# GPU accelerated direct simulation Monte Carlo (DSMC) code using CUDA

*Project report submitted in partial fulfillment of the requirement for the degree of*

**Bachelor of Technology**

by

**Vajjala Paidi Vikramaditya (180103089)**

**Surendra Kumar Kumawat (180103081)**

Under the supervision of

**Dr. Tapan K. Mankodi**


Department of Mechanical Engineering

Indian Institute of Technology, Guwahati

# Abstract

In fluid flow situations, the Navier–Stokes equations cannot adequately describe the transition and the free molecular regimes. In these regimes, the Boltzmann equation of kinetic theory is invoked to govern the flows. But this equation cannot be solved easily, neither by analytical techniques nor by numerical methods. Hence, in order to manoeuvre around this equation, Bird introduced a particle simulation technique called the 'Direct Simulation Monte Carlo' (DSMC) method. It is regarded as a numerical method for solving the Boltzmann equation. In this report, a standard Maxwell-Boltzmann distribution is defined by analogy to the concept of the standard Gaussian distribution. The most important statistical properties of Maxwell- Boltzmann distribution, as well as a simple method for generating random numbers from the standard Maxwell-Boltzmann distribution are presented. By using this approach, it is possible to demonstrate that the temperature of a material is a function only of the fluctuating component of the average molecular kinetic energy, and that it is independent of its macroscopic kinetic energy. The thought of using CUDA and GPUs comes from the fact that the calculations performed are simple arithmetic, indicating low complexity but simply the amount of calculations are so immense that sequential calculation of all parameters takes years of time, parallelizing these calculations, dramatically reduces time. Parallelization is done using Nvidia GPUs and CUDA is a platform through which Nvidia GPUs can be put to use. To take help of python in our code, we use cuPy library which provides pythonic functionalities to CUDA. A test problem i.e Gauss Seidel Iterative algorithm is run on both CPU and GPU to compare speeds and it found that the algorithm ran much faster on the GPU.

# TABLE OF CONTENTS

# CHAPTER - 1 INTRODUCTION

## 1.1 Introduction

Re-entry of vehicles into the earth's atmosphere has always been a topic of interest amongst many researchers, it forms the base of Soyuz vehicles that are responsible for to and fro transfer from the ISS. Also, safe re-entry can save up to millions of dollars. Ensuring safe re-entry allows us to reuse the resources thereby minimizing losses. Hence it's critical to understand the physics and mechanics behind re-entry.

The acceleration due to gravity at the surface of Earth is about 9.8m/s^2. Due to such high acceleration, the vehicles at re-entry achieve a Mach number ranging from about 10 to 25. Such high Mach Number flows can generate high temperatures due to friction between the atmosphere and the vehicle. The temperature reaches as high as in excess of 30000 K. Vehicles should be designed in a way to withstand such conditions and temperatures[1].

Looking at one cubic meter volume of air at sea level, it contains more than $10^{25}$ particles, and each of these particles experiences about $10^{10}$ collisions every second. All particles possess position, velocity, and internal energy. These properties continuously change in space and time due to collisions. Due to such a large number of particles, the need to consider a statistical approach for the distribution function of velocities is inevitable. The normalized velocity distribution function is the probability density function of finding a particle with a velocity within a box[7].

Therefore, by using Maxwell-Boltzmann Distributions which tells us how fast the atoms are moving. The speed distribution obeys the relationship,

$$f_v(v) = (\frac{m}{2\Pi K_B T})^{3/2} 4\Pi v^2 exp^{(\frac{-mv^2}{2K_B T})}$$

Eq 1: **P**robability **D**istribution **F**unction (PDF)

This equation is also known as the **P**robability **D**istribution **F**unction (PDF) of speed. Where '$f_v$' is the fraction of atoms with velocity 'v', '$K_B$' is Boltzmann's constant, 'T' is the thermodynamic temperature (Kelvin), and 'm' is the mass of the atoms. One of the most common ways to generate the random speeds with velocity vectors uniformly distributed over all angles is to use the **C**umulative **D**istribution **F**unction (CDF), $C_v(v)$, which can be found from the probability density function (PDF, which for this case is $f_v(v)$), as follows:

$$C_v(v) = \int_{-inf}^{v} f_v(v')dv'$$

Eq 2: Cumulative Distribution Function (CDF) in Integral form

which implies

$$C_v(v) = erf(\frac{v}{\sqrt{\frac{2K_BT}{m}}}) - \frac{\sqrt{\frac{2}{\prod}}vexp^{(\frac{-mv^2}{2K_BT})}}{\sqrt{\frac{K_BT}{m}}}$$

Eq 3: Simplified Cumulative Distribution Function (CDF)

where erf is the error function[8].

In such flow situations, the Navier–Stokes equations and other eulerian methods cannot adequately describe the transition and the free molecular regimes. In these regimes, the Boltzmann equation of kinetic theory is invoked to govern the flows. But this equation cannot be solved easily, neither by analytical techniques nor by numerical methods. The DSMC (Direct Simulation Monte Carlo) method introduced by Prof G. A. Bird is used. This method is computationally expensive and time-taking. Hence parallel computation is used to speed up our tasks. GPUs(Graphics processing unit) are used which help us parallelize the process. To harness the power and computational capacity of the **G**raphics **P**rocessing **U**nit (GPU**)** in a comfortable manner, a library called **cuPy** is used. cuPy is an open-source array library accelerated with NVIDIA CUDA. cuPy provides GPU accelerated computing with Python. cuPy uses CUDA-related libraries including cuBLAS(Basic Linear Algebra Subprograms), cuDNN (Deep Neural Networks), cuRand (Random number generation), cuSolver (decompositions and linear system solutions), cuSPARSE (sparse matrix library), cuFFT (Fast Fourier Transform), and NCCL (NVIDIA Collective Communication Library) to make full use of the GPU architecture. cuPy's interface is highly compatible with NumPy, in most cases it can be used as a drop-in replacement. All one needs to do is just replace numpy with cuPy in your Python code. Hence, it is comfortable to implement code on the GPU with the help of cuPy. Using cuPy is a great way to accelerate Numpy and matrix operations on the GPU by many times. It’s important to note that the speedups we get are highly dependent on the size of the array one is working with. The below shows that the larger the size of the array the better the speed up will be, this is the case in most of our problems as positions and velocities of millions of particles are being dealt with at the same time.

