APPLICATION OF AUTOMATIC DIFFERENTIATION TO MOLECULAR DYNAMICS

SIMULATION SCIENCES LABORATORY
LUCÍA GULLÓN, NIKHIL CHHABRA AND USHNISH SENGUPTA
**INTRODUCTION**

**MOLECULAR DYNAMICS: A PRIMER**

Molecular Dynamics is a simulation technique that focuses on the description of the atomic and molecular interactions of physical systems, which are modeled semi-classically [1]. The essence of the method lies in the interaction between macroscopic and microscopic properties of a system of particles. Molecular Dynamics is the most widely used simulation method in condensed matter physics. Other fields where MD is commonly used are chemical physics, material sciences or biochemistry. [4]

Molecular dynamics simulations compute the numerical solution of the classical equations of motion for a system of molecules or atoms. This is performed in a step-by-step manner. For instance, for a system consisting of a group of atoms, forces between the particles are derived from some potential model (Equation 1), subsequently yielding equations of motion that can be solved with finite difference algorithms (for example, Equation 2) [2].

\[
F_i = -\nabla V(r) \quad F_i = \sum_{j \neq i} F_{ij} \quad (1)
\]

\[
r_i(t + \delta t) = r_i(t) + v_i(t) + \frac{1}{2} a_i(t) \delta t^2 \quad (2)
\]

Molecular dynamics simulations give us an insight into very complex systems where one typically has little idea of what is going on at the microscopic level. Ideally, the Schrödinger wave equation should be solved if one wishes to model an atomistic system with the highest possible fidelity, however, quantum mechanical calculations are too computationally demanding for systems that consist of more than a few atoms. Molecular dynamics applies classical Newtonian mechanics to model systems but using force-fields that are often derived from Quantum Mechanics: thus representing a good compromise between accuracy and computational expense [3]. In this project, we simulate a Lennard-Jones system, a classic molecular dynamics model, in C++ and then explore the possible applications of a computational tool known as automatic differentiation to this model.

**DIFFERENTIATING DYNAMIC COMPUTER PROGRAMS**

There are many ways to obtain derivative information along with the ordinary output of a program. The most straightforward approach would be to go back to the equations that led to the program. Using these equations, one can write a new set of equations, whose solutions are the derivative of the initial result. Essentially, one ends up writing a new program which helps compute the desired derivatives. But this manual source code transformation, no matter how simple it sounds, is very time-
consuming and involved because it leads to discretizing new equations and then writing a new program. Plus, in some cases the functional form is not known beforehand or might be very complex. Therefore, an easy and cheap method of computing derivatives which simply uses the original program has to be found.

One option is to use finite difference method which uses the original program. For a given set of input parameters \( X \), the output is computed as \( F(X) = Y \). In general, both \( X \) and \( Y \) are vectors. Given now some normalized direction \( dX \) in the space of the inputs, one can run the program \( P \) again on the new set of inputs \( X + h \cdot dX \), where \( h \) is a very small positive number. Then, from a first order Taylor series expansion, a good approximation of the derivative may be computed easily by Equation 5.

\[
F'(x) \approx \frac{F(X+h\cdot dX) - F(X)}{h} + O(h) \quad (5)
\]

The centered finite difference approximation which is computed as indicated in Equation 6

\[
F'(x) \approx \frac{F(X+h\cdot dX) - F(X-h\cdot dX)}{2h} + O(h^2) \quad (6)
\]

usually gives a better, second order approximation, but incurs the added computational cost of an additional execution of the program \( P \). But the results are merely an approximation of the true derivative due to the remainder term in the Taylor series. The exact derivative may only be obtained from these formulas in the limit where \( h \) tends to zero. But this is impractical on a finite precision machine like a computer, since very small values of \( h \) can lead to truncation errors, and therefore to erroneous derivatives. This is the principal drawback of the finite difference method: some tradeoff must be found between truncation errors and approximation errors. Even if an optimal value for the perturbation \( h \) is found, the value of the derivative still carries an error. It is also evident that the computational cost for calculating the Jacobian using finite differences scales proportionally with the number of parameters, which is decidedly a disadvantage when there are lot of parameters in the model.

