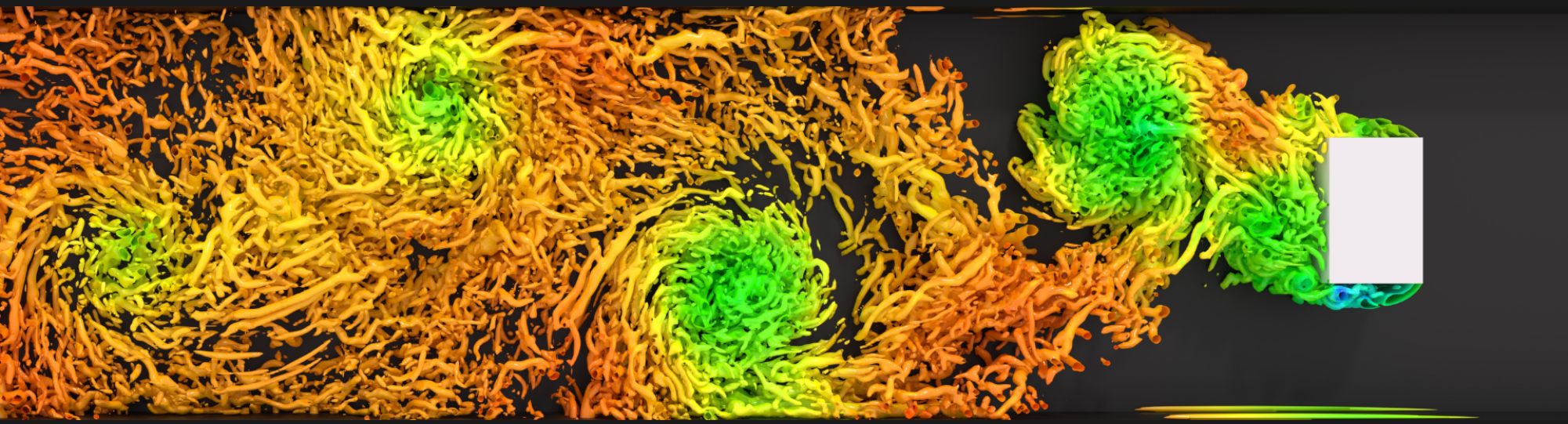


TCLB Solver

Making Lattice Boltzmann Method **efficient** on GPUs.

Łukasz Łaniewski-Wołłk

The Talk

1. Solver
2. How Things Work
3. Tooling

The Talk

1. Solver
2. How Things Work
3. Tooling

TCLB Solver

TCLB Solver is designed to wrap a fully functional, high-performance, parallel solver around a simple to write model description.

Model

By "model" we mean here a Lattice Boltzmann Method scheme, or some **explicit** finite difference scheme.

Model description

Dynamics.R

```
AddField(name="u", dx=c(-1,1), dy=c(-1,1))
AddSetting(name="Alpha")
AddSetting(name="Value", zonal=TRUE)
```

+

Dynamics.c

```
CudaDeviceFunction void Init() {
    u = Value;
}

CudaDeviceFunction void Run() {
  real_t lap_u = u(-1,0) + u(1,0) + u(0,-1) + u(0,1) - 4*u(0,0);
  u = u(0,0) + Alpha * lap_u;
}
```

Model description — concepts

- `Field` — a field which has some value in every node of the lattice, and can be accessed in a declared stencil.
- `Density` — a way to "pull" the value of a `Field` from a predefined direction.
- `Setting` — a value which can be prescribed in the xml file and used in the iteration
- `Quantity` — values exported to the VTK and other files, like velocity or pressure.
- `Global` — global integral values calculated in the iteration and summed (or maxed) over all GPUs
- `Stage` — node operation like LBM collision, which loads, transforms and writes fields. There is no MPI data exchange in a single stage.
- `Action` — A group of `Stages` executed consecutively. In most models there is only `Init` and `Iteration`.
- `NodeType` — A flag that can be set to a node, visible in the iteration.

TCLB as a framework

The resulting solver takes care of:

- Reading `xml` input files
- Simple and STL geometries
- Domain decomposition and MPI parallelism
- Efficient execution on CUDA/HIP enabled GPU
- Memory, communication and kernel execution concurrency
- Efficient parallel reduction (integration over domain)
- Writing output (`VTK`, `HDF5`, `CSV`, ...)
- and other things ...

