MOLECULAR DYNAMICS OF ATOMIC CLUSTERS: AN OBJECT ORIENTED APPROACH

John M. Schneiderman and Brian G. Moore
Chemistry Department
Augustana College
Sioux Falls, SD 57197

ABSTRACT

The purpose of the project was to develop a graphical user interface (GUI) to simulate and display the interactions of a small amount of atoms in a cluster, specifically with interactions defined by a pairwise Lennard-Jones potential energy. Another goal of the project was to design the coding of the program in such a way that it would reflect how the various parts of the program interacted with each other. To this end four classes were developed: an atom, a cluster, a base simulation, and a molecular dynamics Lennard-Jones simulation. The atom class contained all the information that was necessary for the simulations, such as the potential energy of the atom. The program was designed with the ability to add new types of simulations, with no need to significantly modify the GUI code. In the GUI program the ability to watch the simulation as it takes place was introduced, as well as the ability to modify how the atoms themselves appear. Results are presented for a specific system, a cluster of rare gas atoms. A histogram of the potential energy is bimodal for a small (N appprox. 30-200 atom) cluster, a quantitative measure of the two qualitatively different regions--surface and bulk.

Keywords
Clusters, molecular dynamics, computational, simulations, C++, object-oriented, undergraduate, liquid, phase transition

INTRODUCTION

Involving undergraduate students in research in computational chemistry techniques can be very difficult because they must have extensive preparation before they can become effective contributors. In particular, for development of new computational methods, programming experience is required. For many problems, especially those of primarily a numerical nature, a wide variety of software and/or programming languages may be used to accomplish the desired result. Meanwhile, the programming languages and development environments which are commonly taught at the undergraduate level continue to change, reflecting the changing situation in companies which use software and the software development industry. The current undergraduate computer science curriculum
typically introduces object-oriented programming (usually either C++ or Java) early on, and most development is done in a graphical integrated development environment, for example Microsoft Visual Studio.NET.

In this paper, we discuss the adaptation of a classical molecular dynamics simulation program from a procedural language, (Fortran/C), to an object-oriented approach (specifically, using C++). This program, which was previously run only as a console application under UNIX (Linux), was redesigned to make use of the object-oriented capabilities available in C++. In addition, a graphical interface was developed for MS Windows, allowing real-time visualization of the atoms as they progress in the simulation.

In undertaking a computational problem where writing new code is involved, an important consideration is always whether the questions at hand could be answered by an existing software product. Hence there is an important coupling between the scientific questions and the techniques one brings to bear on it. In this paper, we will discuss a specific system, an atomic cluster of rare-gas atoms, and use it as a basis to discuss a particular software approach and development environment that involves integrating the kinds of skills mentioned above—particularly object-oriented programming.

Our goal here was not merely to just convert the syntax of the program from C to C++, we were also interested in how the approach to the problem is different. Object-oriented programming is a completely different paradigm than the structured approach. In the work here, we have tried to completely re-think the problem from scratch, applying objective programming principles.

CLUSTERS AND PHASE TRANSITIONS: THE LIQUID STATE

Atoms or molecules expanded into a vacuum (such as in a molecular beam) will undergo expansion cooling and under certain conditions begin to form small clusters of atoms (Jellinek, 1999). Once formed, we can consider these clusters as isolated systems; typically the clustering occurs in only the initial stages of the expansion. These clusters can then be observed—for example by mass spectrometry, optical spectroscopy, or other means—and their properties studied (Johnson, et al. 2002). Not surprisingly, they can show very different behavior from that seen in the bulk. While the details of the clustering process are of course dependent on the particular nature of the species in the expansion, the phenomenon of cluster formation is quite general; dispersive (van der Waals) attractive interactions are present in any system.

For this reason, we chose in our simulations to focus on the dispersive interactions alone, which may be modeled well by the Lennard-Jones potential (Maitland, et al., 1981); hence, when we speak of the kind of cluster we are modeling, one can perhaps best think of it as that of a rare-gas atom, such as argon or krypton. We have included the dispersive attraction and a short range repulsive interaction, but we have eliminated any possibility of actual “chemistry” occurring—no chemical reactions. The functional form of the Lennard-Jones potential is

\[
 u(r) = 4\epsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^{6} \right]
\]
where $r$ is the interatomic separation, $\sigma$ is the atomic diameter, and $\varepsilon$ is an energy parameter characterizing the depth of the attractive part of the potential well. In reporting results in this paper, we will use dimensionless reduced units: reduced temperature $T^* = kT/\varepsilon$, reduced distance $r^* = r/\sigma$, reduced density $\rho^* = (N/V)\sigma^3$, reduced time $t^* = (\varepsilon/m\sigma^2)^{1/2}t$. The potential energy is taken to be pair-wise additive; the total potential energy of the system is the sum over all the pair interactions in the system.