So, classical methods like symbolic differentiation or numerical differentiation run into various problems. Symbolic differentiation is manual, laborious and inefficient. It also presents the difficulty of converting a computer program into a single expression, whereas numerical differentiation can introduce errors and leads to erroneous derivatives. The problems of both classical methods are exacerbated when calculating higher derivatives. Problems like sensitivity analysis in molecular dynamics and gradient based optimization problems need partial derivatives of a function w.r.t. multiple inputs and numerical differentiation is slow at computing the partial derivatives of a function with respect to many inputs. Automatic differentiation solves all of these problems, at the expense of introducing more software dependencies and library calls.
In mathematics and computer algebra, automatic differentiation (AD), also called algorithmic differentiation or computational differentiation, is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

Automatic differentiation, just like the finite difference method, merely requires the original program $P$. But instead of executing $P$ on different sets of inputs, it builds a new, augmented program $P'$, that computes the analytical derivatives along with the original program. This new program is called the differentiated program. Precisely, each time the original program holds some value $v$, the differentiated program holds an additional value $dv$, the differential of $v$. Moreover, each time the original program performs some operation, the differentiated program performs additional operations dealing with the differential values. For example, if the original program, at some time during execution, executes the following instruction on variables $a, b, c$ and array $T$ (Equation 7).

$$a = b \cdot T(10) \quad \text{(7)}$$

then the differentiated program must execute additional operations on the variables and their differentials $da, db, dc$, and array $dT$, that must somehow amount to Equation 8.

$$da = db \cdot T(10) + b \cdot dt(10) + dc \quad \text{(8)}$$

The derivatives are now computed analytically, using the well known formulas on derivation of elementary operations. The calculated derivatives are now exact up to machine precision.

There are two ways to implement algorithmic differentiation: **Overloading** consists in telling the compiler that each real number is replaced by a pair of real numbers, the second holding the differential. Each elementary operation on real numbers is overloaded, i.e. internally replaced by a new one, working on pairs of reals, that computes the value and its differential. The advantage is that the original program is virtually unchanged, since everything is done at compile time. The drawback is that the resulting program runs slowly because it constantly builds and destroys pairs of real numbers. Moreover, implementing the "reverse mode" of algorithmic differentiation with overloading is technically difficult. The DCO tool for algorithmic differentiation, developed by the Numerical Algorithms Group [7] that was used in this project uses the overloading approach.
The competing approach is source transformation, which adds into the program the new variables, arrays, and data structures that will hold the derivatives, and also the new instructions that compute these derivatives. The advantage is that the resulting program can be compiled into an efficient code, and the "reverse mode" is much easier to implement. Tools are needed to perform this transformation on large programs correctly and rapidly.

Automatic differentiation may be performed in two fundamental modes: Forward/ Tangent Mode and Reverse/ Adjoint Mode.

In the forward mode, a tangent-linear version of the original program is built. The sensitivities of all outputs of the program with respect to its inputs can be computed at a computational cost that is proportional to the number of inputs. The computational complexity is similar to that of finite difference approximation. However, the desired derivatives are computed with machine accuracy. Truncation is avoided.

Reverse mode yields an adjoint program that can be used to perform the same task at a computational cost that is proportional to the number of outputs. For example, in large-scale nonlinear optimization a scalar objective that is returned by the given computer program can depend on a very large number of input parameters. The adjoint program allows for the computation of the gradient (the first-order sensitivities of the objective with respect to all parameters) at a small constant multiple R (typically between 3 and 30) of the cost of running the original program. It outperforms gradient accumulation routines that are based on finite differences or on tangent-linear code as soon as the size of the gradient exceeds R. The ratio R plays a very prominent role in the evaluation of the quality of derivative code.