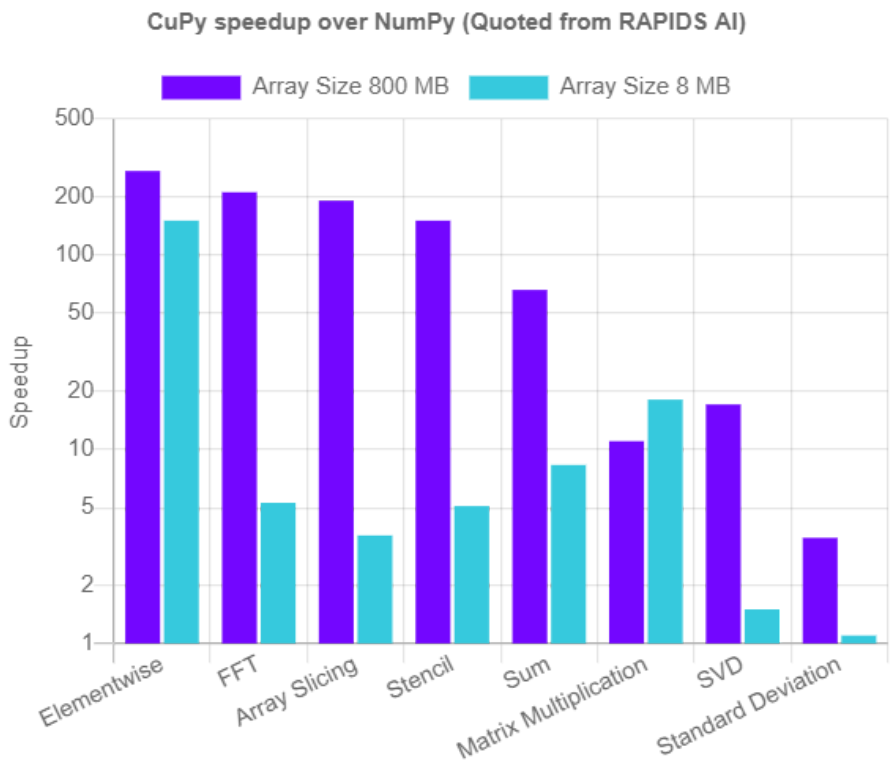


Fig 1: cuPy speedup over Numpy[5]

## 1.2 Objectives

- To understand the working of DSMC method and flow at the time of Re-Entry of vehicles.
- To learn about various parallelization techniques and parallelize DSMC method and run it with GPU and CUDA
- To get acquainted with Python and PyCUDA and to learn and explore various parallelization softwares that include PyCuda, SkCuda, cuPy and evaluate the pros and cons of each of the softwares.
- Learn and implement a practice program involving GPUs (in this case “The Gauss Seidel iterative algorithm”) to get familiar with the syntax and working of each of the aforementioned softwares.
- Plot the histograms of normal distribution of speeds/velocities and fit it with the probability distribution function curve for different temperatures and for different types of gases.

## 1.3 Organization of report

There are a total of 5 chapters in the report, the first chapter consists of introduction, motivation, and history behind the project. This chapter introduces us to all the necessary details required for the rest of the chapters. The second chapter consists of Literature Review of the DSMC method and the GPU Architecture. The third chapter consists of the Methodology followed throughout the project. The fourth chapter includes the methods untaken for development of the Maxwellian code and the parallelization code. The fifth chapter consists of the summary and future work planned for the project.

# CHAPTER - 2 LITERATURE REVIEW

## 2.1 DSMC and GPU Architecture

Direct Simulation Monte Carlo (DSMC) Method is a Lagrangian method introduced by Prof G. A. Bird[2], it is a numerical method to solve the Boltzmann Equation. This method employs simulated molecules of correct physical size, and their number is reduced to a manageable level by regarding each simulated molecule as representing a fixed number of real molecules. In our case, we use Argon gas as it is chemically inert and unimolecular. The velocity components, position coordinates, and internal states of simulated molecules are stored in the computer and are modified with time, as the molecules are concurrently followed through representative collisions and boundary interactions in simulated space. The flow properties are recorded and are used further[5].

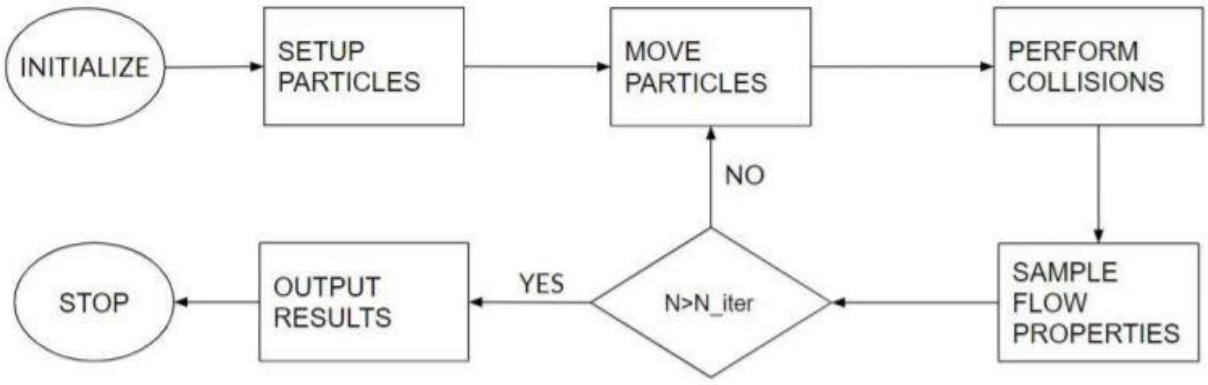


Fig 2: DSMC Flowchart

This semester more focus is put on the theoretical part and initialization of the particles was done. In the classical theory of ideal gases, the velocity distribution function is derived from the Boltzmann transport equation based on the assumption of molecular chaos and a dilute enough gas so that ternary and higher collisions can be ignored. The Maxwell–Boltzmann distribution concerns the distribution of an amount of energy between identical but distinguishable particles. It represents the probability for the distribution of the states in a system having different energies[7]. The normalized Maxwell-Boltzmann velocity distribution function (or probability density function) of a perfect gas in one dimension is given by,

$$f(v_x) = (\frac{m}{2\Pi K_B T})^{1/2} exp^{(\frac{-mv_x^2}{2K_B T})}$$

Eq 4: x-component of **P**robability **D**istribution **F**unction (PDF)

Then the velocity distribution function in three dimensions can be written as the product of the one-dimensional distribution functions[8],

$$f(v_x, v_y, v_z) = f(v_x)f(v_y)f(v_z) = (\frac{m}{2\Pi K_B T})^{3/2} exp^{(\frac{-mv^2}{2K_B T})}$$

Eq 5: **P**robability **D**istribution **F**unction (PDF) in three Dimensions

The number of particles may go up to like 10^12, positions and velocities of all particles must be stored and tracked, The calculations are simple but there are simply too many of them, sequential computing takes days to give any appreciable result, hence parallel computing is the way to go. GPUs (Graphics processing unit) are similar to CPUs, the only difference being they have smaller memory and lesser computational power but a high number of cores. GPUs are used for parallel computation. The computational efficiency of one GPU is equivalent to 10-30 processors of a computational cluster based on multicore CPUs.

CUDA[6] is a parallel computing platform and application programming interface model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled GPU for general purpose processing. In layman terms CUDA allows us to access Nvidia GPUs for computation. While there have been other proposed APIs for GPUs, such as OpenCL, the combination of CUDA and Nvidia GPUs dominates several application areas.

