An introduction to lbxflow: an open-source, computational fluid dynamics solver using the Lattice Boltzmann method

Matthew Grasinger

October 23, 2016

Preface

If you haven't already done so, follow the instructions for installation at github.com/grasingerm/lbxflow

Table of Contents

Physical problem

Poiseuille flow

- Flow through a two-dimensional channel
- Flow driven by a constant pressure gradient
- No-slip boundary conditions at walls

Physical problem

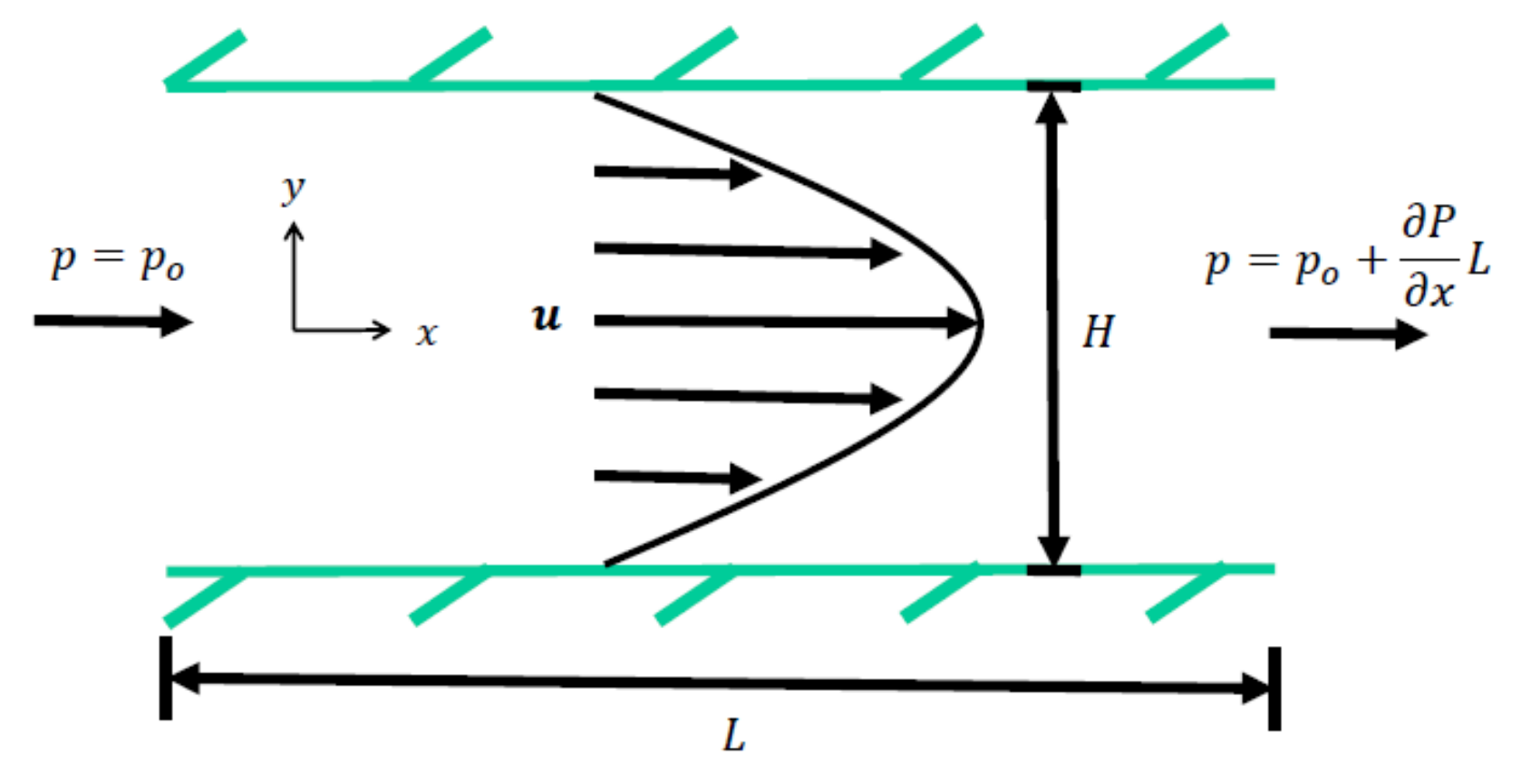

Figure 1: Schematic of Poiseuille flow; no-slip boundary conditions are enforced at the top and bottom boundaries, and periodic boundary conditions are enforced at the left and right boundaries.

Input file

To simulate a physical problem using `lbxflow`, a plain-text, input file must be created. The following slides go through creating an input file for a Poiseuille flow problem. Open a text editor and follow along.

Input file text will be on the left

A description of the parameters will be on the right

Input file

```
# simulation parameters
nsteps: { value: 2000, expr: false}
col_f: >
       BGK_F(init_constit_srt_const(1/6),
             init_sukop_Fk([1e-3; 0.0]))
```

- Time steps: 2000
- Pressure gradient: $\nabla p = -10^{-3}$
- Collision operator: BGK

Input file

```
# simulation parameters
nsteps: { value: 2000, expr: false}
col_f: >
       BGK_F(init_constit_srt_const(1/6),
             init_sukop_Fk([1e-3; 0.0]))
```

```
# lattice specifications
ni:     { value: 20,   expr: false}
nj:     { value: 12,   expr: false}
dx:     { value: 1.0,  expr: false}
dt:     { value: 1.0,  expr: false}
```

- Time steps: 2000
- Pressure gradient: $\nabla p = -10^{-3}$
- Collision operator: BGK
- Domain: $L = 20, H = 12$

Input file

```
# simulation parameters
nsteps: { value: 2000, expr: false}
col_f: >
       BGK_F(init_constit_srt_const(1/6),
             init_sukop_Fk([1e-3; 0.0]))
```

```
# lattice specifications
ni:     { value: 20,   expr: false}
nj:     { value: 12,   expr: false}
dx:     { value: 1.0,  expr: false}
dt:     { value: 1.0,  expr: false}
```

```
# material properties
rho_0:  { value: 1.0,  expr: false}
nu:     { value: 1/6,  expr: true}
```

- Time steps: 2000
- Pressure gradient: $\nabla p = -10^{-3}$
- Collision operator: BGK
- Domain: $L = 20, H = 12$
- Material properties
  - $\nu = 0.167$
  - $\rho = 1.0$

Input file

```
# simulation parameters
nsteps: { value: 2000, expr: false}
col_f: >
       BGK_F(init_constit_srt_const(1/6),
             init_sukop_Fk([1e-3; 0.0]))
```

```
# lattice specifications
ni:     { value: 20,   expr: false}
nj:     { value: 12,   expr: false}
dx:     { value: 1.0,  expr: false}
dt:     { value: 1.0,  expr: false}
```

```
# material properties
rho_0:  { value: 1.0,  expr: false}
nu:     { value: 1/6,  expr: true}
```

```
# boundary conditions
bcs:
  - north_bounce_back!
  - south_bounce_back!
  - periodic_east_to_west!
```

- Time steps: 2000
- Pressure gradient: $\nabla p = -10^{-3}$
- Collision operator: BGK
- Domain: $L = 20, H = 12$
- Material properties
  - $\nu = 0.167$
  - $\rho = 1.0$
- Boundary conditions
  - No-slip on north and south walls
  - Periodic east–west

Input file (Aside)

- Note: input parameters (generally) have a `value` property and an `expr` property.

Input file (Aside)

- Note: input parameters (generally) have a `value` property and an `expr` property.
- The `value` property is straight-forward. It is the value to either be stored, or parsed and evaulated.

Input file (Aside)

- Note: input parameters (generally) have a `value` property and an `expr` property.
- The `value` property is straight-forward. It is the value to either be stored, or parsed and evaulated.
- The `expr` property is a boolean (true or false) that describes whether `value` is an *expression*. If `expr` is true, then `value` is parsed and evaluated. The `expr` flag allows users to pass Julia code in as a `value`.

