ZEIT4013 – ASSIGNMENT 3

DSMC

z5260764

QUESTION 1

Consider the Maxwell-Boltzmann distribution:

$$P(v)\,dv = 4\pi\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}} v^2 e^{-\frac{mv^2}{2kT}}\,dv$$

The derivative of the probability distribution P is found when the above equation is differentiated twice with respect to velocity v. When the derivative is equal to zero, a velocity is obtained that corresponds to the maximum or minimum probability of the distribution. The maximum is the most probable value.

$$\frac{dP}{dv} = \frac{d}{dv}\left(4\pi\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}} v^2 e^{-\frac{mv^2}{2kT}}\right)$$

$$= 4\pi\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}}\left[2v\,e^{-\frac{mv^2}{2kT}} + v^2 e^{-\frac{mv^2}{2kT}}\left(-\frac{2mv}{2kT}\right)\right]$$

$$\therefore \frac{dP}{dv} = 4\pi\left(\frac{m}{2\pi kT}\right)^{\frac{3}{2}}\left[v\cdot e^{-\frac{mv^2}{2kT}}\left(2-\frac{mv^2}{kT}\right)\right]$$

Let $dP/dv = 0$, then:

$$0 = v\cdot e^{-\frac{mv^2}{2kT}}\left(2-\frac{mv^2}{kT}\right)$$

The exponential term only approaches zero for $v \to \infty$. This solution is unphysical and can be ignored. The term can be divided out without consequence. The singular velocity term is only zero for $v = 0$. This solution is trivial which corresponds to the minimum probability. Hence for $v \neq 0$, the most probable velocity is given by:

$$0 = 2 - \frac{mv^2}{kT}$$

$$\therefore v_{mp} = \sqrt{\frac{2kT}{m}}$$

QUESTION 2

Table 1. Simplified algorithm.

1	For all simulations,
2	Initialise the simulation.
3	Index the particles into cells and create referencing structures.
4	For all timesteps,
5	Advect each particle.
6	Update the position and velocity of the particles that collide with the walls.
7	Reindex the particles into cells and recreate referencing structures.
8	For each cell,
9	Update the velocity of the particles that collide with each other.
10	For each cell,
11	Sample the velocity, temperature, and density data.
12	Save the sampling data.
13	Average the saved sampling data.
14	For each timestep,
15	Shift the data such that is it node-centred rather than cell-centred.
16	Plot a contour and save it to a movie.

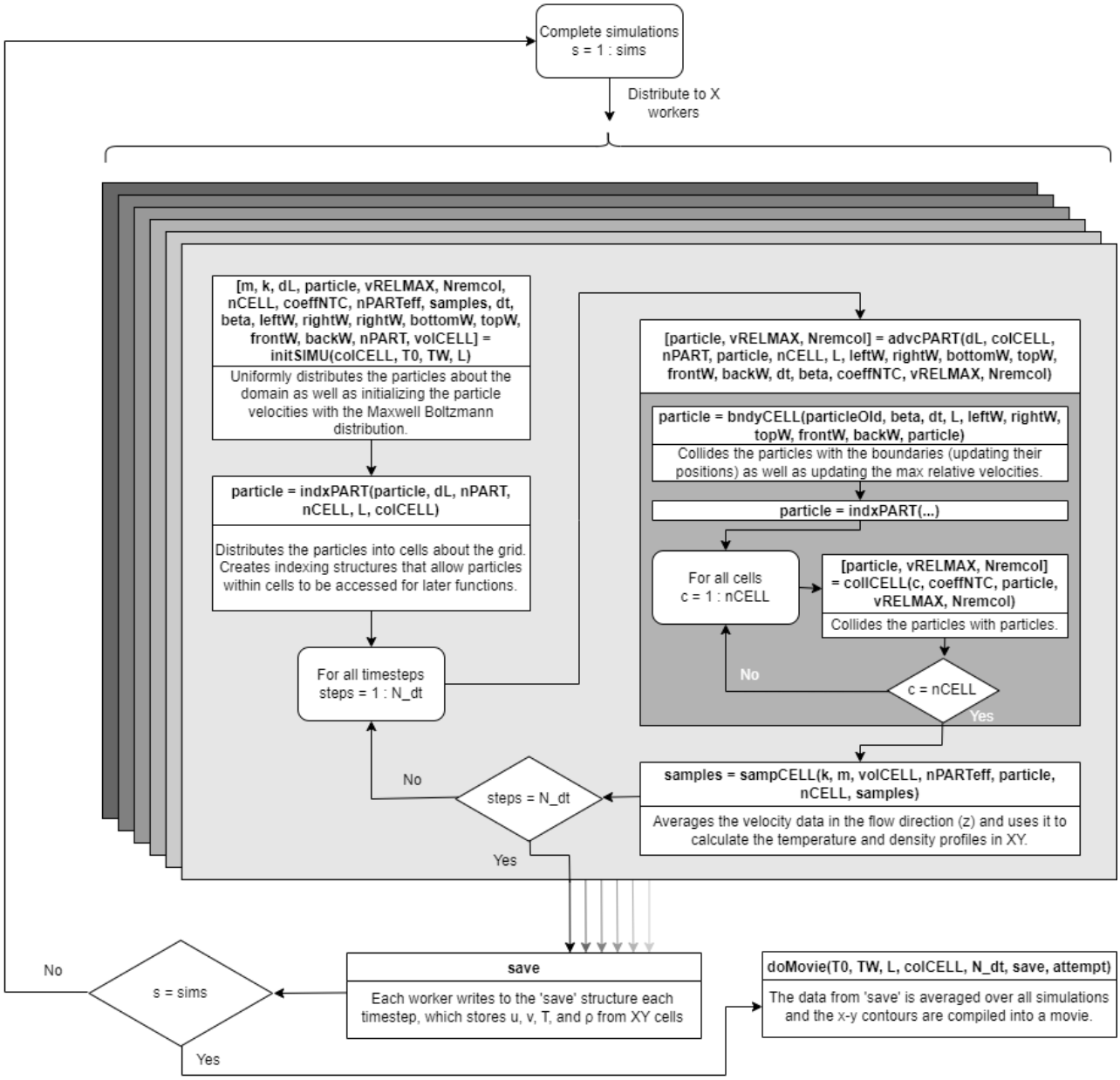

Figure 1. Flow chart illustrating the inputs and outputs of most functions and their interconnectivity.