PyCUDA[3] is a framework that gives us Pythonic access to Nvidia's CUDA parallel computation API. It allows us to write CUDA code in Python. There are several advantages in using PyCUDA, few of them are, Object cleanup is tied to the lifetime of objects making it much easier to write correct, leak- and crash-free code. There is Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions. Python is generally considered slow but the base of PyCUDA is written in C++, hence the speed is not compromised[4].

# CHAPTER - 3 METHODOLOGY

The DSMC method requires randomly initialized particle velocities; it is done by sampling random numbers between (0,1) from a uniform distribution and then using the Box Muller transformation to convert the uniform distribution to Gaussian distribution. To plot the distribution curves, random Gaussian numbers are generated using the NumPy library. Using $f_v$(PDF), plot the distribution of velocities and speeds can be generated using the inverse of $C_v$(CDF), for that 'interpolation' package in Scipy library is used. Then the histograms of speeds are fitted with the PDF curve, and velocity distribution in the z-direction is also plotted.

To simulate the DSMC algorithm track of millions of variables is kept, they are stored as matrices. Initially, a small scale model is built, then go upon developing it further. PyCUDA and other related libraries have several inbuilt functions which allow us to access the GPU, without much difficulty. Currently, due to the lack of a CUDA-compatible GPU on our PC, the GPU support provided by Google Colaboratory and Kaggle is used. Further, access to Param Ishan, the supercomputer in IITG might be needed. GPUs do not support data in them, that is to say, direct initialization of variables cannot be done on the GPU, hence memory allocation has to be done one the CPU and a copy is passed to the GPU via pointers. For that, other Python Libraries like NumPy (Numerical Python) which has arrays in it are used. Furthermore, small samples are run on the CPU and the same on the GPU to compare the speed and note down by how much faster the GPU performs.

Additionally various libraries and softwares which allow us to use CUDA GPUs are tried out and are compared amongst themselves with various factors like speed, ease of use, robustness, reliability and finally after careful scrutiny the library that would be used for the project was decided. Then through a mini-problem (The Gauss Seidel Iterative scheme), familiarity with the syntax and the working of the chosen library was understood. In this semester both theory and computation have progressed independently without much collaboration. Both halves will be combined in future semesters.

# CHAPTER - 4 CODE DEVELOPMENT

## 4.1 Maxwellian Code

To initialize the particles( initialization implies assigning velocities to them), random numbers are sampled between (0,1) from a uniform distribution and use the Box Muller transformation to convert the uniform distribution to Gaussian distribution. For the Maxwell-Boltzman normal distribution of speed, a box is taken with atoms inside (assuming 10^5 atoms). All the atoms are in the vapour phase and we consider the gas to be ideal. The Maxwell- Boltzman normal distribution gives us the information about how fast the atoms are moving at a given temperature and the fraction of atoms $f_v$ having speed v [7] (refer to Appendix Code [5][6]).

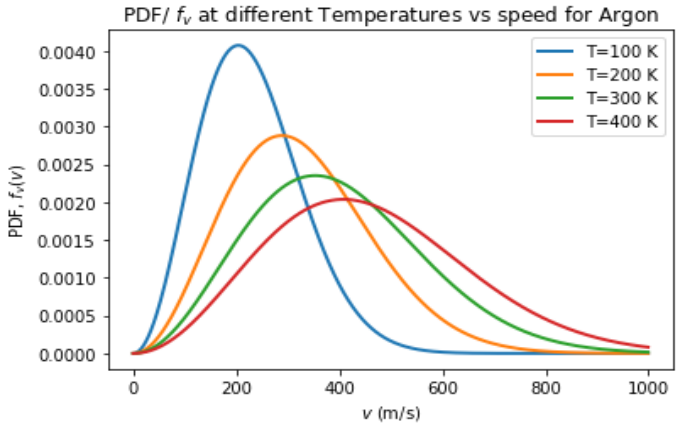


Fig 2: PDF/ $f_v$ at different temperatures vs speed for Argon gas .

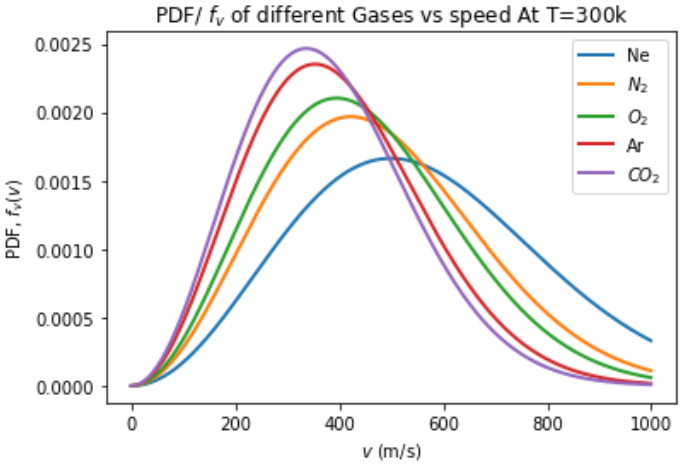


Fig 3: PDF/ $f_v$ at different gases vs speed at 300K

Figure 3 shows the probability Distribution function(PDF) of speed for different gases at T = 300k for Argon gas.

With the rise in temperature 'T', the curve, PDF(v) flattens but with the rise of molecular weight of gases(assuming all the gases as ideal), the curve sharpens.

Now the cumulative distribution function($C_v$) which indicates the probability that a particle has velocity less than equal to 'v' at a given temperature 'T' is plotted (refer to Appendix Code [7][8]). Below figure 4 shows the cumulative distribution function($C_v$) of speed at different temperatures for Argon gas.

$$C_v(v) = erf\left(\frac{v}{\sqrt{\frac{2K_BT}{m}}}\right) - \frac{\sqrt{\frac{2}{\pi}} v exp^{\left(\frac{-mv^2}{2K_BT}\right)}}{\sqrt{\frac{K_BT}{m}}}$$

Eq 6: Simplified Cumulative Distribution Function (CDF)

