

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**BOLTZMANN: AN OBJECT-ORIENTED PARTICLE
SIMULATION PROGRAMMING SYSTEM**

Xinming Allen Lin

**November 1990
CSD-900033**

UNIVERSITY OF CALIFORNIA

Los Angeles

Boltzmann: An Object-Oriented Particle Simulation Programming System

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Xinming Allen Lin

1990

© Copyright by

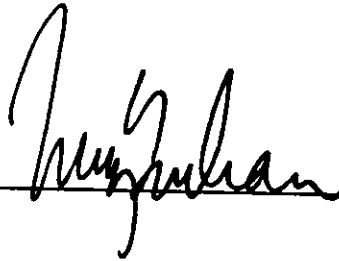
Xinming Allen Lin

1990

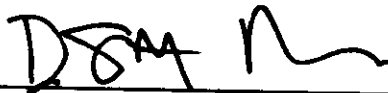
The dissertation of Xinming Allen Lin is approved.



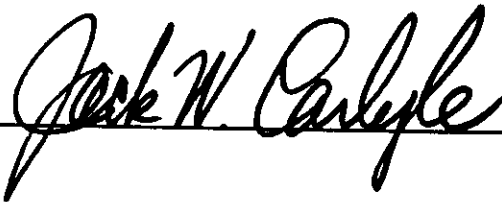
Nasr M. Ghoniem



Tony F. Chan



D. Stott Parker



Jack W. Carlyle



Walter J. Karplus, Committee Chair

University of California, Los Angeles

1990

To my wife Frances Fen-Fang

TABLE OF CONTENTS

1	Introduction	1
1.1	Introduction	1
1.2	The Objective	3
1.3	Literature Survey	5
1.4	Introduction to Particle Simulation	6
1.5	Introduction to Object-Oriented Programming	7
1.6	The Object-Oriented Particle Simulation Methodology	8
1.6.1	A Programming Framework	8
1.6.2	The Particle Class	10
1.6.3	The Scheme Class	12
1.7	Dissertation Scope and Organization	13
2	Vector Arithmetic, Random Variables, and Unit Conversion	15
2.1	Vector Arithmetic	15
2.2	Random Variables	17
2.2.1	Uniform Random Variables	17
2.2.2	Normal Random Variables	17
2.2.3	Maxwellian Random Variables	17
2.2.4	Exponential Random Variables	18
2.3	Unit Conversion	18
3	A Particle Hierarchy	20
3.1	Particle Definition	20
3.2	Particle Classification	21
3.3	The Particle Class	22
3.4	Classical Particle Class (Newton)	24
3.5	Laplace Particle Class	25
3.5.1	Coulomb Particle Class	25
3.5.2	Kepler Particle Class	25
3.6	Fluid Particle Class (Euler)	26
3.7	Incompressible Vortex Class	26
3.7.1	Inviscid Vortex Class	27
3.7.2	The Viscous Vortex Class	27
3.8	Particle Derivation	28

4	A Scheme Hierarchy	30
4.1	Introduction	30
4.2	The Scheme Class	31
4.3	The Particle-Particle Scheme	31
4.4	The Particle-Mesh Scheme	34
5	Initialization and Visualization	37
5.1	Initialization	37
5.1.1	Maxwellian Distribution for Velocity	37
5.1.2	Position Displacement of Small Oscillations	38
5.1.3	Particles Under the Influence of Gravity	39
5.2	Visualization	41
5.2.1	Visualization Windows	41
5.2.2	Drawing Functions	44
5.2.3	Diagnoses	45
6	Applications	49
6.1	Two Coupled Pendulums	49
6.2	Wave Propagation in a Rod	51
6.3	Radiation Displacement of Atoms	55
6.4	Particle Simulation of Plasma	56
6.5	Vortex Simulation of Fluid Flow	58
7	Performance Evaluation	62
7.1	Memory Requirement	62
7.2	“Garbage” Collection	62
7.3	Computation Efficiency	62
8	Conclusion	65
8.1	The Contributions	65
8.2	Further Improvement	66
8.3	Implications	67
	References	69
A	Example Programs	73
A.1	Two Coupled Pendulums	73
A.2	Wave Propagation in a Rod	75
A.3	Radiation Displacement of Atoms	76
A.4	Particle Simulation of Plasma	78
A.5	Vortex Simulation of Fluid Flow	80

B Boltzmann 1.0: A User's Manual	82
B.1 Introduction	82
B.2 The OOPS Framework	82
B.3 Vector Arithmetic	83
B.4 Random Variables	85
B.4.1 Uniform Random Variables	85
B.4.2 Normal Random Variables	86
B.4.3 Maxwellian Random Variables	86
B.4.4 Exponential Random Variables	86
B.5 Unit Conversion	87
B.6 A Particle Hierarchy	88
B.6.1 The Particle Class	88
B.6.2 Classical Particle Class (Newton)	90
B.6.3 Laplace Particle Class	91
B.6.4 Coulomb Particle Class	92
B.6.5 Kepler Particle Class	92
B.6.6 Fluid Particle Class (Euler)	92
B.6.7 Incompressible Vortex Class	93
B.6.8 Inviscid Vortex Class	94
B.6.9 The Viscous Vortex Class	94
B.7 A Scheme Hierarchy	94
B.7.1 The Scheme Class	94
B.7.2 The Particle-Particle (PP) Scheme	96
B.7.3 The Particle-Mesh (PM) Scheme	97
B.8 Particle Derivation	97
B.9 Initialization Functions	99
B.9.1 Random Load	99
B.9.2 Hot Particles	99
B.9.3 Oscillating Particles	100
B.9.4 Particles Under the Influence of Gravity	100
B.10 Visualization Windows	100
B.11 Visualization Functions	104
B.11.1 Cartesian Space Visualization	104
B.11.2 Phase Space Visualization	104
B.11.3 Power Spectrum Analysis	105
B.12 A Programming Example	105
C Boltzmann Definition Files	111
C.1 Array.h	111
C.2 Boltzmann.h	113
C.3 Boundary.h	114
C.4 Constant.h	115
C.5 Coulomb.h	115

C.6 Euler.h	116
C.7 Integrator.h	117
C.8 Kepler.h	119
C.9 Laplace.h	120
C.10 Libxs.h	121
C.11 Macro.h	122
C.12 Newton.h	122
C.13 Particle.h	123
C.14 PMscheme.h	126
C.15 PPscheme.h	127
C.16 RanVar.h	129
C.17 Scheme.h	133
C.18 Unit.h	134
C.19 Vector.h	135
C.20 Vortex.h	137
C.21 Swindow.h	138

LIST OF FIGURES

1.1	A programming framework	9
1.2	Basic steps of OOPS programming	9
1.3	A particle hierarchy	11
1.4	A scheme hierarchy	13
4.1	Short-range influence calculation in PP Scheme	32
4.2	A connection	33
5.1	A visualization window	42
5.2	A menu window	43
5.3	A power spectrum output	48
6.1	Two coupled pendulums	49
6.2	Two coupled pendulums	52
6.3	A particle model of a metal rod	52
6.4	Wave propagation in phase-space in a metal rod	54
6.5	Radiation displacement of atoms	57
6.6	Particle simulation of plasma in phase-space	59
6.7	Vortex simulation of fluid flow	61
B.1	A programming framework	82
B.2	Basic steps of OOPS programming	83
B.3	A particle hierarchy	89
B.4	A scheme hierarchy	95
B.5	A visualization window	102
B.6	A menu window	103
B.7	Radiation displacement of atoms	110

LIST OF TABLES

7.1	Memory requirements	63
7.2	Comparison of time consumptions	64
7.3	Real-time spent in the examples	64

ACKNOWLEDGEMENTS

I am grateful to many people for their help and support at various stages of the research. First of all, I want to express my greatest appreciation to my doctoral advisor and committee chairman, Professor Walter J. Karplus, for his kindest support both academically and financially, his willingness to grant me the freedom to pursue the research in my own style, and his supervision during the entire years of my study at UCLA. Also, I want to thank my other committee members for their suggestions and criticisms, particularly Professor Tony F. Chan and Professor Nasr M. Ghoniem for introducing me to particle methods, Professor Jack W. Carlyle for his kind encouragement and support, and Professor D. Stott Parker for his thoughtful criticism. I want to thank the following people for their contributions: Dr. S. Philip Chou to the simulation of radiation displacement of atoms, Professor John M. Dawson to the plasma simulation, and Professor Russel Caffisch and Professor Christopher R. Anderson to the vortex simulation of fluid flow.

Next, I want to thank all my officemates for sharing the same office and helping me in many ways: E. Robert "Bob" Tisdale for generously spending his time and expertise in various phases of the research, and Dr. Alex Pang, Han-Sen Dai, and Dr. Charles Tong for all the discussions. I want to thank June Myers for her administrative work related to the research, Doris Sublette for her excellent service as the department librarian, and Robert Collins for his voluntary maintenance of Free Software Foundation's software, including GNU C++ compiler and InterViews user-interface toolkit. It is impossible to list everybody by name who has helped; to each one of you, thank you. At last, but not the least, I want to thank my loving wife for sharing the same excitement and frustration with me. To her, the dissertation is dedicated.

The research was funded in part by TRW Space and Technology Group and the State of California under the MICRO program and Lawrence Livermore National Laboratory under the grant UCLLNL B056074.

VITA

- 1982 B.S., Computer Science and Engineering
 Changsha Institute of Technology
 Changsha, Hunan, People's Republic of China
- 1985 M.S., Computer Science
 University of California
 Los Angeles, California
- 1987 Summer Institute in Parallel Computing
 Argonne National Laboratory
 Argonne, Illinois
- 1985-1987 Research Assistant
 Computer Science Department
 University of California, Los Angeles
- 1988 Teaching Assistant
 Computer Science Department
 University of California, Los Angeles
- 1988-1990 Post-Graduate Researcher
 Computer Science Department
 University of California, Los Angeles

PUBLICATIONS AND PRESENTATIONS

- Lin, X. A., and Karplus, J. W., *Doing Physics Simulation in the Boltzmann Programming System*, Object-Oriented Simulation, edited by Raimund K. Ege, the Society for Computer Simulation, 1991.
- Lin, X. A., and Karplus, J. W., *An Object-Oriented Particle Simulation Programming System*, Presented at the 2nd International Conference on Expert Systems for Numerical Computing, Purdue University, Indiana, April 1990, to be published in the Proceedings.
- Lin, X. A., and Karplus, J. W., *Logic Programming for Numerical Computation*, Technical Report CSD-890042, Computer Science Department, University of California, Los Angeles, California, June 1989.
- Lin, X. A., Chan, T. F., and Karplus, J. W., *The Fast Hartley Transform on the Hypercube Multiprocessors*, Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, pp. 1451-1454, California Institute of Technology/ Jet Propulsion Laboratory, California, February 1988.
- Lin, X. A., and Karplus, J. W., *Overlapping Communications with Computations in Static Load-balancing*, Proceedings of the 4th SCS Multiconference on Multiprocessors and Array Processors, pp. 111-116, SCS International, California, January 1988.

ABSTRACT OF THE DISSERTATION

Boltzmann: An Object-Oriented Particle Simulation Programming System

by

Xinming Allen Lin

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1990

Professor Walter J. Karplus, Chair

An object-oriented programming methodology for particle simulations is developed. It is established on the *reductionist* view that many physical phenomena can be reduced to many-body problems. By doing the reduction, many seemingly unrelated physical phenomena can be simulated in a systematic way and a high-level programming system can be constructed to facilitate the programming and the solution of the simulations. In the object-oriented particle simulation methodology, a hierarchy of abstract "particles" is defined to represent a variety of characteristics in physical system simulations. A simulation program is constructed from particles derived from the abstract particles. The object-oriented particle simulation methodology provides a unifying modeling and simulation framework for a variety of simulation applications with the use of particle methods. It allows easy composition of simulation programs from predefined software modules and facilitates software reusability. It greatly increase the productivity of simulation program constructions.

Boltzmann (after Ludwig Boltzmann, 1844-1906) is a prototype programming system in the object-oriented particle simulation methodology. Boltzmann is implemented in C++ and the X Window System. It contains a library of data types and functions that support simulations in particle methods. Moreover, it provides a visualization window to support friendly user-computer interaction. Examples of the application of the Boltzmann programming system are presented. The effectiveness of the object-oriented particle simulation methodology is demonstrated. A user's manual is included in the appendix.

CHAPTER 1

Introduction

1.1 Introduction

Computers have been used widely in solving science and engineering problems, the main driving force behind the development of the first mechanical computers in the earlier centuries and the first electronic computers in the forties [22, 16]. They were first used to evaluate functions such as additions, multiplications, and logarithmic tables. As the power of computers grew, they were used in more complicated computations such as the solution of differential equations to simulate continuous-time physical systems and real-time system control. In computer simulation, finite-difference and finite-element methods, among others, were successfully applied to solve systems of ordinary differential equations (ODEs) and partial differential equations (PDEs). Computer simulation through the solution of a system of differential equations has been and will be an important methodology in scientific applications of computers, and extensive researches have been done on algorithms [55], programming languages [52, 8], and computer architectures [31].

A relatively newer methodology in computer simulation is the simulation of a system through an ensemble of interacting entities called *particles* that constitute the system and are often identified directly with some physical objects in the system. This is called *particle simulation* [45, 30, 27, 25]. The particle simulation approach has been widely used in plasma physics [18, 6, 54], geophysics [56], semiconductor simulation [9], material science [19, 10], fluid dynamics [47], molecular dynamics [1, 48], and many other fields. The advantages of doing particle simulation over differential equation methods such as the finite-difference and finite-element methods include

1. The particle concept is fundamental in many physical sciences and engineering. If the concept of particles exists already in the physical problem, it is natural to use the particle simulation approach.
2. Many physical problems have very simple laws governing the microscopic behavior of the individual particles but very complicated collective macroscopic phenomena. The differential equations may be too complicated to be solved either analytically or numerically.
3. Particle simulation uses a different approach to handle boundary conditions, and for some problems that approach is easier than that used in the differential equation methods.

4. The procedure of particle simulation is very much independent of specific applications. Therefore, standard methods and algorithms can be made as modules that can be used later without re-implementation.

The disadvantages of particle simulation include

1. The number of real physical particles are usually too large to be implemented on any computer. Approximation is usually required so that this large number of real particles can be represented by a relatively small number of super-particles.
2. Some physical problems are better described by continuous equations. In order to apply the particle simulation method to them, a discretization of the problem domain is required. This discretization, if not carefully carried out, may introduce errors that produce non-physical properties and even instability.

Particle simulation therefore provides a powerful method that can solve many problems that the differential equation methods have found difficult or inconvenient, provided it is used with care.

The preparation of a particle simulation differs from that of the differential equation methods in the following aspects:

1. The differential equation methods employ a macroscopic description of the *whole system* changing in time, whereas particle simulation employs a microscopic description of the *parts* that constitute the system changing in time in relation to each other. The description in differential equation methods is often the state equation of the system, whereas the description in particle simulation is often the equation of motion of individual parts. Therefore, using particle simulation one needs only to specify the behavior of one particle and its interaction with other particles. The collective result will be generated automatically.
2. While different classes of problems require very different approaches in the differential equation methods, most particle simulations take the same or similar approaches and procedures. Only the definitions of the particles determine the differences among different particle simulation applications. Therefore, the common models, schemes, and algorithms can be made to be used again and again. Thereby, a great amount of effort can be saved in programming particle simulations.
3. The description in differential equation methods is continuous in space and requires further discretization, whereas the description in particle simulation is already discrete in space.

4. The outputs in differential equation methods are often the direct solution of the state equations, whereas the outputs in particle simulation are often averaged values over different parts. In particle simulation, different outputs can be generated from the same simulation run. Therefore, visualization in particle simulation can be made to meet a variety of requirements.

Because of the unusual characteristics of particle simulations, an object-oriented particle simulation (OOPS) approach is proposed to allow easy composition of particle simulation programs and to facilitate software reusability. A prototype programming system, Boltzmann, is developed to demonstrate the OOPS methodology. In section 1.2, the objective of the proposed research is defined. A literature survey of related research is presented in section 1.3, along with an introduction to particle simulation and to object-oriented programming in section 1.4 and section 1.5. Then, the OOPS methodology is described in section 1.6. Section 1.7 discusses the scope and the organization of the dissertation.

1.2 The Objective

The overall objective is to develop the OOPS methodology and to identify and solve problems in the design and development of OOPS systems. This objective includes the following sub-objectives:

1. Identify the fundamental framework that supports most particle simulation applications. Identify the basic methods, and algorithms that have been used repeatedly in different particle simulation applications. Identify the most commonly required outputs from particle simulations.
2. Define the OOPS methodology. Define a programming framework for particle simulations. Define the partition of functionalities required by the framework. Define the basic procedure of OOPS programming.
3. Design the Boltzmann programming system. Design a hierarchical organization of the abstract *particles* and the computational *schemes* that are independent of specific applications. These particles and schemes will serve as templates for more detailed particles and schemes defined by the users in particle simulation applications. Design the functions of the particles and schemes and their interfaces with each others and with the outside world. Design the necessary routines that facilitate the initialization, control, and the visualization of particle simulations. Design a user interface that allows easy human-machine interaction.
4. Demonstrate the effectiveness of the OOPS methodology through the application of the Boltzmann programming system to scientific and engineering problems. Evaluate the performance of the Boltzmann programming system

and compare it with traditional programming languages in particle simulation.

The importance of the OOPS approach can be understood in light of the following aspects:

1. The OOPS approach permits modularity (*Integrated Software*). Simulations are constructed from carefully designed building blocks like particles and schemes, instead of from low-level description like numbers and numerical arrays. It permits the users to concentrate on the high-level physical concepts instead of on the detailed low-level implementation.
2. The OOPS approach facilitates software reusability (*Software Mass Production*). Frequently-used models, methods, and algorithms are made as object classes that can be used repeatedly without the necessity of re-implementing them. *Class derivation* allows new classes to be defined on top of existing classes with minimum effort.
3. The OOPS approach has a standard procedure of programming. Particle simulation programs are much easier to be composed and much shorter in the OOPS approach than in the traditional approaches.
4. The OOPS approach is independent of computer architectures. This is important because it allows efficient implementations on a variety of computer platforms. Algorithms that implement the classes can be selected carefully and optimization in the implementation has to be done only once. It can also take the advantages of special computer architectures such as multi-processor computers by implementing the classes accordingly. The special architectures are transparent to the users.
5. An OOPS programming system is extensible. Users can replace the pre-defined classes with their own definitions. Since classes are organized hierarchically, new classes can be added easily. Specification of an OOPS programming system in some specialized areas of study can be done by attaching specialized classes of particles and schemes to the existing hierarchies.

The challenge of the OOPS approach is to design particle and scheme classes that capture a variety of simulation applications using particle methods. They should be general enough to be used by different applications but flexible enough to allow easy specialization. The fact that some applications do not render themselves easily to the object-oriented paradigm, while other applications are best represented in it, requires careful tradeoff between ease of programming, software reusability, and efficiency, and iterative refinements of the design of particles and schemes. The research will benefit scientists and engineers who are interested in using particle simulation methods to solve their problems but are not satisfied with the traditional ways a simulation program is constructed.

1.3 Literature Survey

Particle simulation was pioneered by early studies of particle motions in vacuum tubes [29] and plasmas [7, 17] and molecular dynamics [1, 48]. Good introductions to the subject can be found in the books [45, 30, 27, 25]. Most papers on particle simulation discuss the formulation of mathematical models and numerical algorithms to solve them. However, few discuss methodologies for the transformation from mathematical models and numerical algorithms to computer programs that are readily to be run. It gives the impression that the transformation is a straightforward, although tedious, step-by-step process. That impression is misleading, given the size and the complexity of most particle simulations.

One exception to the lack of effort on programming methodologies for particle simulations is the OLYMPUS programming system that has been in use by several simulation groups, particularly in the plasma simulation community ([30, chapter 3], [40]). It is a methodology to assist the transformation from numerical algorithms to computer programs and the maintenance of computer codes. More specifically, it is a top-down structural programming methodology composed of a programming environment, conventions for programs and data structures, and a library of utility subroutines. OLYMPUS helps to organize programs and data structures, but it does not offer to improve the way programs are constructed. Programs in OLYMPUS are still constructed by interfacing a number of subroutines in the same way as programs in FORTRAN do.

Another effort on programming methodologies for particle simulation is an interface system developed at Jaycor Corporation for a specialized particle simulation program called MAGIC [20]. MAGIC is a two-dimensional finite-difference FORTRAN program for Particle-In-Cell plasma simulation [24]. The interface system organizes input parameters about a simulation into high-level structures which are called "objects". The objects can then be manipulated in a number of ways such as being displayed and being modified. A rule-base of constraints enforces restrictions on an object. A symbol manipulator generates inputs to the MAGIC program from its collection of objects. The approach taken by the interface system is good to specialized particle simulation programs as it rightly claims. However, different applications would require different interface systems. Therefore, the approach ignores the similarities among different applications in particle simulation and the issue of software reusability. In addition, the interface system is only object-based but not object-oriented simply because it lacks the major characteristics of object-oriented programming such as function definitions in objects, class inheritance, and automatic binding (polymorphism).

The proposed object-oriented particle simulation approach is new. However, the concept of object-oriented programming and design has been around for a while. Object-oriented programming ideas first appear in the programming language Simula in the sixties [15], a language based on Algol 60 and often used for simulation purposes. They were later carried forward in the programming lan-

guage Smalltalk [21], which is one of the main achievements that are responsible for the popularity of object-oriented programming. Other popular object-oriented programming languages include Eiffel [42] and C++ [53]. Early examples of object-based software designs include the Macintosh Toolbox [3], Hypercard [23], and the X Window System [50]. There are a few examples of real object-oriented software designs, among which are a number of user-interface toolkits [51, 44, 38]. Most of the software designs are within the domains of traditional computer science. Few address issues in scientific computing [49, 35]. None, as far as the author knows, offers a systematic object-oriented methodologies to simulations with the use of particle methods.

1.4 Introduction to Particle Simulation

Particle simulation, according to [30], is a generic term for a class of simulation methods where “the discrete representation of physical phenomena involves the use of interacting particles. The name ‘particle’ arose because in most applications the particles may be identified directly with physical objects. Each particle has a set of attributes, such as mass, charge, vorticity, position, momentum. The state of the physical system is defined by the attributes of a finite ensemble of particles and the evolution of the system is determined by the laws of interaction of the particles.” Particles in particle simulation may have a one-to-one correspondence with particles in the physical systems as in the case of molecular dynamics (*one-to-one particle simulation*). They may have no direct physical meaning when they are introduced by discretizing continuous fluid equations (*one-to-zero particle simulation*). In between, a particle in the simulation may represent many physical particles (*one-to-many particle simulation*). In that case, the particle in the simulation is often called a super-particle. The choice depends upon the problem, the types of the phenomena being investigated, and the computer resources available. For a microscopic description of a small space-scale comparable with the mean inter-particle separation and a small time-scale comparable with the system time-scale such as oscillation or orbital periods, a one-to-one particle simulation may be required. For a description of a relatively large space-scale compared with the mean inter-particle separation and a relatively small time-scale compared the average collision time of the physical system, a one-to-many particle simulation may be appropriate. For a macroscopic description of a much greater space-scale and a much larger time-scale than the physical system, a one-to-zero particle simulation may have to be adopted for the simulation to be meaningful. Examples of one-to-one particle simulations include the study of solid state and stellar clusters. One-to-many particle simulations are more often seen in the study of collisionless plasma and semiconductor devices. The simulation of continuous incompressible fluid, where the vortex elements are treated as particles, is an example of one-to-zero particle simulation.

Particle simulation methods also differ in the underlying *computational scheme*

that specifies how the interaction between particles is calculated. The particle-particle (PP) scheme calculates the force on a particle by adding up the forces created by other particles. The particle-mesh (PM) scheme calculates the force on a particle by computing the potential field over a mesh and then interpolating the force on the particle. The PP scheme is conceptually simpler but computationally more expensive than the PM scheme if a very large number of particles are involved. The PM scheme is generally much faster but less accurate than the PP scheme because of its limited resolution. However, when the wavelengths of importance of the physical system is much larger than the spacing of the grid points of the mesh, the PM scheme can result in a fairly accurate simulation. A hybrid scheme is the multipole expansion (ME) scheme [26, 2], which is a combination of the PP scheme and a multigrid PM scheme. The ME scheme uses the PP scheme to calculate the fast-varying short-range forces and uses the multigrid PM scheme to calculate the slowly-varying long-range forces. It retains the accuracy in calculating the short-range forces and avoids the computational complexity in calculating the long-range forces for many particles.

Each scheme can employ different algorithms for each function it performs. For instance, solving the algebraic field equations in the PM scheme can be accomplished using algorithms ranging from direct methods to iterative methods to special fast solvers. It is the large number of considerations in a particle simulation (actually in any simulation) which complicates the implementation. And it is the complexity involved in a particle simulation and the similarity of schemes and algorithms among different simulations that inspired the idea of building an object-oriented particle simulation system.

1.5 Introduction to Object-Oriented Programming

Object-oriented programming is a programming paradigm that views programming as composition of objects [34, 12, 14, 41, 57]. It contains a number of concepts and characteristics, which include

1. **Class definition:** object-oriented programming allows the definition of classes which not only have memory storages for data members but also define the operations that can be applied to the data members. An instance, or object, of a class is then defined functionally, having its behavior encoded in itself. Unlike in traditional programming, parameters about an object and initialization of the data structures can all be coded in the same object.
2. **Class inheritance:** in object-oriented programming, a new class can be defined to inherit properties from an old class. This allows two similar objects to share the same properties without re-defining the second object.
3. **Automatic binding (polymorphism):** function calls or operations can be made generic so that they resolve to the right operations according to the the

types of the parameters or operands. This allows the user to associate the conceptually-similar operations together without worrying about calling the right functions. The compiler takes care of type resolution automatically.

In the context of particle simulation, a class defines not only data members such as position and velocity but also the interactions within the class and outside the class under certain conditions. For instance, an electron may define charge and mass as well as position and velocity as its data members and repulsion of other electrons and attraction to positively charged particles as its member functions. When two electrons are put in the same field, they will be sure to repel each other, relieving the user from worrying how to update the states of the electrons. Moreover, the user may not like the algorithms coded in the definition of an electron. He is then free to define a new electron derived from the old one with a few algorithms replaced by his own. Then, the new electron will have a behavior identical with the old one except in those attributes that have been replaced with the new definitions.

1.6 The Object-Oriented Particle Simulation Methodology

1.6.1 A Programming Framework

After careful studies of many particle simulation methods and applications, a pattern in the development of particle simulations is identified. In every particle simulation, some kinds of particles are defined that represent the physical system collectively. Associated with every particle are its state variables such as position, velocity, momentum, and energy that represent it in the state space, and equations of motion that determine its state in the future. The equations of motion can be either deterministic as in Newton's second law or stochastic as in Monte-Carlo simulations, or a combination of both. A particle is driven to its next state (assuming time is discretized) according to its equations of motion, which may involve the states of other particles and external conditions. A computational scheme is devised to solve for the unknowns in the equations of motion and to synchronize the advancement of the particles. The computational scheme is often independent of the specific particles the simulation may use.

Therefore, a programming framework is developed to capture the similarities among different particle simulation applications. The framework is composed of two abstract classes: a Particle class and a Scheme class (Figure 1.1). The Particle class is a class of particle templates that can be tailored to specific applications. The Scheme class is a class of computational schemes that will coordinate the particles through their change of states in simulations. Another way to interpret the two classes is that the Particle class is responsible for local computations such as integration, while the Scheme class is responsible for global computations such as force calculation. The framework is also called a "Republic" framework, for the structures of particles decide the structures of schemes and the schemes, in

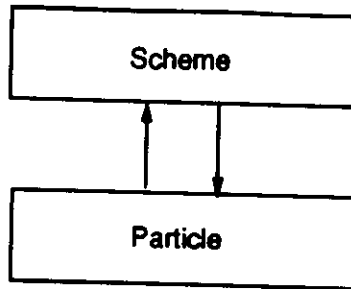


Figure 1.1: A programming framework

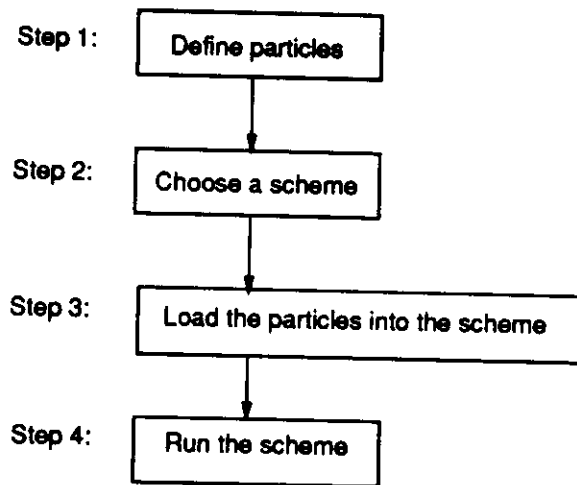


Figure 1.2: Basic steps of OOPS programming

turn, *supervise* the particles in simulations. Contrary to a possible “Democratic” framework where particles are “cellular” automata, the Republic framework has the advantage of efficiency and the flexibility of capturing almost any computational schemes that exist in particle simulations. The significances of the programming framework for OOPS include that the same scheme can supervise different classes of particles and that the same particles can work under different forms of schemes.

An instance of a class is called an object of the class. Particle simulation in the programming framework is simple. A combination of some Particle objects and a Scheme object with the use of special-purpose *OOPS functions* constitutes a particle simulation. In Figure 1.2, particles are loaded into a scheme (Step 3) before the simulation, which is conducted by running the scheme (Step 4). Visualization of a simulation process can be done either by specifying the particles and the scheme appropriately before the simulation or by calling the visualization functions explicitly after the simulation is finished. The characteristics of particles and schemes can be expanded, modified, and deleted by OOPS functions. In addition, OOPS functions are the only ways allowed to change the definition of particles and schemes and are the interface between objects such as particles and

schemes and the human user who employs them to construct his simulation. They are employed to initialize, control, and visualize a simulation, as well as to perform housekeeping operations.

The particle class and the scheme class contain many *subclasses*. Each subclass differs from its parent class in certain functions. For example, a particle subclass may have an extra state variable or a different set of equations of motion and a scheme subclass may use a different computational scheme or algorithm. A subclass *derives* or *inherits* all its functionalities from its parent class, except wherever a new definition of some functions overrides the old one. All the OOPS functions applicable to a class apply also to its subclasses. Also, associated with a class is a number of alternative functions and parameters that are called the *resource* of the class, which can be used to replace functions and parameters of the class. The resource of a class allows the users to choose from a pool of standard algorithms without having to implement them. The resource of a class can be used only by the class or its subclasses.

One of the major advantages of computer simulation is the ability to visualize the process and to gain insight of the simulated system from the simulation results. A visualization window provides the human user an effective instrument to look into the simulation process and to grasp the meaning of the simulation results more easily. Standard diagnosis and analysis functions that measure and display system outputs are provided as buttons on the visualization window. Different displays of the same simulation are effected simply by clicking different buttons.