Debug Functionality | debugMode

A primitive collision routine has been implemented in accordance with the algorithms shown in Table 1 and Fig. 1. The routine is in two MATLAB files. One is executed sequentially (mainDSMC_debug.m) and the other in parallel (mainDSMC_parallel_V2.m). The former is easier to follow and has built in debug functionality. Some functions below have a tutorial on how to take advantage of this in that file. If one wishes to run the parallel file, ensure that MATLAB's Parallel Computing Toolbox is installed on your device.

Initialisation | initSIMU

The variable declaration and simulation initialisation reside in the same function to enable parallel computing. Rather than declaring all variables outside the simulation loop once, they are declared inside so that each worker can have its own instance of the variables. The initialisation function declares all structures, including the position and velocity of the particles, the cell indexing, the remainder collisions, the wall properties, the sample structures, and the save structures. To initialise the position of the particles, a uniformly distributed random function is used across the domain. This data is stored in *particle.pos*.

To initialise the velocities in accordance with the Maxwell Boltzmann probability distribution, the following process is undertaken:

1. Estimate the maximum temperature (*TMAX*) in the problem as 30% greater than the wall temperature (*TW*). This accounts for any local spikes in temperature due to particle collisions. Since the mean free path is so large and hence there are so few particle-particle collisions, this factor likely could have been 5% and achieved the same result.
2. Create an array of velocities (*vSWEEP*) with the maximum being far larger than realistically achievable in the problem. E.g., 5 km/s.
3. Define a probability cut-off (*MBP*) to find the maximum velocity in the scenario using the velocity array, the maximum temperature, and the Maxwell Boltzmann distribution. In the code, this cut-off is set to one in a million.
4. Input the maximum temperature and velocity array into the Maxwell Boltzmann distribution and find which index the cumulative probability surpasses the compliment of the probability cut-off. This is maximum velocity (*vMAX*) in the scenario.
5. Create a velocity array (*vVEC*) whose maximum is equal to the newly calculated *vMAX*. Using this array and the initial temperature (*T0*), generate a cumulative Maxwell Boltzmann distribution (*mbPopA*).
6. For each particle, allocate a uniformly distributed random number to it between 0 and 1. Find the index in the probability distribution array for which these are equal.
7. Use this index to find the initial velocity of each particle using *vVEC* and place the velocities in another array (*vACT*). Place these into *particle.vel* later.

Another important parameter is *N_dt_tau*, the number of timesteps per mean collision time. To determine this, a mix of experimentation and theory was undertaken. The time t it takes for a particle at maximum velocity to cross the length of a cell relative to the timestep dt is chosen to be $N = 40$. Using this, *N_dt_tau* is found:

$$\frac{t}{dt} = N$$

$$dt = \frac{\lambda}{vAVE \cdot N\_dt\_tau}$$

$$t = \frac{dL}{vMAX}$$

$$\Rightarrow N_{dt_{tau}} \approx \frac{vMAX}{vAVE}\frac{\lambda \cdot N}{dL}$$

Alternatively, this relation for $vMAX$ can be used:

$$v_{MAX} = 5\sqrt{2RT_{MAX}}$$

This is $5 \cdot 10^6$ for the default case where there are 50k particles in a 1 mm^3 box.

The input and outputs to this function and others, as well as their descriptions, are shown in Tables 2 – 8.

Table 2. Inputs and outputs of initSIMU in the parallel file.

Inputs	
colCELL	Number of cells in x, y, & z.
T0	Initial temperature of gas.
TW	Wall temperature (equal for all walls).
L	Length of domain in x, y, & z.
Outputs	
m	Mass of individual particle.
k	Boltzmann's constant.
dL	Length of the cells in each axis.
particle	Struct containing the velocity, position, and cell indexing of each particle.
vRELMAX	Maximum relative velocities in each cell. Initially it is set to twice the maximum velocity in one direction from the *vACT* array.
Nremcol	Array of that stores the non-integer remaining collisions in each cell.
nCELL	Number of cells. Nominally 10 x 10 x 10.
coeffNTC	Variable for simplified computation.
nPARTeff	Effective number of particles being simulated. I.e., $n \cdot$ *volCELL / nPART*
samples	Sampling struct that stores the cell velocity, temperature, and density data.
dt	Timestep
beta	Variable for simplified computation.
leftW	Left wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
rightW	Right wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
topW	Top wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
bottomW	Bottom wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
frontW	Front wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
backW	Back wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
nPART	Number of particles. Nominally set to 50k.
volCELL	Volume of a cell, i.e., $L(1) \cdot L(2) \cdot L(3)$

Debug Functionality (Set debugMode = true)

Place break points on lines 121, 161, and 181. The first break point will illustrate the Maxwell Boltzmann distribution. The second will illustrate the particle velocities versus the Maxwell Boltzmann mean velocity and max velocity. The final break point shows all particles inside the domain. Remove these break points.

Indexing | indxPART

The purpose of the indexing function is to place the particles within cells and create a means of retrieving these particles from memory without repeating search routines. The cell location of a particle in an xyz coordinate frame is given by the ceiling of the division between the particle location and the cell length. This assumes a constant cell spacing in x, y, and z:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \text{ceiling}\left(\begin{matrix} \frac{x}{(dL)_x} \\ \frac{y}{(dL)_y} \\ \frac{z}{(dL)_z} \end{matrix} \right)$$

Using the cell index in this 3D coordinate frame, a 1D cell index i is found with the following relation which is commonly used in CFD:

$$i = X + C_x \cdot (Y - 1) + C_x \cdot C_y \cdot (Z - 1)$$

This is graphically illustrated in Fig. 2. This data is stored in *particle.cell*.