Code generation

The resulting solver ("CLB") is generated from the template ("TCLB") using `R` package `rtemplate`.

Advantages

- Code can be optimised before even being compiled.
- Code can be easily inspected.
- Only needed code and variables are included.

Disadvantages

- Many modern tools don't cope well with such a setup.
- Code generation can be sometimes slow.

Code generation — Why R?

Advantages

- I know it.
- Anything is better than `python`

Code generation — Why R?

Advantages

- I know it.
- Anything is better than `python`
- Also: lisp-like functional programming, great integration with C/C++, native matrix/vector operations, very expressive, ...

Code generation — Why R?

Advantages

- I know it.
- Anything is better than `python`
- Also: lisp-like functional programming, great integration with C/C++, native matrix/vector operations, very expressive, ...

Disadvantages

- $ is used rather than dot.
- Cannot be compiled like `julia`.

CUDA

The resulting solver uses CUDA directly.

Advantages

- Imperative code rather than declarative (e.g. OpenMP)
- Much more expressive flow control (if/loop) compared to other SIMD frameworks
- Shifts the responsibility for the "parallelism" onto the programmer.
- No dependence on third-party library (e.g. Kokkos) implementing needed features

Disadvantages

- Frequently hard to debug
- Tying the code to vendor-specific syntax and runtime.
- Memory management can be frequently cumbersome.

(cross-)compilaton for CPU

We say "cross-" because virtually the same code is compiled (including memory transfers), but all CUDA calls are mapped to corresponding "standard" operations.

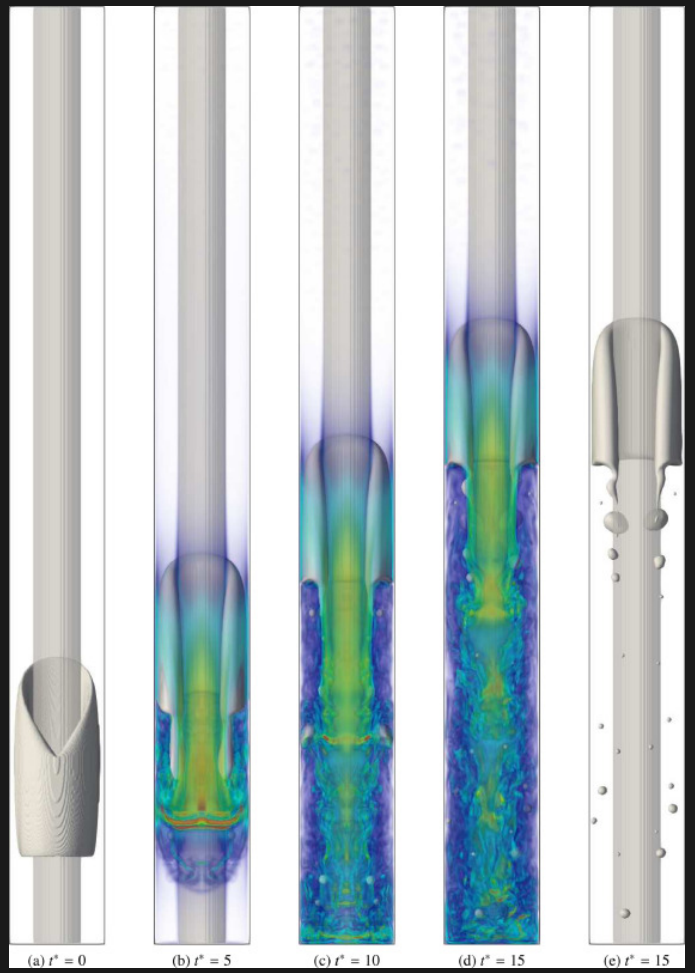
cross.h

```
#if CUDA
  #define CudaKernelRun(a__,b__,c__,d__) a__<<<b__,c__>>>d__;
  #define CudaMemcpy(a__,b__,c__,d__) cudaMemcpy(a__, b__, c__, d__)
#elif HIP
  #define CudaKernelRun(a__,b__,c__,d__) a__<<<b__,c__>>>d__;
  #define CudaMemcpy(a__,b__,c__,d__) hipMemcpy(a__, b__, c__, d__)
#elif CPU
  #define CudaKernelRun(a__,b__,c__,d__) /* A literal for loop */
  #define CudaMemcpy(a__,b__,c__,d__) memcpy(a__, b__, c__)
#endif
```

Capabilities

Simulation of complex fluid dynamics problems (mostly mesoscopic). Models written for single- and multi-phase flows, thermal flows, phase-change, dissolution, and even turbulence modelling.