1.6.2 The Particle Class

An object in the particle class represents a real particle in the simulation program. The particle class contains many subclasses and different subclasses represent different kinds of particles. Each subclass has a name, a set of attributes, and a set of functions. The class name distinguishes the class from other classes. The attributes contain parameters relevant to the class of particles and their state variables. The functions support the functionalities of the particle class, which may include

- Interaction or influence calculation functions: functions that calculate the interaction between two particles and the influence on the particle from an external field. The term *influence* will be used to represent any kind of interaction, among which force is the most common but other kinds of interaction are also possible. Influence calculation functions include the internal inter-particle influence calculation function and the external influence calculation function.
- Integration function: function that integrates the interactions into the equations of motion of the particle and advances the state of the particle from one time step to the next time step. Integration can have multiple steps.

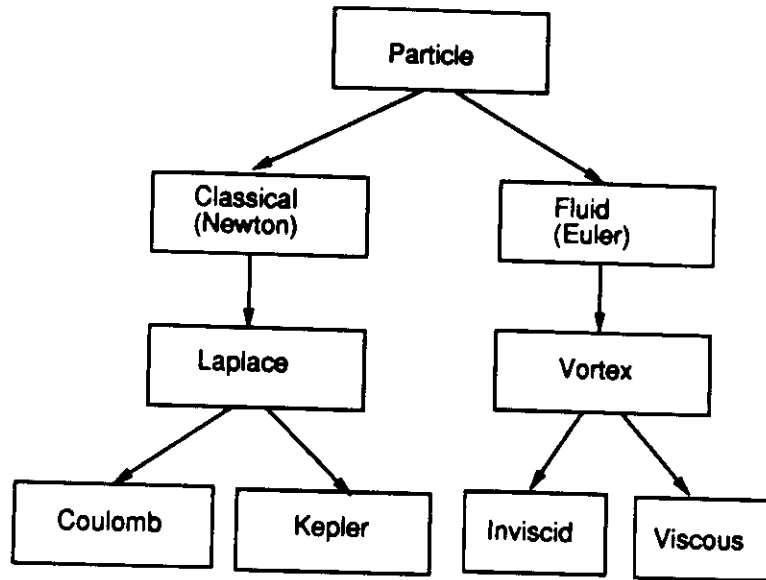


Figure 1.3: A particle hierarchy

- **Display function:** function that provides information about the particle to the outside world. The information may include its relative position in the display coordinate (the true position is determined by the combination of the particle and the scheme which contains information about the space) and its graphic representation (such as shape and color).
- **Miscellaneous functions:** functions that generate a warning message when the particle is used inappropriately and explain the structure of the particle and how it can be used when the particle is inquired.

The core set of functions the Particle class defines the protocol among different classes of particles and between the Particle class and the Scheme class. The significance of separating the functionalities in the Particle class is that every function can be replaced by a new definition without the change of other functions. Since the functions in the Particle class also decide the functions in the Scheme class (the Republic framework), the establishment of a clean and robust protocol, so that later particle classes and scheme classes can be added in without changing the protocol, requires careful design of the functions and iterative refinements.

The Particle class is organized in a tree hierarchy of subclasses where every class, except for the topmost particle class, has a parent class from which most of its properties are inherited. It includes at least two categories: the Classical particles and the Fluid particles to cover two very different classes of particle simulations (Figure 1.3). In the Classical Particle class, particles create force field. Whereas, in the Fluid Particle class,, particles create velocity field. A particle simulation starts with the choice of the particle class which is closest to and more general than the particles to be simulated and by adapting it to the specific requirements.

A particle class is always more general than its subclasses and hence adaptable to more applications, whereas the subclasses are more specific and hence require less effort to adapt when they fit the requirement. Moreover, if a scheme can be applied to a particle class, then the same scheme can be applied to all the subclasses of the particle class. The particle class hierarchy is extensible, that is, new particle subclasses can be attached to the hierarchy.

1.6.3 The Scheme Class

Schemes are particle simulation methods represented as objects. They specify the computational schemes to be used and provide algorithms to facilitate them. Their tasks include

- Maintaining information about the space domain, such as the boundaries and boundary conditions.
- Calculating the influence on each particle, in the methods the computational schemes represent.
- Advancing the particles, that is, running the particles' integration functions.
- Displaying the particles in various forms, with the help of the particles' display functions.

Two most popular computational schemes in particle simulations are the Particle-Particle (PP) scheme and the Particle-Mesh (PM) scheme.

- The Particle-Particle scheme is a computational method that calculates the influence on each particle by adding up all the influences due to the interaction of the particle with other particles. This is a scheme with a computational complexity of $O(N^2)$ for long-range influences and of $O(N)$ for short-range influences, where N is the number of particles.
- The Particle-Mesh scheme is a computational method that solves a field equation for the potential on a mesh over the space and interpolates the influence on each particle from the potentials at the mesh grid points. This is a scheme with a computational complexity of $O(N + M \log M)$, where M is the number of mesh grid points.

Like the particle class, the scheme class is also organized in a tree hierarchy (Figure 1.4). A scheme is always more general than its sub-schemes, that is, if a scheme can be applied to a particle class, then the parent scheme can also be applied to the same particle class, but not necessarily the other way around. The scheme class hierarchy is extensible, that is, new schemes can be attached to the hierarchy. An advantage of the OOPS approach is that the same computational schemes can be applied to different applications, and new computational schemes can be incorporated to work with existing particle classes without changing their definitions.

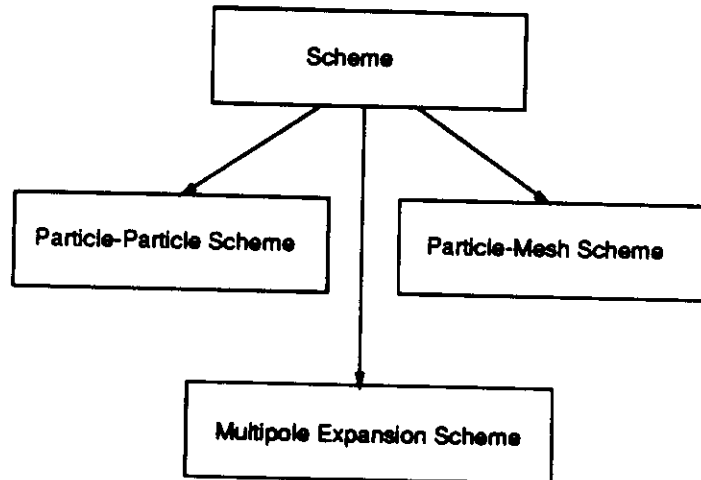


Figure 1.4: A scheme hierarchy

1.7 Dissertation Scope and Organization

The research effort to develop an object-oriented particle simulation programming system consists of the following phases.

1. Design particle classes that characterize phenomena in classical mechanics and electrodynamics, which include, but are not limited to, particle motion in a field, oscillation, particle collisions, many-body interaction, wave propagation, and fluid dynamics.
2. Design scheme classes that represent the computational methods applicable to the particles designed in phase 1. These methods include the action-at-a-distance methods (e.g. Particle-Particle method), the action-at-a-point methods (e.g. Particle-Mesh method), and other methods.
3. Design a user-interface that allows convenient input and output specification. The input includes the initial distribution of the particles and the boundary conditions. The output includes the graphical visualization of various measurements concerning the state of the simulated system, which may include phase-space plot, energy distribution, and power spectrum analysis.
4. Implement Boltzmann in C++ [39, 53], an object-oriented extension to the C programming language [32, 28]. Demonstrate the effectiveness of the OOPS approach by implementing a number of scientific and engineering applications in the Boltzmann programming system.
5. Evaluate the performance of the OOPS approach in comparison with the traditional approaches.

6. Provide a user's manual to facilitate the use of the Boltzmann programming system by engineers and scientists.

The dissertation consists of eight chapters and three appendixes. Chapter 1 is the introduction. Chapter 2 brings in object-oriented vector arithmetic along with random variables and unit conversion, which lay down the prerequisite to the Boltzmann programming system. Chapter 3 and 4 discuss the design of the particle classes and the scheme classes, respectively. In Chapter 5, initialization and visualization are discussed. Chapter 6 demonstrates the applications of Boltzmann to a number of scientific and engineering problems. In Chapter 7, the performance of the Boltzmann programming system is evaluated. Conclusion and further improvements are discussed in Chapter 8. Appendix A contains the Boltzmann programs for the examples in Chapter 6. A user's manual to the Boltzmann programming system is given in Appendix B. The definition files of some of the important classes are listed in Appendix C.

CHAPTER 2

Vector Arithmetic, Random Variables, and Unit Conversion

Vectors are extremely useful in particle simulations because many of the attributes of a particle can be represented by vectors. Operations on vectors are independent of the dimensionality of the problem, which means the same program written in vector arithmetic applies to one, two, and three dimensions without modification. That raises the level of abstraction and reduces the length and the complexity of a program involving vector operations. The Boltzmann programming system is built on top of this vector level. The creation of vectors and vector operations are discussed in this chapter, along with the discussion of random variables and unit conversion. They serve as a prerequisite to the Boltzmann programming system.

2.1 Vector Arithmetic

Vectors are specified in one of the following ways:

```
Vector vector_name(x);
```

```
Vector vector_name(x, y);
```

```
Vector vector_name(x, y, z);
```

which create vectors of one, two, and three dimensions. Vectors that are defined without coordinates, such as

```
Vector vector_name;
```

```
Vector vector_name[N];
```

are treated as zero vectors of either one, two, or three dimensions before they are assigned to other vectors. Vector constants can be specified as

```
Vector(x);
```

```
Vector(x, y);
```

```
Vector(x, y, z);
```

Standard operations on vectors are supported. Operators in the C language are over-loaded with vector operations so that vector arithmetic can be expressed in the same way as float-point arithmetic. Vector operations in the Boltzmann programming system include

- `int dimension(Vector)`, which returns the vector dimensions;
- `double magnitude(Vector)`, which returns the magnitude of the vector;
- `Vector + Vector`, vector addition;
- `Vector - Vector`, vector subtraction;
- `Vector * Vector`, component-by-component vector multiplication;
- `Vector / Vector`, component-by-component vector division;
- `double dot(Vector, Vector)`, the dot product of two vectors;
- `double cross(Vector, Vector)`, the cross product of two vectors. If the two vectors are one-dimensional, the cross product is a zero vector because the two vectors are in the same or opposite directions. If the two vectors are three-dimensional, the cross product is also a three-dimensional vector. When the two vectors are two-dimensional, the cross product is a one-dimensional vector, taking the direction perpendicular to the plane formed by the two vectors;
- `Vector + double`, the addition of a number to every component of a vector. It is commutative;
- `Vector - double`, the subtraction of a number to every component of a vector;
- `Vector * double`, the multiplication of a number to every component of a vector. It is commutative;
- `Vector / double`, the division of a number into every component of a vector;
- `- Vector`, same as `Vector * (-1)`;
- `1 / Vector`, the inversion of every component of a vector;
- `double - Vector`, same as `(- Vector) + double`;
- `double / Vector`, same as `(1 / Vector) * double`;

As an example,

$$\text{cross}(\text{Vector}(1,2), \text{Vector}(3,4)) = \text{Vector}(-2)$$

2.2 Random Variables

Random variables are another class of mathematical entities that are implemented as objects. A random variable, when evaluated, will return a random number in accordance with its probability density function (distribution). In Boltzmann, the evaluation of a random variable x is denoted by $x()$. All the random variables are under the class `Random`, which is defined in the GNU C++ library [37]. The following subsections discuss the Uniform, Normal, Maxwellian, and Exponential random variable classes.

2.2.1 Uniform Random Variables

A Uniform random variable can be specified in one of the two choices:

```
Uniform variable_name(double low, double high);
```

```
Uniform variable_name(double low, double high, RNG *gen);
```

where *low* and *high* are the lower and upper bounds of the random numbers generated by the variable, and *gen* is a “seed” generator for the random variable as discussed in the GNU C++ library manual. When the first specification is used, a default seed generator is employed which is often adequate.

2.2.2 Normal Random Variables

Normal (Gaussian) random variables have the distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad (-\infty < x < +\infty) \quad (2.1)$$

where μ is the mean value and σ the variance. A Normal random variable can be defined in the following ways:

```
Normal variable_name(double  $\mu$ , double  $\sigma$ );
```

```
Normal variable_name(double  $\mu$ , double  $\sigma$ , RNG *gen);
```

2.2.3 Maxwellian Random Variables

Maxwellian distribution

$$p(x) = \sqrt{\frac{m}{2\pi kT}} e^{-mx^2/2kT} \quad (-\infty < x < +\infty) \quad (2.2)$$

is a special case of Normal distribution with $\mu = 0$ and $\sigma = \sqrt{kT/m}$. It is often used to set the initial velocities of particles in equilibrium. A Maxwellian random variable can be as following:

```
Maxwellian variable_name(double T, double m);
```

```
Maxwellian variable_name(double T, double m, RNG *gen);
```

where T is the temperature in Kelvin degree and m is the mass of the particles.

2.2.4 Exponential Random Variables

Exponential random variables have distribution

$$p(x) = \lambda e^{-\lambda x} \quad (0 \leq x < \infty) \quad (2.3)$$

where λ is any positive constant. It occurs frequently as the distribution of waiting times between independent random events, such as the time a particle survives without a collision in an ideal gas. An Exponential random variable is defined in one of the two choices:

```
Exponential variable_name(double  $\lambda$ );
```

```
Exponential variable_name(double  $\lambda$ , RNG *gen);
```

2.3 Unit Conversion

Scientific computing can be performed in many different unit systems. The most common ones include the meter/kilogram/second (*mks*) unit system and the centimeter/gram/second (*cgs*) unit system, but other units are also employed frequently. The selection of appropriate unit system may have a direct impact on the ease of programming and the accuracy and efficiency of the simulation. Sometimes, it is desirable to express a problem in one unit system and to compute in another. To accommodate the variety of different unit systems, a set of macro definitions is provided to convert various unit system to a default one. Moreover, the default unit can be changed by the users.

The set of macro definitions is contained in the `Unit.h` file. The default unit is chosen to be *cgs*, for *mks* is often found too large for particle simulation. Other units are expressed in term of *cgs* unit, such as

```
#define centimeter 1           #define centimeters *centimeter
#define gram 1                 #define grams *gram
#define second 1               #define seconds *second
#define esu 1                  #define esus *esu

#define dyne gram*centimeter/sqr(second)
#define dynes *dyne
#define erg dyne*centimeter    #define ergs *erg
```

```

#define meter 100*centimeter      #define meters *meter
#define kilogram 1000*gram        #define kilograms *kilogram
#define coulomb 3e9*esu           #define coulombs *coulomb
#define newton 1e5*dyne           #define newtons *newton
#define joule 1e7*erg             #define joules *joule

#define angstrom 1e-8*centimeter #define angstroms *angstrom
#define electronvolt 1.60219e-12*erg
#define electronvolts *electronvolt
#define eV electronvolt          #define eVs electronvolts

```

To change the default unit system, it is sufficient to express the *cgs* units in term of the new default unit system. For example, if the unit length is defined to be

$$l_0 = 1.804 \times 10^{-8} \text{ cm} ,$$

the unit mass

$$m_0 = 5.275 \times 10^{-26} \text{ g} ,$$

and the unit time

$$t_0 = 3.275 \times 10^{-15} \text{ sec} ,$$

then it is enough to redefine *cgs* in term of l_0 , m_0 , and t_0 :

```

#define centimeter 1/1.804*1e8
#define gram 1/5.275*1e26
#define second 1/3.273*1e15

```

In that case, one electronvolt would be

$$\begin{aligned}
 1 \text{ eVs} &= 1 \times \text{electronvolt} \\
 &= 1 \times 1.60219 \times 10^{-12} \times \text{erg} \\
 &= 1 \times 1.60219 \times 10^{-12} \times \text{dyne} \times \text{centimeter} \\
 &= 1 \times 1.60219 \times 10^{-12} \times \text{gram} \times \text{centimeter} \\
 &\quad / \text{second}^2 \times \text{centimeter} \\
 &= 1 \times 1.60219 \times 10^{-12} \times 1/5.275 \times 10^{26} \\
 &\quad \times 10^{26} \times 1/1.804 \times 10^8 / (1/3.273 \times 10^{15})^2 \\
 &\quad \times 1/1.804 \times 10^8 \\
 &\approx 1.0 [m_0(l_0/t_0)^2]
 \end{aligned}$$

CHAPTER 3

A Particle Hierarchy

3.1 Particle Definition

The most essential concept in particle simulations is the concept of particles. The nature of the particles plus their initial conditions and the boundary conditions completely determine the characteristics of a simulated system. It is the variety of particles that makes particle simulation a general simulation methodology.

What are particles, anyway? In special-purpose particle simulations, the question has been overlooked since it has been taken for granted that our universe is full of physical particles such as atoms, molecules, and charged particles. In a systematic study of particle simulations, however, the question has to be addressed so that a consistent framework of particle simulations can be established. Here, a mathematical point of view of particles is presented to make the connection among seemingly unrelated particles. It also allow the possibility of discovering "particles" that have no counterparts in the physical world and have never been heard of in particle simulations. It is the identification of particles with Green's functions.

Here is a simple example. A charged particle creates an electric field in the space around it with the potential

$$\phi(\mathbf{r}) = \frac{e_0}{|\mathbf{r} - \mathbf{r}_0|} \quad (3.1)$$

where \mathbf{r}_0 is the location of the charged particle and e_0 is the charge. Any charged particle at \mathbf{r} would experience the existence of the charged particle at \mathbf{r}_0 by an electric force

$$-e\nabla\phi(\mathbf{r}) = \frac{ee_0(\mathbf{r} - \mathbf{r}_0)}{|\mathbf{r} - \mathbf{r}_0|^3} \quad (3.2)$$

where e is the charge of the charged particle at \mathbf{r} . On the other side of the same token, the same interaction can be derived from Poisson's equation

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\phi(\mathbf{r}) = -4\pi e_0\delta(\mathbf{r} - \mathbf{r}_0) \quad (3.3)$$

It is known that (3.1) is the solution of (3.3) (with the boundary condition of zero potential at infinity). If the solution of

$$\nabla^2\phi(\mathbf{r}) = -4\pi\delta(\mathbf{r} - \mathbf{r}_0) \quad (3.4)$$

is defined as Green's function $K(\mathbf{r}, \mathbf{r}_0)$ of Laplace operator ∇^2 , which is

$$K(\mathbf{r}, \mathbf{r}_0) = \frac{1}{|\mathbf{r} - \mathbf{r}_0|},$$

then (3.1) can be written as

$$\phi(\mathbf{r}) = e_0 K(\mathbf{r}, \mathbf{r}_0)$$

and (3.2) can be written as

$$-e \nabla \phi(\mathbf{r}) = -e e_0 \nabla K(\mathbf{r}, \mathbf{r}_0) .$$

That is to say that the interaction of a charged particle with other charged particles can be recovered from Green's function of Laplace operator¹, or the Green's function, plus the information of what field the Green's function represents, defines our charged particles.

Almost every well-behaved differential operator has a Green's function under homogeneous boundary conditions [13]. Therefore, large number of particles can be defined. The pre-defined particles can then be used to simulate many-body problems and distributed-parameter systems that can be reduced to many-body problems. Since Green's functions are the solution of differential equations with a point source, those particles can also be used to construct the solution of differential equations with any distribution of source.

Homogeneous boundary conditions have been assumed in the construction of the particles. Particles with special non-homogeneous boundary conditions can also be defined. Moreover, non-homogeneous boundary conditions can be simulated by image particles and boundary particles [43].

3.2 Particle Classification

Particle classification organizes the large number of particles into a hierarchy and facilitates software reusability. It makes the large number of particles manageable and reveals the connections among different classes of particles. In addition, a systematic classification strategy would allow new particles to be readily added in and the possibility of discovering unheard particles.

One rule should be obeyed in the classification of particles: if a scheme is able to supervise a class of particles, the same scheme should be able to supervise the subclasses of the particle class. That rule is enforced to make schemes reusable. A four-step classification strategy is devised to produce a hierarchy that is conceptually clean, facilitates best software reusability among particles, and obeys the classification rule. It consists of

1. Classification according to the kinds of field the particles create. The kinds of field include the force field, which contains all kinds of forces, and the velocity field. Since force is proportional to the second derivative of the displacement and velocity is the first derivative of the displacement, a third kind of field

¹Not quite! A moving charged particle creates not only an electric field but also a magnetic field. The interactions among moving charged particles are more than the electric force.

may be the zero-th derivative of the displacement or the displacement itself. The class of particles that create force fields is called Classical Particle class, or Newton Particle class. The class of particles that creates velocity fields is called Fluid Particle class, or Euler Particle class. The difference of the two classes is encapsulated in the integration step. Allowing this classification to be the first step reduces the number of classes and produces a conceptually clean hierarchy.

2. Classification according to Green's functions. It is the Green's functions that determine if special schemes can be devised to take advantage of them. Therefore, different classes of particles with the same Green's functions can work with the same schemes. A typical example is Green's function for Laplace operator. The field created by particles with that Green's function can be solved by the solution of a Poisson's equation in whatever fast algorithms that are available.
3. Classification according to the source of interaction. At this point, the kind of field and Green's function have been decided. But the source of interaction is yet to be fixed. For example, both charged particles and gravitational objects create force fields and both have the same Green's function. However, the source of interaction for charged particles is charge, while the source of interaction for gravitational objects is mass.
4. Classification due to particle variations. At this point, particles of the same class are basically the same, except for small variations. Those variations do not change the way particles interact with one another.

Figure 1.3 is a particle hierarchy produced by the classification strategy. The following sections discuss each class in details. After that, particle derivation is discussed.

3.3 The Particle Class

The Particle class is the topmost class in the hierarchy. It provides a consistent framework for all particle classes. The Particle class has at least the following four member functions:

- `void internal(Particle& p)`, which calculates the interaction between this particle and particle `p`.
- `void external()`, which calculates the external influence on the particle.
- `int integrate(double t, int step)`, which does one step of integration and returns the number of the next step. It returns zero when all steps are completed.

- `void update(double t)`, which does the rest updates before the particle is advanced into the next time step.

The Particle class are defined in one of the following ways:

- `Particle particle_name;`
- `Particle particle_array_name[size];`
- `Particle(internal_influence) particle_name;`
- `Particle(internal_influence) particle_array_name[size];`

The former two define a particle and an array of particles, respectively. The later two define the particle(s) with an internal influence function that specifies the interaction between any two such particles. When the influence function is not specified, a default zero influence function is used.

The following OOPS functions return attributes about a particle:

- `Vector& Location(Particle&)`, which returns the position vector of the particle,
- `int Color(Particle&)`, which returns the color code of the particle at display,
- and `float Diameter(Particle&)` returns the diameter of the particle at display,

Moreover, the same set of functions can be used to assign or change an attribute of a particle by providing two parameters, instead of one, where the first parameter is the particle and the second one is the new attribute. For example,

```
Location(a, Vector(x, y));
```

places particle `a` at the position `(x, y)`. In addition, the internal and external influence functions can be changed by the following two OOPS functions:

- `void Internal(Particle&, IntF influence)`, which changes the internal influence function,
- and `void External(Particle&, ExtF influence)`, which changes the external influence function,

where `IntF` and `ExtF` are defined as

- `typedef Vector (*IntF)(Particle&, Particle&);`
- `typedef Vector (*ExtF)(Particle&);`

3.4 Classical Particle Class (Newton)

Classical particles are described by two state variables: the position vector \mathbf{r} and the velocity vector \mathbf{v} where $\mathbf{v} = d\mathbf{r}/dt$. An equation of motion, which is Newton's law

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{F}/m \quad (3.5)$$

is associated with every particle. Classical Particle class is also called Newton Particle class. It is the choice of the force function \mathbf{F} that provides the rich variety of particle simulations that can be done with the two state variables \mathbf{r} and \mathbf{v} and Newton's law (3.5). The position \mathbf{r} and the velocity \mathbf{v} are vectors of either one, two, or three dimensions that are specified in the definition of the particle. In addition, a classical particle contains attributes concerning its correspondence with the outside world, such as its color and size when it is displayed. Not every attribute of a particle has to be specified each time the particle is defined. For some attributes, default values are employed when they are not specified.

Classical particles can be defined in the following way, in addition to the ways general particles are defined:

`Newton particle_name(mass, position, velocity, internal_influence);`

which specifies a Classical particle with its mass, position, velocity, and inter-particle interaction. The following OOPS functions return attributes about a Classical particle, in addition to the OOPS functions that can be applied to general particles:

- `Vector& Velocity(Newton&)`, which returns the velocity vector of the particle,
- `double Mass(Newton&)`, which returns the mass,

Similarly, the same set of functions can be used to assign or change an attribute of a particle by providing two parameters, instead of one, where the first parameter is the particle and the second one is the new attribute.

The equations of motion for a classical particle are Newton's law (3.5), or

$$\begin{cases} \mathbf{r}^{n+1} = \mathbf{r}^n + \int_{t_n}^{t_{n+1}} \mathbf{v} dt \\ \mathbf{v}^{n+1} = \mathbf{v}^n + \int_{t_n}^{t_{n+1}} (\mathbf{F}/m) dt \end{cases} \quad (3.6)$$

where \mathbf{F} is generally a function of the positions and velocities of this and other particles. There exist many numerical algorithms that approximate the solution of the equations of motion, and they can be found in the literature on initial-value problems.

A variety of "experiments" can be set up by the right combination of above functions. For instance, dissipation in wave propagation can be simulated by giving an external force proportional to the velocity of the particles; a fixed end of a metal rod can be simulated by a massive particle at the end or a particle with an empty integration step.

3.5 Laplace Particle Class

Forces that are derived from Green's function for Laplace operator have the form

$$\mathbf{E} = \frac{e\mathbf{r}}{|\mathbf{r}|^3} \quad (3.7)$$

where e is a constant attribute of the particles. They are often seen in the real world, such as the electrostatic force and the gravitational force. In the case of electrostatic force, e is equal to the charge and, in the case of gravitational force, e is equal to the mass multiplied by \sqrt{G} (G is the gravitational constant). The constant e is called the *strength* of the particle.

The specification of the inter-particle influence function of a Laplace particle is reduced to the specification of the strength. Although a Laplace particle can be defined as a Classical particle, it is more advantageous to define the particle as a Laplace particle with the specification of the strength of the particle. Internally, the influence function can be recovered if the particle has been loaded into a scheme that does not recognize Laplace Particle class. Beside its simpler specification, the main advantage of Laplace Particle class over Classical Particle class is the ability to run with faster schemes that take the advantage of the specific influence function. Examples of the faster schemes include the particle-mesh scheme and the fast multipole scheme.

A OOPS function that is defined with Laplace Particle class but not Classical Particle class is

```
double Strength(Laplace&);
```

which returns the strength of the particle. Laplace Particle class is a subclass of Classical Particle class.

3.5.1 Coulomb Particle Class

Coulomb particles interact with one another in electrostatic forces. Only one function has been added to the class in accordance with tradition:

```
double Charge(Coulomb&);
```

which returns the charge of the charged particle. Coulomb Particle class is a subclass of Laplace Particle class.

3.5.2 Kepler Particle Class

Kepler particles interact with one another in gravitational forces. The strength of a Kepler particle is the product of the square-root of the gravitational constant and the mass of the particle. Kepler Particle class is a subclass of Laplace Particle class.

3.6 Fluid Particle Class (Euler)

Fluid equations are often macroscopic field equations derived from averaging microscopic particle descriptions over a space that is large when compared with the interparticle separation and a time that is long when compared with the collision time of the microscopic particles. It is often possible to simulate fluids by particles of field nature that represent properties of the space. Unlike the classical particles, the particles are often driven by the velocity field instead of the force field. The class of particles that create velocity field is called Fluid Particle Class, or Euler Class. The description of fluid with the use of Euler particles is also called Eulerian description. Fluid Particle Class has one required state variable, the position \mathbf{r} , among its attributes. The equations of motion contain, at least,

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \quad (3.8)$$

where \mathbf{v} is the velocity field at the position of the discussed fluid particle. By employing *a priori* fluid particles, many phenomena of fluid flows can be simulated.

Euler particles can be defined in the following way, in addition to the ways general particles are defined:

Euler particle_name(position, influence_function);

The same set of OOPS functions that can be applied to Particle class can also be applied to Fluid Particle class.

3.7 Incompressible Vortex Class

As a subclass of Fluid Particle Class, Incompressible Vortex Class is a good example of how a fluid particle is constructed. Vortices have an attribute, the vorticity ω [47, 5], where

$$\omega = \nabla \times \mathbf{v} \quad (3.9)$$

From the incompressibility (the mass density ρ is constant), the velocity field is divergence-free, i.e.,

$$\nabla \cdot \mathbf{v} = 0 \quad (3.10)$$

It is known that there exists a vector potential ψ such that [4]

$$\mathbf{v} = \nabla \times \psi \quad (3.11)$$

and

$$\nabla^2 \psi = -\omega \quad (3.12)$$

By solving Equation (3.11) and (3.12), it is seen that the velocity field can be recovered from the vorticity of the vortices. That is to say that Green's function for (3.12), together with (3.11), defines the Incompressible Vortex class. From

the velocity field at each vortex, the position of the vortex at the next time step can be calculated. Unlike the charge attribute in Coulomb particles, however, the vorticity of vortices does not remain unchanged in general. It is the governing equation of the fluid that determines the new value of the vorticity of a vortex at the next time step. Very often, the governing equation for incompressible flow is the Navier-Stokes equation expressed in vorticity form:

$$\frac{d\omega}{dt} = (\omega \cdot \nabla)\mathbf{v} + \nu \nabla^2 \omega \quad (3.13)$$

where

$$\frac{d\omega}{dt} = \frac{\partial \omega}{\partial t} + (\mathbf{v} \cdot \nabla)\omega$$

is the rate of vorticity change along the flow lines, and ν is the kinematic viscosity of the fluid. According to the value of the viscosity ν , vortices are further classified into inviscid vortices, where $\nu = 0$, and viscous vortices.

Additional OOPS functions that can be applied to vortices include:

- `Vector& Vorticity(Vortex&)`, which returns the vorticity of the vortex,
- and `void Vorticity(Vortex&, Vector& ω)`, which assigns vorticity ω to the vortex.

A vortex can be defined as

```
Vortex name(Vector& r, Vector& ω);
```

3.7.1 Inviscid Vortex Class

For inviscid vortices, the equations of motion are Equation (3.8) and

$$\frac{d\omega}{dt} = (\omega \cdot \nabla)\mathbf{v} \quad (3.14)$$

Inviscid vortices can be defined as

```
Inviscid vortex(Vector& r, Vector& ω);
```

Inviscid Vortex class is a subclass of Vortex class.

3.7.2 The Viscous Vortex Class

Viscous vortices have one more attribute than the inviscid vortices, the viscosity ν . The equations of motion are Equation (3.8) and (3.13). The viscosity term in (3.13) can be accounted for by simulating the diffusion of vorticity by a random walk [47, 33], which will be part of the equations of motion. Viscous vortices can be defined as

```
Viscous vortex(Vector& r, Vector&  $\omega$ , double  $\nu$ );
```

Additional OOPS functions that can be applied to viscous vortices include:

- `double Viscosity(Viscous&)`, which returns the viscosity of the vortex,
- and `void Viscosity(Viscous&, double ν)`, which assigns viscosity ν to the vortex.

