A Nanoscience Modeling and Computation System

James L. Noyes
Wittenberg University

Presented at the Oberlin Conference on Computation and Modeling: 2005
Oberlin College, Oberlin, Ohio
November 4-6, 2005

Based upon a presentation at the National Academy of Sciences
Keck Futures Initiative Conference: Designing Nanostructures at the Interface between Biomedical and Physical Systems
Arnold & Mabel Beckman Center, Irvine, California
November 18-21, 2004

Abstract
Introduction
Nanoscience Computations
Creating Nanoscience Application Codes
A Prototype Nanoscience System
Conclusion
References
Abstract:
Computational science is the scientific investigation of physical processes by the use of computational models and methods that rely heavily on computer simulation and optimization. The purpose is to gain both a quantitative and qualitative understanding of this process. In nanoscience applications, one deals with atomic and molecular structures at the nanometer ($10^{-9}$ meter) scale. This is a hundred to a thousand times smaller than a typical biological cell or bacterium. Because of this, there are a great many additional challenges related to modeling these physical processes. In nanotechnology, the numbers are especially important because things are to be built or engineered and require very high accuracy. Nanosystems present new types of multiscale modeling and algorithmic time and storage challenges which involve not only the employment of nano building blocks and interfaces, but also in the dynamics, assembly, and growth of these structures.

Hardware and software tools to solve some nanoscience modeling problems already exist. Workstations can now perform many computations, while clusters of computers or supercomputers can successfully be employed for more complex problems. The most complex of these nanoscience problems await the introduction of advanced algorithms as well as even more powerful computers. Software tools and application-oriented computer programming languages that already exist must be assembled and investigated for their suitability in solving nanosystem problems. Some new software tools will need to be created.

One approach to this tool investigation process is to develop a prototype system in a high-level software package, such as Mathematica, where well-tested mathematical tools already exist in the areas of numeric processing, symbolic processing, and visualization. A prototype system can be developed to perform nanoscience modeling and computation in order to identify and investigate how various implemented algorithms perform in terms of time and storage efficiency as well as correctness and robustness.
Introduction:
Computational science (COSC) is the scientific investigation of physical processes by the use of computational models and methods that rely heavily on computer simulation and optimization. The purpose is to gain both a qualitative and quantitative understanding of this process (i.e., to gain insight and numerical estimates). In nanoscience applications, there are many additional challenges related to these physical processes. Here one deals with atomic and molecular structures at the nanometer ($10^{-9}$ meter) scale, typically in the 1-100 nm range. In nanotechnology, the numbers that are used are especially important because things are to be studied or engineered. This means that one can have a very large number of very small things. The number ranges can sometimes be beyond the capability of a standard programming language.

Correspondingly, the creation of a software tool to aid in the understanding of nanoscience processes requires its own types of fundamental computations which can tax a modeling system that, in turn, relies upon application programs, a high-level computing language, and underlying hardware to answer important nanosystem questions. Such a modeling system, of course, allows us to "manipulate" (approximately simulate the manipulation of) real objects that we cannot yet see. As stated in [6], "Because of the rapid advance of experimental investigations in this area [of nanoscience], the need for quantitative understanding of matter at the nanoscale is becoming more urgent, and its absence is increasingly a barrier to progress in the field quite generally."

The equations that are used at the molecular level can sometimes cause the time-honored computer integer and floating-point arithmetic (which are actually approximations to integer and real number arithmetic in mathematics) to fail when implemented in the standard science and engineering computer languages, such as Fortran and C/C++. No doubt new equations and formulas will need to be developed as we learn how more things behave at the nano scale. Quantum mechanics will also play a role. As Feynman stated in [4], "At the atomic level, we have new kinds of forces and new kinds of possibilities, new kinds of effects. The problems of manufacture and reproduction of materials will be quite different."
Nanoscience Computations:
In the following, a few types of computations will be shown. These computations are chosen to be representative of the types that may cause problems for traditional types of software systems when computing with nanoscience models.

