Quantum Monte Carlo: Faster, More Reliable, And More Accurate

Thesis by

Amos Gerald Anderson

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2010

(Defended June 29, 2009)

To my beloved grandmother

Sylvia Kay Anderson, RN

1928 - 2008

From her pikkupoika

Acknowledgments

I would like to thank the many people who helped make my graduate studies as pleasant as possible. First of all, I would like to thank Dan Fisher for sharing in the burden of helping to make QMcBeaver a useful tool. The two of us have worked together to develop the code to the state it is in today, and I don't think it would have been possible for us to get any results without each other's efforts. We came near to throwing in the towel on QMcBeaver, but now I think we have software we can be proud of.

The other members of the Goddard group have also been quite helpful and encouraging. Although they were gone by the time I arrived, both Mike Feldmann and Chip Kent have been continual sources of advice, which was quite helpful since they were the ones who started the QMcBeaver project. The foundation and organization they provided in the source code made further development not only possible, but enjoyable. Mario Blanco has been a joy to work with, since his enthusiasm for the science itself and the bigger picture is contagious. I have enjoyed many interesting conversations with Andrés Jaramillo-Botero here and in Colombia, and he always had encouraging things to say. My work has been supported by the tireless efforts of Darryl Willick, under whose watch our computers were always working. It is also important to recognize all the group members who over the years have helped to make the Goddard group a place to watch exciting new science come to fruition.

For the first several years of my studies, I had the privilege of working with Professor Peter Schröder learning to do chemistry on GPUs. Most importantly, though, he taught me how to write a paper. I also had the opportunity to work with Professor Jack Roberts, doing something practical with the knowledge I'd learned. Having TA'd many classes under Professor Aron Kuppermann, I have been prepared to teach my own classes some day.

Of course the most important member of the group is my advisor, Bill Goddard. Through the years, he has been a fountain of crazy ideas and reality checks, some of which

worked. I am privileged to have had the opportunity to work with him over these years and to learn Quantum Chemistry from the ground up. I know I have learned something because our conversations have advanced from deer in the headlights to debates.

I am grateful for the remarkable group of friends that I have found here at Caltech. I learned quite a bit during my time in Avery House from the other students and the faculty who lived there. The variety of perspectives opened the door for many of the most interesting conversations of my life. The good people of Caltech Christian Fellowship and especially of Trinity Baptist Church provided me with a far more important source of support than merely meals, housing, entertainment, and vacation destinations; keeping an eternal perspective during frustration. I have found a unique and special community here in Pasadena.

Finally, I would like to thank my family for their love, encouragement, and prayers through the years in helping me to make it this far. During my time at Caltech, my sister Micah has been dutifully serving our country in Iraq, helping to bring peace to that region, and to defend the freedoms necessary for intellectual pursuits. I have been deeply inspired by my parents, Jeff and Mary Ann, and their steadfast dedication among the urban poor of Manila, Philippines, since 1985 and still going strong. They have taught me that even in the face of overwhelming challenges, with God's help it is still possible to make a difference.

Maraming salamat at pagpalain kayong lahat ng Diyos!

Abstract

The Schrödinger Equation has been available for about 83 years, but today, we still strain to apply it accurately to molecules of interest. The difficulty is not theoretical in nature, but practical, since we're held back by lack of sufficient computing power. Consequently, effort is applied to find acceptable approximations to facilitate real time solutions. In the meantime, computer technology has begun rapidly advancing and changing the way we think about efficient algorithms. For those who can reorganize their formulas to take advantage of these changes and thereby lift some approximations, incredible new opportunities await.

Over the last decade, we've seen the emergence of a new kind of computer processor, the graphics card. Designed to accelerate computer games by optimizing quantity instead of quality in processor, they have become of sufficient quality to be useful to some scientists. In this thesis, we explore the first known use of a graphics card to computational chemistry by rewriting our Quantum Monte Carlo software into the requisite "data parallel" formalism. We find that notwithstanding precision considerations, we are able to speed up our software by about a factor of 6.

The success of a Quantum Monte Carlo calculation depends on more than just processing power. It also requires the scientist to carefully design the trial wavefunction used to guide simulated electrons. We have studied the use of Generalized Valence Bond wavefunctions to simply, and yet effectively, capture the essential static correlation in atoms and molecules. Furthermore, we have developed significantly improved two particle correlation functions, designed with both flexibility and simplicity considerations, representing an effective and reliable way to add the necessary dynamic correlation. Lastly, we present our method for stabilizing the statistical nature of the calculation, by manipulating configuration weights, thus facilitating efficient and robust calculations.

Our combination of Generalized Valence Bond wavefunctions, improved correlation functions, and stabilized weighting techniques for calculations run on graphics cards, represents

a new way for using Quantum Monte Carlo to study arbitrarily sized molecules.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Although the laws governing the behavior of electrons have been understood for 80 years, progress has been dictated by the advance of computer technology. It is not as though we do not understand the chemical concepts involved; the problem is that sufficient accuracy in the computation depends on minutia which scale with the size of the molecule itself. A chemist is presented with a menagerie of *ab initio* and empirical tools exploiting various tradeoffs between computational expense and accuracy. Several sweet spots have been already been found, ending much of the search. Unfortunately, for those who seek high accuracy, no method has distanced itself from the others. Quantum Monte Carlo (QMC) will become the winner because of several significant advantages it has over its competitors.

1. Its theoretical scaling is a mere $\mathcal{O}(N^3)$, which comes from matrix multiplication and inversion. This is far better than the $\mathcal{O}(N^7)$ to $\mathcal{O}(N!)$ of Coupled Cluster or Full Configuration Interaction techniques. This means that with faster computers, eventually, QMC will be the fastest method.

2. QMC is very easily parallelizable, meaning that if you give it twice as many computers, it can complete its task in nearly half the time. Because of this, calculations using 1000s of computers to complete a QMC calculation is becoming routine. In contrast, other methods, which require the transfer of large amounts of data between processors, can not effectively use more than a handful of processors.

3. Computers are exponentially getting faster, but the amount of memory they have is not rising nearly as fast. QMC requires very little memory, on the order of 10s of megabytes. Other high accuracy methods require gigabytes of memory, a requirement

that scales quite quickly with size, and is their limiting factor in terms of what is possible.

These reasons alone are sufficient to guarantee that QMC will, eventually, be the winner. There are two primary obstacles, computational and theoretical, and we address both of these issues in this thesis. We will show that we can surmount these, paving the way for QMC adoption in the chemistry world.

Computing power is advancing quite rapidly, and will probably continue to do so, a factor which favors QMC approaches over any other. This means that, essentially, we only need to wait in order to win. However, the argument is more subtle than this because of two competing factors. At some point, processors will reach the physical limits of the medium used to carry out the computations, and if no better media is found, then this will signal the end of the road. On the other hand, we can see the rise of new types of computing devices in which several small processors are joined together to accomplish one task. The best known example of these devices is a graphical processing unit (GPU), typically used to accelerate computer games. Exploiting a tradeoff between general computing and specialized computing, GPUs are becoming exponentially faster than CPUs. We were the first to study the possibility of running quantum chemistry software on a GPU, as we discuss in Chapter 3. Even though our 2006 technology has already become obsolete, we were able to run our software at least 6 times faster than a CPU of the same era. Were we to revisit this problem and update our software, it is entirely reasonable that speedups on the order of 100 times faster is possible.

The second issue is theoretical. As we will discuss, while introducing QMC in Chapter 2, a QMC approach is only as accurate as the position of the nodes in the provided wavefunction, introducing a new kind of error, the fixed-node energy. Although there is ongoing research into QMC techniques for optimizing the wavefunction nodes, these necessarily require more computational effort. In our studies however, presented in Chapter 4, we have found that many problems are quite tractable given judicious choice of wavefunction. In particular, Generalized Valence Bond (GVB) wavefunctions can eliminate enough of the fixed-node energy, for both bond breaking and electronic excitation processes, that we can easily obtain accuracy on the order of a few tenths of a kcal/mol. The particular advantage of a GVB wavefunction over more general types of wavefunctions is that GVB scales in

expense quite well with molecule size. It is very modular, allowing one to describe localized regions of chemical activity. With this simple approach to lowering the fixed-node energy, combined with QMC's particular ability to measure all the dynamic correlation in molecule, the two methods are highly complementary.

In Chapter 5, we study several molecules to find out how well the approach works beyond simple hydrocarbons, as well as discuss a few of the most important issues in QMC. In particular, we find that although our QMC-GVB approach fails to describe molecules such as the atomization of CO correctly, adding in Restricted Configuration Interaction (RCI) terms brings us to agreement with the experimental results. However, we also show where even the QMC-RCI approach is insufficient, with for example, the atomization of the CN molecule. Finally, we show that the even more expensive complete active space self-consistent field (CASSCF) wavefunctions are sufficient to study even the difficult ozone electronic excitation.

Even granted the claims we make in this thesis, we will probably never see QMC directly used to study large systems evolve with time. QMC will be used, however, to calculate energy reaction barriers and enthalpies, which can be used to fit force field parameters. Once we have accurate data, we can turn to other methods and model a system in time. We have studied one such method called Kinetic Monte Carlo (KMC), which takes reaction enthalpies and simulates a system for time scales as long as seconds, depending on the system. In Chapter 6, we present an $\mathcal{O}(\log N)$ algorithm we developed for doing so.

Chapter 2

Background

Quantum Monte Carlo (QMC) [1, 2, 3] takes a different approach to solving the Schrödinger equation than the other quantum chemistry methods. Most methods directly minimize the energy of an analytically integrable wavefunction using the variational principle. Unfortunately, the requirement that the wavefunction be analytically integrable is somewhat restrictive, and in particular, it is difficult to use functions of interparticle coordinates. Starting from a wavefunction obtained using some other method, we can obtain a probability density. Because QMC uses Monte Carlo integration over the probability density, we can ease these restrictions by patching up the wavefunction as we like. All QMC requires of the wavefunction is that it be easily differentiable so that we can apply the Hamiltonian, a far simpler criteria to satisfy.

But QMC can do even better than this. If we are prepared to take the time to do a Monte Carlo integration, then there is a simple reformulation of the Schrödinger equation that can permit us a far more accurate calculation than the probability density itself. This reformulation, called Diffusion Quantum Monte Carlo, or sometimes just Diffusion Monte Carlo (DMC) is the foundation of this thesis. Because we consider QMC to be essentially worthless without the DMC modifications, we will sometimes consider the QMC and DMC labels to be synonymous.

A DMC calculation can extract *all* of the dynamic correlation from a probability density, a truly remarkable feature. However, because the statistical error depends upon the quality of the probability density, we are still motivated to obtain the best probability density that we can, in order to minimize the number of statistical data points necessary to reach a specified error margin. This is not enough, though, because the final desired accuracy of a DMC calculation will depend on the quality of the underlying wavefunction nodes; a

systematic error called the fixed-node energy. This error is the focus of subsequent chapters.

2.1 Wavefunctions

According to the postulates of Quantum Mechanics (QM), all matter can be described with a wavefunction, Ψ. The exact wavefunction contains all the data necessary to measure observable properties. Another postulate of QM is that the wavefunction is the probability amplitude, by which the probability of the particle being in a volume element dr around a particular location r can be calculated as

$$\rho(r)dr = |\Psi(r)|^2 dr \tag{2.1}$$

which takes into account the possibility that the wavefunction might be complex valued. In order to find the wavefunction, we must solve the Schrödinger eigenvector equation

$$i\hbar\frac{\partial}{\partial t}\Psi(r,t) = \hat{H}\Psi(r,t) \tag{2.2}$$

or its time-independent analog

$$\hat{H}\Psi(r) = E\Psi(r) \tag{2.3}$$

where the Hamiltonian operator $\hat{H}$ for a molecule with motionless nuclei is

$$\hat{H} = -\frac{1}{2}\sum_{i}^{N}\nabla_i^2 + \sum_{i>j}^{N}\frac{1}{r_{ij}} - \sum_{a}^{N_{nuc}}\sum_{i}^{N}\frac{Z_a}{R_{ai}} \tag{2.4}$$

$$= -\frac{1}{2}\sum_{i}^{N}\nabla_i^2 + V(r), \tag{2.5}$$

where N is the number of electrons and N_{nuc} is the number of nuclei, Z_a is the charge on nucleus a, r_{ij} is the distance between electrons i and j, and R_{ai} is the distance between electron i and nucleus a. Although the potential energy term does depend on the positions of all the electrons and all the nuclei, we only imply this dependency on the position of the nuclei in our notation $V(r)$.

2.2 Antisymmetry

If quantum chemistry was merely the description of an n-body problem and all we had to do was to solve the time-independent Schrödinger Equation 2.3, the problem would be only $\mathcal{O}(N^2)$ hard, since each particle would interact with every other particle in a potential field. However, because electrons are fermions, the solutions are more complicated. Nature dictates that fermion wavefunctions are constrained to be antisymmetric, which says that swapping any two electrons in a wavefunction must produce the negative of the wavefunction. This is the Pauli Antisymmetry Principle:

$$\Psi(..., \boldsymbol{r}_i, \boldsymbol{r}_j, ...) = -\Psi(..., \boldsymbol{r}_j, \boldsymbol{r}_i, ...). \tag{2.6}$$

This constraint raises the complexity of the problem to at least $\mathcal{O}(N^3)$, since we are now required to use the antisymmeterization operator; the determinant. Thus the best scaling any algorithm can achieve is $\mathcal{O}(N^3)$.

Antisymmetry means that within a wavefunction, there will be some regions where $\Psi(r) = 0$, which we call the nodes. By this we do not mean that no electron can ever go somewhere; we mean that given locations of $N-1$ electrons, there are certain places the N^{th} electron can not go. Of course those forbidden regions might become accessible just as soon as one of the other electrons move. What do these nodes look like? The obvious region forbidden by the Pauli principle is where any two electrons coalesce, because

$$\Psi(..., \boldsymbol{r}_i, \boldsymbol{r}_i, ...) = -\Psi(..., \boldsymbol{r}_i, \boldsymbol{r}_i, ...) = 0. \tag{2.7}$$

Unfortunately, however, the nodal region is higher dimensional than this. This is evident when considering the following thought experiment. Consider two (same spin) electrons at different positions near a nucleus (for example a triplet state of Helium), and we write down the value of the wavefunction. Due to symmetry, we can write down the wavefunction as a function of three coordinates: $\Psi(R_1, R_2, r_{12})$. It is entirely possible to swap the positions of these two electrons in such a way that they never meet. Once we have moved them to each other's initial position, Equation 2.6 says the wavefunction will now be exactly the negative

of what we wrote down, since

$$\Psi(R_1, R_2, r_{12}) = -\Psi(R_2, R_1, r_{12}) \tag{2.8}$$

which means that somewhere along any path the wavefunction went to zero. In this case, we can infer that the node is wherever $R_1 = R_2$. One interesting observation is that the nodal structure is more simple than the wavefunction itself, which is not analytically known for even Helium. We know the analytical nodal structure of very few systems.

Electrons come in two flavors of spin which we label as α and β, a distinction derived from relativity, and accordingly the wavefunction is the product of a spatial and a spin function. Any pair of electrons can be said to be parallel spin if they are the same flavor, or opposite spin if not. The antisymmetry condition can be satisfied by either the spatial or the spin function. For example, opposite spin electrons can be given the spin function $\chi(r_1, r_2) = \alpha(r_1)\beta(r_2) - \beta(r_1)\alpha(r_2)$, so that swapping them results in $\alpha(r_2)\beta(r_1) - \beta(r_2)\alpha(r_1) = -\alpha(r_1)\beta(r_2) + \beta(r_1)\alpha(r_2) = -\chi(r_1, r_2)$. For parallel spin electrons we correspondingly assign them a symmetric spin function such as $\chi(r_1, r_2) = \alpha(r_1)\alpha(r_2)$, and apply spatial antisymmetry, as discussed in the next section. It is a violation of the antisymmetry restriction to apply to a pair of electrons an antisymmetric spatial function and an antisymmetric spin function, since the product of two antisymmetric functions is symmetric.

For a given wavefunction we can try to visualize nodes, as we for do for H_2O in Figure 2.1. To draw this image, we ran a QMC simulation for a few thousand iterations, to make sure the electrons are all in somewhat higher probability regions, and then we stop the simulation. We select one electron for a 3D scan over the volume of the molecule, writing to a file the value of the wavefunction at each coordinate. Using good plotting software, we can generate a surface at the contour level of 0. It is interesting to notice that the same nodal plane passes through all the same spin electrons, indicating that we can approach parallel spin electrons over only 2π steradians, which is half the total solid angle.

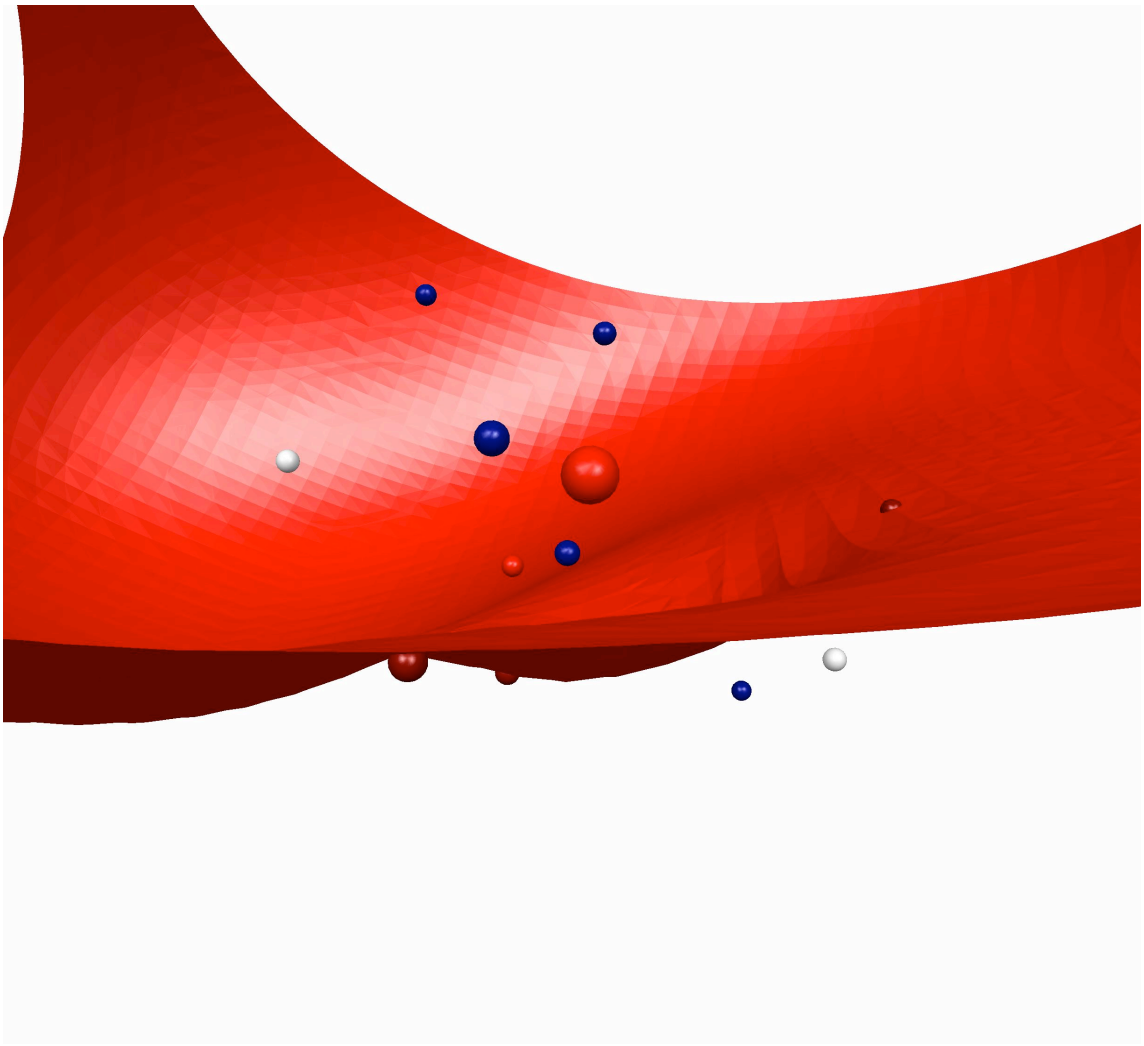

Figure 2.1: Nodal plane in H_2O. The Oxygen atom is the larger sphere in the center, and the 2 Hydrogens are the smaller white spheres. The small red sphere, located below and to the left of the Oxygen is the initial position of our test particle, and the darker red spheres are the other electrons of the same spin. The blue spheres are the electrons of the opposite spin. Notice that the sheet passes through all 3 of the dark red spheres. This image was generated with the help of MacMolPlt.

2.3 Quantum Chemistry

Traditional quantum chemistry programs seek to solve Equation 2.3 analytically. This is done by choosing an approximate form for the wavefunction. We start with atomic orbitals, called basis functions, which are similar to the orbitals of a Hydrogen atom

$$\chi_j(r \leftarrow r_i - R_j) = r_x^{k_j} r_y^{l_j} r_z^{m_j} \sum_n a_{jn} e^{-b_{jn}|r|^2} \tag{2.9}$$

which are simply gaussian functions of the distance of electron i at r_i to the center of the basis function j at R_j, and all the other parameters are fit to model a typical atomic orbital. This functional form for basis functions was motivated by the consideration that the product of two 3D gaussian functions is another gaussian function, a nonnegotiable benefit in computational efficiency for most quantum chemistry methods. In practice, sets of basis functions have been standardized for each element, so that they are independent of any molecule. Standardization is very difficult for functions of the form e^{-r}, which is another reason not to use them, even though they are closer to the Hydrogenic solutions.

An orbital $\phi_k(r_i)$ will typically span across multiple nuclei, meaning that it will necessarily be a linear combination of basis functions

$$\phi_k(r_i) = \sum_j \chi_j(r_i) c_{jk} \tag{2.10}$$

which is sometimes referred to as a molecular orbital, and there will be as many linearly independent orbitals possible as there are basis functions. We can pick the best of these orbitals for the electrons to occupy. For opposite spin electrons we use spin functions to satisfy antisymmetry, meaning that up to two opposite spin electrons can occupy the same orbital. For parallel spin electrons there is only one way to guarantee spatial antisymmetry no matter what the orbitals or basis functions look like. This is to put them into what is called a Slater determinant

$$D = \begin{vmatrix} \phi_1(r_1) & \phi_2(r_1) & \cdots & \phi_N(r_1) \\ \phi_1(r_2) & \phi_2(r_2) & & \\ \vdots & & \ddots & \\ \phi_1(r_N) & & & \phi_N(r_N) \end{vmatrix} \tag{2.11}$$

which represents one orbital for each electron, letting each electron "visit" all the orbitals. This means that if two of these electrons swap places, corresponding to swapping rows, then the determinant will change sign, satisfying the Pauli antisymmetry principle. It is clear that by this construction, there is zero probability that two parallel spin electrons will occupy the same location, or that two parallel electrons can share the same orbital. Putting all the electrons of our molecule into a wavefunction of this type, we can obtain the orbital coefficients c_{jk} by minimizing the energy E self-consistently, which will be our solution to Equation 2.3.

The principle failure of Self-Consistent Field (SCF) methods is that they do not account for all electron-electron interactions. The difference between the energy produced by a SCF method and exact energy is referred to as the correlation energy. Correlation energy can be subdivided into two components; static correlation and dynamic correlation. Static correlation is the error resulting from optimizing an incomplete functional form for the wavefunction during the SCF procedure, and is typically resolved by increasing the complexity of the wavefunction by adding more orbitals and basis functions to the SCF optimization. Dynamic correlation comes from the SCF procedure itself, where an electron sees only an average field of the other electrons, and thus never has to move out of another's way. This is especially critical for a doubly occupied orbital, since those electrons share the same space.

Either of these errors can be minimized in one of two ways. First, remember that when we took linear combinations of the basis functions to make our orbitals, we actually received more orbitals than we needed. Although we put our electrons into the best orbitals, we still have quite a few unoccupied, or virtual, orbitals that we might want our electrons to be able to visit. Even though they were not necessarily the best, they might still be pretty good. For example, where degenerate orbitals play a role, even the ordering of the electronic states might be wrong. In fact, some virtual orbitals might have a negative orbital energy, meaning that an additional electron would be able to bind to the molecule. An electron as a quantum particle will need to visit all of these orbitals. To do this, we add to our wavefunction more determinants. For determinants that use N_{occ} occupied orbitals, if we have N_{virt} unoccupied orbitals, then there are

$$\binom{N_{occ} + N_{virt}}{N_{occ}} \tag{2.12}$$

possible determinants we can make. If we include all the possibilities, then this represents a Full Configuration Interaction (Full CI) calculation. By virtue of spanning the entire Hilbert space, a Full CI wavefunction is by definition the exact wavefunction, if we also use an infinite number of basis functions. Unfortunately, the convergence of the energy in the limit of adding more determinants is very slow, so this approach is impossible in practice.

But there is a second way to minimize the error. Instead of simply adding more determinants to our wavefunction, we can instead think about adding only the most important virtual orbitals, and then reoptimizing our c_{jk} coefficients. For this procedure, called multi-configuration SCF (MCSCF), we use our chemical intuition to identify which orbitals are likely to have the most error relevant to the system we are studying, and we figure out which corresponding orbitals would be the best to correct this error. For example, a bonding orbital is often too evenly balanced between the nuclei, so we might add the antibonding orbital in order to add some "left-right" correlation, permitting the two electrons in the bond to get away from each other a little bit. The set of orbitals that are chosen to need the most correction along with the orbitals used to add the correction is called the active space. This technique is quite effective at lowering the error due to static correlation because typically, there are only a few important virtual orbitals. We could use these improved orbitals in a CI treatment, improving convergence. We will further discuss MCSCF in the context of Generalized Valence Bond (GVB) wavefunctions.

2.4 Variational Monte Carlo

Assuming that the wavefunction is normalized and real-valued, we can rearrange terms in the Schrödinger Equation 2.3 to get

$$\begin{aligned}
\langle E \rangle &= \langle \Psi | \hat{H} | \Psi \rangle && (2.13)\\
&= \int \Psi(r) \hat{H}(r) \Psi(r) dr && (2.14)\\
&= \int \Psi^2(r) \frac{1}{\Psi(r)} \hat{H}(r) \Psi(r) dr && (2.15)\\
&= \int \rho(r) E_L(r) dr, && (2.16)
\end{aligned}$$

where we have defined the local energy as

$$E_L(r) = \frac{\hat{H}(r)\Psi(r)}{\Psi(r)} = -\frac{1}{2}\sum_i^N \frac{\nabla_i^2\Psi(r)}{\Psi(r)} + V(r) \tag{2.17}$$

in order to calculate the expectation value of the energy $\langle E \rangle$. Seen in this formulation, all we need to do is sample the local energy according to the probability density enough times and we will eventually converge to $\langle E \rangle$. This is the Variational Monte Carlo (VMC) method. We define a walker to represent one electronic configuration, which will be moved around the molecule according to the Metropolis algorithm, which ensures that our sampling reproduces $\rho(r)$. Once we choose the number of walkers we want to use, N_w, our method is essentially

$$\begin{aligned}
\langle E \rangle &= \int \rho(r) E_L(r) dr & (2.18)\\
&\simeq \frac{1}{N_t}\sum_{t=1}^{N_t}\left\langle \frac{1}{N_w}\sum_{i=1}^{N_w} E_L(r_{t,i})\right\rangle_{A(r\to r')} + \mathcal{O}\left(\frac{\sigma}{\sqrt{N_t}}\right) & (2.19)\\
& A(r \to r') = \min\left[1, \frac{T(r \leftarrow r')}{T(r \to r')}\frac{\Psi_T^2(r')}{\Psi_T^2(r)}\right], & (2.20)
\end{aligned}$$

where N_t is the number of iterations we take and σ is the standard deviation of each sample. In this equation, $A(r \to r')$ is the acceptance probability which is used to decide whether a walker should move from coordinates r to some trial coordinates r' that iteration. The acceptance probability is designed to satisfy detailed balance, which ensures that on average the distribution of our samples is stationary and reversible. To do this, we need to be able to calculate the transition rate of moving from initial to final coordinates $T(r \to r')$, and the rate of going in reverse. The functional form of $T(r \to r')$ depends on the algorithm used to move electrons, which is merely an efficiency issue.

2.4.1 Error Margins

As Equation 2.19 indicates, the error margins of a VMC calculation go down as $\sqrt{N_t}$. Said another way, if you want to lower your statistical error by a third, you will need to run about 10 times as many iterations. The expected number of required iterations rises exponentially. This requires us to choose a wavefunction that will give us a lower sample error σ. If we are using wavefunctions of the type we described so far, then our immediate choices are to

increase the number of basis functions used, or to use a larger active space, as discussed in Section 2.3.

But we can do even better than that because the computational considerations required for the evaluation of the local energy, which is discussed in detail in Appendix B, are quite different than those of other quantum chemistry algorithms. Specifically, we can now add functions of interelectron coordinates to our wavefunction, which is a significant improvement over electrons only being able to see an average field of the other electrons. These functions, which we will call Jastrows, can now help electrons to avoid each other, beyond the repulsion established by the antisymmetry principle.

2.4.2 Cusp Conditions

Although we are unable to analytically solve for realistic wavefunctions, there are some things that we can say, analytically, about how the wavefunction should behave in some circumstances. The antisymmetry principle is one example of this, but we also know what the wavefunction should look like in the limit that two particles coalesce, since in that limit, the wavefunction is dominated by terms involving only those two particles. We know therefore that

$$\widetilde{\frac{\partial \Psi}{\partial r_{12}}} = \gamma \psi(r_{12} = 0) \tag{2.21}$$

$$\gamma = 1/2 \text{ for opposite spin electrons} \tag{2.22}$$

$$\gamma = 1/4 \text{ for parallel spin electrons} \tag{2.23}$$

$$\gamma = -Z \text{ for electron-nucleus,} \tag{2.24}$$

where $\widetilde{\frac{\partial \Psi}{\partial r_{12}}}$ denotes a spherical average of the derivative of the wavefunction as the distance between any two particles, r_{12}, reaches zero. If we are going to add Jastrows to our wavefunction, then we can easily constrain those functions to satisfy these constraints, called the cusp conditions, and thereby eliminate some of the sources of singularities in the local energy.

2.5 Diffusion Monte Carlo

There is yet another way to solve Equation 2.3, which we find by rewriting the time-dependent Schrödinger Equation (Equation 2.2) in imaginary time, $\tau = it$. Following the arguments as presented by Reynolds and co-workers in [2], we write

$$-\frac{\partial}{\partial\tau}\Psi(r,\tau) = \left[\hat{H} - E_T\right]\Psi(r,\tau), \tag{2.25}$$

where E_T is simply an energy shift whose importance will become evident. What we actually want is the time-independent solution, which is simply the steady state of Equation 2.25. Expanding $\Psi(r,\tau)$ in a complete set of eigenfunctions $\psi_i(r)$ of the Hamiltonian, the wavefunction will look like

$$\Psi(r,\tau) = \sum c_i e^{-(E_i - E_T)\tau}\psi_i(r) \tag{2.26}$$

which at long times will come to be dominated by the state with the eigenvalue closest to E_T

$$\Psi(r,\tau) = c_0 e^{-(E_0 - E_T)\tau}\psi_0(r), \tag{2.27}$$

which will be the exact ground state $\psi_0(r)$ if E_T is adjusted to our best guess. If our Hamiltonian consisted of only the Laplacian $-\nabla^2/2$, then this would be a typical diffusion equation, which we could simulate with walkers, just as we did in VMC with Equation 2.19. On the other hand, if the Hamiltonian was only a potential energy term $V(r)$, then Equation 2.25 is simply a rate equation, which is simulated by using birth and death processes in a population. For a molecular Hamiltonian, we can combine both approaches by enriching or duplicating walkers in regions of favorable potential energy, an approach called Diffusion Monte Carlo (DMC). The only problem is that because we are using a population of walkers to represent the wavefunction, the represented wavefunction must be the same sign everywhere. Since we are simulating fermions which have nodes, we are required to simulate the positive and negative regions separately, and average the results. This represents an approximation if the nodes are not correct, introducing an error, called the fixed-node energy.

The most interesting thing to note about the DMC algorithm is that, except for the

location of the nodes, we do not have to know anything about the wavefunction; in principle any will work. To speed up convergence, we should use our best guess of the wavefunction for importance sampling, and specific choices for this kind of "trial function" will be the subject of later chapters. Designating $\Psi_T(r)$ as our trial function, our population distribution function is $f(r,\tau) = \Psi(r,\tau)\Psi_T(r)$. We multiply Equation 2.25 by $\Psi_T(r)$ to get

$$\begin{aligned}
-\Psi_T(r)\frac{\partial}{\partial\tau}\Psi(r,\tau) &= \Psi_T(r)\left[\hat{H} - E_T\right]\Psi(r,\tau) && (2.28)\\
-\frac{\partial f(r,\tau)}{\partial\tau} &= \Psi_T(r)\left[\hat{H} - E_T\right]\frac{f(r,\tau)}{\Psi_T(r)} && (2.29)\\
&= (V(r) - E_T)f(r,\tau) - \frac{1}{2}\Psi_T(r)\nabla^2\frac{f(r,\tau)}{\Psi_T(r)} && (2.30)\\
-\frac{\partial f(r,\tau)}{\partial\tau} &= -\frac{1}{2}\nabla^2 f + (E_L(r) - E_T)f + \nabla\cdot\left(f\frac{\nabla\Psi_T(r)}{\Psi_T(r)}\right) && (2.31)
\end{aligned}$$

which can be solved with the integral equation

$$f(r',\tau+\delta) = e^{\delta E_T}\int G(r\to r',\delta)f(r,\tau) \qquad (2.32)$$

using the Green's function

$$\begin{aligned}
G(r\to r',\delta) &= (2\pi\delta)^{-3N/2} && (2.33)\\
&\times \exp\left[-\delta\left\{\frac{E_L(r)+E_L(r')}{2} - E_T\right\}\right] && (2.34)\\
&\times \exp\left[-\frac{[r'-r-\delta\frac{\nabla\Psi_T(r)}{\Psi_T(r)}]^2}{2\delta}\right] && (2.35)
\end{aligned}$$

which represents the probability of N particles moving from r to r' for time step δ. The last term is used to move the electrons, drifting them with $\frac{\nabla\Psi_T(r)}{\Psi_T(r)}$. The middle term is incorporated by either weighting the walkers, or by branching them (or both). A DMC calculation starts by generating some walkers which compose $f(r,0)$, and then we apply Equation 2.32 as many times as it is necessary to equilibrate to the steady state of Equation 2.27. After this, we may start sampling the local energies to get our result. The average energy carries with it a time step error, which can be eliminated by extrapolating $\delta\to 0$, as demonstrated in Figure 2.2.

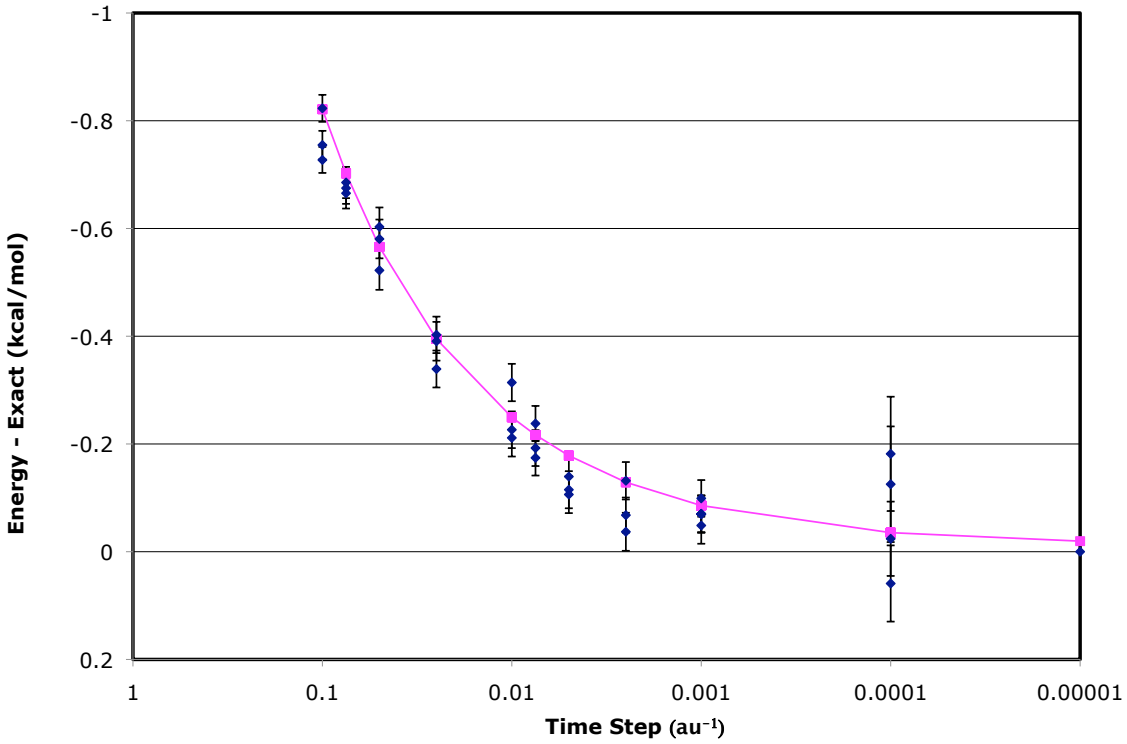

Figure 2.2: An extrapolation to zero time step, demonstrated for a Helium atom using unoptimized, Pade 2 particle Jastrows. This case is particularly easy since He has no wavefunction nodes. We fit our calculations to the formula $E = E_0 + \sum_{i=1}^{6} c_i \delta^{i/2}$, producing $E_0 = -2.903744$ au, which is in error by only 0.012 kcal/mol from the exact answer $E_{exact} = -2.903724$ au. Note: the data at $\delta = 10^{-5}$ actually represents $\delta = 0$.

Each walker now has an associated weight which is multiplied by

$$dW = \exp\left[-\delta\left\{\frac{E_L(r) + E_L(r')}{2} - E_T\right\}\right] \tag{2.36}$$

each iteration. If this weight becomes large, a typical DMC algorithm might then duplicate the walker, giving each of the child walkers half the weight of the parent. A walker whose weight becomes too low is eventually deleted because it is wasting computational resources. Because of this, the final algorithm will end up looking very similar to the algorithm in VMC,

$$\langle E\rangle \simeq \frac{1}{N_t}\sum_{t=1}^{N_t}\left\langle \frac{1}{\sum w_i}\sum_{i=1}^{N_w} w_i E_L(r_{t,i})\right\rangle_{A(r_f|r_i)} + \mathcal{O}\left(\frac{\sigma}{\sqrt{N_t}}\right) \tag{2.37}$$

$$A(r \to r') = \min\left[1, \frac{G(r \leftarrow r', \delta)}{G(r \to r', \delta)}\frac{\Psi_T^2(r')}{\Psi_T^2(r)}\right] \tag{2.38}$$

using the DMC Green's function. It turns out that this choice of transition matrix to move the electrons is a good choice for VMC as well. We can actually use the same software, with the only difference being that $dW = 1$ in VMC.

2.6 Practicum

Quantum Monte Carlo is a good deal more sophisticated than we have presented in this Chapter, and certainly there are quite a few algorithms and variations allowed under the rubric we have presented here. Most of the high level or theoretical aspects of our techniques are addressed in Chapter 4. In that chapter, we present our recommendations for wavefunctions, time steps, and other details. With that chapter, we justify our approach on the basis of the remarkable accuracy of our results.

The experience gained in developing the code with accessory scripts to run a QMC calculation is quite valuable. The software package that we have participated in developing is called QMcBeaver [4], which is available online under the GNU Public License. We have used a Concurrent Versions System (CVS) throughout development. This works by allowing the developer to download a copy of the source code, and edit it at their pleasure. Once that developer is happy with their changes and has checked for bugs, they commit all of their changes back to the online repository, complete with a brief description of what that

commit entailed. With good CVS software, it is possible to observe the exact evolution of any part of the code. Since this thesis represents a significant point in the development of the code, we use the label *amos_phd_thesis* in the repository to record the exact version of all of the source code files corresponding to our work.

Additionally, we document and describe here the most important scripts used for setting up and running QMcBeaver. Let this section, along with the associated Appendices, serve as a QMcBeaver recipe. We do not claim that these scripts can be considered as complete, or that it is unnecessary for a user to edit them. They have only been developed as need arose.

1. Pick an SCF wavefunction. This step will depend on your intuition and the process you want to model, but we discuss our experiences in this regard in Chapter 4. As we discuss there, we have found that extended MCSCF or CASSCF calculations do not necessarily work better due to the uncertainties in optimization. On the other hand, GVB wavefunctions are not sufficient for all problems, with the atomization of CN or NO as examples. If you are using a GVB wavefunction, then we would recommend using Jaguar [5] to make the wavefunction, since it does a good job at making initial guesses. In Appendix C we discuss and provide a script to convert a Jaguar wavefunction into a GAMESS wavefunction. We have found that GAMESS [6] is the most useful program available for producing wavefunctions because it is free, readily downloadable, under active development, and very flexible. One note is that we do not allow users to use an MCSCF calculation directly. Instead, following the recommendation from GAMESS, we require the user to run a CI calculation on the MCSCF natural orbitals to get the best CI coefficients possible. Be sure that you set the print cutoff low enough that GAMESS prints out enough determinants.

2. Visualize your SCF orbitals. We have found that quite often, either Jaguar or GAMESS converged orbitals that were not what we expected. There is an excellent visualization package available, called MacMolPlt [7], for seeing orbitals from a GAMESS calculation. We prefer to use orbitals that are either symmetric or localized, but difficult SCF optimizations might produce anything in between. In these cases, it is helpful to start or restart the optimization with good initial orbitals such as the kind Jaguar can generate.

3. Run the script gamess2qmcbeaver.py, documented in Appendix D, which will extract the wavefunction, and make a *.ckmf* input file for QMcBeaver. This script has been under continual development by several people over the years who have fixed many bugs, and it is fairly complete. This script bases the input file for QMcBeaver on a *.ckmft* file, which is a template containing a good set of non-specific parameters, and we provide our best example in Appendix D. There are two choices to make when using this script. First, you must choose a determinant cutoff, since our script will by default add all of the determinants available in the GAMESS output file. Typically, a cutoff of 0.01 is low enough to capture most of the chemistry, but as we discuss in Chapter 4, that may be too high. For a GVB wavefunction, I typically include all of the determinants since they are not expensive, for reasons documented in Appendix D. Second, you must decide on a tolerance to use for deciding whether two determinants should be constrained to use the same CI coefficient. We have found that constraints can help avoid local minima, but obviously two determinants should only be constrained if there is good enough reason to.

4. The script gamess2qmcbeaver.py will not automatically guess Jastrow functions for you. We do not believe that a generic Jastrow function strategy will work, so we leave it to the user to select Jastrow functions to initialize the optimization. We have found it is more important to match the basis set for picking the starting Jastrow functions than matching the SCF type of wavefunction. The 3 particle Jastrows are particularly difficult to optimize, and for sufficiently large molecules, they add more to the computational cost than they seem to be worth. We either need to develop new 3 particle Jastrows, or find a better way to use the ones we have already. One idea is to fix the length scale of 3 particle Jastrows so that they do not stretch further than the atom on which they are centered, thus limiting their cost. More discussion on Jastrow functions can be found in Chapter 4.

5. If the input file you generated in the previous step used the *.ckmft* file from Appendix D, then this input file is ready for optimization. I typically run a calculation using only 1 or 2 processors, since the number of optimization iterations seems to be more important than the number of samples per optimization iteration. We have found that some Jastrows are particularly troublesome to optimize, and we detail our strategy

for identifying and dealing with these in Chapter 4. As we discuss there, we have found that in the end, most Jastrows look quite similar, even though they vary in height or extent. We have developed a script called *optimized.pl*, which we document in Appendix E, to help decide when a wavefunction is optimized. Typically, we look for the Jastrows not to significantly change between optimization iterations, and for the VMC energy to converge to less than a few tenths of a kcal/mol, or 0.5 kcal/mol at the worst. As soon as it is available for each optimization step, the most recent wavefunction is written to a *.01.ckmf* file.

6. Once the optimization has satisfactorily converged, we edit a few parameters to select a DMC calculation. This involves setting *run_type = diffusion* and *optimize_Psi = 0*, as well as choosing an appropriate time step and number of iterations. I typically run on 4 processors (for a total of 400 walkers) for reasons discussed in Section 5.5. Calculations can take anywhere from a couple of days to a couple of weeks, depending on the molecule size and the processor speeds.

7. It is important to monitor the DMC convergence as it progresses, because sometimes a calculation can "go crazy". Ideally, a DMC calculation will maintain an approximately constant energy through the run, with a few wiggles. We have developed a pair of scripts, which we document in Appendix F, to look at snapshots of the energy or to produce a graph of the energies as they progress. Many runs will display deviations or tails, which we typically ignore if they are less than a few tenths of a kcal/mol. However, if instead of a tail we see a trend with non-zero slope over the length of the calculation, then something is wrong. Perhaps the run should be restarted with possible fixes including returning to the wavefunction optimization stage, adding more equilibration steps, using a smaller time step, using more walkers, or improving the SCF description.

Chapter 3

Quantum Monte Carlo on Graphical Processing Units

3.1 Abstract

Quantum Monte Carlo (QMC) is among the most accurate methods for solving the time-independent Schrödinger equation. Unfortunately, the method is very expensive and requires a vast array of computing resources in order to obtain results of a reasonable convergence level. On the other hand, the method is not only easily parallelizable across CPU clusters, but as we report here, it also has a high degree of *data parallelism*. This facilitates the use of recent technological advances in Graphical Processing Units (GPUs), a powerful type of processor well known to computer gamers. In this paper we report on an end-to-end QMC application with core elements of the algorithm running on a GPU. With individual kernels achieving as much as 30x speed up, the overall application performs at up to 6x relative to an optimized CPU implementation, yet requires only a modest increase in hardware cost. This demonstrates the speedup improvements possible for QMC in running on advanced hardware, thus exploring a path toward providing QMC level accuracy as a more standard tool. The major current challenge in running codes of this type on the GPU arises from the lack of fully compliant IEEE floating point implementations. To achieve better accuracy, we propose the use of the Kahan summation formula in matrix multiplications. While this drops overall performance, we demonstrate that the proposed new algorithm can match CPU single precision.

3.2 Introduction

The rapid increase in GPU floating point performance and their excellent flops/$ characteristics suggests that they may provide cost effective solutions for scientific computation problems. Given that the GPU computing model is (1) quite different from standard CPU models, (2) lacks a fully compliant IEEE floating point implementation, and (3) is optimized for very specific graphics type computational kernels, it is not clear *a priori* which scientific computing tasks are cost effective on GPUs.

A number of scientific computing algorithms have been pursued on the GPU, *e.g.*, fluid simulations [8, 9], elasticity [10], and general finite element methods [11]. At the level of computational mathematics kernels, we have seen work on LU decomposition [12], matrix/vector products [13, 14, 15, 16, 17, 18, 19, 20, 21], iterative solvers [17, 22], and transforms such as Fourier and Wavelet [14, 23, 24, 25]. In some cases the results can be disappointing relative to highly tuned CPU implementations, in particular when high precision answers are required, or when problem sizes do not hit a particular sweet spot (*i.e.*, large matrices, or power-of-2 sized data structures, *etc.*). With continuing hardware development these performance barriers are being ameliorated, and with the recent announcement by nVidia of double precision availability on the GPU in 2007, computational precision is a fading problem as well.