The typical range of sizes of atomic clusters studied in our work was from $N=13$ to about $N=300$. One of the most interesting features of a cluster of atoms interacting through primarily dispersive interactions is the observation of a phase transition (liquid/solid) in the cluster, with characteristics which are dependent on the size of the cluster. This phase transition has been seen clearly not only in computer simulations but also in experimental observations (Hahn and Whetton, 1988). One can think of the process mentally in the following way: an isolated group of atoms in a vacuum at a low enough temperature will stay “stuck” together for long enough to be characterized. Now imagine slowly heating the cluster; this is not possible for a real cluster in a vacuum (nor is it the way that the simulations are conducted in this work), but this thought experiment serves to illustrate the idea. At some point, desorptions of atoms will begin to occur. However, for many cluster sizes in the ranges we studied, the cluster will melt before the evaporation of atoms becomes so rapid as to preclude obtaining structural information about the liquid cluster (Moore and Al-Quraishi, 2000).

The liquid state of the cluster is particularly interesting. For example, if the cluster is of a size near $N = 55$, the melting will occur near a temperature of $T^* = 0.35$, while in a bulk sample of the same atoms, the triple point is at about $T^* = 0.80$. Thus the liquid in the cluster is existing at a temperature very much below where the liquid would cease to be stable in the bulk.

**SIMULATION METHODS: GENERAL APPROACH**

The methods for the molecular dynamics simulation were the same in both the older versions of the program (implemented with a procedural approach) and the newer versions developed for this work (implemented with an object-oriented approach). The velocity Verlet algorithm was used, with constant energy conditions (micro-canonical ensemble) (Verlet, 1967). A standard step size of $dt^* = 0.005$ was used for all runs. Initial conditions for the clusters were face-centered cubic arrangements with all particles having zero velocities. An important adjustable parameter in our approach is the reduced lattice parameter $a^* = a/\sigma$, where $a$ is the edge length of the unit cell of the initial fcc lattice. Our goal is primarily to study liquid clusters, so a relatively large lattice parameter is typically used, in the range of $a^* = 1.7$ to 2.0, corresponding to initial cluster densities in the range of $\rho^* = 0.80$ to 0.50. By comparison, close-packed for the fcc lattice would occur at $a^* = 1.414 = \sqrt{2}$. The larger parameters typically lead to liquid clusters, with complete destruction of the initial fcc lattice. The fact that the simulation bears no “memory” of the initial fcc configuration is also confirmed by the radial distribution functions (RDFs) which were acquired and...
qualitative observations of the cluster using the graphical user interface (GUI) which displays the positions of the atoms as the simulation progresses--developed for this paper. Both of these ways of observing the cluster (RDF and GUI) are discussed in more detail later in later sections.

**PROGRAM DESIGN: PROCEDURAL APPROACH (C/FORTRAN)**

The program to implement the molecular dynamics simulation was originally developed in a mainframe environment (DEC/VAX and SGI systems) in Fortran 77 (f77). Subsequent to that, the program was ported to the C language for use in a Linux environment, most recently using the free GNU gcc compiler (Stallmann). Both versions of the program (C and f77) had very similar designs; the conversion to C was primarily of a syntactical nature. These older versions of the program were run exclusively as console applications, with data files for input and output. Data analysis consisted of using other applications (such as gnuplot) to visualize results after the program finished running.

Both C and Fortran are procedural languages and the program designs reflected that choice. The procedural programming paradigm focuses on the procedures and sequences of steps required for the problem at hand. Functions are used to help organize the processing of the algorithm, etc.

The program was organized roughly into three sections: input and pre-run processing, the dynamics run (including running data acquisition), and post-run operations and writing out of the final output files. Within the run loop (the central section of the program), there were three functions, move1, force, and move2, called in sequence for each move of the atoms, that contained the essence of the Verlet MD algorithm, patterned after routines of these same names provided in the text by Allen and Tildesley (1987). The force loop is the most time-consuming part of the program, as it contains the sum over all unique pairs of atoms, so as to calculate the force on each atom; the number of calculations in the force loop thus scales as roughly N^2, where N is the number of atoms.