3.8 Particle Derivation

Although a great effort has been made to provide particle classes for a variety of applications, it is impossible and unwise to prepare a particle class for every kind of particles that may be used in particle simulations. Particle derivation is an important mechanism to make the particle hierarchy extensible. Specialization of high-level abstract particles is done through particle derivation. It contains facilities to add a variable, to add a new function, or to change the definition of an existing function, among others. Particle derivation can be considered as a collection of second-order functions that take particle classes as their domain. C++, the implementing language underlying the Boltzmann programming system, provides static class derivation/inheritance that can be used to support particle derivation. The syntax for class derivation in C++, however, requires some understanding of the C++ language, which should be minimized according to one of the principles in the design of Boltzmann. To this end, a number of macro definitions are provided to hide certain syntactic details that are not important to the users. At times, it is desired that C++ language provide dynamic class derivation which would allow more freedom in particle derivation.

The addition of a variable or variables to a class can be done as in

```
class derived_class: INHERIT(parent_class, variable_definitions);
```

where *variable_definitions* are regular definition of variables in the C language, such as `int i` and `float x`. To add a function or to change the definition of a function, the same macro definitions can be used with the *variable_definitions* replaced by *function_definitions*, as in

```
class derived_class: INHERIT(parent_class, function_definitions);
```

where *function_definitions* are regular definition of functions in C in the form

```
return_type function_name(parameter_list) {statements}
```

As an example, the following statement defines a subclass, `rkParticle`, of the parent class, `Particle`, that replaces the default leap-frog step function by the 4th-order Runge-Kutta method:

```

class rkParticle: INHERIT(Newton,
    Vector r0, v0, f0, v1;
    int integrate(double t, int step) {
        switch (step) {
            case 1: r0 = r; v0 = v; f0 = f;
                r += 0.5*t*v; v += 0.5*t/m*f;
                return 1;
            case 2: v1 = v; f0 += 2*f;
                r = r0+0.5*t*v; v = v0+0.5*t/m*f;
                return 1;
            case 3: v1 += v; f0 += 2*f;
                r = r0+t*v; v = v0+t/m*f;
                return 1;
            case 4: r = r0+t/6*(v0+2*v1+v);
                v = v0+t/(6*m)*(f0+f);
                return 0;
            default:
                return 0;
        }
    }
);

```

Frequently-used particle derivations are provided by macro definitions more detailed than INHERIT. The following macros return a subclass of the parent particle class with an integration algorithm designated by the names of the macros:

- EULER(*parent_particle*): Euler's method
- LF(*parent_particle*): Leap-frog method
- RK2(*parent_particle*): the 2nd-order Runge-Kutta method
- RK4(*parent_particle*): the 4th-order Runge-Kutta method
- DEACTIVATE(*parent_particle*): empty integration function

For example, the class rkParticle in the previous example can be defined simply as

```
class rkParticle: RK4(Newton);
```

The macro DEACTIVATE returns a subclass with an empty integration function, that is, particles of this class will remain in the same state even though it may have interaction with other particles.

CHAPTER 4

A Scheme Hierarchy

4.1 Introduction

According to the programming framework, particles alone can not complete a simulation. They have to be supervised by computational schemes, or schemes for short, in simulations. The responsibilities of a scheme include influence calculation, collision detection, boundary imposition, among other things. In traditional particle simulation approaches, computational schemes are algorithms that fulfill the responsibilities. In the OOPS approach, however, schemes are objects, or “black boxes”, that have the capability of supervising particles through their change of states in simulations. All the necessary algorithms for a simulation are encapsulated in a scheme. Therefore, a particle simulation can be carried out simply by loading particles into a scheme and then running the scheme.

The most important responsibility of a scheme is the evaluation of particle-particle interactions, or influence calculation. It calculates the total influence on a particle from all other particles due to inter-particle interactions and influence from the external field, and feeds the total influence to the equations of motion of the particle. It is also the most time-consuming part of a scheme and often of the entire simulation, because of the large number of particles that are involved in influence calculation. Several different schemes are established to take advantage of influence calculations of different nature. One of them is the Particle-Particle (PP) Scheme that calculates the influence on a particle simply by summing up all influence contributions from other particles. It is the most general scheme that can be applied to the most general class of particles, the Particle Class. Another scheme is the Particle-Mesh (PM) Scheme that calculates the influence on a particle by solving a Poisson's equation for the potential of influence field. It applies only to Laplace particles. It is considerably faster than the Particle-Particle Scheme for large number of particles. The Multipole Expansion Scheme, applies also to Laplace Particle class and is promised to be even faster than the Particle-Mesh Scheme [26, 2].

In the OOPS approach, schemes also maintain information about the boundaries and the boundary conditions. When particles reach the boundaries, schemes reallocate them in accordance with the boundary conditions. For examples, the kinds of boundary conditions may include open space (Open), no-flow boundaries (Bounded), periodic (Periodic), and homogeneous boundary conditions (Dirichlet).

4.2 The Scheme Class

The Scheme class serves as the basis for all schemes. A scheme always defines its dimension, a computation space in which particles interact with one another, a particle stack that is used to contain particles, and a list of boundaries and boundary conditions. A stack is different from an array, as new particles can be added and are added to the top of the stack. Boundaries surround special regions of the computation space such that when particles are entering or leaving the region, boundary conditions are enforced.

A scheme in the Scheme class is specified in the following way:

```
Scheme scheme_name(Vector& low, Vector& high, BC bc);
```

where *low* and *high* are the lower bound and the upper bound of the computation space, respectively, and *bc* is the boundary condition for the space, which can be one of the enumeration constants `Open`, `Bounded`, `Periodic`, `Dirichlet`, and `Neumann`. The default boundary condition is `Bounded` when *bc* is not given. For a one-dimensional scheme, the lower and upper bounds are vectors of one dimension. For a two-dimensional scheme, they are vectors of two dimensions. The following functions return information about a scheme:

- `int Num(Scheme)` returns the number of particles loaded in the scheme,
- `int Dim(Scheme)` returns the dimensionality of the scheme,
- `Vector& UpperBound(Scheme)` returns the upper bound of the space of the scheme,
- `Vector& LowerBound(Scheme)` returns the lower bound of the space of the scheme,
- and `Particle& Scheme[int i]` returns the *i*th particle in the particle stack of the scheme.

The following member functions define the behavior of the Scheme Class:

- `void Calculation()` calculates the influence on every particle stored in the scheme.
- `void Advance(double t)` advances all the particles one time step of size *t*.

4.3 The Particle-Particle Scheme

The Particle-Particle Scheme, denoted `PPscheme` in the programming system, is a subclass of the Scheme class. Whereas the Scheme class does not specify how the influence on every particle is calculated, the Particle-Particle Scheme calculates

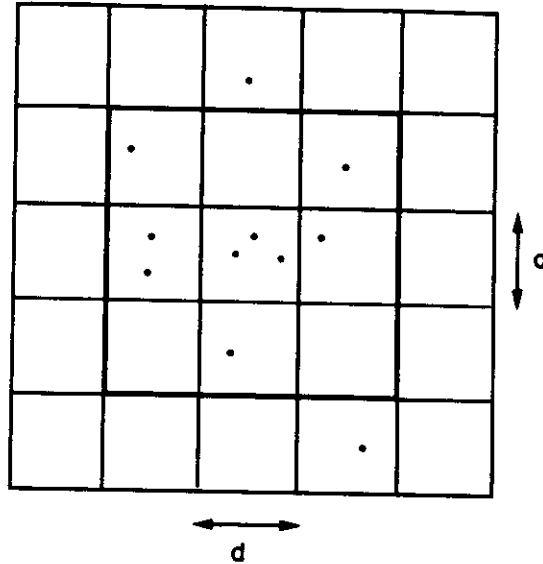


Figure 4.1: Short-range influence calculation in PP Scheme

the influence by summing up all influence contributions from other particles. It is, in general, a $O(N^2)$ scheme in computation time, where N is the number of particles. For certain types of influences, however, the Particle-Particle Scheme can facilitate special arrangements that reduce the computation time, usually to the order of N . Two special arrangements are for the calculation of short-range influences and of influences existing only between fixed pairs of particles.

For short-range influences, the Particle-Particle Scheme allows the specification of a cut-off distance, beyond which the influence of a particle is assumed to be negligible. The arrangement for short-range influences can be illustrated in Figure 4.1, where each cell is a square of the length d equal to the cut-off distance. The influence on a particle is the sum of the influences from other particles in the same cell and from the particles in the neighboring cells. Assuming the number of particles in each cell is N/M on the average, where M is the total number of cells, then the computation time for calculating the influences on the particles in one cell is $O(N/M)^2$. The total computation time for all the cells is then $O(N^2/M)$. Since the number of cells M is approximately equal to $(L/d)^2$, where L is the length of the space, the computation time for the calculation of short-range influences is $O(Nd/L)^2$.

Certain applications require influence calculation only between fixed pairs of particles. It is then necessary to specify where influences exist so that the scheme does not create extra influences and consumes less computation time. The Connection Class is defined to designate a partial specification of the particle pairs where influences exist. An instance of the Connection Class, or *connection*, is denoted by

$$(i : j : k)$$

	0	1	2	3	4
5	6	7	8	9	
10	11	12	13	14	
15	16	17	18	19	
20	21	22	23	24	

(a)

(0:4:1) U (5:9:1) U (10:14:1) U (15:19:1) U (20:24:1) U
 (0:20:5) U (1:21:5) U (2:22:5) U (3:23:5) U (4:24:5)

(b)

Figure 4.2: A connection

which represents the set of particle pairs

$$\{(i, i + k), (i + k, i + 2k), \dots, \\ (i + (\lfloor \frac{j-i}{k} \rfloor - 1)k, i + \lfloor \frac{j-i}{k} \rfloor k)\}$$

For example,

$$(0 : 5 : 2) = \{(0, 2), (2, 4)\}$$

where there exist influences between particle 0 and particle 2 and between particle 2 and particle 4. A union of connections gives a full specification of all particle pairs where influences exist. For example, the connection in Figure 4.2(a), where a line designates an influence between the two particles at the two ends, can be represented by the union in Figure 4.2(b).

A Particle-Particle scheme is specified in the following way:

```
PPscheme scheme_name(Vector& low, Vector& high, BC bc,
double d);
```

where d gives the cut-off distance of the short-range influence. The default cut-off distance, when d is not given, is infinity. To add a connection to a PP scheme, the function `AddConnection` is used:

```
void AddConnection(PPscheme&, int i, int j, int k);
```

4.4 The Particle-Mesh Scheme

A particle creates an influence field \mathbf{E} in the space it occupies. For a Laplace particle, the field is conservative, that is, there exists a potential field ϕ such that

$$\mathbf{E} = -\nabla\phi \quad (4.1)$$

For an influence as defined in Equation (3.7), Poisson's equation states that

$$\nabla^2\phi = -4\pi e\delta(\mathbf{r} - \mathbf{r}_0) \quad (4.2)$$

where the particle is located at \mathbf{r}_0 . The potential of a collection of particles is then a superposition of the potentials created by every individual particle,

$$\nabla^2\phi = -4\pi \sum_i e_i\delta(\mathbf{r} - \mathbf{r}_i) \quad (4.3)$$

The influence on a particle at \mathbf{r} is then

$$\mathbf{F} = e\mathbf{E}(\mathbf{r})$$

The Particle-Mesh Scheme solves the equation (4.3) on a mesh over the space and interpolate influences on the particles from potentials on the mesh. The influence calculation steps are as listed below:

1. Distribute the strength density function on the grid points:

$$\rho(\mathbf{r}) = \sum_i e_i\delta(\mathbf{r} - \mathbf{r}_i)$$

2. Solve the Poisson's equation for the potential on the grid points:

$$\nabla^2\phi = -4\pi\rho \quad (4.4)$$

3. Calculate the influence field on the mesh from Equation (4.1).
4. Interpolate the influence at each particle from the influence field on the mesh.

It is important to realize that distributing the strength density function in Step 1 has the effect of reducing collisions among particles, and thus the Particle-Mesh Scheme is good only when collisions are not important in the physical system. In addition, the finite mesh has the effect of filtering out signals of wavelength smaller than the distance between two neighboring grid points. When the Particle-Mesh Scheme is applicable, however, it usually out-performs the Particle-Particle Scheme in speed. Of the four steps in the influence calculation, Step 1 and Step 4 are of the order of N in computation time, Step 2 is $O(M \log M)$ where M is the number of grid points of the mesh, and Step 3 is $O(M)$. The total computation

complexity, $O(M \log M + N)$, is considerably smaller than the computation complexity, $O(N^2)$, of the Particle-Particle Scheme when N is much larger than M . The size of M is decided by the characteristic wavelength of the physical system.

There exist many approaches for the interpolation of the strength density function on the mesh and the interpolation of influence field from the mesh to the particles. Two popular ones are the Nearest-Grid-Point (NGP) and the Cloud-In-Cell (CIC) approaches. In the NGP approach, the strength of a particle is assigned to the nearest grid point in step 1 and the influence at a particle is made equal to the influence field at the nearest grid point in step 4. In the CIC approach, the strength of a particle is distributed to the surrounding grid points in proportion to the distances between the particle and the grid points in step 1, and the influence field at the surrounding grid points contribute to the influence at the particle in proportion to the distances between the particle and the grid points. In mathematical notation, the interpolation of the strength of a particle p at a mesh point q can be written as

$$\rho(\mathbf{r}_q) = e_p W(\mathbf{r}_p, \mathbf{r}_q)$$

and the interpolation of the influence field at a particle p from a grid point q can be written as

$$\mathbf{E}(\mathbf{r}_q) = \mathbf{E}(\mathbf{r}_p) W(\mathbf{r}_p, \mathbf{r}_q)$$

In one dimension,

$$W_{NGP}(\mathbf{r}_p - \mathbf{r}_q) = \begin{cases} 1 & \text{if } |\mathbf{r}_p - \mathbf{r}_q| < 0.5h \\ 0 & \text{otherwise} \end{cases}$$

and

$$W_{CIC}(\mathbf{r}_p - \mathbf{r}_q) = \begin{cases} 1 - |\mathbf{r}_p - \mathbf{r}_q|/h & \text{if } |\mathbf{r}_p - \mathbf{r}_q| < h \\ 0 & \text{otherwise} \end{cases}$$

In two dimensions,

$$W_{NGP}(\mathbf{r}_p - \mathbf{r}_q) = \begin{cases} 1 & \text{if } |\mathbf{r}_p - \mathbf{r}_q| < |\mathbf{r}_p - \mathbf{r}_{q'}| \ (q' \neq q) \\ 0 & \text{otherwise} \end{cases}$$

and

$$W_{CIC}(\mathbf{r}_p - \mathbf{r}_q) = \begin{cases} (h_x - |x_p - x_q|)(h_y - |y_p - y_q|)/h_x h_y & \text{if } |x_p - x_q| < h_x \\ & \text{and } |y_p - y_q| < h_y \\ 0 & \text{otherwise} \end{cases}$$

Solving Poisson's equation in step 2 is usually the most time consuming step in computation time in the Particle-Mesh scheme. There exist several fast algorithms that have the complexity of $O(M \log M)$, of which the Fast Fourier Transform

(FFT) method is the simplest and is described below for the two-dimensional periodic boundary condition. If the discretized Poisson's equation is

$$\frac{\phi_{j+1,l} - 2\phi_{j,l} + \phi_{j-1,l}}{h_x^2} + \frac{\phi_{j,l+1} - 2\phi_{j,l} + \phi_{j,l-1}}{h_y^2} = -4\pi\rho_{j,l}$$

where the left-hand side approximates the second-order derivative in Equation (4.4), then by taking the two-dimensional FFT on both sides it is found

$$\begin{aligned} \hat{\phi}_{mn} [h_y^2 (e^{i2\pi m/M_x} + e^{-i2\pi m/M_x}) + h_x^2 (e^{i2\pi n/M_y} + e^{-i2\pi n/M_y}) - 2(h_y^2 + h_x^2)] \\ = -4\pi h_x^2 h_y^2 \hat{\rho}_{mn} \end{aligned}$$

or

$$\hat{\phi}_{mn} = \frac{-2\pi h_x^2 h_y^2 \hat{\rho}_{mn}}{h_y^2 \cos(2\pi m/M_x) + h_x^2 \cos(2\pi n/M_y) - (h_y^2 + h_x^2)}$$

where $\hat{\phi}_{mn}$ and $\hat{\rho}_{mn}$ are the Fourier transform of ϕ_{mn} and ρ_{mn} , respectively. Taking the inverse FFT of the both sides of the above equation produces ϕ_{jl} ($j = 0, 1, \dots, M_x, l = 0, 1, \dots, M_y$), which are the solution of the Poisson's equation on the mesh.

A Particle-Mesh scheme is specified in one of the following ways:

```
PMScheme scheme_name(Vector& low, Vector& it high,
int n1, BC bc);
```

```
PMScheme scheme_name(Vector& low, Vector& it high,
int n1, int n2, BC bc);
```

```
PMScheme scheme_name(Vector& low, Vector& it high,
int n1, int n2, int n3, BC bc);
```

Where n_1 , n_2 , and n_3 are the number of grid points along the first, the second, and the third dimension, respectively. The default boundary condition is Periodic when *bc* is not specified. The member functions of the Particle-Mesh Scheme include

- void `DensityAssignment()`, which assigns strength density to the grid points.
- void `PotentialCalculation()`, which solves the Poisson's equation,
- void `InfluenceField()`, which calculates the influence field on the mesh,
- void `Interpolation(Laplace&)`, which interpolates the influence field at the particle,
- and void `Calculation()`, which involves the above functions one after another to calculate the influence on a particle at each time step.

Each function can be replaced by a new definition without the change of other functions.

CHAPTER 5

Initialization and Visualization

From the user's point of view, a particle simulation using the Boltzmann programming system primarily entails input and output, or initialization and visualization. Very much like doing an experiment, a user selects and prepares the correct particles and schemes, arranges them according to the simulation requirements, and then starts the simulation and waits for the result. The user is provided a number of initialization and visualization tools to help setting up the experiment and measuring various outputs of the simulation. It is also possible to the user to write his/her own tools with the use of more primitive functions provided by the Boltzmann programming system.

5.1 Initialization

A user can initialize his/her simulation in a number of ways. The most primitive approach is to assign values to the attributes of particles one at a time, usually in a loop statement. The advantage of that approach is its flexibility in setting up all kinds of initial conditions, but the disadvantage is that the calculation of the attribute values can be very involved. Another approach builds, on top of the primitive functions, routines that automatically assign values to particles according to macroscopic properties of them, such as the temperature and the energy. Principles of statistical physics often determine the values to be assigned to the particles [36]. Common arrangements in the following subsections are made as initialization routines.

5.1.1 Maxwellian Distribution for Velocity

According to Maxwellian distribution, the probability density of particles in equilibrium having velocity $\mathbf{v} = (v_x, v_y, v_z)$ is

$$\left(\frac{m}{2\pi kT}\right)^{3/2} e^{-m(v_x^2 + v_y^2 + v_z^2)/2kT} \quad (5.1)$$

where T is the temperature and k the Boltzmann constant. It can be used to assign velocities to a set of particles in equilibrium based on the temperature, with the use of a Maxwellian random variable to generate values for each component of the velocities. The routine to generate Maxwellian distribution to a set of particles is defined as

```
void MaxwellianDistribution(Particle* particles,
```

```

    double m, double T, int N)
{
    Maxwellian var(m, BoltzmannConstant*T);
    for (int i= 0; i< N; i++)
        switch(dim(particles[i])) {
            case 1: Velocity(particles[i], Vector(var()));
                    break;
            case 2: Velocity(particles[i], Vector(var()),var());
                    break;
            case 3: Velocity(particles[i], Vector(var()),var(),
                            var());
                    break;
        }
}

```

A routine that combines MaxwellianDistribution, mass assignment, and Load is defined as

```

void MaxwellianLoad(Scheme& scheme, Particle* particles,
    double m, double T, int N)
{
    MaxwellianDistribution(particles, m, T, N);
    for (int i= 0; i< N; i++)
        Mass(particles[i], m);
    Load(scheme, particles, N);
}

```

5.1.2 Position Displacement of Small Oscillations

For small oscillations about some equilibrium positions, the probability density of particles having displacement $\mathbf{r} = (x, y, z)$ from the equilibrium positions is

$$\left(\frac{\omega^2 m}{2\pi kT}\right)^{3/2} e^{-\omega^2 m(v_x^2 + v_y^2 + v_z^2)/2kT} \quad (5.2)$$

where ω is the oscillation frequency. It is useful in setting the initial positions of particles oscillating about some equilibrium positions such as crystal lattices. Comparing the above distribution with Maxwellian distribution, it is seen that a Maxwellian random variable can also be used to generate the displacement, with $\omega^2 m$ replacing m in the definition of the random variable, as in

```

void Oscillating(Particle* particles, double m, double T,
    double omega, int N)
{
    Maxwellian var(omega*omega*m, BoltzmannConstant*T);
}

```

```

for (int i= 0; i< N; i++)
  switch(dim(particles[i])) {
  case 1: Location(particles[i],
    Location(particles[i])+Vector(var()));
    break;
  case 2: Location(particles[i],
    Location(particles[i])+Vector(var(),var()));
    break;
  case 3: Location(particles[i],
    Location(particles[i])+
    Vector(var(),var(),var()));
    break;
  }
}

```

A routine that combines Oscillating and MaxwellianLoad for oscillating particles at at some temperature is defined as

```

void OscillatingLoad(Scheme& scheme, Particle* particles,
  double m, double T, double omega, int N)
{
  Oscillating(particles, m, T, omega, N);
  MaxwellianLoad(scheme, particles, m, T, omega, N);
}

```

5.1.3 Particles Under the Influence of Gravity

The distribution of particles under the influence of gravity satisfies Boltzmann's formula

$$n(h) = n(0)e^{-mgh/kT} \quad (5.3)$$

where g is the magnitude of the gravitational acceleration, h the height of space from any reference point in the opposite direction of the gravitational acceleration, and $n(h)$ the number density of particles at height h . If what is wanted is to assign positions to some fixed number of particles at the height from a to b according to Boltzmann's distribution, it can be done by generating a random number y with an exponential distribution at the range from 0 to ∞ , discarding y when $y > b-a$, and letting $h = y + a$. The following routine has the last space dimension distributed according to Boltzmann's formula and the rest of space dimensions distributed uniformly:

```

void Gravity(Particle* particles, double m, double T,
  Vector& low, Vector& high, int N)
{
  Exponential y(GAcceleration*m/(BoltzmannConstant*T));
}

```

```

Uniform u(0, 1);
double h;
for (int i= 0; i< N; i++) {
    while ((h=y()) > (high[dimension(high)-1]-
        low[dimension(low)-1]));
    switch(dim(particles[i])) {
    case 1: Location(particles[i], Vector(h+low[0]));
        break;
    case 2: Location(particles[i], Vector(low[0]+
        u()*(high[0]-low[0]), low[1]+h));
        break;
    case 3: Location(particles[i], Vector(low[0]+
        u()*(high[0]-low[0]),
        low[1]+u()*(high[1]-low[1]), low[2]+h));
        break;
    }
}
}

```

A routine that combines Gravity, MaxwellianLoad, and assignment of gravity : force function for initializing Classical particles under the influence of gravity and at some temperature is defined as

```

void GravityLoad(Scheme& scheme, Particle* particles,
    double m, double T, int N)
{
    Gravity(particles, m, T, LowerBound(scheme),
        UpperBound(scheme), N);
    for (int i= 0; i< N; i++)
        Externalparticles[i], gravity);
    MaxwellianLoad(scheme, particles, m, T, N);
}

```

where gravity is a function constant defined as

```

Vector& gravity(Particle& p)
{
    switch(dim(p)) {
    case 1: return Vector(-Mass(p)*GAcceleration);
    case 2: return Vector(0, -Mass(p)*GAcceleration);
    case 3: return Vector(0, 0, -Mass(p)*GAcceleration);
    }
}

```


5.2 Visualization

Visualization of simulation processes is similar to looking through a microscope or a telescope at a real system. More than that, simulation visualization has the flexibility of displaying various outputs and measurements that may be difficult or even impossible to see in the real situation. Visualization in the Boltzmann programming system is facilitated by *visualization windows* and *drawing functions* defined on the windows. For frequently-used diagnoses, visualization of a simulation is simply the work of clicking the right buttons. For unusual ones, visualization can be easily constructed from high-level drawing functions that require no understanding of the underlying graphics primitives. Visualization windows, drawing functions, and some diagnosis functions are the subject of the following subsections.

5.2.1 Visualization Windows

Simulation results are displayed in visualization windows. A visualization window can be as simple as a “drawing board” on which drawing functions can write points and lines. Or, it can provide functions such as diagnosis selection, tracking, replaying, and scrolling. In Boltzmann, visualization windows are objects, allowing an unlimited number of windows to be supported. An example of visualization windows, called `Swindow`, is described below.

A `Swindow` is defined by declaring

```
Swindow window_name(Vector x-dimension, Vector y-dimension);
```

where *x-dimension* and *y-dimension* are vectors of two elements giving the lower bound and the upper bound of the window in the horizontal and the vertical directions, respectively. The lower and upper bounds are given the unit of space in the simulation, not the unit of the size of the physical screen. This makes it easy to determine what the lower and upper bounds should be in order to visualize a specific region of a simulation.

When activated, a `Swindow` looks like the one in Figure 5.1, where the big blank space is the drawing board and the buttons are for the functions that come with the window. The `POPUP` button is for opening a menu window that allows the selection of diagnosis functions and the modification of system parameters such as the step size. Figure 5.2 displays a menu window. Clicking the `DIAGNOSIS` button in the menu window opens a menu of diagnosis functions. A sample list contains items such as Cartesian Space, Phase Space, and Power Spectrum. User-defined diagnosis functions can be installed into the list by calling the function

```
InstallDraw(Swindow window, char* label, DrawFun func);
```

where `DrawFun` is defined as

```
typedef void (*DrawFun)(Swindow, Scheme);
```

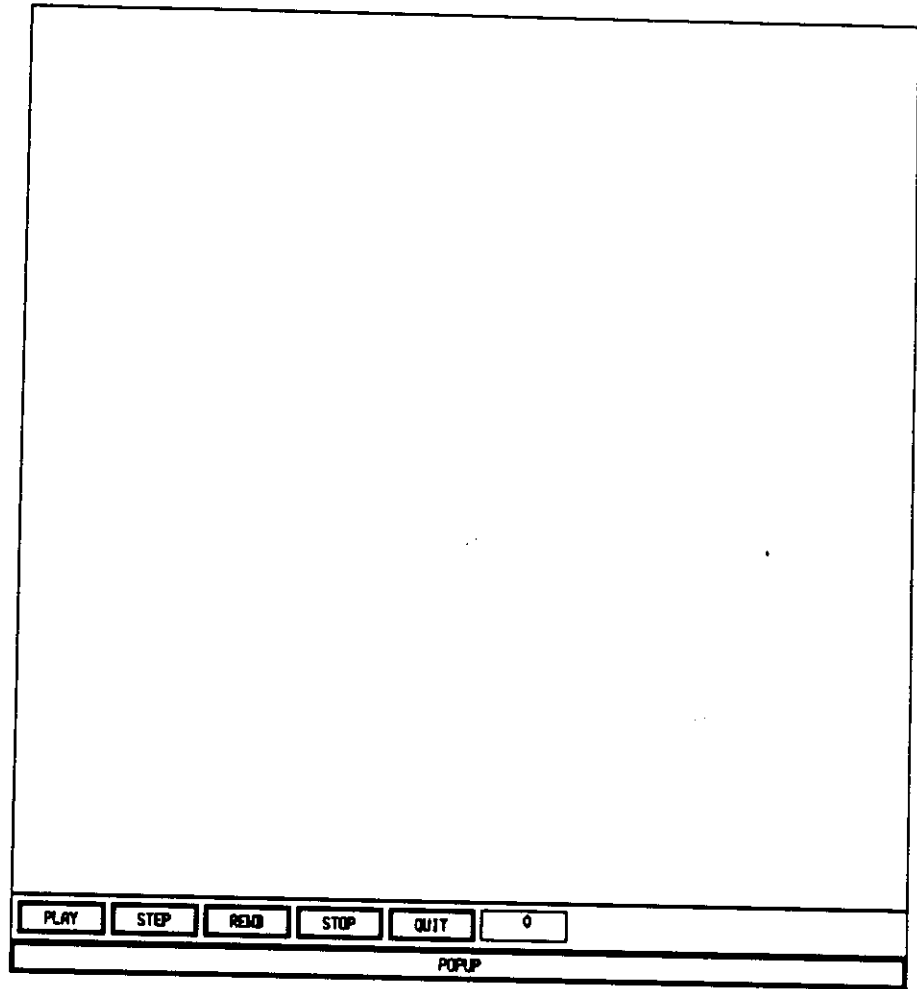


Figure 5.1: A visualization window

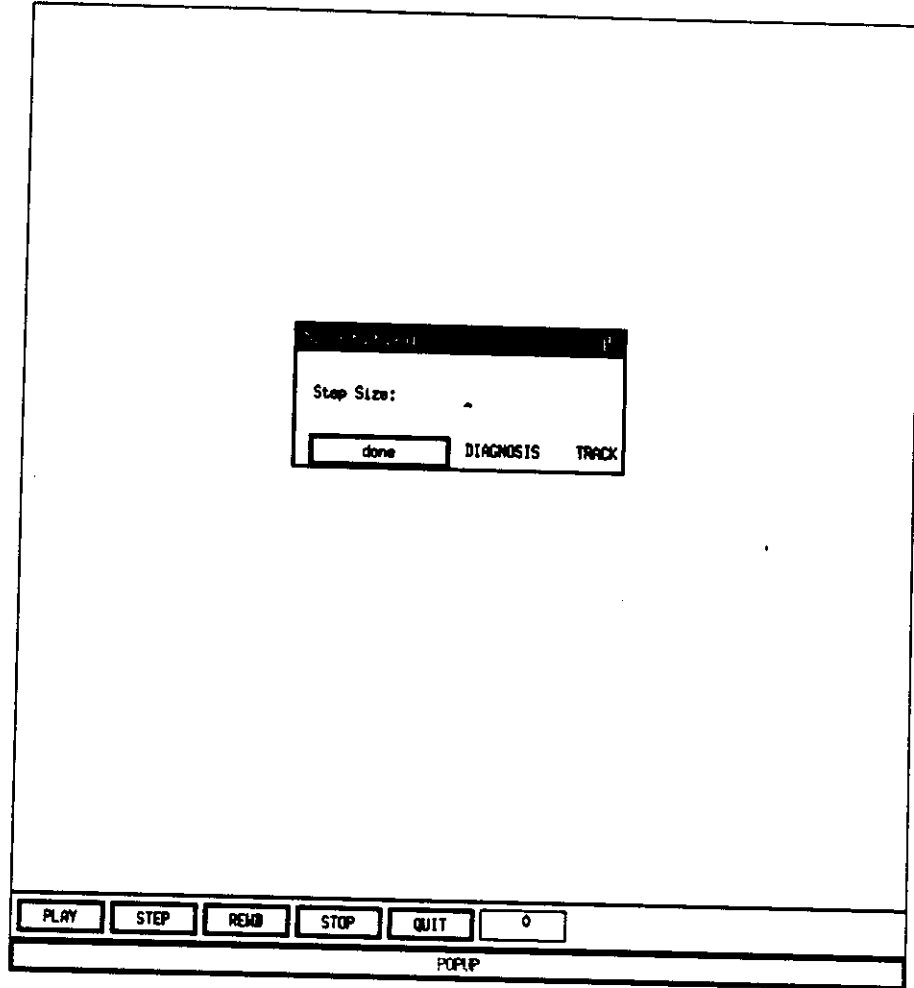


Figure 5.2: A menu window

and *func* is the user-provided diagnosis function. Clicking the TRACK button turns the tracking function of the window on or off, which determines if the results of earlier time steps should be kept unerased. The DONE button closes the menu window.