In this project, we use automatic differentiation for

a) Force calculation: Forces on particles are obtained as the derivative of the interatomic potential energy w.r.t. their positions; hence it can be calculated with automatic differentiation.

b) Sensitivity analysis: In sensitivity analysis, the derivatives of the output thermodynamic quantities w.r.t. the parameters are sought. Prime facie it would seem that a straightforward application of automatic differentiation to our molecular dynamics code will yield sensitivity values. However, it gets more complicated: this will be discussed in the sections that follow.

SENSITIVITY ANALYSIS: THE WHAT AND WHY OF IT

Sensitivity analysis investigates how the variation in the output of a numerical model can be apportioned to variations of its input factors. A molecular dynamics simulation typically has dozens of input variables and parameters and a scientist is often interested in how the output thermodynamic quantities obtained from his simulations vary with respect to these parameters. Sensitivity analyses for molecular dynamics are usually conducted with the following objectives in mind.

a) Model calibration: The parameters of force fields in a molecular dynamics simulation are typically obtained via tuning and calibration of the model so that its
outputs show a good agreement with experiments of quantum mechanical calculations. Sensitivities of the outputs with respect to the force field parameters coupled with a gradient descent or similar algorithm can help us minimize the difference between the predictions of the model and the observations, thereby improving the model.

b) Uncertainty quantification: The parameters in a molecular dynamics model are often obtained via fitting. Hence, their values carry inherent uncertainties. Linear sensitivity values of a derived quantity in a model can be used to calculate the variance in the output thermodynamics quantity stemming from these uncertainties. If one has a quantity of interest \( R(\alpha) \) a first-order Taylor series expansion can be performed in the parameters \( \alpha \) of the model in Equation 9.

\[
\tilde{R}(\alpha) = \tilde{R}(\alpha^0) + \sum_{i=1}^{N} \frac{\partial \tilde{R}}{\partial \alpha_i} (\alpha^0) (\alpha - \alpha_i^0) + \ldots
\]

(9)

Then the variance of \( R \) can be easily obtained as in Equation 10,

\[
\sum_{i,j=1}^{N} S_i S_j \text{Cov}(\alpha_i, \alpha_j)
\]

(10)

where \( S_i \) is the linear sensitivity of \( R \) w.r.t. the parameter \( \alpha_i \) as defined above. [6]

c) Designing molecules: Once we have a well-calibrated and robust model, one can use the sensitivity information to explore the parameter space optimally and find new molecules whose parameters can give us the desired values of the thermodynamic properties.

A naïve approach to sensitivity analysis would be to simply perturb each parameter and running the code over and over with different values of the parameters. However, this is a computationally expensive approach, since a single run of a molecular dynamics simulation can be very processor-hungry. It would be better if one simply had to run the code once and obtain the derivatives directly, with a computational cost merely a small multiple of the original code without derivative calculation. This is the advantage that a sensitivity analysis using automatic differentiation can provide us with.

METHODS

THE SIMULATION SETUP

The model implemented in this work consists of a 2D or 3D system (both models were studied) of Lennard-Jones atoms of an ideal gas such as atoms of Argon. A total number \( N \) of atoms are uniformly distributed in a cubic (in the 3D case) or in a square (in the 2D case) lattice square as shown in Figure 1.
For the 2D case, initialization in a hexagonal lattice was also performed. For simplicity and accuracy, dimensionless units for mass, energy and length are used. This makes all the quantities in our simulation close to unity and prevents loss of precision.