Additionally the solver can be coupled with an Discrete Element Method (DEM) solver for particle laden flows. It works with ESYS-Particles, LAMMPS and LIGGGHTS.

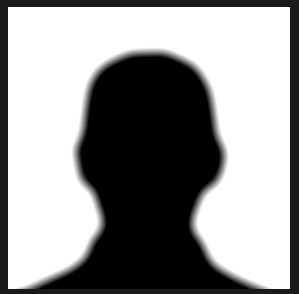


The Talk

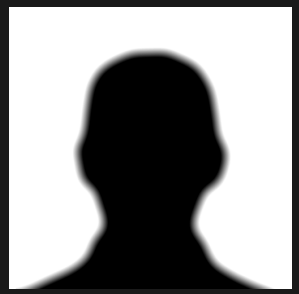
People (currently) involved

C Leonardi

D Sashko



B Hill



Ian Lenane

L Laniewski-Wollk

T Mitchell

N Di Vaira

UQ

PW

M Rutkowski

G Gruszczynski

UW

M Dzikowski

Website:
`tclb.io`

Development / Issues / Webpage / Documentation

The project is maintained on GitHub.
We use GitHub for:

- code storage and user download
- storage of documentation, tutorials and website
- storage of tools
- continuous integration
- issue/bug tracking
- code contribution (pull requests)

CFD-GO/TCLB

Install

```
git clone https://github.com/CFD-GO/TCLB.git
cd TCLB
tools/install.sh rdep [...]
```

Configure

```
make configure
./configure
```

Compile

```
make d2q9
```

Run

```
CLB/d2q9/main example/flow/2d/karman.xml
```

CFD-GO/TCLB_cluster

Install

```
git clone https://github.com/CFD-GO/TCLB.git
cd TCLB
git clone https://github.com/CFD-GO/TCLB_cluster.git p
```

