 0; w  = 0;
25   uu = 0; vv = 0; ww = 0;
26   uv = 0; uw = 0; vw = 0;
27
28   /* timer */
29   Times time(80000, 0.00002); /* ndt, dt */
30   time.first_step(20000);
31
32   /*------------+
33   |  time loop  |
```

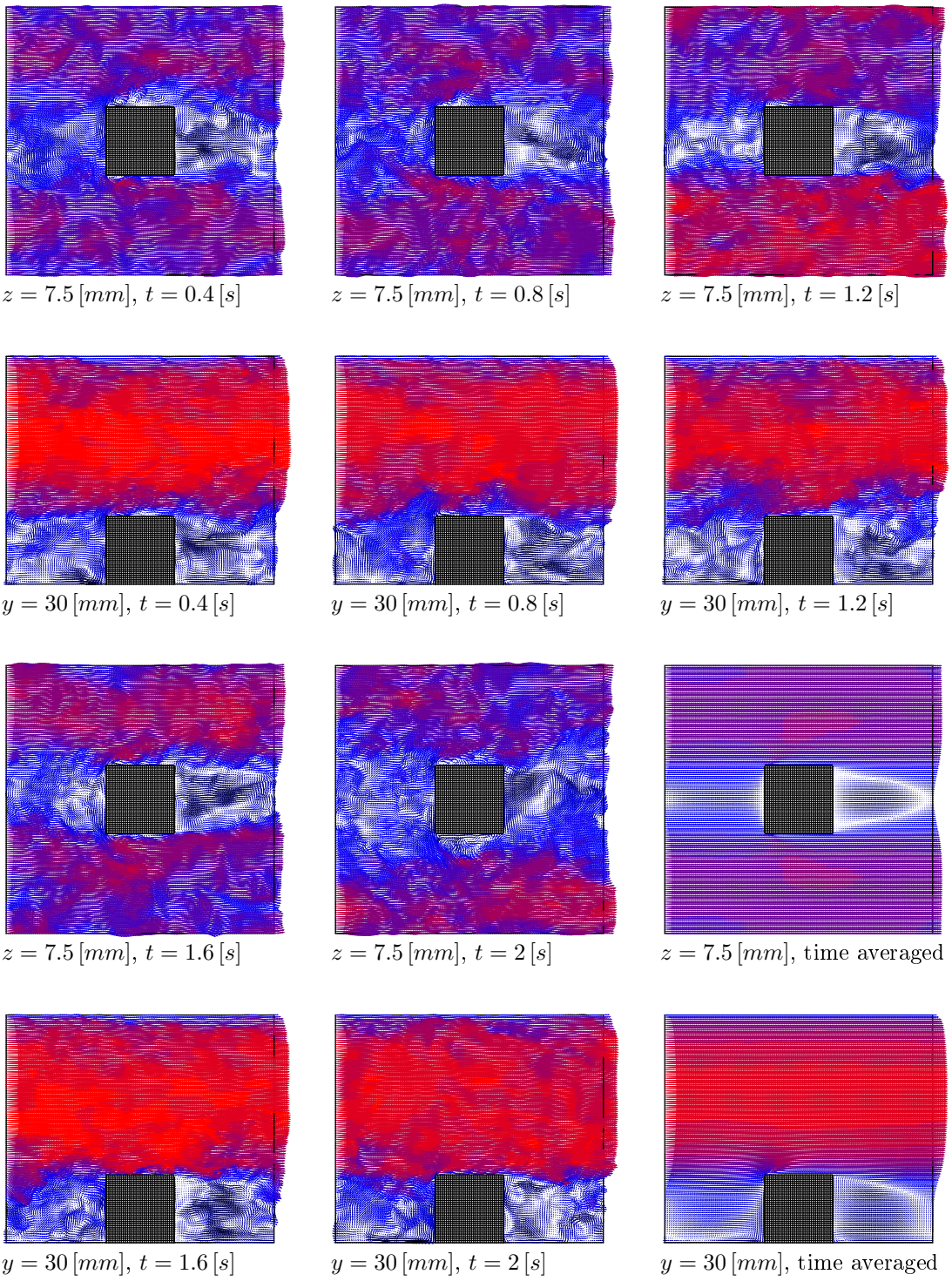



Figure 9.14: Unsteady and time-averaged velocity fields colored by the magnitude of streamwise ($x$) velocity component.