<table>
<thead>
<tr>
<th>Property</th>
<th>Reduced Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>$r^* = r/s$</td>
</tr>
<tr>
<td>Time</td>
<td>$t^* = t/\tau = (m/s^2)^{1/2}$</td>
</tr>
<tr>
<td>Temperature</td>
<td>$T^* = k_BT/e$</td>
</tr>
<tr>
<td>Force</td>
<td>$f^* = f/s$</td>
</tr>
<tr>
<td>Energy</td>
<td>$\phi^* = \phi/e$</td>
</tr>
<tr>
<td>Pressure</td>
<td>$p^* = P/s^2/e$</td>
</tr>
<tr>
<td>Number density</td>
<td>$N^* = N/s^3$</td>
</tr>
<tr>
<td>Density</td>
<td>$\rho^* = s^3/m$</td>
</tr>
<tr>
<td>Surface tension</td>
<td>$\gamma^* = \sigma/s\epsilon$</td>
</tr>
</tbody>
</table>

Both periodic and open boundary conditions have been implemented. We simulate the micro-canonical NVE ensemble, where the total number of particles, $N$, the energy, $E$, and the volume, $V$ are kept constant. The system can experience exchanges between potential and kinetic energy, but the total energy must be conserved.

**PERIODIC BOUNDARY CONDITIONS AND MINIMUM-IMAGE CONVENTION**

Periodic boundary conditions are a specific type of boundary conditions determined by the cyclic behavior in space of the model. This is, the atoms will be interacting not only with all other atoms in the lattice squared, but also with all other images of the particles in the neighboring lattice squares as it is shown in Figure 2 [5]. This is used because we want to simulate the bulk behavior of the material and not its surface.
In order to simplify the computation further, the minimum-image convention is also applied to the model. This is a common form of PBC in which every particle in the simulation only interacts with the closest copy of all other particles in the squared lattice as it is represented in Figure 2 [6]. This is to say, all interactions between particles that are separated with a distance larger than \( \frac{L}{2} \) will be neglected. Figure 2 schematically presents the actual interactions that will be considered in the simulation under the periodic boundary conditions and minimum-image convention framework.

As a result, the simulation saves large amounts of computation since instead of an infinite number of force calculations, only the ones between the closest particles will be computed. This convention can be applied to the present model because the long-range interactions arising from the fast-decaying Lennard-Jones potential are weak and can safely be neglected.

THE ALGORITHM
The Molecular Dynamics simulation is programmed in a C++ code. Every few frames, the positions of the atoms are output in data files which are then visualized in Matlab. The code implemented in the presenting project follows the steps in Figure 3.
First, the system is defined to be under periodic boundary conditions and a microcanonical NVE ensemble. At the same time, position, velocity and acceleration are initialized. Atoms are uniformly distributed within the lattice (Figure 1). Every particle is initialized with the same velocity magnitude but in different directions.

The movement of each of the atoms is defined using Newton’s second law of motion. Forces are calculated from the gradient of the potential w.r.t. the position of each atom. From these two, the net force for each atom is obtained. The net force on one atom will depend on the others’ positions. (Figure 5).

After initialization, new velocities are calculated using Verlet integration. This method integrates Newton’s equations of motion to calculate the trajectories of the atoms [5]. Verlet integration is implemented in the code starting with the new positions calculation. As equation 11 shows, this new positions are derived from current positions, velocities and accelerations [6].

\[ x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 \quad (11) \]
From this, new accelerations are calculated via the gradient of the potential where the previously computed positions are used. The final step of Verlet integration updates the velocities using these two recent positions and acceleration values (Equation 12).

\[ v(t + \Delta t) = v(t) + \frac{1}{2} (a(t) + a(t + \Delta t)) \Delta t \]  

(12)

After performing the Verlet integration, the code applies the periodic boundary conditions as necessary, in order to keep the updated coordinates of the atoms within the square box. This loop will be executed for each time step until the final time of the simulation is reached.

Final physical quantities from MD simulations are obtained by the calculation of ensemble averages computed from theorems. In our code, the temperature is calculated from the kinetic energy using the equipartition theorem (Equation 13) and the pressure from the temperature, positions and forces with the virial theorem (Equation 14). Values for kinetic and potential energies are calculated within the simulation, when we calculate the velocities and the forces respectively.