Returning to the Swindow, the PLAY button initiates the simulation and the display of the results according to the diagnosis function chosen in the menu window. The right-most small window in the same row displays the time step at which the results are being displayed. The STEP button advances the simulation by one time step. The REWD button rewinds the simulation, by running the same simulation with a negative time step. According to classical mechanics, a Newtonian many-body problem is time-reversible, provided the computation error is negligible. Therefore, the REWD button can be used as an error checker. If the simulation can not go back to the original state, significant computation errors have been introduced and caution is required. The STOP button stops the simulation and the QUIT button terminates it.

5.2.2 Drawing Functions

To facilitate the programming of user-defined diagnosis functions, a number of high-level drawing functions are provided that do not require an understanding of the underlying graphics primitives. The first drawing function is

```
void DrawParticle(Swindow window, Particle particle)
```

which draws a circle on the drawing board of *window*. The size and color of the circle depends on the attributes of the particle *Diameter(particle)* and *Color(particle)*. The size of the circle on the drawing board is in proportion to the ratio of *Diameter(particle)* to the size of the window. When *Diameter(particle)* is positive, the circle is drawn in the color designated by *Color(particle)*. When *Diameter(particle)* is negative, the circle is filled with the same color.

The second drawing function is

```
void DrawCircle(Swindow window, Vector origin, double diameter,  
double color)
```

This is equivalent to *DrawParticle* with *origin* replacing *Location(particle)*, *diameter* replacing *Diameter(particle)*, and *color* replacing *Color(particle)*. It is used to draw a circle, usually to represent a particle, when *DrawParticle* is inadequate. *DrawCircle* is more flexible than *DrawParticle* because *origin*, *diameter*, and *color* can be chosen arbitrarily.

The last drawing function discussed here is

```
void DrawLine(Swindow window, Vector r1, Vector r2,  
double color)
```

which draws a line from the point at r_1 to the point at r_2 on the drawing board in the color designated by *color*. These three drawing functions are the most basic ones. An example, the following program defines a function that displays the particles loaded in a scheme in their (x, v_x) phase space:

```
void phase(Swindow& window, Scheme& scheme)
{
    double x, y;
    for (int k= 0; k < Num(scheme); k++) {
        x = Location(scheme[k])[0];
        y = Velocity(scheme[k])[0];
        DrawCircle(window, Vector(x, y),
            Diameter(scheme[k]), Color(scheme[k]));
    }
}
```

It can be installed into the diagnosis list of *window* by calling

```
InstallDraw(window, "X-Phase", phase);
```

5.2.3 Diagnoses

Usually, it is the diagnoses that make simulation attractive. One of the major advantages in organizing particles into hierarchy is that many diagnosis functions can be made independent of specific applications, in a way similar to measuring instrument in real experiments where many instruments are of general purpose. It makes visualization, in many cases, as simple as clicking a button, by providing a list of general-purpose diagnosis functions with visualization windows. In addition, the users are allowed to construct and install their own functions as discussed before. Some general purpose diagnosis functions are discussed below.

5.2.3.1 Temperature

Temperature is a macroscopic quantity defined for a system of particles in thermal equilibrium. When a system of particles is not in equilibrium, temperature can be defined for subsystems of the particles that are in equilibrium. The temperature T of a system of particles in equilibrium can be found from

$$T = \frac{m}{k} \langle v_x^2 \rangle$$

where $\langle v_x^2 \rangle$ is the mean square x-component of velocities of the particles and k is the Boltzmann constant [36]. If the velocities of the particles have more than one component, it is also true that

$$T = \frac{m}{k} \langle v_y^2 \rangle$$

or

$$T = \frac{m}{k} \langle v_x^2 \rangle$$

They can be used to verify if the particles are in equilibrium, for the temperature must be the same no matter whether it is calculated from the x-component, the y-component, or the z-component when the particles are in equilibrium. For a system of particles not in equilibrium, the temperature of local subsystems of the particles can be found in the same way.

5.2.3.2 Total Energy

The total energy of a system of particles consists of three parts: kinetic energy of the particles, potential energy due to the interaction of the particles, and potential energy of the particles in an external field. The kinetic energy can be calculated easily as in

$$K = \frac{1}{2} \sum_{i=1}^N m_i v_i^2$$

where N is the number of particles. However, the potential energies are not so easy to calculate, since they depend on the kinds of interaction and the external field. For Coulomb particles, the potential energy is

$$U = \sum_{i=1}^N e_i \left(\frac{1}{2} \phi_i + \phi'_i \right)$$

where ϕ_i is the potential of the field produced by the particles and ϕ'_i is the potential of the external field, at the location of particle i , and e_i is the strength of particle i .

5.2.3.3 Drag Force on Particles

Force on a particle can be in any direction and the motion of a particle can be complex. Very often, however, there may be a tendency of a particle to move in a certain direction caused by a uniform *drag* force on the particle. Here, the drag is a statistical concept to illustrate the tendency for a particle to move in a certain direction. From Newton's law

$$\mathbf{F} = m \frac{d\mathbf{v}}{dt}$$

the drag can be described by the change of velocities of a group of "similar" particles. Particles are considered similar if they have approximately the same initial velocity. A plot of the mean velocity at different time step would illustrate the effect of a drag force.

5.2.3.4 Velocity Diffusion of Particles

Because of collisions, particles diffuse from each other in velocity. That is, a group of similar particles may have a large spread in velocity after a few time steps. The rate of diffusion can be measured by the mean-square spread in the velocity as a function of time of similar particles (having about the same initial velocity). The mean-square spread is defined to be

$$\langle \Delta v^2 \rangle \equiv \langle [\mathbf{v}(t) - \langle \mathbf{v}(t) \rangle]^2 \rangle$$

where the averaging operator $\langle \rangle$ is over the group of similar particles.

5.2.3.5 Power Spectrum Analysis

A mass of particles can support waves. The wave properties of a system of particles can be studied by power spectrum analysis. For a function $h(x)$, the Fourier transform of that function is defined to be

$$H(f) = \int_{-\infty}^{\infty} h(x) e^{2\pi i f x} dx$$

If $h(x)$ is a function of time, the Fourier transform is a function of frequency. If $h(x)$ is a function of space, then the Fourier transform is a function of wavenumber. The *power spectrum density* (PSD) of function $h(t)$ is defined to be

$$PSD_h(f) = |H(f)|^2 + |H(-f)|^2, \quad 0 \leq f < \infty$$

which indicates the power of the signal in the frequency or wavenumber spectrum.

In the visualization window, a button is provided to estimate the power spectral density of the velocities of the particles across the space. The particles must be uniformly distributed in the space for the estimation to be meaningful. The estimation indicates the wave components that are supported by the particles. A power spectrum estimator using two segments of data from two consecutive time steps and Parzen data windowing to get a smooth estimation [46] is implemented. An example of the power spectrum output is shown in Figure 5.3.

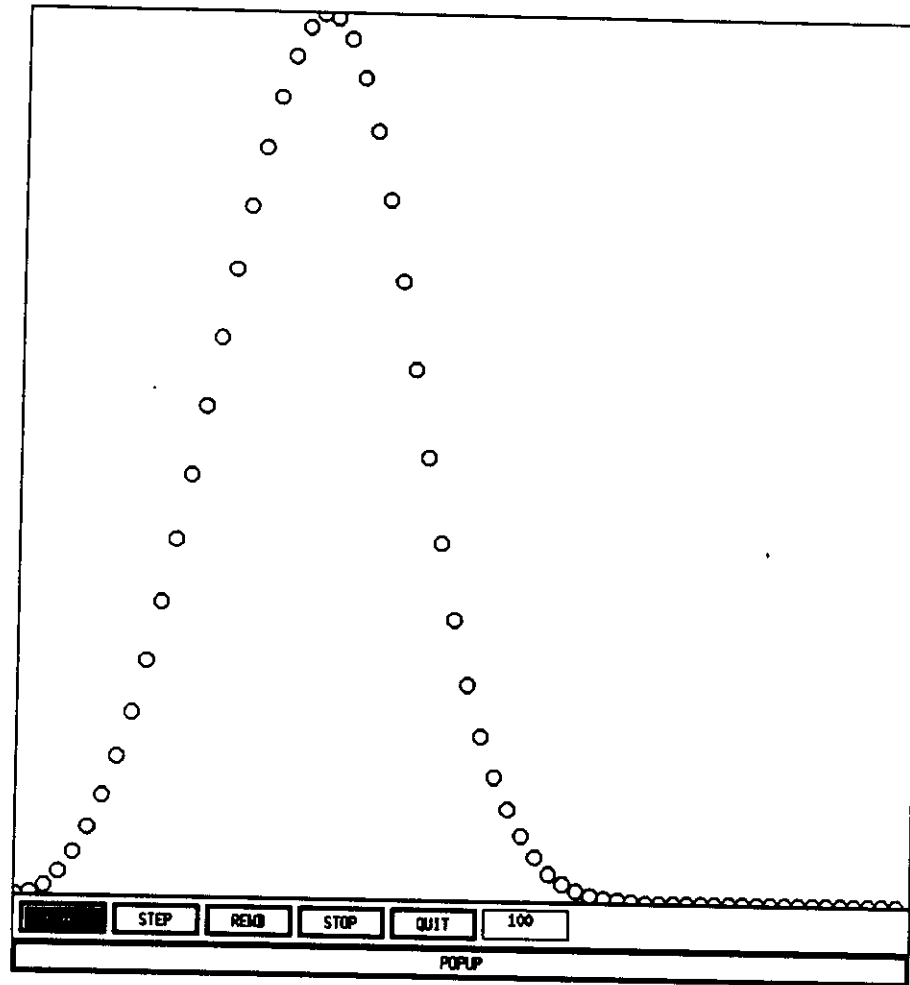


Figure 5.3: A power spectrum output

CHAPTER 6

Applications

6.1 Two Coupled Pendulums

This is to simulate two pendulums coupled by a spring, as shown in Figure 6.1. Each pendulum undergoes oscillatory motion under the influence of gravity and the coupling force from the spring. Since the arm holds the pendulum in a circular orbit, the motion of a pendulum is one-dimensional and can be described by the angle θ formed by the vertical line and the arm of the pendulum. The pendulums then behave like particles with the equations of motion

$$I_i \frac{d^2\theta_i}{dt^2} = (\mathbf{r}_i - \mathbf{r}_{i0}) \times [m_i \mathbf{g} - \kappa(1 - \frac{\xi}{r})(\mathbf{r}_i - \mathbf{r}_{3-i})], \quad (i = 1, 2)$$

where $I_i = m_i l_i^2$ is the moment of inertia of the pendulums, $r = |\mathbf{r}_i - \mathbf{r}_{3-i}|$, ξ the original length of the spring, and κ the spring constant. The first term on the right-hand side is the torque produced by gravity, which is independent of the other pendulum. The second term on the right-hand side is the torque caused by the spring force which is a function of the separation of the two pendulums. Hence, a pendulum can be specified in a way a Newton particle is specified, as in

```
class Pendulum: INHERIT(Newton,  
    Vector pivot;  
    double mass;  
    double length;  
    Pendulum(Vector& theta, Vector& v, double m, double l,
```

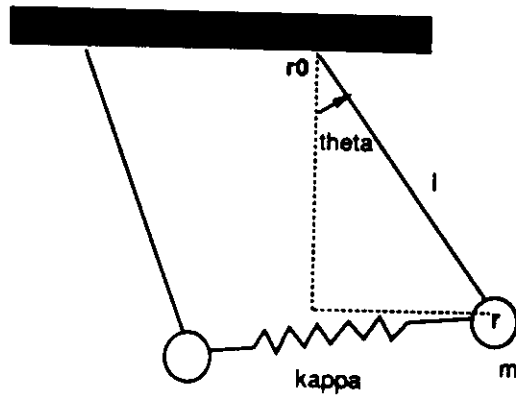


Figure 6.1: Two coupled pendulums

```

        Vector& r): Newton(theta, v, m*l*l)
        {mass = m; length = l; pivot = r;}
    );

    Pendulum p1(Vector( $\theta_1$ ), Vector( $\dot{\theta}_1$ ), m1, l1, r10);
    Pendulum p2(Vector( $\theta_2$ ), Vector( $\dot{\theta}_2$ ), m2, l2, r20);

    Internal(p1, inforce);
    Internal(p2, inforce);
    External(p1, exforce);
    External(p2, exforce);

```

where `inforce` is the inter-particle force function defined as

```

Vector inforce(Pendulum& a, Pendulum& b)
{
    Vector r = Vector(x(b), y(b)) - Vector(x(a), y(a));
    double d = magnitude(r);

    r **= - $\kappa$ *(1- $\xi$ /d);
    return cross(Vector(x(b), y(b))-b.pivot, r);
}

```

and `exforce` is the external force function defined as

```

Vector exforce(Pendulum& a)
{
    Vector vg(0, -a.mass*g);
    return cross(Vector(x(a), y(a))-a.pivot, vg);
}

```

Functions `x` and `y` return the x and y coordinates of a pendulum, respectively. They are defined as

```

double x(Pendulum& p) {return p.pivot[0]+p.length*
    sin(Location(p)[0]);}
double y(Pendulum& p) {return p.pivot[1]-p.length*
    cos(Location(p)[0]);}

```

The switch to Cartesian coordinate from Spherical coordinate was necessary because Spherical coordinate had not been implemented yet.

After the pendulums have been specified, the next step is to specify a Scheme object and to load the pendulums into the scheme. Then, running the scheme with a visualization window would start the simulation, as shown in

```

PPscheme scheme(LowerBound, UpperBound);

Loas(scheme, &p1, 1);
Loas(scheme, &p2, 1);
RunSwindow(scheme, step_size, number_of_steps);

```

The complete program is listed in Appendix A.1. Figure 6.2 is a picture of the traces of the pendulums at time step 100.

6.2 Wave Propagation in a Rod

Wave propagation on an one-dimensional medium can be described by

$$\frac{\partial^2 \phi}{\partial t^2} = c^2 \frac{\partial^2 \phi}{\partial x^2} \quad (6.1)$$

and the initial condition

$$\phi(x, 0) = u(x), \quad \frac{\partial}{\partial t} \phi(x, 0) = v(x) \quad (6.2)$$

Effecting a Fourier transform and solving an ordinary differential equation leads to

$$\hat{\phi}(\xi, t) = \hat{u}(\xi) \cos(c\xi t) + \frac{\hat{v}(\xi)}{c\xi} \sin(c\xi t) \quad (6.3)$$

where $\hat{\phi}(\xi, t)$ is the Fourier transform of $\phi(x, t)$ along x . Now, let us see how the wave propagation is described by a particle model. Consider that the medium is composed of particles of lumped mass m , separated by a distance l , and connected by springs with spring constant κ (Figure 6.3). Then, the force on a particle of index p is

$$-\kappa(\phi_p - \phi_{p-1}) - \kappa(\phi_p - \phi_{p+1})$$

where ϕ_p is the displacement of particle p from its equilibrium position. From Newton's law,

$$m \frac{d^2 \phi_p}{dt^2} = -\kappa(\phi_p - \phi_{p-1}) - \kappa(\phi_p - \phi_{p+1})$$

Let $\omega_0^2 = \kappa/m$, then

$$\frac{d^2 \phi_p}{dt^2} = \omega_0^2 (\phi_{p-1} - 2\phi_p + \phi_{p+1}) \quad (6.4)$$

Taking the discrete Fourier transform along p and solving an ordinary differential equation, it is found that

$$\hat{\phi}_k(t) = \hat{u}_k \cos[2\omega_0 \sin(\pi k/2N)t] + \frac{\hat{v}_k}{2\omega_0 \sin(\pi k/2N)} \sin[2\omega_0 \sin(\frac{\pi k}{2N})t] \quad (6.5)$$

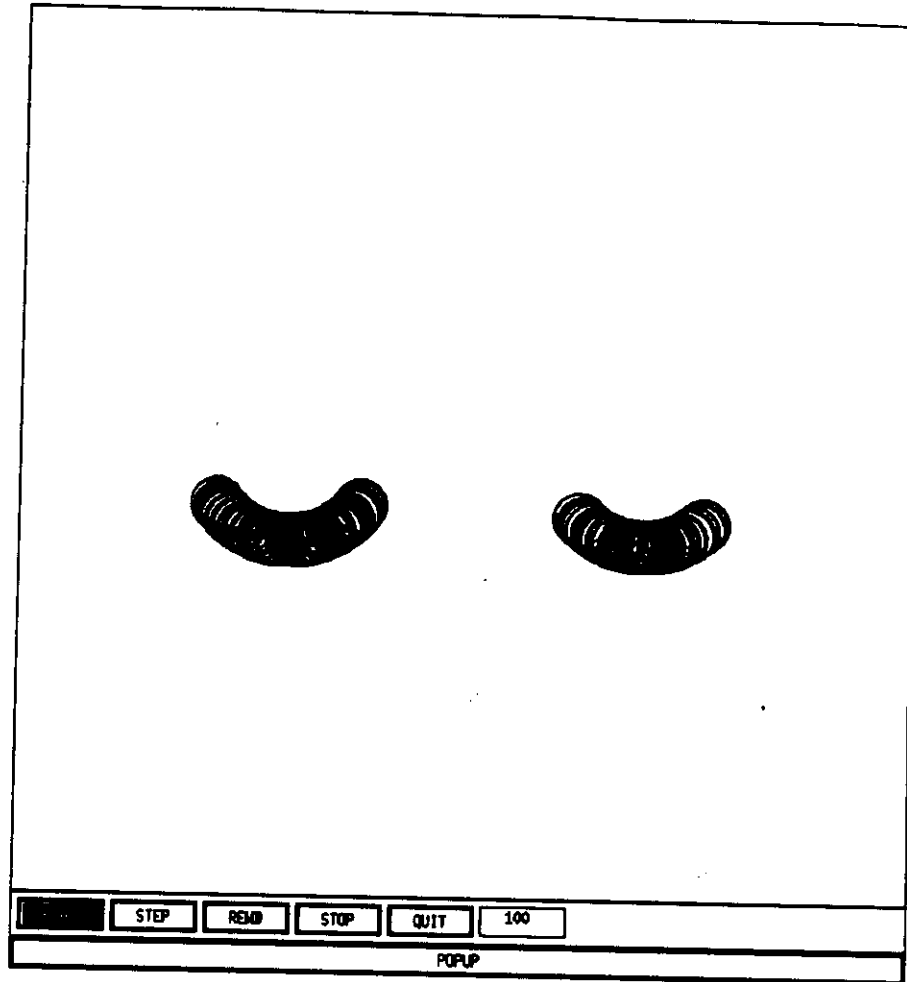


Figure 6.2: Two coupled pendulums

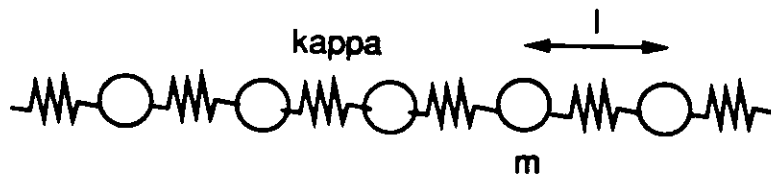


Figure 6.3: A particle model of a metal rod

where $2N$ is the number of particles used in the model and $\hat{\phi}_k(t)$ is the sample points of $\hat{\phi}(\xi, t)$ at interval π/Nl . When k is relatively small compared with N/π , then

$$\hat{\phi}_k(t) \approx \hat{u}_k \cos(2\omega_0 \frac{\pi k}{2N} t) + \frac{\hat{v}_k}{2\omega_0 \pi k / 2N} \sin(2\omega_0 \frac{\pi k}{2N} t) \quad (6.6)$$

Choose $\xi = k\pi/Nl$ in Equation (6.3) and compare it with Equation (6.6), it is seen

$$ck \frac{\pi}{Nl} = 2\omega_0 \frac{\pi k}{2N}$$

or

$$\omega_0 = \frac{c}{l}$$

Therefore, if $\omega_0 = c/l$, or

$$\frac{\kappa}{m} = \left(\frac{c}{l}\right)^2 \quad (6.7)$$

the discrete particles connected by springs quite faithfully simulate the wave propagation in a continuous medium described by Equation (6.1), provided that the wave number ξ is smaller than $(N/\pi)(\pi/Nl) = 1/l$ or the wavelength is larger than l . If the media is a metal rod, it is known that

$$c^2 = \frac{Y}{\rho}$$

approximately, where Y is Young's modulus and ρ is the mass density of the rod. From Equation (6.7), we have

$$\frac{\kappa}{m} = \frac{Y}{\rho l^2}$$

A natural choice of κ and m is then

$$m = \rho l, \quad \kappa = \frac{Y}{l}$$

If the rod has a cross section A , choose

$$m = Al\rho, \quad \kappa = \frac{AY}{l}$$

which is the case that the rod is divided into chunks of length l and each chunk is considered as a lumped particle, and the tension at the boundary between two chunks serve as the spring force

$$-\frac{AY}{l} \phi(x, t)$$

The programming of the simulation in Boltzmann is similar to that of the pendulum problem, except that interactions are only among neighboring particles. The complete program is listed in Appendix A.2. A phase-space snapshot of the metal rod at time step 100 is given in Figure 6.4.

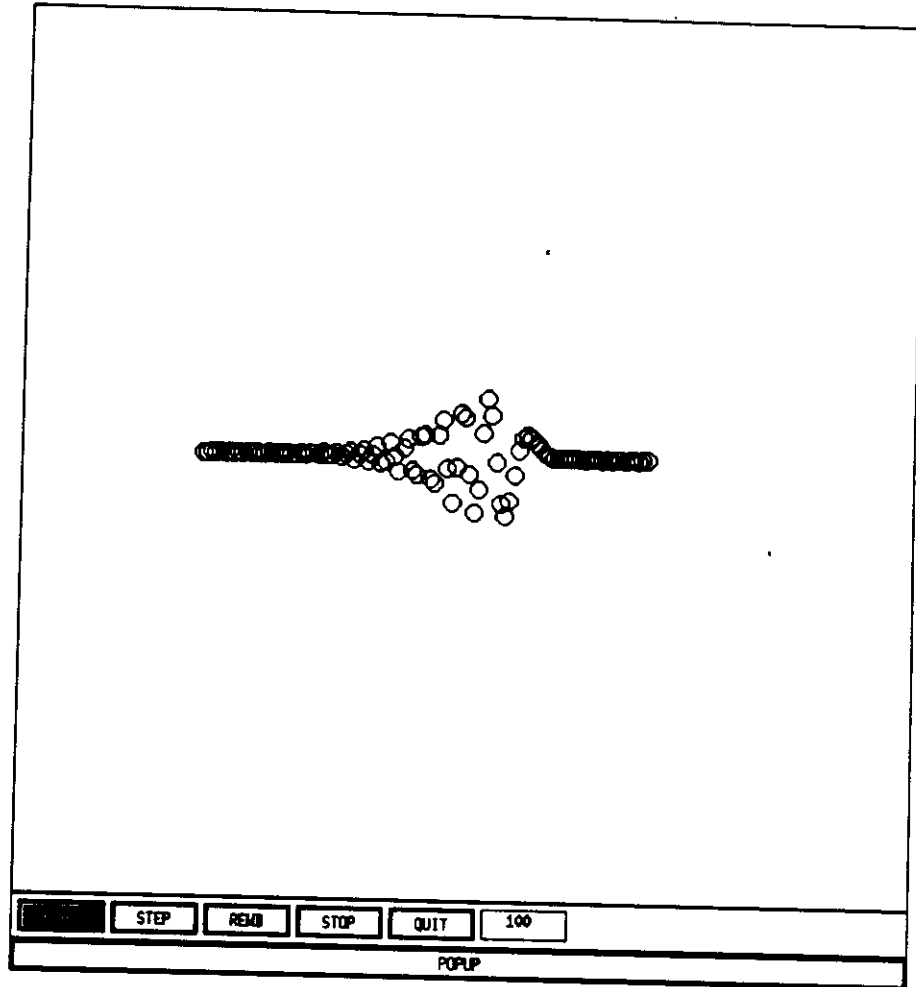


Figure 6.4: Wave propagation in phase-space in a metal rod

6.3 Radiation Displacement of Atoms

We now consider the simulation of the displacement of atoms in a metal under radiation [11]. A metal is modeled by a lattice of atoms and the interactions among the atoms. In our example, the interaction between two atoms is represented by the potential function of the separation of the atoms:

$$\phi(r) = \epsilon \left[\left(\frac{r_0}{r} \right)^{12} - 2 \left(\frac{r_0}{r} \right)^6 \right], \quad (6.8)$$

which is called Lennard-Jones potential; r_0 is the atomic separation at equilibrium and ϵ is the depth of the potential well.

The development of the simulation program starts with the selection of the particle class and the scheme class. Since the force between two atoms is short-ranged and decreases rapidly as the separation becomes large, the `PPscheme` class is selected because it allows the specification of a cut-off distance. Here, a piece of copper is defined in two dimensions and the cut-off distance of inter-particle force is set to be $3r_0$, as in

```
const R0 = 2.551 angstroms;
```

```
Vector low(0, 0), high(100 angstroms, 100 angstroms);  
PPscheme copper(low, high, 3*R0);
```

The atoms are defined to be Newton particles with Lennard-Jones force,

```
Vector force(Newton&, Newton&);  
Particle atoms[N];  
for (i= 0; i< N; i++)  
    Internal(atoms[i], force);
```

where N is the number of atoms used in the simulation and the force is defined by

```
const DEPTH = 0.1 electronvolts;  
  
Vector force(Particle& p, Particle& q)  
{  
    Vector u = Location(q) - Location(p);  
    double r = magnitude(u);  
    double a = R0/r;  
    a = a*a*a;  
    a **= a;  
    u **= 12*DEPTH*a*(a-1)/(r*r);  
    return u;  
}
```

Since the simulated atoms represent only a tiny fraction of the metal, a boundary condition has to be imposed. It can be incorporated in the simulation by specifying certain external force on the boundary particles. A simple approximation of the boundary condition is given by an external spring force, defined as

```

Vector external(Newton& p)
{
    Vector f = Location(p) - original[&p-atoms];
    f *= -kappa;
    return f;
}

```

where `original` is a vector array storing the initial positions of the particles. The complete program is listed in Appendix A.3. A snapshot of the metal at time step 100 is given in Figure 6.5, where the size of the circles represents the magnitude of velocity and thus the kinetic energy of the atoms.

6.4 Particle Simulation of Plasma

Plasma are hot gasses of ionized particles. The behavior of the particles obeys Maxwell's equations, which in theory are the complete particle model of plasma. Given the vast number of particles in a plasma, however, even the most powerful computers fall short of the computational requirement, except in cases where only a small region of the plasma is studied. Some kind of approximation is always needed regardless whether the simulation is done on a supercomputer or on a workstation computer or simply a personal computer.

A widely-used approximation in particle simulation uses the distribution density function $f(\mathbf{r}, \mathbf{v}, t)$ of the particles, where

$$dn = f(\mathbf{r}, \mathbf{v}, t) d\mathbf{r} d\mathbf{v}$$

is the number of particles in the infinitesimal area of dimension $d\mathbf{r}$ and $d\mathbf{v}$ centered at \mathbf{r} and \mathbf{v} in the phase space $\{(\mathbf{r}, \mathbf{v})\}$ at time t . Boltzmann's transport equation gives

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{r}} + \frac{\mathbf{F}}{m} \cdot \frac{\partial f}{\partial \mathbf{v}} = \left(\frac{\partial f}{\partial t}\right)_c$$

or

$$\frac{df}{dt} = \left(\frac{\partial f}{\partial t}\right)_c \quad (6.9)$$

where df/dt is the rate of change of the density function along the path of particles defined by Newton's law

$$\begin{cases} d\mathbf{r}/dt = \mathbf{v} \\ d\mathbf{v}/dt = \mathbf{F}/m \end{cases} \quad (6.10)$$

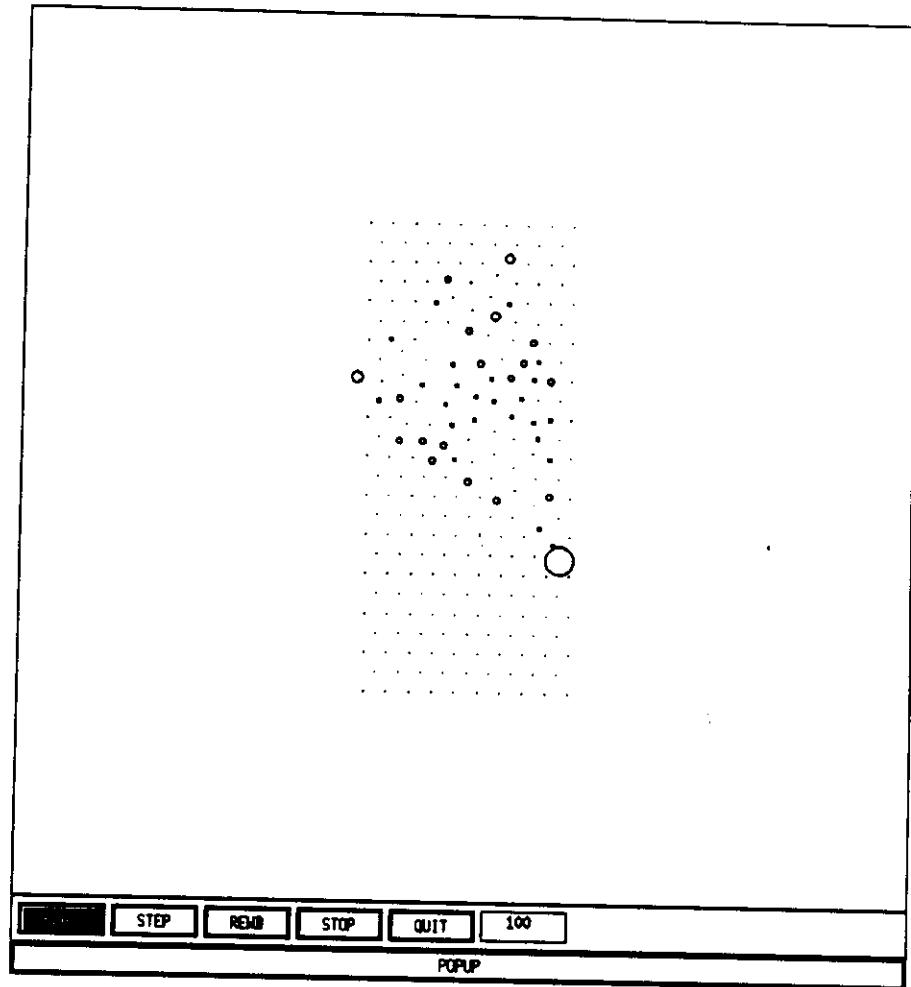


Figure 6.5: Radiation displacement of atoms

Equation (6.9) and (6.10) are continuous functions describing the change of $f(\mathbf{r}, \mathbf{v}, t)$. If the phase space is discretized into finite elements and the particles in every element are treated as super-particles. Then a reduced particle model of the plasma is produced. Given the generality of the above derivation, the approach applies equally well to the construction of many other applications. It illustrates the reductionist approach often seen in particle simulation where a statistical description, which is generated from a microscopic particle model of the system, is reduced back to a particle model of manageable size that can be solved by available computers.