- **Numeric Processing.** The **Nucleon-Number Density** (the number of nucleons per cubic meter) is given by \(0.17 \times 10^{45}\) in Benenson [2]. This value cannot even be represented in single precision or as an integer (32-bits) using a standard computer language because floating-point overflow will result! Double precision (64-bits) is better, but problems can arise there also. These types of problems can occur because the typical word size of the hardware is the limiting factor and may not be large enough to adequately represent the quantities at hand.

The **Lennard-Jones** approximate potential for diatomic molecules (see Benenson [2] and Jonsson [5]), is given below, where \(r\) is the ionic distance and the parameters \(\epsilon\) and \(\sigma\) depend upon the material and have been normalized to "reduced LJ units" below. It combines a strong repulsion, a weak Van der Waals attraction, and a potential well. As \(r\) approaches 0, the potential gets quite large due to the 12th power of \(r\) in the denominator (see Fig. 1 below). This can easily produce underflow if \(r\) is too large and overflow if \(r\) is too small. The common hardware underflow and overflow limits for single and double precision are approximately \(\pm10^{-38}\) & \(\pm10^{+38}\) and \(\pm10^{-308}\) & \(\pm10^{+308}\), respectively. There is no problem with our prototype language, even the function minimum is readily found.

In all of these cases, the suggested prototype system can correctly represent all of the numbers (integer, rational, real, and complex) and all of their computed values. This is because there is no limit (other than the amount of computer storage available) to the size of an integer or rational number and the precision can be arbitrarily extended for any floating-point numbers.

\[
\begin{align*}
\epsilon & = 1; \\
\sigma & = 1; \\
v[r] & := 4\epsilon \left( \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^{6} \right)
\end{align*}
\]
Plot[v[r], {r, 0.85, 3.0},
 AxesOrigin -> {0, 0}, PlotRange -> All, Frame -> True,
 PlotStyle -> {RGBColor[1, 0, 0], Thickness[0.01]},
 Background -> RGBColor[1, 1, 0],
 PlotLabel -> "Fig. 1: Lennard-Jones Model",
 AxesLabel -> {"Distance(r)", "Potential(v)"}];
Print["{Min. Potential, {r\rightarrow Distance}}: ",
 FindMinimum[v[r], {r, 2}]]
Symbolic Processing. Suppose one wished to find the derivative of the previous Lennard-Jones function. With a standard programming language, this could only be done numerically yielding both roundoff errors and truncation errors (due to the type of finite difference method that was used). Numerical differentiation is nearly always a risky proposition. On the other hand, a symbolic differentiation and simplification can be readily performed on the prototype system with no error whatsoever. The resulting function can be readily evaluated and plotted (see Fig. 2 below). In this case the previous minimum point can be confirmed (along with additional complex number solutions which are exact).

Notes: (1) In this example, the differentiation may easily be done by inspection. (2) The partial differentiation of multivariate functions is also easily accomplished and can also be used to confirm or locate all optima and stationary points.

\[
\epsilon = 1; \\
\sigma = 1; \\
v[r_] := 4 \epsilon \left( \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right)
\]
The derivative is: 4 \left( -\frac{12}{r^{13}} + \frac{6}{r^7} \right)

Its roots are:
\{ \{ r \rightarrow -2^{1/6} \}, \{ r \rightarrow 2^{1/6} \}, \{ r \rightarrow -(1)^{1/3} 2^{1/6} \}, \{ r \rightarrow (-1)^{1/3} 2^{1/6} \}, \{ r \rightarrow -(-1)^{2/3} 2^{1/6} \}, \{ r \rightarrow (-1)^{2/3} 2^{1/6} \} \}
String Processing and Pattern Matching. Another important area of symbolic processing for biologists as well as nanoscientists is that of string processing to search for and manipulate long (as well as short) DNA and RNA strings. It should readily be able to access not only local database files, but should be able to read and write genomic and protein databases and interface with commonly used web-oriented software (e.g., BLAST - Basic Local Alignment Search Tool).

