OBJECT-ORIENTED SOFTWARE DESIGN FOR THE THREE-DIMENSIONAL DIRECT SIMULATION MONTE CARLO METHOD

A.V. Kashkovsky, P.V. Vashchenkov and M.S. Ivanov

Institute of Theoretical and Applied Mechanics, Institutskaya 4/1, Novosibirsk, 630091, Russia

Abstract.
The most relevant aspects associated with implementation of a 3D DSMC code are considered. The optimal structure of classes of model particles for an object-oriented DSMC code is demonstrated. The basic components of the algorithm for finding the point of intersection between the particle trajectory and the triangulated spacecraft surface are presented: generation of a rectangular mesh, determining the sequence of cells where the particle trajectory passes, and searching for the point of trajectory intersection with surface elements located in these cells. The new 3D code is tested, and the results are compared with data obtained by the SMILE code.

Keywords: DSMC, Software design
PACS: 47.45-n 02.60.Cb 01.50.hv

INTRODUCTION

The increasing interest in great potentials of the DSMC method [1] involves the development of new models that offer a more detailed description of complicated processes in gases. The study and application of these models require a software code that can be readily modified for new capabilities. Object-oriented programming (OOP) is a key mechanism in this task. This work continues the activities on the development of an object-oriented code for the DSMC method [2, 3].

The process of DSMC simulation is divided into two consecutively executed stages almost independent of each other: transfer of all particles to a prescribed time step and collision of particles with each other. The second stage does not depend on the problem dimension (1D, 2D, 3D, or axisymmetric); it suffices to know the distribution of particles over the cells. The problem dimension is completely determined by the transfer stage. Using the object-oriented programming technology, one can generate classes that provide transfer in different dimensions. If these classes are absolutely independent of the collision-modeling stage, then any new developed model of intermolecular interaction becomes automatically available for all flow dimensions. This offers a possibility of rapid development, testing, and application of new models for a wide range of problems.

3D computations deal with real spacecraft and other vehicles, which have rather complicated configurations. At each time step, all particles have to be checked for their possible interaction with the surface. As the number of particles is very large, it is important to have a fast algorithm of finding the point of intersection of the particle trajectory with the spacecraft surface. The most important aspects of the 3D implementation of particle transfer are considered in the present paper.

PARTICLE STRUCTURE

Model particles have always three components of velocity, whereas the remaining properties are divided into two parts:
1) Move part - coordinates of particles and all necessary parameters for transfer.
2) Collision part - is used for modeling molecular collisions (see Figure 1).

The tPtc template contains (*Velocity*) and a reference to *MovePart* and *CollisionPart*. A real particle is obtained from this template by substituting particular classes (or their inheritors) instead of *MovePart* and *CollisionPart*. For instance, the classes *cPtcC2D* and *cPtcC3D* contain the particle coordinates for the 2D and 3D cases (*cPtcC3Dt* also contains some additional parameters). The class *cPtcEMix* contains the number of the component in a mixture of monatomic

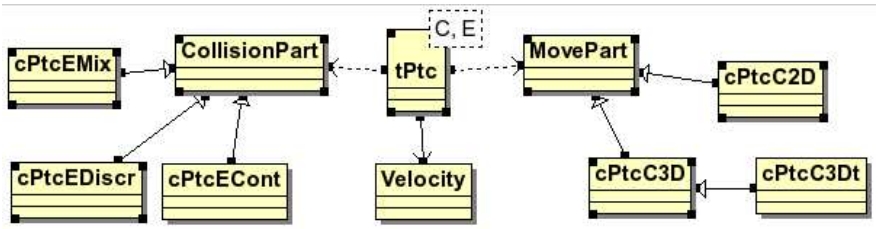

FIGURE 1. Particle structure

gases, and the classes *cPtcEDiscr* and *cPtcECont* are used in a mixture of molecules and, in addition to the number of the component, contain internal properties for molecules with a discrete and a continual description of internal energy, respectively.

The transfer stage operates only with the Move part and velocity, and collisions are simulated with the use of velocity and the Collision part. As both parts are absolutely independent of each other, it is possible to combine various models and algorithms of transfer (for 1-D, 2-D, 3-D, and Axisymmetrical cases) and various models of collisions (elastic collisions, collisions with TR and TV energy transfer, dissociation etc.).

