Small Assignment 10/30 - Code Validation

Skye Rhomberg, Will Solow, and Eric Aaron*

Department of Computer Science, Colby College, Waterville, ME

E-mail: sorhom22@colby.edu

Phone: +123 (0)123 4445556

Abstract

We[1] give an overview of the Diffusion Monte Carlo (DMC) algorithm and its applications into finding solutions to the Schrodinger Equation when no analytical solution is available. An implementation of the DMC algorithm is presented with a specific focus on understanding the behaviour of Clathrate Hydrates—a complex, multi-atomic structure often made up of water molecules and a greenhouse gas. Given the many ways of representing such a system, we present conditions for which different representations produce the computationally efficient results. We show how Importance Sampling can be used to reduce some of the computational inefficiency that abounds when working in high dimensional data sets with results that mirror those of a straightforward DMC implementation. Finally, we provide an analysis of the above algorithm when implemented in a Object Oriented versus script-based format to conclude that script-based programming is more computationally efficient in this scope.

Introduction

We introduce an efficient, generalizable, and validated implementation of the Diffusion Monte Carlo (DMC) algorithm utilizing NumPy. We discuss the problems faced in the world of

Computational Sciences in writing validated, efficient, and widely usable code and how our implementation of the DMC algorithm furthers the work in these areas. We present efficiency comparisons between an Object-Oriented approach and a scripting approach, and discuss the pros and cons of each implementation in the context of verifiability, efficiency and generalizability.

The Shrödinger equation is nearly impossible to solve for complicated molecular systems. To date, the DMC algorithm is one of the best proposed algorithms for creating a tight approximation of solutions to the Schrödinger equation for complex systems. Given the complexity of the DMC algorithm and its potential for time growth in large systems, studying faster implementations of it is broadly applicable for future research; in computational studies, time barriers often prevent work from being done as computers are required to analyze a large amount of possible simulations.

However, a fast implementation is not valuable unless it is correct. We provide techniques through which we validated our work and hope that these techniques and practices can be apllied broadly to future work done in Computational Science.

Related Work

This semester, we code a python based implementation of the Diffusion Monte Carlo algorithm originally presented in Anderson 1975.[1] Our implementation of the DMC algorithm can be directly attributed to Anderson's work, as he was one of the first to propose such a method. At the time, other methods were seen as more computationally efficient for a given level of accuracy. We take Anderson's algorithm and refine it in the scope of being efficient for computing the ground level energy and bond strength of Clathrate Hydrates.

While Anderson was initially only concerned with the calculation of the ground energy of H3, his simplifying approach to the 1D and 6D systems greatly influenced our approach in initial versions of our implementation. We improve on his work by designing a more

efficient, vector-based implementation using the numpy library provided by python. We hope to extend this approach to the task of modelling more complicated Clathrate Hydrates which require an increased computational load. As Anderson points out, the Schrödinger Equation is only analytically solvable for certain, small systems. His DMC algorithm provides such a way to approximate solutions to complex systems. The best solution to modelling Clathrate Hydrates, given the lack of an analytical solution, is an open question. We hope to build on Anderson's work to provide a computationally efficient method for the solution of this problem.

Modeling

The Model

The DMC model is not novel. It[1] was first introduced over 40 years ago and has been refined since. At a high level, the model takes an array of walkers, each representing a possible state of the system, and propagates the atoms in each walker. After propagation, the potential energy of each walker is calculated. Walkers with potential energy that is too high are deleted, while walkers with reasonable potential energy are replicated. Given the pseudo-randomness leveraged in code-based implementations of the DMC algorithm, this serves to approximate every possible configuration of the system, within the bounds in which it would normally exist in nature.

Over time, the rolling average of the potential energy of all systems that are valid converges to the zero point energy, or the ground state energy of the system. This produces an approximate solution of the Schrödinger equation, which is what the DMC algorithm is modelling.

Implementation

In our implementation of DMC, we write a script-based implementation in Python, relying heavily on the NumPy library, which provides "vectorization" for fast operations on large matrices—that is, it allows processors to perform concurrent operations while looping over matrices, greatly reducing runtime compared to conventional looping. The most important variable in the code is a 4D array which stores the Cartesian coordinates of each atom in each molecule in each walker. As we move through the simulation loop of the algorithm, we operate on the walker array to delete and replicate walkers as required by the algorithm.

