An object-oriented serial implementation of a DSMC simulation package

Hongli Liu, Chunper Cai*, Chun Zou

Department of Mechanical & Aerospace Engineering, Jett Hall 505, MSC 3450, New Mexico State University, Las Cruces, New Mexico 88003, United States

ARTICLE INFO

Article history:
Received 30 October 2010
Received in revised form 3 December 2011
Accepted 12 December 2011
Available online 19 December 2011

Keywords:
DSMC implementation
Hybrid grid
Object-Oriented Programming
Rarefied gas flow

ABSTRACT

This paper reports a scalar implementation of a multi-dimensional direct simulation Monte Carlo (DSMC) package named “Generalized Rarefied Gas Simulation Package” (GRASP). This implementation adopts a concept of simulation engine and it utilizes many Object-Oriented Programming features and software engineering design patterns. As a result, this implementation successfully resolves the problem of program functionality and interface confictions for multi-dimensional DSMC implementations. The package has an open architecture which benefits further development and code maintenance. To reduce engineering time for three-dimensional simulations, one effective implementation is to adopt a hybrid grid scheme with a flexible data structure, which can automatically treat cubic cells adjacent to object surfaces. This package can utilize traditional structured, unstructured or hybrid grids to model multi-dimensional complex geometries and simulate rarefied non-equilibrium gas flows. Benchmark test cases indicate that this implementation has satisfactory accuracy for complex rarefied gas flow simulations.

Published by Elsevier Ltd.

1. Introduction

The national new spaceport program stimulates further interest on hypersonic non-equilibrium flows and space weather, including rarefied gas flows, plasma flows, and radiations. A set of compressible gas and plasma simulation packages are implemented to serve as an education and research platform for rarefied gas and plasma flows. These packages are named “Generalized Rarefied Gas Simulation Package” (GRASP) [1]. This paper presents an implementation of the direct simulation Monte Carlo (DSMC) [2] method, for multi-dimensional rarefied gas flow simulations.

The DSMC method is widely adopted for rarefied gas or plasma flow simulations. It is a stochastic simulation method in which each simulation particle represents a large number of physical gas molecules. The simulation particles possess physical properties such as position, velocity and, if applicable, internal energy information. If gas is highly rarefied, molecular movements and collisions are decoupled. Molecules either move freely or interact with boundaries. In the collision step, translational and internal energy is re-distributed between molecules according to a chosen collision model. At wall boundaries, molecules reflect back according to a selected reflection model. When molecules cross inlet or outlet boundaries, they are removed from the simulation without any further interactions. At the same time new molecules are introduced into the flow area from the free stream or inlet regions. The number of molecules introduced into the gas flow area and their velocity components depend on the boundary conditions [2]. During the past 40 years, there have been continuing proof and work to provide stronger supports to the validity, or even some further development for the DSMC method. For example, Wagner [3] provides a rigorous proof that DSMC simulations actually provide solutions to the Boltzmann equation in the limit of vanishing discretization and statistical error. Further evidence indicates that highly refined DSMC simulations provide results that agree with exact solutions to the Boltzmann equation, such as the near-equilibrium infinite-order Chapman–Enskog and the non-equilibrium Moment Hierarchy methods [4-5]. There have been several DSMC implementations in the literature, including those programs by Bird’s DS2V/3 V [2], SMILE [6], MONACO [7], DAC [8], Icarus [9] and MGDS [10].

In software engineering with large scale programming, reusability and maintainability are two important requirements that Object-Oriented Programming (OOP) [11,12] can serve well. There are several major modules in a DSMC simulation package, i.e., movement, collision, particle indexing and input–output modules. Only the movement and particle indexing modules are different for different dimensions. As one of the most popular OOP languages, C++ has encapsulation, inheritance, polymorphism, and other features suitable to address the reusability and maintainability issues. To our best knowledge, this implementation is one of a few DSMC implementations completely utilizing C++ and some features are worthy to be represented to the community. There are two major issues to address in this paper, code architecture and three-dimensional simulations.