Configure

```
p/config
```

Compile

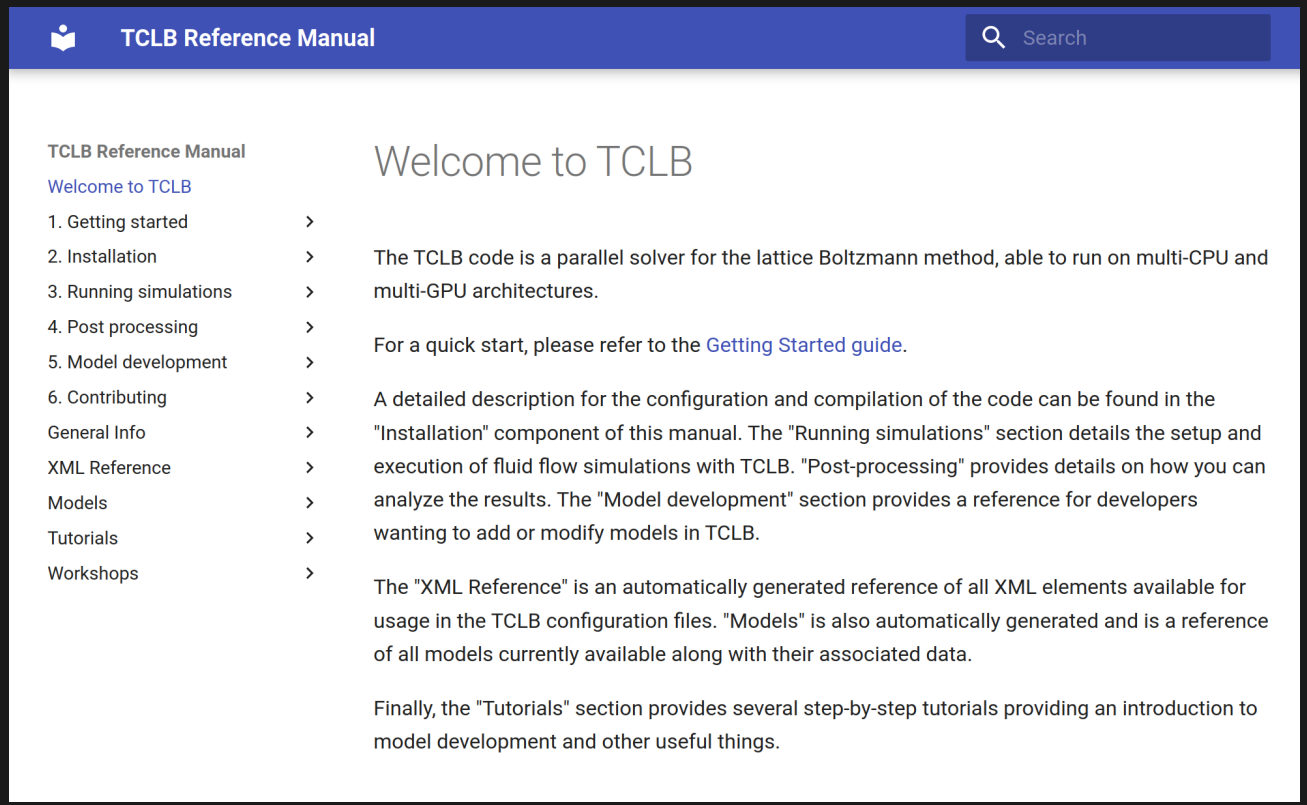
```
p/make d2q9
```

Run

```
p/run d2q9 example/flow/2d/karman.xml
```

CFD-GO/TCLB_docs

This repository stores the documentation and tutorials.
We use `mkdocs` for the documentation and generate and deploy the online documentation using `https://www.netlify.com/`



CFD-GO/TCLB_tools

Wild things live here[1].

[1] ugly python scripts

CFD-GO/TCLB_tests

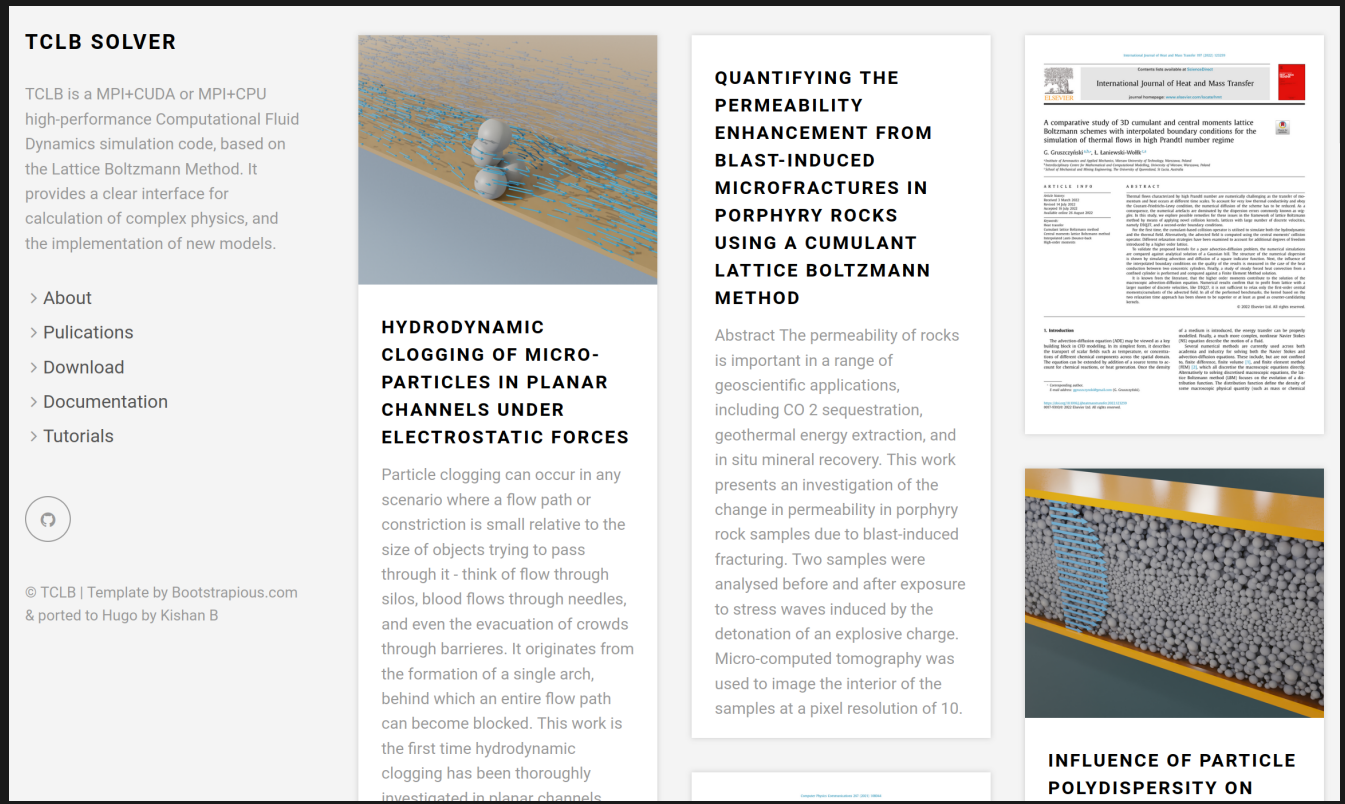
We test TCLB by executing small cases with limited number of iterations and compare to known results. This repository stores the (potentially large) result files.