Notably, the function that calculates potential energy is drastically different from molecular system to molecular system, so the main simulation loop uses calls to the potential energy function for the purpose of extensibility. From system to system, the only other varying part of our code is how we propagate each walker. The distribution of possible propagations for various atoms depends upon their mass, and so subroutine to propogate a particular walker is contingent upon the particular order of atoms therein. This could potentially change with each system, although we hope to find a data structure that allows our code to remain efficient without needing to change how the walkers are propagated at each step. We find some promise in generating the array of propogations—i.e. how far each atom moves in each coordinate—wholesale by stacking together many copies of a list of the walker's atomic masses up to the dimensionality of our 4D array. Otherwise, the entire model is controlled by the initial variables of the simulation.

Through the simulation loop, we rely heavily on the broadcasting features of NumPy to different array dimensions to figure out which walkers are to be deleted or replicated. We like to think that this is done quite cleanly, and do it in a mere 11 lines of code after the walkers are propagated. This efficiency is appealing and has been why we are hesitant to walk away from the 4D array based implementation that we provide. The main disadvantage of this method becomes apparent when considering how to model complicated molecular structures like Clathrate Hydrates, which don't have a consistent number of atoms per molecule as

they exhibit heterogeneity of molecules within a walker. This means that one of the four dimensions of our array is "ragged." There are a few ways to work this raggedness into our data structure, but they are all patches rather than ideal solutions.

Code Validation

In any implementation of an algorithm, rigorous testing is required to ensure that the code is producing accurate results. Ideally, data generated from a simulation would be corroborated with data collected experimentally. However, due to the nature of the study of particles, outside of the simplest systems, it is difficult to accurately calculate the ground state energy. As such, results generated from the DMC algorithm cannot be verified against real world data.

Instead, we turn to manual validation of the code. An outline of the process is as follows: a set of walkers is generated. The validator calculates the potential energy and the reference energy. Using a randomly generated set of thresholds, the validator determines which walkers should be deleted and replicated, and then compares the final array to the final array produced by the algorithm. To be sure of correctness, this process was repeated multiple times with sets of ten walkers over five time steps. These groups of calculations should be comprehensive enough to validate the correctness of the algorithm due to the stochastic nature of the simulation, and generalize well to larger groups of walkers.

As a final measure, the implementaion of the DMC algorithm was tested to model the carbon monoxide bond, given that the Zero-Point energy of the CO bond is well known. When tested over multiple simulations, the 1,000 step rolling average varied on the order of $1 \cdot 10^{-4}$, with a variance on the order of $1 \cdot 10^{-3}$. Based on prior DMC implementations, this amount of inaccuracy is normal.

Simulations

Discussion

When creating a piece of code in the field of Computational Science, its worth is often based on how extendable it is to other projects of interest. In the case of our work, we provide a fast, script-based implementation of the DMC algorithm by leveraging the vectorization provided in the NumPy library. Currently, our work relies on the assumption that the molecules in the modeled system are homogeneous. Clearly, this is not a particularly extendable piece of code, as there are many other interesting systems made up of hetereogeneous molecules.

This begs us to consider how to create generalizable code that can be used in a variety of circumstances without sacrificing efficiency. Observe that NumPy works well when dealing with arrays of many dimensions. As soon as we change the sizes of the molecules in a walker, the dimensions of our array are no longer even, and so NumPy cannot cleanly vectorize the solution in the 4D data structure that we have presented up to this point. From a programming point of view, it would not be particularly difficult to represent each walker on a singular array dimension. However, this also reduces the extendablility of our code as it makes the potential energy function extraordinarily difficult to calculate, given that each molecule and atom corresponds to a particular index on the same array dimesion. It also makes initializing the walker array cumbersome and unintuitive, as the user would have to memorize a potentially very long string of atoms and know their indices to facilitate the potential energy calculations both within and between molecules in a particular walker.

We hope that the users of our code are domain experts, and want to create a scientific instrument that is easy and intuitive for them to use. As such, we continue to try and refine our ideas to create an implementation of DMC that balances both efficiency and ease of use.

Conclusion

Acknowledgement

The authors thank Professor Lindsey Madison of Colby College for her time and expertise.

References

(1) Anderson, J. B. A random-walk simulation of the Schrödinger equation: H+3. *The Journal of Chemical Physics* **1975**, *63*, 1499–1503.