One common issue for DSMC code architectures is the crowdedness of code interfaces and functionalities, especially when it is desired to design and implement one code for multi-dimensional...
flows, i.e., two-dimensional, axisymmetric, and three-dimensional situations. As a common practice, programmers use the conditional compilation, such as "#ifdef ... #else ... #endif", to achieve the goal of one code for all dimension scenarios. This approach is prone to create confusions and usually results in incompleteness of the whole code structure. In software engineering, this can be classified as conflicts between subroutine functionalities and interfaces. To resolve this problem, OOP and several design patterns are adopted.

On the other hand, three-dimensional DSMC simulations are relatively challenging and time-consuming. Usually, there are two categories of meshing treatments for three-dimensional DSMC simulations. One adopts completely unstructured mesh which accurately triangulates an object surface, particles move cell by cell during each step and collide with surface accurately. However, these schemes need to track each particle to determine whether it passes a cell face or hits a surface. For each time step, the algorithm needs to determine which side of a cell that the particle can cross, and monitor the time left in this time step. Further, the type of DSMC simulations depend on the mesh size, for example, to efficiently and accurately simulate hypersonic flows around a reentry vehicle, at different altitudes, different mesh sizes are recreated according to different mean free paths. As a result, the engineering and simulation cost can be expensive. The other category is hybrid or cartesian grids, because DSMC decouples particles' collisions and movement, after the sub-step of collisions, the particles are free to move fast, as long as the particles position can be accurately located in a cell. As a result the computation cost to determine the movement is significantly reduced. Further, simulations do not heavily depend on the mesh regeneration procedure. However, the surface representation can suffer from accuracy loss, when compared with triangulated surfaces in the other category. In general, this implementation can utilize many different grid systems to simulate multi-dimensional rarefied gas flows. Fig. 1 shows several grid types, and unstructured mesh systems are used for two-dimensional and axisymmetric simulations, to guarantee the program's stability and precision. For three-dimensional cases, this hybrid mesh divides the computational domain into solid cubic cells to track molecular trajectories efficiently, whereas the object surface can be triangulated by any major CAD/CAE software packages and read in by the simulator. In general, this implementation is not only accurate with unstructured meshes, but also efficient for three-dimensional simulations. During a three-dimensional simulation process, most of simulated molecules are efficiently tracked and moved within a hybrid or cartesian coordinate system, while a small portion in a small region adjacent to the object surface is treated delicately like an unstructured grid scheme.

The structures of this paper are organized as follows: Section 2 reports the data structure, several design patterns, and OOP styles for this implementation; Section 3 presents a special three-dimensional mesh scheme; Section 4 presents other minor implementation details; Section 5 presents several benchmark test cases; and Section 6 concludes this paper.

2. Data structure and design patterns

This section presents the code data structures and several design patterns to achieve a relatively open implementation architecture for multiple dimensions and methods.

The most important data structures for a DSMC code is particle storage. The complete particle information includes global position, velocity, internal energy and a cell id. The cell id represents in which cell a particle locates. There are two popular particle storage data structures. One is from the book by Bird [2], with a large single array to store information for all particles. Because the number of particles in flow field is difficult to determine before code starts, the particle array size must be set sufficiently large to store all the particles' information. Actually, this storage scheme is popular for implementations. However, the storage space on hardware is wasted inevitably, and within a simulation, if it is desired to clone more particles to achieve higher precision, the whole simulation process may have to restart. Another approach is to utilize linked lists inside each cell, i.e., to divide the single large particle table into many small linked lists for each cell. As well known, a linked list is an efficient solution to achieve resizable array data structure. The requested storage space can always be adjusted to the number of particles. Some advanced implementations adopt two linked lists for each cell, one works as a current list the other one for backup.