(* Read DNA string - randomly generated here *)

dnastr = "";
dalen = 100;
Do[dnastr = StringInsert[dnastr, StringTake["ACGT",
   {Random[Integer, {1, 4}]}], 1], {dalen}];

dnastr

(* Determine its length *)
StringLength[dnastr]

TCTCATTCGCCACCAGTCGGCACTAGGATACAGCGAATACCCCGCTAAGCGATTGGCTATATG
ATCTATATACCCCTAA

(* Find all locations where "CCCG" appears *)

StringPosition[dnastr, "CCCG"]

{{43, 46}}

(* Show with the following character *)

StringCases[dnastr, "CCC ~ _"]

{CCC, CCCT}

(* Show the first with all of the following characters *)

StringCases[dnastr, "CCC ~ _"]

{CCCGCCTAAGCGATTGGCCACCTAAATTTAGTGGTCTATATGCTCTATACCCCTAA}
- **Visualization.** It is often necessary, especially at the nano-level, to be able not only to compute but to visualize as well, in order to gain a better understanding of the behavior of particles, atoms, molecules. 2D and 3D function and data plotting tools are essential. Below is an example given by Russkeepaa in [9] that shows a function with a behavior that might not be obvious (see Fig. 3 below):

```math
Plot3D[Cos[Sqrt[x^2 + y^2]], {x, -7, 7}, {y, -7, 7},
     PlotPoints -> 50, PlotLabel -> "Fig. 3: 3D Figure"];
```

![Fig. 3: 3D Figure](image-url)
Model Expressibility. Standard programming languages are very valuable and have been around since Fortran was designed in the mid-1950s. To use them, a reasonable understanding of the language constructs are needed and an appropriate amount of code must be written and (hopefully) well-tested before it is used. Unfortunately, even a simple formula can sometimes require an entire page of code and the resulting code may bear no resemblance to the original formula. If the user, such as a biochemist or nanoscientist, is not familiar with the programming language, he or she must communicate it to a programmer, adding an interface requirement and usually an additional delay.

The prototype system can solve many of these problems immediately just by the virtue that so many standard solution methods are already built into the system as language primitives. No extensive coding may be needed at all, just the ability to use the desired primitives.

The following formula is taken from Drexler in [3] and Rohlf in [8]. It deals with the Debye Model for solids in which the crystal vibrates as a whole. The vibrational energy of the crystal for a given material (e.g., copper) is quantized and one quantum is called a phonon. Fig. 4 below shows this temperature vs. energy plot for copper. Here the code looks almost exactly like the formula (a numerical integration was necessary here), reducing the chance for a coding error.

\[
N_A = 6.02214199 \times 10^{23};
\]

(* Avogadro's Constant: Units of mole\(^{-1}\) *)

\[
k_0 = 0.13806503 \times 10^{-23};
\]

(* Boltzmann Constant: Units of Joules/Kelvin *)

\[
h = 6.62606876 \times 10^{-34};
\]

(* Planck Constant: Units of Joule Second *)

\[
f_D = \frac{315 k}{h} \quad (* \text{Debye Frequency: } T_D=315^0 \text{ Kelvin for Cu } *)
\]

\[6.56354 \times 10^{12}\]

\[
E_m[T_] := \frac{9.0 N_A (k T)^4}{(h f_D)^3} N \int_{0}^{\frac{h f_D}{k T}} \frac{x^3}{e^x - 1} dx
\]
Plot[E_m[T], {T, 30, 300},
AxesOrigin -> {0, 0}, PlotRange -> All,
PlotStyle -> {RGBColor[1, 0, 0], Thickness[0.01]},
Background -> RGBColor[1, 1, 0],
PlotLabel -> "Fig. 4: Debye Energy Model",
AxesLabel -> {"Temperature(T)", "Energy(E_m)"}];
- **Model Extensibility.** In the cases where no language primitives exist, and no application libraries already exist, one must be able to code in the prototype language. Ideally, this code would be widely tested, standardized, and used to the greatest extent possible.