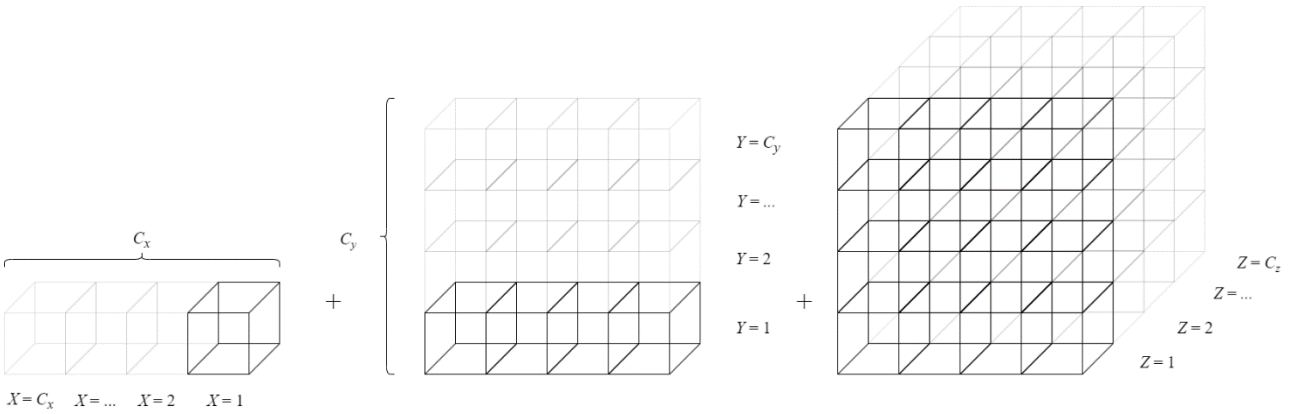

Figure 2. Cell indexing diagram (zoom in!). The dark lines highlight the smallest non-zero unit that can be added in each successive dimension, while the opaque lines highlight the largest unit that can be added.

The indexing function then cycles through all particles and counts how many are within a particular cell. The index that corresponds to the first instance of a particle within a cell is also recorded. With these two data points for every particle, then by completing one reordering routine every timestep the indices of particles within cells can be retrieved with ease. This data is stored in *particle.id*.

Table 3. Inputs and outputs of indxPART in the parallel file.

Inputs	
colCELL	Number of cells in x, y, & z.
particle	Struct containing the velocity, position, and cell indexing of each particle.
dL	Length of the cells in each axis.
nPART	Number of particles. Nominally set to 50k.
nCELL	Number of cells. Nominally 10 x 10 x 10.
L	Length of domain in x, y, & z.
Outputs	
particle	Struct containing the velocity, position, and cell indexing of each particle.

Debug Functionality

Place a break point at line 325 and continue a few times. Each cell will be filled with its particle in a color-coded fashion. Replace this break point with one at line 329 and continue. All the particles will be in their individual cells. Toggle the view for some eye candy. Remove this break point and let debugMode be false.

Advection | advcPART

The advection function translates the particles in space over a timestep using the following:

$$\vec{x}_{s+1} = \vec{x}_s + \overrightarrow{v_s} \cdot dt$$

where $\vec{x}$ and $\vec{v}$ are

$$\vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad \vec{v} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

and are contained in *particle.pos* and *particle.vel* respectively. The advection function then calls the boundary condition routine, which is accounts for the collisions between the particles and the walls. Then the particles are reindexed into their cells by calling indxPART, since their positions have been altered. Finally, for each cell, the particle-particle collision function is called that accounts for the collisions between particles.

Table 4. Inputs and outputs of advcPART in the parallel file.

Inputs	
colCELL	Number of cells in x, y, & z.
L	Length of domain in x, y, & z.
nPART	Number of particles. Nominally set to 50k.
particle	Struct containing the velocity, position, and cell indexing of each particle.
dL	Length of the cells in each axis.
nCELL	Number of cells. Nominally 10 x 10 x 10.
leftW	Left wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
rightW	Right wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
topW	Top wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
bottomW	Bottom wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
frontW	Front wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
backW	Back wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
dt	Timestep
beta	Variable for simplified computation later.
coeffNTC	Variable for simplified computation later.
vRELMAX	Maximum relative velocities in each cell.
Nremcol	Array of that stores the non-integer remaining collisions in each cell.
Outputs	
particle	Struct containing the velocity, position, and cell indexing of each particle.
vRELMAX	Maximum relative velocities in each cell.
Nremcol	Array of that stores the non-integer remaining collisions in each cell.

Boundary Conditions | bndyCELL

The boundary conditions (BC) are applied with the following steps:

1) All particles that lie to the left, right, above, below, forward, and behind of the boundaries are found. I.e., one search routine per BC function call.
2) Using the indices of the particles that escaped the domain, their positions and velocities are updated through a series of wall boundary equations. In this scenario, the left, right, top, and bottom walls are thermal walls while the front and back walls are periodic.
3) The thermal walls have the following general algorithm, whose implementation is credited to Saleen Bhattarai. For each particle that has escaped past a wall, find the axis of wall it escaped through. E.g., if it escaped through the left or right walls, the corresponding axis is x (the axis for the top wall is y and z is for the front). This axis is the one where the velocity will reverse in direction due to the reflection. In all subsequent equations, the axis is assumed to be x. The velocity in the corresponding axis will be updated with the following equation (Alexander & Garcia, 1997, p. 3).

$$u = n\beta\sqrt{-2T_W \ln r}$$

While the other velocities will be updated with the following (Alexander & Garcia, 1997, p. 3):

$$v = w = \beta r_g \sqrt{T_W}$$

where $\beta = \sqrt{k/m}$, r is a uniformly distributed random number between 0 and 1, r_g is a Gaussian-distributed random number between 0 and 1, and T_W is the temperature of that wall. The term n is equal to –1 if the particle travelled in the negative direction of the corresponding axis, while it is +1 if it travelled in the positive direction. This accounts for the direction of travel in the domain. Wall temperature is not included within β to enable variable temperatures on each wall if desired.