The implementation adopts the first particle storage approach [2] with some modifications. This data structure is less demanding since changing particle's location from one cell to another requires only a cell ID change for the particle. As a result, the computation efficiency can be high. Different from Bird's original implementation, a special container class is introduced and serves as a memory manager for the particle array. This class provides a resizable particle array with two integer flags. Whenever the actual particle number equals to the maximum available particle slots, a larger size memory chunk in the free store is applied, particles information in the original smaller array is copied to the new array. When a simulation reaches a steady state, this resizable particle array can shrink to a sufficiently large size for particle storage. This class guarantees that there is always enough memory available even for a clone process.

As mentioned previously, when users plan to design and implement one code for multi-dimensional flows, it is desirable to resolve the conflicts between subroutine functionalities and interfaces. More importantly, the features of the ideal DSMC code should achieve reusability and modifiability. For DSMC simulation
packages, cells and particles must be adopted. However, only using these two levels cannot provide the package a good architecture for further development and does not fully utilize the advantages of OOP. This package adopts the concept of “simulation engine” classes [13] to add another layer for the implementation. They resolve the confictions and benefit future development. This “simulation engine” concept is widely used in engineering software packages, it is a combination of the abstract factory and singleton design patterns [14].

Fig. 2 illustrates the internal derivation relations.

This implementation has obvious merits: many internally related simulation methods, e.g., 2D/AXI/3D versions of DSMC, Particle-In-Cell (PIC) to simulate plasma flows in space weather, and Information Preservation (IP) method to simulate low speed gas flows inside MEMS, a total of $3 \times 3 = 9$ implementations can be incorporated into one code without confusions. This package only needs to be compiled once, then by simply changing one card in the input deck file, user can conveniently simulate many problems with different methods; while for code developer, maintenance work can be reduced to minimum because most classes only possess a few different functions. Theoretically, it is feasible to incorporate as many methods as needed. By comparison, for traditional non-OOP implementations, it is difficult to achieve similar capability by using of conditional compilation with "#ifdef ... #else ... #endif", or selective file compilations with different make-files.

The singleton design pattern is adopted to implement the mathematical concept of a singleton, by restricting the instantiation of one or several classes to one object. The crucial advantage of this design pattern is to ensure a single object which provides a global access point to other functions. DSMC solvers can simulate different flows with multi-dimensions, however, for a specific problem, there should be only one simulation engine object instantiated. The singleton pattern has several benefits suitable for this character of DSMC: [14].

1. Controls access to sole instance. Because the singleton pattern encapsulates its sole instance, it can have strict control over how and when clients access it;
2. Reduces name space. The singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. Permits refinement of operations and representations. The singleton patterns may be sub-classed, and it is convenient to configure an application with an instance of this extended class.

One specific example from the code illustrates this design pattern clearly. For example, the previous simulation engine concept defines many different classes which have internal relations, it is a typical abstract factory design pattern on definitions. The singleton pattern is about initialization an object from only one of these classes. This sole object can be either 2D, AXI or 3D, but not all of them for one simulation. Depending on a control card defined in an input file provided by users, one proper engine object is declared. This function is achieved with the following simple coding:

```cpp
... static AbsEngine *Engine_PTR = NULL;
int main (···);···; if (mode ==1) Engine_PTR = new STD_Engine ();
else if (mode ==2) Engine_PTR = new AXI_Engine ();
else if (mode ==3) Engine_PTR = new 3D_Engine ();
Engine_PTR->ReadCardsInput (···);···;
···; Engine_PTR->MoveParticles (···);···)
... }
```

The above code segment actually guarantees that only one specific simulation engine is initialized. In this implementation, the most important data structures in a simulation engine, is the resizable particle array and a fixed cell table. Fig. 3 illustrates these relations. Different engines may have different function bodies to override the one in the base engine; or there is no solid function body at all - for this situation, the available base class function is

![Fig. 2. Relations among different simulation engine classes.](image)

![Fig. 3. Major data structure of resizable particle and fixed cell tables inside the class of 2D simulation engine.](image)
used for the simulation. For example, the "ReadCardsInput ()" function is only defined in STD_Engine (), and the derived classes from this class do not have any corresponding function at all.

The abstract factory and the singleton patterns are completely compatible and complimentary, one for definitions of several internal related classes, the other is about creating a unique object from these classes [14]. With these two patterns, the package has improved code architecture and maintainability.