Returning to the plasma simulation, if only the high frequency components of the plasma and a small space scale is the focus of the simulation, then the force function is adequately described by the electrostatic force, and the particles fall into Coulomb Particle class. The definition of the particles can be simply

```
Coulomb particles[N];
```

followed by the assignment of mass, initial position and velocity or by the Maxwellian distribution according to the temperature of the plasma, as illustrated below for a one-dimensional simulation.

```
Uniform rand(0, L);
Maxwellian maxv(Temp, Mass);
for (i= 0; i< N; i++) {
    Position(particles[i], Vector(rand()));
    Velocity(particles[i], Vector(maxv()));
    Mass(particles[i], Mass);
}
```

where Uniform and Maxwellian are two random variable classes defined in Chapter 2. The program segment defines N particles randomly distributed in the space $[0, L]$ at the temperature Temp.

In our simulation, two hot streams of electrons are shot in opposite directions into a stream of positively charged ions uniformly distributed in space. Their initial velocities are determined according to the Maxwellian distribution at a temperature of $16,000^\circ K$. The complete program is listed in Appendix A.4. A phase-space snapshot at time step 100 is given in Figure 6.6 which illustrates the vortices formed in the phase space and the instability predicted by theory.

6.5 Vortex Simulation of Fluid Flow

As we have already discussed, vortices can be used to describe fluid flow. The idea is to "induce" a flow field by placing vortices accordingly, although there is no cause-and-effect relation between the flow field and the vortices; vortex is only a mathematical concept, i.e., the curl of the velocity field at a point in the space. In this example, an inviscid flow passing around an infinitely long circular cylinder whose axis is perpendicular to the direction of the flow is simulated (Figure 6.7).

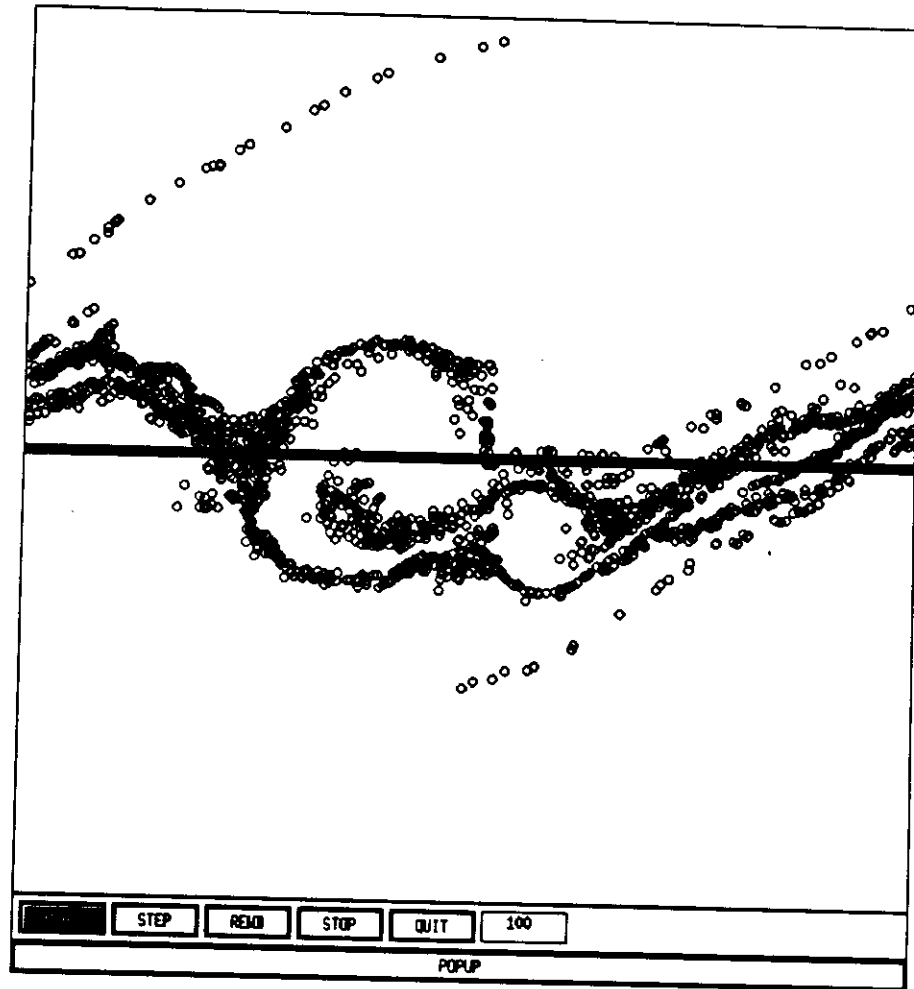


Figure 6.6: Particle simulation of plasma in phase-space

Two vortices of opposite sign are placed at large distance above and below the cylinder to create the horizontal flow from left to right. The cylinder can be simulated by placing image vortices in the region occupied by the cylinder to create a non-flow boundary [47], that is, the normal component of the velocity at the boundary of the cylinder is zero. Since vortices also move with the flow, one way to produce a steady flow is to nullify the integration step of the vortices, as shown in

```
class DeadVortex: INHERIT(Vortex,  
    int integrate(double t, int i) return 0;  
);
```

or simply

```
class DeadVortex: DEACTIVATE(Vortex);
```

both of which are equivalent. A number of test particles are deposited to follow the flow. The complete program is listed in Appendix A.5. Figure 6.7 shows the traces of a number of test particles.

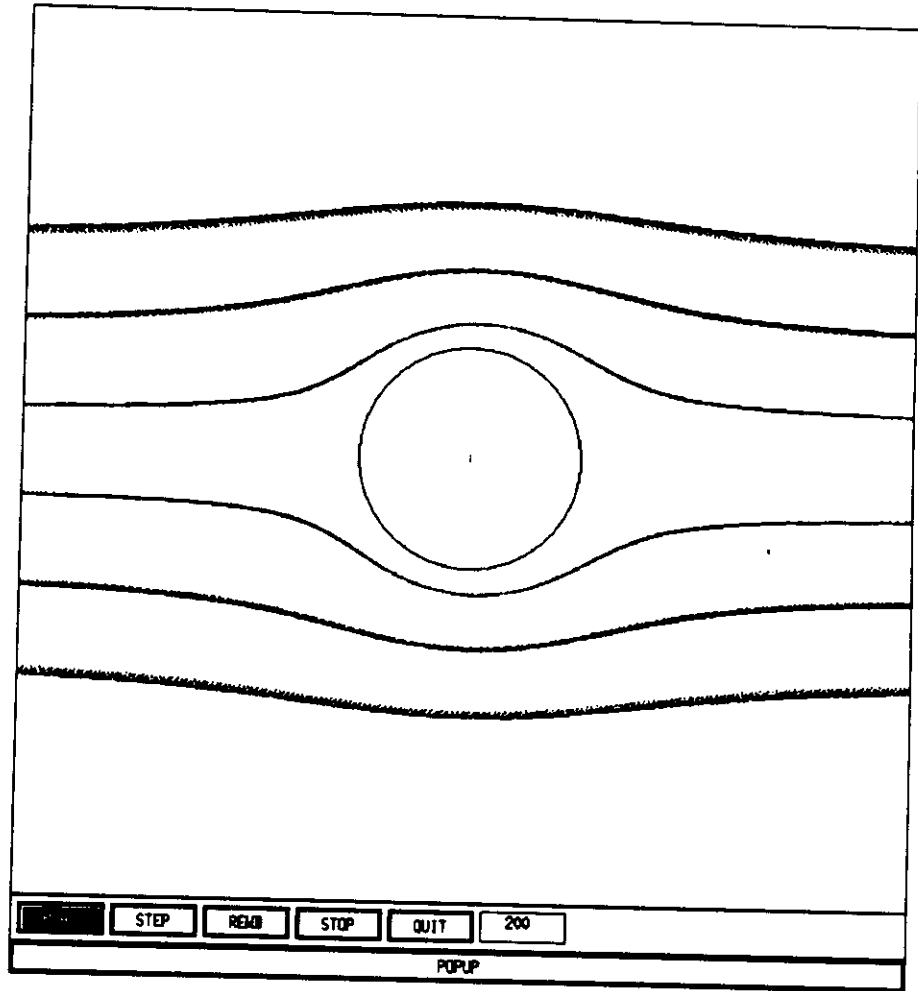


Figure 6.7: Vortex simulation of fluid flow

CHAPTER 7

Performance Evaluation

In this chapter, we study the performance of the Boltzmann programming system in term of its memory requirement and computation efficiency. It should be pointed out that the performance study is based on the current implementation, and therefore the results are not absolute.

7.1 Memory Requirement

Table 7.1 is a comparison of the memory requirements of some data types and particle classes. It gives a rough idea how much memory is required for a simulation. The table doesn't list the memory requirements for schemes and other classes because, unlike particles, they are not used in a large number of copies in a single simulation. For the Vector Class and all particle classes, a range is given instead of a fixed number. The actual number of bytes required depends on the dimension of the problem. For example, a vector of three dimensions occupies 20 bytes, while an uninitialized vector takes only 8 bytes. The lower numbers in the ranges reflect static memory allocation and the numbers larger than them are due to dynamic memory allocation.

7.2 "Garbage" Collection

Vector operations create many intermediate vectors. Because memory for vectors is dynamically allocated, it is possible for large amount of memory allocated for intermediate vectors to become unaccessible. Those unusable memories are called "garbage". It had happened many times that a simulation run out of memory before a garbage-collection mechanism is installed. A simple garbage collector that records the addresses of memories for intermediate vectors at every time step and releases the memories at the next time step has worked fine.

7.3 Computation Efficiency

Programming in a high-level programming system or language often brings some overhead in computation time, despite the fact that modern compilers and architectures dramatically bring down the overhead. It is a matter that how much computation time has to be sacrificed to achieve the ease of programming and other benefits offered by the high-level programming system or language. When a program can be run in a "reasonable" time, the benefits of programming in a

<i>data type / class</i>	<i>number of bytes</i>
integer	4
float	4
double	8
Vector	8 - 20
Particle	56 - 92
Coulomb	56 - 92
Electrostatic	64 - 100
Fluid	56 - 92
Vortex	72 - 120

Table 7.1: Memory requirements

high-level language can not be surpassed by insignificant computation time saved by programming in a low-level language. When every FLOP (floating-point operations) is needed to solve a problem, however, we are left with no choice but to employ every “greedy” method that get the job done, and anything else has to give way to computation efficiency.

The Boltzmann programming system introduces computation overhead mainly from three sources: the general-purpose nature of the programming system in particle simulation, indirect function calls, and dynamic memory management. Unlike hand-coding an application in a low-level language, a programming system has to allow uncertainty about possible applications such as the dimensionality of a problem and what integration algorithms are used. Some of the uncertainties are handled by storing parameters in objects and others are handled by indirect function calls. Fortunately, most of overheads are eliminated at the compilation stage and have no effect on the run-time speed. For those that do smuggle into the run-time, modern computer architectures and optimizing compilers minimize their damage to computation efficiency.

To measure the overhead of Boltzmann in computation time, a three-body problem is implemented in the Boltzmann programming system and the regular C language separately. The problem is to simulate three massive objects under the influence of gravitational forces. The program in Boltzmann is listed in Appendix A.6 and the one in C is listed in Appendix A.7. The story of timing the two simulations is informative and hence is presented here. Both simulations were run 1000 time steps. At the first run, the Boltzmann version takes 5.3 seconds CPU time while the C version takes only 0.6 second CPU time. Compiling both programs with optimization, the Boltzmann version runs 3.6 seconds and the C version runs 0.4 second. By examining the Boltzmann version, it is found that the fourth-order Runge-Kutta integration takes about 72% of the computation time. That is because the integration step is expressed in such vector operations that

<i>task</i>	<i>Boltzmann (sec.)</i>	<i>C (sec.)</i>
integration	0.4	0.2
force calculation	0.4	0.2
force summation	0.3	
function calls	0.1	
others	0.2	
TOTAL	1.4	0.4

Table 7.2: Comparison of time consumptions

<i>example</i>	<i>number of particles</i>	<i>real-time</i>
coupled pendulums	2	13.75"
wave propagation	100	16.53"
atomic displacement	250	1'30"
plasma	4000	2'14"
fluid flow	126	31.23"

Table 7.3: Real-time spent in the examples

constantly create and delete intermediate vectors at each time step which draw the slower dynamic memory manager into the integration cycle. After rewriting the integration in a way that does not create intermediate vectors, the Boltzmann version simulation takes now only 1.4 seconds. Although it is still about 3 times slower than the C version, it demonstrates the kind of improvement that can be achieved by fine-tuning the programming system instead of the user's program. Now, the integration step takes 0.4 second, or 28% of the total computation time. About 0.7 second, or 50%, of the computation time goes to the force calculation. The rest 0.3 second, or 21% is due to function calls and other overheads. Out of the 1.4 second total computation time, vector arithmetic consumes 0.2 second, or 14%, respectively, and vector assignments consume 0.8 second, or 57%. Therefore, any improvement in the vector operations will dramatically speed up the computation. A comparison of time consumptions by the Boltzmann program and the C program is listed in Table 7.2.

Another way to look at the computation efficiency is to measure the real-time the examples in Chapter 6 take. Run with the visualization on a Sun Sparcstation. the five examples take the times shown in Table 7.3 to complete 100 time steps. From the figures in the table, it can be concluded that the Boltzmann programming system is reasonably fast.

CHAPTER 8

Conclusion

As we have seen, the object-oriented particle simulation methodology provides a novel approach to physical system simulations, differing from the traditional function evaluation approach. It provides a unifying modeling and simulation framework for a variety of simulation applications with the use of particle methods. Its emphasis on modularity resembles the real-world scenario of particles and particle interactions, and therefore, allows easy composition of simulation programs from predefined software modules. Its hierarchical organization utilizes the connection among particles and schemes to facilitate software reusability. In this chapter, the contributions from the object-oriented particle simulation methodology and the Boltzmann programming system are summarized, followed by what can be done in the future to improve them. The final section contains reflections upon the implications of this research.

8.1 The Contributions

The object-oriented particle simulation methodology and the Boltzmann programming system have the following contributions:

1. Although particle methods have been used here and there to simulate physical systems, the OOPS methodology offers a systematic approach to particle simulations that encourages the programming of simulations in an object-oriented style. By doing so, it allows fast composition of particle simulation programs. Its emphasis on object orientation, modularity, and software reusability makes particle simulation programming more manageable and less vulnerable to programming errors. More specifically, the contributions of the OOPS methodology include:
 - (a) The separation of local computations (particles) from global computations (schemes) and the establishment of an object-oriented programming framework for particle simulations.
 - (b) The separation of functionalities in particles and schemes so that any of the functionalities can be replaced by new definitions (replaceable components).
 - (c) The identification of particles with Green's functions so that a variety of phenomena can be simulated in particle methods.
 - (d) The classification of particles in a hierarchy that is conceptually clean and robust and best facilitates software reusability.

2. The Boltzmann programming system realizes the OOPS methodology on the C++ platform, offering a complete high-level programming environment from initialization to visualization for particle simulation. Specifically, the Boltzmann programming system is unique in the following aspects:
 - (a) Boltzmann supports vector arithmetic.
 - (b) In Boltzmann, almost every concept is represented by a class of objects. Examples of objects include vectors, random variables, particles, schemes, boundaries, and windows. The initialization, control, and visualization of simulations are carried out on the level of particles and schemes.
 - (c) Visualization of simulation results is conducted in a visualization window, which facilitates friendly human-machine interactions.
 - (d) A set of macro definitions for particle derivation is provided to hide the syntactic details of the underlying programming language C++.
 - (e) A set of macro definitions is provided to support automatic unit conversion.

8.2 Further Improvement

It is expected that more and more simulations will be done in particle methods. The Boltzmann programming system can be further improved to accommodate more sophisticated requirements and to achieve better performance, for example, in the following areas:

1. More coordinate systems; the current version of Boltzmann supports only the Cartesian coordinate system. There is no reason it cannot support other coordinate system such as Spherical and Cylindrical coordinates.
2. Higher dimensionality; the current version supports only one-dimensional and two-dimensional simulation. The extension to three dimensions is straightforward, except at the visualization stage where 3D visualization is required.
3. More physics; the current version supports only the classical physics. It is not clear yet how well quantum physics can be represented by particle models.
4. More particle and scheme classes; more particle classes can be added to support more varieties of particle simulations and more scheme classes can be added to support more computational methods.
5. More abstract data types; beside particles and schemes, the current version has two data types designed for scientific computing in general and particle simulation in specific: vectors and random variables. Other data types that may be considered useful include matrices, tensors, and four-vectors.

6. More diagnosis functions; many useful diagnosis functions can be installed.
7. More visualization windows; the `Swindow` presented in Chapter 5 is only an example. Other arrangements of drawing boards and buttons work just as well. Notably, at least one window should allow running the simulation in background and saving the results in files that can be visualized at a later time.
8. Better user-interface; ideally, a user should be able to compose a particle simulation by dragging icons and clicking buttons on the screen, or at least by typing some simple commands, instead of having to write a C program.
9. Efficient implementation and parallel processing; as illustrated in Section 7.3, there is plenty of room for performance improvement. It is one of the strongest points for the OOPS approach that improvement in the implementation of the programming system requires no modification of the users' programs. It is also true for parallel processing. As an example, the parallel implementation of the Scheme class should not concern the end users except it offers a different execution speed. Because of that, the Boltzmann programming system is highly portable and its implementation is transparent to the users.
10. High-order logic; the current C++ language that supports Boltzmann is only a first-order language. Although it allows classes to be defined on top of other classes, they have to be done statically at compilation time. High-order logic should allow class derivation and function composition dynamically at run time. In higher-order languages, classes and functions may not be very different. The current Boltzmann programming system does most of its class derivations by macro definitions. Higher-order logic certainly adds more flexibilities to the implementation of Boltzmann.

8.3 Implications

The success of the object-oriented particle simulation methodology would help to create a new paradigm in physical system simulations, in which simulations are constructed from predefined software components, and a software industry that manufactures off-the-shelf software modules for simulations. One of the challenges to that end lies in the requirement that the software industry has to agree upon the interfaces among different software modules, and hopefully this dissertation has contributed some valuable experience. Another challenge is to cast the large number of existing numerical techniques and algorithms into the new object-oriented paradigm in such a way that the details of the algorithms are not more important to the users than the internal physics of IC chips to hardware system designers, as the Boltzmann programming system has demonstrated in the construction of the

particles and the schemes. It requires a substantial understanding of the physics of the problem domain and the mathematics of the numerical methods and algorithms from the software industry.

It is expected that, in the future, both scientists and high school students can order their simulation “parts” and construct simulations on their own computers. By doing experiments on computers, they can gain valuable insight about the subjects they study, supplementing their theoretical abstraction and hands-on real-world experimentation.

Bibliography

- [1] B. J. Alder and T. E. Wainwright. Studies in molecular dynamics. *Journal of Chemical Physics*, 13:459-466, 1959.
- [2] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. Technical Report CAM-90-14, UCLA Computational and Applied Mathematics, 1990.
- [3] Apple Computer, Inc. *Inside Macintosh*. Addison-Wesley, 1985.
- [4] George Arfken. *Mathematical Methods for Physicists*. Academic Press, 1970.
- [5] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.
- [6] Charles K. Birdsall and A. Bruce Langdon. *Plasma Physics Via Computer Simulation*. McGraw-Hill, 1985.
- [7] O. Buneman. Dissipation of currents in ionised media. *Physics Review*, 115:503-517, 1959.
- [8] A. F. Cardenas and W. J. Karplus. PDEL — a language for partial differential equations. *Communications of ACM*, 13:184-191, March 1970.
- [9] R. Chen and D. S. Pan. Mobility model in semiconductor device simulators. Technical report, Electrical Engineering Department, University of California, Los Angeles, 1989.
- [10] S. P. Chou and N. M. Ghoniem. Precipitate dissolution by high energy collision cascades. *Journal of Nuclear Materials*, 117:55-63, 1983.
- [11] S. P. Chou and N. M. Ghoniem, 1990. personal communication.
- [12] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, 1990.
- [13] R. Courant and D. Hilbert. *Methods of Mathematical Physics*. John Wiley and Sons, 1962.
- [14] B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.
- [15] O. J. Dahl, B. Myrhaug, and K. Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, 1984.

- [16] Philip J. Davis and Reuben Hersh. *Descartes' Dream, the World According to Mathematics*. Houghton Mifflin Company, 1986.
- [17] John M. Dawson. One-dimensional plasma model. *Physics of Fluids*, 5:445-459, 1962.
- [18] John M. Dawson. Particle simulation of plasmas. *Reviews of Modern Physics*, 55(2):403-447, April 1983.
- [19] J. B. Gibson, A. N. Goland, M. Milgram, and G. H. Vineyard. Dynamics of radiation damage. *Physical Review*, 120(4), November 1960.
- [20] Nevel T. Gladd. Object-oriented interface system for particle-in-cell simulations. Technical report, Jaycor Corporation, 1989.
- [21] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [22] Herman H. Goldstine. *The Computer, from Pascal to von Neumann*. Princeton University Press, 1972.
- [23] Danny Goodman. *The Complete Hypercard Handbook*. Bantam Books, 2nd edition, 1988.
- [24] B. Goplen, R. E. Clark, et al. MAGIC user's manual. Technical Report MRC/WDC-R-126, Mission Research Corporation, April 1987.
- [25] Harvey Gould and Jan Tobochnik. *An Introduction to Computer Simulation Methods*. Addison-Wesley, 1988.
- [26] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325-348, 1987.
- [27] Donald Greenspan. *Computer-Oriented Mathematical Physics*. Pergamon Press, 1981.
- [28] Samuel P. Harbison and Guy L. Steele, Jr. *C: A Reference Manual*. Prentice-Hall, 2nd edition, 1987.
- [29] D. R. Hartree. Some calculation of transients in an electronic valve. *Applied Science Researches*, pages 379-390, 1950.
- [30] Roger W. Hockney and James W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill International Book Company, 1981.
- [31] Walter J. Karplus, editor. *Peripheral Array Processors*, volume 11 of *Simulation Series*. Society for Computer Simulation, 1982.

- [32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd edition, 1988.
- [33] T. W. Körner. *Fourier Analysis*. Cambridge University Press, 1988.
- [34] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of ACM*, pages 40–60, September 1990.
- [35] A. Daniel Kowalski. An object oriented prolog representation of quasilinear partial differential equations. In *Proceedings of the International Symposium on AI Expert Systems and Languages in Modelling and Simulation*. IMACS, June 1987.
- [36] L. D. Landau and E. M. Lifshitz. *Statistical Physics, Part 1*, volume 5 of *Course of Theoretical Physics*. Pergamon Press, 3rd edition, 1980.
- [37] Doug Lea. *User's Guide to GNU C++ Library*. Free Software Foundation, Inc., 1989.
- [38] M. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with interviews. *Computer*, February 1989.
- [39] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, 1989.
- [40] Brendan McNamara. Supercomputer environments for science applications. *Journal of Computational Physics*, 73:41–58, 1987.
- [41] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [42] Bertrand Meyer. Eiffel: The language. Technical Report TR-EI-17/RM, Interactive Software Engineering, Inc., 1989.
- [43] P. N. Morse and H. Feshbach. *Methods of Theoretical Physics*. McGraw-Hill, 1953.
- [44] A. J. Palay, W. J. Hansen, et al. The andrew toolkit – an overview. In *USENIX Association Winter Conference*, Dallas, 1988.
- [45] David Potter. *Computational Physics*. John Wiley and Sons, 1973.
- [46] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C, the Art of Scientific Computing*. Cambridge University Press, 1988.
- [47] Elbridge G. Puckett. An introduction to the random vortex method with vorticity creation. *Lectures in Applied Mathematics*, 1990.

- [48] A. Rahman. Correlations in the motion of atoms in liquid argon. *Physics Review*, 136:405–411, 1964.
- [49] Mark F. Russo, Richard L. Peskin, and A. Daniel Kowalski. A prolog based expert system for modeling with partial differential equations. *Simulation*, 46(4):150–157, 1987.
- [50] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [51] K. Schmucker. *Object Oriented Programming for the Macintosh*. Hayden, 1986.
- [52] Simulation Councils, Inc. The SCi continuous system simulation language (CSSL). *Simulation*, pages 281–292, December 1967.
- [53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [54] Tashiki Tajima. *Computational Plasma Physics*. Addison-Wesley, 1989.
- [55] V. Vemuri and Walter J. Karplus. *Digital Computer Treatment of Partial Differential Equations*. Prentice-Hall, 1981.
- [56] Otis R. Walton. Understanding particulate flow behavior. *Fossil Energy*, pages 6–22, September 1988.
- [57] Rebeccas J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of ACM*, pages 104–124, September 1990.

APPENDIX A

Example Programs

A.1 Two Coupled Pendulums

```
#include <Boltzmann/PPscheme.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/Macro.h>
#include <Boltzmann/Swindow.h>
#include <Boltzmann/Integrator.h>

const double m0      = 1;
const double l0      = 1;
const double g       = 10;
const double L       = 10;
const double kappa   = 0.8;
const double delta   = 4;

class Pendulum: INHERIT(Newton,
    Vector pivot;
    double mass;
    double length;
    Pendulum(Vector& theta, Vector& v, double m, double l,
        Vector& r): Newton(theta, v, m*l*l)
        {mass = m; length = l; pivot = r;}
);

double x(Pendulum& p)
    { return p.pivot[0]+p.length*sin(Location(p)[0]); }
double y(Pendulum& p)
    { return p.pivot[1]-p.length*cos(Location(p)[0]); }

main()
{
    Swindow window(Vector(0, L), Vector(0, L));
    PPscheme space(Vector(0), Vector(L));
    Pendulum p1(Vector(0.3*PI), Vector(0), m0, 10, Vector(3, 5));
    Pendulum p2(Vector(0), Vector(0), m0, 10, Vector(7, 5));
```

```

    Vector exforce(Pendulum&);
    Vector force(Pendulum&, Pendulum&);
    void draw(Swindow&, Scheme&);

    Diameter(p1, 0.6);
    Color(p1, Green);
    External(p1, exforce);
    Internal(p1, force);

    Diameter(p2, 0.6);
    Color(p2, Yellow);
    External(p2, exforce);
    Internal(p2, force);

    Load(space, &p1, 1);
    Load(space, &p2, 1);
    InstallDraw(window, "2D", draw);
    Run(window, space, 0.1);
}

Vector exforce(Pendulum& b)
{
    double f = -b.mass*g*b.length*sin(Location(b)[0]);

    return Vector(f);
}

Vector force(Pendulum& a, Pendulum& b)
{
    Vector f = Vector(x(b), y(b))-Vector(x(a), y(a));
    double d = magnitude(f);

    f **= (d-delta)/d;
    f **= -kappa;
    return cross(Vector(x(b), y(b))-b.pivot, f);
}

void draw(Swindow& sw, Scheme& scheme)
{
    Pendulum *p;

    for (int i= 0; i< Num(scheme); i++) {
        p = (Pendulum *) &scheme[i];
    }
}

```

```

        Load(rod, ps, N);
        MakeConnection(rod, 0, N-1, 1);

        RunSwindow(rod, t);
    }

    Vector force(Newton& a, Newton& b)
    {
        static double kappa = A*Y/l;

        Vector r = Location(b)-Location(a);

        if (r[0] > 0) r -= Vector(1);
        else if (r[0] < 0) r += Vector(1);
        r *= -kappa;

        return r;
    }

```

A.3 Radiation Displacement of Atoms

```

#include <Boltzmann/PPscheme.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/Swindow.h>
#include <Boltzmann/Unit.h>
#include <Boltzmann/Macro.h>
#include <Boltzmann/Integrator.h>
#include <Boltzmann/Function.h>

#define centimeter 1/1.804*1e8
#define second 1/3.273*1e15
#define gram 1/5.275*1e26

const double DEPTH      = 0.1 electronvolts;
const double R0         = 2.551 angstroms;
const double kappa     = 0.2 *electronvolt/(angstrom*angstrom);
const int N             = 250;
const int M             = 2000;

class LFparticle: LF(Newton);
class Atoms: INHERIT(LFparticle,

```

```

        DrawPoint(sw, Vector(x(*p), y(*p)),
                  Diameter(scheme[i]), Color(scheme[i]));
    }
}

```

A.2 Wave Propagation in a Rod

```

#include <Boltzmann/PPscheme.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/Function.h>
#include <Boltzmann/Swindow.h>
#include <Boltzmann/Integrator.h>

const double A      = 1;
const double l      = 0.05;
const double rho    = 8.96e3;
const double Y      = 12e10;
const int N         = 100;
const double L      = 10;

class RKparticle: RK4(Newton);

main()
{
    PPscheme rod(Vector(0), Vector(L));
    RKparticle ps[N];
    Vector force(Newton&, Newton&);

    double m = A*l*rho;
    double t = 1e-5;
    int i;

    for (i= 0; i< N; i++) {
        Location(ps[i], Vector(2+i*1));
        Velocity(ps[i], Vector(0));
        Mass(ps[i], m);
        Diameter(ps[i], 0.2);
        Color(ps[i], Pink);
        Internal(ps[i], force);
    }
    Location(ps[0], Location(ps[0])+Vector(0.01));
    Deactivate(ps[N-1]);
}

```

```

        // simulation in a Swindow
        RunSwindow(copper, 0.2);
    }

    Vector force(Atoms& p, Atoms& q)
    {
        Vector u = Location(q);

        u -= Location(p);
        double r = magnitude(u);
        double a = R0/r;
        a = a*a*a;
        a **= a;
        u **= 12*DEPTH*a*(a-1)/(r*r);
        return u;
    }

    Vector external(Atoms& p)
    {
        Vector f = Location(p);

        f -= r0[&p-base];
        f **= -kappa;
        return f;
    }

```

A.4 Particle Simulation of Plasma

```

#include <Boltzmann/PMscheme.h>
#include <Boltzmann/Coulomb.h>
#include <Boltzmann/RanVar.h>
#include <Boltzmann/Swindow.h>

const double eCHARGE    = 4.8e-10;
const double Me         = 0.91e-27;
const double M          = 1.67e-24;
const double T          = 16e3;
const double KB         = 1.38054e-16;
const int Np            = 2000;
const double L          = 1e6;
const int Ng            = 128;
const int Ns            = 10;

```

```

        void update(double t) {d = 8*magnitude(v) angstroms;}
    );

    Vector r0[N];
    Atoms *base;

    main()
    {
        Vector force(Atoms&, Atoms&);
        Vector external(Atoms&);
        Vector low(0, 0), high(100 angstroms, 100 angstroms);
        int i, t;

        PPscheme copper(low, high, 3*R0);
        Atoms atoms[N];

        // initialization
        for (i= 0; i< N; i++) {
            Mass(atoms[i], M);
            Velocity(atoms[i], Vector(0, 0));
            Internal(atoms[i], force);
        }
        t = BlockLoad(copper, atoms, R0, 25, 10, N);
        Velocity(atoms[60], Vector(0.4, -0.45));
        Color(atoms[60], Pink);

        // boundary condition; external force
        double x0 = high[0], y0 = high[1], x1 = low[0], y1 = low[1];
        double R = 0.6*R0;
        base = atoms;
        for (i= 0; i< t; i++) {
            r0[i] = Location(atoms[i]);
            if (r0[i][0] < x0) x0 = r0[i][0];
            if (r0[i][1] < y0) y0 = r0[i][1];
            if (r0[i][0] > x1) x1 = r0[i][0];
            if (r0[i][1] > y1) y1 = r0[i][1];
        }
        for (i= 0; i< t; i++) {
            if (r0[i][0]-x0 < R || r0[i][1]-y0 < R
                || x1-r0[i][0] < R || y1-r0[i][1] < R)
                External(atoms[i], external);
        }
    }

```

```

        InstallDraw(window, "Phase", phase);

        Run(window, field, t);
    }

void phase(Swindow& window, Scheme& scheme)
    // display in the phase space
{
    double x, y;
    int k, index;

    for (k= 0; k< Num(scheme); k++) {
        x = Location(scheme[k])[0];
        y = Velocity(scheme[k])[0];
        index = Color(scheme[k]);
        DrawPoint(window, Vector(x, y),
            Diameter(scheme[k]), index);
    }
}

```

A.5 Vortex Simulation of Fluid Flow

```

#include <Boltzmann/Vortex.h>
#include <Boltzmann/PPscheme.h>
#include <Boltzmann/RanVar.h>
#include <Boltzmann/Swindow.h>
#include <Boltzmann/Macro.h>

const double L      = 4;
const int M        = 6;
const int N        = 20;
const int S        = 2;
const int R        = 1;

class DeadVortex: DEACTIVATE(Vortex);

main()
{
    int i, j;
    double x, y, z;
    double h = float(L)/N;
    PPscheme field(Vector(-L, -L), Vector(L, L));
}

```

```

main()
{
    PMScheme field(Vector(0), Vector(L), Ng);
    Coulomb es[2*Np];
    Uniform rand(0, L);
    Maxwellian max1(T, Me), max2(T, M);
    Ripple ripp(0, L, 6*PI/L, 0, 0.2, 1);
    double vT = sqrt(KB*T/Me);
    double wp = sqrt(4*PI*Np/(L*Me))*Ns*eCHARGE, t = 0.25/wp;

    // electron stream 1
    for (int i= 0; i< Np/2; i++) {
        Location(es[i], Vector(ripp()));
        Velocity(es[i], Vector(5*vT+max1()));
        Charge(es[i], -1*Ns*eCHARGE);
        Mass(es[i], Ns*Me);
        Diameter(es[i], 1e4);
        Color(es[i], Yellow);
    }
    // electron stream 2
    for (i= int(Np/2); i< Np; i++) {
        Location(es[i], Vector(rand()));
        Velocity(es[i], Vector(-5*vT+max1()));
        Charge(es[i], -1*Ns*eCHARGE);
        Mass(es[i], Ns*Me);
        Diameter(es[i], 1e4);
        Color(es[i], Green);
    }
    // positively charged ions
    for (i= Np; i< 2*Np; i++) {
        Location(es[i], Vector(rand()));
        Velocity(es[i], Vector(max2()));
        Charge(es[i], Ns*eCHARGE);
        Mass(es[i], Ns*M);
        Diameter(es[i], 1e4);
        Color(es[i], Pink);
    }
    Load(field, es, 2*Np);

    // customized window
    Swindow window(Vector(0, 1e6), Vector(-1e8, 1e8));
    void phase(Swindow&, Scheme&);
}

```


APPENDIX B

Boltzmann 1.0: A User's Manual

B.1 Introduction

Boltzmann is an object-oriented particle simulation (OOPS) programming system. It can be used to simulate many-body problems and distributed-parameter systems that can be reduced to many-body problems. It consists of a library of *particle* classes, a library of *scheme* classes, high-level initialization, control, and visualization functions, and visualization windows. Particle classes are building block templates for simulations using particle methods. Schemes are “solution engines” that are able to carry out simulations when they are given a number of particles. High-level functions are “tools” to prepare simulations, and visualization windows facilitate friendly user-computer interactions. Boltzmann is implemented C++, an object-oriented extension to the C programming language. Although knowledge of C++ is helpful, a user is not required to know C++ in order to use Boltzmann. He should know how to program in C. Boltzmann has a C-like syntax, plus some pre-defined data types and functions. Boltzmann 1.0 can run on any computer that supports C++ Release 2.0 and X11 Release 3 or later releases.