Sometimes it is tempting to just "code up" a method found in a textbook, a scientific journal article, or the world-wide web. This should be done with great caution! It is rare to find a scientist or engineer who is also skilled in producing effective algorithm implementations. Most of the really good software implementations do a lot of work "behind the scenes" in terms of both automatic error checking and in the use of hybrid algorithm code that automatically selects the best algorithm implementation for the given user-defined problem. In some cases, more than one algorithm will be used on different parts of the same problem.

According to many, the most commonly used popular molecular dynamics (MD) algorithm for time integration is the so-called Velocity Verlet Method. The derivation of the new particle position $r(t+\delta t)$ is based upon combining a forward and backward Taylor expansion to produce a truncation error that has a desirable local error order $O(\delta t^4)$, where $\delta t$ is the time step. Unfortunately, deriving the corresponding velocity $v(t+\delta t)$, produces an undesirable local error order $O(\delta t^2)$, which dominates the computation. So while this popular algorithm is very intuitive, it can suffer from a large truncation error. This can partially be overcome by a very small time step. Unfortunately (again) this very small time step causes many extra calculation steps (and a corresponding amount of computation time) to reach the same end point. This aggravates the roundoff error in computing the new velocity and position and therefore causes more accumulation error which builds with time. In the following, a fictitious but well-behaved forcing function is used so that the resulting differential equation has an analytic solution and the error can be investigated.

The Verlet algorithm code is shown first. This is followed by a plot (Fig. 5a) of the results of the Verlet algorithm (in one direction - the other two direction components are handled similarly). The resulting curve in black, called $r(t)$, looks plausible, but is badly flawed. The next (much smaller) piece of Mathematica code, calls its solution curve $x(t)$, which is shown in red. It uses a more sophisticated numerical algorithm and its associated plot (Fig. 5b) tell a different story. The third plot below (Fig. 5c) shows the true analytic solution in blue, called $y(t)$. Finally, the fourth plot (Fig. 5d) shows all three curves together and indicates how much better the sophisticated algorithm performed. The true solution "overlays" the sophisticated solution (showing as purple) because the numerical Mathematica result is basically identical to the true solution.
n = 1000;
t0 = 0.0;
tn = 10.0;
r0 = 0;
v0 = 75;
\[ \delta t = \frac{tn - t0}{n} \];
mass = 0.0066666666666667;
force[t_] := -e^{-t} \cos[t] ;
(* Direct Coding of the Verlet Algorithm *)
sol = {{t0, r0}};
told = t0;
rold = r0;
vold = v0;
fold = force[t0];
aold = fold / mass;

Do[
  (tnew = t0 + i \delta t;
   rnew = rold + \delta t vold + \frac{1}{2} \delta t^2 aold;
   sol = Append[sol, {tnew, rnew}];
   vnew = vold + \delta t aold;
   anew = force[tnew] / mass;
   rold = rnew;
   vold = vnew;
   aold = anew
  ), {i, 1, n}];
plotv = ListPlot[sol];

solution = NDSolve[{x''[t] == force[t]/mass,
                      x[t0] == r0, x'[t0] == v0, x, {t, t0, tn}}];
plotn = Plot[x[t] /. solution, {t, t0, tn},
            PlotStyle -> {RGBColor[1, 0, 0], Thickness[0.01]}];

Fig. 5a:  r(t) vs. t    The Verlet Algorithm Results

Fig. 5b:   x(t) vs. t    The Mathematica (Adams/Gear) Algorithm Results
\[ y[t_] := 75 \, e^{-t} \sin[t] \]

\[ ploota = \text{Plot}[y[t], \{t, t0, tn\}, \]

\[ \text{PlotStyle} \to \{\text{RGBColor}[0, 0, 1], \text{Thickness}[0.005]\}; \]

**Fig. 5c:** \( y(t) \) vs. \( t \)  

The True Solution

**Fig. 5e:** \( r(t), x(t), y(t) \) vs. \( t \)  

Comparing Both Results to the Solution
Creating Nanoscience Application Codes:
The needed codes could be grouped into various categories. One class of codes would include nano-related utility routines. These could be produced by a range of skilled individuals.