The class concept in C++ is also used to package different elementary data variables and functions together into one data structure representing a DSMC cell. Its features, such as encapsulation and inheritance, can maintain the code favorable scalability, portability, and commonality. For a three-dimensional simulation situation, there are many space structures (such as cells) similar to cubic boxes. Many three-dimensional operations, such as particles' motions through inner cells, collisions at the wall, are incorporated inside a bounding box or a vector class.

In this implementation, a file named main.cpp is used to control the whole DSMC simulation loop. The abstract_engine.cpp defines major interfaces, and the key class dsnc_std_engine.cpp contains most general DSMC simulation functions for two-dimensional simulations. The other classes contain some special overriding functions for axisymmetric or three-dimensional simulations respectively, by providing different, concrete function bodies with the same function name and parameter lists.

3. A hybrid mesh system for three dimensional simulations

It shall be emphasized that the implementation can perform simulations with different mesh types, unstructured, structured, and hybrid/cartesian ones; for different applications, an adoption of a proper mesh type may improve the simulation efficiency significantly.

For engineering applications, three-dimensional problems are of special interest because they can represent the flow fields more accurately. However, for three-dimensional problems, generating three-dimensional meshes with a CAD/CAE software package usually requires significant engineering effort. In this implementation, a hybrid mesh method is implemented (shown in Figs. 4 and 5) and it requires simpler input of triangulated object surface, while the real three dimensional cubic meshes are generated internally without user's interactions. This is achieved with a special simulation engine class "3D_Cart_engine", where special particle movement and mesh-reading functions are incorporated to override the function members in its base class.

The mesh treatment in this DSMC implementation is highly automatic. The mesh generation starts according to the principle of \( \Delta l \leq \frac{1}{2} \lambda \), the size of each cell \( \Delta l \) is determined, where \( \lambda \) is the gas mean free path based on the characteristic scale obtained from the inflow conditions. Users only need to generate a triangulated object surface with any CAD/CAE software package. In this implementation, there is a special function to read in the object surface configuration data prior to the main DSMC loop. As shown in Fig. 4, only triangular surface vertex coordinates are needed for the package. After reading in the object information, this scheme automatically generates a rectangular bounding box around the object with proper sizes, e.g., \( 40 \Delta l \times 35 \Delta l \times 20 \Delta l \) according to the object actual three directions. This bounding box is slightly larger than the packing box to contain the object. Based on the bounding box, the mesh system develops outwards according to the predefined size of flow field, e.g., \( 200 \Delta l \times 150 \Delta l \times 100 \Delta l \). In the end, the package separates the whole simulation domain into hybrid/cartesian cells. This process is shown in Fig. 5.

In this three-dimensional grid system, particle movement is very efficient. Simulated molecules interact with object surfaces many times. It is a crucial process in the DSMC simulation to handle the interactions between particles and the body surfaces accurately. Therefore, in this special three-dimensional scheme, the most important work is to find the relationship between the object surface and the hybrid/cartesian grids. If the object surface is planar, it may cut a cubic cell into two parts, resulting in an intersection planar surface with 3–6 sides, depending on different number of vertexes cut off. On the other hand, if the object surface is not planar, for example, a sharp point intruding into a cubic cell, then this small vertex is neglected, and the remaining cell volume is computed by Monte Carlo method. For a hypersonic or space weather simulation, this treatment has sufficiently high accuracy for engineering purpose. More accurate and delicate treatment is under further development. This hybrid mesh treatment is one of the crucial pre-processing steps in this scheme. There is a procedure which can directly, accurately identify and record body surface's distribution within the three-dimensional cubic cells [15].
Consequently, the major steps to create a whole hybrid mesh system for a three-dimensional hybrid/cartesian DSMC simulation are as follows [15]:

1. Triangulate the object body surfaces, record their geometry information, and output the results for the solver to read. Any major CAD/CAE software packages can accomplish this task. This is the only external pre-processing step required by the code.