**PROGRAM DESIGN: MODULAR/ OBJECT-ORIENTED APPROACH (C++)**

Our goal was to re-design the program with an object-oriented approach. This was accomplished in stages. The first stage was the relatively simple task of merely rewriting the program in C++, but preserving essentially all the original code, making only minor changes for syntax as required. The second stage was to make use of some of the classes provided in the C++ Standard Template Library (STL) (Stepanov and Lee, 1995)--in other words to make use of this standard set of pre-defined classes but not introducing any new user-defined ones. The third stage was to create a set of classes designed specifically for the problem at hand, yet general enough to be useful for future development of the code (for example to change the simulation method or to change the potential, etc.).
Using the Standard Template Library (STL)

The second stage in this process of moving from C to C++, the use of the STL, is worthy of some comment. The STL provides several very useful classes which can make program code easier to write and read and also provide more flexibility and some protection against errors. An example of this is the vector class. This class is one of several container classes provided in the STL. In C, data arrays must be dimensioned, and making an array larger or smaller is tedious. Errors resulting from trying to access non-existent elements are common. The vector container provides methods to expand or contract the size of the array with a simple syntax; also, access may be checked for out-of-range condition. An example of the ease with which the vector class may be used is shown in the following code-snippet, where data is being read from a file (one value per line) into a vector.

```cpp
vector<double> data;
double value;
while(input >> value) data.push_back(value);
int n = data.size();
cout << n << " data points read\n";
```

In the above example, data is the vector in which we are placing the numbers, which are read one line at a time from the input stream. The methods `push_back()` and `size()` are examples of functions provided with the class, allowing for an automatic increase in the size of the vector as the numbers are put into it and interrogation of the number of elements when this process is finished. The corresponding code to achieve the same results takes up several more lines when rendered in C. Note also in this example that `data` is an example of an “object” in C++. Methods (or functions) are applied to the object using the postfix notation, thus `data.size()` is the operation of the class function `size()` on the object `data`.

Object-Oriented Design: User-Defined Classes

The power of object-oriented programming is that it allows the user to define new classes, similar to the vector class (defined in the STL); another way of thinking of a class is just as an abstract data type. Data types such as `int` or `double` are system provided; C++ allows the user to define completely new data types.

However, the object-oriented programming paradigm is completely different than that for procedural programming. Rather than the main focus being the procedures, the first step is to design and create the classes which will be used. The classes should reflect the nature of the problem at hand. Ideally, the classes should reflect real objects if possible (Stroustrup 1997).

Towards this end, the first main step in re-designing the simulation program in an object-oriented way was the creation of two classes, the atom class and the cluster class. The atom class contains the x, y, and z values for the position, veloc-
ity, and acceleration of the given atom, as well as the kinetic and potential energy of the atom. Accessors are provided to extract these variables, and method functions provided to change them. The class was designed to be flexible, with the idea that for further development, more variables could be added. For example, in our program, only one kind of atom is used. In future programs, one might want to have a mixture of different kind of atoms, so one might want to add variables such as the Lennard-Jones diameter and potential well depth, etc. An important consideration in the program design was the fact that the position, velocity, and acceleration values are frequently accessed. These variables are needed in the force loop, which is the most computationally intensive part. In the older C and Fortran programs, these arrays were declared as common or global variables, accessible to any function. This special status for these particular variables was dealt with by making the accessor functions to these variables in the atom class inline reference functions. The syntax in the class declaration for these functions is shown below. Thus, for example, in the molecular dynamics move1 routine, we can make a statement of the style below.

\[
clus[i].vx() += 0.5*dt*clus[i].ax();
\]

Note here that the \texttt{ax()} accessor is used to retrieve the value of the acceleration, and then the \texttt{vx()} function is used to change the value of the acceleration.

class Atom
{
  private:
    double rxval,ryval,rzval;//positions
    double vxval,vyval,vzval;//velocities
    double axval,ayval,azval;//accelerations
    double Vatom,Katom;// potential energy, kinetic energy
  public:
    /* Class functions. */
    Atom(); // constructor that initializes all values to zero
    double& rx() {return rxval;}
    double& ry() {return ryval;}
    double& rz() {return rzval;}
    double& vx() {return vxval;}
    double& vy() {return vyval;}
    double& vz() {return vzval;}
    double& ax() {return axval;}
    double& ay() {return ayval;}
    double& az() {return azval;}
    double getVatom() const;
    double getKatom() const;
void move(double rxnew, double rynew, double rznew);
void push(double vxnew, double vynew, double vznew);
void jerk(double axnew, double aynew, double aznew);
void putVatom(double val); void putKatom(double val);
}

The cluster class is primarily a collection of atoms, plus related accessors and other methods. One way to implement this is to set up the cluster class to inherit from the STL vector class. Thus, methods such as push_back(), size(), etc. are automatically provided. The cluster object is just a vector of atoms, plus a few other related variables.