Once the velocities are updated, the positions of the reflected particles are altered with the following (Alexander & Garcia, 1997, p. 2):

$$x_s = x_W + u \cdot TOF$$

$$TOF = dt\left(\frac{x - x_W}{x - x_{old}}\right)$$

where x_s is the post collision position, x_W is the position of the wall, and TOF is the time of flight. The latter is the time the particle spends post the wall relative to the time it spends before the wall. This is illustrated in Fig. 3. The other positions, i.e., in the directions that the reflection didn't explicitly take place, are updated with:

$$y_s = y_W + v \cdot TOF$$

$$z_s = z_W + w \cdot TOF$$

Where the wall positions are given by:

$$y_W = y - \frac{TOF}{dt} \cdot (y - y_{old})$$

$$= y - \frac{x - x_W}{x - x_{old}}(y - y_{old})$$

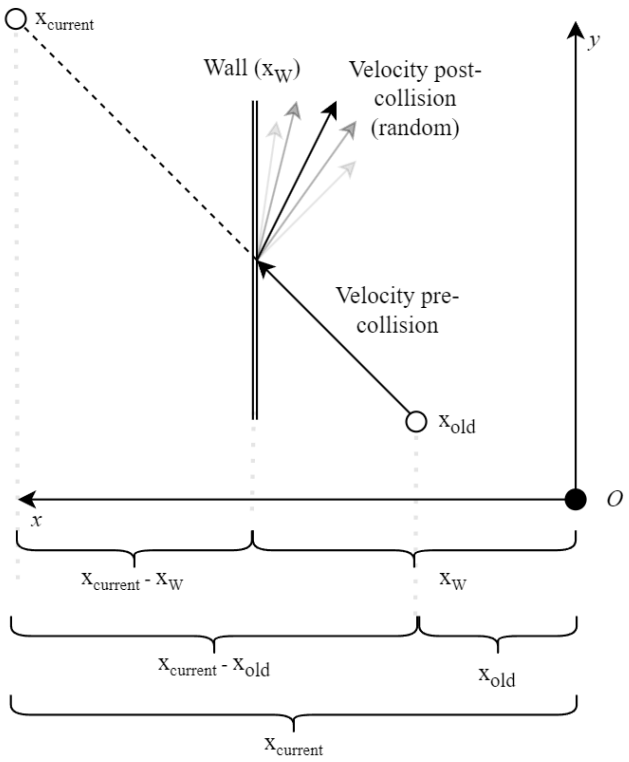

Figure 3. Thermal wall diagram example (x)

I.e., using the ratio of post wall distance to pre-wall distance, the post wall distance is found and subtracted off the current position. This is repeated for z:

$$z_W = z - \frac{x - x_W}{x - x_{old}}(z - z_{old})$$

Versions of these equations are applied to the other thermal walls. They are not detailed here.

4) The velocity of the particles who escape through periodic walls, e.g., the front and back walls, are not updated. Only their positions are updated using a mod function:

$$z_s = z \bmod L$$

Their current position is divided by the length of the domain whose remainder dictates their new position.

The boundary conditions function is called bndyCELL rather than bndyPART because of its original algorithm:

```
for z = 1 : colCELL(3)
    for y = 1 : colCELL(2)
        for x = 1 : colCELL(1)
            % 1D cell index
            c = x + colCELL(1) * (y - 1) + colCELL(1)*colCELL(2) * (z - 1);
            % check if x y z cell locations lies on an edge/boundary
            if x == 1 || x == colCELL(1) || y == 1 || y == colCELL(2)...
            || z == 1 || z == colCELL(3)
                bndyCELL(c, particleOld); % collide particles with boundary
            end
        end
    end
end
```

Since the timesteps are sized such that particles never skip cells at max velocity, then the initial route was to only call the boundary cells. I.e., when X, Y, or Z were equal to 1 or their max values C_x, C_y, C_z. If the code

was parallelised on a GPU, it is likely this method would be faster. However, on a CPU, the bndyCELL function is consequently called many times rather than once. This means the expensive transformations within the function are executed many times. Hence, even though the memory access pattern of the original method is likely more efficient, completing a single search routine within bndyCELL is faster.

After the boundary conditions are applied, the particles are reindexed into their cells with an indxPART call.

Table 5. Inputs and outputs of bndyCELL in the parallel file.

Inputs	
L	Length of domain in x, y, & z.
particleOld	Struct containing the velocity, … etc. of each particle before advection.
particle	Struct containing the velocity, position, and cell indexing of each particle.
leftW	Left wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
rightW	Right wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
topW	Top wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
bottomW	Bottom wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
frontW	Front wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
backW	Back wall struct, contains the reflective coefficient (-1 or 1) and the wall temperature.
dt	Timestep
beta	Variable for simplified computation later.
Outputs	
particle	Struct containing the velocity, position, and cell indexing of each particle.

Collisions | collCELL

This function collides candidate particles together. The algorithm is repeated for each cell as follows:

1) Use the indexing arrays *particle.id* and *particle.cell* to retrieve the particles within the current cell.
2) If there is more than one particle in the cell, retrieve the particle velocities (*vCELL*).
3) Determine the number of potential collisions within the cell during a timestep, *cols*, with the following (Alexander & Garcia, 1997, p. 4). This step uses Saleen's implementation:

$$truecols = \text{coeffNTC} \cdot N_c^2 \langle v_r \rangle + Nremcol(c)$$

$$\text{where coeffNTC} = \frac{\pi d^2 N_e dt}{2V_c}$$

$$cols = \text{floor}(truecols)$$

$$Nremcol(c) = truecols - cols$$

where $V_c = volCELL = dL(1) \cdot dL(2) \cdot dL(3)$, $N_e = nPARTeff$. The variable *coeffNTC* is for efficient computation. The variable *truecols* is the decimal number of collisions while *cols* is the floored integer number of potential collisions. The 'left over' collisions are stored in *Nremcol* and added to *truecols*. Note, since the collCELL function is based off Saleens code, $N_c^2 = partCELL \cdot (partCELL - 1)$ rather than $(partCELL)^2$. It shouldn't make a difference. I only spotted it after I collected my results.

4) For all potential collisions, randomly sample two particles using a uniform distribution. Calculate the relative velocity between these particles (Alexander & Garcia, 1997, p. 4):

$$v_r = \sqrt{\left(u_{p1} - u_{p2}\right)^2 + \left(v_{p1} - v_{p2}\right)^2 + \left(w_{p1} - w_{p2}\right)^2}$$

If this relative velocity is greater than the maximum relative velocity recorded for this cell, the maximum is updated. I.e., if the number of potential collisions is higher than the candidate collisions.