2. Automatically determine the bounding box around the object surface configuration and develop a larger simulation domain for the flow field. This is achieved by a special bounding box class, based on the read in object data. For example, the outer simulation domain of a spacecraft is automatically determined by offsetting the bounding box of the spacecraft surface profile.

3. Divide the whole computational domain into cubic solid cells according to the characteristic scale of inflow conditions; using cubic cells significantly simplifies the process to determine a particle’s location.

4. Record the relations between object surface and hybrid/cartesian mesh adjacent to the surface. First, identify the surface triangles and the spatial three-dimensional cubic cells by two series of numbers; if a body surface triangle relates to one cubic cell, record this relationship; next, record the numbers and the serial numbers of the surface triangles related to the current cell, and record the related cells with every surface triangle one by one. This implementation utilizes two special classes of fixed-size arrays, named Cell_table[i].triangle_list[j] and Triangle_table[j].cell_list[i], to store the information of related cells and surface triangles.

Steps 3, 4 are accomplished by one C++ class. Automatically generating the three-dimensional mesh reduces the engineering time for the whole simulation. At the same time, utilizing cubic mesh can significantly improve the simulation speed, because the module of particle movement is greatly simplified, -no cell-by-cell particle tracking is needed.

4. Other implementation features

This implementation adopts card controlled input which is a common treatment in commercial finite element analysis software packages, as a result many operations can be conveniently enabled/disabled by modifying a card in a input deck file.

The preprocessing and postprocessing are separated from the simulation package. 2D/AXI structured or unstructured mesh are created with CAE software packages, output with an finite element format, e.g. NASTRAN, and read into the code. For three-dimensional hybrid/cartesian mesh, the preprocessing process is much complex: a CAE software package is used to triangulate the object surface, all nodes and cell ids are sorted, cell edges are cleaned, normals are adjusted, and output with an FEA software package output format. This implementation reads in such an input file and performs an internal hybrid/cartesian mesh generation, the procedure is very similar to the DAC package, however, this implementation relies on some matured CAE software package with a Graphic User Interface (GUI) to improve the triangulated surface mesh quality. Due to this reason, it is much convenient to prepare such input file. The next step is internal hybrid mesh generation with hybrid/cartesian mesh for the main flow field, this step is completed prior to the main DSMC loop. This process needs only to be executed once, and there is no significant cost on the whole DSMC loop between the serial and parallel computing versions.

Obviously those OOP implementations of class are very helpful to avoid mistakes and create a robust package. All position and vector related operations are summarized with two classes, Vect2D and Vect3D, for 2D and 3D scenarios. There are many numerical operation overloading, such as addition and cross product. One example is the particle’s three dimensional position, “particle → position,” in a hybrid/cartesian mesh can be simply updated with an addition of “particle → velocity × Δt”.

Fig. 6 shows the moving process of a particle in the 3D hybrid grids. By average, this simple translation of a particle in hybrid/cartesian mesh costs less time than an unstructured mesh system. It is convenient to index the particles to determine in which cell they finally locate. Then, after all particles complete their movement within one complete time step, particles can be conveniently sorted one by one globally according to the rule proposed by Bird [2]. The processes mentioned above can reduce much computation time than the other algorithm with an unstructured mesh system.

When particles move in those cells adjacent to the object surface, a particle-tracing algorithm determines if the particle moves outside the current cell, and either one side of the current cell or one surface triangle it could reach, and which side of the current cell or which surface triangle it might reach. As shown in Fig. 7, the particle-tracing function is called continuously for each face (either the side of cell or the surface triangle of object) related to the cell adjacent to the object surface, respectively. The calling for particle-tracing function would provide the time to hit those faces, according to an inner product between the particle velocity vector and the normal vectors of those faces. The particle is subsequently moved to the point which has the shortest hitting time. The particle-tracing function is then called to move the particle till using up the remaining of the whole time step. This process requires more time than moving particles in the cells outside the
bounding box, but it is more efficient than the similar process in an unstructured mesh system.