Larger coding projects could be undertaken by groups of individuals who are skilled in specific algorithm-related areas and have a solid understanding of the proposed programming language. These would be codes that related to the so-called nano building blocks, such as quantum dots, nanotubes, clusters, and associated interfaces. Other codes could help simulate and study DNA and ribosome activities. Cellular automata could be used to study and learn the rules of combination in terms of molecular patterns [12]. Quantum Monte Carlo methods could be used to investigate the electronic structure of molecules [6]. New evolutionary algorithms will be developed. Multiscale simulation and optimization modeling techniques will become necessary. These many application codes have the potential of being the most taxing and challenging codes in computer science as millions to billions of particle configurations may need to be modeled.

A thorough software evaluation process will be necessary and a collection of benchmark problems will need to be used. The evaluation will include all documentation as well.
**A Prototype System:**

A prototype system to aid in both modeling and solving nanoscience problems should possess the following attributes:

- **Perform All Types of High-Accuracy Computations Using High-Precision and Wide-Range Numeric Processing**
- **Perform All Types of Mathematical and Statistical Processing including Quantum Mechanics**
- **Perform All Types of Symbolic Processing to Create, Simplify, and Verify Equations and Formulas**
- **Permit Easy Visualization and Plot All Types of 2D and 3D Graphics of Data and Functions and Some 4D Functions**
- **Provide an Interactive Molecular Display Capability to Facilitate Nano Designs**
- **Contain Functional Primitives (Commands) to Solve a Broad Collection of Problems:**
  - Manipulate Mathematical Expressions
  - Provide Advanced Operations on Long Strings
  - Provide Function and Data Approximation Methods
  - Provide Ordinary and Multivariable Symbolic Differentiation
  - Provide Ordinary and Multivariable Integration
  - Solve All Linear and Many Nonlinear Systems of Equations
  - Solve All Systems of Ordinary Differential Equations Numerically and Many Symbolically
  - Solve Important Systems of Partial Differential Equations
  - Find the Local and Global Optima for Most Continuous and Many Discrete Optimization Problems
- **Possess an Extremely Flexible Input/Output and Database Access Capability**
- Facilitate the Creation of Realistic Nanoscience Simulation Codes

- Facilitate the Creation of On-Line Documentation for Every Piece of Extensible Code

- Permit Programs to be Readily Constructed, Compiled, and Standard Application Libraries Formed and Distributed

- Permit Interfacing with Existing Standard Language Nanoscience and Nanotechnology Codes

- Execute the Codes on Parallel Processing Computers Such as Clusters and Grids

- Permit Rapid Prototyping when Creating Code for a Standard Language Application is Necessary
Conclusion:
The proposed nanoscience system should be based upon the newly released Mathematica 5.2 system, since Mathematica already has most of the desired capabilities and is the most powerful, flexible, and sophisticated of the so-called "symbolic languages." It works when the standard programming languages cannot. The majority of the work in creating the proposed system would involve the construction of specialized program libraries to solve nanoscience and nanotechnology applications. This would include specialized molecular visualization software. For the situations when a standard programming language is necessary due to huge amounts of data or that real-time execution requirements cannot be met, then the proposed language system could still be used to facilitate rapid prototyping to verify the coding feasibility in a standard programming language.

*Note: All of the text, mathematics, and graphics presented here were done in Mathematica and placed in this printed Mathematica notebook exactly as shown.*
References: