ArvanFlow 0.1

##### July 15 2023

# Introduction

### ArvanFlow

ArvanFlow is an 2D transient LBM-LES (D2Q9) solver written in Julia. ArvanFlow was started in March 2023, originally intended as a 2D finite difference solver integrated with a genetic algorithm code for EUCASS 2023. It can be run on NVIDIA GPUs using the CUDA.jl library. ArvanFlow is developed for speed, allowing it to run many cases rapidly (hence allowing it to be used for iterative optimization problems). As of July 2023, ArvanFlow has no GUI input and can only be modified in code. ArvanFlow can run without LES using both the multiple relaxation time model, and with LES using the single relaxation time model. ArvanFlow can calculate the turbulent intensity and turbulent kinetic energy for the whole domain. As of the time of writing this document, the drag & lift coefficients returned by ArvanFlow are inaccurate for unknown reasons.

# Setup

### Installation instructions

###### Prerequisites:

* You have Julia 1.8.5 installed in VScode.
* You know how to use the Julia Command-Line to install packages.
* You have 250 KB of space (A non-issue)

###### GPU-specific prerequisites (Highly recommended):

* You have Microsoft visual C++ installed.
* The CUDA toolkit is **\*NOT REQUIRED\***
* You have any GPU the CUDA.jl library supports.

###### Required packages:

As ArvanFlow does not yet have an initiation file, packages will need to be downloaded manually from the Julia command-line. The following packages are required:

|  |  |
| --- | --- |
| Plots | v1.38.11 (recommended) or higher |
| Statistics | - |
| ImageFiltering | v0.7.5 (recommended) or higher |
| LinearAlgebra | - |
| Printf | - |
| CUDA | v4.1.4 (recommended) or higher |
| JLD2 | v0.4.31 (recommended) or higher |

###### Running ArvanFlow:

Once all the packages are installed, open ArvanFlow.jl in VScode and run without debugging. If it runs without any errors, it is ready to be used.

#### Common installation errors:

###### Missing/Broken Packages:

Although ArvanFlow is decently robust to package updates, there is a possibility that installing the newest versions of the required packages may break the code. If this error is encountered, it is best to revert to the recommended version of the packages and attempt running ArvanFlow again.

###### CUDA errors:

Note that the CUDA package is used even if the system does not have a functioning CUDA-enabled GPU in the base version of the code. It is highly recommended to use the code on a CUDA-capable machine, as the code has only been tested thus far on CUDA-enabled machines. However, a possible fix to this error may be to turn the use\_GPU flag to false.

# Functions of ArvanFlow

#### Prerequisites:

* Solid understanding of fluid dynamics fundamentals
* Solid understanding of LBM fundamentals
* Solid understanding of the Smagorinsky LES model
* Some understanding of Julia

#### Functions

Since the code uses the GPU, a lot of the matrices cannot be iterated over. This is because scalar indexing is disallowed. Processing GPU arrays one element at a time is highly ineffective, as the loop executes on the CPU by transferring the data from the GPU one element at a time. This will cause significant performance losses. For this reason, a lot of the code is implemented with matrix operations where scalar indexing would have been a simpler solution.

Where scalar indexing cannot be avoided (i.e., when Plots.jl is used) the matrices are converted to a CPU array. If use\_GPU (the GPU flag) is true, a macro called to\_device(x) is used to make the arrays from a CPU array to a GPU array. The conversion back to a CPU array is done by calling the CUDA.jl function Array(x) directly.

##### main():

The main function of ArvanFlow handles the user input for the cases and generates data structures used in multiple functions.

###### Solver settings

The variables GUI, save, use\_GPU and use\_LES are flags to control solver behaviour.

**GUI** – this flag controls whether the GUI pops up (true) or not (false) when the code runs. As the GUI code is not speed optimized, using the GUI slows down the code significantly, and hence must only be used if the entire sequence of the case is to be viewed while the case runs.

**save** – this flag controls whether case data must be saved or not (true) or not (false).

**use\_GPU** – this flag controls whether the GPU must be used (true) or not (false). Note that a CUDA-enabled machine is required to use the GPU.

**use\_LES** – this flag controls whether the smagorinsky LES model is used (true) or not (false).

**saved\_filename** – this variable is a string which determines the first few characters of the saved files for each case when the case data is saved. Currently, the drag, lift and the discrete particle distribution function are saved.

**save\_frequency** – this variable controls how often (i.e the frequency of iterations) the files are saved.

###### Domain settings

As the current code is a single case code, the domain settings thus far are for the case of a stationary cylinder in a flowing fluid. The boundary conditions for the top and bottom surfaces are periodic boundary conditions, i.e., particles flowing into the top surface are transferred to the bottom surface, and vice-versa. The inlet is a constant velocity inlet, and the outlet is a constant pressure outlet. Note that other functions allow the boundary conditions to be changed without too much difficulty. The domain shape is always a rectangle, and the fluid inlet/outlet boundaries are only allowed to be on the sides of the rectangle. The mesh is uniform throughout the entire domain. The X axis is horizontal, the Y axis is vertical, and the Z axis is assumed positive out of the plane of the domain, consistent with the right-hand rule.