The initial part of the trajectory is rejected for the calculation of thermodynamic quantities, since the starting configuration of the atoms is a highly improbable one and biases our calculations towards the initial conditions. We allow the system to equilibrate before we start calculating any thermodynamic quantities.

\[ < K > = \frac{1}{2} m \langle v^2 \rangle = \frac{3}{2} kT \]  

(13)

\[ PV = N k_B T + \frac{1}{3} < \sum_{i=1}^{N} r_i, F_i > \]  

(14)

FORCE CALCULATION USING AUTOMATIC DIFFERENTIATION

Interatomic Potentials are functions that represent the potential energy of a system of atoms. It can be written as a sequence of terms which depends on the interactions of two or more atoms at a time.

The potential must have both an attractive and a repulsive component if binding is to occur. All the molecular dynamics simulations use Interatomic potential as their basis. The total energy of the system can be written as Equation 15.

\[ V_{TOT} = \sum_i^N V_1(\vec{r}_i) + \sum_{i,j}^N V_2(\vec{r}_i, \vec{r}_j) + \sum_{i,j,k}^N V_3(\vec{r}_i, \vec{r}_j, \vec{r}_k) + \cdots \]  

(15)

Here \( V_1 \) is the one-body term, \( V_2 \) is the two-body term, \( V_3 \) is the three-body term, \( N \) is the number of atoms in the system, \( r_i, r_j \) and \( r_k \) are the positions of atoms \( i, j, r \) respectively.

The derivative of the total energy with respect to the positions of the atoms is taken to compute forces that are acting between the atoms. This means, to get the force on
atom \( i \) one should take the partial derivative with respect to the position of atom \( i \) (Equation 16).

\[
F_i = -\frac{\partial V_{\text{total}}}{\partial r_i} \quad (16)
\]

We implemented force calculations for our Lennard-Jones potential in two ways. The first method is the analytically calculated force. This is very easy for our Lennard-Jones system: since only pair-wise interactions are present, the force exerted on particle \( i \) by particle \( j \) can be calculated simply by differentiating the interatomic potential between the two particles w.r.t. \( r_{ij} \). The total force on particle \( i \) is then obtained by summing over all \( j \)-s.

For the algorithmic differentiation approach, we employ the adjoint mode of DCO. The forces are obtained as follows: the total potential energy of the system is calculated, its adjoint is seeded with the value 1 and the tape is interpreted to calculate its derivatives with respect to all the positions. The tangent mode is impractical here, since the total potential is, in general, a function \( 3N \) positions (where \( N \) is the number of particles). We know that the complexity of the tangent mode scales proportionally with the number of inputs, thus, it is much more expensive to calculate forces this way.

All analytical interatomic potentials are just approximations as there is no way to cast true interatomic interactions into analytical form. Approximate interatomic potentials come in many different varieties, with different physical motivations. The most important features of a physically realistic potential are a strong repulsion for small interatomic distances and a weak attraction at large interatomic distances. The most common and simple form of potential is the Lennard-Jones or 6-12 potential proposed by John Edward Lennard-Jones in 1924. It is a mathematically simple model that approximates the interaction between a pair of neutral atoms or molecules. It can be expressed as Equation 17.

\[
V_{\text{LJ}} = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^{6} \right] = \varepsilon \left[ \left( \frac{r_m}{r} \right)^{12} - 2 \left( \frac{r_m}{r} \right)^{6} \right] \quad (17)
\]

\( \varepsilon \) is the depth of the potential well
\( \sigma \) is the finite distance at which the inter-particle potential is zero
\( r \) is the distance between the particles
\( r_m \) is the distance at which the potential reaches its minimum
At \( r_m \), the potential function has the value \(-\varepsilon\)
The distances are related as \( r_m = 2^{\frac{1}{6}}\sigma \approx 1.122\sigma \)