In this implementation, the computational time spent in the collision functions called in one cell is theoretically similar to those in an unstructured mesh system. The total time for collision depends on the total number of the particles and the cells adopted in the simulation. In general, for three-dimensional simulations with a hybrid grid system, this implementation costs less time than the code with an unstructured mesh system.

5. Validation cases

Several test cases are provided in this section to demonstrate the capability of this implementation. The simulation results are compared with those obtained from either the MONACO-V3.0 code, or analytical, or experimental results. The object surfaces in the test cases are assumed completely diffuse, if not specifically mentioned.

5.1. Hypersonic flow over a wedge

The first test case is a hypersonic flow over a 40° wedge with a 10° angle of attack (AoA). The top side of the wedge is equivalent to a 10° one while the bottom side a 30° one. Free stream argon gas flow temperature is 200 K, Mach number is 10, the wedge wall temperature is 300 K. The free stream Knudsen (Kn) number is 0.05, the characteristic length is the wedge base length.

Figs. 8 and 9 show that this implementation and MONACO-V3.0 predict virtually identical results for the temperature and density contours around the wedge, two different shock waves of different strength over the leading edge and expansion waves at the rear side display clearly. Fig. 10 displays normalized surface wall pressure distributions. The two sets of simulation results are sufficiently close, indicating that this implementation and MONACO-V3.0 code can predict flows with similar accuracy.

5.2. Hypersonic flow over a 2D cylinder

Maslach and Schaaf [16] measured the drag coefficients for gaseous flow over a cylinder at Mach numbers ~2, ~4 with Kn numbers from continuum to free molecular flow conditions. In experiments, the sphere diameter is 0.0015 in., sphere surface temperature \( T_w \) is 300 K, and the ratio of the stream flow total temperature \( T_0 \) to \( T_w \) is 0.87.
Figs. 11 and 12 compare of the cylinder drag coefficients predicted by GRASP and measured by experiments. In these simulations, both diffuse and specular reflections for the cylinder surfaces are adopted.

In Maslach’s paper, the experiment data indicated a smooth transition from low Kn number results to free molecular results. These figures indicate that the simulation results with fully diffuse surface agree relatively better with the experiment results.

5.3. Hypersonic flow over a 2D flat plate

This case is a hypersonic rarefied flow over a flat plate with different AoAs. The plate has zero thickness and a length of 1.0 m. The flow conditions simulated are those experienced by the Shuttle Orbiter during a reentry process at 7.5 km/s, for an altitude from 90 to 130 km. For the 1.0 m flat plate, the corresponding Kn numbers are from 0.023 to 8.439, which are the conditions that cover most of the transitional flow regime. The plate surface temperature is $T_w = 1000$ K.

Figs. 13 and 14 reveal the drag and lift coefficients as a function of free stream Kn numbers for the flat plate, with a 40° AoA. Both figures compare GRASP results and Moss et al.’s DSMC simulation [17] results; and expected variations in the transitional flow regime are observed. The drag coefficients increase and the lift coefficients decrease substantially with increasing rarefaction. The agreement between these simulation results is satisfactory.

Figs. 15–17 show rarefication effects on pressure, shear stress and heat flux coefficient distributions on the ram side of the flat plate, respectively, with different free stream Kn numbers. The pressure coefficient increases slightly with decreasing Kn numbers. By comparison, the shear stress and heat flux coefficients are sensitive to rarefaction effects and increase with the Kn number. These observations are consistent with Moss’s conclusions [17].

Figs. 18 and 19 illustrate the AoA effects on the aerodynamic coefficient for Kn = 8.4. The new implementation results are compared with those from free molecular analysis. These results are similar to Moss’s simulation results. The figures reveal that the lift coefficient increases with increasing AoA, with the maximum value at 45°; then it decreases with increasing AoA. The results are much higher than the free molecular values, indicating that transitional effects are evident even at the highly rarefied condition. On the other hand, the drag coefficient agrees well with the free molecular results of all AoAs.
This section includes two numerical examples of hypersonic rarefied gas flow around a sphere. The simulations utilize the axis-symmetric module. One investigates the drag coefficients from near continuum to free molecular flow. The other simulates the density distribution along the stagnation line in front of the sphere with different Reynolds numbers.