5) The two particles are only collided if their relative velocity over the maximum relative velocity in that call is greater than a uniformly distributed random number (Alexander & Garcia, 1997, p. 4). If this condition is met, then using the centre of mass velocity (Alexander & Garcia, 1997, p. 4):

$$v_{cm} = \frac{\vec{v}_{p1} + \vec{v}_{p2}}{2}$$

the particle velocities are updated with:

$$\vec{v}_{p1} = \vec{v}_{cm} + \frac{\vec{v}_{rel}}{2}$$

$$\vec{v}_{p2} = \vec{v}_{cm} - \frac{\vec{v}_{rel}}{2}$$

where the relative velocity vector is found with this sequence of calculations (Alexander & Garcia, 1997, p. 4):

$$v_{rel} = v_r \begin{bmatrix} \cos\theta \\ \sin\theta\cos\phi \\ \sin\phi\sin\theta \end{bmatrix}$$

$$\cos\theta = 2r - 1$$

$$\phi = 2\pi r$$

$$\sin\theta = \sqrt{1 - \cos^2\theta}$$

where in each instance r is a unique uniformly distributed random number between 0 and 1. Saleen's implementation is slightly different to Garcia's due to differences in axis directions. Given there are no pressure gradients in the scenario, it makes no difference to the output of the code.

Table 6. Inputs and outputs of collCELL in the parallel file.

Inputs	
c	Index of the current cell.
particle	Struct containing the velocity, position, and cell indexing of each particle.
coeffNTC	Variable for simplified computation later.
vRELMAX	Maximum relative velocities in each cell.
Nremcol	Array of that stores the non-integer remaining collisions in each cell.
Outputs	
particle	Struct containing the velocity, position, and cell indexing of each particle.
vRELMAX	Maximum relative velocities in each cell.
Nremcol	Array of that stores the non-integer remaining collisions in each cell.

Sampling | sampCELL

This function samples the data from the particles and transforms them into singular velocity, pressure, density, and temperature values for each cell. It operates in the following way:

1) Use the indexing arrays *particle.id* and *particle.cell* to retrieve the particles within the current cell.
2) Retrieve the particle velocities (*vCELL*).
3) Use the velocities in that cell to sample the properties with the following:

$$\begin{Bmatrix} \bar{\rho}(c,t) \\ \bar{\vec{v}}(c,t) \\ v_{ms}(c,t) \end{Bmatrix} = \sum_{c}^{cell} \begin{Bmatrix} \frac{mN_e}{V_c} N_c \\ \frac{\vec{v}_c}{N_c} \\ \frac{1}{N_c} \sqrt{u_i^2 + v_i^2 + w_i^2} \end{Bmatrix}$$

where v_{ms} is the means square velocity, N_c is the particles within that cell, V_c is the volume of the cell, t is the timestep, and N_e is effective number of simulated particles. The temperature is found with (Alexander & Garcia, 1997, p. 4):

$$T(c,t) = \frac{m}{3k}\left(v_{ms}(c,t) - \sum \bar{\vec{v}}(c,t)^2\right)$$

Per Saleen, the square mean velocities are subtracted from the mean square velocity to account for the bulk velocity of the particles in each cell. These properties are saved in *samples.T*, *samples.vel*, *samples.V2*, and *samples.rho*.

Table 7. Inputs and outputs of sampCELL in the parallel file.

Inputs	
m	Mass of individual particle.
k	Boltzmann's constant.
particle	Struct containing the velocity, position, and cell indexing of each particle.
samples	Sampling struct that stores the cell velocity, temperature, and density data.
nCELL	Number of cells. Nominally 10 x 10 x 10.
nPARTeff	Effective number of particles being simulated. I.e., $n \cdot$ volCELL / nPART
volCELL	Volume of a cell, i.e., $L(1) \cdot L(2) \cdot L(3)$
Outputs	
samples	Sampling struct that stores the cell velocity, temperature, and density data.

Saving Data | saveDATA

saveDATA is only a function in the debugging file. In the parallelised file, the save arrays are initialised within the main function, making them common, so that each worker can write to them. For any given simulation, a worker will average the velocity, density, and temperature values across the z cells. That is, for each (X, Y) cell index, the indices c of all z cells are calculated and their properties are averaged:

$$idx = X + C_x \cdot (Y-1) + C_x \cdot C_y \cdot (z-1)$$

$$\text{where } z = [1, 2, \dots, C_z - 1, C_z]$$

$$\begin{Bmatrix} \vec{v}_{av}(X,Y,t) \\ \vec{T}_{av}(X,Y,t) \\ \vec{\rho}_{av}(X,Y,t) \end{Bmatrix} = \frac{1}{C_z} \sum_{i}^{idx} \begin{Bmatrix} \vec{v}_i \\ T_i \\ \rho_i \end{Bmatrix}$$

Once all simulations have finished, these temporary arrays are stored in the struct *saved*.

Visualizing Data | doMovie

To create a movie, the data in the saved struct is shifted to a node centred frame rather than a cell-centred frame. This is due to MATLAB using node centred plotting. See Fig. 4 for a visualisation of the shift.

$$T_e = \frac{1}{4}(T_a + T_b + T_c + T_d)$$

Table 8. Inputs of doMovie in the parallel file.

Inputs	
colCELL	Number of cells in x, y, & z.
T0	Initial temperature of gas.
TW	Wall temperature (equal for all walls).
L	Length of domain in x, y, & z.
N_dt	Number of timesteps (constant for each simulation).
saved	The simulation and z averaged saved data in XY at each timestep
attempt	Identifier for videos and output files

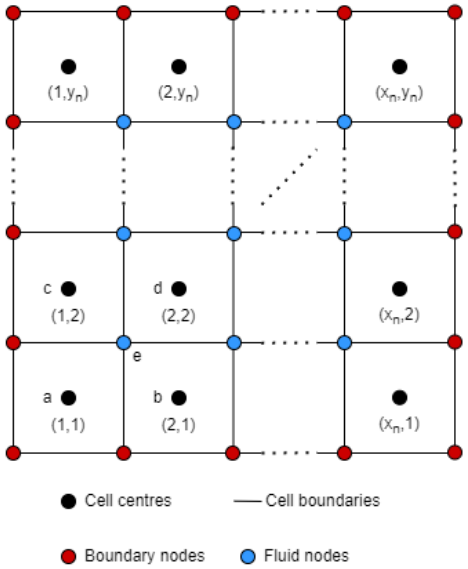

Figure 4 (right). Cell-centred vs node centred layout

Results

The mean free path is:

$$\lambda = \frac{1}{n\pi d^2\sqrt{2}}$$
$$= 1.68 \text{ m}$$

It is assumed that L = 1 mm is the characteristic length. The Knudsen number is:

$$Kn = \frac{\lambda}{L}$$
$$= 1680$$
$$\approx 10^3$$

This is a highly rarefied flow that is suited for collisionless Boltzmann models (Fig. 5).