B.2 The OOPS Framework

The relationship between particles and schemes can be illustrated by a programming framework (Figure B.1). In this framework, the structures of particles decide the structures of schemes. In turn, schemes supervise particles in simulations. Particle are responsible for local data and computations, while schemes are responsible for global data and computations. The same scheme can often supervise different classes of particles and the same particle can often work under different schemes.

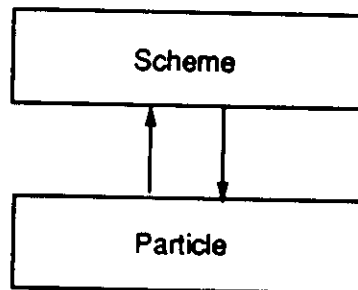


Figure B.1: A programming framework

```

// test particles
Euler test[M*N];
for (i= 0; i< M; i++) {
    y = -0.5*L+i*float(L)/(M-1);
    for (j= 0; j< N; j++) {
        x = -2.8*L+j*float(2*L)/N;
        Location(test[i*N+j], Vector(x, y));
        Diameter(test[i*N+j], h/10);
        Color(test[i*N+j], Green);
    }
}

// vortices to induce the flow
DeadVortex vortices[S];
Location(vortices[0], Vector(0, -10*L));
Vorticity(vortices[0], Vector(-50.0));
Location(vortices[1], Vector(0, 10*L));
Vorticity(vortices[1], Vector(50.0));

// image vortices for the cylinder
DeadVortex images[2*S];
Vector r, r0 = Vector(0, 0);
for (i= 0; i< S; i++) {
    r = Location(vortices[i]);
    z = magnitude(r);
    Location(images[i], R/(z*z)*r+r0);
    Vorticity(images[i], -Vorticity(vortices[i]));
    Size(images[i], h);

    Location(images[S+i], r0);
    Vorticity(images[S+i], Vorticity(vortices[i]));
    Size(images[i], h);
}
Diameter(images[S], 2*R);

Load(field, test, M*N);
Load(field, vortices, S);
Load(field, images, 2*S);
RunSwindow(field, 0.2);
}

```

Vector *vector_name*(*x*, *y*, *z*);

which create vectors of one, two, and three dimensions. Vectors that are defined without coordinates, such as

Vector *vector_name*;

Vector *vector_name*[*N*];

are treated as zero vectors of either one, two, or three dimensions before they are assigned to other vectors. Vector constants can be specified as

Vector(*x*);

Vector(*x*, *y*);

Vector(*x*, *y*, *z*);

Standard operations on vectors are supported. Operators in the C language are over-loaded with vector operations so that vector arithmetic can be expressed in the same way as float-point arithmetic. Vector operations in the Boltzmann programming system include

- **int** *dimension*(**Vector**&), which returns the vector dimensions;
- **double** *magnitude*(**Vector**&), which returns the magnitude of the vector;
- **Vector** + **Vector**, vector addition;
- **Vector** - **Vector**, vector subtraction;
- **Vector** * **Vector**, component-by-component vector multiplication;
- **Vector** / **Vector**, component-by-component vector division;
- **double** *dot*(**Vector**&, **Vector**&), the dot product of two vectors;
- **double** *cross*(**Vector**&, **Vector**&), the cross product of two vectors. If the two vectors are one-dimensional, the cross product is a zero vector because the two vectors are in the same or opposite directions. If the two vectors are three-dimensional, the cross product is also a three-dimensional vector. When the two vectors are two-dimensional, the cross product is a one-dimensional vector, taking the direction perpendicular to the plane formed by the two vectors;
- **Vector** + **double**, the addition of a number to every component of a vector. It is commutative;

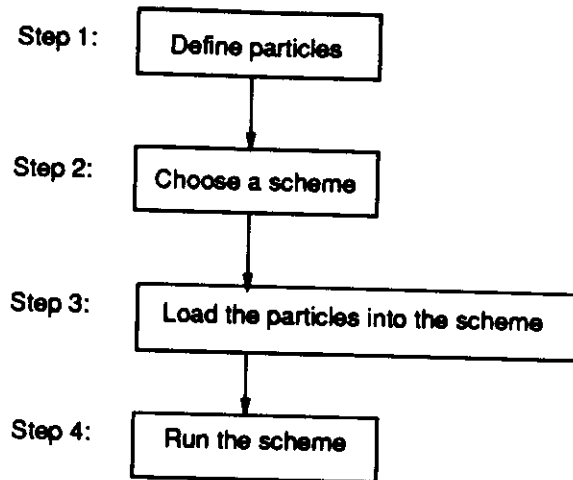


Figure B.2: Basic steps of OOPS programming

Particles and schemes are organized in hierarchies. A particle class or a scheme class may contain several subclasses. Each subclass differs from its parent class in certain functions. A subclass *derives* or *inherits* all its functions from its parent class, except wherever a new definition overrides the old one. All the high-level functions applicable to a class also apply to its subclasses.

Programming in Boltzmann is the composition of software objects. First, a class of particles are derived from a particle template in the particle library to describe entities in the simulated system. A scheme is then selected which is capable of supervising the particles. A simulation is conducted by loading the particles into the scheme and running the scheme in a visualization window (Figure B.2).

The following sections discuss data types and functions in Boltzmann 1.0. Vector arithmetic is widely used in Boltzmann, and therefore, a vector data type and vector operations are introduced in Section B.3, followed by random variables in Section B.4 and unit conversion in Section B.5. Section B.6 and B.7 discuss the particle classes and the scheme classes, respectively. In Section B.8, particle derivation is discussed. In Section B.9, initialization functions are discussed, followed by visualization windows and visualization functions in Section B.10 and B.11, respectively. Section B.12 gives a programming example in the Boltzmann programming system.

B.3 Vector Arithmetic

Vectors are specified in one of the following ways:

`Vector vector_name(x);`

`Vector vector_name(x, y);`

B.4.2 Normal Random Variables

Normal (Gaussian) random variables have the distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad (-\infty < x < +\infty) \quad (\text{B.1})$$

where μ is the mean value and σ the variance. A Normal random variable can be defined in the following ways:

`Normal variable_name(double μ , double σ);`

`Normal variable_name(double μ , double σ , RNG *gen);`

B.4.3 Maxwellian Random Variables

Maxwellian distribution

$$p(x) = \sqrt{\frac{m}{2\pi kT}} e^{-mx^2/2kT} \quad (-\infty < x < +\infty) \quad (\text{B.2})$$

is a special case of Normal distribution with $\mu = 0$ and $\sigma = \sqrt{kT/m}$. It is often used to set the initial velocities of particles in equilibrium. A Maxwellian random variable can be as following:

`Maxwellian variable_name(double T , double m);`

`Maxwellian variable_name(double T , double m , RNG *gen);`

where T is the temperature in Kelvin degree and m is the mass of the particles.

B.4.4 Exponential Random Variables

Exponential random variables have distribution

$$p(x) = \lambda e^{-\lambda x} \quad (0 \leq x < \infty) \quad (\text{B.3})$$

where λ is any positive constant. It occurs frequently as the distribution of waiting times between independent random events, such as the time a particle survives without a collision in an ideal gas. An Exponential random variable is defined in one of the two choices:

`Exponential variable_name(double λ);`

`Exponential variable_name(double λ , RNG *gen);`

- `Vector - double`, the subtraction of a number to every component of a vector;
- `Vector * double`, the multiplication of a number to every component of a vector. It is commutative;
- `Vector / double`, the division of a number into every component of a vector;
- `- Vector`, same as `Vector * (-1)`;
- `1 / Vector`, the inversion of every component of a vector;
- `double - Vector`, same as `(- Vector) + double`;
- `double / Vector`, same as `(1 / Vector) * double`;

As an example,

```
cross(Vector(1,2), Vector(3,4)) = Vector(-2)
```

B.4 Random Variables

Random variables are another class of mathematical entities that are implemented as objects. A random variable, when evaluated, will return a random number in accordance with its probability density function (distribution). In Boltzmann, the evaluation of a random variable x is denoted by $x()$. All the random variables are under the class `Random`, which is defined in the GNU C++ library. The following subsections discuss the Uniform, Normal, Maxwellian, and Exponential random variable classes.

B.4.1 Uniform Random Variables

A Uniform random variable can be specified in one of the two choices:

```
Uniform variable_name(double low, double high);
```

```
Uniform variable_name(double low, double high, RNG *gen);
```

where *low* and *high* are the lower and upper bounds of the random numbers generated by the variable, and *gen* is a “seed” generator for the random variable as discussed in the GNU C++ library manual. When the first specification is used, a default seed generator is employed which is often adequate.

then it is enough to redefine cgs in term of l_0 , m_0 , and t_0 :

```
#define centimeter 1/1.804*1e8
#define gram 1/5.275*1e26
#define second 1/3.273*1e15
```

In that case, one electronvolt would be

$$\begin{aligned} 1 \text{ eVs} &= 1 \times \text{electronvolt} \\ &= 1 \times 1.60219 \times 10^{-12} \times \text{erg} \\ &= 1 \times 1.60219 \times 10^{-12} \times \text{dyne} \times \text{centimeter} \\ &= 1 \times 1.60219 \times 10^{-12} \times \text{gram} \times \text{centimeter} \\ &\quad / \text{second}^2 \times \text{centimeter} \\ &= 1 \times 1.60219 \times 10^{-12} \times 1/5.275 \times 10^{26} \\ &\quad \times 10^{26} \times 1/1.804 \times 10^8 / (1/3.273 \times 10^{15})^2 \\ &\quad \times 1/1.804 \times 10^8 \\ &\approx 1.0 [m_0(l_0/t_0)^2] \end{aligned}$$

B.6 A Particle Hierarchy

Particles are the most important objects in the Boltzmann programming system. The characteristics of particles, plus their initial conditions and the boundary conditions, completely determine the characteristics of a simulated system. A number of particle classes are developed to provide various particle templates at different levels of abstraction. They are organized in a hierarchy to facilitate software reusability. Each particle class has a name, a set of attributes, and a set of functions. The class name distinguishes the class from other classes. The attributes contain parameters relevant to the class of particles and their state variables. The functions support the functionalities of the particle class. Figure B.3 is a hierarchy of some particle classes. Each class is discussed below.

B.6.1 The Particle Class

The Particle class is the topmost class in the hierarchy. It provides a consistent framework for all particle classes. The Particle class has at least the following four member functions:

- `void internal(Particle& p)`, which calculates the interaction between this particle and particle `p`.
- `void external()`, which calculates the external influence on the particle.
- `int integrate(double t, int step)`, which does one step of integration and returns the number of the next step. It returns zero when all steps are completed.

B.5 Unit Conversion

Scientific computing can be performed in many different unit systems. The most common ones include the meter/kilogram/second (*mks*) unit system and the centimeter/gram/second (*cgs*) unit system, but other units are also employed frequently. The selection of appropriate unit system may have a direct impact on the ease of programming and the accuracy and efficiency of the simulation. Sometimes, it is desirable to express a problem in one unit system and to compute in another. To accommodate the variety of different unit systems, a set of macro definitions is provided to convert various unit system to a default one. Moreover, the default unit can be changed by the users.

The set of macro definitions is contained in the `Unit.h` file. The default unit is chosen to be *cgs*, for *mks* is often found too large for particle simulation. Other units are expressed in term of *cgs* unit, such as

```
#define centimeter 1
#define centimeters *centimeter
#define gram 1
#define second 1
#define esu 1
#define dyne gram*centimeter/sqr(second)
#define dynes *dyne
#define erg dyne*centimeter
#define meter 100*centimeter
#define kilogram 1000*gram
#define coulomb 3e9*esu
#define newton 1e5*dyne
#define joule 1e7*erg
#define angstrom 1e-8*centimeter
#define electronvolt 1.60219e-12*erg
#define electronvolts *electronvolt
#define eV electronvolt

#define grams *gram
#define seconds *second
#define esus *esu
#define ergs *erg
#define meters *meter
#define kilograms *kilogram
#define coulombs *coulomb
#define newtons *newton
#define joules *joule
#define angstroms *angstrom
#define eVs electronvolts
```

To change the default unit system, it is sufficient to express the *cgs* units in term of the new default unit system. For example, if the unit length is defined to be

$$l_0 = 1.804 \times 10^{-8} \text{ cm} ,$$

the unit mass

$$m_0 = 5.275 \times 10^{-26} \text{ g} ,$$

and the unit time

$$t_0 = 3.275 \times 10^{-15} \text{ sec} ,$$

Moreover, the same set of functions can be used to assign or change an attribute of a particle by providing two parameters, instead of one, where the first parameter is the particle and the second one is the new attribute. For example,

```
Location(a, Vector(x, y));
```

places particle *a* at the position (x, y) . In addition, the internal and external influence functions can be changed by the following two OOPS functions:

- `void Internal(Particle&, IntF influence)`, which changes the internal influence function,
- and `void External(Particle&, ExtF influence)`, which changes the external influence function,

where `IntF` and `ExtF` are defined as

- `typedef Vector (*IntF)(Particle&, Particle&);`
- `typedef Vector (*ExtF)(Particle&);`

B.6.2 Classical Particle Class (Newton)

Classical particles are described by two state variables: the position vector \mathbf{r} and the velocity vector \mathbf{v} where $\mathbf{v} = d\mathbf{r}/dt$. An equation of motion, which is Newton's law

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{F}/m \quad (\text{B.4})$$

is associated with every particle. Classical Particle class is also called Newton Particle class. It is the choice of the force function \mathbf{F} that provides the rich variety of particle simulations that can be done with the two state variables \mathbf{r} and \mathbf{v} and Newton's law (B.4). The position \mathbf{r} and the velocity \mathbf{v} are vectors of either one, two, or three dimensions that are specified in the definition of the particle. In addition, a classical particle contains attributes concerning its correspondence with the outside world, such as its color and size when it is displayed. Not every attribute of a particle has to be specified each time the particle is defined. For some attributes, default values are employed when they are not specified.

Classical particles can be defined in the following way, in addition to the ways general particles are defined:

```
Newton particle_name(mass, position, velocity, internal_influence);
```

which specifies a Classical particle with its mass, position, velocity, and inter-particle interaction. The following OOPS functions return attributes about a Classical particle, in addition to the OOPS functions that can be applied to general particles:

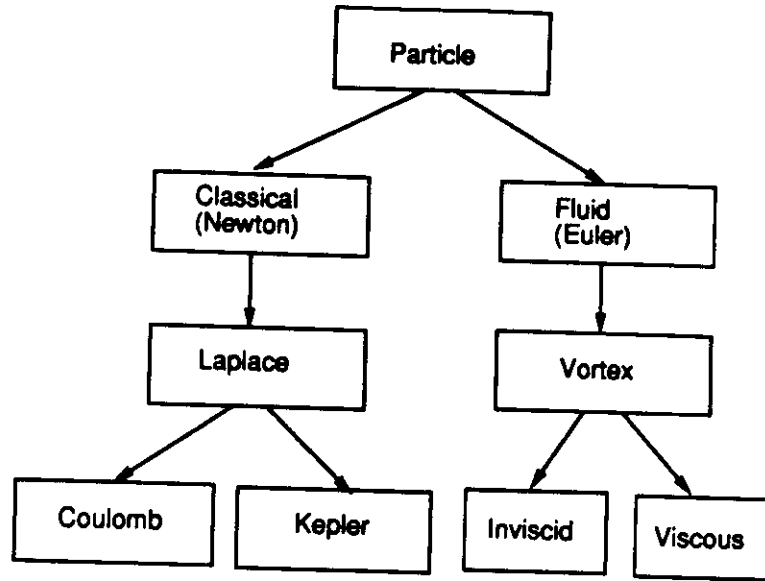


Figure B.3: A particle hierarchy

- `void update(double t)`, which does the rest updates before the particle is advanced into the next time step.

The Particle class are defined in one of the following ways:

- `Particle particle_name;`
- `Particle particle_array_name[size];`
- `Particle(internal_influence) particle_name;`
- `Particle(internal_influence) particle_array_name[size];`

The former two define a particle and an array of particles, respectively. The later two define the particle(s) with an internal influence function that specifies the interaction between any two such particles. When the influence function is not specified, a default zero influence function is used.

The following OOPS functions return attributes about a particle:

- `Vector& Location(Particle&)`, which returns the position vector of the particle,
- `int Color(Particle&)`, which returns the color code of the particle at display,
- and `float Diameter(Particle&)` returns the diameter of the particle at display,

A OOPS function that is defined with Laplace Particle class but not Classical Particle class is

```
double Strength(Laplace&);
```

which returns the strength of the particle. Laplace Particle class is a subclass of Classical Particle class.

B.6.4 Coulomb Particle Class

Coulomb particles interact with one another in electrostatic forces. Only one function has been added to the class in accordance with tradition:

```
double Charge(Coulomb&);
```

which returns the charge of the charged particle. Coulomb Particle class is a subclass of Laplace Particle class.

B.6.5 Kepler Particle Class

Kepler particles interact with one another in gravitational forces. The strength of a Kepler particle is the product of the square-root of the gravitational constant and the mass of the particle. Kepler Particle class is a subclass of Laplace Particle class.

B.6.6 Fluid Particle Class (Euler)

Fluid equations are often macroscopic field equations derived from averaging microscopic particle descriptions over a space that is large when compared with the interparticle separation and a time that is long when compared with the collision time of the microscopic particles. It is often possible to simulate fluids by particles of field nature that represent properties of the space. Unlike the classical particles, the particles are often driven by the velocity field instead of the force field. The class of particles that create velocity field is called Fluid Particle Class, or Euler Class. The description of fluid with the use of Euler particles is also called Eulerian description. Fluid Particle Class has one required state variable, the position r , among its attributes. The equations of motion contain, at least,

$$\frac{dr}{dt} = v \quad (\text{B.7})$$

where v is the velocity field at the position of the discussed fluid particle. By employing *a priori* fluid particles, many phenomena of fluid flows can be simulated.

Euler particles can be defined in the following way, in addition to the ways general particles are defined:

```
Euler particle_name(position, influence_function);
```

The same set of OOPS functions that can be applied to Particle class can also be applied to Fluid Particle class.

- **Vector& Velocity(Newton&)**, which returns the velocity vector of the particle,
- **double Mass(Newton&)**, which returns the mass,

Similarly, the same set of functions can be used to assign or change an attribute of a particle by providing two parameters, instead of one, where the first parameter is the particle and the second one is the new attribute.

The equations of motion for a classical particle are Newton's law (B.4), or

$$\begin{cases} \mathbf{r}^{n+1} = \mathbf{r}^n + \int_{t_n}^{t_{n+1}} \mathbf{v} dt \\ \mathbf{v}^{n+1} = \mathbf{v}^n + \int_{t_n}^{t_{n+1}} (\mathbf{F}/m) dt \end{cases} \quad (\text{B.5})$$

where \mathbf{F} is generally a function of the positions and velocities of this and other particles. There exist many numerical algorithms that approximate the solution of the equations of motion, and they can be found in the literature on initial-value problems.

A variety of "experiments" can be set up by the right combination of above functions. For instance, dissipation in wave propagation can be simulated by giving an external force proportional to the velocity of the particles; a fixed end of a metal rod can be simulated by a massive particle at the end or a particle with an empty integration step.

B.6.3 Laplace Particle Class

Forces that are derived from Green's function for Laplace operator have the form

$$\mathbf{E} = \frac{e\mathbf{r}}{|\mathbf{r}|^3} \quad (\text{B.6})$$

where e is a constant attribute of the particles. They are often seen in the real world, such as the electrostatic force and the gravitational force. In the case of electrostatic force, e is equal to the charge and, in the case of gravitational force, e is equal to the mass multiplied by \sqrt{G} (G is the gravitational constant). The constant e is called the *strength* of the particle.

The specification of the inter-particle influence function of a Laplace particle is reduced to the specification of the strength. Although a Laplace particle can be defined as a Classical particle, it is more advantageous to define the particle as a Laplace particle with the specification of the strength of the particle. Internally, the influence function can be recovered if the particle has been loaded into a scheme that does not recognize Laplace Particle class. Beside its simpler specification, the main advantage of Laplace Particle class over Classical Particle class is the ability to run with faster schemes that take the advantage of the specific influence function. Examples of the faster schemes include the particle-mesh scheme and the fast multipole scheme.

B.6.8 Inviscid Vortex Class

For inviscid vortices, the equations of motion are Equation (B.7) and

$$\frac{d\omega}{dt} = (\omega \cdot \nabla)\mathbf{v} \quad (\text{B.13})$$

Inviscid vortices can be defined as

```
Inviscid vortex(Vector& r, Vector& ω);
```

Inviscid Vortex class is a subclass of Vortex class.

B.6.9 The Viscous Vortex Class

Viscous vortices have one more attribute than the inviscid vortices, the viscosity ν . The equations of motion are Equation (B.7) and (B.12). The viscosity term in (B.12) can be accounted for by simulating the diffusion of vorticity by a random walk, which will be part of the equations of motion. Viscous vortices can be defined as

```
Viscous vortex(Vector& r, Vector& ω, double ν);
```

Additional OOPS functions that can be applied to viscous vortices include:

- `double Viscosity(Viscous&)`, which returns the viscosity of the vortex,
- and `void Viscosity(Viscous&, double ν)`, which assigns viscosity ν to the vortex.

B.7 A Scheme Hierarchy

Schemes are particle simulation methods represented as objects. They specify the computational algorithms to be used in a simulation with the use of particles. Their responsibilities include particle-particle interaction evaluation and boundary imposition, among other things. They are organized in a hierarchy to facilitate software reusability. A scheme is always more general than its subschemes, that is, if a scheme can be applied to a class of particle classes, then the parent scheme can also be applied to the same particle class. Figure B.4 is a hierarchy of scheme classes, each of which is discussed below.

B.7.1 The Scheme Class

The Scheme class serves as the basis for all schemes. A scheme always defines its dimension, a computation space in which particles interact with one another, a particle stack that is used to contain particles, and a list of boundaries and boundary conditions. A stack is different from an array, as new particles can be

B.6.7 Incompressible Vortex Class

As a subclass of Fluid Particle Class, Incompressible Vortex Class is a good example of how a fluid particle is constructed. Vortices have an attribute, the vorticity ω , where

$$\omega = \nabla \times \mathbf{v} \quad (\text{B.8})$$

From the incompressibility (the mass density ρ is constant), the velocity field is divergence-free, i.e.,

$$\nabla \cdot \mathbf{v} = 0 \quad (\text{B.9})$$

It is known that there exists a vector potential ψ such that

$$\mathbf{v} = \nabla \times \psi \quad (\text{B.10})$$

and

$$\nabla^2 \psi = -\omega \quad (\text{B.11})$$

By solving Equation (B.10) and (B.11), it is seen that the velocity field can be recovered from the vorticity of the vortices. That is to say that Green's function for (B.11), together with (B.10), defines the Incompressible Vortex class. From the velocity field at each vortex, the position of the vortex at the next time step can be calculated. Unlike the charge attribute in Coulomb particles, however, the vorticity of vortices does not remain unchanged in general. It is the governing equation of the fluid that determines the new value of the vorticity of a vortex at the next time step. Very often, the governing equation for incompressible flow is the Navier-Stokes equation expressed in vorticity form:

$$\frac{d\omega}{dt} = (\omega \cdot \nabla)\mathbf{v} + \nu \nabla^2 \omega \quad (\text{B.12})$$

where

$$\frac{d\omega}{dt} = \frac{\partial \omega}{\partial t} + (\mathbf{v} \cdot \nabla)\omega$$

is the rate of vorticity change along the flow lines, and ν is the kinematic viscosity of the fluid. According to the value of the viscosity ν , vortices are further classified into inviscid vortices, where $\nu = 0$, and viscous vortices.

Additional OOPS functions that can be applied to vortices include:

- **Vector& Vorticity(Vortex&)**, which returns the vorticity of the vortex,
- and **void Vorticity(Vortex&, Vector& ω)**, which assigns vorticity ω to the vortex.

A vortex can be defined as

Vortex name(Vector& r , Vector& ω);

- `void Calculation()` calculates the influence on every particle stored in the scheme.
- `void Advance(double t)` advances all the particles one time step of size `t`.

B.7.2 The Particle-Particle (PP) Scheme

The PP Scheme is a subclass of the Scheme class. Whereas the Scheme class does not specify how the influence on every particle is calculated, the Particle-Particle Scheme calculates the influence by summing up all influence contributions from other particles. It is, in general, a $O(N^2)$ scheme in computation time, where N is the number of particles. For certain types of influences, however, the Particle-Particle Scheme can facilitate special arrangements that reduce the computation time, usually to the order of N . Two special arrangements are for the calculation of short-range influences and of influences existing only between fixed pairs of particles.

For short-range interactions, the PP Scheme allows the specification of a cut-off distance, beyond which the influence of a particle is assumed to be negligible. Moreover, certain applications require influence calculation only between fixed pairs of particles. It is then necessary to specify where influences exist so that the scheme does not create extra influences and consumes less computation time. The Connection Class is defined to designate a partial specification of the particle pairs where influences exist. An instance of the Connection Class, or *connection*, is denoted by

$$(i : j : k)$$

which represents the set of particle pairs

$$\{(i, i+k), (i+k, i+2k), \dots, \\ (i + (\lfloor \frac{j-i}{k} \rfloor - 1)k, i + \lfloor \frac{j-i}{k} \rfloor k)\}$$

For example,

$$(0 : 5 : 2) = \{(0, 2), (2, 4)\}$$

where there exist influences between particle 0 and particle 2 and between particle 2 and particle 4. A union of connections gives a full specification of all particle pairs where influences exist.

A PP scheme is specified in the following way:

```
PPscheme scheme_name(Vector& low, Vector& high, BC bc,
double d);
```

where d gives the cut-off distance of the short-range influence. The default cut-off distance, when d is not given, is infinity. To add a connection to a PP scheme, the function `AddConnection` is used:

```
void AddConnection(PPscheme&, int i, int j, int k);
```

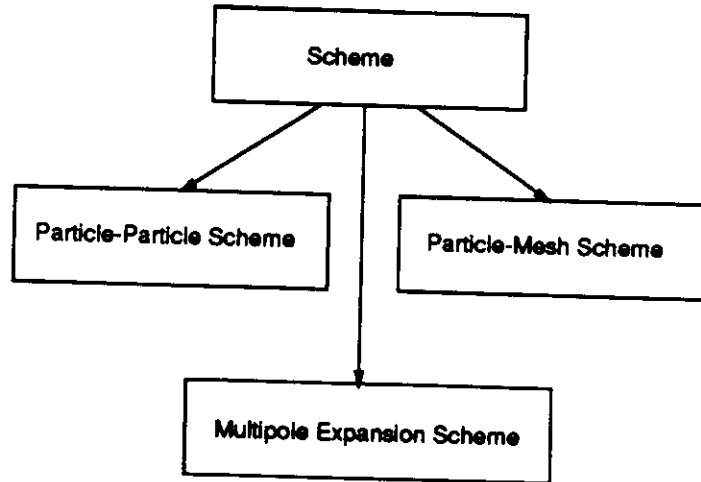


Figure B.4: A scheme hierarchy

added and are added to the top of the stack. Boundaries surround special regions of the computation space such that when particles are entering or leaving the region, boundary conditions are enforced.