The code prints the conversion factors, calculates the Kolmogorov length scale (for use in LES debugging) and throws a warning if the turbulent fluctuations are not fully resolved. It also outputs the percentage difference between the Kolmogorov length scale and the lattice length scale. The code also throws a warning if the Reynolds number in the lattice scale and real scale don’t match.

**X\_len\_real** – this variable controls the length of the rectangular domain in the X-direction, with units in meters.

**Y\_len\_real** – same as X\_len\_real, except in the Y-direction.

**X\_lattice\_density** – this variable controls the number of lattice points per meter of length in the X direction.

**Y\_lattice\_density** – same as X\_lattice\_density, except in the Y-direction. It is highly recommended to have the X\_lattice\_density and the Y\_lattice\_density be equal (the default setting)

**dx** – the length of the space between two lattice points in the X direction.

**dy** – same as dx, in the Y direction.

**c\_t** – the time conversion factor to control the conversion of time from lattice units to real units.

**dt** – the lattice unit of time. It is highly recommended to keep this as 1 (the default setting).

**t** – time in seconds for the total length of the simulation. Note that this variable isn’t used anywhere in the current code yet, and thus is pointless to change.

**param-method** – controls how the physical parameters of the simulation are set. Setting this variable to “tau-Re” uses the user-specified lattice kinematic viscosity (tau), the Reynolds number (Re), the cylinder radius in real units and the density to set the inlet velocity. Alternatively, “physical” allows the user to set the real kinematic viscosity, the inlet velocity, and the cylinder radius in real dimensions, which the code then uses to calculate tau and the Reynolds number. It is recommended to use “tau-Re” for the best balance between stability and physical accuracy.

**X\_len –** X length in lattice units. Need not be changed manually.

**Y\_len** – same as X\_len, but in the Y direction.

**smagorinsky\_BL** – the smagorinsky constant c in the boundary layer of the flow.

**smagorinsky\_FS** – the smagorinsky constant c in the freestream. As viscous effects are stronger in the boundary layer, it is recommended to have a higher value for c in the freestream than in the boundary layer. The smagorinsky constant’s range balances physical accuracy and stability in the range of 0.1-0.2.

**filter\_width\_base** – the filter width of the smagorinsky model in lattice units. The default filter width is the lattice length (recommended) as any eddy having a width smaller than the lattice length is not fully resolved.

**c** – lattice speed in all directions. It is recommended to keep the lattice speed as one, as some functionality may break with other lattice speeds. As the speed of convergence is a function of both the lattice length and the lattice speed, the lattice speed can remain as one while the lattice length is modified appropriately for the desired time for convergence.

**nu\_real** – kinematic viscosity in real units, in the units of square meter per second.

**nu** – kinematic viscosity in lattice units.

**tau** – the lattice kinematic viscosity, which also doubles as the relaxation factor in the SRT model and the eight and ninth moment relaxation weight inverse in the MRT model.

**rho0** – density of the fluid in real units, with units of kilogram per meter square.

**Nt** – the total number of timesteps for the simulation.

**DTI** – the initial domain turbulent intensity. A depreciated variable which was used to set the degree of turbulence in the domain before the simulation started, with perlin noise.

**Vx** – the inlet velocity in the X direction.

**Vy** – the inlet velocity in the Y direction.

**simulation\_type** – sets the type for the simulation: ‘SRT’ being the single relaxation time model, and ‘MRT’ being the multiple relaxation time model.

**force\_type** – sets the model for the forcing function to be used. The force type may be “Guo” [F1], “ShanChen” [F2], or none (for no external forcing). Currently broken.

**g\_real** – force density due to gravity in real units, with units meter per second per second.

**g** – force density due to gravity in lattice units.

**c\_h** – length scale conversion factor to convert between real units and lattice units.

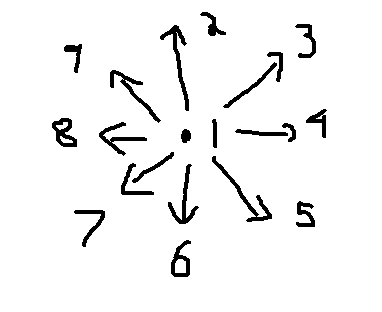
**c\_u** – velocity conversion factor to convert between real units and lattice units.

**c\_rho** – density conversion factor to convert between real units and lattice units.

**c\_f** – force conversion factor to convert between real units and lattice units.

###### Discrete velocity calculation and mesh generation

The velocity set ArvanFlow uses is D2Q9, i.e., 2 dimensional with 9 discrete velocities. The velocity set numbering is as shown:



The weights for each of the discrete velocities are [4/9, 1/9, 1/36, 1/9, 1/36, 1/9, 1/36, 1/9, 1/36]. Additionally, in this section, the S matrix for MRT is created, and the number of X and Y lattice points are calculated.