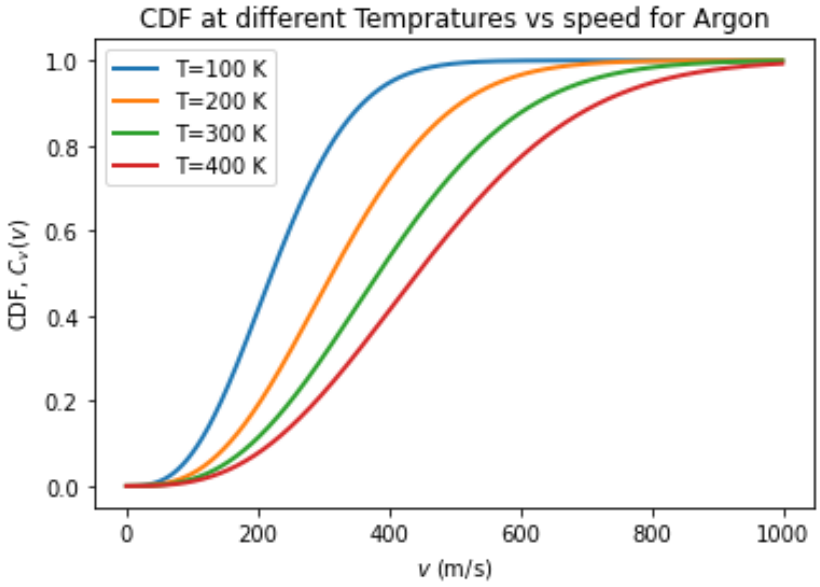


Fig 4: CDF at the different temperatures vs speed for Argon

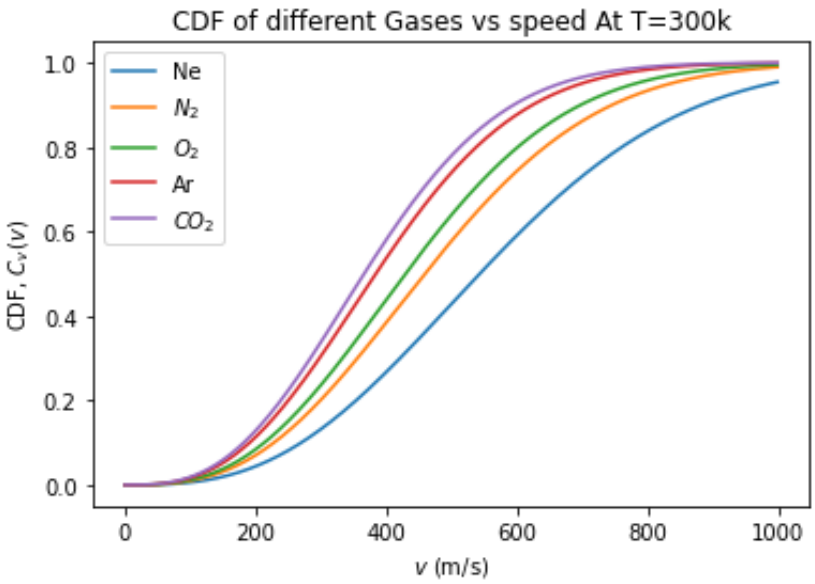


Fig 5: CDFof different Gases vs speed at T = 300K

Figure 5 shows the cumulative distribution function($C_v$) of speed for different gases at T = 300k for Argon gas.

For the histograms of velocities($V_x$, $V_y$, $V_z$) and speed, a set of random numbers uniformly distributed between 0 and 1 is taken. Then the Box-muller transformation is applied to get the set of numbers distributed normally, now the inverse of CDF is used to get the velocities, but there is no analytic inverse of the CDF function, so it has to be done numerically[7]. To achieve that, the 'interpolate' package from SciPy library is used.

Using that the histogram of speed is drawn and fitted with the Probability Distribution Function curve (refer to Appendix Code [11])..

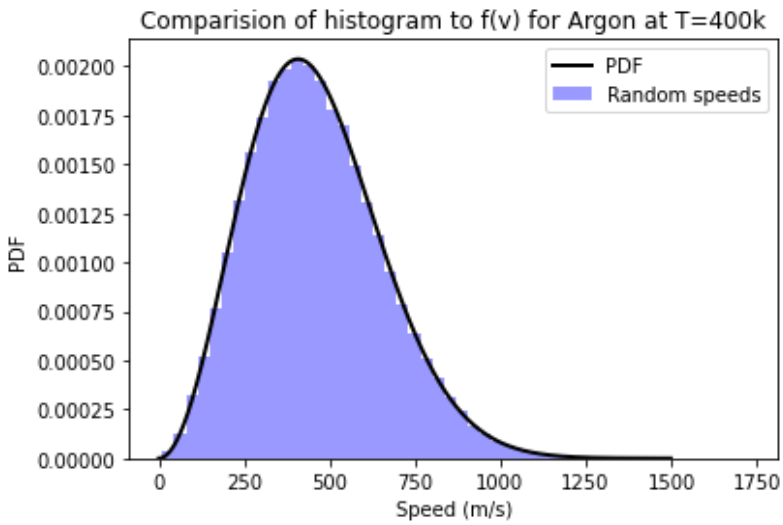


Fig 6: Comparison of Histogram to f(v) for Argon at T = 400K

Figure 6 shows the relation between histograms of speeds and PDF at temperature T= 400k for Argon gas.

Histogram of Z-component of velocity (refer to Appendix Code [12]).:

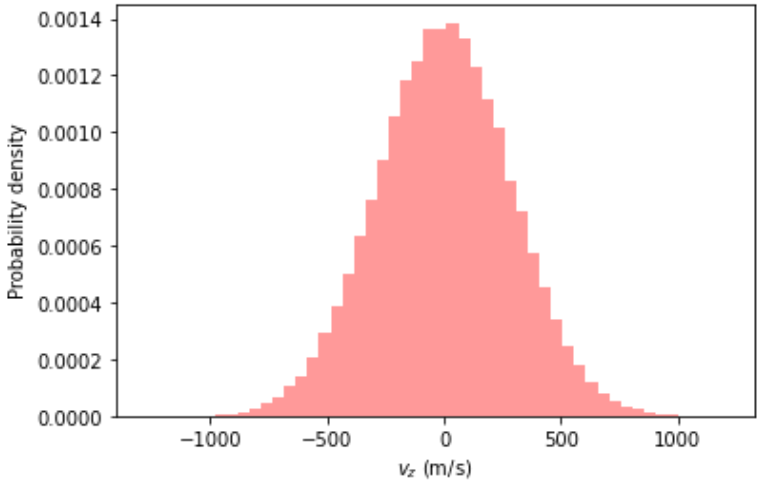


Fig 7: Histogram of z-component of velocity

Figure 7 shows the histogram of the z-component of velocity and z-component of velocity is distributed normally.

The Gaussian (normal) Distribution of $V_z$ is also drawn(refer to Appendix Code [13]).:

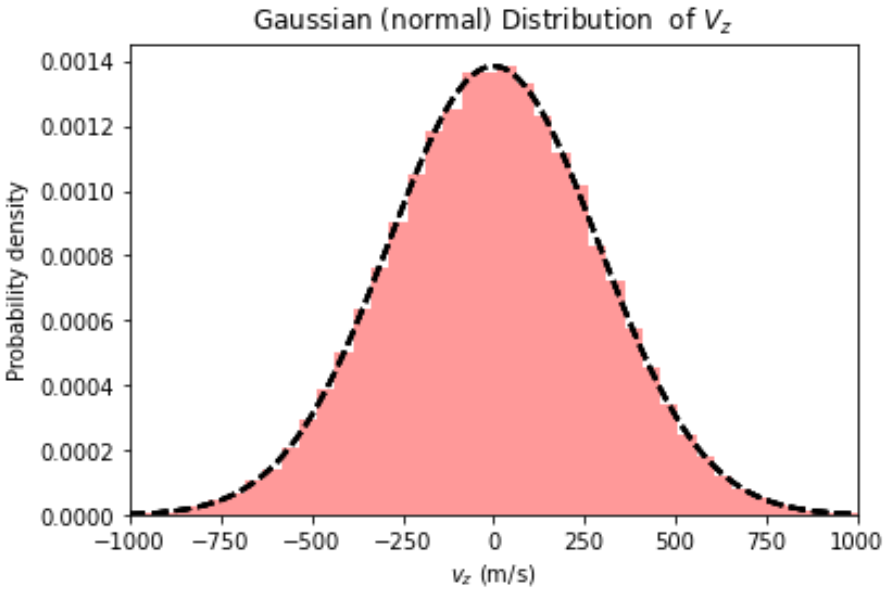


Fig 8: Gaussian (normal) distribution of $V_z$

Figure 8 shows the relation between histograms of z-component of velocity and PDF of z-component of velocity.