A scheme in the Scheme class is specified in the following way:

```
Scheme scheme_name(Vector& low, Vector& high, BC bc);
```

where *low* and *high* are the lower bound and the upper bound of the computation space, respectively, and *bc* is the boundary condition for the space, which can be one of the enumeration constants `Open`, `Bounded`, `Periodic`, `Dirichlet`, and `Neumann`. The default boundary condition is `Bounded` when *bc* is not given. For a one-dimensional scheme, the lower and upper bounds are vectors of one dimension. For a two-dimensional scheme, they are vectors of two dimensions. The following functions return information about a scheme:

- `int Num(Scheme)` returns the number of particles loaded in the scheme,
- `int Dim(Scheme)` returns the dimensionality of the scheme,
- `Vector& UpperBound(Scheme)` returns the upper bound of the space of the scheme,
- `Vector& LowerBound(Scheme)` returns the lower bound of the space of the scheme,
- and `Particle& Scheme[int i]` returns the *i*th particle in the particle stack of the scheme.

The following member functions define the behavior of the Scheme Class:

existing function, among others. Particle derivation can be considered as a collection of second-order functions that take particle classes as their domain. C++, the implementing language underlying the Boltzmann programming system, provides static class derivation/inheritance that can be used to support particle derivation. The syntax for class derivation in C++, however, requires some understanding of the C++ language, which should be minimized according to one of the principles in the design of Boltzmann. To this end, a number of macro definitions are provided to hide certain syntactic details that are not important to the users. At times, it is desired that C++ language provide dynamic class derivation which would allow more freedom in particle derivation.

The following macro functions return a subclass of a particle class:

- `INHERIT(parent_class, new_definitions)`, which returns a subclass of the parent class with the addition of some new definitions. The new definitions can be any variable definitions or function definitions in the C language. Note: each definition in *new_definitions* must end with `;`.
- `EULER(parent_particle)`, which returns a subclass with Euler's method as its integration function.
- `LF(parent_particle)`, the leap-frog method as the intergation function.
- `RK2(parent_particle)`: the 2nd-order Runge-Kutta method as the integration function.
- `RK4(parent_particle)`: the 4th-order Runge-Kutta method as the integration function.
- `DEACTIVATE(parent_particle)`: empty integration function.

A subclass is defined in the following way:

```
class name: macro-functions;
```

For example, a subclass of Newton Class with the integration function replaced by the 4th-order Runge-Kutta method is defined as

```
class name: INHERIT(Newton,
    Vector r0, v0, f0, v1;
    int integrate(double t, int step) {
        switch (step) {
            case 1: r0 = r; v0 = v; f0 = f;
                r += 0.5*t*v; v += 0.5*t/m*f;
                return 1;
            case 2: v1 = v; f0 += 2*f;
                r = r0+0.5*t*v; v = v0+0.5*t/m*f;
```

B.7.3 The Particle-Mesh (PM) Scheme

The PM Scheme represents a computational method that evaluates the interactions among particles by solving a field equation for the potential on a mesh over the space and interpolating the influence on a particle from the potentials at the surrounding grid points. This is a scheme with a computational complexity of $O(N + M \log M)$, where M is the number of grid points on the mesh.

A PM scheme is specified in one of the following ways:

```
PMscheme scheme_name(Vector& low, Vector& it high,  
int n1, BC bc);
```

```
PMscheme scheme_name(Vector& low, Vector& it high,  
int n1, int n2, BC bc);
```

```
PMscheme scheme_name(Vector& low, Vector& it high,  
int n1, int n2, int n3, BC bc);
```

Where n_1 , n_2 , and n_3 are the number of grid points along the first, the second, and the third dimension, respectively (PMscheme in three dimensions is not available in Boltzmann 1.0). The default boundary condition is `Periodic` when `bc` is not specified. The member functions of the PM Scheme class include

- `void DensityAssignment()`, which assigns strength density to the grid points.
- `void PotentialCalculation()`, which solves the Poisson's equation,
- `void InfluenceField()`, which calculates the influence field on the mesh,
- `void Interpolation(Laplace&)`, which interpolates the influence field at the particle,
- and `void Calculation()`, which involves the above functions one after another to calculate the influence on a particle at each time step.

Each function can be replaced by a new definition without the change of other functions.

B.8 Particle Derivation

Although a great effort has been made to provide particle classes for a variety of applications, it is impossible and unwise to prepare a particle class for every kind of particles that may be used in particle simulations. Particle derivation is an important mechanism to make the particle hierarchy extensible. Specialization of high-level abstract particles is done through particle derivation. It contains facilities to add a variable, to add a new function, or to change the definition of an

of the velocities. The function to generate Maxwellian distribution to a set of particles is

```
void MaxwellianDistribution(Particle* particles,
double m, double T, int N);
```

B.9.3 Oscillating Particles

For small oscillations about some equilibrium positions, the probability density of particles having displacement $\mathbf{r} = (x, y, z)$ from the equilibrium positions is

$$\left(\frac{\omega^2 m}{2\pi kT}\right)^{3/2} e^{-\omega^2 m(v_x^2 + v_y^2 + v_z^2)/2kT} \quad (\text{B.15})$$

where ω is the oscillation frequency. It is useful in setting the initial positions of particles oscillating about some equilibrium positions such as crystal lattices. The function that creates small displacements to particles from their equilibrium positions is

```
void Oscillating(Particle* particles,
double m, double T, double omega, int N);
```

B.9.4 Particles Under the Influence of Gravity

The distribution of particles under the influence of gravity satisfies Boltzmann's formula

$$n(h) = n(0)e^{-mgh/kT} \quad (\text{B.16})$$

where g is the magnitude of the gravitational acceleration, h the height of space from any reference point in the opposite direction of the gravitational acceleration, and $n(h)$ the number density of particles at height h . If what is wanted is to assign positions to some fixed number of particles at the height from a to b according to Boltzmann's distribution, it can be done by generating a random number y with an exponential distribution at the range from 0 to ∞ , discarding y when $y > b - a$, and letting $h = y + a$. The following function has the last space dimension distributed according to Boltzmann's formula and the rest of space dimensions distributed uniformly:

```
void Gravity(Particle* particles,
double m, double T, Vector& low, Vector& high, int N);
```

B.10 Visualization Windows

Simulation results are displayed in visualization windows. A visualization window can be as simple as a "drawing board" on which drawing functions can write points and lines, or it can provide functions such as diagnosis selection, tracking,

```

        return 1;
    case 3: v1 += v; f0 += 2*f;
           r = r0+t*v; v = v0+t/m*f;
           return 1;
    case 4: r = r0+t/6*(v0+2*v1+v);
           v = v0+t/(6*m)*(f0+f);
           return 0;
    default:
           return 0;
    }
);

```

or simply

```
class name: RK4(Newton);
```

B.9 Initialization Functions

A user can initialize a simulation in a number of ways. The most primitive approach is to assign values to the attributes of particles one at a time, usually in a loop statement. The advantage of that approach is its flexibility in setting up all kinds of initial conditions, but the disadvantage is that the calculation of the attribute values can be very involved. Another approach builds, on top of the primitive functions, initialization functions that automatically assign values to particles according to macroscopic properties of them, such as the temperature and the energy. Boltzmann 1.0 supports the following initialization functions:

B.9.1 Random Load

This function loads particles in a scheme at locations decided by a random variable:

```
RandomLoad(Scheme#, Particle* particles, Random#, int N);
```

B.9.2 Hot Particles

According to Maxwellian distribution, the probability density of particles at thermal equilibrium having velocity $\mathbf{v} = (v_x, v_y, v_z)$ is

$$\left(\frac{m}{2\pi kT}\right)^{3/2} e^{-m(v_x^2+v_y^2+v_z^2)/2kT} \quad (\text{B.14})$$

where T is the temperature and k the Boltzmann constant. It can be used to assign velocities to a set of particles at equilibrium on the basis of the temperature, with the use of a Maxwellian random variable to generate values for each component

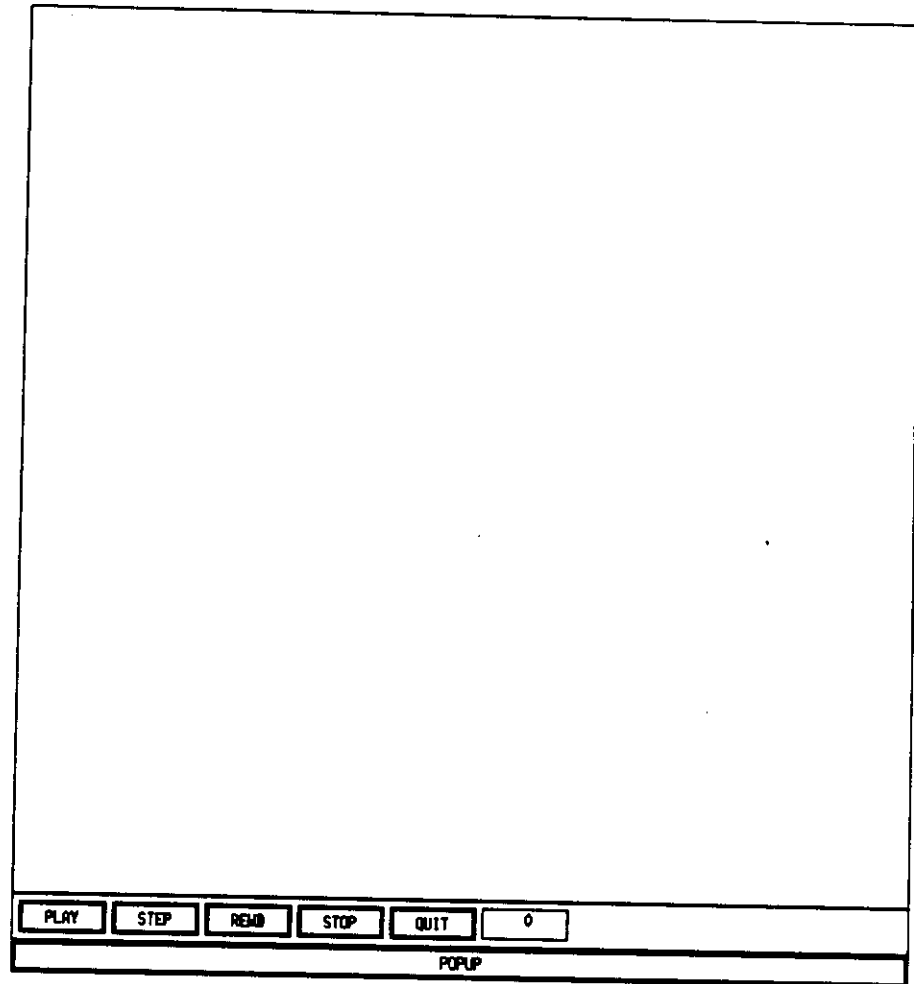


Figure B.5: A visualization window

replaying, and scrolling. In Boltzmann, visualization windows are objects, allowing an unlimited number of windows to be supported. An example of visualization windows, called `Swindow`, is supported in Boltzmann 1.0 and is described below.

A `Swindow` is defined by declaring

```
Swindow window_name(Vector x-dimension, Vector y-dimension);
```

where *x-dimension* and *y-dimension* are vectors of two elements giving the lower bound and the upper bound of the window in the horizontal and the vertical directions, respectively. The lower and upper bounds are given the unit of space in the simulation, not the unit of the size of the physical screen. This makes it easy to determine what the lower and upper bounds should be in order to visualize a specific region of a simulation.

When activated, a `Swindow` looks like the one in Figure B.5, where the big blank space is the drawing board and the buttons are for the functions that come with the window. The `POPUP` button is for opening a menu window that allows the selection of diagnosis functions and the modification of system parameters such as the step size. Figure B.6 displays a menu window. Clicking the `DIAGNOSIS` button in the menu window opens a menu of diagnosis functions. A sample list contains items such as Cartesian Space, Phase Space, and Power Spectrum. User-defined diagnosis functions can be installed into the list by calling the function

```
InstallDraw(Swindow window, char* label, DrawFun func);
```

where `DrawFun` is defined as

```
typedef void (*DrawFun)(Swindow, Scheme);
```

and *func* is the user-provided diagnosis function. Clicking the `TRACK` button turns the tracking function of the window on or off, which determines if the results of earlier time steps should be kept unerased. The `DONE` button closes the menu window.

Returning to the `Swindow`, the `PLAY` button initiates the simulation and the display of the results according to the diagnosis function chosen in the menu window. The right-most small window in the same row displays the time step at which the results are being displayed. The `STEP` button advances the simulation by one time step. The `REWD` button rewinds the simulation, by running the same simulation with a negative time step. According to classical mechanics, a Newtonian many-body problem is time-reversible, provided the computation error is negligible. Therefore, the `REWD` button can be used as an error checker. If the simulation can not go back to the original state, significant computation errors have been introduced and caution is required. The `STOP` button stops the simulation and the `QUIT` button terminates it.

A number of high-level drawing functions can write on a `Swindow` that do not require an understanding of the underlying graphics primitives. The first drawing function is

```
void DrawParticle(Swindow window, Particle particle)
```

which draws a circle on the drawing board of *window*. The size and color of the circle depends on the attributes of the particle *Diameter(particle)* and *Color(particle)*. The size of the circle on the drawing board is in proportion to the ratio of *Diameter(particle)* to the size of the window. When *Diameter(particle)* is positive, the circle is drew in the color designated by *Color(particle)*. When *Diameter(particle)* is negative, the circle is filled with the same color. The second drawing function is

```
void DrawCircle(Swindow window, Vector origin, double diameter,  
double color)
```

This is equivalent to *DrawParticle* with *origin* replacing *Location(particle)*, *diameter* replacing *Diameter(particle)*, and *color* replacing *Color(particle)*. It is used to draw a circle, usually to represent a particle, when *DrawParticle* is inadequate. *DrawCircle* is more flexible than *DrawParticle* because *origin*, *diameter*, and *color* can be chosen arbitrarily. The third drawing function discussed here is

```
void DrawLine(Swindow window, Vector r1, Vector r2,  
double color)
```

which draws a line from the point at r_1 to the point at r_2 on the drawing board in the color designated by *color*.

B.11 Visualization Functions

Visualization functions, also called diagnosis functions, display simulation results in various forms to facilitate easy comprehension of them. Boltzmann 1.0 supports the following visualization functions, which are provided as buttons on *Swindows*.

B.11.1 Cartesian Space Visualization

This function displays, at every time step, particles at their relative locations in real space in Cartesian coordinate system. The shape and color of the particles are determined by their corresponding attributes in the particles. Boltzmann 1.0 displays only the first two dimensions if a simulation is in three dimensions. Later version will support 3D visualization.

B.11.2 Phase Space Visualization

This function displays, at every time step, particles at their relative locations in phase space in Cartesian coordinate system. The phase space is defined to be the space expanded by the real space and the velocity space. Boltzmann 1.0 displays

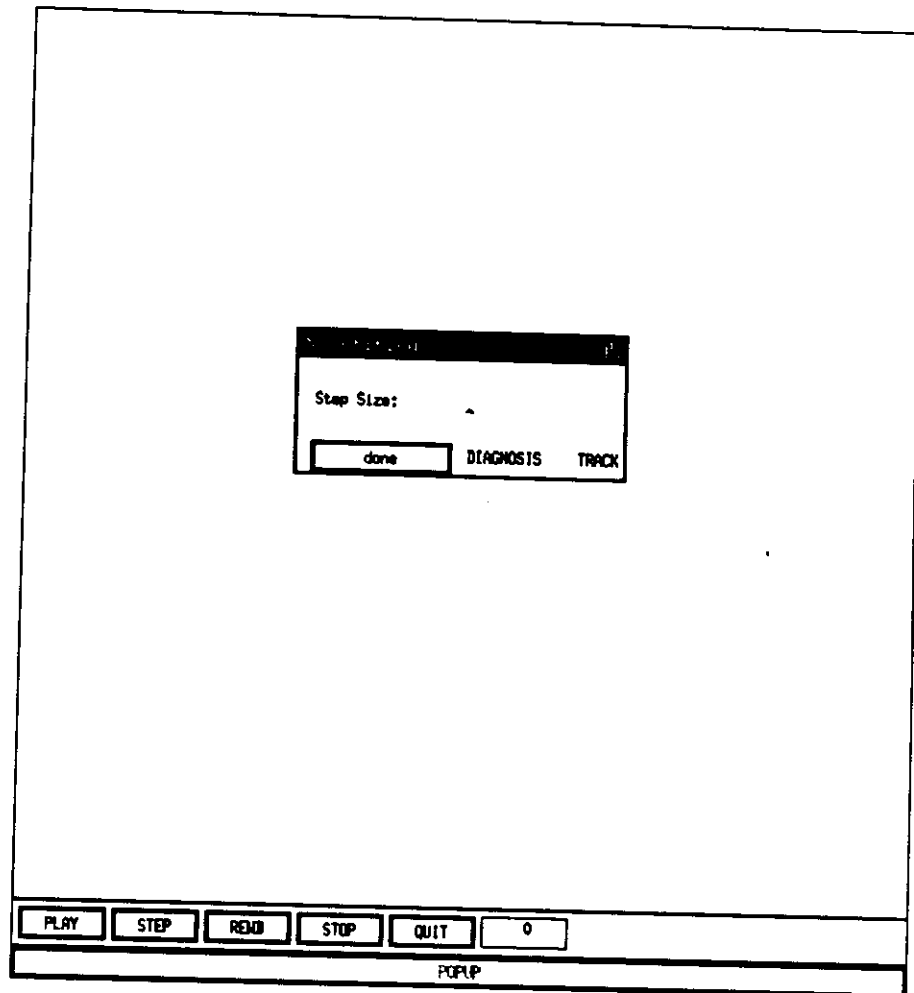


Figure B.6: A menu window


```

#include <Boltzmann/Unit.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/PPscheme.h>

```

```

const R0 = 2.551 angstroms;

```

```

Vector low(0, 0), high(100 angstroms, 100 angstroms);
PPscheme copper(low, high, 3*R0);

```

where `Unit.h`, `Newton.h`, and `PPscheme.h` are header files for Boltzmann definition of the units, Newton class, and the Particle-Particle Scheme class. A header file that includes all other header files is `Boltzmann.h`.

The atoms are defined to be Newton particles with Lennard-Jones force,

```

Vector force(Newton&, Newton&);
Particle atoms[N];
for (i= 0; i< N; i++)
    Internal(atoms[i], force);

```

where `N` is the number of atoms used in the simulation and `force` is defined by

```

const DEPTH = 0.1 electronvolts;

Vector force(Newton& p, Newton& q)
{
    Vector u = Location(q) - Location(p);
    double r = magnitude(u);
    double a = R0/r;
    a = a*a*a;
    a ** a;
    u **= 12*DEPTH*a*(a-1)/(r*r);
    return u;
}

```

Since the simulated atoms represent only a tiny fraction of the metal, a boundary condition has to be imposed. It can be incorporated in the simulation by specifying certain external force on the boundary particles. A simple approximation of the boundary condition is given by an external spring force, defined as

```

Vector external(Newton& p)
{
    Vector f = Location(p) - original[&p-atoms];
    f **= -kappa;
    return f;
}

```

only the first dimension of the phase space (one real dimension and one velocity dimension) if a simulation is in more than one dimension. It supports re-scaling in the velocity dimension.

B.11.3 Power Spectrum Analysis

A mass of particles can support waves. The wave properties of a system of particles can be studied by power spectrum analysis. For a function $h(x)$, the Fourier transform of that function is defined to be

$$H(f) = \int_{-\infty}^{\infty} h(x)e^{2\pi ifx} dx$$

If $h(x)$ is a function of time, the Fourier transform is a function of frequency. If $h(x)$ is a function of space, then the Fourier transform is a function of wavenumber. The *power spectrum density* (PSD) of function $h(t)$ is defined to be

$$PSD_h(f) = |H(f)|^2 + |H(-f)|^2, \quad 0 \leq f < \infty$$

which indicates the power of the signal in the frequency or wavenumber spectrum.

In Boltzmann 1.0, the signal to be power spectrum analyzed is fixed to be the velocity of the particles along the first dimension. The particles must also be uniformly distributed in one dimension in the space for the estimation to be meaningful. The power spectrum estimation uses two segments of data from two consecutive time steps and Parzen data windowing to get a smooth estimation. Later version of Boltzmann will support the power spectrum analysis of arbitrary signals.

B.12 A Programming Example

As a programming example, the simulation of the displacement of atoms in metals under radiation is discussed. A metal is modeled by a lattice of atoms and the interactions among the atoms. In our example, the interaction between two atoms is represented by the potential function of the separation of the atoms:

$$\phi(r) = \epsilon \left[\left(\frac{r_0}{r} \right)^{12} - 2 \left(\frac{r_0}{r} \right)^6 \right], \quad (\text{B.17})$$

which is called Lennard-Jones potential; r_0 is the atomic separation at equilibrium and ϵ is the depth of the potential well.

The development of the simulation program starts with the selection of the particle class and the scheme class. Since the force between two atoms is short-ranged and decreases rapidly as the separation becomes large, the `PPscheme` class is selected because it allows the specification of a cut-off distance. Here, a piece of copper is defined in two dimensions and the cut-off distance of inter-particle force is set to be $3r_0$, as in

```

        Velocity(atoms[i], Vector(0, 0));
        Internal(atoms[i], force);
    }
    t = BlockLoad(copper, atoms, R0, 25, 10, N);
    Velocity(atoms[60], Vector(0.4, -0.45));
    Color(atoms[60], Pink);

    // boundary condition; external force
    double x0 = high[0], y0 = high[1], x1 = low[0], y1 = low[1];
    double R = 0.6*R0;
    base = atoms;
    for (i= 0; i< t; i++) {
        r0[i] = Location(atoms[i]);
        if (r0[i][0] < x0) x0 = r0[i][0];
        if (r0[i][1] < y0) y0 = r0[i][1];
        if (r0[i][0] > x1) x1 = r0[i][0];
        if (r0[i][1] > y1) y1 = r0[i][1];
    }
    for (i= 0; i< t; i++) {
        if (r0[i][0]-x0 < R || r0[i][1]-y0 < R
            || x1-r0[i][0] < R || y1-r0[i][1] < R)
            External(atoms[i], external);
    }

    // simulation in a Swindow
    RunSwindow(copper, 0.2);
}

Vector force(Atoms& p, Atoms& q)
{
    Vector u = Location(q);

    u -= Location(p);
    double r = magnitude(u);
    double a = R0/r;
    a = a*a*a;
    a *= a;
    u *= 12*DEPTH*a*(a-1)/(r*r);
    return u;
}

Vector external(Atoms& p)
{

```

where original is a vector array storing the initial positions of the particles. The complete program is listed below.

```
#include <Boltzmann/PPscheme.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/Swindow.h>
#include <Boltzmann/Unit.h>
#include <Boltzmann/Macro.h>
#include <Boltzmann/Integrator.h>
#include <Boltzmann/Function.h>

#define centimeter 1/1.804*1e8
#define second 1/3.273*1e15
#define gram 1/5.275*1e26

const double DEPTH = 0.1 electronvolts;
const double R0 = 2.551 angstroms;
const double kappa = 0.2 *electronvolt/(angstrom*angstrom);
const int N = 250;
const int M = 2000;

class LFparticle: LF(Newton);
class Atoms: INHERIT(LFparticle,
    void update(double t) {d = 8*magnitude(v) angstroms;}
);

Vector r0[N];
Atoms *base;

main()
{
    Vector force(Atoms&, Atoms&);
    Vector external(Atoms&);
    Vector low(0, 0), high(100 angstroms, 100 angstroms);
    int i, t;

    PPscheme copper(low, high, 3*R0);
    Atoms atoms[N];

    // initialization
    for (i= 0; i< N; i++) {
        Mass(atoms[i], M);
```

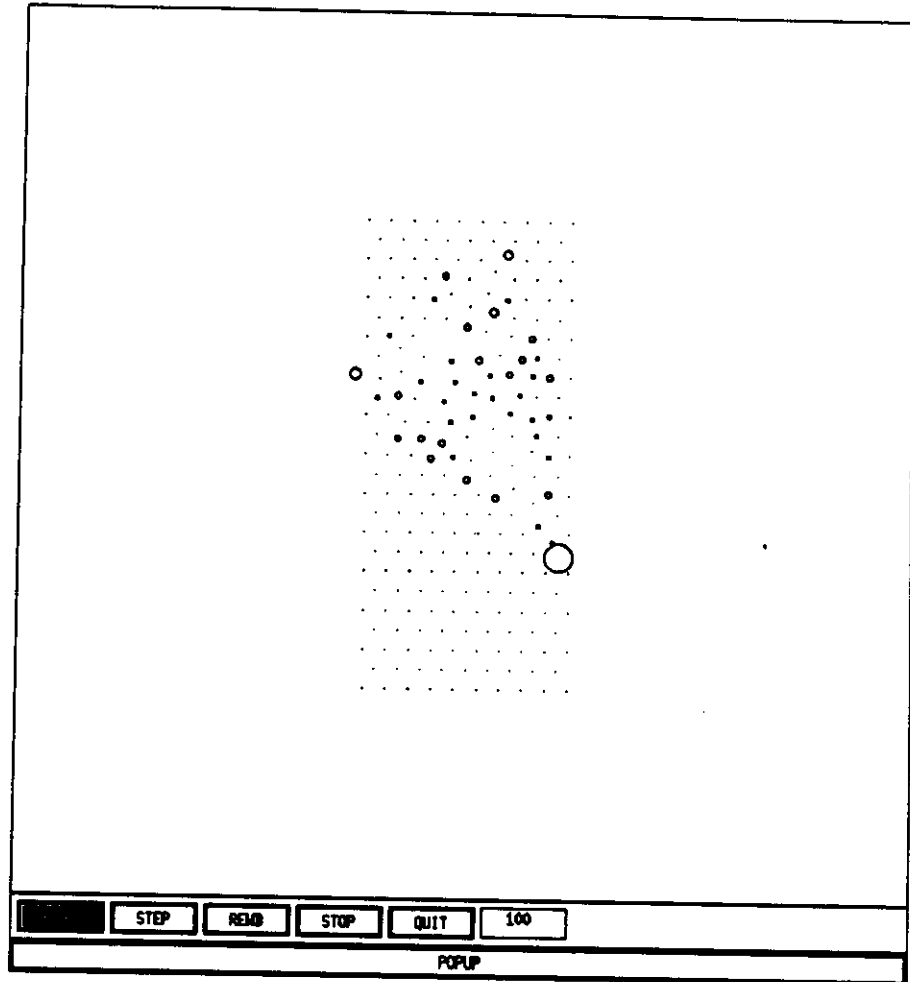


Figure B.7: Radiation displacement of atoms

```
Vector f = Location(p);  
  
f -= r0[&p-base];  
f *= -kappa;  
return f;  
}
```

To compile the program, the following command is issued:

```
bzm program.c
```

where bzm is an alias for

```
g++ -O \!* -lBoltzmann -lg++ -lm -lXw -lXt -lX11
```

A snapshot of the metal at time step 100 is given in Figure B.7, where the size of the circles represents the magnitude of velocity and thus the kinetic energy of the atoms.

```

    VectorArray() {point = 0;}
    VectorArray(int, int dim);
    VectorArray(int, int, int dim);
    VectorArray(int, int, int, int dim);
    ~VectorArray() {}
    Vector& operator[](int i) {return point[i];}
    Vector& operator()(int i) {return point[i];}
    Vector& operator()(int i, int j) {return arrayv[i](j); }
    Vector& operator()(int i, int j, int k)
        {return arrayv[i](j, k);}
};

////////////////////////////////////

inline ScalarArray :: ScalarArray(int n1)
{
    int i;

    // delete point;
    point = new float[n1];
    for (i= 0; i< n1; i++)
        point[i] = 0;
}

inline ScalarArray :: ScalarArray(int n1, int n2)
{
    int i;

    arrayv = new ScalarArray[n1];
    for (i= 0; i< n1; i++)
        arrayv[i].ScalarArray(n2);
}

inline ScalarArray :: ScalarArray(int n1, int n2, int n3)
{
    int i;

    arrayv = new ScalarArray[n1];
    for (i= 0; i< n1; i++)
        arrayv[i].ScalarArray(n2, n3);
}

inline VectorArray :: VectorArray(int n1, int dim)

```

APPENDIX C

Boltzmann Definition Files

C.1 Array.h

```
#ifndef _array_h
#pragma once
#define _array_h 1

#include <Boltzmann/Vector.h>

class ScalarArray {
    friend class PMScheme;
protected:
    union {
        float *point;
        ScalarArray *arrayv;
    };
public:
    ScalarArray() {point = 0;}
    ScalarArray(int);
    ScalarArray(int, int);
    ScalarArray(int, int, int);
    ~ScalarArray() {delete point;}
    float& operator[](int i) {return point[i];}
    float& operator()(int i) {return point[i];}
    float& operator()(int i, int j) {return arrayv[i](j); }
    float& operator()(int i, int j, int k)
        {return arrayv[i](j, k);}
};

class VectorArray {
protected:
    union {
        Vector *point;
        VectorArray *arrayv;
    };
public:
```



```

#include <Boltzmann/Scheme.h>
#include <Boltzmann/PPscheme.h>
#include <Boltzmann/PMscheme.h>

#include <Boltzmann/Vector.h>
#include <Boltzmann/RanVar.h>
#include <Boltzmann/Unit.h>
#include <Boltzmann/Constant.h>
#include <Boltzmann/Macro.h>
#include <Boltzmann/Integrator.h>
#include <Boltzmann/Function.h>
#include <Boltzmann/Swindow.h>

#endif

```

C.3 Boundary.h

```

#ifndef _boundary_h
#pragma once
#define _boundary_h 1

#include <Boltzmann/Particle.h>

enum BC {Open, Bounded, Elastic, Periodic, Dirichlet, Neumann};

class Boundary {
    friend class Scheme;
    friend Boundary* Next(Boundary* p) {return p->next;}
protected:
    int dim;
    Vector low, high;
    Boundary *next;
public:
    Boundary(double x0, double x1)
        {dim = 1; low.Vector(x0); high.Vector(x1); next = 0;}
    Boundary(double x0, double x1, double y0, double y1)
        {dim = 2; low.Vector(x0, y0); high.Vector(x1, y1);
        next = 0;}
    Boundary(Vector& low0, Vector& high0)
        {dim = dimension(low0); low = low0; high = high0;
        next = 0;}
    ~Boundary() {}

```

```

    {
        int i;

        point = new Vector[n1];
        for (i= 0; i< n1; i++)
            point[i].Vector(dim);
    }

inline VectorArray :: VectorArray(int n1, int n2, int dim)
{
    int i;

    arrayv = new VectorArray[n1];
    for (i= 0; i< n1; i++)
        arrayv[i].VectorArray(n2, dim);
}

inline VectorArray :: VectorArray(int n1, int n2, int n3, int dim)
{
    int i;

    arrayv = new VectorArray[n1];
    for (i= 0; i< n1; i++)
        arrayv[i].VectorArray(n2, n3, dim);
}

#endif

```

C.2 Boltzmann.h

```

#ifndef _boltzmann_h
#pragma once
#define _boltzmann_h 1

#include <Boltzmann/Particle.h>
#include <Boltzmann/Newton.h>
#include <Boltzmann/Laplace.h>
#include <Boltzmann/Coulomb.h>
#include <Boltzmann/Kepler.h>
#include <Boltzmann/Euler.h>
#include <Boltzmann/Vortex.h>