5.4.1. Drag coefficients at different Kn numbers

Legge and Koppenwallner [18] experimentally investigated the drag coefficient dependence on Kn numbers, for hypersonic rarefied gas flows over a sphere. The diameter of the sphere is 0.04 m. The total temperatures of the free stream air gas flow and the sphere wall are both 500 K. The Mach number of free stream is 9.0. The DSMC simulations employ the same flow conditions as experiments, with different Kn based on the sphere diameter, the Kn number varies within 0.01–100.

Fig. 20 compares different drag coefficients over the Kn numbers. It is evident that the simulation results agree well with the measured data over the entire flow regime.
5.4.2. Density distributions along the stagnation line

Russell [19] measured density distributions along the stagnation line of a sphere. In that experiment, the nitrogen Mach numbers were 4.2 or 4.38. The free stream total temperature and the surface temperature were both 300 K. The new implementation simulates the experimental conditions which vary with different Reynolds numbers. Fig. 21 compares simulation and experiment results, where the normalized factors are the sphere radius and free stream density. The agreement is remarkably consistent.

5.5. Hypersonic rarefied gas flow over a sphere using 3D version

This case is a hypersonic free molecular flow over a sphere, with the three dimensional simulation scheme and hybrid meshes. The free stream gas is argon, temperature is 300 K, the same as the sphere wall temperature. The sphere radius is 0.5 m. The purpose of this example is to validate the 3D particle movement module of the package, which is crucial for the three-dimensional mesh system.

The hybrid/cartesian grid system is adopted for the whole flow fields. The simulation domain is a cubic zone, and the center region is a bounding box for a sphere, while the other six external surfaces are obtained by offsetting the bounding box. The offset values are regulated by an input parameter from a deck file, as such, adjusting the simulation domain and generating the internal mesh are highly automatic, the corresponding engineering time to solve this problem is thus significantly reduced. The collisionless case is conveniently achieved by turning off the molecular collision functions.

Fig. 22 shows the temperature distributions in the middle plane, for better comparison, only the front side contours are compared. The solid lines represent the GRASP simulation results, while the dashed lines are the analytical results for collisionless flow over a sphere [20]. Even the flow is collisionless, at high Mach

![Fig. 21. Comparison of GRASP density distributions along the stagnation line of a sphere with measured data, $T_0 = T_w = 300$ K.](image)

![Fig. 22. Temperature contours for a hypersonic flow over a $r = 0.5$ m sphere, $Ma = 5$, Unit: K.](image)
nient to maintain, and the package successfully incorporates faces. The architecture is open for further development and conforms to several major classes. The simulation engine concept successfully adopts the concept of simulation engine and incorporates such as inheritance, abstract class factory, and singleton. The package. It utilizes OOP styles and several design patterns, including, but not limited to, factory method, singletons, and observer patterns.

6. Conclusions

This paper presents recent implementation of a DSMC simulation package. It utilizes OOP styles and several design patterns, such as inheritance, abstract class factory, and singleton. The package adopts the concept of simulation engine and incorporates several major classes. The simulation engine concept successfully resolves the confictions between code functionalities and interfaces. The architecture is open for further development and convenient to maintain, and the package successfully incorporates several simulation methods within a single package, and reduces conditional compilations to minimum.

Another novel feature of this implementation is the hybrid/cartesian mesh system in the package with a special class, while unstructured or structured types of meshes are still applicable. For reentry flow simulations, only triangulated object surface geometry is required as inputs, and other major control information is provided through a deck file. The DSMC mesh is generated internally rather than externally, as such, the engineering and computational time for a simulation can be significantly reduced. By utilizing cubic cells, the particle movement module is simplified significantly. Test case results indicate this implementation has high accuracy.

Acknowledgement

The authors gratefully acknowledge partial financial support to this work from NSF-CBET-0854411 and NSF-DMS-0914706.

References