Run tests

```
5  ▸ Run tools/tests.sh d2q9
8
9  Running fails test...
10 [      ] running solver
11 [  OK  ] running solver                                          0.38s
12
13 [  OK  ] fails test finished                                     -----
14
15 Running karman test...
16 [      ] copy karman.xml
17 [  OK  ] copy karman.xml
18 [      ] running solver
19 [  OK  ] running solver                                          8.40s
20 [      ] checking output/karman_Log_P00_00000000.csv (csvdiff)
21 [  OK  ] checking output/karman_Log_P00_00000000.csv (csvdiff)   0.23s
22 [      ] checking output/karman_VTK_P00_00001000.pvti (pvtidiff)
23 [  OK  ] checking output/karman_VTK_P00_00001000.pvti (pvtidiff) 0.03s
24 [      ] running solver
25 [  OK  ] running solver                                          4.57s
26 [      ] checking output/karman_Log_P00_00000000.csv (csvdiff)
27 [  OK  ] checking output/karman_Log_P00_00000000.csv (csvdiff)   0.23s
28 [      ] checking output/karman_VTK_P00_00001000.pvti (pvtidiff)
29 [  OK  ] checking output/karman_VTK_P00_00001000.pvti (pvtidiff) 0.03s
30
31 [  OK  ] karman test finished                                    -----
32
```