Input file (Aside)

- Note: input parameters (generally) have a `value` property and an `expr` property.
- The `value` property is straight-forward. It is the value to either be stored, or parsed and evaulated.
- The `expr` property is a boolean (true or false) that describes whether `value` is an *expression*. If `expr` is true, then `value` is parsed and evaluated. The `expr` flag allows users to pass Julia code in as a `value`.
- A simple example. The following line
  ```
  nu:  { value:  atan(1.0), expr:  true }
  ```
  sets the kinematic viscosity, `nu`, to $\pi/4$.

Input file

```
1   version: 1.0.0
2   # simulation parameters
3   nsteps: { value: 2000, expr: false}
4   col_f: >
5         BGK_F(init_constit_srt_const(1/6),
6               init_sukop_Fk([1e-3; 0.0]))
7   # lattice specifications
8   ni:     { value: 20,   expr: false}
9   nj:     { value: 12,   expr: false}
10  dx:     { value: 1.0,  expr: false}
11  dt:     { value: 1.0,  expr: false}
12  # material properties
13  rho_0:  { value: 1.0,  expr: false}
14  nu:     { value: 1/6,  expr: true}
15  # boundary conditions
16  bcs:
17    - north_bounce_back!
18    - south_bounce_back!
19    - periodic_east_to_west!
```

To run

1. save the input file in a text file called `first-example.yaml` in the `lbxflow` root directory.
2. run `julia lbxflow -f first-example.yaml` (in the `lbxflow` directory) from a terminal.
3. after a short pause, the simulation will complete and `lbxflow` will terminate.

Callback functions

- Input file on the previous slide will simulate the problem, but will not create any interesting output.

Callback functions

- Input file on the previous slide will simulate the problem, but will not create any interesting output.
- Need to also tell `lbxflow` what and how to process and post-process.

Callback functions

- Input file on the previous slide will simulate the problem, but will not create any interesting output.
- Need to also tell `lbxflow` what and how to process and post-process.
- Introduce *callback functions*
  - Functions of the form `f(sim, t)` where sim represents a `Simulation` object and `t` represents the current simulation time.
  - Example:

```
(sim, t) -> if (t % 100 == 0)
  println("Current time: ", t);
end
```

  - Above example will print simulation time every 100 time steps.

Callback functions

Add the following to the input file started on slide 9; it creates a comma delimited file where each subsequent row is the flow velocity profile at $x = 10$ after 100 time steps:

```
# Callback Functions:
# * callback functions are written in the julia language
# * sim.msm.u is a 3D array of flow velocity values
# * sim.msm.u[c, i, j] is the 'c' component of velocity at the 'i'th node in the
#     x-direction and 'j'th node in the y-direction.
callbacks:
  - >
    (sim, t) -> begin
      if t % 100 == 0
        open(f -> write(f, join(vec(sim.msm.u[1, 10, :]), ',') * "\n"),
             "u_profiles.txt", "a");
      end
    end
```

Callback functions

- Callback functions can sometimes be verbose
- Many are provided or created by other functions

Table 1: Some example callback functions

Callback function	Description
`print_step_callback(stepout, name)`	Print out current time step every `stepout` steps
`pyplot_callback(stepout, accessor)`	Create plot from vectors returned from `accessor` every `stepout` steps
`pycontour_callback(stepout, accessor)`	Create contour plot from matrix returned from `accessor` every `stepout` steps
`write_jld_file_callback(datadir, stepout)`	Create a backup file of the simulation in the directory `datadir` every `stepout` steps

Callback functions

Change the callback list of the input file started on slide 9 so that it matches the following:

```
callbacks:
  # plot the velocity profile at x = 10, y = 1:12
  - pyplot_callback(25, vel_prof_acsr(1, 10, 1:12); showfig=true, grid=true)
  - print_step_callback(50, "first-example")
```

This would visualize the simulation results, however a module needs to be loaded in order to use the visualization functions. To do this, we will introduce the `preamble` section.