In this paper we consider quantum chemistry computations, the heart of which is the computation of the electronic structure of a given molecule using the quantum mechanical equations of motion. This information is critical for, among other tasks, finding optimized geometric structures for the molecule, reaction pathways, obtaining vibrational information, and providing a basis for developing higher level approximation methods including molecular dynamics simulations. Accurate results have application in catalysis, nanotechnology, drug design, and fuel cells, among many others.

Due to the large state space ($3N$ for N electrons) and the non linear nature of the time-independent Schrödinger equation, exact results are all but impossible. Consequently a variety of approximation algorithms have been developed. One such approach, Quantum Monte Carlo (QMC) [3], is based on the stochastic evaluation of the underlying integrals and is guaranteed to produce accurate answers in the limit of infinite state space sampling. Even though a very large number of samples are typically required, QMC is easily paral-

lelizable and scales as $O(N^3)$ (albeit with a very large constant). This motivates a search for computational augmentation.

We report on our implementation of QMC on the nVidia 7800 GTX and compare it against a 3.0 gHz Intel P4, considered to be representative of similar levels of development. These technologies are improving very fast, both for CPUs and for GPUs. Currently however, the time to doubled performance on GPUs is noticably shorter than for CPUs, leading to increasing performance advantages for GPUs *if a computation maps well enough onto the GPU*. Since CPUs are beginning to follow the same multicore technology trend, the notion that precision issues are temporal is reinforced.

In the present paper, scientific results as well as underlying formalisms were simplified for purposes of presentation and to focus on the essential computational aspects. We admit that it is unclear how single precision results might be useful, especially for an algorithm designed to produce highly accurate results. In the mean time, our single precision implementation is presented. Aside from the performance of individual kernels we consider (1) precision issues arising from the noticeable differences to single precision IEEE floating point arithmetic, (2) performance issues arising from the specific sizes of matrices we must use, and (3) the overall performance of an end to end application when compared against a heavily tuned CPU based version.

3.3 Introduction to Graphical Processing Units

GPUs have received much interest outside the graphics world recently due to their immense processing power even though they are actually devices designed for very specialized tasks. Many reviews of GPU adaptability and compatibility are already available [26, 8, 27], and we do not attempt to improve upon them. In addition, there has been the development of specialized programming environments [13, 28, 29] for GPUs specifically designed to smooth the porting of non-graphics applications, and GPU vendors themselves have recently released general purpose GPU programming environments.

Our approach was to start from the ground up in hopes of squeezing the best performance we can from the device. To describe our techniques, a truncated description of the technology is required. The motivating principle for GPU design is that simple calculations do not need general processors, so the addition of an auxiliary processor could both

speed up graphics related calculations as well as free the CPU to complete other tasks. Since graphical calculations most typically involve drawing 2D images of colors ultimately intended for a screen, GPUs start with pixels (more generally referred to as fragments or texels) as the atomistic unit of data. Fragments are manifested here as 4 single precision floats, aliased as `xyzw` channels. A 2D array of fragments is called a *texture*, and is the fundamental storage class. A GPU will stream a region of a texture through an array of simple *fragment processors* (our nVidia 7800 GTX has 24), where each of these will produce one fragment as output. A programmer can utilize this process by designating a kernel for the fragment processors to use, resulting in the evaluation of data for a specified region in a texture. This entire procedure is commonly referred to as a *pass*. A kernel is a small program which in the graphics context would typically perform some shading calculation. There is nothing in principle preventing the user from writing a "shader" which performs some scientifically relevant computation using the broad class of functions available at the programmable shader level.

In practice, many considerations are necessary in order to maximize efficiency. Graphics processing can be thought of as a sophisticated queuing system where a CPU sends a list of tasks to one (or more) connected GPUs and collects the results when the calculations are complete. This means that there are also processor communication factors that need to be included. As far as the GPU itself is concerned, we mention here the considerations:

- padding empty slots in texture data with 0 whenever data dimensions do not match dimensions on the GPU,
- running as many passes with a kernel before swapping it for another since the GPU can only have one kernel loaded at a time,
- careful data arrangement,
- a tuning of how much of the computation as a whole should be assigned to each kernel
- and, in general, keeping the GPU busy at all times.

Before discussing how these concerns play out in our setting, we give a brief high level introduction to Quantum Monte Carlo computations to understand the needed computational components which we seek to map to the GPU.

3.4 Introduction to Quantum Monte Carlo

The most important information about a molecule is its ground state energy, calculated by means of the time-independent Schrödinger equation

$$\langle E \rangle = \frac{\int \Psi(\bar{\boldsymbol{r}}) \hat{H} \Psi(\bar{\boldsymbol{r}}) d\bar{\boldsymbol{r}}}{\int \Psi^2(\bar{\boldsymbol{r}}) d\bar{\boldsymbol{r}}}, \tag{3.1}$$

where $\Psi(\bar{\boldsymbol{r}}) : \mathbb{R}^{3N} \rightarrow \mathbb{R}$ is the wavefunction, mapping the $3N$ Cartesian coordinates of N electrons into a probability amplitude related to the probability density in Equation 3.4. (Equation 3.1 includes the common restriction that $\Psi(\bar{\boldsymbol{r}})$ is a real valued function.) The Hamiltonian operator $\hat{H}$ is given by

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(\bar{\boldsymbol{r}}), \tag{3.2}$$

where the Laplacian is over all $3N$ electronic coordinates and calculates the kinetic energy (in the unitless Hartree measure) of the electrons in the molecule. The $V(\bar{\boldsymbol{r}})$ term represents the potential energy due to Coulomb interactions between all pairs of electrons and nuclei. The energy E is the eigen value of $\hat{H}$ operating on the eigen function $\Psi(\bar{\boldsymbol{r}})$. The ground state energy is the lowest such eigen value, and is of primary interest here.

There are many methods to calculate Equation 3.1 with varying degrees of accuracy and computational complexity. The highly accurate QMC family of algorithms [2] uses Metropolis [30] integration to fine tune the result provided by a cheaper method. It uses the *local energy*

$$E_L(\bar{\boldsymbol{r}}) = \frac{\hat{H}\Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} = -\frac{1}{2}\frac{\nabla^2\Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} + V(\bar{\boldsymbol{r}}) \tag{3.3}$$

which represents an evaluation of the energy for a set of electronic coordinates. In terms of the stationary probability distribution of electrons

$$\rho(\bar{\boldsymbol{r}}) = \frac{\Psi^2(\bar{\boldsymbol{r}})}{\int \Psi^2(\bar{\boldsymbol{r}}) d\bar{\boldsymbol{r}}} \tag{3.4}$$

we can transform Equation 3.1 into the Monte Carlo integration form

$$\langle E \rangle = \int \rho(\bar{\boldsymbol{r}}) E_L(\bar{\boldsymbol{r}}) d\bar{\boldsymbol{r}} = \lim_{N_t \rightarrow \infty} \frac{1}{N_t} \sum_{t=1}^{N_t} E_L(\bar{\boldsymbol{r}}_t). \tag{3.5}$$

Here $\bar{\boldsymbol{r}}_t$ are a series of electronic coordinates generated with respect to $\rho(\bar{\boldsymbol{r}})$ by some importance sampling scheme [31]. Since error scales as $1/\sqrt{N_t}$ in Monte Carlo methods a rather large number of samples is required to achieve useful accuracies. Additionally, it is common to run several independent series, called *walkers*, in order to minimize the error due to serial correlation between the N_t data points.

In terms of computational complexity, the difficulty for QMC lies in the evaluation of $\nabla^2\Psi(\bar{\boldsymbol{r}}_t)$ for each $E_L(\bar{\boldsymbol{r}}_t)$ as well as the evaluation of $\Psi(\bar{\boldsymbol{r}}_t)$ and $\nabla\Psi(\bar{\boldsymbol{r}}_t)$ which are used for importance sampling. The most common functional form for $\Psi(\bar{\boldsymbol{r}})$ has at least three nested stages of evaluation. At the first stage, we place a collection of N_{bf} basis functions centered at the nuclei in the $3D$ coordinate space. Typically a given nucleus is associated with multiple basis functions. The basis function takes as argument the local coordinates of a given electron (i) relative to the nucleus (j), $\vec{r}_{ij} = \vec{r}_i - \vec{R}_j$. The best results are achieved with the following functional form

$$\chi_j(x_{ij}, y_{ij}, z_{ij}) = x_{ij}^{k_j} y_{ij}^{l_j} z_{ij}^{m_j} \sum_{n_j} a_{n_j} e^{-b_{n_j} r_{ij}^2}. \tag{3.6}$$

For each basis function, R_j, k_j, l_j, m_j, n_j, a_{n_j} and b_{n_j} are parameters given as input to the QMC program. The $k_j, l_j, m_j \in \mathbb{N}$ parameters give the basis function the required symmetry, and $n_j \in \mathbb{N}^+$ helps select the quality of fit. The other parameters are all real numbers.

The second stage of evaluation takes linear combinations of basis functions to create *molecular orbitals*. The k^{th} orbital is given by $\phi_k(\vec{r}_i) = \sum_j \chi_j(r_{ij}) c_{jk}$, where $c_{jk} \in \mathbb{R}$ are coefficients input to QMC. These orbitals represent the spread of the electron across the entire molecule.

Finally, the third stage of evaluation relevant to this study is the *Slater determinant*, chosen for its antisymmetric properties. For the N_s electrons of a given quantum spin ($N = N_\alpha + N_\beta \sim 2N_\alpha$) the determinant is a function of the ϕ_k (which in turn are functions

of the $\chi_j(r_{ij})$)

$$D_s(\bar{\boldsymbol{r}}_s) = |M_s(\bar{\boldsymbol{r}}_s)| = \begin{vmatrix} \phi_1(\vec{r}_1) & \phi_2(\vec{r}_1) & \cdots & \phi_{N_s}(\vec{r}_1) \\ \phi_1(\vec{r}_2) & \phi_2(\vec{r}_2) & & \\ \vdots & & \ddots & \\ \phi_1(\vec{r}_{N_s}) & & & \phi_{N_s}(\vec{r}_{N_s}) \end{vmatrix} \tag{3.7}$$

(here we partition $\bar{\boldsymbol{r}}$ into $\bar{\boldsymbol{r}}_\alpha$ and $\bar{\boldsymbol{r}}_\beta$) and the wavefunction is

$$\Psi(\bar{\boldsymbol{r}}) = D_\alpha(\bar{\boldsymbol{r}}_\alpha) D_\beta(\bar{\boldsymbol{r}}_\beta).$$

To calculate the kinetic energy, we first obtain $\nabla_i^2 \phi_k(\vec{r}_i) = \sum_j \nabla_i^2 \chi_j(\vec{r}_{ij}) c_{jk}$, and then sum the contributions from all the electrons in all the orbitals

$$\frac{\nabla^2 \Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} = \sum_{s \in \{\alpha,\beta\}} \sum_{i,k \in N_s} \left[M_s^{-1}(\bar{\boldsymbol{r}}_s)\right]_{ki} \nabla_i^2 \phi_k(\vec{r}_i). \tag{3.8}$$

A similar procedure is followed for calculating the gradient of the wavefunction for each electron with the exception that the final summation results in a vector of gradients.

To summarize the algorithm, we are given a set of nuclear coordinates, basis function parameters, and the c_{jk}, which describe the wavefunction as fit by some other (more approximate and cheaper) method. Additionally, we choose some parameters including the number of steps N_t, the number of walkers W, an initial guess scheme for positions $\bar{\boldsymbol{r}}$ of all the electrons, as well as several parameters relating to the importance sampling. Although specific choices are often related to the computational resources available and to the importance sampling method used, W is usually $O(10)$ to $O(10^3)$, N_t is $O(10^4)$ to $O(10^8)$, and the dimensions of c_{jk} are usually between $O(10)$ and $O(10^3)$, depending upon the molecule. With these in hand, the algorithm can be stated as shown in Algorithm 1 (the $\otimes$ represents matrix multiplication), where simplifications have been included based on assumptions about the importance sampling.

The high degree of parallelism is evident since each processor can calculate all the linear algebra for its walkers and only needs to produce a single value; the energy. *

*While some QMC algorithms only update one electron per Monte Carlo step, our method updates all at once [31].

Algorithm 1 The QMC algorithm

$E_{sum} \leftarrow 0$
for $w = 1$ to W **do**
  $\vec{r}_{ij} \leftarrow$ initialize()
  for $t = 1$ to N_t **do**
    for $s = \alpha$ and $s = \beta$ **do**
      $M_s \leftarrow \chi_j(\vec{r}_{ij}) \otimes c_{jk}$
      $X_s \leftarrow \frac{\partial}{\partial x_i}\chi_j(\vec{r}_{ij}) \otimes c_{jk}$
      $Y_s \leftarrow \frac{\partial}{\partial y_i}\chi_j(\vec{r}_{ij}) \otimes c_{jk}$
      $Z_s \leftarrow \frac{\partial}{\partial z_i}\chi_j(\vec{r}_{ij}) \otimes c_{jk}$
      $L_s \leftarrow \nabla_i^2\chi_j(\vec{r}_{ij}) \otimes c_{jk}$
    end for
    Jastrow $\leftarrow J(\bar{\boldsymbol{r}})$
    $\Psi \leftarrow \det M_\alpha * \det M_\beta *$ Jastrow
    $E_{sum} \leftarrow E_{sum} +$
      $E_L(M_s, \text{Jastrow}, \{derivatives\}...)$
    $\vec{r}_{ij} \leftarrow \text{sampling}(\Psi, \vec{r}_{ij}, X_s, Y_s, Z_s, L_s)$
  end for
end for
$E_{avg} \leftarrow E_{sum}/(N_t * W)$

One big advantage of QMC relative to alternative methods is the freedom one has in choosing the functional form of $\Psi(\bar{\boldsymbol{r}})$. This is exploited by multiplying the Slater determinant wavefunction with a set of pairwise interaction terms which explicitly model electron correlation by employing inter-electronic coordinates. The only condition is that these terms, called Jastrow functions, preserve the antisymmetry of the wavefunction. To satisfy this condition, we use the functional form

$$J(\bar{\boldsymbol{r}}) = \prod_{q<p} e^{u_{pq}(r_{pq})} \tag{3.9}$$

which provides a term for each particle-particle interaction, where

$$u_{pq}(r_{pq}) = \frac{\sum_{\kappa=1}^{\Gamma} a_{pq\kappa} r_{pq}^{\kappa}}{1 + \sum_{\kappa=1}^{\Lambda} b_{pq\kappa} r_{pq}^{\kappa}} \tag{3.10}$$

and p and q index all electrons and nuclei, and r_{pq} is the distance separating the two particles. The number of terms (Γ and Λ) is arbitrary, and depends on the quality of fit. These parameters, along with $a_{pq\kappa}, b_{pq\kappa} \in \mathbb{R}$, are input to the QMC algorithm. With this modification, our wavefunction is now $\Psi_{QMC}(\bar{\boldsymbol{r}}) = D_\alpha(\bar{\boldsymbol{r}}_\alpha)D_\beta(\bar{\boldsymbol{r}}_\beta)J(\bar{\boldsymbol{r}})$, and there are chain

rule effects for the gradient and Laplacian. The rationale for these additional terms is the improved convergence if the wavefunction is a better approximation of the eigen function of $\hat{H}$ to begin with. Jastrow functions involving 3 particles were not considered here.

Within the family of QMC algorithms, there are two popular varieties. The first is called Variational Monte Carlo (VMC) in which the procedure described in this section is employed to provide an exact integration for the given wavefunction. The method is termed variational since it is commonly coupled with a wavefunction optimization step. Diffusion Monte Carlo (DMC) uses the wavefunction only as a guide. Instead of a direct integration, it has a mechanism to project out a (mostly) correct wavefunction, and thus provide exact energies for the system. That said, a DMC calculation will converge better for higher quality wavefunctions. The subject matter considered here is agnostic to this choice except that DMC includes slightly more computational effort than VMC.

3.5 Implementation on the GPU

The QMcBeaver [4] code, under development in our group to perform QMC calculations, was used as the CPU implementation on which to base our study of a GPU implementation. In order to locate the computationally expensive components in the code, we minimize file I/O, ignore localization procedures which lead to sparser matrices [32, 33], and we only consider single determinant, restricted Hartree-Fock wavefunctions. Moving all electrons at once allows us to use the highly optimized matrix multiplication routines available in the ATLAS 3.7.11 [34, 35] BLAS library and use the LAPACK extension to ATLAS to perform the necessary matrix inversions. Using this representation of QMC as our starting point, we find that the computational effort on the CPU for N electrons is approximately 11% focused on the 10 dense matrix multiplications at $O(N^3)$ each, 73% on the 10 basis function set evaluations at $O(N^2)$ each, and 4% on the (electron - electron) pairwise Jastrow function evaluations at $O(N^2)$. These fractional estimates are relatively stationary for molecules with as many as 150 electrons. The leading components not yet ported to the GPU include matrix inversion and electron-nuclear Jastrow functions as well as other processes specific to DMC.

For the molecule sizes we are targeting the matrices are small and rectangular; specializations currently overlooked in GPU code. Combined with the fact that the c_{jk} matrix can

be reused for all matrix multiplications, we pursued several optimization strategies in detail. In particular, all of our kernels were designed to evaluate as many walkers simultaneously as GPU hardware limitations permit.

3.5.1 Walker Batch Scheme

The GPU pipeline is very deep, so there is a substantial overhead cost for any calculation we wish to perform. This is in terms of work the GPU has to do to prepare for a given calculation, effort needed to move the GPU into full production efficiency, and any costs incurred by traversing the CPU/GPU boundary. This can be amortized by processing as many fragments simultaneously on the GPU as possible. For Monte Carlo type algorithms, we can accomplish this by increasing the number of walkers processed per GPU pass. This has allowed us to tune both the size of the problem and the texture aspect ratio to the GPU. For example, we can arrange our data in GPU memory according to an empirically optimized pattern such as 4 rows by 4 columns so that each pass amounts to 16 walker evaluations in parallel.

3.5.2 Basis Function Evaluation

The number of basis functions, as well as their controlling parameters, are chosen according to chemical considerations. Typical are 5 basis functions for each Hydrogen and 15 basis functions for each atom Lithium to Neon, leading to a matrix aspect ratio of between 4 and 8. The choice of basis set and all associated parameters are held fixed during a run and evaluation only depends on the $3N$ electronic coordinates, producing value, gradient, and Laplacian.

3.5.2.1 Kernel 1: Data Generation

The major choice regarding basis function evaluation (Equation 3.6) concerns the organization of the output data: different regions of one output texture or separation by channel (`xyzw`) resulting in two output textures. We opted for keeping the output in different regions so as to allow specialization (*i.e.*, derivatives) of the kernels. As regards input data reuse, we opted for evaluating a single basis function for 4 electrons. This choice minimizes texture lookups and increases instruction parallelism since only one n_j from Equation 3.6

is used in the same fragment.

3.5.2.2 Kernel 2: Layout Conversion

Most matrix multiplication approaches on the GPU pack 2x2 submatrices into a single `xyzw` memory slot and we employed this layout as well. The basis function evaluation output is in 4x1 layout, necessitating a conversion which we used to filter out any bad values as well. Due to the batching (Section 3.5.1) texture layout, fences between rows and columns of walkers required special maintenance at this stage.

3.5.3 Matrix Multiplication

For purposes of performance comparison, we used the ATLAS 3.7.11 [34, 35] library's single precision matrix multiplication on our 3 GHz Pentium 4 as a CPU benchmark. For the GPU, several studies of matrix multiplication performance have been performed [14, 15, 16, 18, 20, 21] so our main focus is on the performance for the (relatively) small rectangular matrices we encounter in our application, as well as the fact that we use the same multiplicand for all multiplications.

For the 2x2 layout the inner product for the pixel at `C[i,j]` becomes the series of pixel products

```
for(k=0; k<N; k++){
  C[i,j].xyzw += A[i,k].xxzz*B[k,j].xyxy
                      +   A[i,k].yyww*B[k,j].zwzw;
}
```

with `N` representing the number of pixels used in the inner product. In the GPU vector notation above, the $C[i,j].x$ data written separately is

```
C[i,j].x += A[i,k].x*B[k,j].x
              +   A[i,k].y*B[k,j].z}.
```

The values are stored in row-major format across the `xyzw` channels. This method can be modified to take advantage of multiple render target (MRT) [15] functionality on the GPU. Essentially, MRTs can take advantage of up to 4 related data structures on the GPU with which to arrange and facilitate reuse of data.

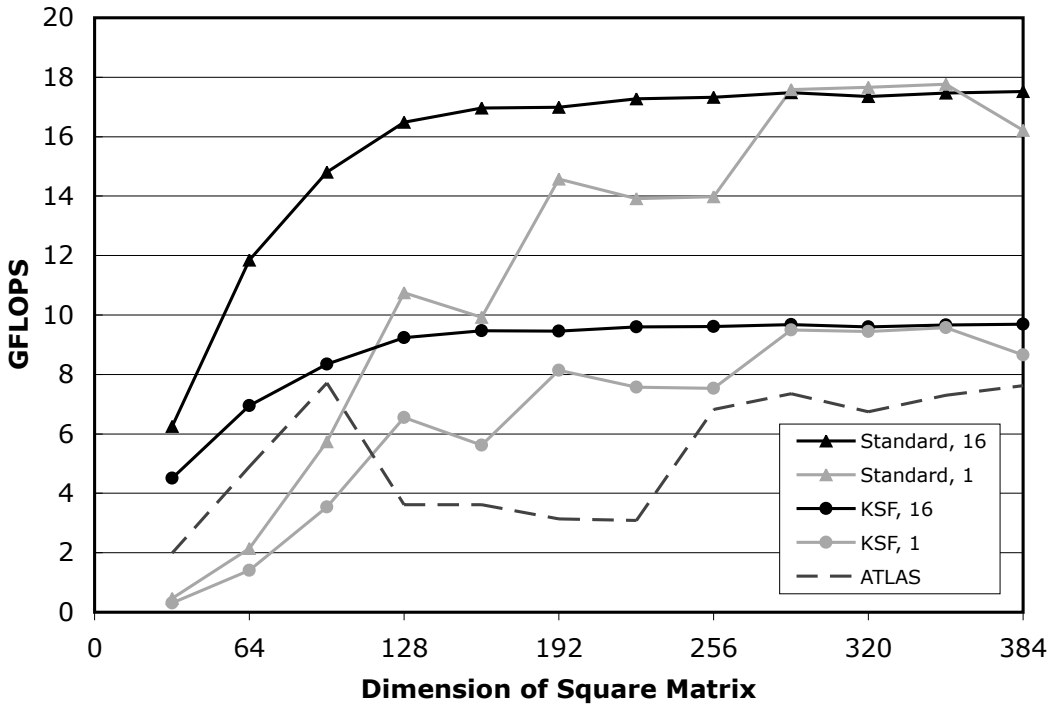

Figure 3.1: The cost of correcting for the summation error in multiplication of square matrices. Indicated is the number of multiplications performed simultaneously, reusing the multiplicand.

The results shown in Figures 3.1 and 3.2 both show the matrix performance speedups for a variety of matrix sizes and parameter choices. The effect of multiplying several matrices simultaneously is to raise the performance level (in terms of GFLOPS) for smaller matrices. When performing calculations using rectangular matrices, the set up costs can be quenched almost entirely. It is also apparent that for some domains, the GPU has significant performance gains relative to the CPU when CPU cache peculiarities play a role. Although the KSF error correcting algorithm (described in Section 3.6.2) negates most speedup gains for the particular technologies compared here, the hidden advantage remaining is that the calculation is performed on the GPU, minimizing GPU/CPU communication.

3.5.4 Jastrow Functions

The third most computationally demanding component of our QMC algorithm is the evaluation of the pairwise Jastrow function in Equation 3.9. For the GPU implementation, we focused on porting the electron-electron terms (electron-nuclei terms are substantially fewer). We need to evaluate N choose 2 polynomials (one for each electron-electron pair)

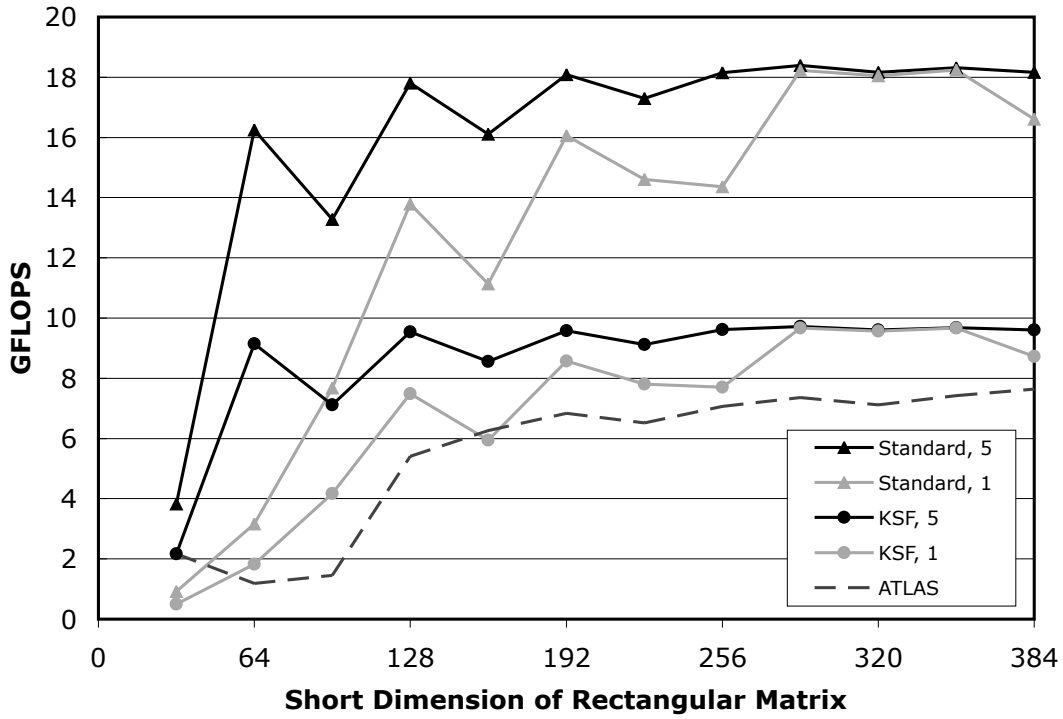

Figure 3.2: The dimension of the inner product is 6 times that of the short dimension shown. The multiplicand is reused for all 5 multiplications.

which are then summed. Since parameters in Equation 3.10 differ between same/opposite spin electron pairs, texture data is partitioned in order to allow kernel specialization.

We proceed in 3 steps:

Kernel 1 evaluates the magnitude and normalized vector between all pairs of electrons for a total of 4 values per fragment.

Kernel 2 finds the value, Laplacian, and gradient of Equation 3.9, writing the first two to one texture and the latter three to another.

Kernel 3 computes the sums, maintaining the electron indices for the gradient summands.

3.6 GPU Floating Point Error

One of the goals of quantum chemistry is the calculation of the electronic energy of a molecule with sufficient accuracy, stated as 1 to 2 kcal/mol. To this end, absolute error of the final result must not be worse than 1×10^{-3} hartrees. An appropriately parameterized QMC calculation can meet this criterion given enough Monte Carlo iterations. For this

study, we want to consider whether single precision is satisfactory. To test this, three simple DMC calculations were performed on a large CPU cluster to compare numerically a result calculated in double precision with exactly the same calculation in single precision. First, a calculation is performed on a Helium atom using a 17s basis set [36] and a 2 determinant expansion in natural orbitals obtained using GAMESS [6]. Figure 3.3 shows that the single and double precision results are very similar, where the exact answer is approximately -2.903724 [37] hartrees. Second, the torsional barrier in ethane was studied using the cc-pCVTZ [38] basis set with CCSD(T) optimized Eclipsed and Gauche configurations [39]. Figure 3.4 again shows similar results between single and double precision, where the experimental value is 2.73 kcal/mol [39]. While these results are by no means conclusive, especially since the quality of the result is dependent upon the quality of the wavefunction, they provide evidence that single precision is not altogether unreasonable. This is can be seen since the iterates are decoupled to some degree from each other by random numbers, and since the Monte Carlo statistics itself happens in double precision. Furthermore, if a pathological electronic configuration is identified, it can always be more delicately handled on the CPU in double precision. Lastly, single precision QMC calculations might be useful in an independent VMC wavefunction optimization calculation. Since DMC only employs the wavefunction as a guide, variationally optimized parameters are far less restrictive in terms of precision.

As far as our nVidia 7800 GTX GPU is concerned, we studied the floating point error to obtain a best estimate for single point evaluations. We considered two principal sources of error relevant to our problem as compared to the level of error available on a CPU: underflow and effects of rounding. The evaluation of basis functions (Equation 3.6), for example, can easily underflow if the b_{n_j} are too negative. We investigated whether the lack of de-normals on GPUs was a problem since this means a GPU will underflow faster than a CPU. As regards rounding, the IEEE floating point standard calls for a relative error of $\pm 0.5 \times 10^{-7}$ in the basic arithmetic operations for single precision. On current GPUs the relative error in these operations appears to be [40] at least $\pm 0.5 \times 10^{-7}$ and $\pm 1.0 \times 10^{-7}$. For dense linear algebra, this yields a difference in error between CPU and GPU computed results.

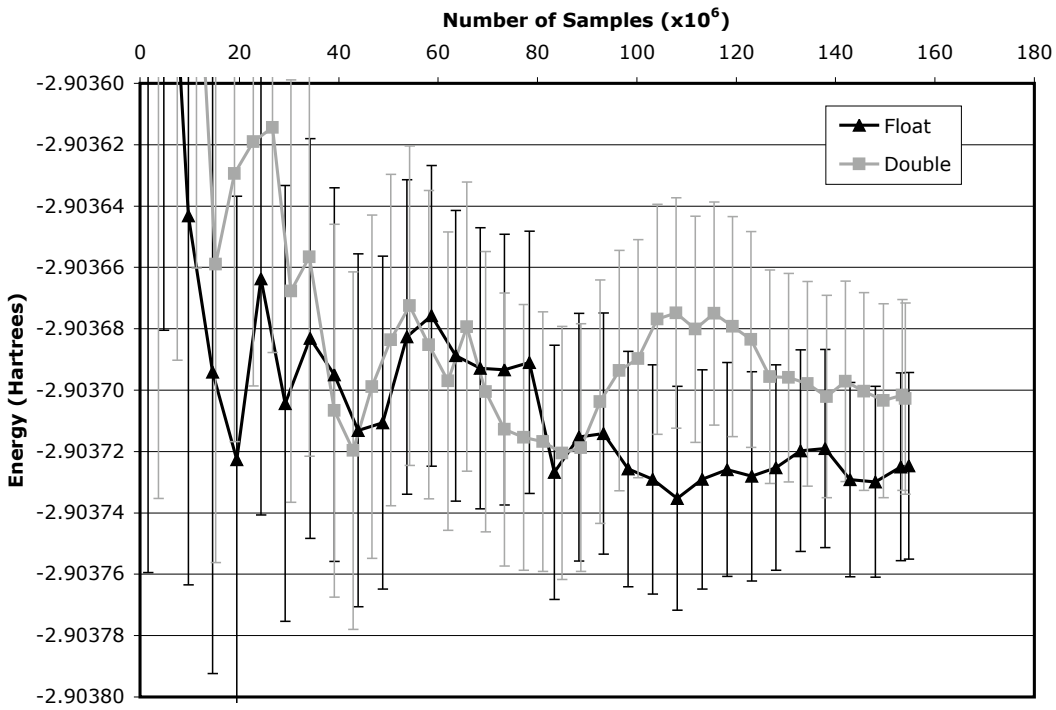

Figure 3.3: Helium calculation showing the average and the error as the calculation progresses. The calculation was done at dt = 0.001, with 200 walkers each on 128 CPU processors.

3.6.1 Underflow Corrections

To begin with, it is questionable whether one would permit de-normals to be included in calculations even on some CPUs. Many processor manufacturers elect software implementations of de-normals, which severely penalize the processing speed. Since we were unable to get decent timing results in matrix multiplication on the CPU unless de-normals were flushed to zero before multiplication, our performance comparisons actually already represent a lack of de-normals on both processors.

Basis function evaluation involves exponentials with arguments negative enough to cause underflow, an effect we do not want to ignore. To avoid underflow error one may simply scale relevant variables to avoid the de-normal range, but must do so carefully to avoid the worse problem of overflow. The effect of this type of error depends heavily on the distribution of parameters, which is highly specific to our application. Thus we measured the effect of these shifts on the final calculated $E_L(\bar{\boldsymbol{r}})$ for each iteration, compared to the same calculation as performed on the CPU in double precision.

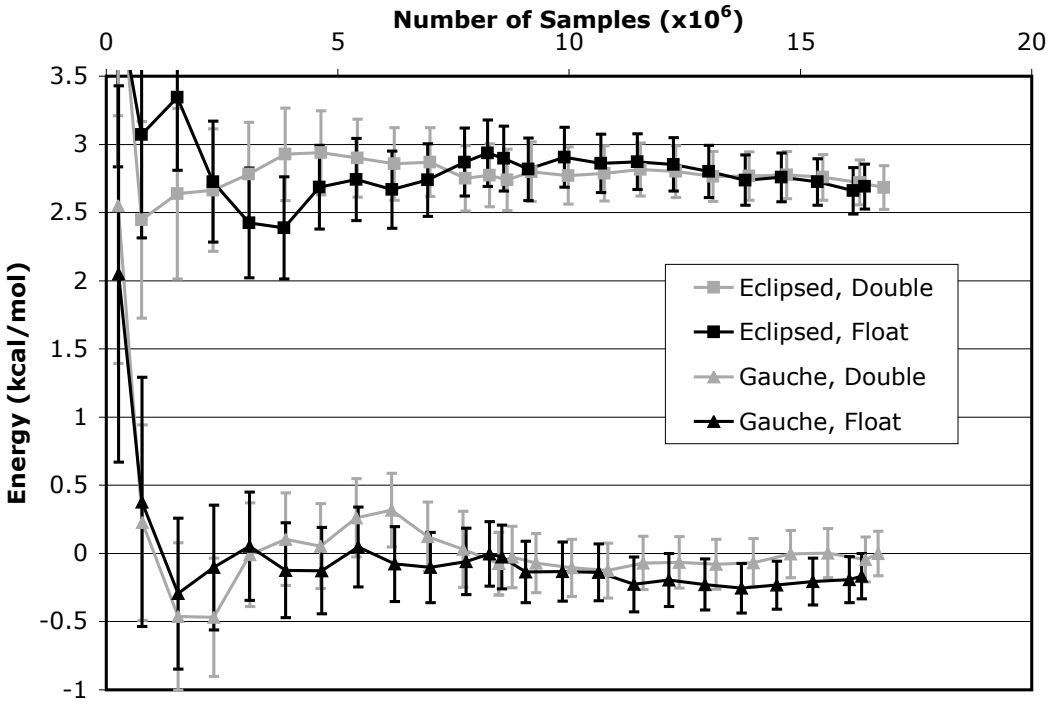

Figure 3.4: Ethane calculation showing the average and the error as the calculation progresses. The calculation was done at dt = 0.005, with 200 walkers each on 128 CPU processors.

The effect of shifting the exponential turns out to be relatively small for the set of parameters we considered. We conclude that shifting helps, but the lack of de-normals on the GPU turned out not to be a significant source of error. For parameter sets which consistently produce de-normals, single precision should probably be avoided entirely.

3.6.2 Kahan Method

Dense matrix multiplication is the most significant source of error in our computations when run on the GPU. Figure 3.5 shows the roundoff error inherent in matrix multiplication, as estimated by multiplying two matrices created with a uniform distribution of data. As a function of the dimension of the inner product, we calculate the relative error averaged over all the elements in the resultant 1000x1000 matrix using CPU double precision as our reference data. The problem is due to the propagation of errors, which scales approximately linearly with the length of the inner products. A CPU typically minimizes this by performing the calculations at a higher precision than the data type.

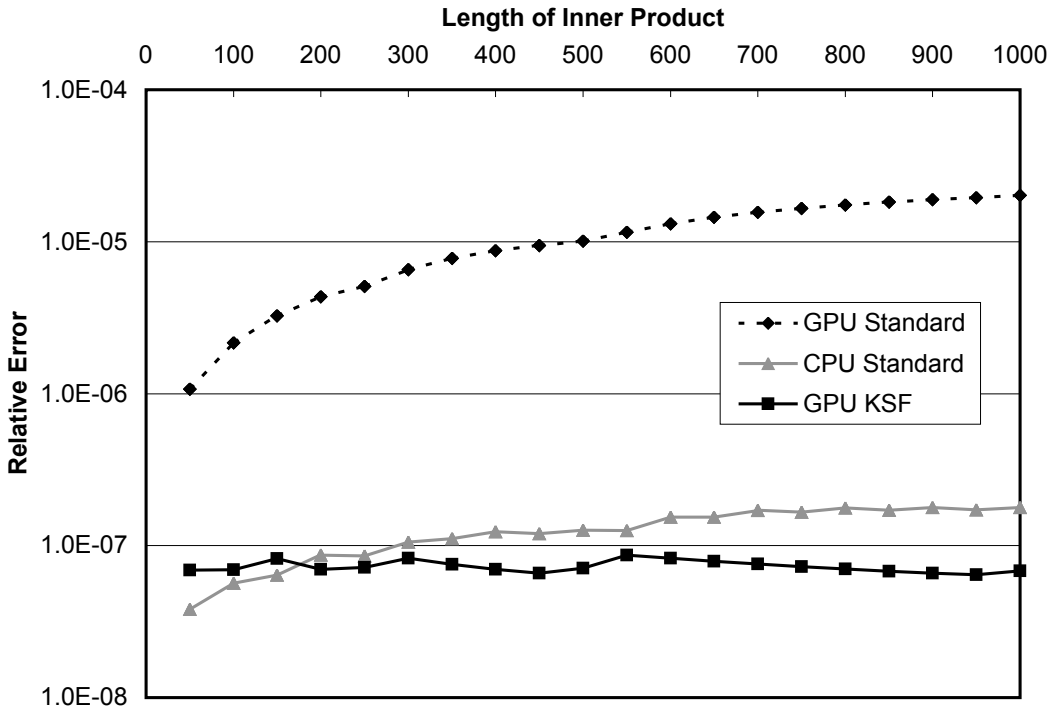

Figure 3.5: KSF corrects for rounding error in matrix multiplication. The resultant matrix is 1000x1000, and the operand data is sampled from a uniform distribution [0,1].

When summing a sequence of floating point numbers using the basic formula $\sum x_j$, the floating point result is $\sum x_j(1+\delta_j)$, where the perturbation error is defined as $|\delta_j| < (n-j)\epsilon$ and ϵ is the machine error. To compensate for the propagation of errors, we use the Kahan summation formula (KSF) [41, 42] in the context of matrix multiplication. This alternative method for summing a sequence of n numbers is shown below:

```
S = x[1];
C = 0;
for(j=2; j<=n; j++){
   Y = x[j] - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
}
```

This method is algebraically equivalent, but if these steps are preserved during compilation, the algorithm has the power to produce the result $\sum x_j(1+\delta_j) + O(n\epsilon^2)\sum |x_j|$, where

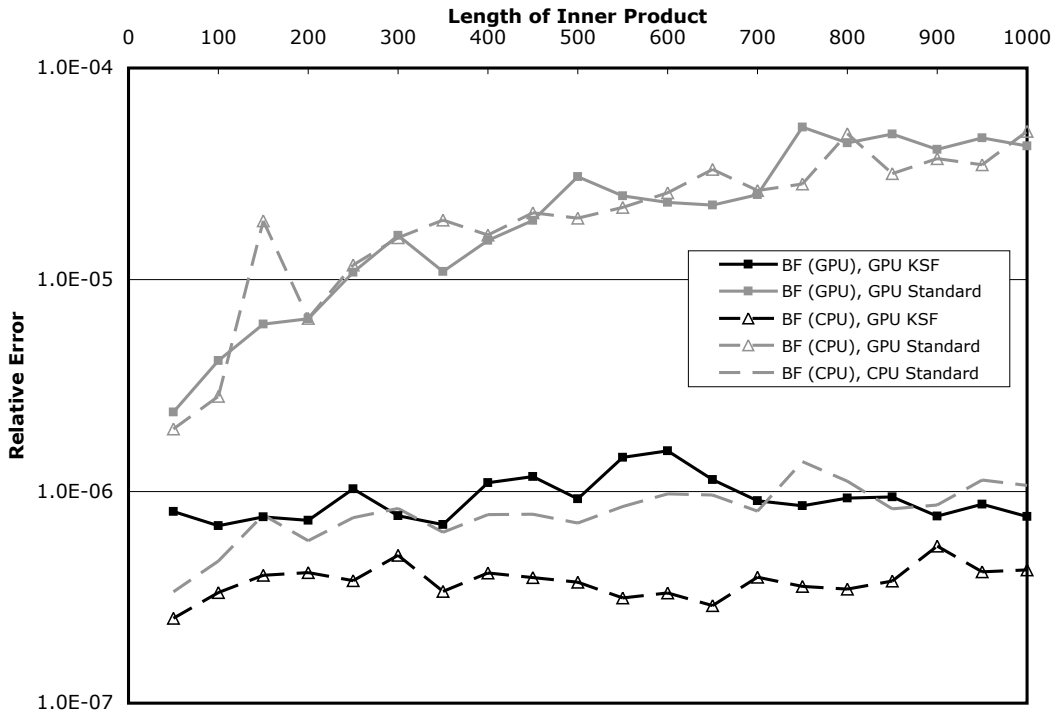

Figure 3.6: The "QMC-Distributed" data for the multipliers was generated either on the CPU or on the GPU, and the matrix multiplication was either corrected using KSF or left as the standard method

$|\delta_j| \leq 2\epsilon$ [43]. To explain this algorithm, one first observes that the low order bits of Y are lost when adding it to S. These bits can be recovered with the correction term C. The value for C is found by subtracting Y from the part of Y which is properly accounted for in the sum (the parenthesis are critical). This is not the only summation improvement available although it does compete well [44].

A simple modification makes the KSF suitable for use in matrix multiplications as shown in Algorithm 2. Here (i, j) represents the coordinates of the element in the product matrix we are working on. It is important to note that the propagation error in addition is corrected for, but not any error due to multiplication, even though such corrections are possible [45]. However, as Figure 3.5 shows, the improvement is enough to even beat single precision on the CPU for long enough inner products.

To estimate the improvement that KSF provides for our QMC methods, we move to a "QMC distribution" of data for our multiplier matrices while keeping the multiplicand (representing c_{jk}) as a uniformly random matrix. The distribution was formed by generating a representative set of basis function parameters and a pseudo-random configuration of

Algorithm 2 KSF-corrected GPU Matrix Multiplication

```
float4 T = 0, C = 0, Y = 0, S=0;
int j = 0;
while(j < N){
   Y = A[i,k].xxzz*B[k,j].xyxy - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
   Y = A[i,k].yyww*B[k,j].zwzw - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
   j++;
}
return S;
```

electrons. This distribution was evaluated either on the GPU or on the CPU and then sent to the GPU for multiplication. The relative error was again estimated against double precision on the CPU. Although the results in Figure 3.6 have a higher variance, it shows that using the KSF method, we are able to approximately obtain equivalent results as CPU single precision.

3.7 Results

To test the GPU port of our code, we sample 7 arbitrary molecules spanning the range over which we wish to measure performance. We present speedup estimates for the calculation time spent on equivalent tasks performed on both our 7800 GTX GPU and our 3GHz Pentium 4, as well as compare the final cost of incorporating the KSF correction. We ran the calculations long enough to converge the speedup ratio.

It is evident that for the range of molecules considered, the speed penalty incurred with KSF rose as the matrix multiplication cost became more prominent. The KSF formula served to keep the relative error in the calculated $E_L(\bar{\boldsymbol{r}})$ to a constant across all molecules at approximately $1x10^{-6}$. It is worth noting that KSF did not make a significant difference in either speed nor correction for many of the smaller molecules.

To provide an estimate for the impact of these speedup factors, we point out that for HMX, the calculation is now 5 to 7 times faster. This means that the new fractions

Name	Formula	Number of Electrons	Number of Basisfunctions	Total Speedup Standard	Total Speedup KSF	Basisfunction Speedup	Jastrow Speedup
Acetic acid	CH_3COOH	32	80	3.2	3.1	18.2	0.7
Benzaldehyde	C_6H_5CHO	56	150	4.4	4.1	25.9	2.1
[10]Annulene	$C_{10}H_{10}$	70	200	6.3	5.6	30.2	3.4
Diazobenzene	$C_{12}H_{10}N_2$	96	326	5.3	4.5	31.6	6.4
Lysine	$C_6H_{14}N_2O_2$	102	280	4.5	3.9	29.2	7.2
Arginine	$C_6H_{14}N_4O_2$	116	387	4.9	4.1	28.5	9.3
HMX	$C_4H_8N_8O_8$	152	516	6.6	5.3	33.3	14.0

Figure 3.7: QMC performance results on arbitrary molecules picked to represent varying problem sizes. Speedup is defined as the time spend processing on the CPU divided by the time spend processing on the GPU.

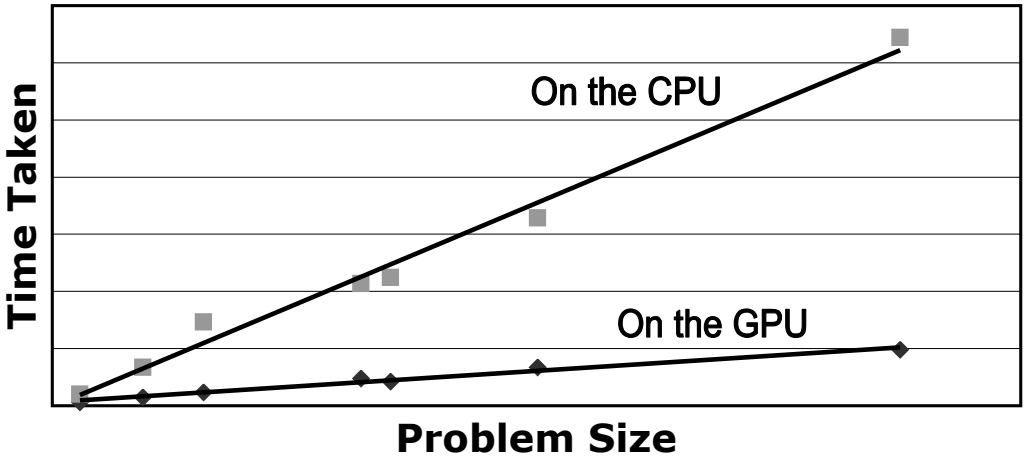

Figure 3.8: Problem size is defined as the number of basis functions × the number of electrons. The data points are from the arbitrary molecules listed in Figure 3.7

.

of evaluation cost are that matrix multiplication, which formerly composed 15% of the cost, is now only 4% (non-KSF) of the original total cost; the basis function cost went from 73% to 2.2%; and the electron-electron Jastrow evaluations, which used to cost 3.5% of the effort, are now 0.3%. If we approximate the effect of improving GPU technology over CPU technology as well as the possibility of multiple GPUs per CPU by setting the residual percentages at 0%, the original unaccounted for 8% suggests a theoretical factor of 13 speedup. A recent calculation [46] on free-base porphyrin which has 162 electrons and 938 basis functions in the cc-pVDZ basis set cost 40,000 CPU hours on an IBM SP POWER3+ cluster. Thus, ignoring the precision issue, we speculate that this calculation could theoretically cost 3,000 processor hours.

Although some of the performance numbers for the individual kernels are very good, the

code suffers from Amdahl's Law type inefficiencies because of diminishing returns discovered during porting. This is for several reasons. A few of the elements of the computation, like the Monte Carlo statistical manipulations, can not be permitted to be run in single precision. Furthermore, there are several portions of the code for which a GPU port is currently unsuitable due to a lack of sufficient data parallelism either as $O(N)$ components or as problems with GPU-unfriendly data interdependencies. With increasing capability on the GPU, more of the code will be available to porting considerations.

It is obvious however that there is a GPU kind of Gustafson's Law [47] advantage available. Specifically, if basis function and Jastrow function evaluations can be considered as essentially free, then one is encouraged to employ whatever functional form is deemed best, regardless of computational complexity. This is likely to increase both the quality of individual iterates as well as improve the overall convergence characteristics of a Monte Carlo calculation. Of course this assumes that these advantages are not washed out by precision errors stemming from other parts of the code.

3.8 Conclusion

QMC type algorithms for first principles chemistry calculations are simple to parallelize and capable of exploiting the *data parallel* aspects of GPU based computing. While the matrix sizes needed in actual application practice are on the small side, recent generation GPUs, coupled with a few tricks, have become significantly better in achieving high performance at these sizes. The overall result is a 3x to 6x speedup in the end to end simulation application with a modest increase in hardware cost, making this a very cost effective solution. The lack of full IEEE floating point support is perhaps the most critical issue for QMC. We were able to correct for the error propagation, albeit only with a performance penalty due to the more complex evaluation cost of the Kahan summation formula. Clearly a more complete IEEE floating point treatment would be an excellent improvement, and forthcoming improvements will be welcomed.

Beyond that, we note that due to the rapid evolution of GPU hardware (and the associated driver software), attaining a sweet spot in the performance landscape is a never ending quest of parameter and algorithm tweaking. We speculate that adoption of the GPU as a computational engine will be greatly facilitated if approaches such as ATLAS [34, 15] and

application specific libraries can be further brought to the GPU arena.

Chapter 4

Generalized Valence Bond Wavefunctions in Quantum Monte Carlo

4.1 Abstract

We present a comprehensive technique for using Quantum Monte Carlo (QMC) to obtain high quality energy differences. We use Generalized Valence Bond (GVB) wavefunctions, for an intuitive approach to capturing the important sources of static correlation. Using our modifications to walker branching and Jastrows, we can then use Diffusion Quantum Monte Carlo to add in all the dynamic correlation. This simple approach is easily accurate to within 0.2 kcal/mol for a variety of problems, which we demonstrate for the adiabatic triplet-singlet splitting in methylene, the vertical and adiabatic singlet-triplet splitting in ethylene, the ethylene twist barrier, and the 2+2 cycloaddition to make cyclobutane.

4.2 Introduction

The Quantum Monte Carlo (QMC) algorithm is rapidly advancing as a tool competitive with the best available *ab initio* electronic structure methods. It has already been used with remarkable success to calculate energies and other properties for a wide variety of molecules and periodic systems across the periodic table. Although it will probably never replace cheaper methods such as Density Functional Theory (DFT), given advances in computing power, it will surely begin to serve as a complementary method, brought in for calibration or to resolve disagreements.

The principle failure of Self-Consistent Field (SCF) methods is that they do not include all electron-electron interactions. The difference between the energy produced by an SCF method and exact energy is referred to as the correlation energy. Correlation energy can be further subdivided into two components; static correlation and dynamic correlation. Static correlation is the error resulting from using an incomplete functional form for the wavefunction during the SCF procedure, and is typically resolved by increasing the complexity of the wavefunction by adding more orbitals and basis functions to the SCF optimization. Dynamic correlation comes from the SCF procedure itself, where an electron sees only an average field of the other electrons, and thus never has to move out of another's way. This error is typically corrected after SCF with a Configuration Interaction procedure, in which determinants are added to the wavefunction by combinatorially choosing different orbitals for the electrons to occupy.

QMC methods can capture the correlation energy in two ways. First, a privilege shared with many Monte Carlo approaches, we are free to use whatever representation of the wavefunction we want, since we never need to analytically integrate anything. That is, we can add purpose-designed functions, called Jastrow functions, to explicitly model inter-particle interactions. Second, and even better, provided a guess for the wavefunction nodes, we can include all the dynamic correlation energy through the diffusion Monte Carlo (DMC) algorithm. The nodal assumption results in an error called the fixed-node energy, which is not negligible. Fortunately, the same techniques used to deal with the static correlation energy can be used to lower the fixed-node energy, and thus multi-configuration SCF (MCSCF) techniques can be considered to be quite complementary to DMC.

Fully accurate SCF techniques can be expensive, typically scaling quite poorly with molecule size, motivating a search methods which do not overkill the problem. On the other hand, we need at least within chemical accuracy of 1 kcal/mol, so underkill is undesirable. Explored in this paper is an evenkill solution where we use Generalized Valence Bond (GVB) wavefunctions [48] to correct for the fixed-node error. By working with valence bond orbitals, GVB has the advantage over more general approaches of being chemically intuitive and of scaling well with molecule size, while efficiently correcting for the important sources of static correlation.

To demonstrate the validity of the GVB approach, as well as to validate our overall methodology, we present a study of a few molecules for which experimental data or reliable

calculations are available, testing excitation energies and bond breaking. The methylene triplet-singlet adiabatic splitting is one of the few processes for which experimental data is available, accurate to tenths of a kcal/mol. Thus it is a good test to see exactly how close to the exact answer we can get. Among the most studied processes is surely the ethylene singlet-triplet splitting (both adiabatic and vertical), with quite a few experimental and computational studies. Both of these processes have even been the subject of other QMC studies, providing an excellent basis on which to compare our results with those of more standardized approaches. We go further than this with ethylene, examining the energy barrier of a rotation about the CC axis, which breaks the double bond. Lastly, we look at the cycloaddition of two ethylene molecules to make a cyclobutane molecule and show how our approach is successful at modeling multiple bond changes at once.

4.3 Method

In this section, we present the QMC approach we use in our QMcBeaver [4] code, which is available online. First, we discuss our choice of trial wavefunction, which is to use GVB for the SCF part of the wavefunction, and second, our modifications to the Jastrow functions recommended by Drummond and co-workers [49]. Third, we talk about our experiences in optimizing this kind of wavefunction, starting from the approach of Toulouse and Umrigar [50]. Fourth, we diverge from Umrigar's DMC algorithm [1] to use the reconfiguration method for walker branching provided by Assaraf and co-workers [51], with more of our modifications. Finally, we summarize our approach.

4.3.1 Generalized Valence Bond Wavefunctions

A GVB wavefunction [48] starts with a localized restricted Hartree-Fock (RHF) wavefunction and replaces an orbital (e.g., a single bond) with two singlet paired orbitals in a geminal called a perfect pair

$$\Psi_{GVB} = \mathcal{A}\left[\{core\}\{\varphi_u\varphi_v\}\{\alpha\beta - \beta\alpha\}\right], \tag{4.1}$$

where $\mathcal{A}$ is the antisymmeterizer, or determinant, operator. Although we allow φ_u and φ_v to overlap each other, they are orthogonal to all the other orbitals in the wavefunction. This can be thought of as permitting each electron to have its own orbital. We can rotate

these intuitive orbitals into the more computationally useful, but fully equivalent, natural orbital form:

$$\Psi_{GVB} = \mathcal{A}\left[\{core\}\left\{\sigma_u\phi_u^2 - \sigma_v\phi_v^2\right\}\{\alpha\beta\}\right], \tag{4.2}$$

where $\sigma_u^2 + \sigma_v^2 = 1$. We typically interprete ϕ_u as a "bonding" orbital, and ϕ_v as an "antibonding" orbital. Where a perfect pair is used to represent a single bond, the benefit is to add left-right correlation to the bond, allowing the electrons to get away from each other a little bit, and this is the simplest wavefunction that permits H_2 to dissociate to 2H. In the same way, we can add left-right correlation to double or triple bonds. When it comes to lone pairs, the perfect pairing scheme can be used to add in an important orbital left out by RHF (such as 1 b_1 in $^1A_1$ methylene) to incorporate some angular correlation, or, in other cases, to add in-out correlation to the lone pair.

Although GVB is a subset of MCSCF calculations, the main advantage to GVB over MCSCF is that it is the only variety that is able to avoid integral transformations [52]. But additionally, it allows a simple, modular, and balanced way of selecting the active space, since everything is localized. The researcher perhaps does not even need to look at any orbitals to do this, since reliable routines exist to generate good initial guesses [53] for a GVB wavefunction based on RHF orbitals.

For our QMC wavefunctions, we expand the geminals in each N_{GVB} pair wavefunction into the equivalent $2^{N_{GVB}}$ determinant wavefunction. Although the number of determinants grows quickly, we use a simple algorithm to sort these determinants such that sequential determinants in the wavefunction differ by only one column (orbital). To calculate the local energy of the wavefunction, the algorithm only needs to perform one Sherman-Morrison update per determinant in the wavefunction. This is a significant performance boost where many pairs are used.

All of the cases we present here are adequately modeled with perfect pairing. However, for increased accuracy in some of our calculations, we can add Restricted Configuration Interaction (RCI) terms [54] to the GVB reference wavefunction, without reoptimizing the orbitals. With these terms, the GVB-RCI geminal now takes the "excited" form $\left\{\sigma_u\phi_u^2 + \phi_u\phi_v - \sigma_v\phi_v^2\right\}$, adding some charge-transfer character in the pair. Although we could add these RCI terms to all geminals, for a total of $3^{N_{GVB}}$ determinants, we excite only up to 2 geminals at per determinant.

4.3.2 Length Scaled Jastrows

We implemented the 2 and 3 particle Jastrow functions recommended by Drummond and co-workers [49] because we like the cutoffs, flexible shapes, and simplicity. However, we found that their length scale parameter L was too difficult to optimize for the algorithms we use, so we use the following modifications instead. For 2 particle interactions, we use the functional form

$$u_{ij}\left[x \leftarrow r_{ij}S\right] = (x-1)^3\left(\sum_{k=0}^{M} a_k x^k\right), \quad \text{if} \quad 0 \leq \mathrm{x} \leq 1 \tag{4.3}$$

$$= 0, \quad \text{if} \quad \mathrm{x} > 1, \tag{4.4}$$

where r_{ij} is the distance between the two particles (electrons or nuclei) i and j, S is the length scale parameter ($x = r_{ij}S$), and a_1 is constrained to satisfy the cusp conditions. The $(x-1)^C$ prefactor is used to force the $C-1$ lowest order derivatives to go to zero at the cutoff. We have found that $C < 3$ inhibits the optimization of S using our routines, and that $C > 3$ does not make much difference. Our three customizations are that the function uses the scaled coordinate x instead of r, we optimize $1/L$ instead of L, and we only use $C = 3$. These do not change the variational flexibility of the function, but they make the a_k parameters less dependent on S, easing their optimization. This makes a total of $M+1$ independent parameters, and in all calculations presented here, we use $M = 8$. Optimizing the a_k parameters was still delicate during concurrent optimization with S, so we eventually turn off the optimization of S for some final fine tuning, as discussed in Section 4.3.3. We make analogous modifications to their electron-electron-nuclear Jastrows for our software.

Our tests did not indicate that differentiating between spin for electron-nuclear Jastrows significantly changed the energy, so we use the same Jastrow for all electrons. For hydrocarbons, then, we use four 2 particle Jastrow classes: Carbon-Electron, Hydrogen-Electron, Opposite-Electron, and Parallel-Electron. Adding the 8 parameters for the Jastrow's polynomial and the 1 length scale parameter, there are 9 parameters for each 2 particle Jastrow, for a total of 36 parameters for 2 particle Jastrows in all of our calculations.

Similarly, we ignore spin distinctions in our 3 particle Jastrows, leaving us with only one 3 particle Jastrow per element represented in the molecule. Although there are 4^3 terms of the form $x_i^a x_j^b x_{ij}^c$ in the polynomial for 3 particle Jastrows, there are several necessary

constraints including symmetry and cusp conditions. Thus, the number of independent parameters is reduced considerably to only 27 parameters, including the length scale, per Jastrow class. As a further simplification, we have found 3 particle Jastrows centered on Hydrogen atoms to be unhelpful. This makes physical sense given that these Jastrows are primarily useful for modeling the interaction of two 1s electrons with the nucleus, and on average only 1 electron will be near a Hydrogen nucleus.

With minimalistic Jastrows added to single determinant wavefunctions, we estimate that Jastrow function evaluation uses 10% or less of the time spent during a QMC calculation. With the addition of 3 particle complexity to Jastrows, however, this fraction can increase to 70% or 80% or higher. In the future, however, we believe [55] that SIMD computing technology in devices such as GPUs will eliminate the cost (comparatively) of Jastrow evaluation. In the mean time, however, it is important to seek practical short cuts.

4.3.3 Wavefunction Optimization

To optimize our wavefunctions, we use the method recommended by Toulouse and Umrigar [50], with the following modifications. To make a wavefunction, we copy into our input file the best Jastrows we have from among similar systems, noting that it is more important that we match the basis set than the type of SCF wavefunction. If we found that two CI coefficients were the same (or additive inverses) to within a relative difference of 10^{-5}, we constrained them to maintain the relationship. Furthermore, even though QMC is insensitive to the normalization of the wavefunction, we do not take the opportunity to eliminate a degree of freedom in the CI coefficients. Starting at around 20,000 samples per optimization step, we double the number of samples collected per iteration, with a maximum of 500,000 samples, if the variational energy does not go below the statistical error between successive iterations. Umrigar makes use of an a_{diag} factor to stabilize the eigenvector from the solver. Just as he does, we obtain this factor on the basis of a short correlated sampling run in between optimization steps. Our correlated sampling runs are produced using the best optimized wavefunction from the previous iteration as the guiding trial function, and including 7 wavefunctions produced with preselected a_{diag} factors, logarithmically spaced between 10^{-7} and 10^{3}. The larger a_{diag} is, the less the wavefunction will change as compared with the previous iteration. The wavefunction for the next step is chosen from the 7

by selecting the one i with the lowest quantity:

$$0.95(E_i - E_0) + 0.05(\sigma_i^2 - \sigma_0^2)/\sigma_0^2,$$

thus optimizing for lowering the energy compared to the guiding trialfunction, indexed at 0, while penalizing a wavefunction with too large of a sample variance. With this scheme, if an optimization step goes bad, the step can effectively be ignored by choosing $a_{diag} = 1000$.

There are two problems with this procedure applied to our Jastrow functions. First, despite our improvements, the length scale parameter remains a source of instability. Thus once we observe the length scale to be changing by less than a few percent, we turn off its optimization, allowing us to fine tune the other Jastrow parameters. Second, the algorithm occasionally leads to a local minima. Some of our wavefunctions, for example the $^3B_1$ methylene wavefunction, initially optimized to an absurd parallel spin Jastrow, which was only discovered upon examining a plot of the Jastrow itself. In these cases, neither the energy nor the variance were suspicious, since after all, we did not know how deep the global minima goes. The problem is that some of the local minima we found raised the VMC energy by about a few kcal/mol. For DMC, this is not a problem upon time step extrapolation, but we are not doing time step extrapolation as discussed in Section 4.3.5. For this reason, and since the CI coefficients might be affected by poor Jastrows, we carefully monitored our Jastrows during optimization.

Once satisfactorily optimized, all of the Jastrows within each class looked qualitatively very similar. A few examples are plotted in Figure 4.1, exponentiated. With this in mind, we were easily able to identify bad Jastrows as ones which cross the $\exp(u_{ij}) = 1$ line, which were not monotonic, or which took on extremely high or low values. In some exceptional cases, the global minima was only obtained by first optimizing all Jastows except the troublesome one, constraining it to a good Jastrow from another system. Once that converges, we optimize the troublesome Jastrow (and possibly its length scale) holding all the others fixed. We repeat this cycle until all of the Jastrows are sufficiently close to the global minima that concurrent optimization of all the Jastrows can lock it in. There were not many cases like this, but this problem casts doubt on the rest of our optimization efforts, especially for the CI coefficients which we can not monitor visually.

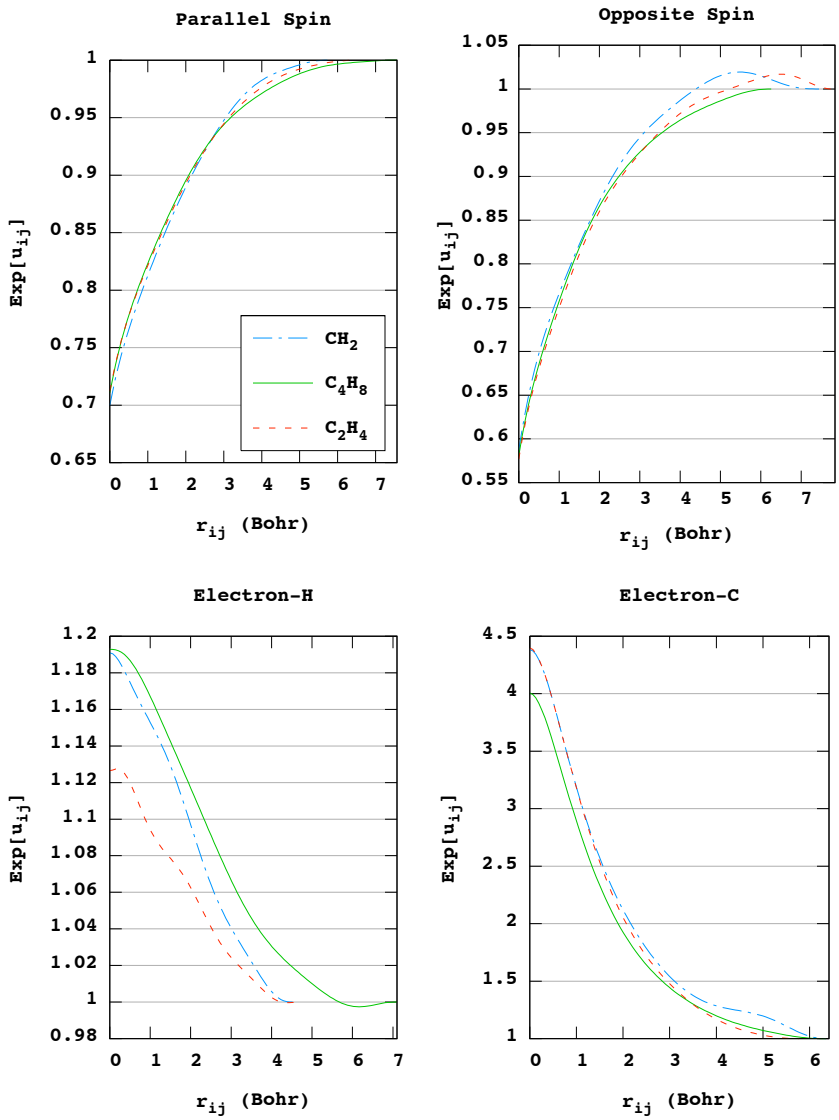

Figure 4.1: Typical ground state Jastrow functions used in this study, for the aug-tz basis set. Based on our experience, we do not believe that any Jastrows would look significantly different than these. In our optimization, we ignored minor flaws in the Jastrows, such as wiggles in the Electron-H Jastrows, or the brief crossing of the $\exp(u_{ij}) = 1$ line in Opposite Spin Jastrows.

4.3.4 Walker Reconfiguration

There are a variety of ways to design the branching process such that the number of walkers is always constant, and we use the algorithm designed by Assaraf and co-workers [51], which we dub ACK reconfiguration. This is made possible with a reconfiguration step, where low weight walkers are replaced with duplicates of high weight walkers. This is done by calculating the average walker weight W_{avg}, and using W_{avg} to bifurcate the list of walkers. We delete a total of

$$N_{replacements} \propto \sum_{i \in \{w_i < W_{avg}\}} \left| \frac{w_i}{W_{avg}} - 1 \right| \tag{4.5}$$

walkers, where a walker with weight w_i is selected with probability proportional to $|w_i/W_{avg} - 1|$. The same proportionality relation is used to select enough high weight walkers for duplication, so that the total number of walkers is restored. After this, the weights of all walkers are set to W_{avg}, so that the total weight of the walkers is also unchanged. This method adds significant stabilization to the ordinary DMC process since any instabilities affecting one walker are instantaneously disbursed to the others.

We add further stabilization to the method, partly because of the added instability of our all-electron move iterations. This is done simply by selectively ignoring in the duplication and elimination candidate lists walkers which fail our criteria. That is, we keep W_{avg}, the probabilities, and $N_{replacements}$ the same as prescribed. The only difference is that the actual length of either of the two lists might be different than the ACK algorithm predicted. The penalty for this is that in rare cases, the algorithm will be unable to maintain the same number of walkers it started with. We modify our elimination lists to ensure that walkers with $w_i < 10^{-5}$ are guaranteed to be replaced this iteration, since they are a complete waste of computational effort. Defining age as the number of iterations since the walker last moved and dW as the multiplicative factor by which the weight changed this iteration, our acceptable duplication criteria are:

1. $age > 4$,
2. $\mathrm{pow}(dW, age + 1) > 5$,
3. or if the walker has not been duplicated this iteration.

Persistent walkers, those stuck in one location, can be a problem in a Monte Carlo calculation. Our improvement is to ensure through Criteria (1) that at least these walkers never become duplicated. Duplication will also be prohibited by Criteria (2) if a slow walker is in a location where its weight grows too fast. The reason is that we have found that some walkers can become stuck close to a wavefunction node, which is a singularity in the local energy, where they often spawn more quickly than they can move away. Lastly, with Criteria (3), we do not allow a walker to duplicate more than once per iteration, a fail-safe to slow the damage that one walker might cause.

4.3.5 Further Details

To make our wavefunctions, we have used both Jaguar [5] and GAMESS-US [6], and we obtained our basis sets from the EMSL website [56, 57]. For this study, we have chosen two basis sets, which we label aug-tz, and tz. Our aug-tz basis set is aug-cc-pwCVTZ, which is Peterson and Dunning's new [58] weighted basis functions, which were optimized with the inclusion of some core-core correlation energy for better overall performance. This basis set puts 25 basis functions from [4s3p2d] on H, and 69 basis functions from [7s6p4d2f] on C. We also use cc-pCVTZ, labeled here as tz, which uses their recent [38] scheme for adding core-valence correlation. This basis puts 15 basis functions from [3s2p1d] on H, and 49 basis functions from [6s5p3d1f] on C. All Hartree-Fock, coupled-cluster [59], and GVB [48], and MCSCF results were obtained in GAMESS using the same geometry as the corresponding QMC calculation. We included all determinants in the CI expansion, except where noted. All of our DFT calculations were done using Jaguar with high precision settings. We include results using the LDA, PBE, B3LYP, and the m06-2x [60] functionals, using the same geometries as the corresponding QMC wavefunctions. We used Jaguar to make our GVB wavefunctions, since it has a good mechanism for generating initial guesses [53]. But any wavefunctions that we used were handed over to GAMESS for final convergence since Jaguar restricts us to 7 f basis functions, and we want to use all 10 cartesian functions.

Our QMC calculations are done using the QMcBeaver [4] software developed in our group. The C++ source code is available online under the GNU Public License. Starting with a script generated input file based on an SCF calculation and similar Jastrows, we use our own efficient algorithm [61] to initialize the walkers. We evaluate the local energy in all-electron updates, using the cusp replacement algorithm of Ma and co-workers [62]. We

use Variational Quantum Monte Carlo to optimize all CI coefficients and Jastrows by the method recommended by Toulouse and Umrigar [50], with our modifications as outlined in Section 4.3.3. Using the resulting optimized parameters, we run Diffusion Quantum Monte Carlo based on Umrigar's seminal algorithm [1], with our modifications as described here. Our calculations are run on 4 CPU cores, for a total of 400 walkers, using a different parallelization [63] technique than is typical for QMC calculations. All energies reported have been fully decorrelated using our efficient algorithm [64], which automatically finds the smallest decorrelated block size.

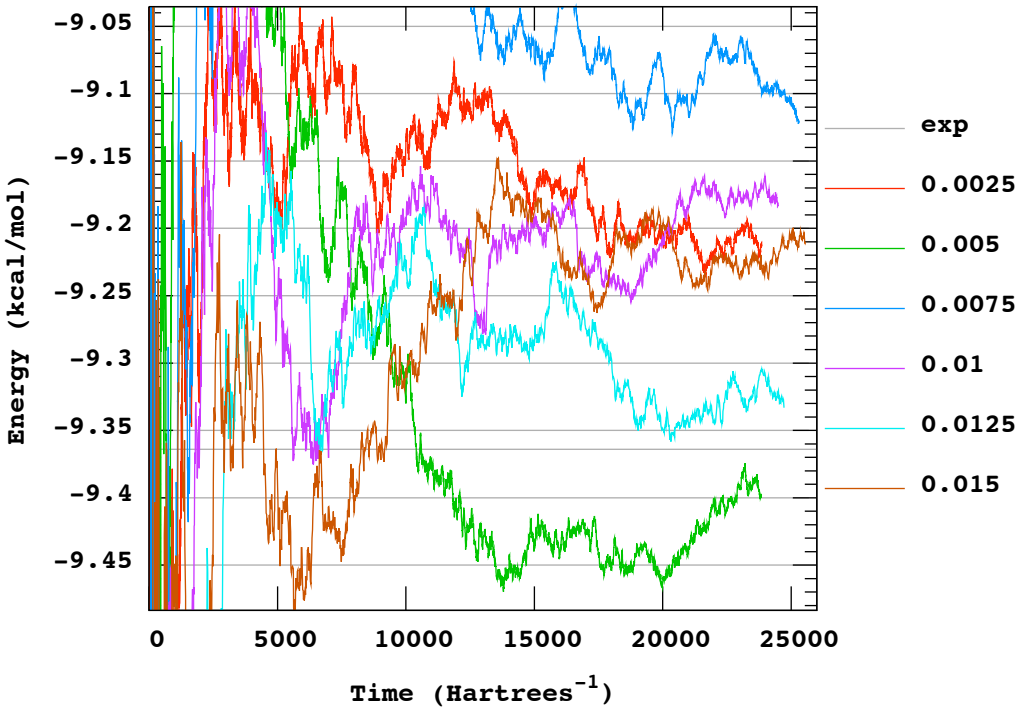

Figure 4.2: Cancellation of time step error between triplet to singlet energies in methylene, using 3 pair delocalized GVB wavefunction. For this plot, individual calculations were stopped when they reached a statistical error of exactly 0.065 kcal/mol, corresponding to an error of 0.092 kcal/mol for the difference. We plot the differences here against the amount of simulated time, iterations $\times$ time step $\times$ average move acceptance probability.

Based on the results shown in Figure 4.2 and other comparisons we have done not included here, we can see that the majority of the time step error cancels off for each time step. This indicates that the dominant source of error is not the time step error itself, but an instability on the order of a few tenths of a kcal/mol. With this in mind, the consensus result appears to be about 9.2(1) kcal/mol. Since the computational cost of the

calculation scales linearly with the time step, we are motivated to choose just one time step, as large as reasonable. We can also see that after running for about 15,000 au^{-1}, most of the calculations have converged to within the 0.092 kcal/mol statistical error. Based on this observation, we typically choose a time step of 0.0075 and run for 20,000 au^{-1}, which corresponds to 2.7 million iterations. Looking ahead at Table 4.1, our converged result is 9.239(88) kcal/mol for this case, in agreement with our qualitative assessment of Figure 4.2.

The length of time for each calculation varied with many factors, but ranged from about 40 hours on methylene to about 100 hours on ethylene to about 400 hours on cyclobutane. However, for these same calculations, each processor only required about 15 to 40 megabytes of RAM* each. It is illustrative to compare these performance numbers with coupled cluster methods, which not only scale poorly in computation time with larger molecules, but scale poorly in memory requirements as well. Even if a researcher is willing to wait for completion, memory is certainly a finite resource, and random access memory will remain a bottleneck resource for the foreseeable future. In contrast, even though QMC scales somewhat poorly in computation time at $\mathcal{O}(N^3)^\dagger$ with a large prefactor, where N is the number of electrons, the memory requirements are negligible. This is a favorable situation since machines are rapidly getting faster, and it is even possible to run QMC on a Graphical Processing Unit [55] for remarkable speedups.

4.4 Results

We present our results for several related molecules for which good experimental or computational results are available to use as a reference. We wish to examine the effectiveness of adding GVB pairs to our wavefunctions, as well as the importance of different basis sets. In this section, we examine methylene, ethylene, and cyclobutane.

4.4.1 Methylene

The singlet-triplet splitting in methylene is among the most studied problems in quantum chemistry. It has been notoriously difficult to get correct results for, and thus it remains a very useful benchmark for QMC. The 2s and 2p atomic orbitals on Carbon are nearly

*Low memory requirements are one of the benefits of all-electron updates.

†assuming dense matrices

degenerate, necessitating the inclusion of all 4 into any Carbon containing molecule. Any $^3B_1$ wavefunction does this, while one orbital is left out at the RHF level for $^1A_1$. Thus the simplest reasonable description of the $^1A_1$ state is to add the missing orbital by perfect pairing it with the lone pair as an angular correlation term. It is important to also recognize that triplet paired electrons are much better correlated, due to orbital orthogonality, than closed shell counterparts in a singlet wavefunction. Consistency requires at least that the number of orbitals on each side of a comparison is the same, adding another reason for the perfect pairing.

We present our results in Table 4.1. Our GVB-1 calculations represent RHF for the triplet state, and one GVB perfect pair for the singlet, indicating our policy of using the label from the comparison with the highest number of pairs. The GVB-3 level adds correlation to the bonds, for a total active space of 6 orbitals, and there are two ways to do this. GVB-3 is supposed to use localized bonding and anti-bonding orbitals, but we also include a version with the same 4 orbitals delocalized, even though the GVB splittings are 0.02 kcal/mol different. The RCI-3 level of theory excites up to two perfect pairs into their corresponding open shell singlet, without optimizing the orbitals. Finally, by CAS-3, we mean the complete active space in the 6 orbitals, optimizing the orbitals in SCF. There is some question about which zero point energy (ZPE) we should use since we see two values used in the literature to convert the experimental [65] T_0=3147 $\pm$ 5 cm^{-1} to T_e. First, we find that many people use ΔZPE=68 cm^{-1} to produce T_e=9.192(14) kcal/mol, a ZPE derived [66] by fitting a potential energy surface to reproduce experimental excitation energies. We also find a theoretical ΔZPE=128 $\pm$ 18 cm^{-1} obtained [67] with accurate quartic force fields leading to T_e=9.364(53) kcal/mol. We use the latter value for our comparisons. We also note that in contrast with other theoretical studies of this system, we do not incorporate any other energy corrections to our measurements.

For methylene, we have run each calculation shown in Table 4.1 twice so that we can average some of the instabilities out, a luxury we do not employ for our other molecules. Additionally, one of these two runs for our GVB-3/aug-tz was run for much longer, since we were surprised that the localized orbitals are further from experiment. This error is compensated for at the RCI-3 level. All of our results are within 0.4 kcal/mol of the experimental estimate, with the exception of our RHF calculation which does quite poorly at an error of about 4 kcal/mol. Additionally, we include our estimation of the lowest

Table 4.1: Methylene excitations: $^1A_1 \leftarrow ^3B_1$ and $^1B_1 \leftarrow ^3B_1$. For $^3B_1$, $[R_{CH},\Theta_{HCH}]$ = [1.0753Å,133.93] from experiment [65], for $^1A_1$ [1.107Å,102.4] from experiment [68], and for $^1B_1$ [1.0723Å,142.44] from theory. [69]. By 'B', we are indicating our basis, by 'O' we are indicating, where it matters, whether our GVB pairs are localized or delocalized, and by 'J' we are indicating whether we are using 2 or 3 particle Jastrows.

SCF	B	O	J	Δ_e	$^1A_1$	$^3B_1$
				kcal/mol	au	au
GVB-3	aug-tz	L	2	9.071(80)	-39.121669(91)	-39.136124(89)
GVB-3	aug-tz	D	2	9.239(88)	-39.120847(81)	-39.13557(11)
GVB-3	aug-tz	D	3	9.340(71)	-39.124461(79)	-39.139345(82)
Exp[a]				9.364(53)		
RCI-3	aug-tz	L	2	9.37(11)	-39.12176(13)	-39.13670(12)
GVB-1	aug-tz		2	9.40(10)	-39.12149(12)	-39.13648(12)
RCI-3	aug-tz	D	2	9.519(95)	-39.12137(11)	-39.13654(11)
GVB-3	tz	D	2	9.53(10)	-39.12065(11)	-39.13584(12)
GVB-3	tz	D	3	9.557(74)	-39.123975(83)	-39.139205(83)
GVB-1	aug-tz		3	9.560(76)	-39.124248(91)	-39.139483(80)
GVB-1	tz		2	9.65(11)	-39.12093(12)	-39.13631(12)
GVB-1	tz		3	9.673(73)	-39.123838(84)	-39.139253(81)
CAS-3	aug-tz		2	9.792(92)	-39.12353(10)	-39.13913(10)
RHF	aug-tz		2	13.80(10)	-39.11449(12)	-39.13648(12)
RHF	aug-tz		3	13.844(73)	-39.117421(85)	-39.139483(80)
SCF	B	O	J	Δ_e	$^1B_1$	$^3B_1$
				kcal/mol	au	au
PES[b,c]				31.897		
GVB-1	aug-tz		2	32.06(11)	-39.08539(12)	-39.13648(12)
GVB-1	aug-tz		3	32.114(71)	-39.088306(80)	-39.139483(80)
MRCI[c]				32.807		

a) Experimental $T_e = T_0 + \Delta ZPE$, where T_0 [65]=3147 ± 5 cm^{-1} and ΔZPE [67] =128 ± 18 cm^{-1} b) From Ref [[66]] c) From Ref [[69]]

singlet-singlet vertical excitation, even though there is little consensus for what the right answer should be. Adding augmented basis functions improves our estimates by 0.1 to 0.2 kcal/mol, while 50% more basis functions added computational time of only 10% to 30%. There is no reason not to use the augmented version of the chosen basis set class. Looking at our timing data, we see that if we had stopped our calculations at an error of 0.1 kcal/mol, our 3 particle wavefunctions would have finished 30% to 40% quicker, demonstrating their value in variance reduction. This comparison encourages their use, but this conclusion changes for cyclobutane.

It is clear that beyond the statistical error, there are some additional sources of error. As mentioned previously in reference to Figure 4.2, there is some error due to instability in the convergence, which we have attempted to minimize for methylene by running each calculation twice. But more importantly, there appears to be some error due to incomplete optimization of wavefunction parameters. For example, using our tz basis set, the addition of 3 particle Jastrows does not appreciably change the energy difference, a result which makes sense given our assumption that the time step error cancels out. This is not the case for our aug-tz basis set, which changes by at least 0.1 kcal/mol with the addition of 3 particle Jastrows. We are also puzzled by our CAS-3 results. In this case, our first optimized wavefunctions produced a DMC splitting of 9.877(92) kcal/mol, which is clearly wrong. We returned to the optimization stage, keeping the optimized Jastrows but starting with the original CI coefficients, and this time we improved to 9.792(92) kcal/mol. This indicates that we eliminated a local minima in the wavefunctions worth 0.085 kcal/mol. We also tried using a determinant cutoff of 0.01 so that there were fewer parameters to optimize, but this produced 10.291(94) kcal/mol. Clearly, there is no fundamental flaw with CAS wavefunctions themselves, which work quite well for us in ethylene. But this leaves us in a precarious balance where theoretically better wavefunctions are perhaps more likely to fall into local minima during optimization.

We wanted to discover the effect of optimizing different parts of the wavefunction. We pursued this by choosing some standard state for each atom for the Jastrows, and then selectively optimizing parts of the wavefunction, and comparing these results to the comparable result from Table 4.1. Our results, shown in Table 4.2, tell us that optimizing the CI coefficients was worth 0.5 kcal/mol, and that optimizing the electron-nuclear Jastrows was worth another 0.4 kcal/mol. Of course, in the limit of zero time step, there should

Table 4.2: The effect of optimizing different parts of our aug-tz GVB-3 delocalized methylene wavefunctions, with 2 particle Jastrows, and all calculations run at 0.0075 time step. The starting point for these calculations are the CI coefficients from GVB, Electron-Carbon and Electron-Electron Jastrows from optimized Carbon GVB-1 atom, and Electron-Hydrogen Jastrows from an optimized GVB-1 H_2 molecule. Each row corresponds to a different set of parameters which were optimized, where EN stands for electron-nuclear and EE for electron-electron.

Optimization	Δ_e	$^1A_1$	$^3B_1$
	kcal/mol	au	au
EN and CI	9.20(11)	-39.12184(13)	-39.13650(12)
Fully Optimized	9.239(88)	-39.120847(81)	-39.13557(11)
EE and CI	9.51(11)	-39.12133(13)	-39.13648(12)
CI	9.58(11)	-39.12161(13)	-39.13687(12)
EE and EN	9.97(11)	-39.11872(12)	-39.13460(12)
EE	10.05(11)	-39.11801(12)	-39.13403(12)
No Optimization	10.08(11)	-39.11806(13)	-39.13412(11)

only be two results in this table, since in that case Jastrows should not matter, so much of the error here can be called time step error. But it appears to be crucial that we optimize the CI coefficients. Returning to the question of the CAS discrepancy, we tried the experiment of optimizing all the Jastrows, while keeping the original CI coefficients. This produced 15.80(11) kcal/mol as the DMC energy splitting, the worst of all the results we have obtained. Since this represents the comparison of the SCF functions directly without worrying about whether the VMC optimization is falling into local minima, we can see that given our methodology, a better SCF wavefunction does not always improve accuracy. In separate investigations, we have found that CAS wavefunctions are necessary.

Table 4.3: Methylene excitations using single determinant wavefunctions. Using our aug-tz basis set, we obtained orbitals from RHF or DFT, and added 2 particle Jastrows.

	Δ_e	$^1A_1$	$^3B_1$
	kcal/mol	au	au
CAS-3	13.76(10)	-39.11499(13)	-39.13693(10)
RHF	13.80(10)	-39.11449(12)	-39.13648(12)
B3LYP	14.05(11)	-39.11539(12)	-39.13778(12)
LDA	14.39(15)	-39.11481(18)	-39.13773(16)
PBE	14.64(17)	-39.11527(24)	-39.13860(12)

Another interesting consideration is whether we could use DFT orbitals. In Table 4.3, we present our results for several single-determinant representations of the trial wavefunction. Here we can see that none of these wavefunctions are capable of addressing the missing

angular correlation. Furthermore, although all the results are poor, the DFT wavefunctions are even worse than RHF.

4.4.2 Ethylene

There has been continued interest in calculating various excitation energies for ethylene in QMC, from the ground state singlet ^1A_g, also known as the N state, to the first excited triplet $^3B_{1u}$, the T state, or singlet $^1B_{1u}$, the V state. Experimentally measured energies for the N-T splitting will tend to be artificially low since the molecule twists immediately upon excitement to the triplet state, and indeed, measured values span a range of 4.32 eV [70] to 4.6 eV [71]. Calculations have been in better agreement, with results ranging from recent QMC calculations [72, 73] both producing 4.50(2) eV and 4.51 eV for a CCSD(T)/CBS [74] calculation, up to about 4.6 eV for MRCI [75] and auxiliary field Monte Carlo [76]. In the many comparisons made with experimentally based results, researchers typically do not bother to account for the zero point energy, which is difficult to calculate for the vertical triplet state, so we do not bother to incorporate this either.

To our surprise, even DMC was off by several kcal/mol from the correct energy splitting when we used RHF wavefunctions. Part of the problem, as discussed for methylene, is that the RHF level of theory is inconsistent between the N and T states. Thus the simplest level of theory for which we obtained correct results was the GVB-1 level, which perfect-pairs the π^* orbital to the π orbital for the N state. This indicates that for ethylene, the most important source of fixed-node error is the left-right correlation in the double bond. The $\pi\pi^*$ electrons in the triplet RHF wavefunction are already correlated at the GVB-1 level since they occupy orthogonal orbitals, and both states use the same 9 orbitals, satisfying consistency. The next level of theory is GVB-2, which adds left-right correlation to the CC single bond for both states. Finally, for GVB-6, we add correlation to all 4 CH bonds. RCI and CAS have the same meaning as we described for methylene.

Among our consistent results for aug-tz, shown in Table 4.4, we can see agreement to within 0.26 kcal/mol with each other and with the other DMC results, with the exception of our RCI-6 calculation. This is a clear indication that the GVB level of theory is sufficient to capture the chemistry, and that going beyond this is unnecessary. Here, we can see that, unlike our methylene CAS wavefunctions, our ethylene CAS wavefunctions are correct. On the other hand, our RCI calculation seems to have a problem whereas our methylene RCI

Table 4.4: Vertical ethylene: ${}^3B_{1u} \leftarrow {}^1A_g$ and ${}^1B_{1u} \leftarrow {}^1A_1$. For ${}^3B_{1u}$. For all calculations, we used $R_{CC} = 1.339$Å, $R_{CH} = 1.086$Å, and $\Theta_{HCH} = 117.6$, in order for our results to be directly comparable with Schautz [73]. The entries below the horizontal line are inconsistent, with the number of GVB pairs indicated in parenthesis for each state individually.

SCF	B	J	Δ_e	${}^3B_{1u}$	${}^1A_g$
			kcal/mol	au	au
Exp[a]			100.54		
GVB-1	tz	2	103.13(16)	-78.39872(18)	-78.56307(18)
GVB-6	tz	2	103.38(27)	-78.39781(40)	-78.56256(16)
GVB-2	aug-tz	2	103.45(17)	-78.39759(18)	-78.56245(22)
DMC[b]	Partridge	3	103.5(3)		
DMC[c,d]		2	103.5(5)		
CAS-6[h]	aug-tz	2	103.51(26)	-78.40208(38)	-78.56703(17)
GVB-6	aug-tz	2	103.56(14)	-78.39742(16)	-78.56246(16)
CAS-2	aug-tz	2	103.68(41)	-78.39781(17)	-78.56303(63)
GVB-1	aug-tz	2	103.71(42)	-78.39724(18)	-78.56251(64)
GVB-2	tz	3	103.91(39)	-78.40320(60)	-78.56879(14)
GVB-2	tz	2	103.98(16)	-78.39676(18)	-78.56246(18)
CCSD(T)[e]	CBS		104.1		
RCI-6[i]	aug-tz	2	104.29(14)	-78.39897(16)	-78.56516(16)
RCI-6[h]	aug-tz	2	105.14(38)	-78.39727(15)	-78.56483(59)
Exp[f]			106.1		
RHF	tz	2	100.17(31)	-78.39872(18)	-78.55836(47)
RHF	aug-tz	2	101.91(38)	-78.39724(18)	-78.55964(57)
GVB-(1,1)	aug-tz	2	103.49(42)	-78.39759(18)	-78.56251(64)
SCF	B	J	Δ_e	${}^1B_{1u}$	${}^1A_g$
			kcal/mol	au	au
Exp[g]			177.57		
DMC[c,d]			182.9(5)		
CAS-2	aug-tz	2	190.81(41)	-78.25896(19)	-78.56303(63)
"CAS 6-6"[c]			192.6(5)		
CAS-6	aug-tz	2	199.65(15)	-78.24887(16)	-78.56703(17)

a) Energy-Loss spectra, from Ref[[70]] b) Single determinant from CASSCF(4,8), using pseudopotentials, from Ref [[72]] c) Using pseudopotentials and their custom basis set, from Ref [[73]] d) These DMC results use VMC optimized orbitals. e) Computed value from Ref [[74]]]. f) Optical spectra, from Ref[[71]] g) Adsorption spectra, from Ref [[77]] h) Only determinants with coefficients > 0.01 were included. i) Only determinants with coefficients > 0.001 were included.

calculations were good. We believe that these outliers are evidence again of our wavefunctions getting caught on local minima during optimization. Presumably, we could pay as much attention to these wavefunctions as we did for our methylene CAS wavefunction and perhaps improve the result, but this would represent an unfair selection bias to our overall methodology. Either way, this speaks well for GVB, which does not appear to have any problems.

Examining our inconsistent results, below the horizontal line, we can see that left-right correlation in the double bond (found by comparing GVB-1 with RHF) is worth 1.80 or 2.96 kcal/mol. Our GVB-(1,1) case, an inconsistent wavefunction which correlates the CC single bond for the T state, but only the double bond for the N state, does produce an excellent energy difference, showing that consistency is not always critical. Once the double bond's correlation is included, the QMC results have reached convergence, suggesting that the remaining correlation energy from the SCF perspective is almost entirely dynamical. Looking at the SCF results, the RHF splitting was 83 kcal/mol, GVB calculations all produced about 100 kcal/mol, RCI calculations produced 108 kcal/mol, and our CASSCF(12,12) calculation produced 110 kcal/mol. We can clearly see the advantage of QMC over other approaches, even when inconsistent.

Table 4.5: Vertical ethylene: ${}^3B_{1u} \leftarrow {}^1A_g$ and ${}^1B_{1u} \leftarrow {}^1A_1$. For ${}^3B_{1u}$. For these calculations, we used MP2 optimized $R_{CC} = 1.331046$Å, $R_{CH} = 1.080564$Å, and $\Theta_{HCH} = 121.35$.

SCF	B	J	Δ_e	${}^3B_{1u}$	${}^1A_g$
			kcal/mol	au	au
GVB-2	aug-tz	2	105.05(16)	-78.39480(18)	-78.56220(18)
GVB-6	aug-tz	2	105.14(14)	-78.39556(16)	-78.56311(16)
GVB-6	tz	2	105.38(14)	-78.39464(16)	-78.56257(16)
RCI-6[a]	aug-tz	2	105.55(14)	-78.39558(16)	-78.56379(15)
GVB-2	tz	2	105.64(16)	-78.39401(18)	-78.56236(18)
CAS-2	aug-tz	2	105.82(16)	-78.39502(18)	-78.56366(19)
GVB-1	tz	2	105.99(16)	-78.39400(18)	-78.56290(18)
GVB-2	tz	3	106.14(13)	-78.39984(15)	-78.56899(14)
RCI-2	tz	2	106.16(16)	-78.39460(18)	-78.56377(18)
CAS-6[a]	aug-tz	2	106.54(14)	-78.39733(15)	-78.56711(15)
SCF	B	J	Δ_e	${}^1B_{1u}$	${}^1A_g$
			kcal/mol	au	au
CAS-2	aug-tz	2	192.27(17)	-78.25726(19)	-78.56366(19)
CAS-2	tz	2	193.83(18)	-78.25489(22)	-78.56377(18)

a) Only determinants with coefficients > 0.001 were used.

Originally, we had used an MP2 and the tc basis set to obtain our ethylene geometry, and we include those results in Table 4.5. Concerned about the disagreement of about 2 kcal/mol between these results and the other DMC results, we decided to switch and use exactly the same geometry as Schautz [73], and our results did agree. We include these results to illustrate a few key lessons. First, we point out that most of the difference came from the energy of the T state, underscoring its steep energy slope, an error that not even QMC can correct. Second, notice that previously our RCI-6 calculation was not as much of an outlier, as it is with the new geometry. One difference was that previously, we had used a determinant cutoff of 0.001 for our RCI-6 and our CAS-6 wavefunctions, whereas for the new geometry, we raised the cutoff to 0.01 so that they would run faster (about 2 to 3 times for the N state). This change in truncation appears to have helped the CAS-6 calculation relative to consensus, but hurt the RCI-6 calculation. Thirdly, in rerunning the calculation, we used the optimized Jastrows in the new wavefunctions, and reoptimized everything. This appears to have helped improve consistency, which can be seen by comparing the spread in Δ_e for aug-tz. If RCI-6 and CAS-6 are this sensitive to determinant cutoffs, then this is yet another reason not to use them.

For the N-V vertical splitting, at the bottom of Table 4.4, our energies are 8 kcal/mol higher than the best values reported by Schautz and Filippi [73], for which they even optimized orbitals within their QMC treatment. This underscores the importance of including dynamic correlation during [78, 73] orbital optimization. We use the same geometry, but our results are only comparable when neither of us optimize orbitals. Our CAS-2 N-V splitting, based on a CASSCF(4,4) calculation, is about 2 kcal/mol better than their "CAS 6-6". The difference could be due to pseudopotentials, or because we did not need to truncate our CI expansion like they did, for coefficients below 0.01. Our CAS-6 calculation, based on a CASSCF(12,12) wavefunction with determinants truncated at 0.01, is 7 kcal/mol worse than theirs, perhaps due to a failure on our part to fully optimize this wavefunction.

The N-T vertical splitting is difficult to study experimentally, since the triplet state is far from its D_{2d} minimum. In Table 4.6, we examine the adiabatic splitting, the geometry for which we obtained by optimizing the structure with MP2 using the tz basis set. We use the same N state QMC energies as before, but include them again in this table for completion. Although there doesn't appear to be sufficient experimental data to make a good comparison, we do have some recent high quality CCSD(T)/CBS results [74] to

Table 4.6: Adiabatic ethylene: $^3B_{1u} \leftarrow {}^1A_g$. For $^3B_{1u}$, we use $R_{CC} = 1.449148$Å, $R_{CH} = 1.080469$Å, and $\Theta_{HCH} = 121.5$, and we use the same geometry as previously for $^1A_1$. The entries below the horizontal line are unbalanced in terms of the number of orbitals.

SCF	B	J	Δ_e	$^3B_{1u}$	^1A_g
			kcal/mol	au	au
Exp[a]			61.(3)		
CCSD(T)[a]	CBS		68.8		
GVB-1	aug-tz	2	69.14(42)	-78.45233(18)	-78.56251(64)
GVB-2	aug-tz	2	69.20(17)	-78.45217(17)	-78.56245(22)
DMC[a]	Partridge	3	69.6(3)		
GVB-2	tz	2	69.79(16)	-78.45124(18)	-78.56246(18)
GVB-1	tz	2	70.13(16)	-78.45131(17)	-78.56307(18)
GVB-6	tz	2	70.31(14)	-78.45051(16)	-78.56256(16)
CAS-6[b]	aug-tz		72.13		
RHF	tz	2	67.17(31)	-78.45131(17)	-78.55836(47)
RHF	aug-tz	2	67.34(37)	-78.45233(18)	-78.55964(57)

a) We "uncorrect" the experimental value from Ref [79] of 58(3) kcal/mol and all the computed results from Ref [74] by ΔZPE = 3.2 kcal/mol, so that we can directly compare calculations. b) Single determinant from CASSCF(4,8), using pseudopotentials, computed value from Ref [72]. c) Our own CASSCF(12,12) calculation.

compare with, and with which our best result only differs by 0.5 kcal/mol. Akramine and co-workers [72] also recently studied this transition using QMC, and our results are in agreement with theirs, even given the differences in our wavefunctions.

Finally, we investigate singlet D_{2d} ethylene, obtained by twisting the D_{2h} ground state 90 degrees around the CC bond leaving all other degrees of freedom fixed. Upon twisting, the two π orbitals become degenerate, a complication that many theoretical methods fail to handle correctly. A $\pi\pi^*$ GVB perfect pair for the planar wavefunction becomes a double helix, affecting not only the CC single bond, but the CH bonds as well. We have been unable to find any experimental results for this, so we compare our results against our own CASSCF(12,12) calculation. Our best result, shown in Table 4.7, is only 0.2 kcal/mol higher than the best literature value.

4.4.3 2+2 Cycloaddition

The ethylene + ethylene reacting to make cyclobutane is the textbook example of a concerted reaction forbidden by the Woodward-Hoffman rules. We are only doing a two point calculation, one for an isolated ethylene molecule, and one for an isolated cyclobutane molecule, bypassing any questions related to allowed reaction paths. This is one of the

Table 4.7: Ethylene Twist: $D_{2h} \rightarrow D_{2d}$. The geometry is the same as previously, except that now we have twisted the CC bond by 90 degrees. These results were produced with the MP2 geometry, and are being rerun with the new geometry.

SCF	B	J	Δ_e	D_{2d}	D_{2h}
			kcal/mol	au	au
GVB-1	aug-tz	2	76.96(42)	-78.43987(17)	-78.56251(64)
GVB-2	tz	2	77.04(16)	-78.43968(18)	-78.56246(18)
GVB-2	aug-tz	2	77.14(18)	-78.43952(19)	-78.56245(22)
GVB-6	aug-tz	2	77.30(41)	-78.43928(63)	-78.56246(16)
GVB-6	aug-tz	2	77.89(18)	-78.43834(23)	-78.56246(16)
GVB-1	tz	2	77.54(16)	-78.43951(18)	-78.56307(18)
GVB-6	tz		78.37		
GVB-6	tz	2	78.61(14)	-78.43728(16)	-78.56256(16)
CAS-6	aug-tz		78.88		

simplest reactions that DFT gets wrong, disagreeing with experiment by 5 to 10 kcal/mol, even with some of the more recent functionals, so we consider this to be an ideal test case for QMC. Our cyclobutane geometry was obtained by optimizing the D_{2d} structure with MP2 using the tc basis set.

Table 4.8: Cycloaddition: $2C_2H_4 \leftarrow C_4H_8$. We use the same ethylene geometry as previously, and our cyclobutane geometry is R_{CC} = 1.545029Å, R_{CHax} = 1.089404Å, R_{CHeq} = 1.0877Å, and Θ_{HCH} = 109.18. Below the solid horizontal line are inconsistent calculations, where the number of GVB pairs for the two states are in the parenthesis.

SCF	B	J	Δ_e	C_2H_4	C_4H_8
			kcal/mol	au	au
GVB-4	tz	2	21.98(28)	-78.56246(18)	-157.15993(27)
GVB-4	aug-tz	2	22.05(32)	-78.56245(22)	-157.16004(27)
Exp[a]			22.3(2)		
CCSD(T)	tc		22.54		
GVB-4	tz	3	22.65(23)	-78.56879(14)	-157.17367(23)
M06-2x	tz	2	25.67(30)	-78.56121(19)	-157.16333(30)
RHF	tz	2	27.37(61)	-78.55836(47)	-157.16034(28)
GVB-(1,0)	tz	2	21.45(28)	-78.56307(18)	-157.16034(28)
GVB-(1,4)	aug-tz	2	21.98(82)	-78.56251(64)	-157.16004(27)
GVB-(0,4)	aug-tz	2	25.58(74)	-78.55964(57)	-157.16004(27)
GVB-(0,4)	tz	2	27.12(61)	-78.55836(47)	-157.15993(27)

a) Enthalpy [80, 81] difference of 68.97(71) kJ/mol, corrected with ΔZPE [82] 5.84 kcal/mol

Below the dashed line in Table 4.8 we show our single determinant results using RHF, and also using the orbitals from an M06-2X DFT calculation. We were disappointed to be

unable to get any single-determinant DMC calculation to do any better than DFT, with errors of 3-4 kcal/mol. This process breaks and then makes two bonds, suggesting that at least 2 GVB pairs should be used. However, since the CC bonds in cyclobutane are equivalent, we can not justify using fewer than 4 GVB pairs on either side of the reaction. Indeed, upon adding left-right correlation to the bonds, our best answer agrees perfectly with our experimental estimate to within our 0.2 kcal/mol statistical error. We should mention here that this near perfect agreement should be considered coincidental, since there is perhaps as much error in the ZPE and geometry as in the calculation.

Looking back to our tc ethylene calculations, we estimated that the static correlation in the double bond was worth 2.96 kcal/mol. Seeing here that our single-determinant calculation is in error by about 5.4 kcal, we conclude that most of this error comes from ethylene. With this in mind, we could have accepted decent results by only correlating the double bond in ethylene, which is our GVB-(1,0) result from below the horizontal line in Table 4.8. Although this provides some opportunity for short-cuts in larger calculations, when possible only balanced calculations should be considered, such as those above the line.

We note that all 3 of our calculations were successful, disagreeing by only 0.3 kcal/mol. Therefore, the augmented basis functions did not make a difference. We do not have timing comparisons since they were run on different machines, so unfortunately we can not estimate how much computer time was "wasted." If we would have stopped all cyclobutane calculations once they reached 0.2 kcal/mol error, our cyclobutane RHF/tc wavefunction with 3 particle Jastrows (not in the Table) would have spent 33% more computational time than the equivalent wavefunction without the 3 particle Jastrows. Additionally, the analogous GVB-4/tc 3 particle Jastrow calculation would have taken 4% more time than when we left the 3 particle Jastrows out. The reason is because at a length scale of over 6 a_0, the 3 particle Jastrows can reach almost 4 times as many electrons in cyclobutane than in methylene. In contrast with our conclusions for methylene, the 3 particle Jastrows are not worth the hassle, even if we were entirely confident in their optimization.

4.5 Conclusion

In this paper, we have use QMC to study the effect of various types of wavefunctions on calculations for which we have high quality results to compare against. We have found that

in all cases presented here, a GVB wavefunction was sufficient to obtain results accurate to a few tenths of a kcal/mol, whereas RHF wavefunctions have not been sufficiently accurate. Based on this, we conclude that wavefunction consistency is necessary and sufficient in obtaining the correct wavefunction nodes. This conclusion is drawn with the exception of singlet-singlet ethylene, for which our simple wavefunctions were unable to obtain the correct splitting.

Furthermore, we have discussed our difficulty in studying these same problems using extended CASSCF wavefunctions and RCI wavefunctions. There are two issues that have affected our results. First, our results have been somewhat sensitive to how we truncate the CI expansion for inclusion in our QMC wavefunctions, and it appears that 0.01 is not always good enough for them to perform even as well as GVB. Second, even where we have applied concentrated effort in optimizing CASSCF wavefunctions with all determinants included, there are still concerns that our optimizations are becoming trapped in local minima, such as our CAS-3 methylene result.

Finally, regardless of perhaps minor issues, it is remarkable how well QMC performs even for difficult cases, since all our consistent calculations were within chemical accuracy. We believe that given a simple GVB description with 2 particle Jastrows, we are able to describe a significant amount of chemistry, and given the excellent scalability of both QMC and GVB, we are confident that this high accuracy approach can be applied with confidence to ever larger molecules.

Chapter 5

Additional Work

In this section, we provide more results and commentary on the state of the software. Over the last several years, the software has changed significantly, as we learned what was necessary for a successful QMC calculation, and where we can take short cuts. We document here several results, such as they are, so that future users can understand the conclusions that we have drawn.

5.1 Optimization

When we began our work on QMC, our software was unable to optimize wavefunctions. At first, it was unclear that this was even a problem, given the theoretical claim that DMC results are independent of symmetric Jastrow functions, which do not affect the nodes which are the result of antisymmetry. There was code in place to optimize wavefunctions [83], but the problem was that using it involved writing gigabytes of walker configuration data to disk, and then reading all of that data back in several times for each optimization step. This is a prime example of the Von Neumann bottleneck. This meant that, for example, 2000 samples in a methylene calculation would take only minutes to produce, but a few hours to generate the next optimization iteration, even after converting the files to be written in binary instead of ascii. We tried to fix this in a variety of different ways. First, we attempted to improve the genetic algorithm optimization routines and the line search algorithms to see if they could make more effective use of the data, expensive as they were. Eventually, we concluded that these algorithms simply needed far more data samples. The next thing we tried was to convert the data streams to use the HDF5 file structure from UIUC, but this did not significantly lower the read/write cost of accessing the walker data. In the end,

we switched to the routines that we use now, which do not write anything to disk.

We still believe that the optimization routines that we were using should have worked, since those methods retain their popularity in other research groups. Our problem was most likely simply a poor demarcation of what to write to file, and what to recalculate during optimization. However, it was around the time that we were coming to understand that a major reprogramming effort would be necessary when we discovered that there was another way [50]. Although it was a lot of effort to add analytic derivatives with respect to the optimizable parameters, it was worth the effort. The improvements available [84] would be worth investigating. Now our Jastrows were not only completing their optimization, but doing so in less time than it took our previous methods to fail.

5.2 Jastrows

Riding on the sudden success of our optimization routines, we proceeded to explore more sophisticated Jastrow functions. Initially, we had only been using single parameter Pade Jastrow functions, which took the form

$$u(r_{ij}) = \frac{a r_{ij}}{1 + b r_{ij}} \tag{5.1}$$

as a function of the interparticle distance, r_{ij}. These functions have only a single optimizable parameter, since the coefficient a is fixed by the cusp condition. Although these Jastrows are easily generalized to longer expansions such as

$$u_{ij}(r_{ij}) = \frac{\sum_k^M a_{ijk} r_{ij}^k}{1 + \sum_k^N b_{ijk} r_{ij}^k}, \tag{5.2}$$

they never seemed to work as well as the form we settled on

$$u_{ij}\left[x \leftarrow r_{ij}S\right] = (x-1)^3 \left(\sum_{k=0}^{M} a_k x^k\right), \quad \text{if} \quad 0 \leq \mathrm{x} \leq 1 \tag{5.3}$$

$$= 0, \quad \text{if} \quad \mathrm{x} > 1, \tag{5.4}$$

where S is the length scale, inspired by the functional form of Drummond and co-workers [49], which we discussed extensively in Section 4.3.2. Almost all QMC results found in the literature use 3 particle Jastrows, implying their necessesity. We felt thus behooved to add

them to our own code. We decided to continue using the functional form from Drummond, since the functional form from Huang [85] is significantly more complicated, and as shown in Table 5.1, the results are allegedly almost the same. We added

$$f_{Aij}\left[x_i, x_j, x_{ij}\right] = \left(x_i - 1\right)^3 \left(x_j - 1\right)^3 \left(\sum_{l=0}^{M-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} c_{lmn} x_i^l x_j^m x_{ij}^n\right) \tag{5.5}$$

$$= 0, \quad \text{if} \quad \mathrm{x_i} > 1 \tag{5.6}$$

$$= 0, \quad \text{if} \quad \mathrm{x_j} > 1 \tag{5.7}$$

to our software. These functions have proven to be fairly expensive to evaluate, partly because of the expense of calculating the Laplacian of a function U, which depends upon the coordinates of 2 electrons. This expression turns out to be

$$\nabla^2 r_1 U + \nabla_{r_2}^2 U = \frac{4}{r_{12}} \frac{\partial U}{\partial r_{12}} + 2 \frac{\partial^2 U}{\partial r_{12}^2} \tag{5.8}$$

$$+ \frac{2}{r_1} \frac{\partial U}{\partial r_1} + \frac{\partial^2 U}{\partial r_1^2} + \frac{2}{r_2} \frac{\partial U}{\partial r_2} + \frac{\partial^2 U}{\partial r_2^2} \tag{5.9}$$

$$+ 2\vec{r'}_{12} \cdot \left(\vec{r'}_1 \frac{\partial^2 U}{\partial r_1 \partial r_{12}} - \vec{r'}_2 \frac{\partial^2 U}{\partial r_2 \partial r_{12}}\right). \tag{5.10}$$

The main expense of these functions, however, turns out to be the process of converting derivatives with respect to all of the parameters into derivatives with respect to the independent, optimizable parameters. The constraints must satisfy symmetry, so that the function is unchanged if we swap the 2 electrons, and cusp conditions, which must be zero since we are not using these Jastrow functions to obtain the correct cusps. It is (almost) necessary to apply the constraints on the polynomial for each sample because of how the independent derivatives are used in the various expectation values necessary for optimization.

Furthermore, we feel that this was a poor choice for our 3 particle Jastrow function because after constraining the function, very few of the N^3 terms survived to be independent parameters. For example, if $N = 3$, then the 27 terms reduce to only 8 independent parameters, and if $N = 4$, then the 64 terms reduce down to 26 independent parameters, for totals of 9 and 27 when we include the length scale parameter. The main problem is that two of the basis functions, x_i and x_j, in the 3 particle Jastrow functions themselves do not satisfy symmetry. If they did, then we would not have to spend so much time constraining the polynomial. For example, a better choice would have been the polynomial

basis functions $x_i x_j$ and $x_i + x_j$.

Despite the conclusions we drew in Chapter 4, we think that it is still possible that 3 particle Jastrows could be worth their computational effort. However, we would need to try something different. The functional form in Equation 5.5 is still probably a good idea because the ability to truncate Jastrows seems to be important. The key is probably to use different basis functions. On the advice of Goddard, we attempted to use Chebyshev polynomials for our 2 particle Jastrows, with the hope that they would be easier to optimize. We also attempted to use the basis function $(x - 1)$ instead of just x. Neither of these noticeably helped, but it would be good to revisit the ideas now that we have some definitive results of our own to measure progress against.

5.3 More Calculations

5.3.1 Ne

Now that we were able to optimize good Jastrows, we evaluated their effectiveness using the Neon atom. A summary of our efforts is provided in Table 5.1. We first remark that the time step of 0.001 is quite small, and should have been sufficiently close to the zero extrapolation that the results are of sufficient quality to evaluate the Jastrows themselves without actually extrapolating.

Our first discovery was that optimization was absolutely essential. It was quite surprising to us to observe that a VMC calculation, which is an effective tool in its own right, was unable to do better than even Hartree-Fock itself unless we first optimized the Jastrow. Even after optimizing that Jastrow, our results were still quite poor using this Jastrow function. The other interesting discovery was that a poor Jastrow leads to DMC energies which are below the exact value. We can see, by comparing the optimized and unoptimized versions of the Pade Jastrow, that convergence in DMC is from below. For results not extrapolated to zero time step, the fact that lower does not mean better should be taken into account. This is our explanation for why the 3 particle Jastrow DMC energy is "worse" than the 2 particle Jastrow DMC energy.

Turning now to the improved functional form, we were again surprised, this time because our results were much better than any other published 2 particle Jastrows. In this case, it is perhaps our technique which is responsible, since we can optimize all 3 length scale

Table 5.1: Neon atom, using the aug-cc-pwCVTZ basis set. At the top of the table, we provide the VMC energies, and at the bottom of the table we use the same wavefunctions in a DMC calculation. The Drummond results are from [49], and the Huang results are from [85]. We do not include their variances, because they depend on more factors than the Jastrow. Unfortunately, we have lost the original files, but we estimate that our uncertainties are smaller than the last digit.

	VMC	Variance
Pade, Unoptimized	-128.295	6.90
HF	-128.547	
Pade	-128.620	1.52
2 Particle, Huang	-128.713	
2 Particle, Drummond	-128.757	
2 Particle	-128.810	0.11
3 Particle, 27 terms	-128.866	
3 Particle, 64 terms	-128.873	0.23
3 Particle, Similar, Drummond	-128.886	
3 Particle, Best, Drummond	-128.898	
3 Particle, Huang	-128.901	
4 Particle, Huang	-128.903	
	DMC (dt=0.001)	Variance
CCSD(T)	-128.884	
3 Particle, 64 terms	-128.932	0.08
2 Particle	-128.933	0.12
Exact	-128.938	
B3LYP	-128.977	
Pade	-129.081	2.06
Pade, Unoptimized	-129.245	3.40

parameters, whereas they only optimize 2, holding both electron-electron length scales to be fixed. The other difference is that they use C=2 instead of our C=3 as the exponent of the cutoff prefactor. We found that C=2 prevented smooth optimization of the length scale parameters. Our best explanation for the difference is that our length scales are better optimized than theirs. Our improvement is despite the fact that it seems their Hartree-Fock orbital representation is probably better than our basis set, aug-cc-pwCVTZ.

On the other hand, our best wavefunction with 3 particle Jastrows, which have a total of 54 independent parameters, is worse than the 49 parameter wavefunction of theirs which is probably the most similar. We do not believe that our wavefunctions have been caught in a local minima since we spent quite a bit of effort optimizing them, but it is hard to be sure. Looking at the variances we report, the 3 particle variance should be lower than the 2 particle variance, so perhaps this wavefunction needs more work. We did try allowing the opposite and parallel spin 3 particle Jastrows to vary independently from each other, to attempt to reproduce their "Best" result of -128.898 au, but this did not work.

5.3.2 $\mathbf{Be_2 \rightarrow 2Be}$

The Be_2 molecule is notable as being particularly difficult to study using most quantum chemistry methods. The difficulty stems from the near degeneracy of the 2s and 2p orbitals, significantly distorting the symmetry of the Be atom, and making the RHF wavefunction a particularly poor choice. In the molecular orbital (MO) picture, the 4 valence electrons fill the 2s and 2s* bonding orbitals, resulting in an interaction that is not quite a bond.

The well depth of the dimer has been studied recently by Toulouse [84], where they demonstrate their orbital and basis function optimization, along with all the other first row dimers. There are numerous experimental results, or at least, potential energy fits (PES) to experimental spectra. It is difficult to distinguish which among them is the most accurate for us to compare against, especially since we are not including relativity, and neither do we necessarily have either the exact or the optimized geometry.

We present our results for the well depth of the Beryllium dimer in Table 5.2, using the same aug-cc-pwCVTZ basis set (called aug-tz) used in our other studies. For the Beryllium atom, we use a GVB pair to describe the 2s electrons, correlating them with a 2p orbital. For the dimer, we use a GVB pair to describe the single bond, and a second GVB pair to correlate the lone pair electrons. This wavefunction was particularly difficult to obtain,

Table 5.2: Be ← Be_2 at the experimental geometry of 4.65 bohr. The DMC results from Toulouse include their basis function optimizations. Our GVB approach is intermediate to theirs, and we do not optimize basis functions.

δt	Δ_e	Be	DMC	Be_2	DMC
au^{-1}	kcal/mol		au		au
.	1.769(18)			DMC, full valence CAS from [84]	
0.0075	2.24(11)	atc0p1o	-14.660410(69)	atc0p2o	-29.32439(11)
.	2.259(86)				Exp from [86]
.	2.37(18)			DMC/CASSCF(4,8) from [87]	
.	2.399(29)				Exp from [88]
0.0075	2.44(12)	atc0p1	-14.660254(72)	atc0p2	-29.32439(12)
0.0075	2.555(85)	atc4p1o	-14.660549(54)	atc4p2o	-29.325169(84)
.	2.582(23)			Non Rel. Exp Fit from [89]	
.	2.699(71)			PES fit to Spectra from [90]	
0.0075	2.772(90)	atc4p1	-14.660479(52)	atc4p2	-29.325375(98)
.	2.882568			DMC, 1 det from [84]	

because the GVB calculations would typically swap during convergence the $\sigma*$ orbital for a *pi* orbital. Where a 4 replaces a 0 in the labels for our calculations, we are indicating that 3 particle Jastrows were employed.

We have implemented a simple form of orbital optimization, where we optimize all orbital coefficients as parameters equivalent with all other Jastrow and CI parameters. The only difference is that we hold fixed any coefficients given to QMcBeaver as 0.0. We do not pay attention to any considerations beyond this. Our results using this simple technique are presented with the suffix 'o' in Table 5.2, where we are now 0.019 or 0.159 kcal/mol from the experimental data for our calculations which use only 2 particle Jastrows. Unfortunately, for Be_2 this already includes 292 parameters, and the analytical calculation of orbital derivatives is not cheap. For these reasons, this is the only reaction that we ended up using orbital optimization to study. We also note that the error dropped by about 5%.

5.3.3 $\mathbf{O_3}$ $^1A_g \rightarrow \mathbf{O_3}$ $^3B_2$

Another interesting problem in Quantum Chemistry is ozone excitation from the ground state into the lowest triplet state. The difficulty here is that ozone is a highly multi-configurational molecule, necessitating the inclusion of quite a few configurations into the wavefunction. Our studies have found that this system is not easy for QMC either, since our GVB wavefunctions did not prove to be of sufficient quality. We thus turned to CASSCF

wavefunctions, and have run simulations using these. We present two calculations in Table 5.3 for the adiabatic transition, and one for the vertical transition.

There are a total of 12 valence orbitals in ozone, which are not necessarily all important, and we have chosen 2 subsets. Our CASSCF-7 calculations include the complete active space of the 4 single bond orbitals (bonding and anti-bonding), as well as the 3 π orbitals. This turned out not to be sufficient, so we have added a CASSCF-9 which is the active space of all nine 2p atomic orbitals.

Table 5.3: Adiabatic ozone excitation: $^3B_2 \leftarrow {}^1A_g$. We assume that the CASSCF-9 result has similar errors as the CASSCF-7 calculation, but the error estimators did not converge for this calculation due to problems at the ends of the runs. Our ground state geometry is [r,θ]=[1.27276Å,116.7542] from [91], and for the excited state we used [r,θ]=[1.3542Å,108.54] from [92].

SCF	B	J	Δ_e	$^3B_2$	^1A_g
			eV	au	au
CASSCF-7	aug-tz	2	1.278(11)	-225.31625(28)	-225.36323(28)
CASSCF-9	aug-tz	2	1.343(31)	-225.3201(11)	-225.36949(34)
			1.346	Exp[a]	
CASSCF-7[b]	aug-tz	2	1.557(11)	-225.30601(29)	-225.36323(28)

a) T_0 = 1.30 eV, from [93], ΔZPE = 0.046 from [94] b) vertical transition.

5.3.4 $\mathbf{SiH_2}$ $^1A_1 \rightarrow \mathbf{SiH_2}$ $^3B_1$

Given all of our efforts in methylene, it seemed reasonable to also study silylene, and we present the results of these calculations in Table 5.4. Part of the reason why this system is interesting is because in contrast with methylene, the singlet state is the ground state in silylene. The difference is that for methylene, the 2s and the 2p orbitals are nearly degenerate on the Carbon atom, but for Silicon, the 3p orbitals are far higher in energy relative to the 3s orbitals due to electron shielding of the sub-valence electrons.

Silylene offers a few challenges to our technique that have not been fully resolved. First of all, we see that because we are running all electron simulations instead of using pseudopotentials or one electron at a time iterations, we are getting far more warning messages in our output files. Second, we see significant trends in the convergence of the energies shown in Figure 5.1, so it looks like our results could be off by as much as 1.0 kcal/mol. As bad as this might seem, it is worth noting that values typically seen in the literature are about 18 kcal/mol, which is different from our result by up to 4 kcal/mol! We differ from Berkowitz

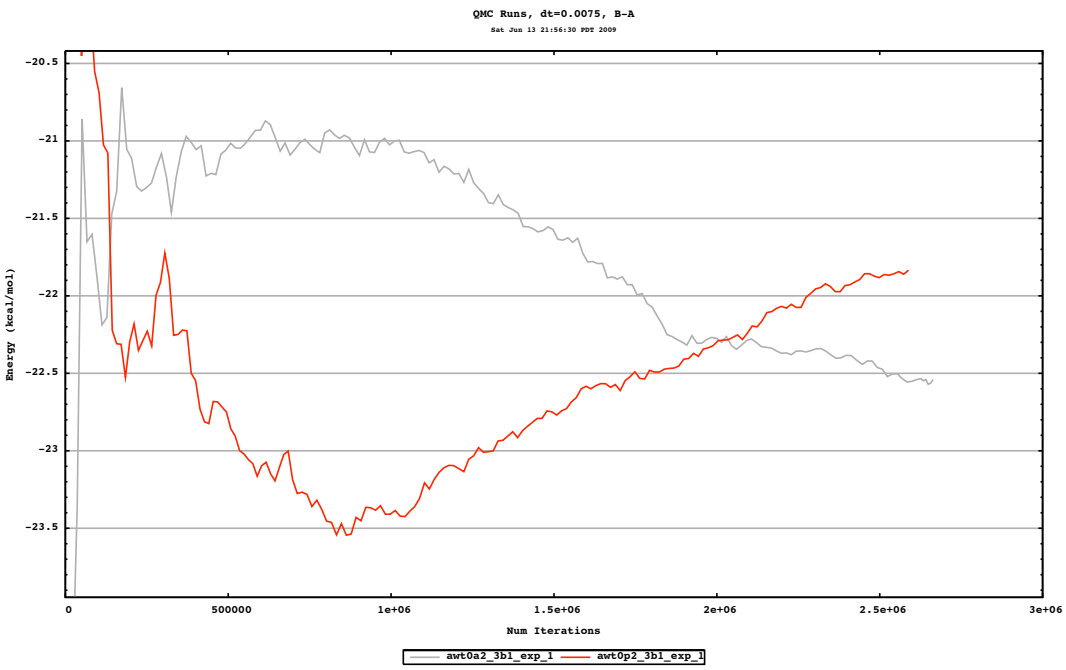

Figure 5.1: Convergence of silylene $SiH_2\ ^1A_1 \rightarrow SiH_2\ ^3B_1$ excitation. Results with trends in the data like these should be discarded, but they are the only ones we have for this system.

and co-worker's[95] best experimental value by only 0.79 kcal/mol, but they leave the door open to the 18 kcal/mol value. This is perhaps a better result than we deserve, given the convergence results. On the other hand, this result helps to affirm that if we have captured the essential chemistry in the SCF part of the wavefunction, then QMC will achieve at least chemical accuracy. The problems indicated in Figure 5.1 are clear evidence of the need for pseudopotentials.

Table 5.4: $SiH_2\ ^1A_1 \rightarrow SiH_2\ ^3B_1$. The experimental results are from [95], where their best result is 0.91 eV. They say that an alternative interpretation of their data would indicate 0.78 eV.

SCF	B	J	Δ_e	$^3B_1$	$^1A_1$
			kcal/mol	au	au
Exp			18.0(7)		
Exp			21.0(7)		
GVB-3	aug-tz	2	21.78(60)	-290.57724(91)	-290.61195(29)
CAS-3	aug-tz	2	22.54(18)	-290.57871(28)	-290.6146209464

5.3.5 Survey of G1 Atomization Energies

Given the success of our GVB wavefunctions in QMC calculations, we wanted to find out how well this type of calculation would work in general. To do this, we ran several calculations from the G1 test set, following after the work of Grossman [96], who also performed this type of calculation. These results are not intended to be representative of the best we can do, but instead find the boundaries of where our approach would work. Unfortunately, the online database from which we obtained our geometries was taken down, so we do not know exactly how they were obtained. As we have remarked in Chapter 4, there is possibly as much error in a poor geometry or in the zero point energies as there is in a high quality QMC calculation itself. Regardless of the relatively poor quality of these results, we present them in Table 5.5 as they are because there is already a lot we can learn. These results were based on GVB wavefunctions, and we added doubly excited RCI determinants to some of them.

Table 5.5: Calculations from the G1 test set. The O_2 calculation was actually based on a CAS wavefunction, and not an RCI calculation. Grossman's results [96] were obtained with single determinant wavefunctions by selecting the best determinant from a CASSCF wavefunction. The errors are measured as the absolute difference from the G1 recommended experimental.

Molecule	p	QMC-GVB	Error	QMC-RCI	Error	Grossman02	Error
H2	1	109.47	0.08				
LiH	1	57.78	0.03			55.3	0.70
BeH	1	53.14	3.45	53.132	3.44	43	3.95
CH	2	83.60	0.23			79.5	0.42
CH2.trp	2	189.12	0.86			181.9	1.87
CH2.sng	3	180.01	0.64			169.7	0.89
CH3	3	306.96	0.36			290.9	1.65
CH4	4	419.81	0.29			395	2.58
NH	2	81.29	2.20	82.297	1.19	78.2	0.78
NH2	3	181.03	0.49			169.2	0.80
NH3	4	297.10	0.34			276.5	0.20
OH	1	106.01	0.33			101.2	0.04
H2O	2	231.40	0.83			219.4	0.04
HF	4	140.95	0.14			135.9	0.65
SiH2.sng	3	154.46	3.18			145.5	1.32
SiH2.trp	2	132.78	2.24			125.8	2.59
Li2	1	22.92	1.45			23.5	0.44
H2C_CH2	6	561.48	2.02			533.5	1.59
H3C-CH3	7	710.88	0.13			669.3	3.23
CN	4	167.26	13.66	173.87	7.05	170.5	7.89
CO	5	253.50	5.79	258.67	0.62	253.2	2.98
N2	5	221.84	6.73	228.18	0.39	221	4.05
NO	2	143.50	9.27	144.33	8.44	142.9	7.03
O2	3			119.34	1.19	111.7	6.28
F2	7	36.13	2.39			32	4.93
C4H8.d2d	4	1144.18	2.92				
SH	1	87.99	1.12				
Average			2.30		1.61		2.37

As we can see from these results, it looks like GVB wavefunctions are sufficient to study most of the molecules in the table. We can see that we get essentially the exact result for H_2, the only wavefunction here with no nodes. Beyond this, we get decent results for almost all molecules except CN, CO, N_2, NO, which fail catastrophically. This observation should be sufficient to dispel any remaining doubt that the fixed-node error can be quite large. For CO and N_2, however, we see that an RCI wavefunction is sufficient to capture the remaining error in these nodes. On the other hand, even though RCI helps, apparently we are not yet using a wavefunction of significant quality to study CN or NO. The problem with these two ground state doublet molecules is that the unpaired electron has significant occupation in orbitals that would otherwise be GVB paired. For the molecule which we employed a CAS wavefunction, O_2, we managed to measure a respectable atomization energy, even if the error is larger than we would like. A few of the other molecules expressed large errors, but did not take the time to isolate the problems. Our calculations produced a lower average error than those of Grossman, but since our calculations were run without pseudopotentials and his were, we were not able to run as many molecules as he.

We are glad to observe that most of our results, where we seem to have captured the essential chemistry, are within the error margins of chemical accuracy. Pointing out again that there is often as much error in the geometry as there is in the zero point energy, we should not necessarily expect better results than those we have presented here, given the survey nature of these results. However, we would have expected to do better for our ethylene atomization calculation because of the attention to detail from Chapter 4, which here is in error by about 2 kcal/mol. Our error for cyclobutane, for which our geometry is only mostly accurate, was in error by 3 kcal/mol. We assume this is because we are only adding perfect pairs to the CC bonds, and not for any of the CH bonds. With these considerations in mind, we are cautious about using QMC and our methodology to calculate atomization energies, even though we have seen several such calculations in the literature.

5.4 A Crazy New Idea

We have investigated a few of the fundamental elements of the QMC algorithm, starting with the accept/reject step. Both VMC and DMC measure the quantity (introduced in

Equation 2.19)

$$p = \frac{T(r \leftarrow r')}{T(r \rightarrow r')} \frac{\Psi_T^2(r')}{\Psi_T^2(r)}, \tag{5.11}$$

where T is some transition matrix. The acceptance probability

$$A = \max[\min[1, p], 0] \tag{5.12}$$

is compared to a uniformly distributed random number to determine whether a proposed trial configuration should be accepted for the walker in question. We present a distribution of the value p for all electron moves in Figure 5.2 using GVB/tz wavefunctions, where we can see that about half of the distribution is above 1. Looking to the left of $p = 0$, which corresponds to crossing a node, we can see roughly how far walkers try to jump past the node. The fixed-node condition sets the probability of all such moves to zero, as shown in Equation 5.12. Figure 5.2 shows that the peak at $p = 1$ broadens as either the time step gets larger, or as the molecule gets larger, a feature which results in a lower average acceptance probability. Crudely assuming symmetry in the distribution, one might guess that the average acceptance probability would not drop below $0.5 \langle p > 1 \rangle + 0.25 \langle p < 1 \rangle = 0.75$, but this remains to be determined.

There is a very interesting feature visible when we bin the data after the fixed-node condition has been applied for the same data, as we have done for Figure 5.3. Here, we can see that all the $p < 0$ tail has been mapped into the $p > 0$ region, producing the large peak in the left-most bin. But what is more obvious now is that the peak does not appear to be discontinuous and that the abnormalities are seen out to about $p = 0.05$. This suggests a new strategy for the acceptance probability

$$A = \max[\min[1, p], 0.05] \tag{5.13}$$

which not only prohibits the $p < 0$ moves forbidden by the fixed-node condition, but also a few more moves. For our methylene 0.05 time step, the cumulative probability up to $p = 0.05$ is approximately 3.4%, of which 2.3% was in the $p = 0$ bin. This means that our new strategy would prohibit an additional 1.1% of the moves, possibly helping the calculation to avoid some of the instabilities that have made calculations difficult. If we apply this new rule, we get the results shown in Table 5.6, where both results have improved.

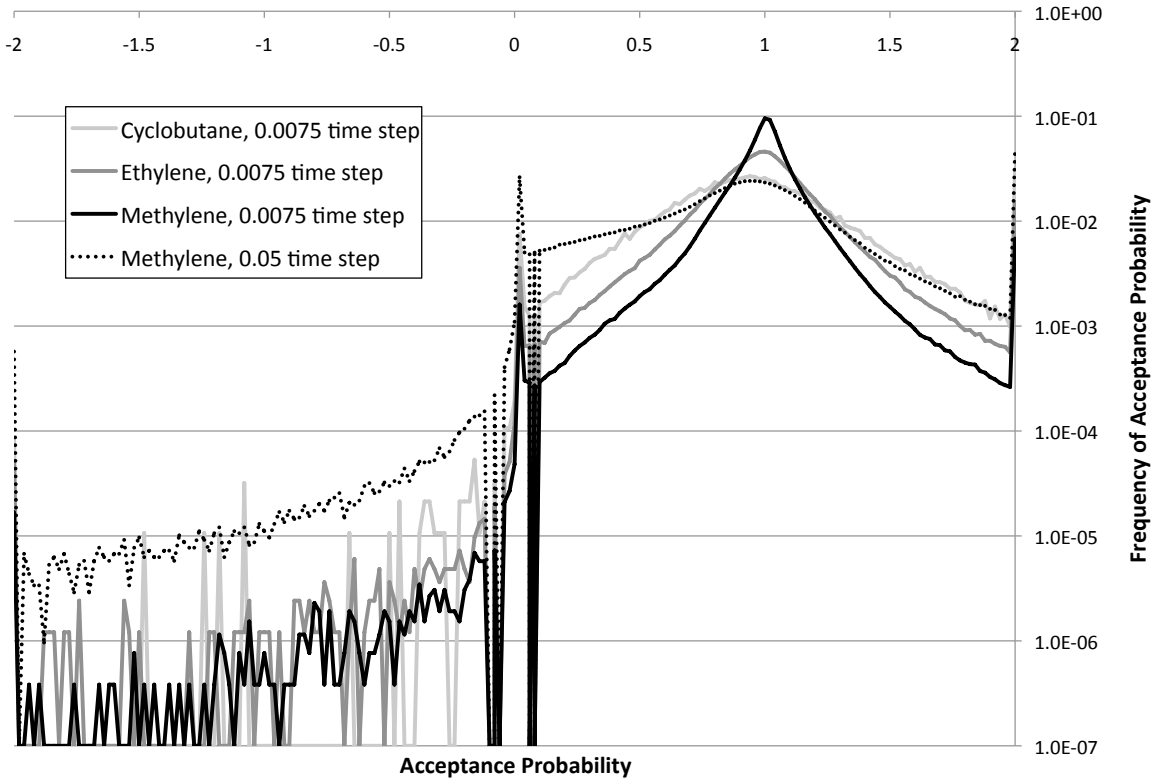

Figure 5.2: The probability distribution function of the acceptance probability over the range -2 to 2. This data was collected by binning the acceptance probability before manipulation. Values outside of this range were added to the nearest bin for the histogram.

These very preliminary results are quite encouraging, and we believe that pursuit of this route will be fruitful.

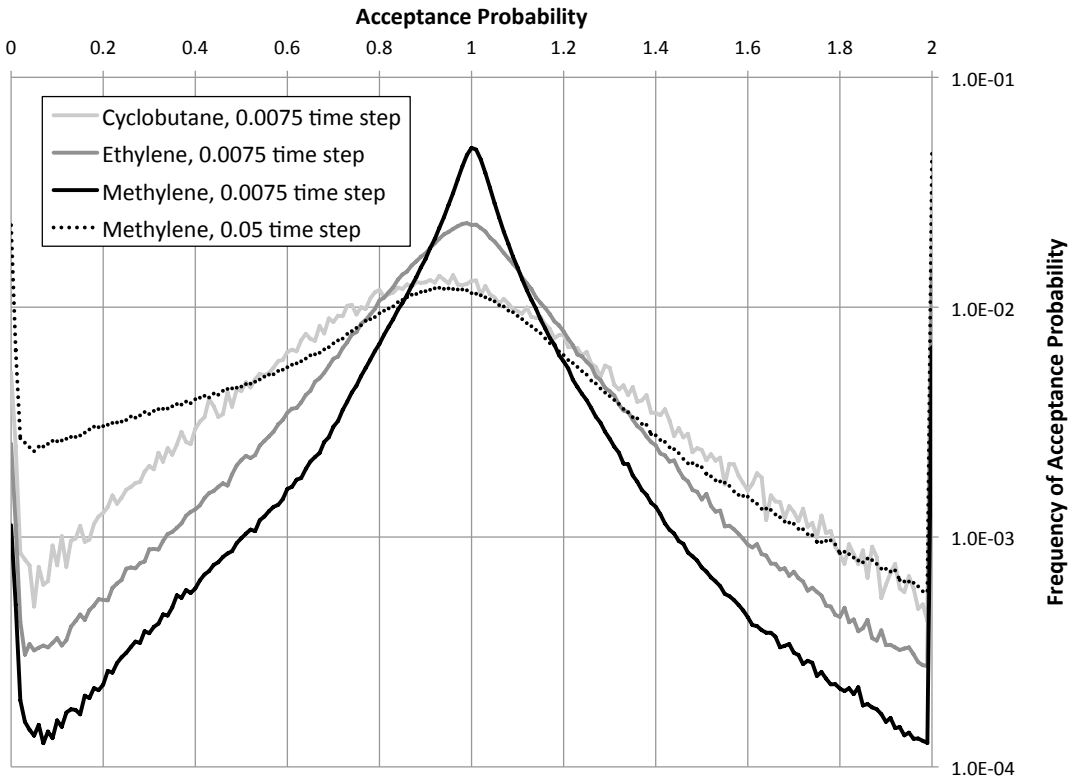

Figure 5.3: The probability distribution function of the acceptance probability over the range 0 to 2. This data was collected by binning the acceptance probability after the fixed-node condition has been applied. Values outside of this range were added to the nearest bin for the histogram.

5.5 The Preferred Number of Processors

Most QMC programs keep the number of walkers N_w constant as the number of processors N_p is increased by putting N_w/N_p walkers on each processor. We, on the other hand, put N_w walkers on each of the processors for a total of $N_w N_p$ walkers in the calculation, synchronizing the energies across all the processors every few iterations. For us, this introduces the important question of how many processors should we use in a calculation since, now, the error associated with a finite walker population depends on this decision. To test this question, we ran several GVB-3/aug-tz (delocalized) with 2 particle Jastrows methylene calculations, keeping $N_t N_p = 6.4$ million, where N_t is the number of iterations with a 0.01

Table 5.6: A new acceptance probability strategy, as shown in Equation 5.13, where L indicates localized GVB orbitals, and D indicates delocalized GVB orbitals.

SCF	B	J	Δ_e	$^3B_1$	$^1A_1$
			kcal/mol	au	au
GVB-3*	L/aug-tz	2	9.071(80)	-39.121669(91)	-39.136124(89)
GVB-3	L/aug-tz	2	9.178(97)	-39.12145(10)	-39.13607(11)
Exp*			9.364(53)		
GVB-3	D/tz	2	9.500(93)	-39.120544(90)	-39.13568(12)
GVB-3*	D/tz	2	9.53(10)	-39.12065(11)	-39.13584(12)

* results copied from Table 4.1.

time step. This is equivalent to holding the computational effort to a constant. We present our results in Figure 5.4 and Table 5.7.

In this data, we can see that the energies are relatively independent of the number of processors, with no deviations more than 0.15 kcal/mol from the reference, and as seen in Table 5.7, the errors are also relatively constant. The largest deviation is for 16 processors, which is probably because at 400,000 iterations, it did not have enough *time* to sample the entire wavefunction. This conclusion is supported by our results from varying the time step Section 4.3.5, where we concluded that a 0.01 time step needs at least 1.5 million iterations on 4 processors. If this is strictly the case, then by this experiment's design, only our 4 processor calculation, which ran for 1.6 million iterations, is reliable. This is perhaps the most significant conclusion from this data because it implies that a calculation needs to be run for a minimal number of iterations on each processor; that the prerequisite among of time can not be parallelized.

Table 5.7: The effect of holding the number of samples collected constant at $N_t N_p = 6.4$ million, where N_t is the number of iterations of 0.01 time step, while varying the number of processors used. We have added our fully converged value for reference, which was run at 0.0075 time step.

N_p	Δ_e	$^1A_1$	$^3B_1$
	kcal/mol	au	au
16	9.05(11)	-39.12086(13)	-39.13529(13)
1	9.17(12)	-39.12072(13)	-39.13534(13)
4	9.23(11)	-39.12064(13)	-39.13536(12)
	9.239(88)	-39.120847(81)	-39.13557(11)
8	9.25(10)	-39.12064(13)	-39.13538(11)
2	9.38(11)	-39.12060(13)	-39.13554(13)

The other conclusion to draw here is that one processor is not enough, and that two

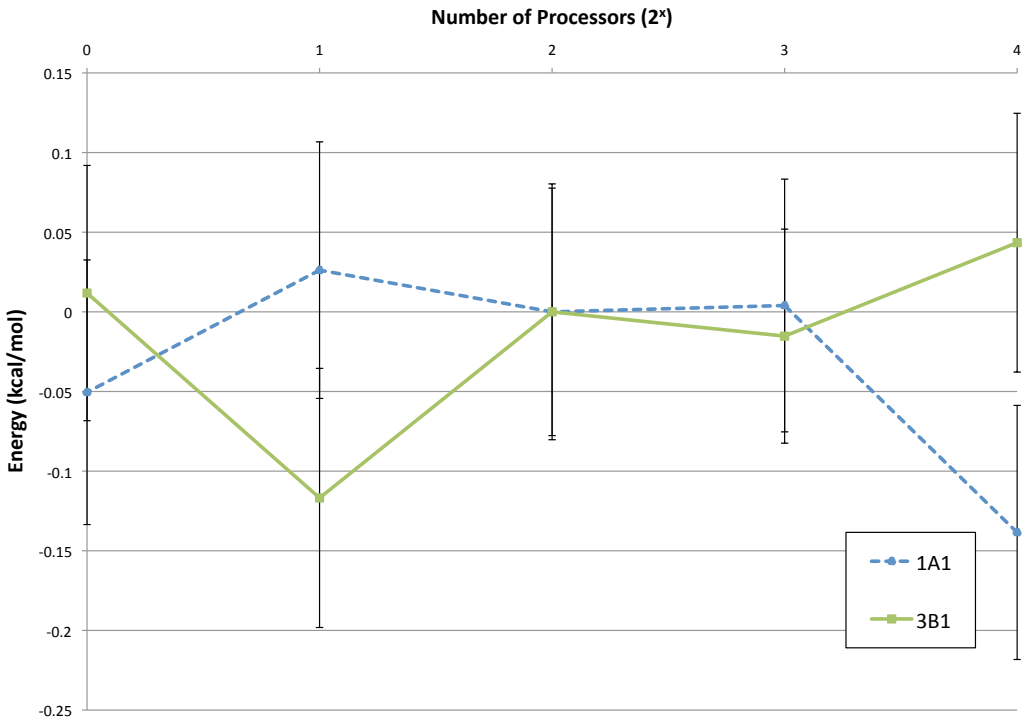

Figure 5.4: An experiment holding the number of samples collected constant at $N_t N_p = 6.4$ million, where N_t is the number of iterations, while varying the number of processors used. Each calculation spent 5000 iterations from the total equilibrating, after our high quality initialization. The energy is measured relative to energy on 4 processors.

is suspect, even though both of them were run for more than the 1.5 million minimum we guessed are necessary from our 4 processor simulations. Part of the problem, we suspect, lies in the total number of walkers in the calculation, and that some of the error incurred is due to finite population bias. The larger source of error is likely from the instabilities in the algorithm itself, and that runs on multiple processors are benefitting from some cancelation of errors between the individual processors. Indeed, it would appear that only our 4 and 8 processor calculations produced good results.

To be clear, we are not claiming that a calculation can never be run on more than 8 processors, we are only claiming that 4 or 8 processors made the most efficient use of equivalent processing power in this test. A more thorough test would quadruple the number of iterations for each calculation so that all of them run for at least 1.6 million iterations. In our research, we have had access to several Department of Energy machines, including large clusters at LANL and LLNL, where we were routinely able to use up to 256 processors at a time. The problem was that those processors were slow, leading us to attempt to compensate for slow individual processors by using more processors to collect the desired number of samples quicker. However, we now view this strategy with suspicion.

5.6 Pseudopotentials

We have attempted to perform more calculations than those we have presented here which by some standard have worked. Our very best results were in Chapter 4, results for which we paid close attention to detail. The results from this chapter were significantly more time consuming both in terms of computation time, and in terms of trying wavefunctions or geometries that did not work. This is substantial justification for using pseudopotentials. It is not because they save computational effort on a per iteration basis, which they often do not, but because sample errors become significantly lower when the troublesome core electrons are removed.

We spent some time attempting to add pseudopotentials to our code, but never got them to work, even though we believe we are really close. We observed reasonable VMC results, but when we ran our calculations in DMC, the energy would very frequently jump to some absurd value. We were very disappointed that despite our efforts, we were never able to get a transition metal calculation to succeed. This was mentioned in a recent conversation

with Ken Esler at NCSA, who pointed out that other software packages put a lot of effort into "protecting" walkers from the new singularities introduced with pseudopotentials. For example, some will watch for jumps in the energy, which when encountered will backtrack several iterations. So perhaps this type of procedure is all that we need. This illustrates one of the difficulties in working on this project, that none of us have had much opportunity to discuss the unpublished "folklore" or conventional wisdom with experts in the field. There are a lot of ideas in the literature, and a lot of time to waste implementing all of them. Pseudopotentials are necessary.

Chapter 6

Kinetic Monte Carlo

6.1 Abstract

We present an $\mathcal{O}(\log N)$ implementation of lattice based Kinetic Monte Carlo (KMC), where N is the number of grid sites. In our initial tests, we can run for extremely long simulation times, of the order of seconds, on a single processor in a couple of days, depending on the system. Furthermore, our computation time scales as a constant with respect to the number of molecule types and reaction types included. This implementation opens up the KMC method to a wide variety of applications, and we present one involving CH_3Cl(g) molecules adsorbing onto Copper slabs with Silicon impurities.

6.2 Introduction

Quantum chemistry has become capable of reliably, quickly, and accurately producing an abundance of data, measuring energy differences between different species, energy barriers, and reaction rates, but only for very limited numbers of small molecules at a time. There are several ways to take data from these calculations, and simulate such a system as might be found in a test tube, and on useful time scales. For one popular technique, molecular dynamics (MD), we might fit quantum data to potential energy curves representing bond strengths, bending angles, and other representative motions in a molecule, and then simulate a box of these molecules as particle simulation of forces. This method has been enormously successful at modeling anything from materials to proteins. While MD does not normally allow bonds to break or to be made, it can be modified to do so for a price. The downside of MD is that it is really slow, and is typically only run for a simulation time between nano

and micro seconds, although with enormous expense, simulations of up to milliseconds are now becoming possible.

KMC, an increasingly popular alternative, swaps continuous spatial coordinates for a grid, allowing the user to specify reaction rates directly. This approximation permits fast algorithms to be developed which easily run for as long as a second of simulation time, depending of course on the nature of the system being studied. It is well known that KMC should scale as $\mathcal{O}(\log N)$ per iteration, where N is the size of the grid, and we discuss a simple algorithm which achieves this. We also show how we achieve constant time scaling with respect to the number of species or reactions in our system.

6.3 What is Kinetic Monte Carlo?

There are a wide variety of approaches to kinetics using Monte Carlo techniques. We focus here on those which use a grid of some form since we want to include spatial competition in our model. Furthermore, we only specify a set of species which can exist on the surface, and provide reaction constants (forward and reverse) to convert between them. This not only includes reactions, but diffusion, adsorption, and desorption processes. Thus, given any state of the surface, we can count all possible "actions" that could happen. The effects we include in our model allow molecules to diffuse to an adjacent empty site, an empty site to receive a molecule from the gas phase, or two adjacent molecules could react, possibly to form gas products. For all of these actions, we have a rate calculated from the energy barrier, which is directly proportional to the probability P_a that it will happen in a given iteration. There are a couple ways of handling this, for example, the Metropolis method would pick a random particle and associated action, and act on it with probability P_a. The best way of handling this is described in Algorithm 6.3.

Algorithm 3 KMC algorithm

loop
  Identify all possible actions, and their associated rates r_i
  $R \leftarrow \sum_1^N r_i$
  $u \sim [0, R]$
  Find j such that: $\sum_1^j r_i < u < \sum_1^{j+1} r_i$
  Perform action 'j'
  $t = t - \frac{\ln([0,1])}{R}$
end loop

This efficiently produces a Poisson distribution, meaning that all the events are independent from each other. This allows the same molecule to be involved in successive actions, if it has sufficient propensity to do so. The computationally expensive steps is the first one listed here, of collecting all possible actions, as well as the fourth step, of searching the list.

6.3.1 The General Solution

At a first glance, this is a linearly scaling process, since one would need to first make an array, and then search the array for a cumulative probability. But this array of actions does not change much from step to step, especially if there is some degree of locality to each change. It is well known that any KMC algorithm could scale as $\mathcal{O}(\log N)$ per event through the use of binary trees [97] to perform the array updates and searching. However, the realization of this is often application dependent [98]. Although we have found several theoretical analyses [97, 98] of KMC, we have found very few discussions of actual implementation and how to design the binary trees.

6.3.2 Our Solution

To initialize a calculation, we start by evaluating the rates of all the forward and reverse reactions that were specified in an input file. We then separate all processes into two categories: those which only involve one site, and those which involve two sites. For all the one site processes, we create a static array indexed to each specie in the system (which includes the "empty" specie), summing all the rates for processes it might perform. For example, an empty site might receive a non-disociating gas molecule, or might receive an atom percolating up from below the surface. In these instances, the reaction would look like the conversion of an empty site into an occupied site. Two site processes are handled in an analogous fashion by allocating a matrix with each dimension indexed to the species in the system. If two species can interact in any processes, we sum all the corresponding rates into that matrix element. Based on this understanding, the matrix is symmetric.

To model reactions on a rectangular 2D surface, we design a binary tree such that each node contains the sum of all the reaction constants for a $1/2^L$ fraction of the surface, where L is the level in the binary tree of the node in question. Thus the root node covers the whole surface with $L = 0$ and stores R. Its left and right children are defined as partial sums such that $R = R_l + R_r$, and upper and lower distinctions are specified at

the grandchildren level so that $R = R_{ul} + R_{ur} + R_{ll} + R_{lr}$, and so on. This means that a sum of the reaction constants across all the nodes for a given level will produce R. The leaves of this binary tree are the grid sites themselves, which store the reaction constants for everything that can happen at that site individually, as well as half (so that we do not double count) of its neighbors. Of course this could be adjusted for the topology of any 2D surface. Furthermore, this approach is general enough to allow some 3D systems by either modeling them as connected 2D structures, or by adding a dimensional index to the labels of the species in the input file.

To find a particular reaction starting at the root, we see if U, our uniformly drawn random number on $[0, R]$, is lower than the left child. If it is, then we proceed down the left branch passing along the same value for U. If it is higher, then we proceed down the right branch using $U - R_l$ as our new uniformly distributed random number. Either way, U is uniformly distributed between 0 and the cumulative value R for the lower node, so we repeat this comparison moving down the tree until we reach a grid site.

Once we reach a grid site, we note that the residual value of U is uniformly distributed on $[0, r]$ where r is the sum of the 4 r_i representing everything that can happen between ourself (1 term) and half of our neighbors (3 terms). The cost of this search is constant, since the r_i were precomputed, resulting in a maximum of 3 comparisons and 3 subtractions. Once we have found the r_i corresponding to U, we scan through only the precomputed list separating all the few ways that the two species can interact. After performing the sought after process, we update the r_i which changed and then update r. Then follows one update per level, in an $\mathcal{O}(\log N)$ overall update of the binary tree.

This implementation has several merits. First, in contrast with the general techniques described [98], we never have to update any lists of nearest neighbors, or change the structure of our binary tree. Each node in the binary tree remains responsible for exactly the same grid sites throughout the calculation, and an update only involves propagating terms like $R = R_l + R_r$ up a tree, involving very little math for each update. Second, our method scales in constant time with respect to the number of species in the system, since the sum of all the ways they can interact is precomputed and stored in a matrix in random access memory (RAM). Thirdly, our method scales as constant time with respect to the number of reactions included in the input file. The reason is that all the ways two species can interact is known at compile time, so instead of writing code which searches through the list of reactions to

find all the ways two species can relate, we use a script to generate sparse search code based on the input file which automatically skips any meaningless comparisons. This stores the equivalent of a matrix into the executable itself. Thus our RAM requirement scales as the square of the number of species, and our hard disk requirement scales as the square of the number of reactions. In our studies, neither of these requirements have been high. With this implementation, we believe that we have an original $\mathcal{O}(\log N)$ implementation of the KMC method that is more efficient than any other.

It is worth pointing out that because the simulated time increment scales as $1/R$, the number of iterations required to reach a desired simulation time scales as $\mathcal{O}(N)$ for an over all scaling of $\mathcal{O}(N \log N)$. However, this scaling also depends on other parameters such as temperature and pressure, which affect the individual rates leading to R. The biggest concern however comes from large differences in the energy barriers. If an input file specifies reactions which all have comparable energy barriers, then they'll also have comparable rates, and fewer iterations should be necessary to produce interesting results. However, the rates scale exponentially with the energy barrier, so if you include fast processes (like diffusion) along with slow but interesting processes, then the probability per iteration that an interesting reaction will occur will drop exponentially. These factors underscore the vast variability in the number of iterations that might be required to complete a calculation.

6.4 Our First Application

For our initial project, our research group has been working on modeling a system for converting MeCl(g) into $\mathrm{Me}_n\mathrm{SiCl}_{4-n}(g)$ on a copper surface, where Si atoms diffuse through the copper slab, and MeCl(g) is maintained at a constant pressure against the copper. We are particularly interested in factors that influence the product branching ratio, including the rate of introducing Si atoms, pressure of MeCl(g), temperature, and others. Additionally, we have been interested in how to change the branching ratio by placing constraints on various species $\mathrm{Me}_x\mathrm{SiCl}_y$ intermediate on the surface. A lot of work has been done to identify all the important molecules on the surface, and to write equations for reactions between all the species. We had taken the approach of modeling the system using a differential equation solver, but it was felt that surface competition might play a major role, so we have been developing this KMC algorithm to model this. Since this represents work

in progress, we have little to report as far as final conclusions, but we can provide some preliminary results as produced by KMC.

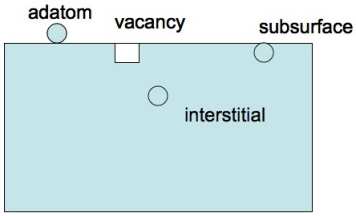

Figure 6.1: This image was provided by Mario Blanco, and illustrates a few of the surface sites possible.

The species in play are classified in 5 categories, illustrated in Figure 6.1. We have gas molecules, denoted with the (g) suffix, subsurface atoms denoted with (s), and adatoms which normally would have the (a) suffix, but as the *default* state, the suffix is implied and left off. Under this notation, Cu(s) is the label for a normal, or empty site. We include Cu(v) to denote a vacancy in the copper surface. Lastly, we also have Si(int), for interstitial Silicon, diffusing through the surface but doesn't fill a grid site, waiting to be converted into Si(s). Gas molecules are not explicitly included in the model. Product gas molecules are counted, but then immediately disappear. Only MeCl(g) gas exists above the surface, but only implicitly until one of them lands and sticks to the surface. Neither are Si(int) atoms countable, until they push up to the subsurface layer, at a prescribed rate.

We start by hypothesizing that the rates can be adequately modeled using the following simple expressions, written in terms of the energy barrier E_b of the process, and the temperature T and pressure P:

$$\begin{aligned} r_{diffusion} &= \frac{k_B T}{h} \exp\left(\frac{-0.5}{RT}\right) \sim 8.1 \times 10^3 \text{ nHz} \\ r_{reaction} &= \frac{k_B T}{h} \exp\left(\frac{-E_b}{RT}\right) \sim 10^{-17} \text{ to } 1.3 \times 10^4 \text{ nHz} \\ r_{adsorption} &= \frac{k_B T}{h} \exp\left(\frac{-E_b}{RT}\right) \frac{p_{con}\pi P}{4\mu} \sqrt{\frac{\pi\mu}{8 r_{gmv} T}} \sim 5.0 \times 10^{-5} \text{ nHz}, \end{aligned}$$

where all E_b for the forward and the reverse processes are provided as input to the software. For the sake of illustration, the rates have been estimated for 600K and 2 atm. The adsorption reaction only applies to MeCl(g), the only gas phase reactant. In the adsorption reaction, μ is the mass of MeCl(g), r_{gmv} is a gas velocity factor, and p_{con} is a unit conversion

factor. We do not necessarily include all possible reactions, since the rates quickly die off for barriers above ∼40 kcal/mol.

6.5 Preliminary Results

In our initial simulation runs, we found that Si atoms were filling the surface, blocking any MeCl(g) from landing and beginning the reaction chain. Thus we introduced a somewhat artificial way of limiting the number of Si atoms by preventing the introduction of more Si(int) if Si occupies more than some percent of the surface. This is exactly the sort of problem that a differential equation solver would be unable to detect. We are seeing that whatever choices are made for the input file, all the available empty sites are filled, inhibiting quite a few diffusion possibilities, or slowing the introduction of more Me or Cl.

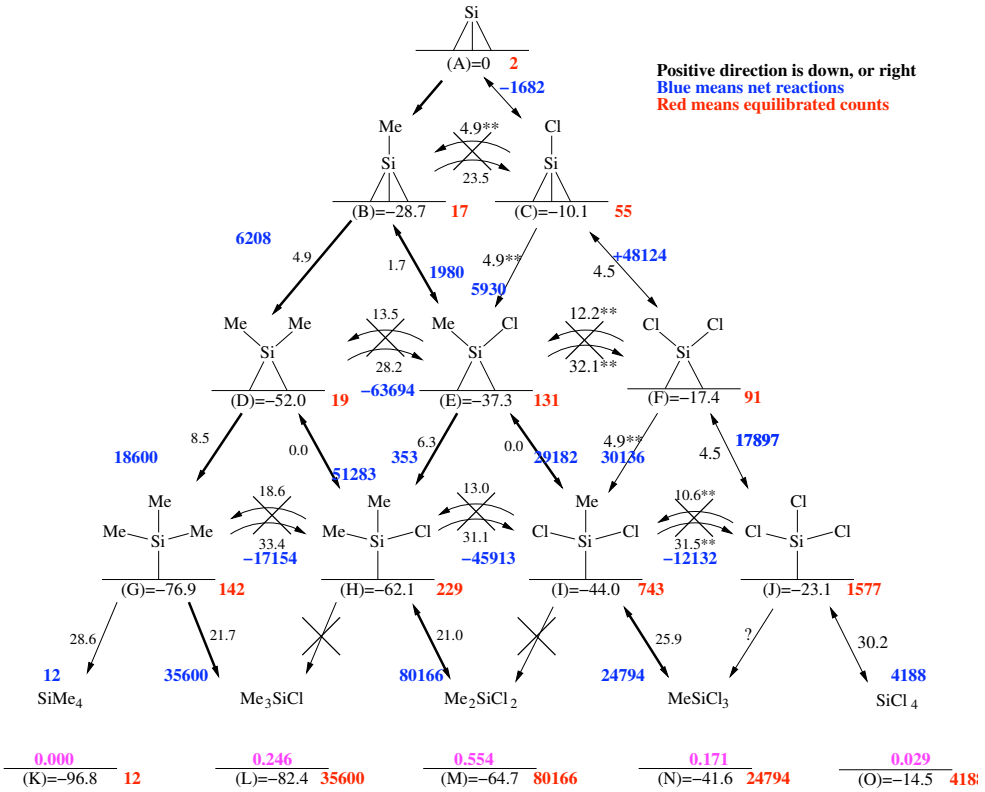

Figure 6.2: The reaction data was provided by Francesco Faglioni, annotated to include KMC data. In magenta, we've written the product fractions. The numbers shown are based on speculative energy barriers in our incomplete model, and are shown here for illustrative purposes only.

As demonstrated in Figure 6.2, we show the cascade of reactions representing all possible

Me_xSiCl_y species. During a calculation, we count the number of times each reaction occurs, as well as the number of times the reaction is reversed. At any point we wish, we calculate the net number of times a reaction occurred, count the number of each species currently on the grid, and print these numbers on the pyramid graph. We only include the reactions that relate one species directly to an adjacent species, meaning that some of the numbers do not add up, since there may be additional sources/sinks from off of the pyramid. For some of the species, though, all associated reactions are included, so if you add up all the sources and sinks, you will arrive at the species count shown.

This simulation ran for about 23 hours representing simulated time of 0.01 seconds from 29 billion iterations. The vast majority of these iterations were spent doing diffusion and other readily reversible reactions. In fact, only about 0.001% of the iterations did something interesting, so we will want to incorporate averaging techniques to increase this percent. This calculation was done on a 100x100 grid. Referring back to the scaling issue, if we were to double the grid to 200x100, the calculation should run for $23 * \mathcal{O}(\log(2N)) \sim 46$ hours, and if 200x200, then 69 hours. It is remarkable how fast $\mathcal{O}(\log N)$ scaling is.

These results are highly speculative, since they represent energy barriers that do not even include entropic effects. The model is not yet complete, and we are uncertain about whether we are correctly handling the introduction of Si or MeCl(g). However, the core of the algorithm is complete.

6.6 Conclusion

We have developed a new $\mathcal{O}(\log N)$ algorithm for KMC. This scaling is extremely fast, recommending further research into improving the model so that it can compete with more expensive methods. We have discussed an initial application, for which simulations as long as 0.01 seconds are easy to achieve. We believe that these results could make a significant impact in computational chemistry.

Appendix A

Asymptotic Scaling

As mentioned, there are a number of approximations one might take in order to simulate systems of molecules. With each approximation, the calculation becomes more simple, and thus will require less computer time. To describe computational complexity, we use big-O notation. We want to be able to study how much computer time a calculation will take, as a function of some parameter N like the number of particles, grid size, basis functions, orbitals, or something else. We take an algorithm and decompose it into tasks, and count how many primitive computer operations (e.g., +,-,*,/) it takes to complete each task as a function of N, and our calculation will look like a polynomial in powers of N. As an example, scalar operations are $\mathcal{O}(1)$, vector addition is $\mathcal{O}(N)$, matrix addition is an $\mathcal{O}\left(N^2\right)$ operation, matrix multiplication is $\mathcal{O}\left(N^3\right)$, Hartree-Fock and density functional theory (DFT) are $\mathcal{O}\left(N^4\right)$, and Full Configuration Interaction (FCI) calculations are $\mathcal{O}(N!)$. In a program with multiple tasks, these terms probably have constant prefactors, so that if a task has a complexity of $1000N^2 + N^3$, there will be a crossover point. In big-O notation, we just state the highest power and ignore the prefactors such that $\mathcal{O}\left(1000 * N^2\right) = \mathcal{O}\left(N^2\right)$, since asymptotically, it's only the highest power that really matters.

Although these scaling estimates are rigorous, they are not always useful because there are numerous additional approximations that can be added at each level in order to simplify the calculation. For example, matrix diagonalization is $\mathcal{O}\left(N^3\right)$, but if the matrix is sufficiently sparse, we can write it as a block diagonalized matrix. It now costs $\mathcal{O}\left(BM^3\right)$ to diagonalize the matrix, where B is the number of blocks and M is the dimension of a single block. If increasing the size of the calculation changes B but not M, then the cost of diagonalizing the block can be treated as a constant k, and the new price to pay is merely $\mathcal{O}(kB) = \mathcal{O}(B)$, or linear. This simplification in the computational complexity is typically

derived from localization in computational chemistry. Another example is in molecular dynamics, since even though the Coulomb interaction between two charges is substantial, even if they are far away, eventually, we will be able to truncate the term so that each molecule need only interact with the neighbors within some prescribed radius. This can lower the computational cost from $\mathcal{O}\left(N^2\right)$ down to a scaling of $\mathcal{O}\left(N \log N\right)$. Theoretically speaking, any of these methods could be lowered down to $\mathcal{O}\left(N\right)$ once the simulated system is large enough since eventually, two particles will be so far away that closer interactions will dominate. This can be described as "linear scaling" because even though a localized cluster retains the original computational scaling, adding another cluster only doubles the cost. That is, originally our scaling was measured in terms of the number of particles, but now we can measure in terms of the number of clusters, albeit with a large prefactor representing the time to calculate for each cluster. Although it's unclear how large a molecule must be for this to work, all *ab initio* methods are eventually linear in complexity. Unfortunately, the crossover point from $\mathcal{O}\left(N^k\right)$ to $\mathcal{O}\left(N\right)$ is for molecules much larger than we currently have processing power for.

Appendix B

The Local Energy

We have seen a number of different approaches in the literature, explaining how to evaluate the local energy of a wavefunction. Not very many of them present what we feel is the simplest, and therefore easiest to understand, formulation of the process, so we include it here. We start with the time-independent Schrödinger equation:

$$E|\Psi\rangle = \hat{H}|\Psi\rangle, \tag{B.1}$$

where $\hat{H}$ is the Hamiltonian for the system. For an isolated molecule, the Hamiltonian operator, in atomic units of energy, will be

$$\hat{H} = -\frac{1}{2}\sum_{i}^{N}\nabla_i^2 + \sum_{i>j}^{N}\frac{1}{r_{ij}} - \sum_{a}^{N_{nuc}}\sum_{i}^{N}\frac{Z_a}{R_{ai}} \tag{B.2}$$

$$= -\frac{1}{2}\sum_{i}^{N}\nabla_i^2 + V(\boldsymbol{r}), \tag{B.3}$$

where N is the number of electrons and N_{nuc} is the number of nuclei, Z_a is the charge on nucleus a, r_{ij} is the distance between electrons i and j, and R_{ai} is the distance between electron i and nucleus a. This equation does not take the motion of the nuclei into effect. Making the additional assumption that the wavefunction is normalized and real-valued, we

can write that

$$\begin{aligned}
\langle E\rangle &= \langle\Psi|\hat{H}|\Psi\rangle && \text{(B.4)}\\
&= \int \Psi(\boldsymbol{x})\hat{H}(\boldsymbol{x})\Psi(\boldsymbol{x})d\boldsymbol{x} && \text{(B.5)}\\
&= \int \Psi^2(\boldsymbol{x})\frac{1}{\Psi(\boldsymbol{x})}\hat{H}(\boldsymbol{x})\Psi(\boldsymbol{x})d\boldsymbol{x} && \text{(B.6)}\\
&= \int \rho(\boldsymbol{x})E_L(\boldsymbol{x})d\boldsymbol{x}, && \text{(B.7)}
\end{aligned}$$

where we have now defined the local energy as

$$E_L(\boldsymbol{x}) = \frac{\hat{H}(\boldsymbol{x})\Psi(\boldsymbol{x})}{\Psi(\boldsymbol{x})} = -\frac{1}{2}\sum_i^N \frac{\nabla_i^2\Psi(\boldsymbol{x})}{\Psi(\boldsymbol{x})} + V(\boldsymbol{r}), \tag{B.8}$$

where our notation implies that the operator must act on the wavefunction before we can divide by the wavefunction. A typical wavefunction in QMC will be the product of a determinant based wavefunction, which we will represent as ψ, with one or more configurations, times a Jastrow function, written as $e^U = e^{u_{12}}e^{u_{13}}e^{u_{13}}...$, the product of all particle interactions, which might involve 2 or 3 particles. For simplicity, we drop the explicit coordinate dependence of the wavefunction, and ignore the summation. If we use $\Psi = \psi e^U$, we find that

$$\begin{aligned}
E_L &= -\frac{1}{2}\frac{\nabla^2\left(\psi e^U\right)}{\psi e^U} + V && \text{(B.9)}\\
&= -\frac{1}{2}\left[\frac{\nabla^2\psi}{\psi} + 2\left(\frac{\nabla\psi}{\psi}\right)\cdot\nabla U + \nabla U\cdot\nabla U + \nabla^2 U\right] + V. && \text{(B.10)}
\end{aligned}$$

It is interesting to note that e^U is never explicitly evaluated here, although it is evaluated as a part of the Metropolis algorithm. Furthermore, since only derivatives of U show up in the local energy, the local energy is independent of any constant terms in U. We will not consider the evaluation of the derivatives of U here, which are typically polynomials or other simple to evaluate terms.

The evaluation of the gradient and laplacian terms of ψ is more complicated. In general, $\psi = \sum_t c_t D_{t\alpha} D_{t\beta}$, where ψ is a t indexed linear combination of determinants D, weighted by coefficient c_t, specific to each spin α or β. In a Hartree-Fock calculation, for example, there will only be one term in the summation, but there can be as many as millions of

terms in MCSCF or CI wavefunctions. QMC is typically used with less than a thousand for practical reasons. We can break the problem down into evaluating the derivatives of a single Slater determinant with respect to electronic coordinates and then build up to the gradient and laplacian of one determinant D. Now in our case, we are using Slater determinants as our antisymmeterizing operator,

$$D = |A| = \begin{vmatrix} \phi_1(r_1) & \phi_2(r_1) & \cdots & \phi_N(r_1) \\ \phi_1(r_2) & \phi_2(r_2) & & \\ \vdots & & \ddots & \\ \phi_1(r_N) & & & \phi_N(r_N) \end{vmatrix} \tag{B.11}$$

$$\phi_k(r_i) = \sum_j \chi_j(r_i) c_{jk} \tag{B.12}$$

$$\chi_j(r \leftarrow r_i - R_j) = r_x^{k_j} r_y^{l_j} r_z^{m_j} \sum_n a_{jn} e^{-b_{jn}|r|^2}, \tag{B.13}$$

where k indexes the orbital, j the basis function χ, and i the electron. The electron index i does not include all the electrons, but only the electrons with the same spin (α or β) go into the same determinant. Each term t in the wavefunction will use a different set of orbitals, so over all there will be more orbitals than electrons. However, each determinant represents one orbital for each electron, letting each electron "visit" all the orbitals. This means that if two of these electrons swap places, corresponding to swapping rows, then the determinant will change sign, satisfying the Pauli antisymmetry principle. It is obvious that by this construction, there is zero probability that two electrons will occupy the same location, or that two electrons will share the same orbital. Of course two electrons with different spins are also required to satisfy antisymmetry, but that happens another way.

For a matrix A, the Jacobi formula tells us that

$$\nabla|A| = |A| \mathrm{tr}\left(\nabla A \; A^{-1}\right) \tag{B.14}$$

and the derivative of A with respect to one of the coordinates of the i^{th} electron will just be the

$$\nabla_i \phi_k(r_i) = \sum_j \nabla_i \chi_j(r_i) c_{jk} \tag{B.15}$$

terms put on the k^{th} row in $\nabla_i A$, the only row that is a function of electron i, filling the rest of the matrix with zeros. Multiplying this by A^{-1} will leave only a single element on the diagonal,

$$\frac{1}{|A|}\nabla_i|A| = \text{tr}\left(\nabla_i A A^{-1}\right) = \sum_k \nabla_i \phi_k(r_i) A^{-1}_{ki}. \tag{B.16}$$

The second derivative of the determinant is more work, but all we need to know is that the trace is a linear operator and that $dA^{-1} = -A^{-1}\ dA\ A^{-1}$, so that

$$\begin{aligned}
\nabla^2|A| &= \nabla|A|\text{tr}\left(\nabla A\ A^{-1}\right) + |A|\text{tr}\left(\nabla^2 A\ A^{-1}\right) + |A|\text{tr}\left(\nabla A\ \nabla A^{-1}\right) && \text{(B.17)}\\
&= |A|\left(\text{tr}\left(\nabla A\ A^{-1}\right)\right)^2 + |A|\text{tr}\left(\nabla^2 A\ A^{-1}\right) + |A|\text{tr}\left(\nabla A\ \nabla A^{-1}\right) && \text{(B.18)}\\
\frac{1}{|A|}\nabla^2|A| &= \left(\text{tr}\left(\nabla A\ A^{-1}\right)\right)^2 + \text{tr}\left(\nabla^2 A\ A^{-1}\right) - \text{tr}\left(\nabla A\ A^{-1}\nabla A\ A^{-1}\right) && \text{(B.19)}\\
&= \text{tr}\left(\nabla^2 A\ A^{-1}\right). && \text{(B.20)}
\end{aligned}$$

Now we point out that, as before, the matrix ∇A is zero everywhere except row k. This multiplied by A^{-1} also results in a row matrix, and thus the third term in the above expression uses the product of two row matrices, the result of which is also a row matrix. This means we can cancel the first and third terms because the diagonal element in $\left(\nabla A\ A^{-1}\nabla A\ A^{-1}\right)$ is the square of the diagonal element of $\left(\nabla A\ A^{-1}\right)$. Therefore, by comparing with Equation B.16, we can see that

$$\frac{1}{|A|}\nabla_i^2|A| = \text{tr}\left(\nabla_i^2 A A^{-1}\right) = \sum_k \nabla_i^2 \phi_k(r_i) A^{-1}_{ki} \tag{B.21}$$

In software we can calculate all these terms simultaneously. First, we make 5 matrices, all dimensioned $N_{electrons} \times N_{basisfunctions}$. One is the evaluation of each basis function at the coordinates of each electron, three of the matrices are the x, y, and z first derivatives of each basis function with respect to each electron, and finally, the last matrix is the laplacian of each basis function with respect to each electron. These 5 matrices are all multiplied by the coefficient matrix dimensioned $N_{basisfunctions} \times N_{orbitals}$, to produce square matrices dimensioned $N_{electrons} \times N_{orbitals}$. When we generalize this to the calculation of many terms in ψ, since we will be doing the same algebra for each orbital, we might as well do them all at once, so the matrix multiplication does not produce square matrices, requiring us

to pull out the appropriate square matrices as needed. After this, we calculate both the determinant and inverse of the matrix A in one step using LU decomposition. Summing over all electrons, the laplacian is

$$\frac{1}{D}\nabla^2 D = \sum_i \frac{1}{D}\nabla_i^2 D = \sum_i \sum_k \nabla_i^2 \phi_k(r_i) D_{ki}^{-1} \tag{B.22}$$

which is merely a dot product, if the inverse matrix was stored in memory as its transpose. Calculating the gradient of determinant,

$$\frac{1}{|D|}\nabla_i |D| = \sum_k \nabla_i \phi_k(r_i) D_{ki}^{-1} \tag{B.23}$$

is only slightly more complicated than the laplacian only because we can not sum over the electrons. The final results for one determinant are 2 scalar quantities, one for the determinant itself and the other for the laplacian, and one matrix, dimensioned $N_{electrons} \times N_3$ for the gradient. For each term in ψ then, there will be twice as much data, half for the α electrons, and half for the β electrons. Although each term in ψ will use a different set of orbitals, the core orbitals will never change, and among the active space orbitals, only a few (of the columns in A) will change at a time. This means that we can use the Sherman-Morrison formula to update a column in the determinant, a procedure done once per different orbital compared to the previous term. In Generalized Valence Bond (GVB) wavefunctions, as explained in Appendix D, there is a simple way of sorting all the terms in ψ so that we only need to update one orbital to get the next determinant from the current determinant. Standard chain rule calculus is then used to combine results from the individual terms to find the gradient and laplacian of $\psi = \sum_t c_t D_{t\alpha} D_{t\beta}$.

As these steps make clear, there is a substantial amount of work that needs to be done to calculate the local energy for just one electronic configuration. The relative expense of each step depends on $N_{basisfunctions}$, $N_{orbitals}$, and $N_{electrons}$. For augmented basis sets, $N_{basisfunctions} \simeq 10\ N_{electrons}$, meaning that the matrix multiplication step will take approximately $N_{basisfunctions} * N_{electrons} * N_{orbitals} \simeq 10\ N_{electrons}^3$ computer operations. By comparison, the matrix inversion step will cost about $N_{electrons}^3$, since the product of the matrix multiplication is much smaller. This means that the matrix multiplication step will dominate the expense asymptotically. For smaller molecules, the cost evaluating the basis

functions will dominate the calculation because even though this cost will scale as only 10 $N_{electrons}^2$, each basis function evaluation requires us to calculate the e^x function.

One variation on the procedure outlined above is the ability to update just one electron per iteration, instead of all of them. In this case, the matrix multiplication turns into a vector-matrix multiplication, and instead of LU decomposition to calculate the inverse of the matrix, we would use the Sherman-Morrison formula to update rows in our Slater determinant. The motivation for this is that it will substantially raise the acceptance probability, which might drop quite low for large molecules. But if we move one electron at a time, there are several computational penalties to consider.

First of all, if we move all electrons each iteration, then we can see that we do not need to keep any of the intermediate data, since the next step will start from scratch. The fact that we do not need to keep it means that when we move to evaluating the local energy for the next walker (which has a different electronic configuration) then the same memory is immediately available to us and our memory requirement does not increase with the number of walkers we are using. If on the other hand we are evaluating one electron at a time, then we must keep the data, because we will need it during the next iteration when updating the determinant and inverse. For single electron updates, our memory requirements will scale linearly with the number of walkers we want to use.

Secondly, the number of algebra steps during the matrix multiplication stage remains the same, since the matrix multiplication will take exactly as many operations a vector-matrix multiplication done $N_{electrons}$ times. But because we are doing this on a computer, the cost will not be exactly equivalent. The reason is because it takes a computer some time to load memory into high level cache in order to do the operations efficiently. Furthermore, efficient matrix multiplication routines are able to reuse memory. That is, once some values are loaded into registers, they can be used to update several elements of the product matrix simultaneously. Although not all is lost during a vector-matrix multiplication, we do incur a computational penalty.

Thirdly, the number of operations involved in updating a matrix one row (or column) at a time is twice as many operations as it would have taken do perform the whole inversion in one step. Furthermore, all of the subsequent steps, inconsequential though they were before, will now be $N_{electrons}$ times expensive because we have to do all of it each time an electron is updated. It will depend on a number of other factors to determine the final

impact on the calculation.

We have run several test cases to compare the relative merits of one electron or all electron updates. In our measurements, we ran a calculation to a predetermined error level. Because our software is able to decorrelate serial iterations on the fly, the software knew exactly when to stop. To compare then, we simply look at how long it took to complete each calculation, and we found that for methylene, one electron updating took twice as long if when we used all electron updates.

Although QMC algorithms are more sophisticated than what has been presented here, all of them have to work through these steps. Most codes, including ours, allow individual electron updates.

Appendix C

Jaguar Initial GVB Guesses and GAMESS

I have found this script to be very useful because Jaguar does an excellent job of generating initial guesses [53] for GVB wavefunctions, whereas GAMESS is very hard to use. On the other hand, Jaguar is somewhat limited in terms of what types of SCF calculations it can do, whereas GAMESS is quite general and capable, once you have a good initial guess. For example, if you want to run a CASSCF calculation using the GVB orbitals, then you will want to mix the capabilities of both software packages. With these considerations in mind, I developed this script to take a Jaguar wavefunction, and convert it to a GAMESS input file.

It turns out that Jaguar and GAMESS order their f basis functions differently, and this script does not attempt to fix this. However, this is not a problem if you reconverge the GVB wavefunction in GAMESS before you do anything else with the wavefunction. The other limitation of this script is that Jaguar can not handle basis functions beyond f. Perhaps this limitation will be fixed for Jaguar in the future. In the meantime, a useful future modification to this script might be to allow it to add extra basis functions, initializing their coefficients to zero.

C.1 Script: jaguar2gamess.pl

```
1  #!/usr/bin/perl
2
3  #input input file, $header, @orbitals
4  #output num electrons
5  sub parseJagInput {
6      my ( $filename, $sub_Header, $sub_Orbitals ) = @_;
7
```

```
8       my $sub_Orbital;
9       my $sub_Protons = 0;
10      $$sub_Header = "";
11
12      open IN, "<$filename" or die;
13      my $read = 2;
14      my $line = <IN>;
15      while ($line) {
16
17          if ( $read == 0 ) {    #nothing special so far
18
19              if ( $line =~
20  /(\s+)(\d+) Orbital Energy\s+([\-0-9.]+)\s+Occupation\s+([\-0-9.]+)/
21                )
22              {
23
24                  #we only want to grab the occupied orbitals
25                  if ( $4 >= 0.0 ) {
26                      $sub_Protons += 2.0 * $4;
27                      printf
28                        "Orbital %2i has occupation %20.10e and energy %20.10f\n",
29                        $2, $4, $3;
30                      $read = 1;
31
32                      #$sub_Orbital = $line;
33                      $sub_Orbital = "";
34                  }
35              }
36              $line = <IN>;
37
38          }
39          elsif ( $read == 1 ) {    #we're reading an orbital
40
41              if ( $line =~
42  /(\s+)(\d+) Orbital Energy\s+([\-0-9.]+)\s+Occupation\s+([\-0-9.]+)/
43                )
44              {
45                  $read = 0;
46                  push( @$sub_Orbitals, $sub_Orbital );
47              }
48              else {
49                  $sub_Orbital .= $line;
50                  $line = <IN>;
51              }
52
53          }
54          elsif ( $read == 2 ) {    #we're reading in the header
55
56              if ( $line =~
57  /(\s+)(\d+) Orbital Energy\s+([\-0-9.]+)\s+Occupation\s+([\-0-9.]+)/
58                )
59              {
60                  $read = 0;
61              }
62              else {
63                  $$sub_Header .= $line;
64                  $line = <IN>;
65              }
66
67          }
68      }
69
70      close IN;
71      return $sub_Protons, $#$sub_Orbitals + 1;
72  }
73
74  sub printGamessOrb {
```

```
75      my ( $index, $orb ) = @_;
76
77      if ( $$orb eq "" ) {
78          print "Error: orbital $index is blank!\n";
79          die;
80      }
81
82      my $output = "";
83      $$orb =~ s/^\s+//;
84      @coeffs = split /\s+/, $$orb;
85
86      $count = 0;
87      $linec = 0;
88      foreach $co (@coeffs) {
89          if ( $count % 5 == 0 ) {
90              $linec += 1;
91              $output .= sprintf "\n%2i%3i", $index % 100, $linec;
92          }
93          $output .= sprintf "%15.8e", $co;
94          $count += 1;
95      }
96      return $output;
97  }
98
99  die "Need Jaguar restart (with ip168=2) file, not $ARGV[0]\n"
100   if ( $#ARGV < 0 || $ARGV[0] !~ /.\d\d.in$/ );
101 if ( $ARGV[1] ) {
102     open( OUTFILE, ">$ARGV[1]" );
103     $outfh = *OUTFILE;
104 }
105 else {
106     $newfile = $ARGV[0];
107     $newfile =~ s/.01.in/.inp/;
108     open( OUTFILE, ">$newfile" );
109     print "Writing $newfile\n";
110     $outfh = *OUTFILE;
111
112     #$outfh = *STDOUT;
113 }
114
115 $file   = $ARGV[0];
116 $output = $file;
117 $output =~ s/.01.in/.out/;
118 $gamin = $file;
119 $gamin =~ s/.01.in/.gamess/;
120
121 my @orbitals;
122 my $header;
123 ( $numP, $numO ) = parseJagInput( $file, \$header, \@orbitals );
124
125 my %gvb_coeffs;
126 my %gvb_pairs;
127 $doubleocc = 0;
128 $singleocc = 0;
129 open( OUTPUT, "<$output" );
130 while (<OUTPUT>) {
131     $doubleocc = ( split /\.+/ )[1] if (/number of doubly-occ/);
132     $doubleocc = int($doubleocc);
133     $singleocc = ( split /\.+/ )[1] if (/number of open shell orbs/);
134     $singleocc = int($singleocc);
135
136     if (/first natural orbital/) {
137         $_ = <OUTPUT>;
138         $_ = <OUTPUT>;
139         $_ = <OUTPUT>;
140         $_ = <OUTPUT>;
141         while (/[0-9]/) {
```

```
142             @pairdata = split /\s+/;
143             $gvb_pairs{ $pairdata[2] } = $pairdata[6];
144             $gvb_coeffs{ $pairdata[2] } = sprintf "%11.8f,%11.8f", $pairdata[5],
145               $pairdata[9];
146
147             #printf
148             print "pair $gvb_coeffs{$pairdata[2]} $_";
149             $_ = <OUTPUT>;
150         }
151
152     }
153 }
154
155 my $npair = keys(%gvb_pairs);
156 my $norb  = $doubleocc + $singleocc + 2 * $npair;
157
158 print "ERROR: GAMESS can't handle more than 12 GVB pairs!" if ( $npair > 12 );
159 print
160 "Start printing GAMESS input file with $norb orbs: $npair pairs, $doubleocc doubly occ orbs, $singleocc singly occ orbs
161 printf $outfh " \$SCF NCO=%i NSETO=%i NPAIR=%i", $doubleocc, $singleocc, $npair;
162
163 if ( $singleocc > 0 ) {
164     printf $outfh " NO(1)=";
165     for ( my $so = 0 ; $so < $singleocc ; $so++ ) {
166         printf $outfh "1,";
167     }
168
169 }
170
171 if ( $npair > 0 ) {
172     printf $outfh "\n CICOEF(1)=";
173     foreach $key ( sort keys %gvb_pairs ) {
174         printf $outfh "$gvb_coeffs{$key},\n";
175     }
176 }
177 printf $outfh " \$END\n";
178 printf $outfh " \$GUESS GUESS=MOREAD NORB=%i PRTMO=.TRUE. \$END\n", $norb;
179 printf $outfh " \$SYSTEM MWORDS=200 \$END\n",                       $norb;
180
181 open( GAMESS, "<$gamin" );
182 while (<GAMESS>) {
183     if ( /contrl/ && $npair > 0 ) {
184         chomp;
185
186         #I need to intercept the contrl group to change the scftyp
187         printf $outfh "$_ maxit=100 scftyp=gvb\n";
188     }
189     else {
190
191         #The $data group is already good to go
192         printf $outfh $_;
193     }
194 }
195
196 print $outfh " \$VEC";
197
198 $orbIndex = 1;
199 for ( my $i = 1 ; $i <= $doubleocc + $singleocc ; $i += 1 ) {
200     $orb = $orbitals[ $i - 1 ];
201     print $outfh printGamessOrb( $orbIndex, \$orb );
202     if ( $i <= $doubleocc ) {
203         print "Closed orbital $i\n";
204     }
205     else {
206         print "Open   orbital $i\n";
207     }
208     $orbIndex += 1;
```

```
}

my $pairCount = 1;
foreach $key ( sort keys %gvb_pairs ) {
    print "GVB ${pairCount}u orbital $orbIndex <-- $key\n";
    $orb = $orbitals[ $key - 1 ];
    print $outfh printGamessOrb( $orbIndex, \$orb );
    $orbIndex += 1;

    print "GVB ${pairCount}v orbital $orbIndex <-- $gvb_pairs{$key}\n";
    $orb = $orbitals[ $gvb_pairs{$key} - 1 ];
    print $outfh printGamessOrb( $orbIndex, \$orb );
    $orbIndex  += 1;
    $pairCount += 1;
}

print $outfh "\n \$END\n";
```

Appendix D

Making the .ckmf file

This script will convert a GAMESS output file into a QMcBeaver input file, and has been the subject of many bug fixes by Amos Anderson (myself) and Dan Fisher, as well as the original developers, Mike Feldmann and Chip Kent. We have programmed it to handle a variety of different wavefunctions, but we have not made an effort to get it to handle all possible minutia of a GAMESS calculation. This script will look for a *.ckmft* file on which to base the input file it produces, and we provide the *.ckmft* file below.

One point of interest is how this script chooses to sort the determinants. For a GVB wavefunction, there is a simple ordering available wherein each determinant differs from the previous determinant by only one orbital. This means that the QMcBeaver code only needs to run one Sherman-Morrison column update on each determinant to get to the next determinant. Any other sorting might involve several updates per determinant. For wavefunctions with many determinants, these savings add up significantly. This has been embedded in the loop recursion near line 539, and here we provide an example for a 3 pair wavefunction. The 2^3 determinants in the 6 orbitals look like

```
    a b c d e f
1.  1 0 1 0 1 0
2.  0 1 1 0 1 0
3.  0 1 0 1 1 0
4.  1 0 0 1 1 0
5.  1 0 0 1 0 1
6.  0 1 0 1 0 1
7.  0 1 1 0 0 1
8.  1 0 1 0 0 1,
```

where 1 indicates that orbital is occupied, and a 0 indicates that orbital is not occupied

for that determinant. For the sake of the discussion, we assume there are no core orbitals. Each determinant will contain a different set of 3 orbitals. To go from determinant 1 to determinant 2, we only need to swap the first orbital from a to b. To go from determinant 2 to 3, we swap the second orbital from c to d. Then we swap the first orbital from b to a, then the third orbital from e to f, and so on. That is, moving from one determinant from the next only involves one update. Consider the following example for the 8 highest coefficient determinants from a 6 orbital CAS wavefunction.

```
    a b c d e f
1.  1 1 1 0 0 0
2.  1 1 0 1 0 0
3.  1 0 1 0 1 0
4.  0 1 1 1 0 0
5.  1 1 0 1 0 0
6.  0 1 1 0 0 1
7.  0 1 1 1 0 0
8.  0 1 1 0 0 1
...
```

In this case, we had to update 1, 2, 2, 2, 3, 1, 1, ... columns in our progression through the determinants, for a total of 5 more updates than the GVB wavefunction required, which is almost twice as expensive! This is yet another advantage of GVB wavefunctions. It is likely that there are ways to minimize this kind of expense for CAS wavefunctions, but we have not worked out the details.

D.1 Script: gamess2qmcbeaver.py

```
#!/usr/bin/env python

# This script will convert the output from a GAMESS calculation into an input
# file for QMcBeaver.
# * It will copy all the basis function data and orbitals
# * It will look for the energies calculated in GAMESS
#   and add them as comments.
# * To find a good set of QMcBeaver flags, it will look for a "ckmft" file
#   in a few directories (see "templatedir" variable below)
#   to copy a good set of defaults.
#
# Usage:
# gamess2qmcbeaver.py <GAMESS output file> [determinant cutoff = 0.0]
# * We recognize .log and .inp.out as GAMESS output extensions.
# * If the absolute value of the CI coefficient is below the determinant
#   cutoff, then it will not be included in the ckmf file.
#
# Permissible RUNTYP = ENERGY and OPTIMIZE
# Permissible SCFTYP = anything other than MCSCF
```

```
#
# To use SCFTYP=MCSCF:
# 1) Run the MCSCF calculation in GAMESS. This is the hardest part... Look in
#    the GAMESS manual and the "Further Information" document for hints.
# 2) Make a 2nd GAMESS input file with a $VEC section from the natural orbitals
#    and minimized geometry of the MCSCF run. Specify SCFTYP=NONE and CITYP=ALDET.
#    This will produce a CI expansion in these orbitals. You might be able to use
#    other CITYP, but we haven't programmed them.
# 3) This script can read the ALDET output file, and will find as many determinants
#    as were printed out, and put them in the ckmf file. You might need to modify
#    PRTTOL in the $DET section to get more determinants.
#
# NOTE: check your ALDET runs... I've found that the occupations don't always match
# the orbitals printed! For one of my runs, it sorted the natural orbitals according to occupation,
# which was different from the input order.

import re
import sys
import copy
import math
import string
import time
import os
from utilities import *

if len(sys.argv) < 2:
    print "gamess2qmcbeaver.py <filename>[.log, .inp.out] [detcutoff=0.0]"
    sys.exit(0)

Infile = sys.argv[1]
IN = open(Infile,'r')
gamess_output = IN.readlines()
IN.close()

filebase = ""
if string.find(Infile,'.inp.out') != -1:
filebase = sys.argv[1][0:len(sys.argv[1])-7]
elif string.find(Infile,'.log') != -1:
filebase = sys.argv[1][0:len(sys.argv[1])-3]
else:
print "The file ", Infile, " is not recognized as a GAMESS log file!"
sys.exit(0)

Datafile = filebase + "dat"
IN2 = open(Datafile,'r')
gamess_data = IN2.readlines()
IN2.close()

Outfile = filebase + "ckmf"
OUT = open(Outfile,'w')

run_type  = "ENERGY"
scf_type  = "RHF"
ci_type   = "NONE"
pp_type   = "NONE"
spin_mult = 1
istate    = 1

detcutoff = 1e-10
if len(sys.argv) == 3:
    detcutoff = string.atof(sys.argv[2])
    print "Removing all determinants with coefficients less than ",detcutoff

# Get run type and scf type

for i in range(len(gamess_output)):
    if string.find(gamess_output[i],'$CONTRL OPTIONS') != -1:
```

```
k = i
while string.find(gamess_output[k],'$SYSTEM OPTIONS') == -1:
    line = re.split('[\s=]+',gamess_output[k])
    for j in range(len(line)):
if string.find(line[j],'SCFTYP') != -1:
    scf_type = line[j+1]
if string.find(line[j],'VBTYP') != -1:
    if string.find(line[j+1],'NONE') == -1:
scf_type = line[j+1]
if string.find(line[j],'RUNTYP') != -1:
    run_type = line[j+1]
if string.find(line[j],'CITYP') != -1:
    ci_type = line[j+1]
if string.find(line[j],'MULT') != -1:
    spin_mult = string.atoi(line[j+1])
if string.find(line[j],'PP') != -1:
    pp_type = line[j+1]
    k += 1

if ci_type == "GENCI":
    #These are effectively the same kind of calculation
    #Just different lists of determinants
    ci_type = "ALDET"

#################### EXTRACT GEOMETRY: START ####################

# Find where the geometry is stored.

if run_type == "ENERGY" or run_type == "HESSIAN":
    for i in range(len(gamess_output)):
        if string.find(gamess_output[i], 'RUN TITLE') != -1:
            start_geometry = i
        if string.find(gamess_output[i], 'INTERNUCLEAR DISTANCES') != -1:
            end_geometry = i
            break

elif run_type == "OPTIMIZE":
    for i in range(len(gamess_output)):
        if string.find(gamess_output[i],'EQUILIBRIUM GEOMETRY LOCATED') != -1:
            start_geometry = i
            for j in range(i,len(gamess_output)):
                if string.find(gamess_output[j],'INTERNUCLEAR DISTANCES') !=-1:
                    end_geometry = j-1
                    break
                elif string.find(gamess_output[j],'INTERNAL COORDINATES') !=-1:
                    end_geometry = j-3
                    break
                elif string.find(gamess_output[j],'SUBSTITUTED Z-MATRIX') !=-1:
                    end_geometry = j-1
                    break
            break

try:
    geom_data = gamess_output[start_geometry:end_geometry]
except:
    print "Failed to find geometry for run_type = ", run_type
    raise
geometry = []

start = 0

if run_type == "ENERGY":
    for line in geom_data:
        if start: geometry = geometry + [line]
        if string.find(line,'CHARGE') != -1: start = 1
    geometry = geometry[:len(geometry)-1]

```

```
elif run_type == "OPTIMIZE":
    for line in geom_data:
        if start == 2: geometry = geometry + [line]
        if string.find(line,'CHARGE') != -1: start = start + 1
    geometry = geometry[1:]

#split up the data
for i in range(len(geometry)):
    geometry[i] = string.split(geometry[i])
    for j in range(2,5):
        geometry[i][j] = string.atof(geometry[i][j])

#convert from ANGs to BOHR if necessary

ANGtoBOHRconversion = 1.0/0.529177249

for line in geom_data:
    if string.find(line,'(ANGS)') != -1:
        for i in range(len(geometry)):
            for j in range(2,5):
                geometry[i][j] = geometry[i][j] * ANGtoBOHRconversion
        break

#################### EXTRACT GEOMETRY: END ######################

#################### EXTRACT BASIS SET: BEGIN ###################

start_basis = 0
end_basis = 0
for i in range(len(gamess_output)):
    if string.find(gamess_output[i], 'ATOMIC BASIS SET') != -1:
        start_basis = i
    if string.find(gamess_output[i], '$CONTRL OPTIONS') != -1:
        end_basis = i
        break
basisdata = gamess_output[start_basis:end_basis]

end = 0
for i in range(len(basisdata)):
    if string.find(basisdata[i],'TOTAL NUMBER OF SHELLS') != -1 :
        end = i
        break
    if string.find(basisdata[i],'TOTAL NUMBER OF BASIS SET SHELLS') != -1 :
        end = i
        break
basisdata = basisdata[7:end]

basis = []
atom = []
bf = []
atomnumber = -1
bfnumber = 0
for line in basisdata:
    if line != '\n':
        line = string.replace(line,')',' ')
        line = string.replace(line,'(',' ')
        line = string.split(line)
        if len(line) == 1:
            # We are starting a new atom
            if bf != []:
                # We add the old contracted basis function to the atom and clear the temp
                atom = atom + [bf]
                bf = []
            if atom != []:
                # We add the old atom to the basis and clear the temp space
                basis = basis + [atom]
                atom = []
```

```
            # We start the new atom with the label
            atom = atom + [line]

        elif len(line) > 1 and line[0] != str(bfnumber):
            bfnumber = string.atoi(line[0])
            # We are starting a new contracted basis function for this atom
            if bf != []:
                # We add the old contracted basis function to the atom and clear the temp
                atom = atom + [bf]
                bf = []
            # We start the new contracted basis function
            temp = [line[1]] + line[3:]
            if len(line) > 6:
                temp = temp + [line[6]]
            line = temp
            bf = [line]

        elif len(line) > 1 and line[0] == str(bfnumber):
            # We are continuing to add primitive basis functions to the contracted one
            temp = [line[1]] + line[3:]
            if len(line) > 6:
                temp = temp + [line[6]]
            line = temp
            bf = bf + [line]
atom = atom + [bf]
basis = basis + [atom]

# extract some flags data from this section
for line in gamess_output:
    if string.find(line,'TOTAL NUMBER OF BASIS FUNCTIONS') !=-1 :
        nbasisfunc = string.atoi(string.split(line)[6])
    if string.find(line,'NUMBER OF CARTESIAN GAUSSIAN BASIS FUNCTIONS') !=-1 :
        nbasisfunc = string.atoi(string.split(line)[7])
    if string.find(line,'CHARGE OF MOLECULE') !=-1 :
        charge = string.atoi(string.split(line)[4])
#we'll rely on orbital occupations to indicate charge
# charge = 0
    if string.find(line,'TOTAL NUMBER OF ATOMS') !=-1 :
        atoms = string.atoi(string.split(line)[5])
    if string.find(line,'NUMBER OF OCCUPIED ORBITALS (ALPHA)') != -1 :
nalpha = string.atoi(line.split('=')[1])
    if string.find(line,'NUMBER OF OCCUPIED ORBITALS (BETA )') != -1 :
nbeta = string.atoi(line.split('=')[1])

##################### EXTRACT BASIS SET: END ########################

##################### EXTRACT WAVEFUNCTION: BEGIN ###################


# First, we want to load in all the $VEC .. $END sections we can find.
# We look in both the .dat file, and in the .inp file, and we save
# the name that GAMESS gave it.
collecting = 0
Inputfile = filebase + "inp"
INPFILE = open(Inputfile,'r')
input_data = INPFILE.readlines()
INPFILE.close()

name = "MOREAD orbitals from " + Inputfile + "\n"
raw_orbitals = []
orbital_vecs = []
orbital_name = []
iorder_vecs  = []
norder_vecs  = 0
for i in range(len(input_data)):
    m = re.search('norder\s*=(\d+)',input_data[i],re.I)
    if m:
```

```
288 norder_vecs = int(m.group(1))
289     m = re.search('iorder\((\d+)\)=([\d,]+)',input_data[i],re.I)
290     if m:
291 iorder_vecs.append(int(m.group(1)))
292 iorder_vecs += [int(k) for k in m.group(2).split(',')]
293 
294     if string.find(input_data[i],'$END') != -1 and collecting == 1:
295 collecting = 0
296 orbital_vecs = orbital_vecs + [raw_orbitals]
297 orbital_name = orbital_name + [name]
298 raw_orbitals = []
299     if string.find(input_data[i],'$VEC') != -1:
300 collecting = 1
301     elif collecting == 1:
302 raw_orbitals = raw_orbitals + [input_data[i]]
303 
304 if norder_vecs == 1:
305     print "Notice: found IORDER section for MOREAD orbitals: ",iorder_vecs
306 
307 for i in range(len(gamess_data)):
308     if string.find(gamess_data[i],'NO-S OF CI STATE') != -1 or \
309    string.find(gamess_data[i], 'GVB ORBITALS') != -1 or \
310    string.find(gamess_data[i], 'LOCALIZED') != -1 or \
311    string.find(gamess_data[i], 'OPEN SHELL ORBITALS') != -1 or \
312    string.find(gamess_data[i], 'CLOSED SHELL ORBITALS') != -1 or \
313    string.find(gamess_data[i], 'MP2 NATURAL ORBITALS') != -1 or \
314    string.find(gamess_data[i], 'OPTIMIZED MCSCF') != -1 or \
315    string.find(gamess_data[i], 'NATURAL ORBITALS OF MCSCF') != -1:
316 name = gamess_data[i]
317     if string.find(gamess_data[i],'$END') != -1 and collecting == 1:
318 collecting = 0
319 orbital_vecs = orbital_vecs + [raw_orbitals]
320 orbital_name = orbital_name + [name]
321 raw_orbitals = []
322     if string.find(gamess_data[i],'$VEC') != -1:
323 collecting = 1
324     elif collecting == 1:
325 raw_orbitals = raw_orbitals + [gamess_data[i]]
326     m=re.search('^E\([\w\-]+\)=\s*([\d\-\.]+)',gamess_data[i])
327     if m:
328 energy = m.group(1)
329     m=re.search('CI STATE\s+\d+\sE=\s*([\d\-\.]+)',gamess_data[i])
330     if m:
331 energy = m.group(1)
332 
333 if scf_type == "RHF":
334     default_orb_string = 'CLOSED SHELL ORBITALS'
335 elif scf_type == "ROHF":
336     default_orb_string = 'OPEN SHELL ORBITALS'
337 elif scf_type == "UHF":
338     default_orb_string = ''
339 elif scf_type == "GVB":
340     default_orb_string = 'GVB ORBITALS'
341 elif scf_type == "NONE" and ci_type == "ALDET":
342     default_orb_string = 'MOREAD'
343 elif scf_type == "VB2000":
344     cicoef=[]
345     for i in range(len(gamess_output)):
346         if string.find(gamess_output[i],'Normalized structure coefficients') != -1:
347     line = gamess_output[i+1]
348             cicoef += string.split(line)
349         if string.find(gamess_output[i],'ENERGY AND DIFF OF MACROITER') != -1:
350     line = gamess_output[i]
351             energy = (string.split(line))[7]
352 
353     pcum = 0
354     for i in range(len(cicoef)):
```

```
cicoef[i] = string.atof(cicoef[i])
pcum += cicoef[i]*cicoef[i]
    print "VB Coeff = ",cicoef, "\nnorm = ",pcum
    print "VB Energy = ",energy

    print "Fix VB2000 to get the right orbitals!";
    sys.exit(0)
    Datafile = filebase + "vec"
    IN2 = open(Datafile,'r')
    gamess_data = IN2.readlines()
    IN2.close()
else:
    print "SCFTYP", scf_type, "is not supported."
    sys.exit(0)

orb_choice = len(orbital_name)-1
print "We found %i orbital sets:"%len(orbital_name)
for i in range(len(orbital_name)):
    lines_per_orb = int(nbasisfunc/5)+1
    print "%i) %g orbitals"%(i,((len(orbital_vecs[i]))/float(lines_per_orb))),
    print "for: ", orbital_name[i],
    if string.find(orbital_name[i],default_orb_string) != -1:
orb_choice = i
try:
    orb_choice = string.atoi(raw_input("Your choice [%i]:"%orb_choice))
except:
    print "",

# Get the wavefunction parameters from the .dat file.
orbital_number = []
orbital_coeffs = []
wavefunction   = []
current_index  = 1
for n in range(len(orbital_vecs[orb_choice])):
    orbital_index = string.atoi(orbital_vecs[orb_choice][n][0:2])

    #if the index changed, then we completed the old orbital, so we add it to the wf
    if orbital_index != current_index:
wavefunction.append(orbital_coeffs)
current_index = orbital_index
orbital_coeffs = []

    #turn the text into numbers: index coeff1 coeff2 coeff3 coeff4 coeff5
    len_line = len(orbital_vecs[orb_choice][n])
    number_of_entries = len_line/15
    line_data = range(number_of_entries)
    for i in range(number_of_entries):
        line_data[number_of_entries-i-1] = \
                    orbital_vecs[orb_choice][n][len_line-15*(i+1)-1:len_line-15*i-1]
    for j in range(len(line_data)):
        line_data[j] = string.atof(line_data[j])

    #append the current line to the current orbital
    orbital_coeffs = orbital_coeffs + line_data

#Add that last orbital
wavefunction.append(orbital_coeffs)
norbitals = len(wavefunction)

if re.search("MOREAD",orbital_name[orb_choice],re.I) and len(iorder_vecs) > 0 and norder_vecs == 1:
    print "Reordering according to IORDER: ",
    new_orbitals = []
    num = len(iorder_vecs)-1
    print num, ", first = ",iorder_vecs[0]
    for i in range(iorder_vecs[0]-1):
print i+1,
new_orbitals.append(wavefunction[i])
```

```
    for i in range(num):
print iorder_vecs[i+1],
new_orbitals.append(wavefunction[iorder_vecs[i+1]-1])
    for i in range(iorder_vecs[0]-1+num,norbitals):
print i+1,
new_orbitals.append(wavefunction[i])
    print "\n"
    wavefunction = new_orbitals

################ Set up the occupation and CI coefficent arrays.
if scf_type == "RHF" or scf_type == "ROHF" or scf_type == "UHF":
    AlphaOcc    = [0]
    BetaOcc     = [0]
    AlphaOcc[0] = range(norbitals)
    BetaOcc[0]  = range(norbitals)

    for j in range(0,norbitals):
        AlphaOcc[0][j] = 0
        BetaOcc[0][j]  = 0

    for i in range(nalpha):
        AlphaOcc[0][i] = 1

    #The VEC section is double in size... The first half are the alpha
    #orbitals, and the second half are the beta orbitals... If that statement
    #isn't always true, then this will have problems.
    beta_start = 0
    if scf_type == "UHF":
if norbitals % 2 == 0:
    beta_start = norbitals/2
else:
    print "Error: unexpected problem reading UHF wavefunction...\n"
    sys.exit(0)

    for i in range(beta_start,nbeta+beta_start):
        BetaOcc[0][i] = 1

    ncore = 0
    ndeterminants = 1
    CI = [1]

elif scf_type == "GVB":
    #the CI Coefficients in the dat file are more precise, so we want them
    #this might have a problem if too many GVB pairs are used
    cicoef=[]
    for i in range(len(gamess_data)):
        if string.find(gamess_data[i],'CICOEF') != -1:
    line = gamess_data[i]
    p = re.compile("\(\s+")
    line = p.sub("(",line)
    line = string.replace(line,',',' ')
            cicoef += string.split(line)

    core_line_number = -1
    start_ci = 0
    end_ci = 0
    for i in range(len(gamess_output)):
        if string.find(gamess_output[i],'ROHF-GVB INPUT PARAMETERS') != -1:
            core_line_number = i+3
            core_line = string.split(gamess_output[core_line_number])
    norb = string.atoi(core_line[2])
            ncore = string.atoi(core_line[5])
    pair_line = string.split(gamess_output[core_line_number+1])
    npair = string.atoi(pair_line[2])
    nseto = string.atoi(pair_line[5])
    odegen = 0
    if re.search("NO",gamess_output[core_line_number+2],re.I):
```

```
no_line = string.split(gamess_output[core_line_number+2])
odegen  = string.atoi(no_line[2])

    print "GVB settings: mult=",spin_mult,"ncore=",ncore,"norb=",norb,"npair=",npair,"nseto=",nseto

    #if odegen > 0:
# print "Error: open shell too complicated, no = ",odegen
# sys.exit(0)
        #    break

    AlphaOcc = range(1)
    CI = range(1)
    CI[0] = 1.0

    for i in range(len(AlphaOcc)):
        AlphaOcc[i] = range(norbitals)

    for i in range(len(AlphaOcc)):
        for j in range(norbitals):
    if j < ncore:
AlphaOcc[i][j] = 1
    else:
AlphaOcc[i][j] = 0

    if npair == 0:
ndeterminants = 1
ncore = 0

    if npair > 0:
#The two perfect paired electrons are spin coupled into a singlet
# See Eq 56 from "SCF Equations for GVB" by Bobrowicz and Goddard
# WF = anti[(c1 11 - c2 22)ab] = c1 anti[11ab] - c2 anti[22ab]

for j in range(core_line_number,len(gamess_output)):
    if string.find(gamess_output[j],'CI COEFFICIENTS') != -1:
start_ci = j+2
break

for i in range(start_ci,len(gamess_output)):
    ci_line = string.split(gamess_output[i])
    if len(ci_line) != 9:
end_ci = i
break

# we need to expand out the geminal pairs into separate determinants
# the start in the form (c1 + c2) * (c3 + c4) * (c5 + c6)...
# expanding to (c1*c3 + c1*c4 + c2*c3 + c2*c4)*(c5 * c6)*(...)
# I've implemented a recursion in a loop.
index = 2
idet = 0
for p in range(npair):
#for p in range(npair-1,-1,-1):
    old = len(CI)
    coef1 = string.atof(cicoef[index])
    index += 1
    coef2 = string.atof(cicoef[index])
    index += 2

    ci_line = string.split(gamess_output[p+start_ci])
    orb1 = string.atoi(ci_line[1]) - 1
    orb2 = string.atoi(ci_line[2]) - 1

    pairCI    = range(old*2)
    pairAlpha = range(old*2)
    for i in range(len(pairAlpha)):
pairAlpha[i] = range(norbitals)

```

```
556      for ci in range(len(pairCI)):
557  branch1 = ci%old
558  branch2 = old-1-ci%old
559  #branch2 = branch1
560  for i in range(len(AlphaOcc[ci%old])):
561      if ci < old:
562  pairAlpha[ci][i] = AlphaOcc[branch1][i]
563      else:
564  pairAlpha[ci][i] = AlphaOcc[branch2][i]
565
566  if ci < old:
567      pairCI[ci] = CI[branch1] * coef1
568      pairAlpha[ci][orb1] = 1
569  else:
570      pairCI[ci] = CI[branch2] * coef2
571      pairAlpha[ci][orb2] = 1
572
573      #for ci in range(len(pairAlpha)):
574      # for o in range(2*(p+1)):
575      #     if o % 2 == 0:
576      # print pairAlpha[ci][o+ncore],
577      # print
578      print "Geminal ",p," with coeffs ",coef1, ", ",coef2, " uses orbitals ", orb1, " and ", orb2
579      #print "Determinant CI coefficients = ",pairCI
580
581      CI = pairCI
582      AlphaOcc = pairAlpha
583  ndeterminants = pow(2,(end_ci - start_ci))
584  #end npair > 0
585
586      BetaOcc = copy.deepcopy(AlphaOcc)
587
588      if nseto > 0 and nseto != 2 and npair > 0:
589  print "\n\nWarning: the script maybe doesn't know how to handle npair=",npair, " with nseto=",nseto
590      if nseto > 2:
591  print "\n\nWarning: the script does not handle nseto=",nseto," correctly!!!"
592
593      #if nseto == 2 and spin_mult == 3:
594      if nseto > 0 and spin_mult > 1:
595  # This case is easy, since the NSETO orbitals are alpha
596  # WF = anti[12aa]
597  for det in range(len(AlphaOcc)):
598      for orb in range(len(AlphaOcc[det])):
599  if orb >= ncore and orb < ncore+nseto:
600      AlphaOcc[det][orb] = 1
601  # If you want the other triplet (even though there doesn't appear to be a difference):
602  # WF = anti[12(ab + ba)] = anti[12ab] - anti[21ab]
603  # then you need to use spinCouple
604
605      elif nseto == 2:
606  #The two electrons are spin coupled into a singlet
607  # See Eq 33a from "SCF Equations for GVB" by Bobrowicz and Goddard
608  # WF = anti[12(ab - ba)] = anti[(12 + 21)ab] = anti[12ab] + anti[21ab]
609  for j in range(core_line_number,len(gamess_output)):
610      if string.find(gamess_output[j],'OPEN SHELL ORBITALS') != -1:
611  start_ci = j+1
612  break
613
614  ci_line = string.split(gamess_output[start_ci])
615  orb1 = string.atoi(ci_line[4])-1
616  ci_line = string.split(gamess_output[start_ci+1])
617  orb2 = string.atoi(ci_line[4])-1
618  (AlphaOcc,BetaOcc,CI) = spinCouple(orb1,orb2,AlphaOcc,BetaOcc,CI,spin_mult)
619
620  ndeterminants = len(CI)
621
622      assert(ndeterminants == len(CI))
```

```
623
624 elif scf_type == "VB2000":
625     rumer = []
626     for i in range(len(gamess_output)):
627         if string.find(gamess_output[i],'GENERAL CONTROLS ($GENCTL)') != -1:
628             core_line_number = i+8
629             core_line = string.split(gamess_output[core_line_number])
630             ncore = string.atoi(core_line[3])
631     print "VB2000 settings: ncore=",ncore
632 if string.find(gamess_output[i],'RUMER PATTERN') != -1:
633     for r in range(len(cicoef)):
634 line = gamess_output[i+r+1]
635 #the first number is just the index
636 rumer = rumer + [(string.split(line))[1:]]
637
638     coreO = range(1)
639     for i in range(len(coreO)):
640         coreO[i] = range(norbitals)
641
642     for i in range(len(coreO)):
643         for j in range(norbitals):
644     if j < ncore:
645 coreO[i][j] = 1
646     else:
647 coreO[i][j] = 0
648
649     AlphaOcc = []
650     BetaOcc = []
651     CI = []
652     for r in range(len(rumer)):
653 print "rumer =", rumer[r]
654 tempA = copy.deepcopy(coreO)
655 tempB = copy.deepcopy(coreO)
656 tempC = range(1)
657 tempC[0] = cicoef[r]
658 pcum += tempC[0]*tempC[0]
659 for p in range(len(rumer[r])/2):
660     orb1 = ncore-1 + string.atoi(rumer[r][2*p])
661     orb2 = ncore-1 + string.atoi(rumer[r][2*p+1])
662     (tempA,tempB,tempC) = spinCouple(orb1,orb2,tempA,tempB,tempC,1)
663 #for ci in range(len(tempA)):
664 #    print tempC[ci], ": ",
665 #    for o in range(ncore,norbitals):
666 # print tempA[ci][o],
667 #    print
668 AlphaOcc = AlphaOcc + tempA
669 BetaOcc  = BetaOcc + tempB
670 CI = CI + tempC
671
672     ndeterminants = len(CI)
673     #sys.exit(0)
674 elif scf_type == "NONE" and ci_type == "ALDET":
675     core_line_number = -1
676     nstates = 1
677     for i in range(len(gamess_output)):
678         if string.find(gamess_output[i],'NUMBER OF CORE ORBITALS') != -1:
679             core_line_number = i
680             core_line = string.split(gamess_output[core_line_number])
681             ncore = string.atoi(core_line[5])
682 if string.find(gamess_output[i],'NUMBER OF CI STATES REQUESTED') != -1:
683             line = string.split(gamess_output[i])
684             nstates = string.atoi(line[6])
685 if string.find(gamess_output[i],'PARTIAL TWO ELECTRON INTEGRAL TRANSFORMATION') != -1:
686             break
687
688     print ""
689     for i in range(core_line_number,len(gamess_output)):
```

```
690          if string.find(gamess_output[i],'ENERGY=') != -1 and \
691             string.find(gamess_output[i],'CONVERGED') == -1:
692      print gamess_output[i],
693
694      if nstates > 1:
695  istate = string.atoi(raw_input("Choose which CI state you want [1 to %i]: "%nstates))
696
697      for i in range(core_line_number,len(gamess_output)):
698          if string.find(gamess_output[i],'ENERGY=') != -1 and \
699             string.find(gamess_output[i],'CONVERGED') == -1:
700      line = string.split(gamess_output[i])
701      if string.atoi(line[1]) == istate:
702  start_mc_data = i
703  break
704
705      end_ci = -1
706      start_ci = -1
707      for j in range(start_mc_data,len(gamess_output)):
708  m=re.search('STATE\s+\d+\s+ENERGY=\s*([\d\-\.]+)',gamess_output[j])
709  if m:
710      energy = m.group(1)
711
712  if string.find(gamess_output[j],'ALPH') != -1:
713              start_ci = j+2
714  if start_ci != -1 and len(gamess_output[j]) == 1:
715      end_ci = j-1
716      break
717
718      for i in range(start_ci,end_ci):
719  try:
720      ci_line = string.split(gamess_output[i])
721      coeffs = string.atof(ci_line[4])
722      if abs(coeffs) < detcutoff:
723  end_ci = i-1
724  break
725  except:
726      print "Error extracting ALDET state ", istate, ":"
727      print "First det line = ", start_ci
728      print "Last  det line = ", end_ci
729      print "ENERGY= line   = ", start_mc_data
730      print "Cur det index  = ", i
731      print "Cur det data   = ", gamess_output[i]
732      raise
733
734  if string.find(gamess_output[i+1],'DONE WITH DETERMINANT CI') != -1 or string.find(gamess_output[i+1],'DONE WITH GENERA
735      end_ci = i
736      break
737
738      ndeterminants = end_ci - start_ci + 1
739
740      AlphaOcc = range(ndeterminants)
741      BetaOcc = range(ndeterminants)
742      CI = range(ndeterminants)
743
744      for i in range(ndeterminants):
745          AlphaOcc[i] = range(norbitals)
746          BetaOcc[i] = range(norbitals)
747
748      for i in range(ndeterminants):
749          for j in range(ncore):
750              AlphaOcc[i][j] = 1
751              BetaOcc[i][j] = 1
752
753      for i in range(ndeterminants):
754          ci_line = string.split(gamess_output[i+start_ci])
755          CI[i] = string.atof(ci_line[4])
756          alpha_occ = ci_line[0]
```

```
         beta_occ = ci_line[2]
         for j in range(len(alpha_occ)):
             AlphaOcc[i][j+ncore] = string.atoi(alpha_occ[j])
             BetaOcc[i][j+ncore] = string.atoi(beta_occ[j])
         for k in range(len(alpha_occ)+ncore,norbitals):
             AlphaOcc[i][k] = 0
             BetaOcc[i][k] = 0

######### Use a cutoff criteria to decide which CSFs to include
for i in range(ndeterminants-1,-1,-1):
    try:
if abs(CI[i]) < detcutoff:
    #print "Removing",i,"with",CI[i]
    del CI[i]
    del AlphaOcc[i]
    del BetaOcc[i]
    except:
continue
ndeterminants = len(CI)

######## Remove any orbitals that aren't used

for i in range(norbitals-1,-1,-1):
    keep_this_orbital = 0
    for j in range(ndeterminants):
if AlphaOcc[j][i] == 1 or BetaOcc[j][i] == 1:
    keep_this_orbital = 1
    break
    if(keep_this_orbital == 1):
continue
    del wavefunction[i]
    for j in range(ndeterminants):
del AlphaOcc[j][i]
del BetaOcc[j][i]
norbitals = len(wavefunction)

##################  PRINT FLAGS: BEGIN    ########################
#
# We'll use ckmf template files (extension ckmft) to help make
# our ckmf file. The reason is to save us from having to go through
# and manually choose all our parameters. These template files are meant
# to be pretty close to what we'll end up wanting.
my_path, my_name = os.path.split(__file__)

#A couple default placed to look for "ckmft" files
templatedir = [".","..","../..","../examples","/ul/amosa/ckmf_origs",my_path]

templates = []
for dir in templatedir:
    if os.path.exists(dir):
for file in os.listdir(dir):
    if file[-5:] == "ckmft":
templates.append(dir+"/"+file)

print "\nAvailable ckmf templates:"
for i in range(len(templates)):
    print " %3i : " % i, templates[i]
    choice = i

try:
    choice = string.atoi(raw_input("Your choice [%i]:"%choice))
except:
    print "",

myStandardFlags=open(templates[choice],'r')

OUT.write('# Created on %s\n'%(time.strftime("%a, %d %b %Y %H:%M:%S", time.gmtime())))
```

```
824 OUT.write('# Using gamess output file:  %s\n'% os.path.abspath(Infile))
825 OUT.write('# Using ckmft template file: %s\n'% templates[choice])
826 OUT.write('# Orbitals are: %s'%orbital_name[orb_choice])
827 #let's save some of the important info from a GAMESS calculation
828 for i in range(len(gamess_output)):
829     line = -1;
830     if string.find(gamess_output[i],'RUN TITLE') != -1:
831 line = i+2
832     if string.find(gamess_output[i]," STATE   %i"%istate) != -1 and \
833    string.find(gamess_output[i],'ENERGY=') != -1 and \
834    string.find(gamess_output[i],'SYM=') != -1:
835 line = i
836     if string.find(gamess_output[i],'CCSD(T) ENERGY:') != -1:
837 line = i
838     if string.find(gamess_output[i],'CCSD[T] ENERGY:') != -1:
839 line = i
840     if string.find(gamess_output[i],'CCSD') != -1 and \
841    string.find(gamess_output[i],'ENERGY:') != -1 and \
842    string.find(gamess_output[i],'CORR.') != -1:
843 line = i
844     if string.find(gamess_output[i],'MBPT(2) ENERGY:') != -1:
845 line = i
846     if string.find(gamess_output[i],'CORR.') != -1 and \
847    string.find(gamess_output[i],'CR-CC') != -1:
848 line = i
849     if string.find(gamess_output[i],'FINAL') != -1:
850 line = i
851     if string.find(gamess_output[i],'$BASIS') != -1:
852 line = i
853     if string.find(gamess_output[i],'ITER:') != -1:
854 line = -1
855
856     if line > 0:
857     OUT.write('#%s' % gamess_output[line])
858     print '#%s' % gamess_output[line],
859
860
861 OUT.write("\n")
862 OUT.write(myStandardFlags.read());
863 myStandardFlags.close()
864
865 OUT.write('atoms\n %i\n'%atoms)
866 OUT.write('charge\n %i\n'%charge)
867 OUT.write('energy\n %s\n'%energy)
868
869 if string.atof(energy) >= 0.0:
870     print "\nEnergy",energy," didn't converge!!! Quitting.\n"
871     sys.exit(0)
872
873 OUT.write('norbitals\n %i\n'%norbitals)
874 OUT.write('nbasisfunc\n %i\n'%nbasisfunc)
875 OUT.write('ndeterminants\n %i\n'%ndeterminants)
876 OUT.write('&\n')
877
878 ##################  PRINT FLAGS: END      ########################
879
880 ##################  PRINT GEOMETRY: BEGIN ########################
881
882 OUT.write('&geometry\n')
883 p = re.compile("[0-9]+")
884 for line in geometry:
885     atom = line[0]
886     # if the atom title has a number in it, then we need to remove it
887     # so that QMC believes that all the atoms are the same,
888     # since Jastrows are specific to the label.
889     atom = p.sub("",atom)
890     OUT.write('%s\t%i\t%f\t%f\t%f\n'\
```

```
      %(atom,string.atof(line[1]),line[2],line[3],line[4]))
OUT.write('&\n')

##################  PRINT GEOMETRY: END  ########################

##################  PRINT BASIS: BEGIN   ########################

# calculate the number of basis functions for atom and maximum gaussians
# in any basis function
for i in range(len(basis)):
    label = basis[i][0][0]
    basis[i] = basis[i][1:]
    nbf = 0
    maxgaussian = 0
    for bf in basis[i]:
        if bf[0][0] == 'S' : nbf = nbf + 1
        elif bf[0][0] == 'P' : nbf = nbf + 3
        elif bf[0][0] == 'D' : nbf = nbf + 6
        elif bf[0][0] == 'F' : nbf = nbf + 10
elif bf[0][0] == 'G' : nbf = nbf + 15
elif bf[0][0] == 'H' : nbf = nbf + 21
elif bf[0][0] == 'I' : nbf = nbf + 28
        elif bf[0][0] == 'L' : nbf = nbf + 4
else:
    print "Error: we don't know about basis function type: ",bf[0][0]
    sys.exit(0)
        if len(bf) > maxgaussian : maxgaussian = len(bf)
    basis[i] = [[label,nbf,maxgaussian]] + basis[i]

OUT.write('&basis\n')
for atom in geometry :
    for ATOM in basis:
        if atom[0] == ATOM[0][0] :
            atomicbasis = ATOM
    head = atomicbasis[0]
    atomicbasis = atomicbasis[1:]
    OUT.write('%s\t%i\t%i\n'%(head[0],head[1],head[2]))
    for pbf in atomicbasis :
# There are a few special basis function types, and you have to
# program them individually
        if pbf[0][0] == 'L' :
    mterms = getM('S')
    for m in mterms:
OUT.write('\t%i\t%s\n'%(len(pbf),m))
for gs in pbf:
    OUT.write('\t\t%s\t%s\n'%(gs[1],normalize(m,gs[2],gs[1])))
    mterms = getM('P')
    for m in mterms:
OUT.write('\t%i\t%s\n'%(len(pbf),m))
for gs in pbf:
    OUT.write('\t\t%s\t%s\n'%(gs[1],normalize(m,gs[3],gs[1])))
else:
    mterms = getM(pbf[0][0])
    for m in mterms:
OUT.write('\t%i\t%s\n'%(len(pbf),m))
for gs in pbf:
    OUT.write('\t\t%s\t%s\n'%(gs[1],normalize(m,gs[2],gs[1])))


OUT.write('&\n')

##################  PRINT BASIS: END     ########################

##################  PRINT WAVEFUNCTION: BEGIN ###################

OUT.write('&wavefunction\n\n')
print "charge     = %i"%charge
```

```
print "norbitals  = %d\nnbasisfunc = %d\nenergy     = %s\n" % (norbitals,nbasisfunc,energy)


for i in range(norbitals):
    if len(wavefunction[i]) != nbasisfunc:
print "Error: Orbital",i,"has",len(wavefunction[i]),"basisfunctions, instead of the expected",nbasisfunc
sys.exit(0)
    for j in range(nbasisfunc):
try:
    OUT.write('%20s'%wavefunction[i][j])
except:
    print "Error:\nnorbitals = %d\nnbasisfunc = %d\ni = %d\nj = %d\n" % (norbitals,nbasisfunc,i,j)
    print "wavefunction is %d by %d\n" % (len(wavefunction),len(wavefunction[i]))
    sys.exit(1)
if (j+1)%5 == 0:
    OUT.write('\n')
    OUT.write('\n\n')

print "Alpha Occupation:"
OUT.write("Alpha Occupation\n")
for i in range(ndeterminants):
    nume = 0
    for j in range(norbitals):
nume += AlphaOcc[i][j]
        OUT.write('%i '%AlphaOcc[i][j])
if j >= ncore:
    sys.stdout.write('%i'%AlphaOcc[i][j])
    OUT.write('\n')
    print " (=",nume,")"
OUT.write('\n')
print ""
print "Beta Occupation:"
OUT.write("Beta Occupation\n")
for i in range(ndeterminants):
    nume = 0
    for j in range(norbitals):
nume += BetaOcc[i][j]
        OUT.write('%i '%BetaOcc[i][j])
if j >= ncore:
    sys.stdout.write('%i'%BetaOcc[i][j])
    OUT.write('\n')
    print " (=",nume,")"
OUT.write('\n')
print ""
constraints = []
OUT.write("CI Coeffs\n")
for i in range(ndeterminants):
    match = 0
    constraints.append(-1)
    if ci_type == "ALDET" or 1:
for j in range(i):
    ratio = string.atof(CI[i])/string.atof(CI[j])
    if abs(abs(ratio)-1.0)< 1e-5:
match = 1
# There are some couplings that are required to get the correct spin function.
# We want to include the constraints so that QMC knows which are free to optimize.
# You'll get ratio = 1 for singlet (ab-ba), and ratio = -1 for triplet (ab+ba)
#print "Using CI constraint: Det[%i] = %5.3f * Det[%i]"%(i,ratio,j)
constraints[i] = j
OUT.write('c %i %5.3f\n'%(j, ratio))
break
    if match == 0:
OUT.write('%s\n'%CI[i])
OUT.write('\n')

print str(ndeterminants) + " CI determinant(s) used, with coefficients:"
cum = 0
```

```
for i in range(ndeterminants):
    try:
ci = string.atof(CI[i])
cum += ci*ci
print "%3i) %25.7e has percentage %15.8f, cumulative remaining %15.8e" % (i+1,ci,ci*ci*100,1.0-cum),
if constraints[i] == -1:
    print ""
else:
    rel_diff = ci/string.atof(CI[constraints[i]])-1.0
    print ", constrained to %2i %25.7e"%(constraints[i]+1,rel_diff)
    except:
print "%3i) %25s has percentage %15.10e, cumulative remaining %15.10e" % (i+1,CI[i],0,1.0-cum)



OUT.write('&\n')

##################  PRINT WAVEFUNCTION: END   ##################

################## PRINT JASTROW: BEGIN ##############################

# Make a list of all the different atom types
atom_types = []
atom_type_charges = []
for atom in geometry:
    is_in_list = 0;
    for atom_type in atom_types:
        if atom[0] == atom_type:
            is_in_list = 1;
    if not is_in_list:
        atom_types = atom_types + [atom[0]]
        atom_type_charges = atom_type_charges + [atom[1]]

# write out the jastrow
OUT.write('\n&Jastrow\n\n')

if 0:
    # up down jastrow
    if nalpha > 0 and nbeta > 0:
OUT.write('ParticleTypes: Electron_Up Electron_Down\n')
OUT.write('CorrelationFunctionType: Cambridge2\n')
OUT.write('NumberOfParameterTypes: 2\n')
OUT.write('NumberOfParametersOfEachType: 1 8\n')
OUT.write('Parameters: 0.30 0.3\n')
OUT.write('NumberOfConstantTypes: 2\n')
OUT.write('NumberOfConstantsOfEachType: 1  1\n')
OUT.write('Constants: 0.5  3\n')
OUT.write('\n')

# up up jastrow
if nalpha > 1:
    OUT.write('ParticleTypes: Electron_Up Electron_Up\n')
    OUT.write('CorrelationFunctionType: Cambridge2\n')
    OUT.write('NumberOfParameterTypes: 2\n')
    OUT.write('NumberOfParametersOfEachType: 1 8\n')
    OUT.write('Parameters: 0.30 0.1\n')
    OUT.write('NumberOfConstantTypes: 2\n')
    OUT.write('NumberOfConstantsOfEachType: 1  1\n')
    OUT.write('Constants: 0.25  3\n')
    OUT.write('\n')

# down down jastrow
if nbeta > 1:
    OUT.write('ParticleTypes: Electron_Down Electron_Down\n')
    OUT.write('CorrelationFunctionType: Cambridge2\n')
    OUT.write('NumberOfParameterTypes: 2\n')
    OUT.write('NumberOfParametersOfEachType: 1 8\n')
```

```
    OUT.write('Parameters: 0.30 0.1\n')
    OUT.write('NumberOfConstantTypes: 2\n')
    OUT.write('NumberOfConstantsOfEachType: 1  1\n')
    OUT.write('Constants: 0.25  3\n')
    OUT.write('\n')

# up nuclear jastrow
if nalpha > 0:
    for i in range(len(atom_types)):
OUT.write('ParticleTypes: Electron_Up ' + atom_types[i] + '\n')
OUT.write('CorrelationFunctionType: Cambridge2\n')
OUT.write('NumberOfParameterTypes: 2\n')
OUT.write('NumberOfParametersOfEachType: 1 8\n')
OUT.write('Parameters: 0.30 -0.3\n')
OUT.write('NumberOfConstantTypes: 2\n')
OUT.write('NumberOfConstantsOfEachType: 1  1\n')
OUT.write('Constants: 0  3\n')
#        OUT.write('Constants: -' + atom_type_charges[i] + '\n')
OUT.write('\n')

# down nuclear jastrow
        if nbeta > 0:
    for i in range(len(atom_types)):
OUT.write('ParticleTypes: Electron_Down ' + atom_types[i] + '\n')
OUT.write('CorrelationFunctionType: Cambridge2\n')
OUT.write('NumberOfParameterTypes: 2\n')
OUT.write('NumberOfParametersOfEachType: 1 8\n')
OUT.write('Parameters: 0.30 -0.3\n')
OUT.write('NumberOfConstantTypes: 2\n')
OUT.write('NumberOfConstantsOfEachType: 1  1\n')
OUT.write('Constants: 0  3\n')
OUT.write('\n')
else:
    print "\nDont forget to add jastrows!"

OUT.write('&Jastrow\n')

################### PRINT JASTROW: END ################################

################### PRINT PSEUDOPOTENTIAL: BEGIN  ######################
if pp_type != "NONE":
    Inpfile = filebase + "inp"
    INP = open(Inpfile,'r')
    gamess_input = INP.readlines()
    INP.close()

    #Copy the PP right from the input file. QMcBeaver is programmed to use
    #exactly the same format, except it needs to be a GEN PP
    OUT.write('&pseudopotential\n')
    for i in range(len(gamess_input)):
if string.find(gamess_input[i].upper(),'$ECP') != -1:
    k = i+1
    while string.find(gamess_input[k].upper(),'$END') == -1:
line = string.split(gamess_input[k])
if len(line) == 4 and line[1].upper() != "GEN" and line[1].upper() != "NONE":
    print "Pseudoptential for",line[0], "is",line[1],
    print ": is unknown. It needs to be GEN or NONE.\n";
OUT.write(gamess_input[k])
k += 1
    OUT.write('&\n')
################### PRINT PSEUDOPOTENTIAL: END  ########################

print "\nFinished writing file ", Outfile
```

D.2 A Good Set of Parameters

A listing of *examples/optimize.ckmft*, representing a good set of parameters to use for beginning the optimization. The *gamess2qmcbeaver.py* script will look for a file with a *.ckmft* suffix such as this one on which to base the input file it produces.

```
&flags
# Parameters for QMC
run_type
 variational
dt
 0.01
dt_equilibration
 0.01
number_of_walkers
 100
max_time_steps
 20000
equilibration_steps
 5000
desired_convergence
 0
iseed
 0
optimize_Psi
 1
max_time
 -1
one_e_per_iter
 0
output_interval
 1000

# Parameters for wavefunction optimization
optimize_UD_Jastrows
 1
optimize_UU_Jastrows
 1
optimize_DD_Jastrows
 1
optimize_EN_Jastrows
 1
optimize_NEE_Jastrows
 0
optimize_L
 0
optimize_CI
 1
optimize_Orbitals
 0
optimize_Psi_method
 automatic
optimize_Psi_criteria
 generalized_eigenvector
a_diag
 -1e-05
ksi
 0.5
max_optimize_Psi_steps
 30
equilibrate_first_opt_step
 1
equilibrate_every_opt_step
 1
```

```
optimization_max_iterations
 1
optimization_error_tolerance
 0.001
singularity_penalty_function_parameter
 1e-06
optimize_Psi_barrier_parameter
 1
numerical_derivative_surface
 umrigar88
line_search_step_length
 Linearize
ck_genetic_algorithm_1_population_size
 1000
ck_genetic_algorithm_1_mutation_rate
 0.2
ck_genetic_algorithm_1_initial_distribution_deviation
 1

# Parameters specific to the Green's function
sampling_method
 umrigar93_importance_sampling
QF_modification_type
 umrigar93_unequalelectrons
umrigar93_equalelectrons_parameter
 0.5
warn_verbosity
 0
rel_cutoff
 100
limit_branching
 1
energy_modification_type
 umrigar93
energy_cutoff_type
 umrigar93
lock_trial_energy
 0
synchronize_dmc_ensemble
 0
synchronize_dmc_ensemble_interval
 1000

# Parameters specific to weights, branching, and fusion
walker_reweighting_method
 umrigar93_probability_weighted
branching_method
 nonunit_weight_branching
branching_threshold
 2
fusion_threshold
 0.45
population_control_parameter
 1
correct_population_size_bias
 1
old_walker_acceptance_parameter
 50

# Parameters for initialization
use_equilibration_array
 0
equilibration_function
 ramp
CKAnnealingEquilibration1_parameter
 500
walker_initialization_method
```

```
 dans_walker_initialization
walker_initialization_combinations
 3

# Parameters for added functionality/improvements
calculate_bf_density
 0
use_hf_potential
 0
hf_num_average
 100
replace_electron_nucleus_cusps
 1
print_replacement_orbitals
 0
nuclear_derivatives
 none
future_walking
 0

# Parameters relating to output
checkpoint
 0
checkpoint_interval
 100000
use_available_checkpoints
 0
checkpoint_input_name
 awt0p0_1
zero_out_checkpoint_statistics
 1
checkpoint_energy_only
 0
print_configs
 0
print_config_frequency
 50
temp_dir
 /temp1/amosa/awt0p0_1
write_all_energies_out
 0
write_electron_densities
 0
max_pair_distance
 -1
print_transient_properties
 0
print_transient_properties_interval
 10000

# Parameters for computation/MPI
parallelization_method
 manager_worker
mpireduce_interval
 100
mpipoll_interval
 5
walkers_per_pass
 1
use_basis_function_interpolation
 0
number_basis_function_interpolation_grid_points
 1000
basis_function_interpolation_first_point
 1e-10

# Parameters for the molecule and wavefunction
```

```
trial_function_type
 restricted
pseudo_gridLevel
 1
pseudo_cutoff
 0.0001
link_Jastrow_parameters
 1
link_NEE_Jastrows
 2
link_Orbital_parameters
 1
link_Determinant_parameters
 1
reproduce_NE_with_NEE_jastrow
 1
reproduce_EE_with_NEE_jastrow
 1

# Other parameters
chip_and_mike_are_cool
 Yea_Baby!
```

Appendix E

Wavefunction Optimization

Optimizing a wavefunction is a process that can take a very long time, and we have made only tentative steps towards algorithm assessment of convergence. Therefore, it is quite useful to be able to monitor progress manually, and we use this script to do so. If this script is given an output file, it will generate a plot showing the Jastrows for each optimization step.

E.1 Optimization by Example

As an example, we provide sample output from optimizing a GVB-4 cyclobutane wavefunction in Figure E.1. In this figure, we can see that after the first two steps, the Jastrows did not significantly change. This optimization was the result of starting from another set of optimized cyclobutane Jastrows, demonstrating that the Jastrows often do not need to change by very much.

E.2 Script: optimized.pl

```
1  #!/usr/bin/perl
2  #assume utilities.pl is in the same directory as summary.pl
3  my $path = `dirname $0`;
4  chomp($path);
5  require "$path/utilities.pl";
6
7  my $publication = 0;
8  my $printFunc   = 1;
9  my $useScaled   = 0;
10 my $multiPlot   = 1;
11 my $showOpt     = 1;
12 my $makeGraph   = 1;
13
14 #my $summary   = 1;
15 my $i_active = 1;
16
17 #put the jastrow in the exponential
18 my $useExp = 1;
19
20 #square the whole thing (so that the y axis
21 #can be interpreted as a percentage)
```

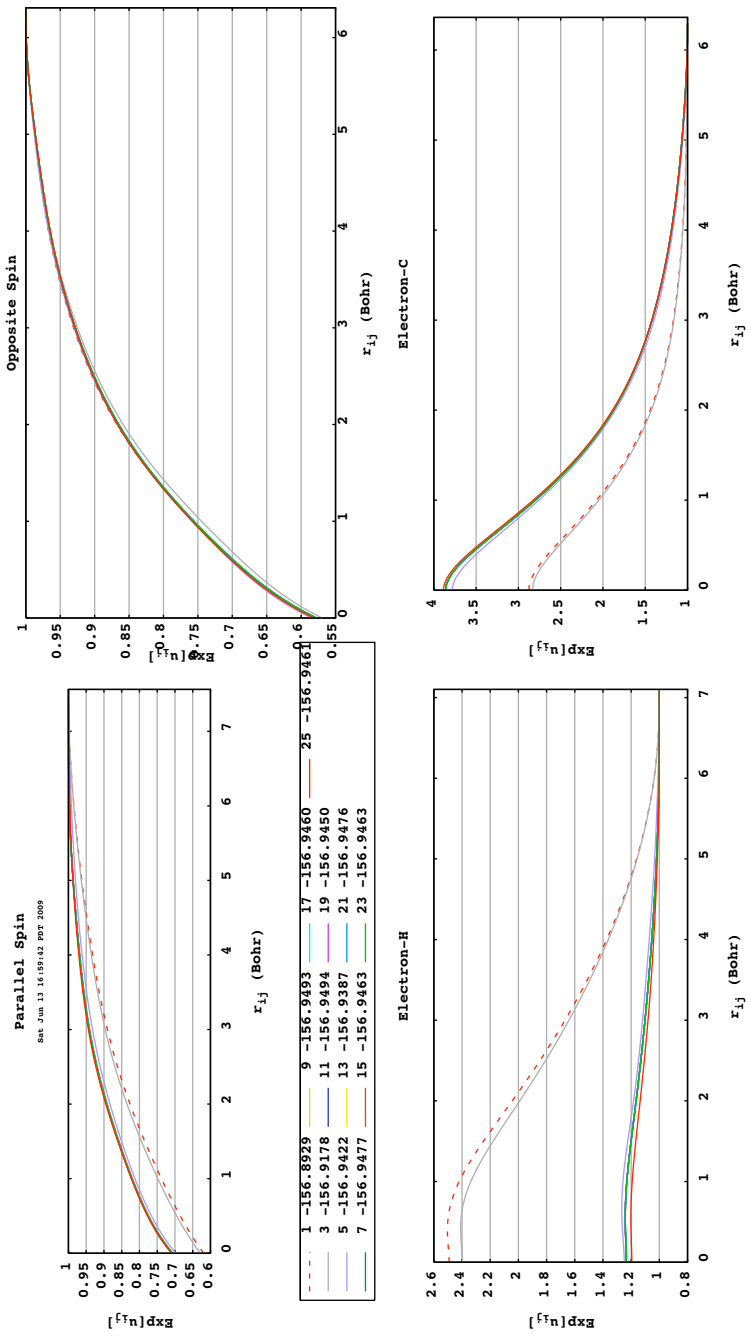

Figure E.1: Sample output from the script *optimized.pl* showing the progression of optimizing a GVB-4 cyclobutane wavefunction.

```
22  my $useSqr = 0;
23
24  my $date = `date`;
25  chomp $date;
26
27  while ( $#ARGV >= 0 && $ARGV[0] =~ /^-/ ) {
28      $type  = shift(@ARGV);
29      $param = "";
30
31      if ( $type eq "-o" ) {
32          $showOpt = !$showOpt;
33          print "Using showOpt = $showOpt\n";
34      }
35      elsif ( $type eq "-p" ) {
36          $makeGraph = ( $makeGraph + 1 ) % 2;
37          print "Using makeGraph = $makeGraph\n";
38      }
39      elsif ( $type eq "-i" ) {
40          $i_active = ( $i_active + 1 ) % 2;
41          print "Using i_active = $i_active\n";
42      }
43      elsif ( $type eq "-f" ) {
44          $param = shift(@ARGV);
45          push( @fileFilters, $param );
46          print "Adding file filter $param\n";
47      }
48      elsif ( $type eq "-x" ) {
49          $param = shift(@ARGV);
50          push( @exclusionFilters, $param );
51          print "Adding file exclusion filter $param\n";
52      }
53      elsif ( $type eq "-u" ) {
54          $param = shift(@ARGV);
55          if ( $param == 0 ) {
56              $units  = 627.50960803;
57              $unitsL = "kcal/mol";
58          }
59          elsif ( $param == 1 ) {
60              $units  = 27.211399;
61              $unitsL = "eV";
62          }
63          elsif ( $param == 2 ) {
64              $units  = 2625.5002;
65              $unitsL = "kJ/mol";
66          }
67          elsif ( $param == 3 ) {
68              $units  = 219474.63;
69              $unitsL = "cm^-1";
70          }
71          elsif ( $param == 4 ) {
72              $units  = 1;
73              $unitsL = "au";
74          }
75          print "Using $unitsL energy units, conversion = $units\n";
76      }
77      else {
78          print "Unrecognized option: $type\n";
79          exit;
80      }
81  }
82
83  my @files = sort @ARGV;
84  if ( $#files < 0 ) {
85      push( @files, "." );
86  }
87
88  getFileList( ".out", \@files );
89  $showOpt = 1 if ( $#files == 0 );
90
91  my $Cnormal = "\x1b[0m";
92  my $Chilite = "\x1b[37m";
93
94  %jastrows;
95  %plotters;
96  my %optEnergies;
97  my $base    = "";
98  my $numjw   = "";
99  my $numjwID = 0;
100 my $numbf   = 0;
101 my $numci   = 1;
102 my $refE    = 0;
103 my $step    = 1;
104 my $lastS   = "";
105 my $short   = "";
106 for ( my $index = 0 ; $index <= $#files ; $index++ ) {
107     $lastS = $short;
108     $base  = substr( $files[$index], 0, -4 );
109     $short = `basename $base`;
110     chomp $short;
111     $short =~ s/_[\d]+$//g;
112
```

```
113       #print "base = $base\n";
114       my @stuff = split /[\s:]+/, getCKMFSummary("$base.ckmf");
115       $numci  = $stuff[7];
116       $numbf  = $stuff[8];
117       $optStr = $stuff[9];
118       $refE   = $stuff[10];
119
120       open( CKMFFILE, "$base.ckmf" );
121       $numjw   = "";
122       $numjwID = 0;
123
124       while ( <CKMFFILE> !~ /Jastrow/ ) { }
125       while (<CKMFFILE>) {
126           if (/NumberOfParametersOfEachType/) {
127               if ( !( $numjw eq "" ) ) {
128                   $numjw .= ",";
129               }
130               my @line    = split /\s+/;
131               my $numthis = "$line[1]";
132               $numjwID += $line[1];
133               for ( $i = 2 ; $i <= $#line ; $i++ ) {
134                   $numthis .= "$line[$i]";
135                   $numjwID += $line[$i];
136               }
137               $numjw .= "$numthis";
138           }
139       }
140       $numjw = "$numjwID=$numjw";
141       close CKMFFILE;
142
143       open( FILE, "$files[$index]" );
144       my $name = "";
145       my $L    = 1;
146       my $best;
147       if ( $showOpt == 1 ) {
148           $best = $step;
149       }
150
151       if ( $lastS ne $short ) {
152           $step = 1;
153           if ( $showOpt == 1 ) {
154               $best = $step;
155           }
156       }
157
158       my $iterNRG = 0;
159       my $iterSTD = 0;
160       my $iterN   = 0;
161       my @dat;
162       my $line;
163       while ( !eof FILE ) {
164           $line = <FILE>;
165           @dat = split /\s+/, $line;
166           my $func = "";
167           if ( ( $line =~ /Eup/ || $line =~ /Edn/ ) && $line !~ /parameters/ ) {
168               $name = $line;
169               chomp($name);
170               if ( $line =~ /Nuclear/
171                   && ( $line =~ /EupE/ || $line =~ /EdnE/ ) )
172               {
173
174                   #it's a 3 body jastrow
175                   while ( $line !~ /x/ ) {
176                       $line = <FILE>;
177                   }
178                   @dat = split /\s+/, $line;
179                   chomp;
180                   $L = $dat[4];
181               }
182               else {
183
184                   #it's a 2 body jastrow
185                   $line = <FILE>;
186                   my $type = $line;
187                   chomp;
188                   if ( $type =~ /Fixed/ ) {
189                       $func = <FILE>;
190                       chomp($func);
191                       $func .= <FILE>;
192                       chomp($func);
193                       $L = 10.0;
194                   }
195                   else {
196                       $line = <FILE>;
197                       $line = <FILE>;
198                       chomp($func);
199                       @dat  = split /\s+/, $line;
200                       $L    = $dat[4];
201                       $func = <FILE>;
202                   }
203                   chomp($func);
```

```
204             }
205
206             $name =~ s/[:()]//g;
207             $name =~ s/Nuclear//;
208             $name =~ s/EupEup/Parallel Spin/g;
209             $name =~ s/EupEdn/Opposite Spin/g;
210             $name =~ s/Eup/Electron\-/g;
211
212             $jastrows{"$name&$best&$refE&$numci,$numbf&$numjw&$short"} =
213               "$step&$L&$func&$base";
214         }
215
216         if ( $line =~ /full step/ ) {
217             $step += 2;
218             if ( $showOpt == 1 ) {
219                 $best = $step;
220             }
221         }
222
223         if ( $line !~ /[A-Za-df-z]/ && $#dat == 9 ) {
224             $iterNRG = $dat[2];
225             $iterSTD = $dat[3];
226             $iterN   = int( $dat[4] / 1000 + 0.5 );
227
228             #print "energy line nrg=$iterNRG std=$iterSTD iterN=$iterN\n";
229         }
230
231         if ( $line =~ /Objective Value/ && $line !~ /params/ ) {
232             $optEnergies{"$short&$best"} = "$iterNRG&$iterSTD&$iterN&$optStr";
233         }
234     }
235
236     if ( !defined $optEnergies{"$short&$best"} ) {
237         $optEnergies{"$short&$best"} = "$iterNRG&$iterSTD&$iterN&$optStr";
238     }
239
240     close(FILE);
241 }
242
243 printf
244   "%15s %4s %11s %11s %7s  %10s %8s %8s %10s %8s   %-30s   %5s    %8s %-s\n",
245   "Type", "Iter", "RefE", "VMC E", getOPTHeader(), "Corr E ", "std.e.",
246   "% diff", "L (bohr)", "% diff", "Jastrow", "NumBF", "NSmpl(k)", "File Name";
247
248 my $lastL = 0;
249 my $lastE = 0;
250 my $lastN = "";
251
252 my @optAvg;
253 my @optWeight;
254 my $avgLen    = 4;
255 my $startStep = -1;
256
257 foreach $key ( sort a2n3 keys %jastrows ) {
258
259   #$jastrows{"$name&$best&$refE&$numci,$numbf&$numjw&$base"} = "$step&$L&$func";
260     ( $jName, $best, $refE, $dType, $jType, $short ) = split /&/, $key;
261     ( $step, $L, $func, $base ) = split /&/, $jastrows{$key};
262
263     ( $nrg, $std, $nsamples, $optStr ) = split /&/,
264       $optEnergies{"$short&$best"};
265     $corrE = ( $refE - $nrg ) * 627.5095;
266     $std *= 627.5095;
267
268     if ( $base ne $startStep ) {
269         @optAvg    = ();
270         @optWeight = ();
271     }
272     $startStep = $base;
273
274     my $stepVar = 0;
275     push( @optAvg,    $corrE );
276     push( @optWeight, $std );
277     shift @optAvg    if ( $#optAvg >= $avgLen );
278     shift @optWeight if ( $#optAvg >= $avgLen );
279     my $x  = 0;
280     my $x2 = 0;
281     my $ws = 0;
282     for ( my $i = 0 ; $i <= $#optAvg ; $i += 1 ) {
283         my $val = $optAvg[$i];
284         my $w   = $optWeight[$i];
285         $ws += $w;
286         $x  += $val * $w;
287         $x2 += $val * $val * $w;
288     }
289     $x  /= $ws if ( abs($ws) > 0 );
290     $x2 /= $ws if ( abs($ws) > 0 );
291     $stepVar = $x2 - $x * $x;
292     $stepVar = sqrt( abs($stepVar) );
293
294     printf "%-15s %4i %11.6f %11.6f %7s", $jName, $step, $refE, $nrg, $optStr;
```

```
295
296         my $corrEstr = "";
297         if ( abs($corrE) > 1e4 || $std == 0 ) {
298             $corrEstr = sprintf "  %10.1e", $corrE;
299         }
300         else {
301             $corrEstr = sprintf "  %-10s", getEnergyWError( $corrE, $std );
302         }
303         if ( $corrE < 0 ) {
304             printf "$Chilite$corrEstr$Cnormal";
305         }
306         else {
307             printf "$corrEstr";
308         }
309
310         printf " %8.2f", $stepVar;
311         if ( $jName ne $lastN ) {
312             $lastL = $L;
313             $lastE = $corrE;
314             $lastN = "$jName";
315             printf " %8s", "";
316             printf " %10.5f %8s", $L, " ";
317         }
318         else {
319             $diffE = $corrE - $lastE;
320             if ( abs($diffE) > 1e4 ) {
321                 printf " %8.1e", $diffE;
322             }
323             else {
324                 printf " %8.2f", $diffE;
325             }
326             printf " %10.5f %8.2f", $L, 100.0 * ( $L - $lastL ) / $L;
327         }
328         $lastL = $L;
329
330         #$lastE = $corrE;
331
332         printf "   %-30s %7s    %8g %-s\n", $jType, $dType, $nsamples, $base;
333
334         if ( !( $func eq "" ) ) {
335             $plotters{$jName} .= "$jName&$dType&$L&$jType&$func&$nrg&$short&$step#";
336         }
337     }
338
339     exit if ( $makeGraph == 0 );
340
341     my $gnuplot = "/ul/amosa/bin/gnuplot";
342     $base =~ s/_[\d]+$//g if ( !$showOpt );
343     my $modbase = $base;
344     $modbase =~ s/_/\\\\_/g;
345     my $printedHeader = 0;
346     my @goodlt;
347     push( @goodlt, 3 );    # if($publication == 0);
348     push( @goodlt, 1 );
349     push( @goodlt, 5 );
350     push( @goodlt, 4 );
351     push( @goodlt, 6 );
352     push( @goodlt, 7 );
353
354     my $allPlots = "set key outside below box Left reverse;\\\n";
355     my $lastPlot = "set key outside below box Left reverse;\\\n";
356
357     foreach $key ( reverse sort keys %plotters ) {
358         my $filename;
359         if ($multiPlot) {
360             $file_name = "jastrows";
361         }
362         else {
363             $file_name     = "$key";
364             $printedHeader = 0;
365         }
366         if ($showOpt) {
367
368             #$file_name .= "_${base}_plot.pdf";
369             $file_name .= "_plot.pdf";
370         }
371         else {
372             $file_name .= "_plot.pdf";
373         }
374
375         if ($showOpt) {
376             $caption .= ", $modbase";
377         }
378         else {
379             $caption .= ", key L; CI,BF; JW; ID";
380         }
381         my $xlabel = "r_{ij} (Bohr)";
382         my $ylabel = "u_{ij}";
383
384         if ($useExp) {
385             $ylabel = "Exp[$ylabel]";
```

```
386           if ($useSqr) {
387               $ylabel = "|$ylabel|^2";
388           }
389       }
390
391       print "Adding graph of $key to: $file_name\n";
392
393       if ( !$printedHeader ) {
394           if ($i_active) {
395               $gnuplot .=
396                 " -geometry 1280x740";    #this is optimized for Amos' laptop...
397               open( GNUPLOT, "|$gnuplot" );
398               print GNUPLOT
399   "set terminal x11 persist raise enhanced font \"Courier-Bold,12\" title \"$file_name\" dashed linewidth 2\n";
400           }
401           else {
402               `/bin/rm -f $file_name`;
403
404               #open(GNUPLOT, ">gnuplot.gnu");
405               open( GNUPLOT, "|$gnuplot" );
406               if ( $publication == 1 ) {
407                   print GNUPLOT
408   "set term pdf color enhanced font \"Courier-Bold,16\" linewidth 10 dashed dl 3 size 17.5,10\n";
409               }
410               else {
411                   print GNUPLOT
412   "set term pdf color enhanced font \"Courier-Bold,14\" linewidth 5 dashed dl 3 size 17.5,10\n";
413               }
414               print GNUPLOT "set output \"$file_name\"\n";
415           }
416           print GNUPLOT <<gnuplot_Commands_Done;
417   #fonts with extensions "ttf" and "dfont" will work
418   #here is a list of available fonts: Chalkboard Helvetica Times
419   #Courier Monaco LucidaGrande
420   #set term gif crop enhanced font 'Monaco' 8
421
422   #fonts on hive:
423   #set term gif crop enhanced font 'VeraMono' 8
424   #set term svg dynamic enhanced font "VeraMono,8"
425
426   #fonts built into PDFLib Lite:
427   #Courier, Courier-Bold, Courier-Oblique, Courier-BoldOblique,
428   #Helvetica, Helvetica-Bold, Helvetica-Oblique, Helvetica-BoldOblique,
429   #Times-Roman, Times-Bold, Times-Italic, Times-BoldItalic, Symbol, ZapfDingbats
430   set size 0.9,1
431   unset colorbox
432   show style line
433   #set logscale y 2
434   set grid ytics
435   set mytics
436   set tics scale 1.5, 0.75
437   set nokey
438   #set key noenhanced
439   set xlabel "$xlabel"
440   set ylabel "$ylabel"
441   #set yrange[$y_min:$y_max]
442   gnuplot_Commands_Done
443
444           if ($multiPlot) {
445               $numPlots = scalar keys %plotters;
446               $numR     = 2;
447               $numC     = 2;
448               $numC     = 3 if ( $numPlots > 4 );
449               $numR     = 3 if ( $numPlots > 6 );
450               die "Too many plots: $numPlots" if ( $numPlots > 9 );
451               $allPlots .= "set multiplot layout $numR,$numC;\\\n";
452               $lastPlot .= "set multiplot layout $numR,$numC;\\\n";
453           }
454       }
455
456       my $caption = "$key";
457       $caption =~ s/Eup/E_{up} /g;
458       $caption =~ s/Edn/E_{dn} /g;
459       $caption =~ s/Nuclear([\w]+)/$1/g;
460       $caption = "$caption Jastrow Functions";
461       if ( $printedHeader || $publication == 1 ) {
462           $allPlots .= "set title \"$caption\";\\\n";
463           $lastPlot .= "set title \"$caption\";\\\n";
464       }
465       else {
466           $allPlots .= "set title \"$caption\\n{/=8${date}}\";\\\n";
467           $lastPlot .= "set title \"$caption\\n{/=8${date}}\";\\\n";
468       }
469       $printedHeader = 1;
470
471   #$jastrows{"$name&$best&$refE&$numci,$numbf&$numjw&$base"} = "$step&$L&$func&$energy";
472       my @plots = split /\#/, $plotters{$key};
473
474       my $xmax      = 0;
475       my $longestJW = 0;
476
```

```
477       for ( my $i = 0 ; $i <= $#plots ; $i++ ) {
478           if ($useScaled) {
479               $xmax = 1;
480           }
481           else {
482               my $new = ( split /&/, $plots[$i] )[2];
483               if ( $new > $xmax ) {
484                   $xmax = $new;
485               }
486           }
487       }
488
489       $allPlots .= "plot [0:$xmax]";
490       $lastPlot .= "plot [0:$xmax]";
491
492       #    for(my $i=0; $i<=$#plots; $i++){
493       for ( my $i = $#plots ; $i >= 0 ; $i -= 1 ) {
494           ( $jName, $dType, $max, $jw, $func, $optE, $example, $step ) =
495             split /&/, $plots[$i];
496           $jw =~ s/18,//g;
497           $jw =~ s/18//g;
498
499           my $title;
500
501           if ($showOpt) {
502               $title = sprintf "%2i %8.4f", $step, $optE;
503
504               #$title = $example;
505           }
506           else {
507               if ( $max >= 10.0 ) {
508                   $title = sprintf "%-4.1f;", $max;
509               }
510               else {
511                   $title = sprintf "%-4.2f;", $max;
512               }
513
514               #$title = "";
515               $title .= sprintf " %3s; %s; %s", $dType, $jw, $example;
516           }
517           $title =~ s/_/\\\\_/g;
518
519           #change the font size of the key
520           #$title = "{/=10$title}";
521
522           $func =~ s/\^/**/g;
523           $func =~ s/ +//g;
524
525           #a polynomial might not be completed. it might end with a +)
526           $func =~ s/\+\)/\)/g;
527
528           #add in the implicit multiplications
529           #this line confuses emacs' indentation algorithm...
530           $func =~ s/([\d])([x\(])/$1*$2/g;
531
532           if ($useScaled) {
533               $max = 1;
534           }
535           else {
536               $func =~ s/x/(x\/$max)/g;
537           }
538
539           if ($useExp) {
540               $func = "exp($func)";
541               if ($useSqr) {
542                   $func = "(${func})**2";
543               }
544           }
545
546           my $lt;
547           if ( $publication == 1 ) {
548               $lt = $goodlt[ $i % 12 ];
549           }
550           else {
551               $lt = $goodlt[ int( $i / 12 ) ];
552           }
553           my $lc = ( $i + 1 ) % 12;
554
555           #print "line number $i has type lc $lc lt $lt\n";
556           $func = "x > $max ? 1/0 : $func";
557
558           #$func = "x";
559           $func = " $func lc $lc lt $lt title \"$title\"";
560           $allPlots .= $func;
561
562           #print GNUPLOT " [0:$kd[2]] $func title \"$kd[3]\"";
563           if ( $i == 0 ) {
564               $allPlots .= ";\\";
565               $lastPlot .= " $func;\\\n";
566           }
567           elsif ( $i == 1 ) {
```

```
568             $allPlots .= ",\\";
569             $lastPlot .= " $func,\\\n";
570         }
571         else {
572             $allPlots .= ",\\";
573         }
574         $allPlots .= "\n";
575     }
576 
577     if ($multiPlot) {
578 
579         #In order to only include the key once, we must make sure that the
580         #plots are always sorted the same!!!
581         $allPlots .= "set nokey;\\\n";
582         $lastPlot .= "set nokey;\\\n";
583     }
584 
585     #`/bin/rm $_.dat`;
586 
587     #`open $file_name`;
588 }
589 $allPlots .= "unset multiplot";
590 $lastPlot .= "unset multiplot";
591 
592 #print $lastPlot;
593 #print "bind e 'v=v+1; if(v%2) $lastPlot; else $allPlots;'\n";
594 #die;
595 print GNUPLOT "v=0\n";
596 print GNUPLOT "bind l 'v=v+1; if(v%2) $lastPlot; else $allPlots'\n";
597 
598 print GNUPLOT "$allPlots\n";
599 
600 if ($multiPlot) {
601 
602     #print GNUPLOT "unset multiplot\n";
603 }
604 print GNUPLOT "pause mouse button2\n";
605 close(GNUPLOT);
606 
607 if ( $i_active == 0 ) {
608     my $email = "nitroamos\@gmail.com";
609     print "Check $email...\n";
610 `bash -c \"echo Current directory \" | /usr/bin/mutt -s \"[jastrows] $file_name\" -a $file_name $email`;
611     `rm $file_name`;
612 }
```

Appendix F

Convergence Scripts

The downside of a Monte Carlo simulation is that many iterations are required to lower the statistical error, which goes down only as slowly as $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$. On the other hand, a Monte Carlo simulation will fairly quickly give a reasonable estimate of the final converged value, a fact that it is useful to take advantage of. Therefore, we consider it very important that we are able to quickly examine the progress of a calculation because if the calculation has gone bad, then we will want to stop it and fix the problem before wasting more computer power.

To do this, we provide several tools. First, we have a script *summary.pl* which will intelligently scan through directories looking for DMC results, and can figure out which calculations are comparable. For example, if it is pointed at directories containing ethylene and cyclobutane calculations, it will figure out the stoichiometric ratio it needs to provide you with the best estimate possible, and associated error, for the difference in energy. This script has numerous time saving features. Second, using a file produced by *summary.pl*, we have developed a second script called *plotter.pl* which can produce time series data using gnuplot. Third, we provide numerous routines in *utilities.pl* that provide helpful services, such as estimating how long a calculation will take to finish based on parameters in the input file and the amount of time taken so far.

F.1 Summarizing by Example

Most of the features that *summary.pl* provides are detailed by running the script with the help option, -h, or by reading lines 50-77. The script works by using the *getFileList* routine form *utilities.pl*. This routine will be passed all the files or directories provided on the

command line to *summary.pl*, defaulting to the directory '.' if nothing else was provided, and will recursively scan all provided directories to find all the files whose names end with *.qmc*. Although you can select files using the command line, the script *summary.pl* also provides ways to exclude or include files based on their names. For example, if we descend into a directory containing all of our ethylene calculations, we can run the command as follows.

```
../bin/summary.pl -x Ntw -x Ta -x V -c -u eV
<snip>
  3  15) 0.0075    ct0p0_Tv_exp_1  -    ct0p1_N_exp_1  =  4.4723(69) eV VMC =   4.54592
 10  22) 0.0075    ct0p5_Tv_exp_1  -    ct0p6_N_exp_1  =   4.483(12) eV VMC =   4.40799 GVB =   4.28825
  8  18) 0.0075   awt0p1_Tv_exp_1  -   awt0p2_N_exp_1  =  4.4862(76) eV VMC =   4.30773 GVB =   4.25262
 12  25) 0.0075   awt0a5_Tv_exp_1  -   awt0a6_N_exp_1  =   4.488(11) eV VMC =   4.54305 CI =   4.70210
  9  23) 0.0075   awt0p5_Tv_exp_1  -   awt0p6_N_exp_1  =  4.4910(61) eV VMC =   4.43636 CI =   4.26616
  7  21) 0.0075   awt0a1_Tv_exp_1  -   awt0a2_N_exp_1  =   4.496(18) eV VMC =   4.53507 CI =   4.56880
  5  16) 0.0075   awt0p0_Tv_exp_1  -   awt0p1_N_exp_1  =   4.497(18) eV VMC =   4.49094 GVB =   4.25834
  6  19) 0.0075    ct0p1_Tv_exp_1  -    ct0p2_N_exp_1  =  4.5088(70) eV VMC =   4.52583 GVB =   4.27410
 11  24) 0.0075   awt0r5_Tv_exp_1  -   awt0r6_N_exp_1  =   4.560(17) eV VMC =   4.50111 CI =   4.67214
```

Here we have selected to exclude any files with Ntw in the title (the twisted geometry), to exclude Ta (adiabatic triplet), and V (vertical singlet). We have also selected to include whatever underlying SCF comparisons are available (with -c), and we have chosen to display the results in energy units of eV. In the output shown here, each row starts with two indices which were defined in the output that we have not displayed. We can then see the time step that calculation represented, the file names represented, the energy differences, and the other comparisons. The VMC results are typically stored as comments at the beginning of the input file based on optimization iterations.

F.2 Script: summary.pl

```
1  #!/usr/bin/perl
2
3  # Quick start guide is found by running: summary.pl -h
4  #
5  #
6  #
7  #use strict;
8  #assume utilities.pl is in the same directory as summary.pl
9  my $path = `dirname $0`;
10 chomp($path);
11 require "$path/utilities.pl";
12
13 # First, select the default values for all our parameters.
14 my $useVar       = 0;
15 my $dtFilter     = 0;
16 my $orbFilter    = 1;
17 my $compareE     = 0;
18 my $sumResults   = 1;
```

```
my $latexHelp    = 0;
my $latexDTcol   = 0;
my $averageTitle = 0;
my $estd_stop    = 0.0;    #in $units

#my $extraTag  = "trail_eps2";

my @fileFilters;
my @exclusionFilters;

my $units  = 627.50960803;
my $unitsL = "kcal/mol";

#keep only 1 line every $drop lines
#also look at $every in plotter.pl
my $drop = 1;
if ( $drop != 1 ) {
    print "Keeping only 1 line in $drop\n";
}

#Second, read in user input.
my @files;
while ( $#ARGV >= 0 ) {
    $type  = shift(@ARGV);
    $param = "";

    if ( $type !~ /^-/ ) {

        #assume for now that it an output file
        push( @files, $type );
    }
    elsif ( $type eq "-h" ) {
        print "Usage:\n";
        print "-h Print this help.\n";
        print "-v Include VMC calculations (currently = $useVar).\n";
        print "-a Average equivalent files (currently = $averageTitle).\n";
        print
"-t <param> Only include dt=<param> (or all if 0, currently = $dtFilter).\n";
        print "-f <param> Only include files that match <param>.\n";
        print "-x <param> Exclude files that match <param>.\n";
        print
"-u <param> Convert energy units to <param> units. E.g. <param> = ev or kcal\n";
        print
"-o Include comparisons between inconsistent orbitals (currently = $orbFilter).\n";
        print
"-c Include non-DMC energy comparisons, if available (currently = $compareE).\n";
        print
"-e <param> Stop reading calculations when the error goes below <param>, in the selected units (currently = $estd_stop)
        print
          "-s Summarize output if sumResults=1 (currently = $sumResults).\n";
        print "-l Make a LaTeX table (currently = $latexHelp).\n";
        print
"Any option not starting with a '-' will be interpreted as a calculation file/directory.\n";
        print
"Directories are recursively scanned, ignoring any directories named \"hide\"\n";
        print
"If you don't include any calculation files, then we'll add directory \".\"\n";
        print "So far, you've selected \"@files\".\n";
        exit;
    }
    elsif ( $type eq "-v" ) {
        $useVar = ( $useVar + 1 ) % 2;
        print "Using useVar = $useVar\n";
    }
    elsif ( $type eq "-a" ) {
        $averageTitle = ( $averageTitle + 1 ) % 2;
        print "Using averageTitle = $averageTitle\n";
```

```
86      }
87      elsif ( $type eq "-t" ) {
88          $param = shift(@ARGV);
89          $dtFilter = $param if ( $param >= 0 );
90          print "Using dt filter $dtFilter\n";
91      }
92      elsif ( $type eq "-f" ) {
93          $param = shift(@ARGV);
94          push( @fileFilters, $param );
95          print "Adding file filter $param\n";
96      }
97      elsif ( $type eq "-x" ) {
98          $param = shift(@ARGV);
99          push( @exclusionFilters, $param );
100         print "Adding file exclusion filter $param\n";
101     }
102     elsif ( $type eq "-u" ) {
103         $param = shift(@ARGV);
104         $param = lc($param);
105         if ( $param =~ /kcal/ ) {
106             $units  = 627.50960803;
107             $unitsL = "kcal/mol";
108         }
109         elsif ( $param =~ /ev/ ) {
110             $units  = 27.211399;
111             $unitsL = "eV";
112         }
113         elsif ( $param =~ /kj/ ) {
114             $units  = 2625.5002;
115             $unitsL = "kJ/mol";
116         }
117         elsif ( $param =~ /cm/ ) {
118             $units  = 219474.63;
119             $unitsL = "cm^-1";
120         }
121         elsif ( $param =~ /au/ || $param =~ /hart/ ) {
122             $units  = 1;
123             $unitsL = "au";
124         }
125         print "Converting energy units: 1.0 $unitsL = $units au\n";
126     }
127     elsif ( $type eq "-o" ) {
128         $orbFilter = ( $orbFilter + 1 ) % 2;
129         if ( $orbFilter == 1 ) {
130             print "Filtering to only include balanced orbitals\n";
131         }
132         else {
133             print "Not filtering results based on orbital usage.\n";
134         }
135     }
136     elsif ( $type eq "-c" ) {
137         $compareE = ( $compareE + 1 ) % 2;
138         print "Comparing with reference energies, compareE = $compareE.\n";
139     }
140     elsif ( $type eq "-e" ) {
141         $param     = shift(@ARGV);
142         $estd_stop = 1 * $param;
143
144         #Print the message later, once we're sure $unitsL has been set
145     }
146     elsif ( $type eq "-s" ) {
147         $sumResults = ( $sumResults + 1 ) % 2;
148         print "Summarize report, sumResults = $sumResults.\n";
149     }
150     elsif ( $type eq "-l" ) {
151         $latexHelp = ( $latexHelp + 1 ) % 2;
152         print "LaTex Helper, latexHelp = $latexHelp.\n";
```

```
153     }
154     else {
155         print "Unrecognized option: $type\n";
156         die;
157     }
158 }
159 
160 if ( $estd_stop > 0.0 ) {
161     print
162 "Notice: we will stop reading calculations once they reach an error of $estd_stop $unitsL!\n\n";
163 }
164 
165 push( @files, "." ) if ( $#files < 0 );
166 
167 #getFileList(".out",\@files);
168 getFileList( ".qmc", \@files );
169 open( DATFILE, ">plotfile.dat" );
170 
171 my $Cnormal = "\x1b[0m";
172 my $Chilite = "\x1b[37m";
173 
174 my $lenLong = 0;
175 my $num_results;
176 my $ave_result;
177 my $headerLine = "";
178 
179 my %dt_ave_results;
180 my %label;
181 my %dt_err_results;
182 my %dt_num;
183 my %dt_num_results;
184 my %dt_nme_results;
185 my %summary;
186 my %shortnames;
187 my %referenceE = ();
188 
189 my $lastlines = "";
190 for ( my $index = 0 ; $index <= $#files ; $index++ ) {
191     my $cur = $files[$index];
192     next if ( !( -f $cur ) );
193 
194     my $isIncluded  = 1;
195     my $filterMatch = 0;
196     foreach $filter (@fileFilters) {
197 
198         #we only include a file if it matches one of the filters
199         $filterMatch = 1 if ( $cur =~ /$filter/ );
200     }
201     $isIncluded = 0 if ( $filterMatch == 0 && $#fileFilters >= 0 );
202 
203     foreach $filter (@exclusionFilters) {
204 
205         #exclude a file if it matches one of the exclusion filters
206         #print "filter = $filter cur = $cur\n";
207         $isIncluded = 0 if ( $cur =~ /$filter/ );
208     }
209     next if ( $isIncluded == 0 );
210 
211     my $base = "";
212     if ( $cur =~ /.out$/ ) {
213         $base = substr( $cur, 0, -4 );
214     }
215     elsif ( $cur =~ /.qmc$/ ) {
216         $base = substr( $cur, 0, -4 );
217     }
218     else {
219         next;
```

```
220     }
221     my $short = `basename $base`;
222     chomp($short);
223
224     #remove the restart index
225     $short = $1 if ( $short =~ /([\w\d]+)\.\d\d$/ );
226
227     if ( $averageTitle == 1 ) {
228
229         #remove any _\d at the end
230         $short = $1 if ( $short =~ /([\w\d]+)_[\d]+$/ );
231     }
232
233     my $vare = "";
234
235     my $dt          = 0;
236     my $oepi        = 0;
237     my $nw          = 0;
238     my $effnw       = 0;
239     my $opt         = -1;
240     my $isd         = -1;
241     my $hfe         = 0;
242     my $numbf       = 0;
243     my $numci       = 0;
244     my $numor       = 0;
245     my $use3        = 0;
246     my $extraVal    = 0;
247     my %refEnergies = ();
248
249     open( CKMFFILE, "$base.ckmf" );
250     while (<CKMFFILE>) {
251         if ( $_ =~ /^\#/ && $_ !~ /[A-DF-Za-df-z]+/ && $vare eq "" ) {
252             chomp;
253             my @line = split /[ ]+/;
254
255             #This is from the header; the top energy is the best
256             $vare = $line[2];
257             $refEnergies{"VMC"} = $vare;
258         }
259
260         $refEnergies{"RHF"} = ( split /\s+/ )[5] if (/FINAL RHF ENERGY/);
261         $refEnergies{"RHF"} = ( split /\s+/ )[5] if (/FINAL ROHF ENERGY/);
262         $refEnergies{"GVB"} = ( split /\s+/ )[5] if (/FINAL GVB ENERGY/);
263         $refEnergies{"CI"}  = ( split /\s+/ )[4]
264           if ( /^\#/ && /STATE/ && /ENERGY/ );
265         $refEnergies{"$1"} = ( split /\s+/ )[3]
266           if ( /^\#/ && /\s+([\w\d\(\)]+)\s+ENERGY:/ );
267
268         if ( $_ =~ m/^\s*run_type\s*$/ ) {
269             $_ = <CKMFFILE>;
270             chomp;
271             my @line = split /[ ]+/;
272             $isd = $line[1];
273             if ( $useVar == 0 ) {
274                 last if ( $isd eq "variational" );
275             }
276         }
277         if ( $_ =~ m/^\s*dt\s*$/ ) {
278             $_ = <CKMFFILE>;
279             chomp;
280             my @line = split /[ ]+/;
281             $dt = $line[1];
282         }
283         if ( $_ =~ m/^\s*one_e_per_iter\s*$/ ) {
284             $_ = <CKMFFILE>;
285             chomp;
286             my @line = split /[ ]+/;
```

```
287             $oepi = $line[1];
288         }
289         if ( $_ =~ m/^\s*number_of_walkers\s*$/ ) {
290             $_ = <CKMFFILE>;
291             chomp;
292             my @line = split /[ ]+/;
293             $nw = $line[1];
294         }
295         if ( $_ =~ m/^\s*optimize_Psi\s*$/ ) {
296             $_ = <CKMFFILE>;
297             chomp;
298             my @line = split /[ ]+/;
299             $opt = $line[1];
300         }
301         if ( $_ =~ m/^\s*energy\s*$/ ) {
302             $_ = <CKMFFILE>;
303             chomp;
304             my @line = split /[ ]+/;
305             $hfe = $line[1];
306         }
307         if ( $_ =~ m/^\s*nbasisfunc\s*$/ ) {
308             $_ = <CKMFFILE>;
309             chomp;
310             my @line = split /[ ]+/;
311             $numbf = $line[1];
312         }
313         if ( $_ =~ m/^\s*norbitals\s*$/ ) {
314             $_ = <CKMFFILE>;
315             chomp;
316             my @line = split /[ ]+/;
317             $numor = $line[1];
318         }
319         if ( $_ =~ m/^\s*ndeterminants\s*$/ ) {
320             $_ = <CKMFFILE>;
321             chomp;
322             my @line = split /[ ]+/;
323             $numci = $line[1];
324         }
325         if ( $_ =~ m/^\s*use_three_body_jastrow\s*$/ ) {
326             $_ = <CKMFFILE>;
327             chomp;
328             my @line = split /[ ]+/;
329             $use3 = $line[1];
330         }
331         if ( $extraTag ne "" ) {
332             if ( $_ =~ m/^\s*$extraTag\s*$/ ) {
333                 $_ = <CKMFFILE>;
334                 chomp;
335                 my @line = split /[ ]+/;
336                 $extraVal = $line[1];
337             }
338         }
339         if ( $_ =~ m/&geometry$/ ) {
340             last;
341         }
342     }
343 
344     #next if($opt == 1);
345     if ( $useVar == 0 ) {
346         next if ( $isd eq "variational" );
347     }
348     if ( $nw < 100 ) {
349         print "Not including $base because it has $nw walkers.\n";
350         next;
351     }
352     next if ( $dtFilter != 0 && $dt != $dtFilter );
353 
```

```
354     while ( <CKMFFILE> !~ /Jastrow/ ) { }
355     my $numjw   = "";
356     my $numjwID = 0;
357     while (<CKMFFILE>) {
358         if (/NumberOfParametersOfEachType/) {
359             if ( !( $numjw eq "" ) ) {
360                 $numjw .= ",";
361             }
362             my @line    = split /\s+/;
363             my $numthis = "$line[1]";
364             $numjwID += $line[1];
365             for ( $i = 2 ; $i <= $#line ; $i++ ) {
366                 $numthis .= "$line[$i]";
367                 $numjwID += $line[$i];
368             }
369             $numjw .= "$numthis";
370         }
371     }
372     $numjw = "$numjwID=$numjw";
373     close CKMFFILE;
374
375     open( RUNFILE, "$base.run" );
376     my $machine = "";
377     while (<RUNFILE>) {
378         if (/lamboot/) {
379             $machine = "m";
380         }
381         elsif (/machinefile/) {
382             $machine = "h";
383         }
384     }
385     close(RUNFILE);
386
387     open( QMCFILE, "$cur" );
388     my $line;
389     my @data;
390     my $more = 1;
391     my $eavg;
392     my $estd;
393     my $iteration;
394     my $num_samples     = 0.00001;
395     my $fordatfile      = "";
396     my $counter         = 0;
397     my $wallclock       = "";
398     my $totalclock      = "";
399     my $sampleclock     = "";
400     my $effdt           = 0;
401     my $sampleVar       = 0;
402     my $sampleVarCorLen = 0;
403     my $corLength       = 0;
404
405     while (<QMCFILE>) {
406         $headerLine = $_ if ( /iteration/ && /Eavg/ && /Samples/ );
407
408         next if ( $estd > 0 && $estd * $units < $estd_stop );
409
410         #this is to avoid processing lines with warnings
411         next if ( $_ =~ /[=:]/ && $_ !~ /Results/ );
412
413         chomp;
414         @data = split /[ ]+/;
415
416         #this is the number of data elements per line
417         #it can have the letter 'e' or 'E' since scientific notation uses them
418         if ( $#data >= 8 && $_ !~ /[A-DF-Za-df-z]+/ && $more ) {
419             $counter++;
420             $iteration = $data[1];
```

```
421             $iteration /= 8 if ( $oepi == 1 );
422             $eavg = $data[2];
423             $estd = $data[3];
424
425     #In the old format, this was trial energy
426     #In the new format, this is the acceptance probability, which uses parenthesis
427             if ( $data[6] =~ /\(/ ) {
428
429                 #new output format
430                 $num_samples = $data[4];
431                 $corLength   = $data[5];
432                 $effnw       = $data[7];
433                 if ( $isd eq "variational" ) {
434                     $effdt = $dt;
435                 }
436                 else {
437                     $effdt = $data[10];
438                 }
439             }
440             else {
441                 $effnw = $data[4];
442
443                 #old output format
444                 if ( $isd eq "variational" ) {
445                     $effdt       = $data[5];
446                     $num_samples = $data[6];
447                 }
448                 else {
449                     $effdt       = $data[7];
450                     $num_samples = $data[8];
451
452                 }
453             }
454
455             #this is equal to sample variance * correlation length
456             $sampleVarCorLen = $estd * $estd * $num_samples;
457
458             next if ( $num_samples <= 0 );
459
460             #make sure we have the first and last data points included
461             next
462               if ( $counter % $drop != 0
463                 && $counter != 1
464                 && $iteration % 100 == 0 );
465             next if ( $iteration < 0 );
466             $fordatfile .= sprintf "%20i %20.10f %20.10f %20i\n", $num_samples,
467               $eavg, $estd, $iteration;
468             if ( $extraTag ne "" ) {
469                 $line = sprintf "%30s $_ %15s\n", "$base", "$extraVal";
470             }
471             else {
472                 $line = sprintf "%30s $_\n", "$base";
473             }
474
475         }
476         elsif (/Results/) {
477             $more = 0;
478         }
479     }
480     close QMCFILE;
481
482     my @times = `grep Time $base.out`;
483     $wallclock   = ( split /\s+/, $times[0] )[11];
484     $totalclock  = ( split /\s+/, $times[1] )[11];
485     $sampleclock = ( split /\s+/, `grep "per sample per" $base.out` )[8];
486     chomp($sampleclock);
487     my $numwarnings = `grep WARNING $base.out | wc -l`;
```

```
488
489     #This is "number of warnings per 1000 samples"
490     #$numwarnings /= $num_samples;
491     $numwarnings = sprintf "%4i", $numwarnings;
492     $numwarnings = " $Chilite$numwarnings$Cnormal" if ( $numwarnings > 0.5 );
493     my $numerrors = `grep ERROR $base.out | wc -l`;
494
495     #$numerrors /= $num_samples;
496     $numerrors = sprintf "%4i", $numerrors;
497
498     $lastlines .= "$line";
499     my $key = "$dt&$numbf&$numjw&$nw&$numci&$numor&$oepi&$short";
500     if ( $vare eq "" ) {
501
502         #use the value for energy in the key
503         $key = "$hfe&$key";
504     }
505     else {
506
507         #use the variational energy from the header in the key
508         $key = "$vare&$key";
509     }
510
511     my $runage = getFileAge( "$base.out", 1 );
512
513     # updated in the last 15 minutes
514     $short = "*$short" if ( $runage < 900 && $estd * $units > $estd_stop );
515     if ( exists $shortnames{$key} ) {
516         my $orig = $shortnames{$key};
517         $shortnames{$key} = $short if ( length $orig > length $short );
518     }
519     else {
520
521         $shortnames{$key} = $short;
522     }
523     $lenLong = length $short if ( length $short > $lenLong );
524
525     foreach $etype ( keys %refEnergies ) {
526         $referenceE{$key}{$etype} = $refEnergies{$etype};
527     }
528
529     if ( $eavg < 0 ) {
530         my $weight = $num_samples / 100000;
531
532         $dt_ave_results{$key} += $eavg * $weight;
533         $dt_num_results{$key} += $weight;
534         $dt_num{$key}         += 1.0;
535
536         if ( $estd > 0 ) {
537             $dt_err_results{$key} += $estd * $estd * $weight;
538             $dt_nme_results{$key} += $weight;
539         }
540
541         $ave_result += $eavg;
542         $num_results++;
543     }
544     my $in_kcal = $eavg * $units;
545
546 #printf "%50s %15s %15s E_h=%20.14f E_kcal=%20.10f Err=%i Warn=%i\n","$base","dt=$dt","nw=$nw",$eavg,$in_kcal,$numerror
547
548     $summary{$key} .=
549       sprintf "..  %-30s%1s%7s %5s %16s %4s %5s",
550       "$base", "$machine", "$dt", "$effnw", getEnergyWError( $eavg, $estd ),
551       $numerrors, $numwarnings;
552
553     if ( $wallclock ne "" ) {
554
```

```
555         #the calculation completed, and some extra data is available
556         if ( abs($corLength) < 1e-10 ) {
557
558             #The old format of output printed the Sample variance directly
559             $sampleVar = ( split /\s+/, `grep "Sample variance" $base.out` )[3];
560             $corLength = $sampleVarCorLen / $sampleVar;
561         }
562         else {
563             $sampleVar = $sampleVarCorLen / $corLength;
564         }
565
566         #This is similiar to the Kappa from the 2007 Dolg ECP paper.
567         #Lower is better. Sample clock is in microseconds.
568         my $wfEfficiency = $dt * $sampleVar * $corLength * $sampleclock * 10.0;
569
570         $summary{$key} .=
571           sprintf " %10.3e %10.2f %10.2f %10s %10s %10s %15.5f\n",
572           $sampleVar,
573           $corLength,
574           $wfEfficiency,
575           $wallclock, $totalclock,
576           $iteration, ( $effdt * $iteration );
577     }
578     else {
579         $summary{$key} .=
580           sprintf " %10s %10s %10s %10s %10s %10s %15.5f\n",
581           "", "", "", "", "",
582           $iteration, ( $effdt * $iteration );
583     }
584
585     #if we are in enhanced text mode, we need to double escape the "_"
586     #$base =~ s/_/\\\_/g;
587     printf DATFILE "#%19s %20s %20s %40s\n", "dt=$dt", "$base", "E=$eavg",
588       "$key";
589     print DATFILE "$fordatfile\n\n";
590 }
591 close DATFILE;
592
593 chomp($headerLine);
594 if ( $extraTag ne "" ) {
595     printf "%30s $headerLine  %15s\n$lastlines", " ", $extraTag;
596 }
597 else {
598     printf "%30s $headerLine \n$lastlines", " ";
599 }
600
601 foreach $key ( sort byenergy keys %dt_ave_results ) {
602     if ( !exists $label{$key} ) {
603         $label{$key} = sprintf "%2i", ( scalar keys %label ) + 1;
604     }
605 }
606 printf "ID  %-30s %7s %5s %16s %4s %5s %10s %10s %10s %10s %10s %10s %15s\n",
607   "File Name", "dt", "nw", "Avg E", "Err", "Warn",
608   "Variance",
609   "Corr Len", "WF Eff", "Wall", "Total", "Iter", "effdt*iters";
610
611 foreach $sum ( sort bydt keys %summary ) {
612     $summary{$sum} =~ s/../$label{$sum}/;
613     print "$summary{$sum}";
614 }
615
616 die if ( $num_results <= 0 );
617
618 $ave_result /= $num_results;
619
620 #print "Average result = $ave_result\n";
621 $labelLen = $lenLong;
```

```
622 $labelLen = length "Label" if ( length "Label" > $labelLen );
623 printf "%5s %*s %10s %1s %20s %5s %7s   %-25s %5s %20s %20s %10s\n",
624   "ID", $labelLen, "Label",
625   "dt", "e", "Ref. Energy", "Num", "CI:BF", "NumJW", "NumW", "Average",
626   "Corr. E.", "Weight";
627 my %qref;
628 my %href;
629 my $dtref = 0;
630 my $cure  = "";
631
632 foreach $key ( sort byenergy keys %dt_ave_results ) {
633     my @keydata = split /&/, $key;
634
635     if ( $dt_num_results{$key} > 0 ) {
636         $dt_ave_results{$key} /= $dt_num_results{$key};
637     }
638     else {
639         print "Why does $key have $dt_num_results{$key} results?\n";
640         die;
641     }
642
643     if ( $dt_nme_results{$key} > 0 ) {
644         $dt_err_results{$key} =
645           sqrt( $dt_err_results{$key} / $dt_nme_results{$key} );
646     }
647     else {
648
649     }
650
651     printf
652       "%5i %*s %10s %1i %20s %5i %3i:%-3i   %-25s %5i %20s %20.10f %10.5f\n",
653       $label{$key},
654       $labelLen,
655       $shortnames{$key},
656       "$keydata[1]", $keydata[7], "$keydata[0]",
657       $dt_num{$key},
658       $keydata[5],
659       $keydata[2],
660       $keydata[3],
661       $keydata[4],
662       getEnergyWError( $dt_ave_results{$key}, $dt_err_results{$key} ),
663       ( $keydata[0] - $dt_ave_results{$key} ),
664       $dt_num_results{$key};
665 }
666
667 print "\n\n";
668
669 #matrix output
670 #the data is sorted according to dt first
671 #we calculate the difference for for all results available
672 #but we don't compare calculations if dt and energy are different
673 my %comparisons;
674
675 #A + B = C + D
676 foreach $A ( sort bydt keys %dt_ave_results ) {
677     my @Adata = split /&/, $A;
678     foreach $C ( sort bydt keys %dt_ave_results ) {
679         my @Cdata = split /&/, $C;
680         next if ( !areComparable( $A, $C ) );
681
682         foreach $B ( sort bydt keys %dt_ave_results ) {
683
684             #next if($A eq $B || $A eq $C || $B eq $C);
685             next if ( !areComparable( $A, $B ) );
686
687             my @Bdata = split /&/, $B;
688             my $a     = $dt_ave_results{$A};
```

```
689             my $b     = $dt_ave_results{$B};
690             my $c     = $dt_ave_results{$C};
691             next
692               if ( $a < $c || $a < $b )
693               ;  #otherwise we'll get two of every comparison
694                  #next if($a < $c); #otherwise we'll get two of every comparison
695
696             my $aOrb = $Adata[6];
697             my $bOrb = $Bdata[6];
698             my $cOrb = $Cdata[6];
699
700             ( $aMult, $bMult, $cMult ) =
701               getFormula( $Adata[2], $Bdata[2], $Cdata[2], $orbFilter );
702
703             #print "$a $b $c ($aMult,$bMult,$cMult) \n" if($bMult != 0);
704             next if ( $a < $b && $bMult > 0 );
705             next
706               if ( $aMult == 0 || $cMult == 0 )
707               ;    #the results are not comparable if either is zero
708                    #This eliminates a lot of the meaningless comparisons
709             my $orbsMatch = 0;
710             $orbsMatch = 1 if ( $aMult * $aOrb == $cMult * $cOrb );
711             next if ( $orbsMatch == 0 && $orbFilter == 1 && $bMult == 0 );
712
713             #So that we're comparing the difference
714             $cMult *= -1;
715
716 #print "$orbsMatch = ($aMult * $aOrb == $cMult * $cOrb)   ($aMult,$bMult,$cMult)\n";
717 #print "$orbsMatch \n";
718
719             my $diff  = $a * $aMult + $b * $bMult + $c * $cMult;
720             my $stdA  = abs( $dt_err_results{$A} * $aMult );
721             my $stdB  = abs( $dt_err_results{$B} * $bMult );
722             my $stdC  = abs( $dt_err_results{$C} * $cMult );
723             my $diffe = sqrt( $stdA * $stdA + $stdB * $stdB + $stdC * $stdC );
724
725             $diff  *= $units;
726             $diffe *= $units;
727
728             my $comparison = "";
729             my $aStr =
730               getEnergyWError( $dt_ave_results{$A}, $dt_err_results{$A} );
731             my $bStr =
732               getEnergyWError( $dt_ave_results{$B}, $dt_err_results{$B} );
733             my $cStr =
734               getEnergyWError( $dt_ave_results{$C}, $dt_err_results{$C} );
735             my $diffStr = getEnergyWError( $diff, $diffe );
736
737 #print "($Adata[2],$Bdata[2],$Cdata[2]) => ($aMult,$bMult,$cMult) := $diffStr\n";
738
739             if ( $sumResults == 1 ) {
740                 $comparison .= sprintf "%3i %3i) %6s", $label{$A}, $label{$C},
741                   $Adata[1];
742                 my $aM = $aMult;
743                 my $bM = $bMult;
744                 my $cM = $cMult;
745                 $aM = " "  if ( $aMult == 1 );
746                 $bM = " "  if ( $bMult == 1 );
747                 $cM = "- " if ( $cMult == -1 );
748
749                 my $compType = sprintf " ${aM} %*s ", $lenLong, $shortnames{$A};
750
751                 if ( $bMult != 0 ) {
752                     $compType .= sprintf " +${bM} %*s ",
753                       $lenLong, $shortnames{$B};
754                 }
755                 $compType .= sprintf " +${cM} %*s ", $lenLong, $shortnames{$C};
```

```
756
757                 $compType =~ s/\+\-/\-/g;
758                 $comparison .= sprintf "%s =", $compType;
759             }
760             else {
761                 my $AJW = ( split /=/, $Adata[3] )[0];
762                 my $BJW = ( split /=/, $Bdata[3] )[0];
763                 my $CJW = ( split /=/, $Cdata[3] )[0];
764                 $comparison .= sprintf "%3i) %*s %15s %6s %3s:%2s:%-3s %5s | ",
765                   $label{$A}, $lenLong, $shortnames{$A},
766                   $aStr, $Adata[1], $Adata[5], $aOrb, $Adata[2], $AJW;
767
768                 if ( $bMult != 0 ) {
769                     $comparison .=
770                       sprintf "%3i) %*s %15s %6s %3s:%2s:%-3s %5s | ",
771                       $label{$B}, $lenLong, $shortnames{$B},
772                       $bStr, $Bdata[1], $Bdata[5], $bOrb, $Bdata[2], $BJW;
773
774                     $compType = "${aMult}A+${bMult}B+${cMult}C";
775                 }
776                 else {
777                     $compType = "${aMult}A+${cMult}B";
778                 }
779
780                 $comparison .= sprintf "%3i) %*s %15s %6s %3s:%2s:%-3s %5s | ",
781                   $label{$C}, $lenLong, $shortnames{$C},
782                   $cStr, $Cdata[1], $Cdata[5], $cOrb, $Cdata[2], $CJW;
783
784                 $compType =~ s/1//g;
785                 $compType =~ s/\+\-/\-/g;
786                 $comparison .= sprintf "%6s=", $compType;
787             }
788
789             if ($orbsMatch) {
790                 $comparison .= " ";
791             }
792             else {
793                 $comparison .= "*";
794             }
795
796             if (0) {
797                 $comparison .= sprintf " %9.5f", $diff;
798                 $comparison .= sprintf " +/- %-9.5f $unitsL", $diffe;
799             }
800             else {
801                 $comparison .= sprintf " %10s $unitsL", $diffStr;
802             }
803
804             if ($compareE) {
805                 foreach $etype ( reverse sort keys %{ $referenceE{$A} } ) {
806                     $eA = $referenceE{$A}{$etype} * $aMult;
807                     $eB = $referenceE{$B}{$etype} * $bMult;
808                     $eC = $referenceE{$C}{$etype} * $cMult;
809                     my $temp = ( $eA + $eB + $eC ) * $units;
810
811                     next
812                       if (!exists $referenceE{$C}{$etype}
813                         || abs($temp) < 1e-10 );
814
815                     if ( $sumResults == 1 ) {
816
817 #$comparison .= sprintf " %*s = %9.5f %s\n",(2*$lenLong+22),"",$temp,$etype;
818                         $comparison .= sprintf " %s = %9.5f", $etype, $temp;
819                     }
820                     else {
821                         $comparison .=
822                           sprintf
```

```
"\n     %*s %15.10f %23s |      %*s %15.10f %23s | %7s  %9.5f %s",
                            $lenLong, "", $referenceE{$A}{$etype}, "", $lenLong,
                            "", $referenceE{$C}{$etype}, "", "", $temp, $etype;
                    }
                }
            }
            $comparison .= sprintf "\n";

            if ($latexHelp) {

                #strip off any of the title after the first underscore
                my $nameA = $1 if ( $shortnames{$A} =~ /([\dA-Za-z]+)_/ );
                my $nameC = $1 if ( $shortnames{$C} =~ /([\dA-Za-z]+)_/ );

                $tempStr = "";
                $tempStr = sprintf "%5s & ", $Adata[1] if ($latexDTcol);
                $tempStr .= sprintf "%20s & %15s & %20s & %15s & %15s \\\\",
                  $nameA, $aStr, $nameC, $cStr, $diffStr;
                $tempStr =~ s/\./\&/g;
                $comparison = "$tempStr\n";
            }

            $comparisons{$comparison} = $diff if ( abs($diff) < 1000 );
        }
    }
}

if ($latexHelp) {
    if ($latexDTcol) {
        print <<LATEX_HEADER;
\\begin{center}
\\begin{table}[htdp]
\\caption{A \$\\leftarrow\$ B}
\\label{table:gen}
\\begin{tabular}{r\@{.}l r r\@{.}lr r\@{.}lr\@{.}l}
\\hline \\hline
\\multicolumn{2}{c}{\$\\delta t\$} & A &
\\multicolumn{2}{c}{DMC} & B &
\\multicolumn{2}{c}{DMC} &
\\multicolumn{2}{c}{\$\\Delta\$} \\\\
\\multicolumn{2}{c}{au\$^{-1}\$} &  &
\\multicolumn{2}{c}{au} &  &
\\multicolumn{2}{c}{au} &
\\multicolumn{2}{c}{$unitsL} \\\\
\\hline
LATEX_HEADER
    }
    else {
        print <<LATEX_HEADER;
\\begin{center}
\\begin{table}[htdp]
\\caption{A \$\\leftarrow\$ B}
\\label{table:gen}
\\begin{tabular}{r r\@{.}lr r\@{.}lr\@{.}l}
\\hline \\hline
A &
\\multicolumn{2}{c}{DMC} & B &
\\multicolumn{2}{c}{DMC} &
\\multicolumn{2}{c}{\$\\Delta\$} \\\\
  &
\\multicolumn{2}{c}{au} &  &
\\multicolumn{2}{c}{au} &
\\multicolumn{2}{c}{$unitsL} \\\\
\\hline
LATEX_HEADER
    }
}
```

```
890 foreach $key ( sort { $comparisons{$a} <=> $comparisons{$b} }
891     keys %comparisons )
892 {
893     print "$key";
894 }
895 if ($latexHelp) {
896     print <<LATEX_TAIL;
897 \\hline \\hline
898 \\end{tabular}
899 \\end{table}
900 \\end{center}
901 LATEX_TAIL
902 }
903
904 print "\n\n";
```