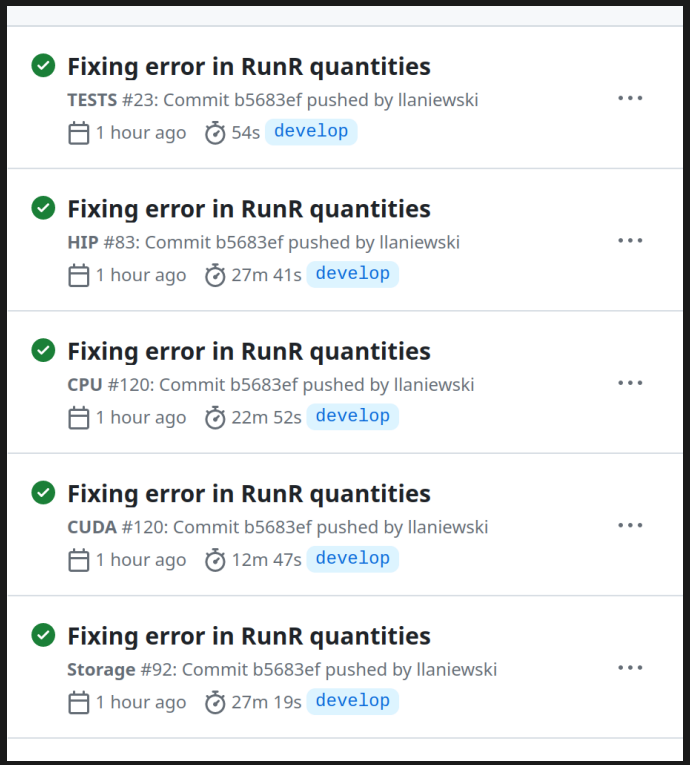
CFD-GO/sites

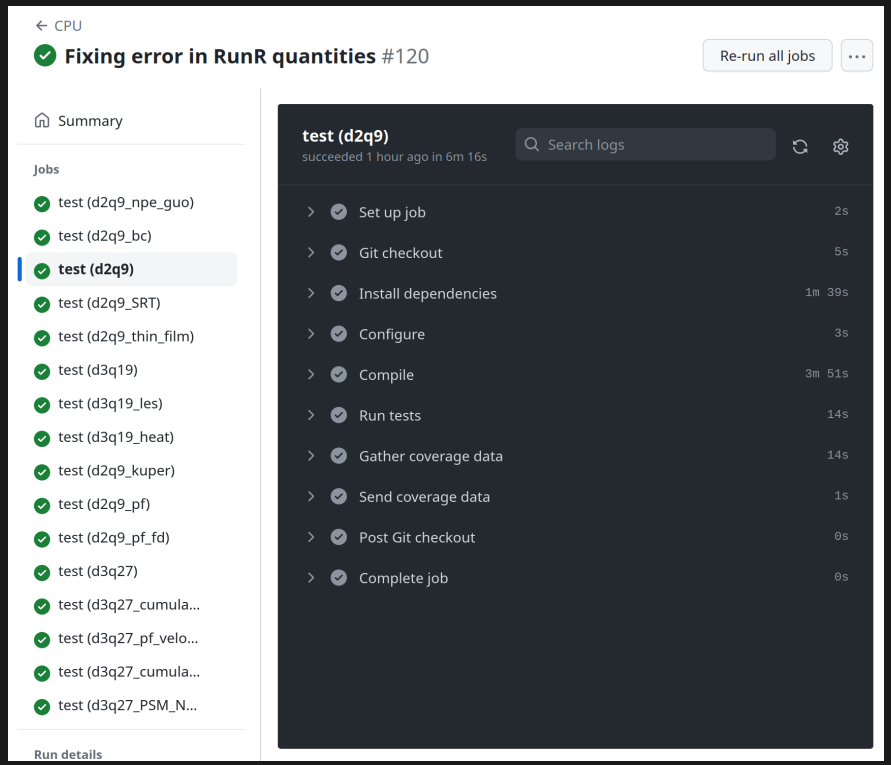
This repository is for TCLB and group sites. We use `hugo` for generation of static webpage. We generate and deploy the sites using `https://www.netlify.com/`



GitHub Actions

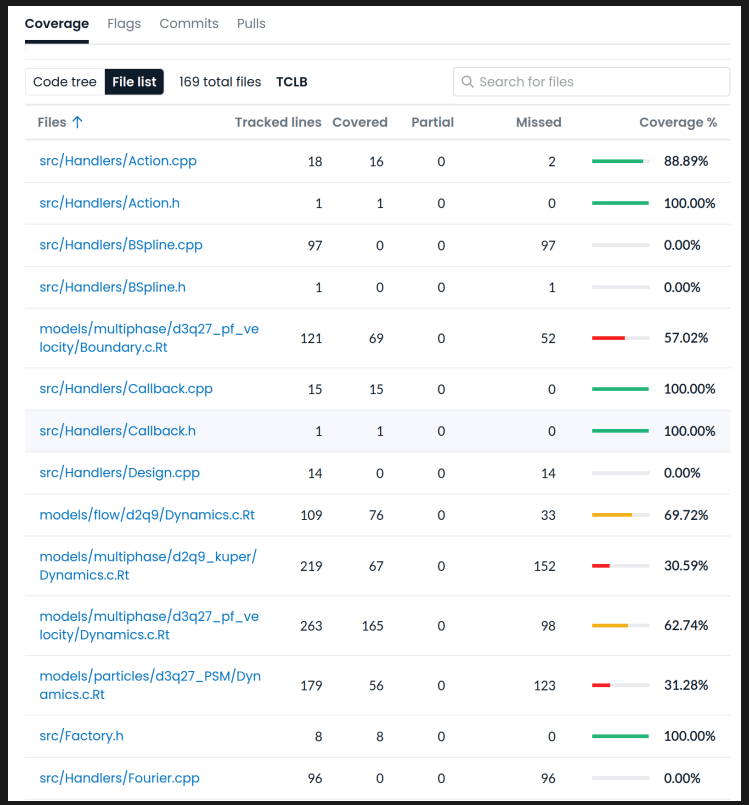
We use GitHub Actions for continious integration (CI). Compilation is tested for CUDA, ROCm and CPU. The code is executed on CPU and results are compared with `TCLB_tests` if available.