## 4.2 CUDA programming basics

Regarding the parallelization and GPU part, PyCuda library was explored, the goal was to use a python implementation of CUDA, so that our code can be run on the GPU, that indirectly meant raw CUDA code is not intended to be written, raw CUDA code has a syntax similar to C++ and involves object oriented programming.

Inside **PyCuda** library, different packages like “compiler”, “gpuarray” etc are explored. “compiler” package has a “SourceModule” function that allows us to write raw CUDA code and then compile it and store it in a Python object. “gpuarray” package allows us to transfer arrays on a GPU, the “gpuarray” package cannot be used to create arrays on GPU because GPU’s have very limited space. Hence arrays are created on CPU using NumPy module and transferred to GPU using the “to_gpu” function in the “gpuarray” package. The output of the code (refer to Appendix Code [1]) multiplies two **1024x1024** matrices using GPU(PyCuda) and CPU and compares the difference in their times, The outputs obtained are **GPU - 0.01176 s** and **CPU - 0.22462 s**. The limitations of this method include, writing raw cuda code, extensive code, and manual transfer of arrays between cpu and gpu.

**SkCUDA[11]** is a library which provides Python interfaces to many of the functions in the CUDA device/runtime[5]. It acts like a python wrapper to many CUDA functions. The “linalg” package of SkCUDA library provides various linear algebra functions for direct use. The output

of the code (refer to Appendix Code [2]) multiplies two **1024x1024** matrices using GPU(PyCuda) and CPU and compares the difference in their times. The outputs obtained are **CPU - 0.20921 s** and **GPU - 0.003847 s**. The benefits of using SkCUDA over direct PyCUDA is that the need of writing direct CUDA code is eliminated. However, the manual transfer of arrays between CPU and GPU and difficulty in initializing variables still persists.

**cuPy[9][10]** is the 3rd library that's been used to parallelize our computations. It is an open-source array library accelerated with NVIDIA CUDA. The reason why this particular library is chosen is that it is extremely similar to NumPy and there is no manual transfer of arrays between the CPU's and GPU's, also the need for writing raw cuda code is eliminated. Below is the output of the code (refer to Appendix Code [3]) which shows the difference in times of multiplying 2 random matrices of the same size.

```
CPU_TIME - GPU_TIME = 0.05274343490600586 for mulptiplying 2 arrays of size (500x500)
CPU_TIME - GPU_TIME = 0.3833436965942383 for mulptiplying 2 arrays of size (1000x1000)
CPU_TIME - GPU_TIME = 1.3124504089355469 for mulptiplying 2 arrays of size (1500x1500)
CPU_TIME - GPU_TIME = 4.196331024169922 for mulptiplying 2 arrays of size (2000x2000)
CPU_TIME - GPU_TIME = 11.130253791809082 for mulptiplying 2 arrays of size (2500x2500)
CPU_TIME - GPU_TIME = 18.449620008468628 for mulptiplying 2 arrays of size (3000x3000)
```

Fig 11: Difference of times(s) taken to multiply two matrices of different sizes using cuPy

It is observed that as the size of matrix increases the difference in time also increases, hence the use of GPU for small matrices is not recommended as the time taken for array transportations itself will supersede the actual computation time and inferior results will be obtained.

cuPy library suits our needs the most, hence to get better acquainted with cuPy, the **Gauss Seidel Iterative Scheme** was implemented using both CPU and GPU to get the temperature distribution of a 2D surface whose three edges are at same temperature and the 4th edge is at a different temperature. In this case the inputs are taken as

a. Length of one side of the 2D surface = 100 units.
b. Length of other side of the 2D surface = 100 units
c. Temperature of 3 sides = 20 units
d. Temperature of 4th side = 100 units
e. Number of iterations = 10000
f. Number of partitions on one side = 100
g. Number of partitions on the other side = 100

The output graph of the code(refer to Appendix Code [4]) appeared as

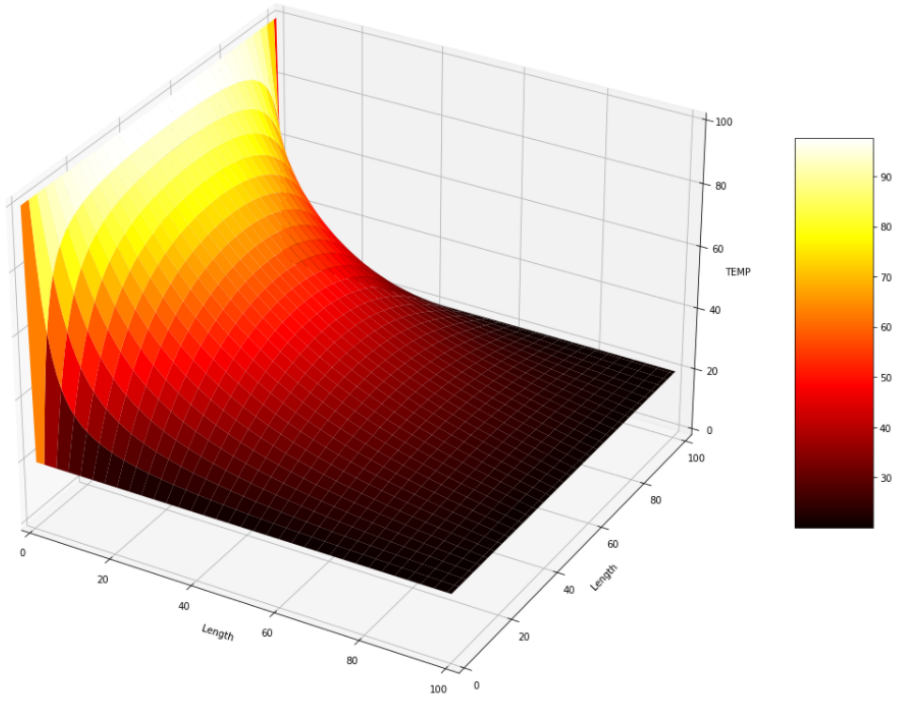


Fig 12: Surface plot of output of Gauss Seidel Iterative Scheme with Temperature on the Z-axis and boundaries on the X and Y axes.