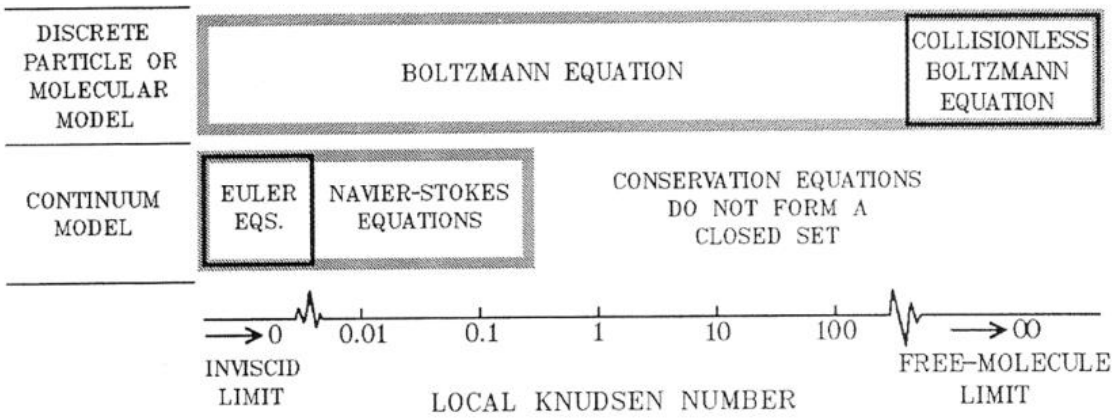

Figure 5. Recommended models based on Knudsen number (from slides).

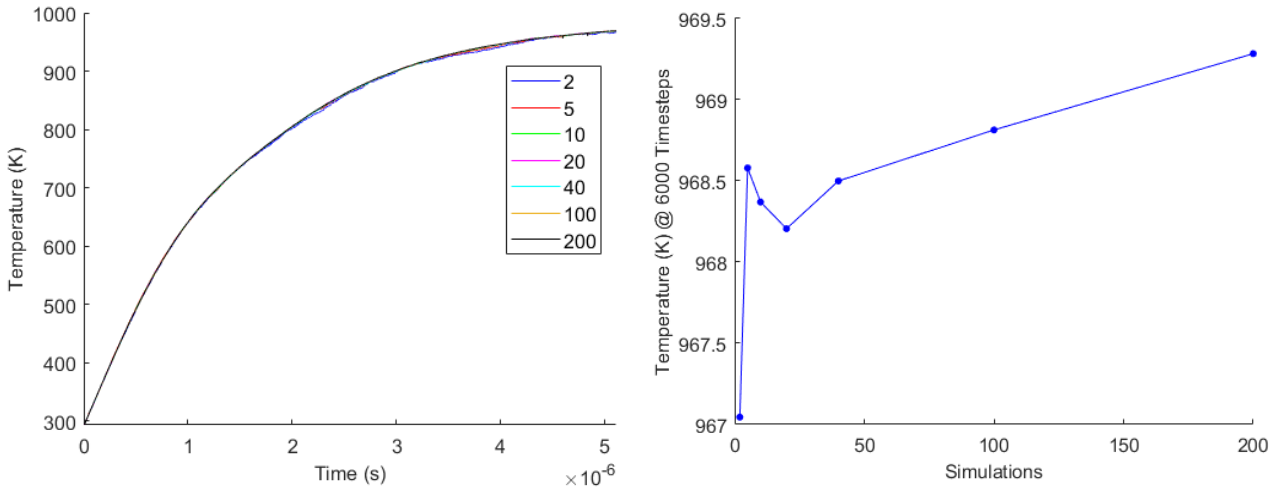

Figure 6. Left: Temperature convergence for different groups of simulations. Right: Mean temperature at 6000 timesteps versus simulation count. Figures 6 & 7 have 1e18 particles/m^3, 50k particles, *volDOMA* = 1 mm^3, 6000 steps, and *N_dt_tau* = 5e6.

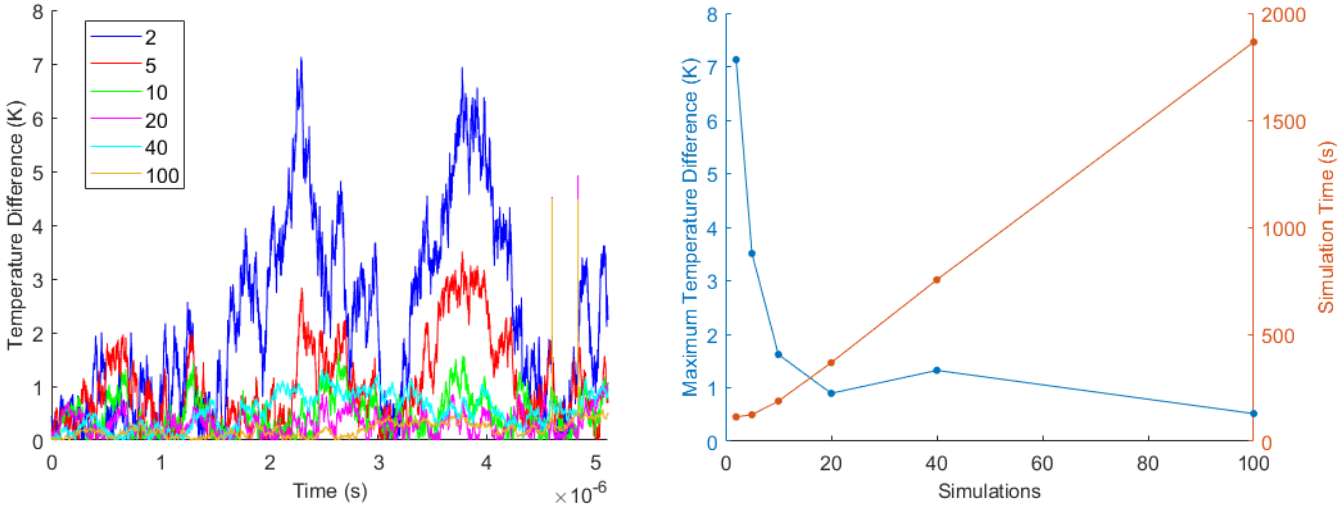

Figure 7. Left: Temperature difference to the 200x simulation for different groups of simulations. The properties are the same as in Fig. 5. Right: Maximum temperature difference and computation time for the different simulations. The properties are the same as in Fig. 6.