Preamble

- A preamble section of the input file can be provided.
- Any `julia` code in the preamble is executed before the rest of the input file is parsed and evaluated.
- Useful for importing packages and defining variables.

```
preamble: >
          import PyPlot;
          const ni = 20;
          const nj = 12;
```

...

```
ni:       { value: ni, expr: true }
nj:       { value: nj, expr: true }
```

...

```
callbacks:
  - >
    pyplot_callback(25, vel_prof_acsr(1, convert(Int, ni/2), 1:nj);
                    showfig=true, grid=true)
  - print_step_callback(50, "first-example")
```

A first example–concluding remarks

- Note: the `preamble` section in the previous slide allowed us to keep the input file DRY. We defined variables `ni` and `nj` to describe the number of nodes in the lattice and then used them throughout the input file.

A first example–concluding remarks

- Note: the `preamble` section in the previous slide allowed us to keep the input file DRY. We defined variables `ni` and `nj` to describe the number of nodes in the lattice and then used them throughout the input file.
- For more examples similar to this one, see the `example_sims/poiseuille` directory.
  - `julia lbxflow.jl -f example_sims/poiseuille/poiseuille_velocity-profile.yaml`
  - `julia lbxflow.jl -f example_sims/poiseuille/poiseuille_pressure-contours.yaml`

A first example–concluding remarks

- Note: the `preamble` section in the previous slide allowed us to keep the input file DRY. We defined variables `ni` and `nj` to describe the number of nodes in the lattice and then used them throughout the input file.
- For more examples similar to this one, see the `example_sims/poiseuille` directory.
  - `julia lbxflow.jl -f example_sims/poiseuille/poiseuille_velocity-profile.yaml`
  - `julia lbxflow.jl -f example_sims/poiseuille/poiseuille_pressure-contours.yaml`
- For other examples, explore subdirectories in the `example_sims`.

Table of Contents

Input file parameters

What follows is a list of input file parameters accompanied by a brief description for each. Then more detailed information about boundary conditions and collision operators is covered.

Input file parameters

Table 2: Brief description of input file parameters

Parameter name	Description
`version`	Version of `lbxflow` required to run the input file
`preamble`	(Optional) Julia source code to be evaluated and parsed in the global scope. Particularly useful for importing modules, defining variables, and defining functions. Evaluated as an expression by default.
`datadir`	Directory to store simulation data that, if does not exist, is created. If run with the `-R` flag, `lbxflow` checks this directory for a backup file.
`rho_0`	Reference density. Lattice is initialized with this density.
`nu`	Reference kinematic viscosity. For non-Newtonian constitutive relationships this is the initial guess for effective viscosity.
`dx`	(Optional) Distance between lattice sites. **Note:** this parameter is not yet used for anything. It is 1.0 by default
`dt`	(Optional) Time step size. **Note:** this parameter is not yet used for anything. It is 1.0 by default
`ni`	Number of nodes in the x-direction (horizontal).
`nj`	Number of nodes in the y-direction (vertical).

Input file parameters

Parameter name	Description
`simtype`	(Optional) Simulation type. `free_surface` is for a free surface flow simulation.
`nsteps`	Number of time steps to be simulated.
`col_f`	Collision operator function. Evaluated as an expression by default.
`bcs`	List of boundary conditions. These are evaluated as expressions by default. Boundary conditions can be found in `inc/boundary.jl`
`callbacks`	(Optional) List of processing and post-processing functions.Must be of the interface `(AbstractSim, Real) -> Void`. These are evaluated as expressions by default. Functions for creating callback functions are available in `inc/io/readwrite.jl` and `inc/io/animate.jl`
`test_for_term`	(Optional) A function that takes two `MultiscaleMaps` as parameters and, if returns true, ends the simulation. Typically tests to see if flow has reached steady state. Predefined functions can be found in `inc/convergence.jl`.
`finally`	List of functions to be executed when simulation has ended. Typically processing and post-processing functions. Of interface `(AbstractSim) -> Void`

Input file parameters

Different simulation types, such as `free_surface` simulations, require additional parameters. Check `example_sims/free_surface` for examples.