```
34   +------------*/
35   real count = 0;
36
37   const Comp U = Comp::u();
38   const Comp V = Comp::v();
39   const Comp W = Comp::w();
40
41   for(time.start(); time.end(); time.increase()) {
42
43     /*-------+
44     |  load  |
45     +-------*/
46     if( time.current_step() % 100 == 0 ) {
47       boil::oout << "##########################" << boil::endl;
48       boil::oout << "# GETTING TIME STEP: " << time.current_step() << boil::endl;
49       boil::oout << "#-------------------------" << boil::endl;
50
51       count ++;
52
53       uvw.load("uvw", time.current_step());
54
55       for_vijk(u,i,j,k) {
56         ut[i][j][k] = 0.5 * (uvw[U][i][j][k] + uvw[U][i+1][j]  [k]  );
57         vt[i][j][k] = 0.5 * (uvw[V][i][j][k] + uvw[V][i]  [j+1][k]  );
58         wt[i][j][k] = 0.5 * (uvw[W][i][j][k] + uvw[W][i]  [j]  [k+1]);
59       }
60
61       u  += ut;     v  += vt;     w  += wt;
62       uu += ut*ut;  vv += vt*vt;  ww += wt*wt;
63       uv += ut*vt;  uw += ut*wt;  vw += vt*wt;
64     }
65   }
66
67   u  /= count;  v  /= count;  w  /= count;
68   uu /= count;  vv /= count;  ww /= count;
69   uv /= count;  uw /= count;  vw /= count;
70
71   uu -= u*u;  vv -= v*v;  ww -= w*w;
72   uv -= u*v;  uv -= u*v;  vw -= v*w;
73
74   /*-------+
75   |  plot  |
76   +-------*/
77   boil::plot->plot(u, v, w,   "velocity-mean", time.current_step()-1);
78   boil::plot->plot(uu,vv,ww,  "stresses-mean", time.current_step()-1);
79
80   boil::timer.stop();
81   boil::timer.report();
82 }
```

This program includes the same file (`09-03-common.h`) as the program for unsteady flow simulation to ensure that grids and domains are the same. Fields for unknowns are defined in lines 18–22. The `Vector uvw` is defined to loading the results generated in previous step, and 12 more scalars which follow will hold temporary (unsteady) velocity components (`ut`, `vt`, `wt`), time-averaged velocity components (`u`, `v`, `w`) and Reynolds stress components (`uu`, `vv`, ... `vw`). The `Scalar` field which are used for accumulating statistics are initialized in lines 24–26.

Line 29 defines the `Timer`, but contrary to the previous program, it will execute *only* 80000 time

steps. However, the time stepping does not start from zero, but from time step 20000, stipulated in line 30. By doing so, we will discard first 20000 time steps (corresponding to $0.4\,[s]$ physical time), thus gather statistics from the time when the flow is already fully developed[8].

`real` variable `count` is defined and initialized in line 35. It stores the number of samples (flow field realizations) used in the computation of statistics.

Time loop starts at line 41. This loop increases the counter in line 51 and then reads the results performed in previous simulation (program `09-03-main.cpp`) in line 53. `Vector`'s member function `load` is used to read the results and it does exactly the opposite from `save`: it reads the binary data stored in the `.bck` file and loads it directly into variable's memory space. This line will work properly only if the program is executed *on the same number of processors* as the simulation program was.

Once the `Velocity` field is loaded into `uvw` field variable, it is interpolated into three `Scalar` fields in lines 55–59. `Scalar` fields `u`, `v` and `w` hold the unsteady velocity field components, which are added into mean ones in line 61, and to variables which will hold Reynolds stresses in lines 62 and 63. For the mean velocity components, simple addition is performed, while for the Reynolds stress variables appropriate multiplications are added. When the time loop ends (after line 65) these values are normalized by the number of samples (lines 67–69) and Reynolds stresses are finally computed in lines 71 and 72.