These parameters can be adjusted to reproduce experimental data or approximate quantum mechanical calculations. The Lennard-Jones potential is used widely in computer simulations even though more accurate potentials exist because of its computational simplicity. Also, there are many existing calculations and simulation results for L-J potential, which makes L-J potential an attractive candidate (Figure 6).
Figure 6: Lennard Jones Interatomic Potential

SENSITIVITY ANALYSIS

The idea of a sensitivity analysis using automatic differentiation is simple: we seed the input parameters (sigma and epsilon for our Lennard-Jones system) with tangent values, run the code and harvest the Jacobian. The adjoint mode, which is technically the superior choice since the tangent mode has a computational cost proportional to the number of parameters, was unfortunately, not quite practical to implement. It was found that the size of the tape grew quite quickly with every time step and made impossibly large RAM demands.

This straightforward application to calculate sensitivities did not work for most cases. The molecular dynamics system is a chaotic one; this means it is exquisitely dependent on even the smallest changes in initial conditions or model parameters. For example, two trajectories with a tiny difference in their initial condition will see their trajectories diverge exponentially away from each other. This behavior is known as Lyapunov instability and can be stated mathematically as $\Delta r(t) \sim e^{\lambda t}$, where $\lambda$ is the Lyapunov exponent. Small perturbations in the force field parameters can, likewise, be amplified as the system evolves in time.

This has very real consequences for the sensitivity analysis: while the derived thermodynamics quantity from a molecular dynamics model might converge, the exact first-order derivative of that quantity w.r.t. a parameter won’t, if one has a long enough trajectory. Instead, it will blow up to incredibly large values, because in a long
trajectory the tiniest perturbations in a parameter can lead to a completely different trajectory and thus, a not-so-insignificant change in the output thermodynamic quantity. This phenomenon is clearly demonstrated in a parameter study that has been depicted graphically below (this particular simulation is two-dimensional and periodic, with number of particles $N = 100$, trajectory length $T = 100$) where one observes what seems like a smooth variation of a thermodynamic quantity if the plot points are spaced widely enough. But, if one were to “zoom in” on this curve, making smaller perturbations, it would be seen that the curve is indeed very spiky in nature: which means very large first derivatives are obtained.

![Figure 7: Variation of temperature with epsilon](image)

This presents a challenge: it is clear that what one really wants to know is the “average slope” of the curve in some sense, not the explosively large values of the exact derivative that arise due to the intrinsic spikiness of the thermodynamic quantity vs parameter curve. We made an interesting observation on the nature of the derivative: it is not strictly monotonically increasing as the trajectory evolves in time. Rather, it grows quickly in the beginning, stays nearly constant at a certain value and ultimately, at the certain value, the chaotic effects blow the derivative up (Figure 7).
We hypothesize that the stable value of the derivative could give us the averaged out sensitivity value that we seek. Since we find the derivative is stable in the range of 100 – 2200 time steps, we can divide the trajectory into chunks of 100 – 2200 time steps, obtain the derivative value for each of those chunks and then average over all the chunks to arrive at a so-called “chunk derivative”. As expected, the value of the chunk derivative is robust to the choice of chunk size, unless the chunk size is too small or too large. A small chunk size has the disadvantage that we neglect much of the time correlations (history dependence) that exist in the motions of particles. The velocity cross and auto-correlation functions take some time to decay (see figure 9 below) to small values, so the chosen chunk size should be larger than the time scale of that decay. Of course, it cannot be too large; otherwise the value of the derivatives explodes due to chaotic amplification.
Figure 9: Decay of the velocity auto-correlation function for the Lennard-Jones fluid

The value of the chunk size is chosen by studying the behavior of the derivative in the equilibrating portion of the trajectory that is not used for calculation of thermodynamic quantities, to find the range in which the derivative stays stable. As previously discussed, the choice does not affect the value of the derivative too greatly as long as one stays away from extreme choice, so a rough estimate serves us fine. The variation of the “chunk derivative” with the chunk size is shown in Figure 10.

Figure 10: Variation of the chunk derivative with the choice of chunk size

Whether this approach yielded reasonable results will be discussed in the next section.