TRAJECTORY INTERSECTION WITH THE BODY SURFACE

At the stage of particle transfer, the trajectory of each particle has to be checked for intersection with the surface of the model under study. As there is a large number of particles and the surface can have a sophisticated shape, the efficiency of the 3D code depends on the time needed to find the point of intersection between the particle trajectory and the surface. The surface of a spacecraft is described by a set of triangles. The search for the intersection point with the spacecraft surface is reduced to a check of trajectory intersections with all triangles and the choice of the nearest one to the start point.

Point of intersection with the plane of the triangle

Let us consider how the event of particle-trajectory intersection with a surface triangle is checked. As the particle trajectory between the collisions is a straight line, it is convenient to write the equation of motion in a parametric form

$$\vec{p} = \vec{p_0} + \vec{V} \cdot t \tag{1}$$

where $\vec{p_0}(x_0, y_0, z_0)$ is the initial particle position, $\vec{V}$ is the particle velocity, and $\vec{p}(x, y, z)$ is the particle position at the time t. The triangle is defined by three vertices: $\vec{v_1}$, $\vec{v_2}$, and $\vec{v_3}$. The triangle plane is described by the equation

$$N_x \cdot x + N_y \cdot y + N_z \cdot z + D = 0 \tag{2}$$

where $\vec{N}$ is the normal vector of the triangle plane and D is a coefficient:

$$D = -\vec{N} \cdot \vec{v_1}; \quad \vec{N} = \vec{a} \times \vec{b}; \quad \vec{a} = \vec{v_2} - \vec{v_1}; \quad \vec{b} = \vec{v_3} - \vec{v_1}; \tag{3}$$

Equations 1 and 2 yield the time of particle/plane intersection and coordinates of the intersection point:

$$t_c = -\frac{\vec{N} \cdot \vec{p_0} + D}{\vec{N} \cdot \vec{V}}; \quad \vec{p_c} = \vec{p_0} + \vec{V} \cdot t_c \tag{4}$$

Checkup of intersection-point location inside the triangle

For the triangle, one can construct a basis on the vectors $\vec{a}$, $\vec{b}$, and $\vec{N}$ (see Eq. 3 and Fig. 2). Let the coordinates of a certain point $\vec{p'}(x', y', z')$ in this basis be known. The coordinates of this point in the global coordinate system can be obtained as follows:

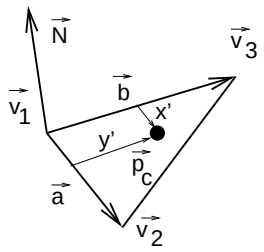

FIGURE 2. Intersection with a triangle

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_x & b_x & N_x \\ a_y & b_y & N_y \\ a_z & b_z & N_z \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} + \begin{pmatrix} v_{1_x} \\ v_{1_y} \\ v_{1_z} \end{pmatrix} \quad \text{or} \quad \vec{p} = A\,\vec{p'} + \vec{v_1}$$

Correspondingly, the point coordinates in the global coordinate system being known, they can be converted to the triangle basis as

$$\vec{p'} = A^{-1}\,(\vec{p} - \vec{v_1}) = A^{-1}\vec{p} - A^{-1}\vec{v_1} = A^{-1}\vec{p} + \vec{c} \tag{5}$$

where A^{-1} is the matrix inverse to A, which can be readily found by Kramer's technique, and $\vec{c} = -A^{-1}\vec{v_1}$ is the vector that takes into account the displacement of the triangle basis with respect to the global coordinate system. The point of trajectory/triangle intersection $\vec{p}_c$ being known (see Eq. 4), it can be transformed to the triangle basis $\vec{p'_c}$. As $\vec{p}_c$ is in the triangle plane, the coordinate z' is always zero, and it need not be transformed. Thus, we obtain

$$\begin{aligned} x'_c &= \vec{ort}_x \cdot \vec{p}_c + c_x \\ y'_c &= \vec{ort}_y \cdot \vec{p}_c + c_y \end{aligned}$$

where $\vec{ort}_x$ and $\vec{ort}_y$ are the first and second rows of the matrix A^{-1}, respectively, and c_x and c_y are the components of the vector $\vec{c}$ (see Eq. 5). The values of $\vec{N}$, D, $\vec{ort}_x$, $\vec{ort}_y$, c_x, and c_y are computed for each triangle before the computation is started and are stored in the computer memory. The condition necessary for the point $\vec{p'_c}$ to be located inside the triangle is

$$\begin{aligned} x'_c &> 0 \\ y'_c &> 0 \\ x'_c + y'_c &< 1 \end{aligned}$$

Thus, the following algorithm is used to verify the trajectory/triangle intersection event:

1. if $\vec{N} \cdot \vec{V} = 0$, the trajectory is parallel to the plane, and there is no intersection.
2. $t_c = -(\vec{N} \cdot \vec{p}_0 + D)/(\vec{N} \cdot \vec{V})$; $\quad \vec{p}_c = \vec{p}_0 + \vec{V} \cdot t_c$
3. $x'_c = \vec{ort}_x \cdot \vec{p}_c + c_x$; $\quad y'_c = \vec{ort}_y \cdot \vec{p}_c + c_y$
4. verification of the condition: $x'_c > 0$; $\quad y'_c > 0$; $\quad x'_c + y'_c < 1$

Trajectory tracing algorithm

Among the entire set of intersected triangles, the trajectory intersects the closest triangle, i.e., the triangle with the minimum value of t_c. If the consecutive check of triangles shows that t_c of the current triangle is greater than t_c of the previously checked triangle, no further verification of intersection for the current triangle is needed, because the particle cannot reach the current triangle.

Since the number of triangles is rather large, the check of all triangles takes a long time. This procedure can be significantly accelerated by reducing the number of triangles to be checked. For this purpose, the model is inscribed

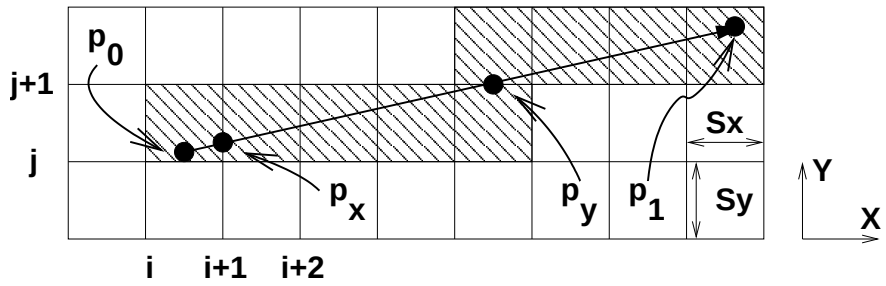

FIGURE 3. Search for cells along the trajectory

into a parallelepiped. All sides of this parallelepiped are perpendicular to the coordinate axes and are located at a minimum distance from the body. A uniform rectangular *tracing* mesh is constructed inside this parallelepiped. For each cell, the triangles located there are determined. This mesh is independent of the collision mesh and is nested inside the latter. If the trajectory does not cross the parallelepiped of the tracing mesh, the particle is shifted without being checked for intersection. For trajectories inside the parallelepiped (at least partially), cells through which the trajectory passes are determined. Only the triangles located in these cells are checked for intersection.

To find cells through which the trajectory passes, a specially developed *tracing algorithm* is used. The idea of the algorithm is a faster computation of the time of intersection with a cell side and transition to a new cell.

Let us consider this algorithm by an example of a 2D problem. There is a uniform rectangular mesh with a cell size S_x and S_y; the initial location of the particle p_0 and the final location p_1 of the particles after a prescribed time step are assumed to be known (see Fig. 3). The task is to find consecutively all cells through which the segment connecting these points passes (hatched region).

Let the initial point be located in the cell (i, j). If there are no triangles in this cell or there are no intersections with triangles of this cell, it is necessary to pass to the next cell. Let us calculate the time counters t_x and t_y for trajectory intersection with the nearest X and Y boundaries of the horizontal and vertical layers of cells where this cell belongs (points p_x and p_y):

$$t_x = (X_{i+1} - P_{0x})/V_x \qquad t_y = (Y_{j+1} - P_{0y})/V_y$$

In this example, we have $t_y > t_x$. It is seen from the figure that the trajectory does not enter the next vertical layer until the time t_y, and it suffices to increase the horizontal index by unity to obtain the index for the next cell. The time when the trajectory leaves the new cell is $t_x^\star = t_x + D_{tx}$, where $D_{tx} = S_x/V_x$ is a constant, because the trajectory is a straight line and the mesh is uniform; it is calculated prior to tracing. Sampling of cells in the X direction is terminated at $t_x > t_y$. Then there follows the transition to the next Y layer: $j^\star = j + 1$, $t_y^\star = t_y + D_{ty}$, where $D_{ty} = S_y/V_y$, like D_{tx}, is a constant and is calculated in advance. Obviously, now we have $t_x < t_y$, and the next cell is again taken in the X direction. The tracing process is finalized when both time counters t_x and t_y become greater than the time step (a cell containing the point p_1 is reached) or the trajectory leaves the tracing mesh. In a 3D case, it is necessary to calculate three times and sort these values in increasing order: the X coordinate will be the coordinate with the smallest time, and the Y coordinate will be the coordinate with the middle time.

The algorithm works most rapidly when the trajectory is parallel to coordinate axes. Figure 4 shows the time of calculating the flow around a sphere at different angles of attack and mesh splitting. A sphere is chosen because its aerodynamic characteristics are independent of the angle of attack. Owing to the symmetry of the sphere, the number of cells is identical in all coordinate directions. The time needed to check for intersection with triangles (Cross, thin line) decreases with increasing number of cells, because the number of triangles in the cells becomes smaller. This time is weakly dependent on the angle of attack. The time needed for tracing (Trace, dush line) increases linearly with increasing number of cells and strongly depends on the angle of attack, as it could be expected. The longest time is needed for 45°, with transitions from horizontal to vertical layers at each step. The sum of these times is the total computation time (Total, thick line). The minimum total time suggests that the optimal mesh size for these computations is $40 \div 50$ cells along each axis. The position of this minimum depends on the geometry and cannot be fixed for all computations. Thus, the algorithm is sensitive to the free stream direction and mesh dimension.

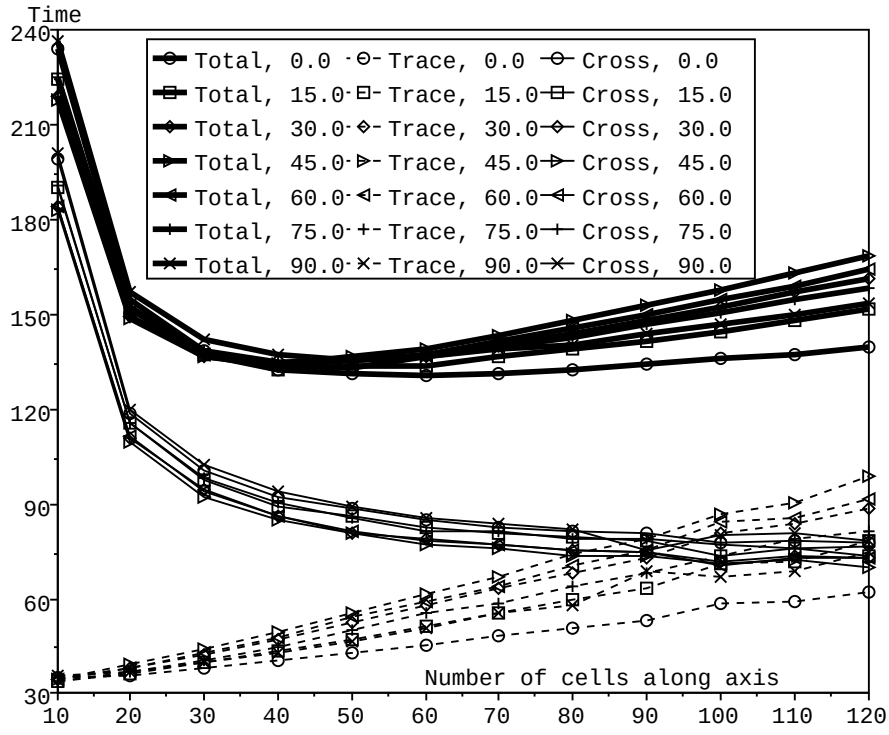

FIGURE 4. Effect of the angle of attack and the number of cells on computation time.

Cell size determination

Automatic selection of the number of cells along each axis of the tracing mesh implies the allowance for the size of the model and surface triangles. The triangles may be extended along some axis, which makes extended cells preferable. In the case of a flat plate perpendicular to the X axis, there must be a single cell along the X axis. Since this plate will always lie within one layer of cells, all other X layer cells will be empty. In the presently implemented method, the cell size proportional to the mean size of triangles along each coordinate axis is used:

$$S_i = \frac{\sum_{j=1}^{N_\triangle} |v1_{ij} - v2_{ij}| + |v2_{ij} - v3_{ij}| + |v3_{ij} - v1_{ij}|}{3N_\triangle} \cdot K; \qquad M_i = L_i / S_i$$

where i is one of the three coordinates (X,Y,Z); j is the triangle index; $N_\triangle$ is the number of triangles; L_i, S_i,and M_i are the computational domain size, cell size, and number of cells along the coordinate i, respectively; K is the scale factor for mesh storage in a prescribed memory. This factor is defined for choosing the cell size so that the information stored there does not saturate the memory arrays allocated. Initially, $K = 1$, but if the number of cells is $M_{cell} = M_x \cdot M_y \cdot M_z > M_{max}$, then $K = \sqrt[3]{M_{max}/M_{cell}}$. M_{max} is set by the user.

The proposed method for determining the cell size suggested a 75x73x73 mesh for the sphere described above, which yields a computation time of 148.8 s for an angle of attack 45°. This is approximately 10% greater than the computation time (134.7 s) with the optimal mesh size (40x40x40). Hence, such an algorithm offers automated selection of the mesh size, and the computation time is only slightly longer than the time of computations on the mesh of the optimal size, which cannot be constructed without some additional special procedures.

VERIFICATION OF THE 3D CODE

The algorithms described have been implemented in a 3D Object-Oriented code. For verification of this code, the flight of the Apollo capsule at an altitude of 135 km was computed and compared with SMILE [4] computations. The

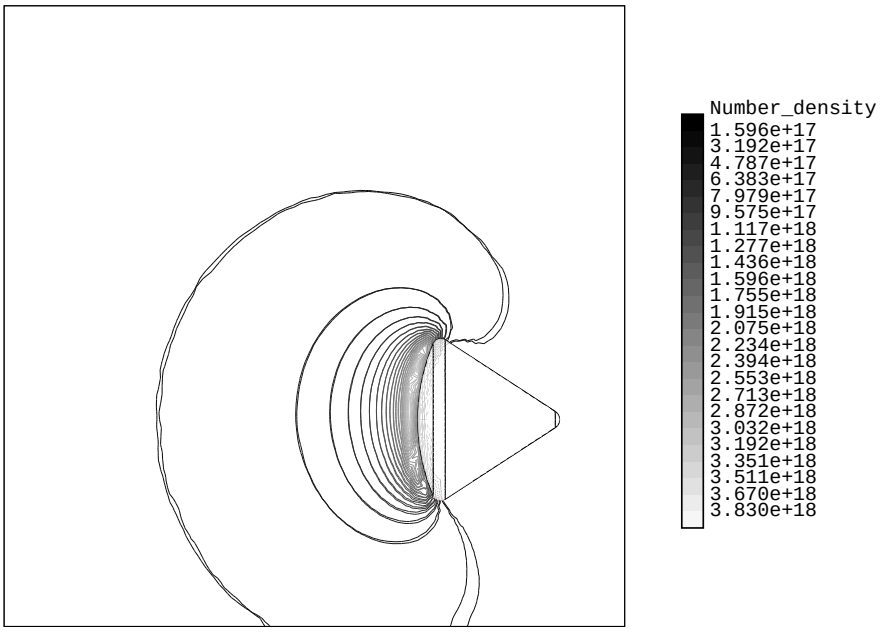

FIGURE 5. Superposition of isolines for SMILE and the present code

velocity was 9600 m/s, and the angle of attack was -25^o. Figure 5 shows a superposition of density isolines in the symmetry plane for these computations. The test shows that the presented code is faster than SMILE (up to 10%).

CONCLUSIONS

The developed set of particle classes allows easy implementation of a multidimensional Object-Oriented DSMC code. The presented algorithms of cell-size determination and trajectory tracing for a 3D case show good efficiency. The developed code is fast enough but still have room for modernization and acceleration.

ACKNOWLEDGMENTS

The research conducted at ITAM was performed under the ISTC Project 2298p and RFBR Project 06-08-00687. This support is gratefully acknowledged.

REFERENCES

1. G.A. Bird,*Molecular Gas Dynamics and the Direct Simulation of Gas Flows*, Clarendon Press, Oxford, 1994.
2. A.Kashkovsky, G.Markelov and M.Ivanov, An Object-Oriented Software Design for the Direct Simulation Monte Carlo Method, AIAA 2001-2895.
3. A.V. Kashkovsky, Ye.A. Bondar, G.A. Zhukova, M.S. Ivanov, S.F. Gimelshein, Object-Oriented Software Design of Real Gas Effects for the DSMC Method, Rarefied Gas Dynamics: 24th Int. Symp., Porto Giardino, Italy, July 10-16, 2004, (Ed. M.Capitelly), AIP Conference proceedings, Vol. 762, 2005, pp.583-588.
4. M.S. Ivanov, G.N. Markelov, S.F. Gimelshein, Statistical Simulation of Reactive Rarefied Flows: Numerical Approach and Applications. AIAA Paper 98-2669, Albuquerque, June 1998.