Given that the inputs are low, the time taken for array transportation from CPU to GPU supersedes the actual computation time, hence overall time of computation is greater for GPU, Hence, we increase the inputs and re-run the algorithm and compare the difference between the times of CPU and GPU.

Inputs taken for the above result is:

a. Length of one side of the 2D surface = 5000 units.
b. Length of other side of the 2D surface = 5000 units
c. Temperature of 3 sides = 20 units
d. Temperature of 4th side = 100 units
e. Number of iterations = 100
f. Number of partitions on one side = 5000
g. Number of partitions on the other side = 5000

It is observed that CPU time is **93.23** seconds whereas GPU time is **75.47** seconds. This improvement can only be seen if the array sizes are large enough or else the CPU time would be much lower (Refer to Appendix code [4]).

# CHAPTER - 5 CONCLUSIONS AND FUTURE WORK

## 5.1 Conclusions

The speed of the DSMC particles were initialized using Maxwell-Boltzmann distribution by random sampling. The initialized velocities match well with the probability distribution function at the given temperature. This was repeated at different temperatures and for different gases. For the Maxwell-Boltzman distribution of speed, a box is taken with atoms inside and have initialized velocities to each of them using random numbers and Box-Muller Transformation, and have plotted histograms of speed and successfully fitted them with the PDF(v) curve. The distribution of the Z-component of velocity($V_z$) is also plotted and it is observed that it is distributed normally(Gaussian) as expected; it was also fitted with the PDF($V_z$) curve which confirms the same.

For the parallelization aspect, inspection of various libraries of python is done, and are evaluated on various parameters. Currently the cuPy library is finalized, successful implementation of a practice program using cuPy is done and have witnessed the speed-up achieved with the help of GPUs.

## 5.2 Future Work

Thorough understanding of the theoretical part of our project and the computational part of our project was achieved independently, in the next semester the aim is to combine our individual independent efforts and develop a DSMC code with GPU parallelization. Initialization is done, next is the implementation of the movement and collision module of the DSMC code. And parallelize it using CUDA, comparison of the two prototypes with various factors will be done and then move upon visualization using graphs. Final finishing touches will be done in the last semester.

# References

[1] P. S. Prasanth and Jose K. Kakkassery “Direct simulation Monte Carlo (DSMC): A numerical method for transition-regime flows–A review”, National Institute of Technology Calicut (2006)

[2] G. A. Bird “The DSMC Method”, CreateSpace Independent Publishing Platform, (2013)

[3] Official website of PyCUDA https://pypi.org/project/pycuda/

[4] Andreas Kl¨ockner , Nicolas Pinto , Yunsup Lee , Bryan Catanzaro , Paul Ivanov , Ahmed Fasih “PyCUDA: GPU Run-Time Code Generation for High-Performance Computing” (2009)

[5] R.V. Reji and S. Anil Lal “SIMULATIONS OF HYPERSONIC FLOW PAST A RE-ENTRY CAPSULE USING DSMC METHOD”, Frontiers in Heat and Mass Transfer (2016)

[6] OC.-C. Sua, C.-W. Hsieh, M. R. Smith, M. C. Jermy and J.-S. Wu, “Parallel Direct Simulation Monte Carlo Computation Using CUDA on GPUs” (2011)

[7] Iain D. Boyd (University of Michigan)., 1964, Thomas E. Schwartzentruber (University of Minnesota)., 1977, “Nonequilibrium gas dynamics and molecular simulation”.

[8] Mohazzabi, P. and Shankar, S. (2018) Maxwell-Boltzmann Distribution in Solids. Journal of Applied Mathematics and Physics, 6, 602-612.

[9] Official Website of cuPy https://cuPy.dev/

[10] GitHub Page of cuPy https://github.com/cuPy/cuPy

[11] Official page of SkCUDA https://scikit-cuda.readthedocs.io/en/latest/\

# Appendix (CODES)

## CODE: [1]

```
# IMPORTING THE NECESSARY LIBRARIES
import numpy as np # NUMPY FOR CPU ARRAYS
from pycuda import compiler, gpuarray, tools
import pycuda.driver as cuda
import pycuda.autoinit
# DEFINING VARIABLES
MATRIX_SIZE = 1024
BLOCK_SIZE = 32
# create two square matrices
a_cpu = 4*np.ones((MATRIX_SIZE, MATRIX_SIZE)).astype(np.float32)
b_cpu = 5*np.ones((MATRIX_SIZE, MATRIX_SIZE)).astype(np.float32)
c_cpu = np.empty_like(a_cpu)
# STORING THE START TIME
start= time.time()
# THE RAW CUDA CODE TO MULTIPLY 2 MATRICES
kernel_code_template = """
__global__ void matrixmulti(int matrixsize,float *a, float *b, float *c)
{
        // 2D Thread ID
        int tx = blockDim.x*blockIdx.x + threadIdx.x; // Compute column index
        int ty = blockDim.y*blockIdx.y + threadIdx.y; // Compute row index
        if((ty <matrixsize) && (tx < matrixsize))
        {
        float Pvalue = 0;
    for(int k=0; k<matrixsize;++k)
        {
        float Aelement = a[ty*matrixsize +k];
        float Belement = b[k*matrixsize +tx];
    Pvalue += Aelement * Belement;
        }
        c[ty * matrixsize + tx] = Pvalue;
        }
}
"""
# transfer host (CPU) memory to device (GPU) memory
```

```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.to_gpu(c_cpu)
# compile the kernel code
mod = compiler.SourceModule(kernel_code_template)
# get the kernel function from the compiled module
matrixmul = mod.get_function("matrixmulti")
# set grid size
if MATRIX_SIZE%BLOCK_SIZE != 0:
    grid=(MATRIX_SIZE//BLOCK_SIZE+1,MATRIX_SIZE//BLOCK_SIZE+1,1)
else:
    grid=(MATRIX_SIZE//BLOCK_SIZE,MATRIX_SIZE//BLOCK_SIZE,1)
matrixsize = MATRIX_SIZE
# call the kernel on the 2 previously initialized matrices
matrixmul(np.uint32(matrixsize),
        # inputs
    a_gpu, b_gpu,
        # output
    c_gpu,
    grid=grid,
        block = (BLOCK_SIZE, BLOCK_SIZE, 1),
        )
#printing the final C matrix
print(c_gpu)
print("TIME ON GPU: " + str(time.time() - start))
# MULTIPLYING THEM ON CPU
startt = time.time()
c_cpu = np.dot(a_cpu,b_cpu)
print("TIME ON CPU: " + str(time.time() - startt))
```