RESULTS AND DISCUSSION

VALIDATION OF CODE

The first step in the validation of our code should be testing how well our numerical integrator conserves energy. In figure 11, the evolution of kinetic (blue), potential
energy (green) and total energy (red) during a 2D, periodic, 100 particle simulation with initial kinetic energy per particle $kT = 1$ is plotted.

![Figure 11: Changes in the energy during a simulation](image)

The fluctuations in the total energy give an idea of how close the code is simulating a truly NVE ensemble and are a good proxy quantity for the numerical truncation error in the Verlet integration. For a time step $dt = 0.0001$ in the 2D case, fluctuations of the energy are found to be satisfactorily small around 0.01% of the total energy. This is the value of $dt$ we have used throughout. It is also observed that due to the symplectic nature of the Verlet integrator, there is almost no drift in the value of the total energy over time.

The Maxwell-Boltzmann equation, which forms the basis of the kinetic theory of gases, defines the distribution of speeds for a system of particles in thermodynamic equilibrium, experiencing elastic collisions.

$$f(v) = \frac{1}{\sqrt{\pi}} \frac{m}{2\pi kT} \frac{3}{4} \frac{1}{\sqrt{2\pi kT}} e^{-\frac{mv^2}{2kT}}$$

We run a simulation with 36 particles in a 6 x 6 periodic box. The simulation was initiated with equal velocity magnitudes but in random directions, which means the distribution of speeds is initially concentrated at a single point. However, as the simulation progresses, it is observed that the distribution converges to Maxwell-Boltzmann distribution very quickly, as expected from theory. (Figure 12)
Figure 12: Comparison of the distribution of velocities for our simulation against the Maxwell-Boltzmann distribution

With the normal code now validated, we check the distribution for the code that uses automatic differentiation for force calculations and it is found to be identical. The temperature/pressure outputs of the two codes are also in close agreement.

**RUNTIME OF CODE FOR FORCE COMPUTATIONS**

<table>
<thead>
<tr>
<th>Timings for force calculation</th>
<th>Without AD</th>
<th>With Adjoint Mode AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Seconds]</td>
<td>23.10</td>
<td>47.35</td>
</tr>
</tbody>
</table>

Table 4: Timings for the MD code when force Calculations are performed with and without AD

It is observed that the adjoint code is 2.05 times slower than the hand coded derivative. This is not a surprise: since algorithmic differentiation introduces additional software dependencies, hand coded derivatives are always faster than AD. The Lennard-Jones potential has been used in this project which is a relatively simple function for which analytical derivatives can be easily coded. However, this may not be the case for more complex potential functions like multi-body Stilinger-Weber Potential for crystalline silicon or the CHARMM Potential for biomolecules that considers many interactions like bond vibrations, twisting, torsion, angular vibrations, electrostatic interactions, non-bonded interactions, etc. Sometimes, the functional form may not even be known: e.g., iterative solvers may be used to calculate the solvation energy in implicit solvation models like COSMO. Automatic differentiation could greatly ease the calculation of its gradients w.r.t. the positions of the atoms here. A scientist inventing a new potential model could also cut down greatly on developer time using this approach, since she would only need to evaluate the total potential energy, the rest would automatically be taken care of.

**SENSITIVITY RESULTS**
First, a simple sensitivity analysis is run with no chunking in a simulation where the particles start off in their equilibrium configuration, i.e., in a 2D dimensional hexagonal lattice and we provide the low initial kinetic energies. (boundary conditions: open, number of particles = 300, kT = 0.05)

![Particles initialized in a 2D Hexagonal Lattice](image)

This system is not truly chaotic; the particles merely execute small oscillations around the equilibrium position. Hence, the derivative obtained without chunking the trajectory does not explode, even when the trajectories are long. The derivatives obtained via finite difference to those that are obtained from algorithmic differentiation and find that they agree almost exactly!