Figure 6 left indicates that the temperature profile does not change significantly when more simulations are conducted. This is quantified in Fig. 6 right, where the mean temperature at 6000 timesteps only varies by, at most, 2K between 2x simulations and 200x simulations. The difference is on the order of a Kelvin between 5x simulations and 200x simulations. Fig. 6 left also indicates that **it takes 4.55 μs to reach a mean temperature of 960K for 200 simulations**. Interestingly, in Fig. 6 right the difference in mean temperature temporarily lowers as the simulation count goes up, before decreasing again and converging at ~970K.

Figure 7 left shows the difference between the temperature profiles in Fig. 6 left and the 200x simulations profile over time. The maximum temperature difference for all groups of simulations, as well as the computation time, is shown in Fig. 7 right. Initially, the computation time does not vary significantly. This is because the pool of parallel workers selected was 10, meaning the computation times for 2x, 5x, and 10x simulations were equivalent. As the number of simulations increases beyond this, the computation time increase linearly. The maximum temperature difference drops quickly with simulation count and is near its minimum value at 20x simulations. For this reason, all analyses were conducted with 20x simulations regardless of the other input properties.

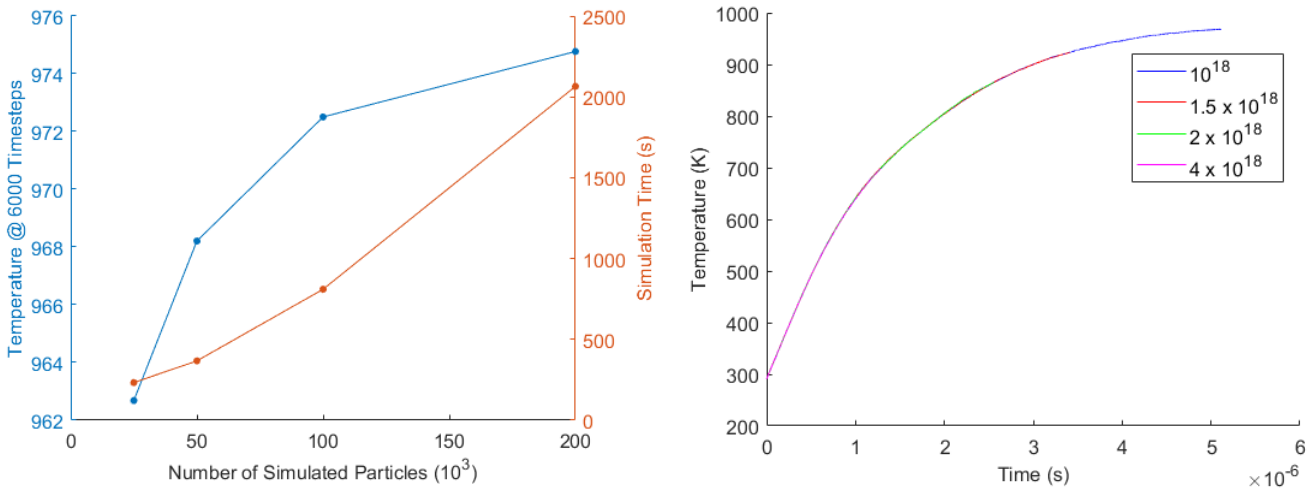

Figure 8. Left: Temperature and simulation time dependence on particle count. Right: Temperature convergence for different densities (particles/m^3).

The difference in mean temperature at 6000 timesteps between 25k and 200k particles simulated particles is ~10K (Fig. 8 left). This is small enough that 50k particles is sufficient. The computation time increases with the square of the number of particles. This plot is equivalent to increasing the ratio of effective particles.

Increasing density while holding simulated particles and cell count constant does not change the temperature convergence. This is illustrated in Fig. 8 right. This is because almost no collisions are occurring since the mean free path is much greater than the domain size. With no collisions, increasing the number of particles won't change the temperature profile. It will only change the number of particles hitting the boundaries.

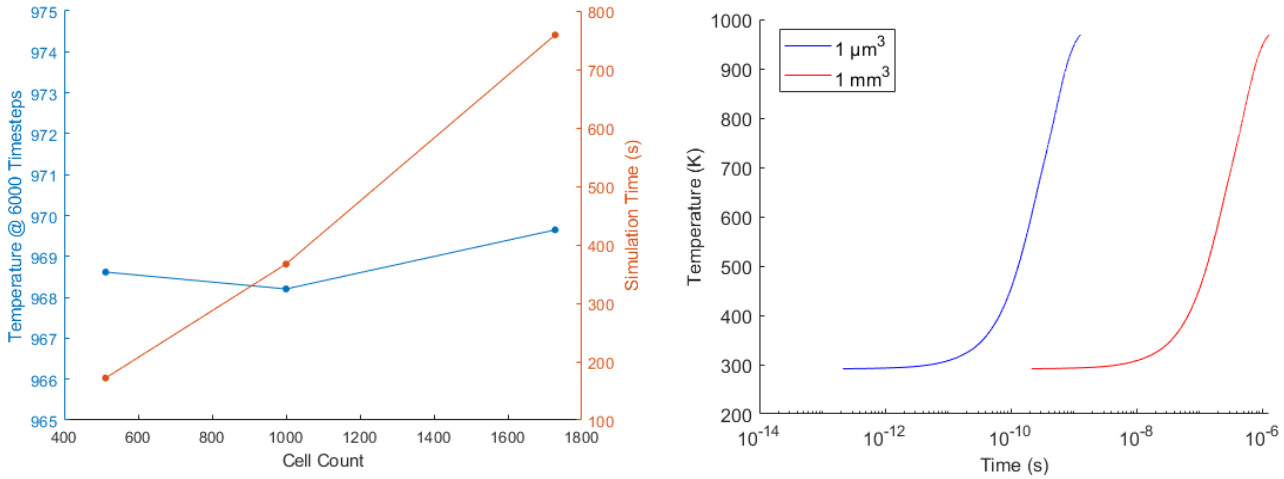

Figure 9. Left: Temperature and simulation time dependence on cell count. Right: Temperature convergence profile for different domain sizes. Properties: 1e18 particles/m^3, 50k particles, 6000 steps, *N_dt_tau* = [5e9, 5e6].