The procedure for computation of Reynolds stresses might need more explanation. A Reynolds stress component ($\overline{u'u'}$, for instance), at specified position is defined as:

$$\overline{u'u'} = \frac{1}{T}\int_T u'(t)u'(t)dt \tag{9.12}$$

where over-bar denotes time-averaged value, $T$ is the period of time averaging and $u'(t)$ is velocity fluctuation defined as:

$$u'(t) = u(t) - \overline{u} \tag{9.13}$$

where $u(t)$ is the magnitude of velocity component at time $t$, while $\overline{u}$ is the time-averaged velocity component defined as:

$$\overline{u} = \frac{1}{T}\int_T u(t)dt \tag{9.14}$$

What we really compute in line 56, and what we finally have after line 59 is actually:

$$\overline{uu} = \int_T u(t)u(t)dt \tag{9.15}$$

The relation between $\overline{u'u'}$ (which we want) and $\overline{uu}$ (which we can compute from flow realizations) can be obtained if definition of $u'(t)$ (Eq. 9.13) is introduced into Eq. 9.12:

$$\begin{aligned}\overline{u'u'} &= \frac{1}{T}\int_T (u(t)-\overline{u})(u(t)-\overline{u})dt \\ &= \underbrace{\frac{1}{T}\int_T u(t)u(t)dt}_{\overline{uu}} - 2\overline{u}\,\underbrace{\frac{1}{T}\int_T u(t)dt}_{=\overline{u}} + \overline{u}\,\overline{u}\,\underbrace{\frac{1}{T}\int_T dt}_{=1}\end{aligned} \tag{9.16}$$

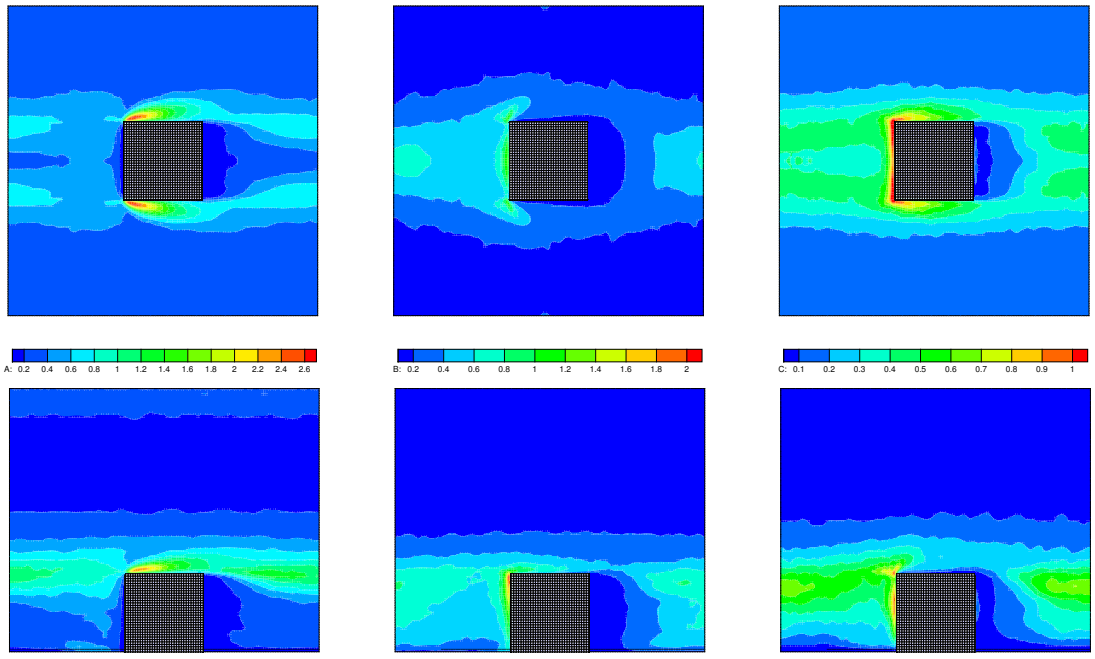
where we placed $\overline{u}$ is placed in front of time integrals, because it is constant in time by definition. So the final form of equation for computing $\overline{u'u'}$ is:

$$\overline{u'u'} = \overline{uu} - \overline{u}\,\overline{u} \tag{9.17}$$

which is exactly what program line 71 does.

[8] Actually, a much more concise procedure would be needed to ensure this, but it is beyond the scope of this tutorial.

Once all the statistics are computed, the program saves the time-averaged velocities from line 77 and diagonal trace of the Reynolds stress tensor from line 78. Time-averaged velocity fields are shown in bottom right corner of Fig. 9.14 and diagonal trace components of the Reynolds stress tensor in Fig. 9.15.




$\overline{u'u'}$ contours $\overline{v'v'}$ contours $\overline{w'w'}$ contours

Figure 9.15: Diagonal trace components of the Reynolds stresses tensor for the flow around the cube matrix. Top: contours in plane $z = 7.5\,[mm]$; bottom: contours in plane $y = 30\,[mm]$.

Section 9.3 in a nutshell

- Turbulent flows in PSI-Boil are simulated using LSS techniques: DNS and LES.

- Simulation by LSS techniques consists of two steps: unsteady flow simulation and computation of statistics.

- In PSI-Boil a separate program is written for each step.

- It is a good idea to place objects shared by these two programs in a separate include (`C++ header`) file.

- Bulk velocity in a computational `Domain` in $x$, $y$ or $z$ direction is in PSI-Boil computed using `Momentum`'s member functions `Momentum::bulk_i(real)`, `Momentum::bulk_j(real)` or `Momentum::bulk_k(real)`.

- `Scalar` and `Vector` type objects have member functions `save` and `load` which write and read their values in binary format. These file have the extension `.bck`.