<table>
<thead>
<tr>
<th>Sigma</th>
<th>0.99999</th>
<th>1.000</th>
<th>1.00001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>0.0281148</td>
<td>0.0280096</td>
<td>0.02790436</td>
</tr>
</tbody>
</table>

Table 5: Perturbation of sigma around 1.0 to calculate the derivative of the temperature with finite differences

The algorithmic differentiation code computes the derivative of the temperature w.r.t sigma to be -10.5115 whereas the derivative obtained via finite difference is -10.5114.

<table>
<thead>
<tr>
<th>Epsilon</th>
<th>0.99</th>
<th>1.00</th>
<th>1.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>0.0279459</td>
<td>0.0280096</td>
<td>0.02807327</td>
</tr>
</tbody>
</table>

Table 6: Perturbation of sigma around 1.0 to calculate the derivative of the temperature with finite differences
The algorithmic differentiation code computes the derivative of the temperature w.r.t. epsilon to be -10.5115 whereas the derivative obtained via finite difference is -10.5114. This gives us confidence in the correctness of our sensitivity analysis code.

Of course, one is usually not interested in running MD simulations for ultra-low temperatures, since the behavior of such systems is not interesting. For a moderately high temperature simulation, the derivative explodes due to chaos, so the trajectory must be chunked to obtain a reasonable value for the derivation. We perform this time a periodic 2D simulation with 100 particles in a 10 x 10 box, and an initial kinetic energy per particle kT = 1. In the finite difference approach, we make somewhat large perturbations to sigma and epsilon, since very small perturbations will lead to large derivatives due to the spikiness. We use a chunk size of 2000 and compare the chunk derivative with the slope obtained from the finite difference approach.

<table>
<thead>
<tr>
<th>Sigma</th>
<th>0.99</th>
<th>1.00</th>
<th>1.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>1.323</td>
<td>1.464</td>
<td>1.589</td>
</tr>
</tbody>
</table>

Table 7: Perturbation of sigma around 1.0 to calculate the derivative of the temperature with finite differences

The perturbation approach gives us a slope of 13.3 whereas the chunk derivative w.r.t. sigma is evaluated to be 10.8.

<table>
<thead>
<tr>
<th>Epsilon</th>
<th>0.7</th>
<th>1.00</th>
<th>1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>1.1766</td>
<td>1.464</td>
<td>1.7209</td>
</tr>
</tbody>
</table>

Table 8: Perturbation of epsilon around 1.0 to calculate the derivative of the temperature with finite differences

Here, the perturbation approach gives us a slope of 0.9 whereas the chunk derivative w.r.t. sigma is evaluated to be 0.73. The chunk derivative seems to underestimate the “average slope” by about 30% in both cases but the two values seem to be in rough agreement. A discrepancy is perhaps only to be expected, since the “chunking” of the trajectory assumes that these individual pieces of trajectory are independent of each other and therefore, neglects their history-dependence. Different applications require different accuracies and precision for the sensitivity information and it is left up to the user to decide whether the “chunk derivative” is good enough for his purposes.
CONCLUSIONS

In the current project, we have implemented 2D and 3D Lennard-Jones systems of particles in C++ and written a MATLAB code to visualize our results. Automatic differentiation has been successfully utilized for calculating forces and the sensitivities of thermodynamic quantities with respect to model parameters. Automatic differentiation has not received much attention in the molecular simulation literature so far. This pilot project shows that its applications to MD could be a promising and fruitful area of investigation. Future work could focus on calculating force with AD for more complicated potential functions. The “chunk derivative” approach needs to be justified mathematically and tested more intensively as well.

BIBLIOGRAPHY

LITERATURE


EXTERNAL LINKS

- Atoms in motion MD used in figures (http://atomsinmotion.com/book/chapter5/md)
- Verlet Integration Demo and Code as a Java Applet (http://verlet.googlecode.com/)
- The Verlet algorithm (http://www.fisica.uniud.it/~ercolessi/md/md/node21.html)