```

```

Coulomb(): Laplace() {charge = 0;}
Coulomb(Vector& r0, Vector& v0, double m, double q)
    :Laplace(r0, v0, m) { charge = q; }
~Coulomb() {}
int isCoulomb(Particle& p) {return p.classID() == COULOMB;}
virtual void internal(Particle& e) {
    Vector ff = eforce(CONVERT(e, Coulomb));
    e.influence() += ff; f -= ff;
}
virtual double strength() {return charge;}
virtual Vector eforce(Coulomb& e);
virtual int size() {return sizeof(*this);}
// this is necessary to get the right size of the particle
virtual CLASS_ID classID() {return COULOMB;}
};

inline Vector Coulomb::eforce(Coulomb& e)
{
    double d;
    Vector u = e.r;

    u -= r;
    d = magnitude(u);
    switch (dimension(u)) {
    case 1:
        u *= charge*e.charge/d;
        return u;
    case 2:
        u *= charge*e.charge/(d*d);
        return u;
    case 3:
        u *= charge*e.charge/(d*d*d);
        return u;
    }
}

#endif

```

C.6 Euler.h

```

#ifndef _euler_h
#pragma once

```

```

    int InTheBoundary(Vector&);
    void Bounce(Particle&, BC);
};

#endif

```

C.4 Constant.h

```

#ifndef _constant_h
#pragma once
#define _constant_h 1

#include <Boltzmann/Unit.h>

const double SpeedOfLight      = 2.998e10 * centimeter/second;
const double ElectronCharge     = 4.8e-10 esus;
const double ElectronMass      = 0.911e-27 grams;
const double ProtonMass        = 1.6725e-24 grams;
const double GConstant         =
    6.67e-11 * newton*meter*meter/(kilogram*kilogram);
const double GAcceleration     =
    9.806650e2 * centimeter/(second*second);
const double BoltzmannConstant = 1.38054e-16 ergs;

#endif

```

C.5 Coulomb.h

```

#ifndef _coulomb_h
#pragma once
#define _coulomb_h 1

#include <Boltzmann/Laplace.h>
#include <Boltzmann/Macro.h>

class Coulomb: public Laplace {
    friend double& Charge(Coulomb& p) {return p.charge;}
    friend void Charge(Coulomb& p, double q) {p.charge = q;}
protected:
    double charge = 0;
public:

```

```

// 4th-Order Runge-Kutta

#define RK4(base) public base { \
protected: \
    Vector r0, v0, f0, v1; \
public: \
    virtual int integrate(double t, int i) { \
        switch (i) { \
            case 1: r0 = r; v0 = v; f0 = f; \
                v *= 0.5*t; r += v; f *= 0.5*t/m; \
                v = v0; v += f; return 1; \
            case 2: v1 = v; f *= 2; f0 += f; v *= 0.5*t; \
                r = r0; r += v; f *= 0.25*t/m; \
                v = v0; v += f; return 1; \
            case 3: v1 += v; f *= 2; f0 += f; v *= t; \
                r = r0; r += v; f *= 0.5*t/m; \
                v = v0; v += f; return 1; \
            case 4: v1 *= 2; v1 += v0; v1 += v; v1 *= t/6; \
                r = r0; r += v1; f += f0; f *= t/(6*m); \
                v = v0; v += f; return 0; \
            default: return 0; \
        } \
    } \
    virtual int size() {return sizeof(*this);} \
}

// 2nd-Order Runge-Kutta

#define RK2(base) public base { \
protected: \
    Vector r0, v0; \
public: \
    virtual int integrate(double t, int i) { \
        switch (i) { \
            case 1: r0 = r; v0 = v; v0 *= 0.5*t; r += v0; \
                v0 = v; f *= 0.5*t/m; v += f; return 1; \
            case 2: r = v; r *= t; r += r0; \
                v = f; v *= t/m; v += v0; return 0; \
            default: return 0; \
        } \
    } \
    virtual int size() {return sizeof(*this);} \
}

```

```

#define _euler_h 1

#include <Boltzmann/Particle.h>

class Euler: public Particle {
public:
    Euler() {}
    Euler(Vector& r0) {r = r0;}
    ~Euler() {}
    int isEuler(Particle& p) {return p.classID() == EULER;}
    virtual int integrate(double t, int s) {
        if (s > 1) return 0;
        r += t*f; return 0;
    }
    virtual int size() {return sizeof(*this);}
    // this is necessary to get the right size of the particle
    virtual CLASS_ID classID() {return EULER;}
};

typedef Euler Fluid;

#endif

```

C.7 Integrator.h

```

#ifndef _integrator_h
#pragma once
#define _integrator_h 1

#include <Boltzmann/Vector.h>

// Euler

#define EULER(base) public base { \
public: \
    virtual int integrate(double t, int i) { \
        if (i > 1) return 0; \
        Vector u = v; \
        u ** t; r += u; f ** t/m; v += f; return 0; \
    } \
    virtual int size() {return sizeof(*this);} \
}

```

```

double d;
Vector u = e.r;

u -= r;
d = magnitude(u);
switch (dimension(u)) {
case 1:
    u *= GravitationalConstant*m*e.m/d;
    return u;
case 2:
    u *= GravitationalConstant*m*e.m/(d*d);
    return u;
case 3:
    u *= GravitationalConstant*m*e.m/(d*d*d);
    return u;
}
}

#endif

```

C.9 Laplace.h

```

#ifndef _laplace_h
#pragma once
#define _laplace_h 1

#include <Boltzmann/Newton.h>

class Laplace: public Newton {
    friend class PMScheme;
    double Strength(Laplace& p) {return p.strength();}
public:
    isLaplace(Particle& p) {return p.classID() == LAPLACE;}
    virtual double strength() {return 0;}
    virtual int size() {return sizeof(*this);}
    // this is necessary to get the right size of the particle
    virtual CLASS_ID classID() {return LAPLACE;}
};

#endif

```

```
// Leap-Frog
```

```
#define LF(base) public base { \  
public: \  
    virtual int integrate(double t, int i) { \  
        if (i > 1) return 0; \  
        f += t/m; v += f; f = v; f += t; r += f; \  
        return 0; \  
    } \  
    virtual int size() {return sizeof(*this);} \  
} \  
  
#endif
```

C.8 Kepler.h

```
#ifndef _kepler_h  
#pragma once  
#define _kepler_h 1  
  
#include <Boltzmann/Laplace.h>  
#include <Boltzmann/Macro.h>  
#include <Boltzmann/Constant.h>  
  
class Kepler: public Laplace {  
public:  
    int isKepler(Particle& p) {return p.classID() == KEPLER;}  
    virtual void internal(Particle& e) {  
        Vector ff = gforce(CONVERT(e, Kepler));  
        e.influence() += ff; f -= ff;  
    }  
    virtual Vector gforce(Kepler& e);  
    virtual double strength()  
        {return sqrt(GravitationalConstant)*m;}  
    virtual int size() {return sizeof(*this);}  
    // this is necessary to get the right size of the particle  
    virtual CLASS_ID classID() {return KEPLER;}  
};  
  
inline Vector Kepler::gforce(Kepler& e)  
{
```



```
extern void add_pane(Widget, menu_struct*, XContext, int);

#endif
```

C.11 Macro.h

```
#ifndef _macro_h
#pragma once
#define _macro_h 1

#define REAL double
#define TYPE(class, result) result class::
#define SQR(x) ((x)*(x))
#define INHERIT(base, fun) public base { \
public: \
    fun \
    virtual int size() {return sizeof(*this);} \
}
#define CONVERT(base, subclass) (*((subclass *) &base))

inline int MOD(int x, int y)
    { if(x >= 0) return x%y; else return MOD(y+x, y); }

#define DEACTIVATE(base) public base { \
public: \
    virtual int integrate(double t, int i) {return 0;} \
    virtual int size() {return sizeof(*this);} \
}

#endif
```

C.12 Newton.h

```
#ifndef _newton_h
#pragma once
#define _newton_h 1

#include <Boltzmann/Particle.h>
#include <Boltzmann/Macro.h>

class Newton: public Particle {
```

C.10 Libxs.h

```
#ifndef _libxs_h
#pragma once
#define _libxs_h 1

extern "C" {
#include "X11/Intrinsic.h"
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <Xw/Xw.h>
#include <Xw/Form.h>
#include <Xw/RManager.h>
#include <Xw/SText.h>
#include <Xw/TextEdit.h>
#include <Xw/WorkSpace.h>
#include <Xw/PButton.h>
#include <Xw/BBoard.h>
#include <Xw/MenuBtn.h>
#include <Xw/Cascade.h>
#include <Xw/PopupMgr.h>
}
extern "C" void exit(int);

#define FONTHEIGHT(f) \
    ((f)->max_bounds.ascent + (f)->max_bounds.descent)

typedef struct {
    char    name[20];
    void    (*switcher)();
    caddr_t func;
    caddr_t data;
} menu_struct;

extern void get_dimension(Widget w, Dimension *width,
    Dimension *height);
extern void get_GC(Widget w, GC *gc);
extern void get_EraseGC(Widget w, GC *xgc);
extern void get_Pixmap(Widget w, Pixmap *pix);

extern Widget create_one_line_text_widget(char*, Widget, Arg*, int);
extern Widget create_menu_pane(Widget, char*);
```

```

    NEWTON, LAPLACE, COULOMB, KEPLER,
    EULER, VORTEX, INVISCID, VISCOUS
};

class Particle {
    friend class Scheme;
    friend class ParticlePtrArray;
    friend Vector& Location(Particle& p) {return p.r;}
    friend Vector& Velocity(Particle& p) {return p.v;}
    friend float Diameter(Particle& p) {return p.d;}
    friend int Color(Particle& p) {return p.color;}
    friend void Location(Particle& p, Vector& r) {p.r = r;}
    friend void Velocity(Particle& p, Vector& v) {p.v = v;}
    friend void Diameter(Particle& p, float diam) {p.d = diam;}
    friend void Color(Particle& p, int index) {p.color = index;}
    friend void Internal(Particle& p, Vector (*force)())
        {p.inforce = (IntF) force;}
    friend void External(Particle& p, Vector (*force)())
        {p.exforce = (ExtF) force;}
protected:
    Vector r, v, f;
    float d = 0;
    unsigned int color = 0;
    IntF inforce = 0;
    ExtF exforce = 0;
    Particle *next; // is only used in class ParticlePtrArray
public:
    Particle() {inforce = 0; exforce = 0; next = 0; }
    Particle(Vector& r0, Vector& v0, IntF force=0)
        {r = r0; v = v0; inforce = force; exforce = 0; next = 0;}
    ~Particle() {}
    int isParticle(Particle& p) {return p.classID() == PARTICLE;}
    virtual void internal(Particle& p) {
        if (inforce) p.f += (*inforce)(*this, p);
        if (p.inforce) p.f += (*(p.inforce))(p, *this);
    }
    virtual void external()
        { if (exforce) p.f += (*exforce)(*this); }
    virtual void reset() {f = 0;}
    virtual int integrate(double t, int s) {
        if (s > 1) return 0; v *= t; r += v; return 0;
    }
    virtual void output(ostream& os) {

```

```

        friend double& Mass(Newton& p) {return p.m;}
        friend void Mass(Newton& p, double mass) {p.m = mass;}
protected:
    double m = 1;
public:
    Newton(): Particle() {}
    Newton(Vector& r0, Vector& v0, double mass, IntF force=0)
        : Particle(r0, v0, force) {m = mass;}
    ~Newton() {}
    int isNewton(Particle& p) {return p.classID() == NEWTON;}
    virtual void internal(Particle& p) {
        Vector ff;
        if (inforce) ff = (*inforce)(*this, CONVERT(p,Newton));
        p.influence() += ff; f -= ff;
    }
    virtual int integrate(double t, int s) {
        if (s > 1) return 0;
        f *= t/m; v += f; f = v; f *= t; r += f; return 0;
    } // leap-frog
    virtual int size() {return sizeof(*this);}
    virtual CLASS_ID classID() {return NEWTON;}
};

typedef Newton Classical;

#endif

```

C.13 Particle.h

```

#ifndef _particle_h
#pragma once
#define _particle_h 1

#include <Boltzmann/Vector.h>
#include <stream.h>

class Particle;
typedef Vector (*IntF)(Particle&, Particle&);
typedef Vector (*ExtF)(Particle&);

enum CLASS_ID {
    PARTICLE,

```

```

};

inline Particle& ParticleBag::operator[](int i)
{
    int t = 0;
    ParticleArray *p = cluster;
    while (p != 0) {
        if (i-t < p->arraySize) return (*p)[i-t];
        t += p->arraySize;
        p = p->next;
    }
}

#endif

```

C.14 PMScheme.h

```

#ifndef _pm_scheme_h
#pragma once
#define _pm_scheme_h 1

#include <Boltzmann/Scheme.h>
#include <Boltzmann/Laplace.h>
#include <Boltzmann/Array.h>

class PMScheme: public Scheme {
protected:
    void NGPassignment(Laplace&, ScalarArray&);
    void CICassignment(Laplace&, ScalarArray&);
    void NGPinterpolation(Laplace&, VectorArray&);
    void CICinterpolation(Laplace&, VectorArray&);
protected:
    int len[3];
    Vector h;
    ScalarArray phi, rho;
    VectorArray beta;
public:
    PMScheme(): Scheme() {}
    PMScheme(Vector& low, Vector& high, int n1, BC b=Periodic);
    PMScheme(Vector& low, Vector& high, int n1, int n2,
             BC b=Periodic);
    PMScheme(Vector& low, Vector& high, int n1, int n2, int n3,

```

```

        os << r << v << "\n"; }
    virtual void update(double t) {}
    IntF intf() {return inforce;}
    ExtF extf() {return exforce;}
    Vector& influence() {return f;}
    virtual int size() {return sizeof(*this);}
    virtual CLASS_ID classID() {return PARTICLE;}
};

inline ostream& operator<<(ostream& os, Particle& p)
    {p.output(os); return os;}
inline Vector& lfLocation(Particle& p, double t)
    { return Location(p)-0.5*t*Velocity(p); }
inline void lfInit(Particle& p, double t) {
    // Location(p, Location(p)+0.5*t*Velocity(p));
}

////////////////////////////////////

class ParticleArray {
    friend class ParticleBag;
    unsigned particleSize, arraySize;
    void *array;
    ParticleArray *next;
public:
    ParticleArray()
        {arraySize = 0; particleSize = 1;
         array = 0; next = 0;}
    ParticleArray(int n, Particle *p)
        {arraySize = n; array = p;
         particleSize = p->size(); next = 0;}
    ~ParticleArray() {}
    int np() {return arraySize;}
    Particle& operator[](int i)
        {return *((Particle *) (array+i*particleSize));}
};

class ParticleBag {
    ParticleArray *cluster;
public:
    ParticleBag() {cluster = 0;}
    putIn(ParticleArray *pa) {pa->next = cluster; cluster = pa;}
    Particle& operator[](int i);
};

```

```

int Dim() {return dim;}
void Reset() {
    int i, k; // switch (dim)
    if (dim == 1) k = size[0];
    else if (dim == 2) k = size[0]*size[1];
    for (i= 0; i< k; i++) cell[i] = 0;
}
void Assign(Particle *p);
void Calculation();
void Calculation(Particle *p, int i, int j);
};

```

////////////////////////////////////

```

class PPscheme;
class Connection {
    friend class PPscheme;
protected:
    unsigned int    first;
    unsigned int    last;
    unsigned int    interval;
    Connection      *next;
public:
    Connection() {next = 0;}
    Connection(int i, int j, int k)
        {first = i; last = j; interval = k; next = 0;}
    ~Connection() {}
};

```

////////////////////////////////////

```

class PPscheme: public Scheme {
protected:
    Connection *conn = 0;
    ParticlePtrArray grid;
public:
    PPscheme(Vector& low0, Vector& high0, BC b=Open)
        : Scheme(low0, high0, b) {}
    PPscheme(Vector& low0, Vector& high0, double r,
        BC b=Open);
    ~PPscheme() {delete conn;}
    void MakeConnection(int i, int j, int k = 1) {
        Connection *p = new Connection(i, j, k);
    }
};

```

```

        BC b=Periodic);
    ~PMscheme() {}
    virtual void DensityAssignment();
    virtual void PotentialCalculation();
    virtual void InfluenceField();
    virtual void Interpolation(Laplace& e) {
        CICinterpolation(e, beta);
        e.influence() *= e.strength();
    }
    void Calculation();
};

#endif

```

C.15 PPscheme.h

```

#ifndef _pp_scheme_h
#pragma once
#define _pp_scheme_h 1

#include "Scheme.h"

////////////////////////////////////
typedef Particle *ParticlePtr;

class ParticlePtrArray {
    int dim;
    double low[3];
    double step[3];
    unsigned int size[3];
    ParticlePtr *cell;
    void DualPartListForce(Particle *p, Particle *q);
public:
    ParticlePtrArray() {dim = 0;}
    ParticlePtrArray(double x1, double h1, int n1)
        { dim = 1; low[0] = x1; step[0] = h1; size[0] = n1;
          cell = new ParticlePtr[n1]; }
    ParticlePtrArray(double x1, double h1, int n1,
        double x2, double h2, int n2)
        { dim = 2; low[0] = x1; step[0] = h1; size[0] = n1;
          low[1] = x2; step[1] = h2; size[1] = n2;
          cell = new ParticlePtr[n1*n2]; }

```



```

    virtual double operator()();
};

inline Uniform::Uniform(double low, double high, RNG *gen):(gen)
{
    if (gen == 0) pGenerator = *_gen;
    pLow = (low < high) ? low : high;
    pHigh = (low < high) ? high : low;
    delta = pHigh - pLow;
}

inline double Uniform::low() { return pLow; }

inline double Uniform::low(double x) {
    double tmp = pLow;
    pLow = x;
    delta = pHigh - pLow;
    return tmp;
}

inline double Uniform::high() { return pHigh; }

inline double Uniform::high(double x) {
    double tmp = pHigh;
    pHigh = x;
    delta = pHigh - pLow;
    return tmp;
}

inline double Uniform::operator()()
{
    return( pLow + delta * pGenerator -> asDouble() );
}

////////// Normal Random Variable //////////

class Normal: public Random {
    char haveCachedNormal;
    double cachedNormal;

protected:
    double pMean;
    double pVariance;
};

```

```

        p->next = conn;
        conn = p;
    }
    void Calculation();
protected:
    virtual void FixedForceCal();
};

overload MakeConnection;
inline void MakeConnection(PPscheme& scheme, int i, int j, int k)
    { scheme.MakeConnection(i, j, k); }
inline void MakeConnection(PPscheme& scheme, int i, int j)
    { scheme.MakeConnection(i, j); }

#endif

```

C.16 RanVar.h

```

#ifndef _randomvar_h
#pragma once
#define _randomvar_h 1

#include <Random.h>
#include <MLCG.h>
#include <stream.h>

////////// Uniform Random Variable //////////

static MLCG _gen(0, 1);

class Uniform: public Random {
    double pLow;
    double pHigh;
    double delta;
public:
    Uniform(double low, double high, RNG *gen = 0);

    double low();
    double low(double x);
    double high();
    double high(double x);

```

```

public:
    Maxwellian(double temp, double mass, RNG *gen = 0);
    virtual double operator()();
};

inline Maxwellian::Maxwellian(double temp, double mass, RNG *gen)
    : (gen)
{
    if (gen == 0) pGenerator = &_gen;
    T = temp;
    m = mass;
    haveCachedMaxwellian = 0;
}

////////// Ripple Random Variable //////////////////////////////////////

class Ripple: public Random {
protected:
    double alp, bet, muB, nuC, aa, bb;

public:
    Ripple(double alpha, double beta, double mu, double nu,
           double a, double b, RNG *gen = 0);
    virtual double operator()();
};

inline Ripple::Ripple(double a, double b, double alpha, double beta,
                      double mu, double nu, RNG *gen = 0) : (gen)
{
    if (mu*a+nu < 1 || mu*b+nu < 1) {
        cerr << "illegimate distribution function\n";
        exit(1);
    }
    if (gen == 0) pGenerator = &_gen;
    alp = alpha;
    bet = beta;
    muB = mu;
    nuC = nu;
    aa = a;
    bb = b;
}

```

```

    double pStdDev;

public:
    Normal(double xmean, double xvariance, RNG *gen = 0);
    double mean();
    double mean(double x);
    double variance();
    double variance(double x);
    virtual double operator()();
};

inline Normal::Normal(double xmean, double xvariance, RNG *gen)
    : (gen)
{
    if (gen == 0) pGenerator = &_gen;
    pMean = xmean;
    pVariance = xvariance;
    pStdDev = sqrt(pVariance);
    haveCachedNormal = 0;
}

inline double Normal::mean() { return pMean; };
inline double Normal::mean(double x) {
    double t=pMean; pMean = x;
    return t;
}

inline double Normal::variance() { return pVariance; }
inline double Normal::variance(double x) {
    double t=pVariance; pVariance = x;
    pStdDev = sqrt(pVariance);
    return t;
};

////////// Maxwellian Random Variable //////////

class Maxwellian: public Random {
    char haveCachedMaxwellian;
    double cachedMaxwellian;

protected:
    double T;
    double m;

```

```
        { for (int i= 0; i< n; i++) s.Advance(t); }  
  
#endif
```

C.18 Unit.h

```
#ifndef _unit_h  
#pragma once  
#define _unit_h 1  
  
// default unit: cgs  
  
#define centimeter 1  
#define centimeters *centimeter  
#define gram 1  
#define grams *gram  
#define second 1  
#define seconds *second  
#define esu 1  
#define esus *esu  
#define dyne gram*centimeter/(second*second)  
#define dynes *dyne  
#define erg dyne*centimeter  
#define ergs *erg  
  
// mks  
  
#define meter 100*centimeter  
#define meters *meter  
#define kilogram 1000*gram  
#define kilograms *kilogram  
#define coulomb 3e9*esu  
#define coulombs *coulomb  
#define newton 1e5*dyne  
#define newtons *newton  
#define joule 1e7*erg  
#define joules *joule  
  
// others  
  
#define angstrom 1e-8*centimeter  
#define angstroms *angstrom
```

```
#endif
```

C.17 Scheme.h

```
#ifndef _scheme_h
#pragma once
#define _scheme_h 1

#include <Boltzmann/Boundary.h>
#include <Boltzmann/Particle.h>

class Scheme {
    friend void Install(Scheme& s, Boundary& rg) {s.Install(rg);}
    friend void Load(Scheme& scheme, Particle* ps, int number) {
        scheme.np += number;
        scheme.particles.putIn(new ParticleArray(number, ps));}
    friend int Num(Scheme& s) {return s.np;}
    friend int Dim(Scheme& s) {return s.dim;}
    friend Vector& UpperBound(Scheme& s) {return s.high;}
    friend Vector& LowerBound(Scheme& s) {return s.low;}
public:
    // dim: dimension of the space; np:number of particles;
    int dim, np;
    Vector low, high;
    ParticleBag particles;
protected:
    BC boundary;
    Boundary *region = 0;
public:
    Scheme() {dim = 0; np = 0; region = 0; boundary = Open;}
    Scheme(Vector& low0, Vector& high0, BC b=Open);
    ~Scheme() {}
    Particle& operator[](int k) {return particles[k];}
    virtual void Calculation();
    virtual void Advance(double t);
    void Install(Boundary& rg) {rg.next = region; region = &rg;}
protected:
    virtual void InternalForceCal();
};

inline void Advance(Scheme& s, double t) { s.Advance(t); }
inline void Run(Scheme& s, double t, int n)
```

```

Vector& operator-(Vector&);
Vector& operator*(Vector&);
Vector& operator+(double);
Vector& operator*(double);
Vector& operator-() {return (*this)*(-1);}
Vector& operator+=(Vector&);
Vector& operator-=(Vector&);
Vector& operator/=(Vector&);
Vector& operator+=(double x)
    {for (int i= 0; i< dim; i++) vec[i] += x; return *this;}
Vector& operator-=(double x)
    {for (int i= 0; i< dim; i++) vec[i] -= x; return *this;}
Vector& operator*=(double x)
    {for (int i= 0; i< dim; i++) vec[i] *= x; return *this;}
Vector& operator/=(double x)
    {for (int i= 0; i< dim; i++) vec[i] /= x; return *this;}
Vector& operator=(Vector&);
Vector& operator=(double);
float& operator[](int i) {return vec[i];}
print();
};

```

```

inline Vector& operator+(double x, Vector& r) {return r+x;}
inline Vector& operator-(double x, Vector& r) {return -r+x;}
inline Vector& operator*(double x, Vector& r) {return r*x;}
inline Vector& operator-(Vector& r, double x) {return r+(-x);}
inline Vector& operator/(Vector& r, double x) {return r*(1/x);}
inline double magnitude(Vector& r) {return sqrt(dot(r, r));}

```

```

class Memory {
    Vector *vec;
    Memory *next;
public:
    Memory(Vector* p, Memory *m) {vec = p; next = m;}
    ~Memory() {if (next != 0) delete next; delete vec;}
};

```

```

extern Memory *garbage;
inline void DeleteGarbage() {delete garbage; garbage = 0;}

```

```

#endif

```

```

#define electronvolt 1.60219e-12*erg
#define electronvolts *electronvolt
#define eV electronvolt
#define eVs electronvolts
#define calorie 4.186*joule
#define calories *calorie

#endif

```

C.19 Vector.h

```

#ifndef _vector_h
#pragma once
#define _vector_h 1

extern "C" {
#include <math.h>
#include <sys/file.h>
    // this line is added to override conflict definition for flock
}
#include <stream.h>

class Vector {
    friend ostream& operator<<(ostream& os, Vector& v);
    friend int& dimension(Vector& v) {return v.dim;}
    friend double dot(Vector&, Vector&);
    friend Vector& cross(Vector&, Vector&);
protected:
    unsigned int dim;
    float *vec;
public:
    Vector() {dim = 0; vec = 0;}
    Vector(double x) {dim = 1; vec = new float[dim]; vec[0] = x;}
    Vector(double x, double y)
        {dim = 2; vec = new float[dim]; vec[0] = x; vec[1] = y;}
    Vector(double x, double y, double z) {dim = 3;
        vec = new float[dim]; vec[0] = x; vec[1] = y; vec[2] = z;}
    ~Vector() {delete vec;}
    Vector(const Vector&);
    void Setup(int);
    Vector& operator+(Vector&);

```



```

    if (dimension(omega) == 0) return Vector();

    Vector u = Location(a); u -= r;
    switch (dimension(r)) {
    case 1: cerr << "ERROR: no 1D Vorticity\n";
            exit(1);
    case 2:
            x = magnitude(u); t = 1/(2*PI*x);
            if (x < delta)
                return omega[0]*t/delta*Vector(-u[1], u[0]);
            else
                return omega[0]*t/x*Vector(-u[1], u[0]);
            break;
    case 3:
            cerr << "Sorry: 3D has not been implemented yet\n";
            exit(1);
    }
}

```

```
typedef Vortex Inviscid;
```

```
#endif
```

C.21 Swindow.h

```

#ifndef _swindow_h
#pragma once
#define _swindow_h 1

#include <Boltzmann/Libxs.h>
#include <Boltzmann/Scheme.h>

#define MAXCOLOR      50
#define MAXPOINTS     500

enum COLOR {
    White, Black, Blue, Navy, LightBlue, SeaGreen,
    Green, Yellow, Brown, Orange, Pink, Red
};

class Swindow;

```

C.20 Vortex.h

```
#ifndef _vortex_h
#pragma once
#define _vortex_h 1

#include <Boltzmann/Euler.h>
#include <Boltzmann/Macro.h>

class Vortex: public Euler {
    friend class VPscheme;
    friend Vector& Vorticity(Vortex& v) {return v.omega;}
    friend double& Size(Vortex& a) {return a.delta;}
    friend void Vorticity(Vortex& v, Vector& w) {v.omega = w;}
    friend void Size(Vortex& a, double d) {a.delta = d;}
protected:
    Vector omega;
    double delta = 1e-3;
public:
    Vortex(): Euler() {}
    Vortex(Vector& r0, Vector& w, double size=1e-3)
        :Euler(r0) {omega = w; delta = size;}
    ~Vortex() {};
    int isVortex(Particle& p) {return p.classID() == VORTEX;}
    virtual void internal(Particle& e) {
        if (dimension(omega) != 0)
            e.influence() += velocity(CONVERT(e, Euler));
        if (isVortex(e)) {
            Vortex *p = (Vortex *) &e;
            if (dimension(Vorticity(*p)) != 0)
                f += p->velocity(*this);
        }
    }
    virtual Vector velocity(Euler&);
    virtual int size() {return sizeof(*this);}
    // this is necessary to get the right size of the particle
    virtual CLASS_ID classID() {return VORTEX;}
};

inline Vector Vortex::velocity(Euler& a)
{
    double x, t;
```

```

~Swindow() {}
void Init(Widget);
void Activate(Scheme&, double);
void Draw(Scheme&);
void DrawPoint(Vector&, float = 0, int = 0);
void DrawLine(Vector&, Vector&, int = 0);
void SetBound(Vector& x, Vector& y)
    {xdim = x; ydim = y;}
void SetXBound(Vector& x) {xdim = x;}
void SetYBound(Vector& y) {ydim = y;}
void SetStop() {stop_flag = 1;}
void SetQuit() {quit_flag = 1;}
void ResetStop() {stop_flag = 0;}
int IfStop() {return stop_flag;}
int IfQuit() {return quit_flag;}
void InstallDraw(char*, DrawFun);
void Timing(int);
void Cartesian(Scheme&);
void PhaseSpace(Scheme&);
void Spectrum(Scheme&);
};

overload DrawPoint;
overload DrawLine;

inline void InstallDraw(Swindow& window, char* label, DrawFun fun)
    { window.InstallDraw(label, fun); }
inline void DrawPoint(Swindow& window, Vector& r, float d, int color)
    { window.DrawPoint(r, d, color); }
inline void DrawPoint(Swindow& window, Vector& r)
    { window.DrawPoint(r); }
inline void DrawLine(Swindow& window, Vector& r1, Vector& r2,
int color) { window.DrawLine(r1, r2, color); }
inline void DrawLine(Swindow& window, Vector& r1, Vector& r2)
    { window.DrawLine(r1, r2); }
inline void SetBound(Swindow& window, Vector& x, Vector& y)
    { window.SetBound(x, y); }
inline void SetXBound(Swindow& window, Vector& x)
    { window.SetXBound(x); }
inline void SetYBound(Swindow& window, Vector& y)
    { window.SetYBound(y); }
extern void RunSwindow(Scheme&, double);
inline void Run(Swindow& window, Scheme& scheme, double t)

```

```

typedef void (*DrawFun)(Swindow&, Scheme&);
typedef void (Swindow::*DrawMenu)(Scheme&);

typedef struct {
    GC gc;
    GC xorgc;
    Pixmap pix;
    Scheme *ptr;
    Swindow *window;
    DrawFun drawfun;
    DrawMenu draw;
    int clean_window = 1;
    double t;
} scheme_struct;

struct {
    XArc      data[MAXCOLOR][MAXPOINTS];
    int       npoints[MAXCOLOR];
} points;

class Swindow {
    friend void InstallColor(Swindow& w, char** colors, int n)
        { w.colorlist = colors; w.n_colors = n; }
protected:
    menu_struct drawlist[10];
    unsigned int n_menus = 0;
    char **colorlist;
    unsigned int n_colors = 0;
    unsigned int ncolors;
    unsigned int stop_flag = 0;
    unsigned int quit_flag = 0;
    Widget toplevel, canvas, clock;
    Widget play, step, rewd, stop, quit, popup;
    Vector xdim, ydim;
    Vector xdim0, ydim0;
    unsigned long colors[100];
public:
    scheme_struct data;
    Pixmap empty;
    Dimension canvas_width, canvas_height;
public:
    Swindow() {}
    Swindow(Vector&, Vector&);

```

```
    { window.Activate(scheme, t); }  
#endif
```