class Cluster: public vector<Atom>
{
private:
    double xcm, ycm, zcm; // center of mass
    double vxcm, vycm, vzcm; // velocity "center of mass"
    double V, K; // total potential energy, kinetic energy
public:
    /* Class functions. */
    Cluster(); // constructor

    /* fccgen is not a constructor, but it does kind of the same thing. It generates an fcc lattice. */
    void fccgen(int m, double rhostar);

    /* Accessors. */
    double getV() const;
    double getK() const;
    void pollVK(ostream& ostr) const;
    double getxcm() const; double getycm() const; double getzcm() const;
    double getvxcm() const; double getvycm() const; double getvzcm() const;

    /* Calculators & such. */
    void calcVK();
    void calccm();
    void putV(double value); void putK(double value);
    void changecm(double x, double y, double z, double vx, double vy, double vz);
    void readcn(string name);
    void writecn(string name);
};

The atom and the cluster class are just data storage and retrieval objects. The actual algorithmic procedures to implement the simulation must act on these objects. Toward this end, we developed two more classes, which are more abstract, but reflect the organization of tasks in the program. The main idea was to create a class that represented the simulation run itself—the central part of the original
C program. The specific method we chose to implement the molecular dynamics simulation involved using the Verlet method, Lennard-Jones potential, but these choices are only one specific case of many possibilities. Thus we decided to develop first a base simulation class, containing features common to any molecular dynamics simulation. Then we created a class called simulationmdlj containing the specific details of the method we have chosen (Lennard-Jones molecular dynamics). The simulationmdlj class inherits all the characteristics of the more general simulation class. The class declarations for these two classes are shown below.

```cpp
class Simulation
{
  public:
    // class constructor
    Simulation();
    void setdefaults(string filename, int nstep);
    void load(string currname, string prevname, int nstep);
    void analyze();
    void setstarttime(string base_name);
    void writeout();
    void summarize(int n, int nstep);
  protected:
    Cluster clus;
    string curr_run_name, prev_run_name, defaults_file;
    int nstep, iprint, igr, irho, iPKhist, iscreen;
    double V, K, E, EN, VN, KN, Temp;
    double t, dt, tstart;

    /* Histograms */
    Rdfhist rdfdata;
    Rhohist rhodata;
    PEhist PEdata;
    KEhist KEdata;
};

class SimulationMDLJ : public Simulation
{
  public:
    // class constructor
    SimulationMDLJ();

    void run(int nstep);

    // MD LJ core functions
    void force();
    void move1();
    void move2();
};
```
Note the presence of the three functions $\text{move1}$, $\text{force}$, and $\text{move2}$ in the $\text{simulationmdlj}$ class; these are essentially the same functions as those found in the original C/f77 versions of the program but adapted to the new programming environment.

Graphical User Interface (GUI)

The description of the program design so far applies only to the “number crunching” procedures of the simulation program. Data is read in, either from input files or the command line, the simulation progresses, and output is written out. This is a console application and can be compiled and run in any environment where a C++ compiler is available. On a Linux system, the GNU gcc compiler is always available, and is the one we used solely for our UNIX runs. Under MS Windows, the console application can be developed in a variety of ways. Microsoft has a fully featured C++ compiler as part of their Visual Studio. NET development environment. Other compilers for MS Windows are also available, for the console application development. One can also use Windows ports of the free GNU gcc compiler; in our work, using gcc with Windows, we used either the mingw package or the cygwin environment.

The MS Visual C++ approach to development provides an application framework which has support for development of programs with a graphical user interface (GUI). Fairly standard features, such as developing an input screen, with boxes for input parameters, are easily added. We decided as part of the program development to also add a screen which would show the positions of the particles on the screen as the simulation progressed. An example snapshot of the screen during a simulation is shown in Figure 1. The program has several features, not all of which will be described here. A few examples are presented in Figures 2 and 3. The development of the GUI provided a good example of a graphical development problem for the undergraduate student in this case and involved skills which built directly on those taught in programming courses. In addition, the ability to directly visualize the cluster during the simulation provided a very real scientific benefit in terms of helping to verify whether or not the initial lattice structure was destroyed.

RESULTS AND DISCUSSION

The main benefit from writing code to approach a computational problem is that one can ask questions which have never been asked before. Commercial and free programs are available to do molecular dynamics simulations on simple systems, but it is not easy to modify these programs to take data in a new way. We show in this section a few examples of the results from the program, with an emphasis on features which are hard to (or impossible to) realize in readily-available existing programs.