Table of Contents

Boundary conditions

Boundary condition	Description
`north_bounce_back!`	No slip boundary condition where flow is assumed to be south of the boundary.
`south_bounce_back!`	No slip boundary condition where flow is assumed to be north of the boundary.
`east_bounce_back!`	No slip boundary condition where flow is assumed to be west of the boundary.
`west_bounce_back!`	No slip boundary condition where flow is assumed to be east of the boundary.
`north_reflect!`	Pure slip boundary condition where flow is assumed to be south of the boundary.
`south_reflect!`	Pure slip boundary condition where flow is assumed to be north of the boundary.
`east_reflect!`	Pure slip boundary condition where flow is assumed to be west of the boundary.
`west_reflect!`	Pure slip boundary condition where flow is assumed to be east of the boundary.

Boundary conditions

Boundary condition	Description
`periodic_east_to_west!`	Periodic boundary conditions between the east and west parts of the domain.
`periodic_north_to_south!`	Periodic boundary conditions between the north and south parts of the domain.
`north_velocity!`	Velocity inlet or outlet where flow is occuring south of the boundary.
`south_velocity!`	Velocity inlet or outlet where flow is occuring north of the boundary.
`east_velocity!`	Velocity inlet or outlet where flow is occuring west of the boundary.
`west_velocity!`	Velocity inlet or outlet where flow is occuring east of the boundary.
`north_pressure!`	Pressure inlet or outlet where flow is occuring south of the boundary.
`south_pressure!`	Pressure inlet or outlet where flow is occuring north of the boundary.
`east_pressure!`	Pressure inlet or outlet where flow is occuring west of the boundary.
`west_pressure!`	Pressure inlet or outlet where flow is occuring east of the boundary.

Boundary conditions

Boundary condition	Description
`north_open!`	Pressure gradient normal to boundary is set to zero. Flow is assumed to be occuring south of boundary.
`south_open!`	Pressure gradient normal to boundary is set to zero. Flow is assumed to be occuring north of boundary.
`east_open!`	Pressure gradient normal to boundary is set to zero. Flow is assumed to be occuring west of boundary.
`west_open!`	Pressure gradient normal to boundary is set to zero. Flow is assumed to be occuring east of boundary.

Table of Contents

Collision operators

- Collision operators control much of the physics of interest (constitutive relationship, external forcing, etc.)

Collision operators

- Collision operators control much of the physics of interest (constitutive relationship, external forcing, etc.)
- The BGK collision operator is more computationally efficient than the MRT collision operator. However, the MRT collision operator is more numerically stable, and as a consequence, is usually more accurate.

Collision operators

- Collision operators control much of the physics of interest (constitutive relationship, external forcing, etc.)
- The BGK collision operator is more computationally efficient than the MRT collision operator. However, the MRT collision operator is more numerically stable, and as a consequence, is usually more accurate.
- Both the BGK and MRT collision operators can have artificial dissipation added. Artificial dissipation is analogous to flux limiters in FEM and FVM. Artificial dissipation is generally used to enhance numerical stability. Collision functions with artificial dissipation can be found in `inc/col/filtering.jl`.

BGK collision operator

inc/col/bgk.jl

```
#! Bhatnagar-Gross-Krook collision operator
type BGK <: ColFunction
  feq_f::LBXFunction;
  constit_relation_f::LBXFunction;

  BGK(constit_relation_f::LBXFunction; feq_f::LBXFunction=feq_incomp) = (
                                                  new(feq_f, constit_relation_f));
end

#! Bhatnagar-Gross-Krook collision operator with forcing
type BGK_F <: ColFunction
  feq_f::LBXFunction;
  constit_relation_f::LBXFunction;
  forcing_f::Force;

  BGK_F(constit_relation_f::LBXFunction,
        forcing_f::Force;
        feq_f::LBXFunction=feq_incomp) = (
                                      new(feq_f, constit_relation_f, forcing_f));
end
```

BGK collision operator

- `constit_relation_f` - Constitutive relationship function. These can be found in `inc/col/constitutive.jl`. Note: must be an srt constitutive relationship.
- `forcing_f` - Forcing function. These can be found in `inc/col/forcing.jl`.
- `feq_f` - Quasiequilibrium distribution function. This is `feq_incomp`, or the quasiequilibrium distribution function for incompressible flow, by default.