**Q** – the number of discrete velocities

**cxs** – components of the discrete velocities in the x direction

**cys** – components of the discrete velocities in the y direction

**Nx** – number of X lattice points

**Ny** – number of Y lattice points

**S** – matrix containing the MRT weights

###### MRT/SRT definitions

In this section of the code, the MRT and SRT definitions are created. For SRT, the weight matrix is not used. When LES is used, the local smagorinsky viscosity varies according to the strain rate tensor. Accordingly, the adjusted viscosity post the LES correction varies spatially as well. To account for this, the relaxation time factor for the smagorinsky-enabled cases is made into a matrix of the size of the domain. In the case of SRT flows, this is the only operation which is carried out. In the case of MRT-enabled flows, the weight matrix needs to be created. In the default case, the four weights wv , wq, we and weps are set as 1/tau, 1.2, 1.1 and 1. As scalar indexing is disallowed, the weights are set before the array is converted to the GPU array using the to\_device() macro. If LES is disabled, the relaxation time does not vary spatially and hence is a scalar value.

The moment matrix is then generated using the components of the discrete velocities. The moment matrix converts the particle distribution function from the ‘real’ particle space to the moment space. To convert the moment space distributions back into the ‘real’ space, the inverse is also calculated. By converting the distributions from the real space to the moment space, each individual moment can be assigned its own relaxation factor, hence allowing for more ‘dials’ to fine tune the stability of the case.

After the inverse of the moment matrix is calculated, the moment matrix and its inverse are converted to the device. The matrix cannot be inverted on the GPU as the inverse function involves scalar indexing. However, since the matrix and the inverse are small, and the values of the matrices are stored, this causes almost negligible performance losses.

**MomentMatrix** – the moment matrix variable

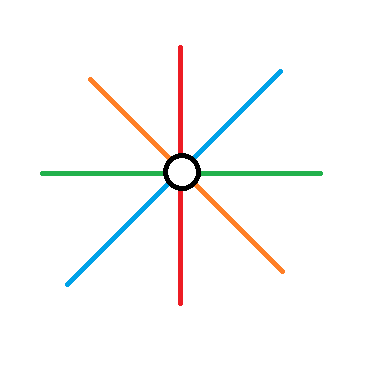
**MomentMatrixInv** – the inverse of the moment matrix variable

###### Initial Conditions

The particle distribution function is created in this section of the code, with a shape Ny:Nx:Q. The solid domain and fluid domain are tracked by two separate binary value arrays, one called SB (solid body) and the other FB (Fluid body), both having a shape of Ny:Nx. A 1 denotes that the node is inside the corresponding body, and a 0 corresponds to a node outside the body. The inner boundary of the cylinder (consisting of the nodes on the solid side) and the outer boundary of the cylinder (consisting of the nodes on the fluid side) is also created in this loop. Additionally, the boundary particle distribution (denoting the particle distribution function inside the solid), the fluid density, the fluid X and Y velocities, the force density distribution, the equilibrium particle distribution, the local forces in the X and Y directions, the lift history, the drag history, the TKE history, and the TI history are also initialized as arrays. An iteration parameter ‘iter’ denotes the number of iterations for the simulation.

The initial density is set by summing the particle distribution function parameter and dropping the third dimension. This effectively creates a 2-dimensional array of scalars corresponding to the ‘particle density’. The solid body, the fluid body, and the inner and outer boundaries are generated using the ‘cylinder’ function, which is detailed in later sections. The wall flag with the solid boundary, which denotes the walls of the domain.

The smagorinsky constant is then initialized for the entire domain with the set\_smagorinsky function. After scalar indexing is completed, the arrays are converted to GPU arrays. The boundary particle distribution is converted, and mirrored about the centre point in accordance with the following diagram:



The lattice velocities are then calculated with the velocity conversion constant. If the lattice velocity is larger than a specified constant (The default is 0.2), the simulation will be unstable due to numerical compressibility effects (The speed of sound is 1 in lattice units, and anything greater than 0.3a where a denotes the speed of sound results in compressibility corrections being required. As this error correction is for a 95% accurate physical relation, and blowup happens at a lower Mach number, i.e. any inlet velocity exceeding 0.2 for the inlet lattice velocity. Note that the physical Mach number does not correspond to the lattice Mach number). On the other hand, if the lattice velocity is too small, the simulation does not converge quickly because of boundary reflection effects. As the pressure boundary is initialized, a small pressure “shockwave” travels throughout the domain of the simulation,

# References

[F1] Guo, Z., Zheng, C. and Shi, B., 2002. Discrete lattice effects on the forcing term in the lattice Boltzmann method. Physical review E, 65(4), p.046308.

[F2] Shan, X. and Chen, H., 1993. Lattice Boltzmann model for simulating flows with multiple phases and components. Physical review E, 47(3), p.1815.