## CODE :[2]

```
#IMPORTING VARIOUS LIBRARIES
import pycuda.gpuarray as gpuarray
import numpy as np
import skcuda.linalg as linalg
import pycuda.autoinit
#INITIALIZING THE PACKAGE
linalg.init()
#INITIALIZING VARIABLES
```

```
MATRIX_SIZE = 1000


a_cpu = 4*np.ones((MATRIX_SIZE, MATRIX_SIZE)).astype(np.float32)
b_cpu = 5*np.ones((MATRIX_SIZE, MATRIX_SIZE)).astype(np.float32)
g_time = time.time()
#TRANSFERRING THE ARRAYS TO GPU
A_gpu = gpuarray.to_gpu(A)
B_gpu = gpuarray.to_gpu(B)
# MATRIX MULTIPLICATION IN GPU
C_gpu = linalg.dot(A_gpu, B_gpu)
e_time = time.time()
print(C_gpu)
print("GPU time(seconds) : " + str(e_time - g_time))
c_time = time.time()
c_cpu = np.dot(a_cpu,b_cpu)
print("CPU time(seconds) : " + str(time.time() - c_time))
```

## CODE: [3]

```
import cuPy as cp
import time
def time_diff(matrix_size,seed):
    cp.random.seed(seed)
    start_gpu = time.time()
        x_gpu = cp.random.rand(matrix_size,matrix_size)
        y_gpu = cp.random.rand(matrix_size,matrix_size)
        z_gpu = cp.dot(x_gpu,y_gpu)
    net_gpu = time.time() - start_gpu
    np.random.seed(seed)
    start_cpu = time.time()
        x_cpu = np.random.rand(matrix_size,matrix_size)
        y_cpu = np.random.rand(matrix_size,matrix_size)
        z_cpu = np.dot(x_cpu,y_cpu)
    net_cpu = time.time() - start_cpu
    return (net_cpu - net_gpu)
print("CPU_TIME - GPU_TIME = "+ str(time_diff(500,1)) + " for mulptiplying 2 arrays of size
(500x500)")
print("CPU_TIME - GPU_TIME = "+ str(time_diff(1000,1)) + " for mulptiplying 2 arrays of size
(1000x1000)")
print("CPU_TIME - GPU_TIME = "+ str(time_diff(1500,1)) + " for mulptiplying 2 arrays of size
```

```
(1500x1500)")

print("CPU_TIME - GPU_TIME = "+ str(time_diff(2000,1)) + " for mulptiplying 2 arrays of size
(2000x2000)")
print("CPU_TIME - GPU_TIME = "+ str(time_diff(2500,1)) + " for mulptiplying 2 arrays of size
(2500x2500)")
print("CPU_TIME - GPU_TIME = "+ str(time_diff(3000,1)) + " for mulptiplying 2 arrays of size
(3000x3000)")
```

# CODE :[4]

```
# SURFACE_PLOTTER
def surface_plotter(temp_array,delta_x,delta_y):
    m = temp_array.shape[0]
    n = temp_array.shape[1]
    x = np.zeros_like(temp_array)
    y = np.zeros_like(temp_array)
    for i in range(1,n):
        x[:,i] = x[:,i-1] + delta_x
    for i in range(1,m):
        y[i,:] = y[i-1,:] + delta_y
    # Creating figure
    fig = plt.figure(figsize =(20, 15))
    ax = plt.axes(projection ='3d')
    # Creating color map
    my_cmap = plt.get_cmap('hot')
    # Creating plot
    surf = ax.plot_surface(x, y, temp_array,
                           cmap = my_cmap,
                           edgecolor ='none')
    fig.colorbar(surf, ax = ax,
                 shrink = 0.5, aspect = 5)

    ax.set_title('Surface plot')
    ax.set_xlim([0,delta_x*m])
    ax.set_ylim([0,delta_y*n])
    ax.set_zlim([0,np.max(temp_array)])
    ax.set_zlabel("TEMP")
    ax.set_xlabel("Distance (m)")
    ax.set_ylabel("Distance (m)")
```

```
    # show plot


plt.show()
    #CPU GAUSS SEIDEL
def GAUSS_SEIDAL_GPU(len1, len2, t1,t2,t3,t4,iterations,part_x,part_y):
    delta_x = len1/part_x
    delta_y = len2/part_y
    beta = delta_x/delta_y
    start = time.time()
    temp_array = cp.zeros((part_y + 1, part_x + 1))
    next_array = cp.zeros((part_y + 1, part_x + 1))
    # INITIALIZE ALL BC
    temp_array[part_y,:] = t3
    temp_array[0,:] = t1
    temp_array[:,0] = t4
    temp_array[:,part_x] = t2
    next_array[part_y,:] = t3
    next_array[0,:] = t1
    next_array[:,0] = t4
    next_array[:,part_x] = t2
    flag = 1
    for iter in range(iterations):
        if flag == 1:
            for i in range(1,part_x):
                next_array[1:part_y,i] = (temp_array[1:part_y,i+1] +
                temp_array[1:part_y,i-1])/(2*(1 + (beta**2)))
            for j in range(1,part_y):
                next_array[j,1:part_x] += (temp_array[j+1,1:part_x] +
                temp_array[j-1,1:part_x])*beta*beta/(2*(1 + (beta**2)))
            flag = 0
        else:
            for i in range(1,part_x):
                temp_array[1:part_y,i] = (next_array[1:part_y,i+1] +
                next_array[1:part_y,i-1])/(2*(1 + (beta**2)))
            for j in range(1,part_y):
                temp_array[j,1:part_x] += (next_array[j+1,1:part_x] +
                next_array[j-1,1:part_x])*beta*beta/(2*(1 + (beta**2)))
            flag = 1
    print(time.time() - start)
```

```
    if flag == 0:
        return next_array
    return temp_array
# CPU GAUSS SEIDEL
def GAUSS_SEIDAL_CPU(len1, len2, t1,t2,t3,t4,iterations,part_x,part_y):
    delta_x = len1/part_x
    delta_y = len2/part_y
    beta = delta_x/delta_y
    start = time.time()
    temp_array = np.zeros((part_y + 1, part_x + 1))
    next_array = np.zeros((part_y + 1, part_x + 1))
# INITIALIZE ALL BC
    temp_array[part_x,:] = t3
    temp_array[0,:] = t1
    temp_array[:,0] = t4
    temp_array[:,part_y] = t2
    next_array[part_x,:] = t3
    next_array[0,:] = t1
    next_array[:,0] = t4
    next_array[:,part_y] = t2
    flag = 1
    for iter in range(iterations):
        if flag == 1:
            for i in range(1,part_x):
                next_array[1:part_y,i] = (temp_array[1:part_y,i+1] +
                temp_array[1:part_y,i-1])/(2*(1 + (beta**2)))
            for j in range(1,part_y):
                next_array[j,1:part_x] += (temp_array[j+1,1:part_x] +
                temp_array[j-1,1:part_x])*beta*beta/(2*(1 + (beta**2)))
            flag = 0
        else:
            for i in range(1,part_x):
                temp_array[1:part_y,i] = (next_array[1:part_y,i+1] +
                next_array[1:part_y,i-1])/(2*(1 + (beta**2)))
            for j in range(1,part_y):
                temp_array[j,1:part_x] += (next_array[j+1,1:part_x] +
                next_array[j-1,1:part_x])*beta*beta/(2*(1 + (beta**2)))
            flag = 1
    print(time.time() - start)
```

```
    if flag == 0:
        return next_array
    return temp_array
```