In Fig. 9 left, the particles per cell is kept constant at 50 and the cell count in each axis is held constant. The plot indicates that computation time goes up linearly with cell count and mean temperature changes negligibly.

Figure 9 right shows how the temperature profile varies with time for different sized domains. Both domains feature the same number of particles per cubic meter. The smaller domain (1 μm^3) reaches equilibrium 4 orders of magnitude quicker than the larger domain. Even though the wall area has reduced, the particles have less space to move around yet remain at the same initial temperature (and hence, velocity). Consequentially, more wall collisions are expected. This increases the temperature gain rate substantially.

The temperature contours for the default 200x simulation case are shown in Fig. 10 at various timesteps. Videos of the simulations can be accessed through this link for a limited time: DSMC z5260764

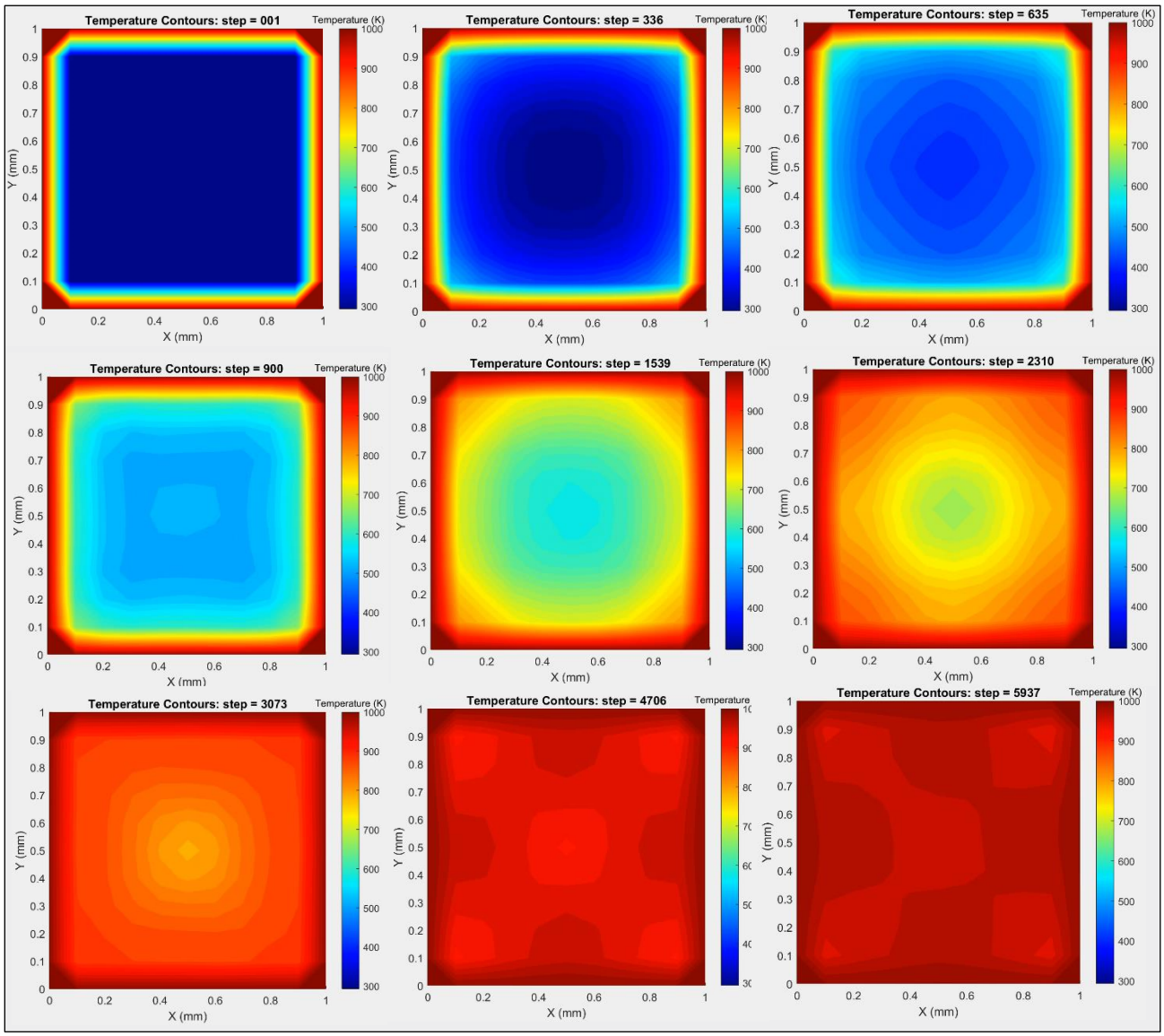

Figure 10. 1e18 particles/m^3, 50k particles, 1 mm^3, 200 simulations, 6000 steps, *N_dt_tau* = 5e6.

Improvements in Efficiency & Accuracy

To run the code in parallel, a pool of workers is allocated whose size is dependent on the user's CPU. These workers are allocated indices from the simulation loop. To improve the efficiency, two routes could be taken:

1. Within the already parallelised code, call kernels for the boundary conditions and particle collision functions. This parallelises those functions, which would reduce simulation time significantly. According to the MATLAB profiler, those functions take up the most computation time.
2. Instead of utilising CPU workers, rewrite the code to be compatible with GPU workers. E.g., a cell is allocated to a block on a streaming processer. This is analogous to modern CFD and is quite difficult.
3. If the original boundary conditions algorithm, i.e., only call boundary cells, is implemented on a GPU, the increase in efficiency could be accompanied by an increase in accuracy by using another function. Particles that hit the corners of the domain are likely to be reflected *outside* the domain. It is too inefficient with sequential computing to account for these escaped particles in a particular time step. Accounting for these particles may become affordable when run on a GPU architecture.
4. The timestep could be variable. In the sampling function, the maximum velocity could be found and dt could be adjusted based on this.

Acknowledgements

Saleen Bhattarai's lecture was instrumental in completing the assignment.

References

Alexander, F. & Garcia, A. (1997). The Direct Simulation Monte Carlo Method. *Computers in Physics*. *11*(6), 588-593. doi: 10.1063/1.168619.