GitHub Actions — codecov

For the test runs, coverage data is gathered with `gcov` and sent to `https://codecov.io/`



Coverage Flags Commits Pulls

Code tree | File list | 169 total files | TCLB | Search for files

Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
src/Handlers/Action.cpp	18	16	0	2	88.89%
src/Handlers/Action.h	1	1	0	0	100.00%
src/Handlers/BSpline.cpp	97	0	0	97	0.00%
src/Handlers/BSpline.h	1	0	0	1	0.00%
models/multiphase/d3q27_pf_velocity/Boundary.c.Rt	121	69	0	52	57.02%
src/Handlers/Callback.cpp	15	15	0	0	100.00%
src/Handlers/Callback.h	1	1	0	0	100.00%
src/Handlers/Design.cpp	14	0	0	14	0.00%
models/flow/d2q9/Dynamics.c.Rt	109	76	0	33	69.72%
models/multiphase/d2q9_kuper/Dynamics.c.Rt	219	67	0	152	30.59%
models/multiphase/d3q27_pf_velocity/Dynamics.c.Rt	263	165	0	98	62.74%
models/particles/d3q27_PSM/Dynamics.c.Rt	179	56	0	123	31.28%
src/Factory.h	8	8	0	0	100.00%
src/Handlers/Fourier.cpp	96	0	0	96	0.00%

The Talk

R template

The package `rtemplate` was designed to generate text files with R. The syntax is similar to PHP. The package first transforms a `.Rt` file into pure R code, and then executes it.

For example:

```
<?R for (var in c("a","b","c")) { ?>
double <?%s var ?> = 0.0;
<?R } ?>
```

will generate:

```
double a = 0.0;
double b = 0.0;
double c = 0.0;
```

R template

The tool was originally made to help with generating configuration files in elaborate optimisation and exploration setups. And still can be very useful for that.

For example:

```
<?R for (d in c(1,2,4,8)) {
 sink(paste0("case_",d,".xml")) ?>
<CLBConfig output="output/">
 <Geometry nx="<?%d 20*d ?>" ny="<?%d 5*d ?>">
  <MRT><Box/></MRT>
  <Wall>
   <Box dx="<?%d 2*d ?>" nx="<?%d d ?>" ny="<?%d d ?>"/>
  </Wall>
  ...
</CLBConfig>
<?R sink(); } ?>
```

will generate four files for different grid resolutions.

R template

One can use it from command line:

```
tools/RT -f file.cpp.Rt -o file.cpp
```

... and makefiles:

```
main: main.o lib.o
	$(CXX) $(CXXFLAGS) $^ -o $@
%.o: %.cpp
	$(CXX) $(CXXFLAGS) -c $< -o $@
%: %.Rt
	$(RT) $(RTFLAGS) -f $< -o $@
```

polyAlgebra

A niche package to manipulate multivariate polynomials and output their efficient C representations.

For example:

```
x = PV("x",1:3)
y = PV("y",1:3)
g = PV("grav")
M = matrix(c(3,0,0,1,2,0,0,2,2),3,3)
C(y, g * M %*% x)
```

will generate:

```
y1 = ( x2 + x1*3. )*grav;
y2 = ( x3 + x2 )*grav*2.;
y3 = x3*grav*2.;
```