Example: `col_f:`

```
BGK(init_constit_srt_const(0.2))
```

MRT collision operator

inc/col/mrt.jl

```
#! Multiple relaxation time collision operator
type MRT <: ColFunction
  feq_f::LBXFunction;
  constit_relation_f::LBXFunction;
  M::Matrix{Float64};
  iM::Matrix{Float64};
  S::LBXFunction;

  function MRT(constit_relation_f::LBXFunction; feq_f::LBXFunction=feq_incomp,
               S::LBXFunction=S_fallah)
    const M   = @DEFAULT_MRT_M;
    const iM  = @DEFAULT_MRT_IM;
    return new(feq_f, constit_relation_f, M, iM, S);
  end
end

#! Multiple relaxation time collision operator with forcing
type MRT_F <: ColFunction
  feq_f::LBXFunction;
  constit_relation_f::LBXFunction;
  forcing_f::Force;
  M::Matrix{Float64};
  iM::Matrix{Float64};
  S::LBXFunction;

  function MRT_F(constit_relation_f::LBXFunction,
                 forcing_f::Force;
                 feq_f::LBXFunction=feq_incomp, S::LBXFunction=S_fallah)
```

MRT collision operator

- `constit_relation_f` - Constitutive relationship function. These can be found in `inc/col/constitutive.jl`. Note: must be an mrt constitutive relationship.
- `forcing_f` - Forcing function. These can be found in `inc/col/forcing.jl`.
- `feq_f` - Quasiequilibrium distribution function. This is `feq_incomp`, or the quasiequilibrium distribution function for incompressible flow, by default.
- `S` - Relaxation matrix function. These can be found in `inc/col/mrt_matrices.jl`.

Filtered collision operators

inc/col/filtering.jl

```
#! Filtered collision function with fixed DS threshold
type FltrFixedDSCol <: FltrColFunction
  feq_f::LBXFunction;
  inner_col_f!::ColFunction;
  metric::LBXFunction;
  scale::LBXFunction;
  diss!::LBXFunction;
  ds_threshold::Real;
  fltr_thrsh_warn::Real;

  function FltrFixedDSCol(inner_col_f!::ColFunction; metric::LBXFunction=__METRIC,
                          scale::LBXFunction=__SCALE, diss::LBXFunction=__DISS,
                          ds_threshold::Real=__DS,
                          fltr_thrsh_warn::Real=__FLTR_THRSH_WARN)
    new(inner_col_f!.feq_f, inner_col_f!, metric, scale, diss, ds_threshold,
        fltr_thrsh_warn);
  end
end
```

Filtered collision operator

- `inner_col_f!` - Collision function to filter.
- `metric` - Function for measuring how far a lattice site is from equilibrium.
- `scale` - Function for determining how to rescale distribution functions.
- `diss!` - Function that introduces artificial dissipation.
- `ds_threshold` - Define threshold that, if exceeded, artificial dissipation is introduced.
- `fltr_thrsh_warn` - If the percentage of lattice sites that were dissipated exceeds this value then warn the user that results may be nonphysical.

Constitutive relationship

- `init_constit_srt_const(mu)` - Newtonian relationships for BGK collision operator with dynamic viscosity `mu`.
- `init_constit_mrt_const(mu)` - Newtonian relationships for MRT collision operator with dynamic viscosity `mu`.
- Constitutive relationships for power law fluids, Bingham plastic fluids, and Herschel-Bulkley fluids. See `inc/col/constitutive.jl` for more details.

Quasiequilibrium

See `inc/col/equilibrium.jl` for more details.

External forcing

See `inc/col/forcing.jl` for more details.

Concluding remarks

Any questions, concerns, errors, bugs, etc. can be directed to grasingerm@gmail.com. Please notify the author of any spelling mistakes, grammatical errors, etc. found in this tutorial or elsewhere in the documentation by email grasingerm@gmail.com.