## CODE: [5]

```
#importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

##probability density function (PDF, which for this case is $f_v(v)$)
def MB_speed(v,m,T):
    """ Maxwell-Boltzmann speed distribution for speeds """
    kB = 1.38e-23
    return (m/(2*np.pi*kB*T))**1.5 * 4*np.pi * v**2 * np.exp(-m*v**2/(2*kB*T))

fig = plt.figure()
ax = fig.add_subplot(111)

v = np.arange(0,1000,1)
amu = 1.66e-27
mass = 40*amu

for T in [100,200,300,400]:
    fv = MB_speed(v,mass,T)
    ax.plot(v,fv,label='T='+str(T)+' K',lw=2)

ax.legend(loc=0)
ax.set_xlabel('$v$ (m/s)')
ax.set_ylabel('PDF, $f_v(v)$')
plt.title('PDF/ $f_v$ at different Temperatures vs speed for Argon')
plt.draw()
```

## CODE: [6]

```
##probability density function (PDF, which for this case is $f_v(v)$)
def MB_speed(v,m,T):
    """ Maxwell-Boltzmann speed distribution for speeds """
```

```
    kB = 1.38e-23
    return (m/(2*np.pi*kB*T))**1.5 * 4*np.pi * v**2 * np.exp(-m*v**2/(2*kB*T))

fig = plt.figure()
ax = fig.add_subplot(111)

v = np.arange(0,1000,1)
amu = 1.66e-27
#mass = 85*amu
Temp=300
gas = ['Ne','$N_2$','$O_2$','Ar','$CO_2$']
for i,m in enumerate([20,28,32,40,44]):
    fv = MB_speed(v,m*amu,Temp)
    ax.plot(v,fv,label=gas[i],lw=2)

ax.legend(loc=0)
ax.set_xlabel('$v$ (m/s)')
ax.set_ylabel('PDF, $f_v(v)$')
plt.title("PDF/ $f_v$ of different Gases vs speed At T=300k")
plt.draw()
```

## CODE: [7]

```
## cumulative distribution function (CDF)
from scipy.special import erf
def MB_CDF(v,m,T):
    """ Cumulative Distribution function of the Maxwell-Boltzmann speed distribution """
    kB = 1.38e-23
    a = np.sqrt(kB*T/m)
    return erf(v/(np.sqrt(2)*a)) - np.sqrt(2/np.pi)* v* np.exp(-v**2/(2*a**2))/a


fig = plt.figure()
ax = fig.add_subplot(111)

v = np.arange(0,1000,1)
amu = 1.66e-27
mass = 40*amu

for T in [100,200,300,400]:
```

```
    fv = MB_CDF(v,mass,T)
    ax.plot(v,fv,label='T='+str(T)+' K',lw=2)
ax.legend(loc=0)
ax.set_xlabel('$v$ (m/s)')
ax.set_ylabel('CDF, $C_v(v)$')
plt.title("CDF at different Tempratures vs speed for Argon")
plt.draw()
```

## CODE: [8]

```
## cumulative distribution function (CDF)
from scipy.special import erf
def MB_CDF(v,m,T):
    """ Cumulative Distribution function of the Maxwell-Boltzmann speed distribution """
    kB = 1.38e-23
    a = np.sqrt(kB*T/m)
    return erf(v/(np.sqrt(2)*a)) - np.sqrt(2/np.pi)* v* np.exp(-v**2/(2*a**2))/a

fig = plt.figure()
ax = fig.add_subplot(111)

v = np.arange(0,1000,1)
amu = 1.66e-27
#mass = 40*amu
Temp=300
gas = ['Ne','$N_2$','$O_2$','Ar','$CO_2$']
for i,m in enumerate([20,28,32,40,44]):
    fv = MB_CDF(v,m*amu,Temp)
    ax.plot(v,fv,label=gas[i])
ax.legend(loc=0)
ax.set_xlabel('$v$ (m/s)')
ax.set_ylabel('CDF, $C_v(v)$')
plt.title("CDF of different Gases vs speed At T=300k")
plt.draw()
```

## CODE: [9]

```
from scipy.interpolate import interp1d as interp
amu = 1.66e-27
mass = 40*amu
```

```
T = 400
# create CDF
vs = np.arange(0,2500,0.1)
cdf = MB_CDF(vs,mass,T) # essentially y = f(x)

#create interpolation function to CDF
inv_cdf = interp(cdf,vs) # essentially what we have done is made x = g(y) from y = f(x)
                         # this can now be used as a function which is called in the same way as
normal routines
```

## CODE: [10]

```
def generate_velocities(n):
    """ generate a set of velocity vectors in 3D from the MB inverse CDF function """
    rand_nums = np.random.random(n)
    speeds = inv_cdf(rand_nums)

    # spherical polar coords - generate random angle for velocity vector, uniformly distributed
over the surface of a sphere
    # see http://mathworld.wolfram.com/SpherePointPicking.html for more info (note theta and phi
are the other way around!)
    theta = np.arccos(np.random.uniform(-1,1,n))
    phi = np.random.uniform(0,2*np.pi,n)

    # convert to cartesian units
    vx = speeds * np.sin(theta) * np.cos(phi)
    vy = speeds * np.sin(theta) * np.sin(phi)
    vz = speeds * np.cos(theta)

    return speeds, vx, vy, vz
```

## CODE: [11]

```
spd, vx, vy, vz = generate_velocities(100000)
fig = plt.figure()
ax = fig.add_subplot(111)
#generate histogram of velocities
ax.hist(spd,bins=50,density=True,fc='b',alpha=0.4,lw=0.2)
#compare this histogram to f(v) - this is MB_speed that we wrote earlier
vs = np.arange(0,1500)
kB = 1.38e-23
amu = 1.66e-27
```

```
mass = 40*amu
T = 400
fv = MB_speed(vs,mass,T)
ax.plot(vs,fv,'k',lw=2)
ax.set_xlabel('Speed (m/s)')
ax.set_ylabel('PDF')
plt.title('Comparision of histogram to f(v) for Argon at T=400k')
plt.show()
```

## CODE: [12]

```
fig = plt.figure()
ax = fig.add_subplot(111)
h,b,c = ax.hist(vz,bins=50,density=True,fc='r',alpha=0.4,lw=0.2)
print (h.sum()*(b[1]-b[0]))
ax.set_xlabel('$v_z$ (m/s)')
ax.set_ylabel('Probability density')
plt.show()
```

## CODE: [13]

```
## it is a Gaussian (normal) form:
u = np.sqrt(2*kB*T/mass)
vzt = np.arange(-4*u,4*u,u/50)
mbz = np.exp(-vzt**2/u**2)/np.sqrt(u*np.pi)
mbz /= (mbz.sum() * (vzt[1]-vzt[0]))
ax.set_title('Gaussian (normal) Distribution  of $V_z$')
ax.plot(vzt,mbz,'k--',lw=2)
ax.set_xlim(-1000,1000)
display(fig)
```