One important measure of cluster structure is the radial distribution function (RDF). The idea of the RDF is a sort of “local” density, referenced to the particle frame. To acquire the RDF, we first pick a particular atom, then col-
Figure 1: A snapshot of the graphical user interface while the simulation is running. The cluster shown contained originally 108 atoms in a face-centered cubic lattice. In this run, several atoms were desorbed during the initial transient period of the run, and the lattice structure was completely destroyed, leaving a smaller liquid cluster. The desorbed atoms are discarded for subsequent runs and structural features are extracted on the remaining liquid cluster.

Figure 2: The file viewer screen included as part of the GUI. Shown are some of the summary numbers which are written out for the entire run. The units used are in reduced form, defined in the text.
lect density information in radial shells centered on the chosen atom. Then we resolve this data into radial bins, normalize, and divide by the bulk density; the RDF shows deviations away from the bulk density in the environment nearby a given atom.

An example of the RDF taken for a cluster of about \( N = 55 \) atoms is shown in Figure 4. Note that the large radius limit goes to zero instead of one as in a bulk sample. This is because we are working with a finite system; for large enough radial distance, eventually we are outside the cluster. Shown in Figure 4 are two representative examples, for liquid and solid clusters of about the same size. The liquid cluster shows a typical solvation-type shell of a few atomic diameters, similar to a bulk liquid, however, note the split second peak. This feature, which is not present in a bulk liquid, is ubiquitous for the cluster liquids we have studied. Such a feature is also often seen in glassy liquids. The solid cluster RDF has a few very distinct differences compared to the liquid cluster. First, the relative amplitudes of the two contributions to the split second peak are different; the larger radius contribution is enhanced. Secondly, the solid shows a clear peak at about \( r^* = 1.7 \) which is not present at all in the liquid cluster. We believe this feature to be a unique signature of the presence of solid ordering to some extent in the cluster, and we have used it to help identify when the cluster is in the pure liquid state.

Another unique type of analysis we have performed is that of acquiring histograms of the kinetic energy and potential energy from the liquid cluster runs. Our initial intention was to try and use these histograms as an aid in quickly identifying particles which have desorbed from the cluster. These histograms have proved useful for that purpose, but they have turned out to be interesting in other ways as well. First, the kinetic energy histogram is unsurprising. Its form is close to that predicted by a Maxwell-Boltzmann distribution. However, the potential energy histogram appears qualitatively as shown in Figure 5. It is very strongly bimodal and appears to be dominated by two smooth peaks. An obvious interpretation is that we are seeing the two qualitatively different regions, surface and bulk, contributing to these histograms. This interpretation is reinforced by the dependence on size. When we compare the \( N = 256 \) histogram...
Figure 4: The radial distribution (RDF) of two clusters of approximately the same size, one in the liquid state, one in the solid state. The qualitative features of the RDF function—especially the lack of a peak near \( r^* = 1.7 \) in the liquid state—are typical.

Figure 5: Potential energy histograms of clusters in the liquid state, for two different sizes. The lower energy peak presumably represents the bulk contribution, the higher energy peak the surface contribution.
to the N=55 histogram, we find that the lower energy peak (the bulk part) has been enhanced at the expense of the higher energy peak (the surface part). These histograms have been normalized to one, so the total area under both curves is the same on the plot. It makes sense that the relative contribution of the bulk part will increase as the cluster size increases; the fraction of the atoms at or near the surface decreases with cluster size. We are in the process of performing more studies to confirm our interpretation of the potential energy histograms. In particular, we hope to be able to spatially resolve the histogram acquisition process, thus conclusively proving exactly which part of the cluster gives rise to each peak in the histogram.

To our knowledge, the potential energy histograms shown here are the first time such data has been collected for a simulation of a liquid cluster or droplet. This shows the power of being able to write unique code to analyze a problem of this nature.

PROGRAM AVAILABILITY

This project was not undertaken with the goal of reaching a program for general distribution, but rather to illustrate the general concept of incorporating object-oriented features as part of program design, in part as a way to help integrate undergraduate student participation at an early stage. However, we will provide a copy of the program (under the general terms of a GNU license) at the corresponding author’s website, http://faculty.augie.edu/~bmoore, for those who are interested in seeing more details of the code developed relating to this paper. Both a console version and a Windows GUI version (with the same internal molecular dynamics methods) have been developed.

ACKNOWLEDGEMENTS

The development of the object-oriented approach to molecular dynamics was assisted by conversations with Dr. Chuck Yeung, Penn State Erie. This work was partially supported by an ARAF grant from Augustana College, through the Bush foundation.

LITERATURE CITED