F.3 Convergence by Example

After we have run *summary.pl*, a data file was produced summarizing the data into a format that gnuplot can understand. To create the plots, we simply run the *plotter.pl* script in the same directory. For example, the command:

```
../bin/plotter.pl -i -t 2 -err -e 100
```

will run in non-interactive mode, selected by -i, instead of X11 mode. This command also selects the x-axis to be time, instead of iterations, -err indicates that we want error bars, and -e 100 specifies that only 1 out of every 100 data points should be plotted. The output is shown in Figure F.1. In this figure, we can see a few points of interest. First, notice that the result for awt0r5_Tv_exp_1 is quite distinct from the other data points. This could indicate a variety of problems, but on the other hand, it is fairly constant, so the calculation seems to be ok. We also notice that the error bars for this stream jump towards the end, which is evidence that the calculation ran into some problems. We see a similar jump in the error bars for some of the other calculations even though on the whole, there is remarkable agreement around the 103.5 kcal/mol expected answer.

F.4 Script: plotter.pl

```
1 #!/usr/bin/perl
2 #use strict;
3 my $path = `dirname $0`;
4 chomp($path);
5 require "$path/utilities.pl";
6
7 my $orbFilter = 1;
8 my $calcDiff  = 1;
```

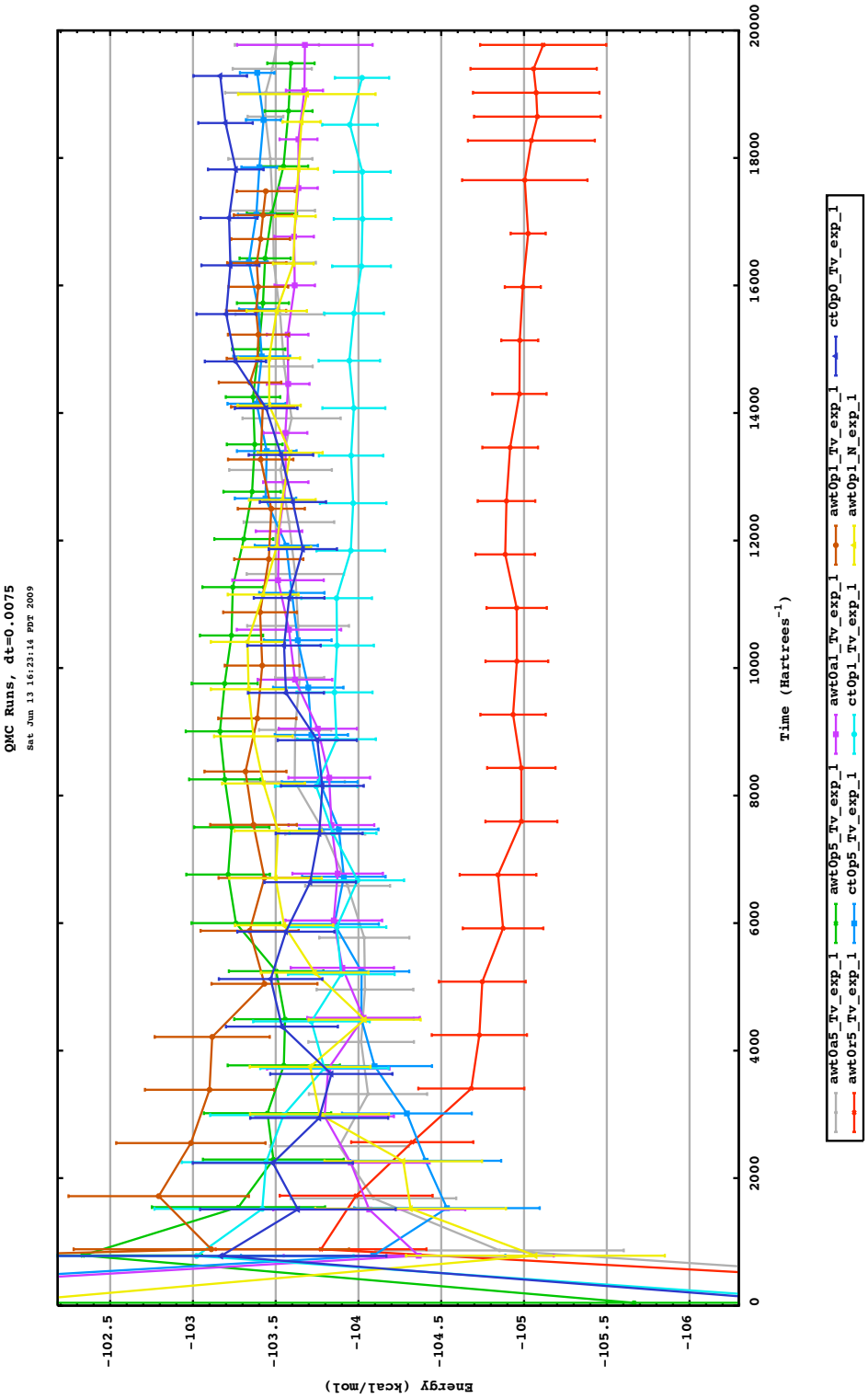

Figure F.1: Sample output from the script *plotter.pl*.

```
my $useAvg    = 1;
my $withErr   = 0;
my $spacef    = 0.3;
my $i_active  = 1;

#absolute energies (=0) or relative (=1) to each other?
my $shift = 1;

#should the x axis be iteration (=0), samples (=1) or time (=2)?
my $xtype = 0;

#add lines with these values:
my @exact_titles;
my @exact;

my $every = 15;
if ($withErr) {

    #error lines can be very messy, so decrease the
    #freqency of points
    $every = 100;
}

my $units  = 627.50960803;
my $unitsL = "kcal/mol";

while ( $#ARGV >= 0 && $ARGV[0] =~ /^-/ ) {
    $type  = shift(@ARGV);
    $param = "";

    if ( $type eq "-s" ) {
        $calcDiff = ( $calcDiff + 1 ) % 2;
        print "Using calcDiff = $calcDiff\n";
    }
    elsif ( $type eq "-a" ) {
        $useAvg = ( $useAvg + 1 ) % 2;
        print "Using useAvg = $useAvg\n";
    }
    elsif ( $type eq "-o" ) {
        $orbFilter = ( $orbFilter + 1 ) % 2;
        print "Using orbFilter = $orbFilter\n";
    }
    elsif ( $type eq "-i" ) {
        $i_active = ( $i_active + 1 ) % 2;
        print "Using interactive = $i_active\n";
    }
    elsif ( $type eq "-t" ) {
        $param = shift(@ARGV);
        $xtype = $param;
        print "Using xtype = $xtype\n";
    }
    elsif ( $type eq "-f" || $type eq "-space" ) {
        $param  = shift(@ARGV);
        $spacef = $param;
        print "Using spacef = $spacef\n";
    }
    elsif ( $type eq "-e" || $type eq "-every" ) {
        $param   = shift(@ARGV);
        $every   = $param;
        $withErr = 1;
        print "Using every = $every\n";
    }
    elsif ( $type eq "-err" || $type eq "-error" ) {
        $withErr = ( $withErr + 1 ) % 2;
        print "Using withErr= $withErr\n";
    }
    elsif ( $type eq "-exp" ) {
```

```
76          $title = shift(@ARGV);
77          $nrg   = shift(@ARGV);
78          push( @exact_titles, $title );
79          push( @exact,        $nrg );
80          print "Adding line called $title at $nrg\n";
81      }
82      elsif ( $type eq "-x" ) {
83          $param = shift(@ARGV);
84          if ( $param eq "ch2" ) {
85              push( @exact_titles, "exp" );
86              push( @exact,        -9.353 );
87          }
88          elsif ( $param == 1 ) {
89              push( @exact_titles, "exp" );
90              push( @exact,        -21.5539 );
91              push( @exact_titles, "ccsdt" );
92              push( @exact,        -22.5373 );
93          }
94          else {
95              print "Unrecognized energy choice: $param\n";
96          }
97      }
98      elsif ( $type eq "-u" ) {
99          $param = shift(@ARGV);
100         $param = lc($param);
101         if ( $param =~ /kcal/ ) {
102             $units  = 627.50960803;
103             $unitsL = "kcal/mol";
104         }
105         elsif ( $param =~ /ev/ ) {
106             $units  = 27.211399;
107             $unitsL = "eV";
108         }
109         elsif ( $param =~ /kj/ ) {
110             $units  = 2625.5002;
111             $unitsL = "kJ/mol";
112         }
113         elsif ( $param =~ /cm/ ) {
114             $units  = 219474.63;
115             $unitsL = "cm^-1";
116         }
117         elsif ( $param =~ /au/ || $param =~ /hart/ ) {
118             $units  = 1;
119             $unitsL = "au";
120         }
121         print "Using $unitsL energy units, conversion = $units\n";
122     }
123     else {
124         print "Unrecognized option: $type\n";
125         die;
126     }
127 }
128 if ( $#ARGV >= 0 ) {
129     print "Unrecognized options: @ARGV\n";
130 }
131 my $d = qx! date +%F.%H-%M-%S !;
132 chomp($d);
133 my $date = `date`;
134 chomp $date;
135
136 #my $gnuplot = "/usr/local/bin/gnuplot";
137 my $gnuplot = "/ul/amosa/bin/gnuplot";
138
139 #you might need to add this command to your .cshrc
140 # `setenv GDFONTPATH /Library/Fonts:/System/Library/Fonts`;
141 # `setenv GDFONTPATH /usr/share/fonts/bitstream-vera/`;
142
```

```
143 my @gnutype = split / +/, `$gnuplot -V`;
144 if ( $gnutype[1] < 4.3 ) {
145
146     #we need the extra features that version 4.3 has
147     print
148       "GNUPLOT version = $gnutype[1] is incompatible for executable $gnuplot\n";
149     die;
150 }
151
152 sub deleteData {
153     foreach $test (@_) {
154
155         #print "Deleting $test\n";
156     }
157
158     open( DATA, "plotfile.dat" );
159
160     my $newdata = "";
161     my $match   = 0;
162     my $num     = 0;
163     my $line    = <DATA>;
164     while ($line) {
165         if ( $num < $#_ + 1 ) {
166             foreach $test (@_) {
167                 if ( $line =~ /$test/ ) {
168                     chomp($line);
169                     $match = 1;
170                 }
171             }
172         }
173
174         if ( $match == 1 ) {
175             $line = <DATA>;
176             while ( $line =~ /\d/ ) {
177                 $line = <DATA>;
178             }
179             $line  = <DATA>;
180             $match = 0;
181             $num += 1;
182         }
183         else {
184             $newdata .= "$line";
185         }
186
187         $line = <DATA>;
188     }
189
190     close(DATA);
191
192     open( NEWDATA, ">new_plotfile.dat" );
193     print NEWDATA "$newdata";
194     close(NEWDATA);
195
196     `mv new_plotfile.dat plotfile.dat`;
197 }
198
199 sub operateTwo {
200     my $newdata = "";
201
202     #
203     # Operate on two streams:
204     # final = $fconst * $fkey + $sconst * $skey
205     my ( $fconst, $fkey, $sconst, $skey ) = @_;
206     printf "%10.5f * (%-60s) + %10.5f * (%-60s)\n", $fconst, $fkey, $sconst,
207       $skey;
208
209     open( DATA, "plotfile.dat" );
```

```
210
211     my $line = <DATA>;
212     $line = <DATA> while ( $line !~ /$fkey$/ && $line !~ /$skey$/ );
213     if ( $line =~ /$skey/ && $fkey ne $skey ) {
214
215         #we found the second key first, so swap
216         $temp = $fkey;
217         $fkey = $skey;
218         $skey = $temp;
219
220         $temp   = $fconst;
221         $fconst = $sconst;
222         $sconst = $temp;
223     }
224
225     my @ftitle = split /[ =]+/, $line;
226     $line = <DATA>;
227     my @first_data;
228     while ( $line =~ /\d/ ) {
229         push( @first_data, $line );
230         $line = <DATA>;
231     }
232
233     #this is the second blank line
234     $line = <DATA>;
235
236     #this is header of the next data
237     $line = <DATA>;
238
239     while ( $line !~ /$skey$/ ) {
240         $line = <DATA>;
241     }
242     my @stitle = split /[ =]+/, $line;
243     $line = <DATA>;
244     my @second_data;
245     while ( $line =~ /\d/ ) {
246         push( @second_data, $line );
247         $line = <DATA>;
248     }
249
250     if ( $#first_data < $#second_data ) {
251
252         #it's easier to add the shorter to the longer
253         my @temp = @first_data;
254         @first_data  = @second_data;
255         @second_data = @temp;
256
257         @temp   = @ftitle;
258         @ftitle = @stitle;
259         @stitle = @temp;
260
261         $temp = $fkey;
262         $fkey = $skey;
263         $skey = $temp;
264
265         $temp   = $fconst;
266         $fconst = $sconst;
267         $sconst = $temp;
268     }
269
270     $first_max  = ( split / +/, $first_data[$#first_data] )[1];
271     $second_max = ( split / +/, $second_data[$#second_data] )[1];
272     chomp($first_max);
273     chomp($second_max);
274     $first_min  = ( split / +/, $first_data[0] )[1];
275     $second_min = ( split / +/, $second_data[0] )[1];
276     chomp($first_min);
```

```
277     chomp($second_min);
278
279 #print "First max = $first_max Second max = $second_max fmin = $first_min smin = $second_min\n";
280 #print "Data first = $#first_data Data second = $#second_data\n";
281 #print "last = $first_data[$#first_data]";
282     my $s  = 0;
283     my @sl = split / +/, $second_data[$s];
284     my $si = $sl[1];
285
286     ( $fe, $fw ) = split /:/, $ftitle[5];
287     ( $se, $sw ) = split /:/, $stitle[5];
288
289     #if the stream hasn't been a weight yet, then initialize it with 1
290     $fw = 1.0 if ( $fw eq "" );
291     $sw = 1.0 if ( $sw eq "" );
292
293     my $fbase = `basename $ftitle[3]`;
294     chomp($fbase);
295
296     #$fbase =~ s/_[\d]+$//g;
297     my $sbase = `basename $stitle[3]`;
298     chomp($sbase);
299
300     #$sbase =~ s/_[\d]+$//g;
301
302     my $title_new;
303     my $new_weight;
304     if ( $fconst * $sconst > 0 ) {
305
306         #we're adding streams
307         if ( length $fbase < length $sbase ) {
308             $title_new = "$fbase";
309         }
310         else {
311             $title_new = "$sbase";
312         }
313
314         #the weight of the product stream will be
315         #the sum of the weights from the input streams,
316         #each scaled by a constant
317         $new_weight = $fw * $fconst + $sw * $sconst;
318         $fconst *= $fw / $new_weight;
319         $sconst *= $sw / $new_weight;
320     }
321     else {
322
323         #we're subtracting streams
324
325         #$title_new = "${fconst}x${fbase}-${sconst}x${sbase}";
326         my $ffactor;
327         if ( abs( $fconst + 1 ) < 1e-5 ) {
328
329             # -1
330             $ffactor = "-";
331         }
332         elsif ( abs( $fconst - 1 ) < 1e-5 ) {
333
334             # +1
335             $ffactor = "";
336         }
337         else {
338             $ffactor = "$fconst";
339         }
340         my $sfactor;
341         if ( abs( $sconst + 1 ) < 1e-5 ) {
342
343             # -1
```

```
344             $sfactor = "-";
345         }
346         elsif ( abs( $sconst - 1 ) < 1e-5 ) {
347 
348             # +1
349             $sfactor = "";
350         }
351         else {
352             $sfactor = "$sconst";
353         }
354 
355         #normalize the weights now
356         $new_weight = $fw + $sw;
357         $title_new  = "$fbase:${ffactor}A+${sfactor}B";
358         $title_new =~ s/A\+\-B/A\-B/;
359         $title_new =~ s/-A\+B/B\-A/;
360     }
361 
362     my $e_new = sprintf "%-.10f", ( $fconst * $fe + $sconst * $se );
363 
364     #printf " E_New: $fconst * $fe + $sconst * $se = $e_new\n";
365     $e_new .= ":$new_weight";
366 
367     $newdata .= sprintf "#%19s %20s %20s %40s", "dt=$ftitle[2]", "$title_new",
368       "E=$e_new", "$ftitle[6]=$ftitle[7]";
369 
370 #printf         "#%19s %20s %20s %40s\n", "dt=$ftitle[2]","$ftitle[3]","E=$e_new","$ftitle[6]=$ftitle[7]";
371 
372     for ( my $f = 0 ; $f <= $#first_data ; $f++ ) {
373         @fl = split / +/, $first_data[$f];
374         $fi = $fl[1];
375 
376         my $new;
377 
378         #num samples
379         $new = ( $fl[1] + $sl[1] ) / 2;
380         $newdata .= sprintf "%20s ", $new;
381 
382         #energy
383         $new = $fconst * $fl[2] + $sconst * $sl[2];
384         $newdata .= sprintf "%20.10e ", $new;
385         if ( $f == 0 ) {
386 
387 #printf "Energy: %10.5f * (%-20.10f) + %10.5f * (%-20.10f) = %20.10f\n",$fconst, $fl[2], $sconst, $sl[2], $new;
388         }
389 
390         #variance
391         $new =
392           ( $fconst * $fl[3] ) * ( $fconst * $fl[3] ) +
393           ( $sconst * $sl[3] ) * ( $sconst * $sl[3] );
394         $newdata .= sprintf "%20.10e ", sqrt($new);
395 
396         #num samples
397         $new = ( $fl[4] + $sl[4] ) / 2;
398         $newdata .= sprintf "%20s ", $new;
399 
400         $newdata .= sprintf "\n";
401 
402         #print "Averaging $f:$s  $fi with $si\n";
403 
404         while ( $si < $fi && $s <= $#second_data ) {
405             @sl = split / +/, $second_data[$s];
406             $si = $sl[1];
407             $s += 1;
408         }
409     }
410 
```

```
411      close(DATA);
412
413      $newdata .= "\n\n";
414      return $newdata;
415  }
416
417  sub averageTwo {
418
419   #
420   # This function will look for two plots that represent equilvalent data and can
421   # be averaged.
422   #
423
424      my @lines = `grep dt plotfile.dat`;
425      chomp(@lines);
426      foreach ( my $fset = 0 ; $fset < $#lines ; $fset++ ) {
427          @fdata = split /\s+/, $lines[$fset];
428          chomp @fdata;
429          foreach ( my $sset = $fset + 1 ; $sset <= $#lines ; $sset++ ) {
430              @sdata = split /\s+/, $lines[$sset];
431              chomp(@sdata);
432
433              if ( $fdata[4] eq $sdata[4] ) {
434                  print "Average $fdata[2] with $sdata[2]\n";
435                  my $newdata = operateTwo( 1.0, $fdata[4], 1.0, $sdata[4] );
436                  deleteData( $lines[$fset], $lines[$sset] );
437
438                  open( NEWDATA, ">>plotfile.dat" );
439                  print NEWDATA "$newdata";
440                  close(NEWDATA);
441                  return 1;
442              }
443          }
444      }
445      return 0;
446  }
447
448  sub subtractTwo {
449      my @lines = `grep dt plotfile.dat`;
450      my @keys;
451      my @titles;
452      foreach $line (@lines) {
453          my @data = split / +/, $line;
454          chomp @data;
455          push( @keys,   $data[4] );
456          push( @titles, $data[2] );
457      }
458      @keys = sort byenergy @keys;
459
460      my %newdata;
461      for ( my $i = $#keys ; $i >= 0 ; $i-- ) {
462          $iKey = $keys[$i];
463          my @iData = split /&/, $iKey;
464          for ( my $j = 0 ; $j < $i ; $j++ ) {
465              $jKey = $keys[$j];
466              my @jData = split /&/, $jKey;
467              next if ( !areComparable( $iKey, $jKey ) );
468
469              my $temp = 0;
470              ( $iMult, $temp, $jMult ) =
471                getFormula( $iData[2], 0, $jData[2], $orbFilter );
472              my $orbsMatch = 0;
473              $orbsMatch = 1 if ( $iMult * $iData[6] == $jMult * $jData[6] );
474              next if ( $orbsMatch == 0 && $orbFilter == 1 && $temp == 0 );
475
476              #the results are not comparable if either is zero
477              next if ( $iMult == 0 || $jMult == 0 );
```

```
478
479         #printf "(%2i,%2i) $iMult x %-60s : $jMult x %-60s\n",$i,$j,$iKey,$jKey;
480         #printf "(%2i,%2i) ",$i,$j;
481             $key = "";
482             $key .= ( $iMult > $jMult ? $iMult : $jMult );
483             $key .= "x";
484             $key .= ( $iMult < $jMult ? $iMult : $jMult );
485
486             print "Subtracting: $i) $titles[$i] - $j) $titles[$j]\n";
487             $newdata{"$key"} .=
488               operateTwo( $iMult, $iKey, -1.0 * $jMult, $jKey );
489         }
490     }
491
492     return 0 if ( scalar keys %newdata == 0 );
493
494     open( NEWDATA, ">new_plotfile.dat" );
495     foreach $key ( reverse sort keys %newdata ) {
496
497         #assume that one one with the highest numbers in the formula
498         #are the ones we want to print
499         print NEWDATA "$newdata{$key}";
500
501         #if you want all, commment this line:
502         last;
503     }
504     close(NEWDATA);
505     return 0;
506 }
507
508 if ($useAvg) {
509     while ( averageTwo() ) { }
510 }
511
512 if ($calcDiff) {
513     my $once = 0;
514     subtractTwo();
515
516     if ( -e "new_plotfile.dat" ) {
517         `mv new_plotfile.dat plotfile.dat`;
518     }
519 }
520
521 #now it's time to generate gnuplot gifs
522 my @titles;
523 my @energies;
524 my @dt_values;
525 my @keys;
526
527 my $all_dt   = "";
528 my $all_form = "";
529
530 #let's not assume we know what's in the data files
531 my @lines = `grep dt plotfile.dat`;
532 chomp(@lines);
533 foreach $line (@lines) {
534     my @data = split /[= ]+/, $line;
535     my ( $nrg, $num ) = split /:/, $data[5];
536
537     #print "$line\n";
538     printf
539 "%-30s: from $num data sets, dt=$data[2], with final energy %20.10e $unitsL\n",
540       $data[3], ( $nrg * $units );
541
542     if ( $all_dt eq "" ) {
543         $all_dt = $data[2];
544     }
```

```
545      elsif ( $all_dt eq "-1" ) {
546
547      }
548      elsif ( $data[2] ne $all_dt ) {
549          $all_dt = "-1";
550      }
551
552      if ($calcDiff) {
553          my @td = split /:/, $data[3];
554          if ( $all_form eq "" ) {
555              $all_form = $td[1];
556          }
557          elsif ( $all_form == -1 ) {
558
559          }
560          elsif ( "$all_form" ne "$td[1]" ) {
561              $all_form = -1;
562          }
563      }
564  }
565
566  my $y_min;
567  my $y_max;
568  my $y_err;
569  open( DAT_FILE, "plotfile.dat" );
570  my $line = <DAT_FILE>;
571  while ($line) {
572      chomp $line;
573      my @data = split /[= ]+/, $line;
574
575      if ( $line =~ "dt=" ) {
576          push( @energies,  "$data[5]" );
577          push( @dt_values, "$data[2]" );
578          push( @keys,      "$data[6]" );
579
580          my @td = split /:/, $data[3];
581          my $ti = $td[0];
582          if ( $all_dt == -1 && $data[2] != 0 ) {
583              $ti .= ", dt=$data[2]";
584          }
585          if ( $all_form == -1 && $data[2] != 0 ) {
586              $ti .= ", $td[1]";
587          }
588
589          my $key = ( split / +/, $line )[4];
590          my @kd = split /&/, $key;
591
592          my $bf = $kd[2];
593          my $jw = $kd[3];
594
595          $jw =~ s/18,//g;
596          $jw =~ s/18//g;
597
598          $ti .= sprintf " %3s; %s", $bf, $jw;
599
600          push( @titles, "$ti" );
601      }
602
603      $line = <DAT_FILE>;
604
605      #Make sure we have the last line in a series
606      if ( $line !~ /[0-9]/ && "$data[2]" =~ /[0-9]/ ) {
607          $y_err = $data[3];
608          if ( $data[2] < $y_min || $y_min == 0 ) {
609              $y_min = $data[2];
610              $y_min -= $y_err if ( $withErr || $#lines == 0 );
611          }
```

```
612         if ( $data[2] > $y_max || $y_max == 0 ) {
613             $y_max = $data[2];
614             $y_max += $y_err if ( $withErr || $#lines == 0 );
615         }
616     }
617 }
618 close(DAT_FILE);
619
620 $y_min *= $units;
621 $y_max *= $units;
622 my $intr;
623 my $reference;
624 if ( $calcDiff == 0 ) {
625
626     #make a guess for the stoicheometry
627     my $ratio = $y_min / $y_max;
628     $intr = int( $ratio + 0.5 );
629
630     #and shift the axis to reflect this
631     #we'll have gnuplot shift the plots
632     $y_max *= $intr;
633     $reference = $y_min;
634
635     if ( $shift == 1 ) {
636         $shift = $y_min;
637         $y_max = $y_max - $y_min;
638         $y_min = 0;
639     }
640     else {
641         $shift = 0;
642     }
643 }
644 else {
645
646     # since we use shift as a flag and a parameter, we need to
647     # set to zero before plotting
648     $shift = 0;
649 }
650
651 my $space = $spacef * ( $y_min - $y_max );
652 $y_min += $space;
653 $y_max -= $space;
654
655 my $file_name   = "qmc";
656 my $title_extra = "";
657 if ( $all_dt != -1 && $all_dt != 0 ) {
658     $title_extra .= ", dt=$all_dt";
659     $file_name   .= "_$all_dt";
660 }
661 if ( $all_form != -1 && $all_form ne "" ) {
662     $title_extra .= ", $all_form";
663 }
664
665 my $ylabel = "Energy ($unitsL)";
666 my $xindex;
667 my $xlabel;
668 if ( $xtype == 0 ) {
669     $xindex = 4;
670     $xlabel = "Num Iterations";
671 }
672 elsif ( $xtype == 1 ) {
673     $xindex = 1;
674     $xlabel = "Num Samples";
675 }
676 elsif ( $xtype == 2 ) {
677     $xindex = 4;
678     $xlabel = "Time (Hartrees^{-1})";
```

```
}
$file_name .= sprintf "_%i", ( $#{titles} + 1 );
$gnuplot .= " -geometry 1280x740";    #this is optimized for Amos' laptop...
open( GNUPLOT, "|$gnuplot" ) or die "Can't open GNUPLOT= $gnuplot\n";

#open(GNUPLOT, ">gnuplot.gnu") or die "Can't open GNUPLOT= $gnuplot\n";

if ($i_active) {
    print "Plotting graph $file_name with X11\n";

#print GNUPLOT "set terminal x11 reset persist enhanced font \"Courier-Bold,12\" linewidth 2\n";
    print GNUPLOT
"set terminal x11 persist raise enhanced font \"Courier-Bold,12\" title \"$file_name\" dashed linewidth 2\n";
}
else {
    $file_name .= sprintf "_$d.pdf", ( $#{titles} + 1 );
    print "Writing graph in: $file_name\n";
    `/bin/rm -f $file_name`;
    print GNUPLOT
"set term pdf color enhanced font \"Courier-Bold,12\" linewidth 7 dashed dl 3 size 17.5,10\n";
    print GNUPLOT "set output \"$file_name\"\n";
}

print GNUPLOT <<gnuplot_Commands_Done;
#fonts with extensions "ttf" and "dfont" will work
#here is a list of available fonts: Chalkboard Helvetica Times
#Courier Monaco LucidaGrande
#set term gif crop enhanced font 'Monaco' 8

#fonts on hive:
#set term gif crop enhanced font 'VeraMono' 8
#set term svg dynamic enhanced font "VeraMono,8"
set mouse zoomjump
set size 0.9,1

set nokey
set key outside below box noenhanced Left reverse
set yrange[$y_min:$y_max]
set xrange[0:]
set title "QMC Runs${title_extra}\\n{/=8${date}}"
set xlabel "$xlabel"
set ylabel "$ylabel"
set grid ytics
set mytics
set tics scale 1.5, 0.75

gnuplot_Commands_Done

my $numLC = 11;
my @goodlt;
push( @goodlt, 1 );
push( @goodlt, 3 );
push( @goodlt, 5 );
push( @goodlt, 4 );
push( @goodlt, 6 );
push( @goodlt, 7 );

my $plotline = "plot ";
if ( $#exact >= 0 ) {
    for ( my $i = 0 ; $i <= $#exact ; $i++ ) {
        $plotline .= "$exact[$i] title \"$exact_titles[$i]\" with lines,\\\n";
    }
}
for ( my $i = 0 ; $i <= $#titles ; $i++ ) {
    my $factor = 1;

    if ( $calcDiff == 0 ) {
```

```
746
747         #now we calculate the factor used to indicate stoicheometry
748         if ( abs( $intr * $energies[$i] - $reference / $units ) < 0.1
749             && $intr != 1 )
750         {
751             $factor *= $intr;
752             $titles[$i] .= " x$factor";
753         }
754     }
755
756     my $xfactor = 1;
757     $xfactor = $dt_values[$i] if ( $xtype == 2 );
758
759     my $lt = $goodlt[ int( $i / $numLC ) ];
760     my $lc = $i % $numLC;
761
762     $plotline .=
763 " \"plotfile.dat\" index $i every vEvery using (\$$xindex * $xfactor):(\$2*vUnits*$factor-$shift):(\$3*vUnits) lc $lc l
764     $plotline .= " with yerrorlines";
765
766     #$plotline .= ",\\" if($i != $#titles);
767     #$plotline .= "\n";
768     $plotline .= "," if ( $i != $#titles );
769
770 }
771 my $plotline_noerr = $plotline;
772 $plotline_noerr =~ s/yerrorlines/lines/g;
773 print GNUPLOT "vEvery = $every\n";
774 print GNUPLOT "vUnits = $units\n";
775 if ($withErr) {
776     print GNUPLOT "$plotline\n";
777 }
778 else {
779     print GNUPLOT "$plotline_noerr\n";
780 }
781
782 print GNUPLOT "v=0\n";
783 print GNUPLOT "bind e 'v=v+1; if(v%2) $plotline; else $plotline_noerr'\n";
784 print GNUPLOT "k=0\n";
785 print GNUPLOT
786   "bind k 'k=k+1; if(k%2) set nokey; replot; else set key; replot'\n";
787 print GNUPLOT
788   "bind '-' 'vEvery=vEvery+5; if(v%2) $plotline; else $plotline_noerr'\n";
789 print GNUPLOT
790 "bind '=' 'vEvery=vEvery-5; if(vEvery < 1) vEvery = 1; if(v%2) $plotline; else $plotline_noerr'\n";
791 print GNUPLOT
792 "bind '1' 'vUnits=1; set yrange [$y_min/$units:$y_max/$units]; if(v%2) $plotline; else $plotline_noerr'\n";
793
794 print GNUPLOT "pause mouse button2\n";
795
796 #print GNUPLOT "pause -1 'Hit return to continue'\n";
797 #print GNUPLOT "pause -1\n";
798 close(GNUPLOT);
799
800 #`/bin/rm $_.dat`;
801 #`open $file_name`;
802 if ( $i_active == 0 ) {
803 `bash -c \"echo Current directory \" | /usr/bin/mutt -s \"[jastrows] $file_name\" -a $file_name nitroamos\@gmail.com`;
804     `rm $file_name`;
805 }
```

F.5 Script: utilities.pl

Finally, we include our script *utilities.pl* which contains several routines used by our other scripts. Of particular interest is the routine *areComparable* which is used to decide whether two calculations are comparable. Some of the checks are based on what it finds in the input file, such as time step, but others are based on the file names themselves, which can be used to store some additional information about the calculation. For example, I might want to make sure that the last number in the file name, which is usually used as an index, matches. The routine *getFormula* is used to guess the stoichiometry of a reaction, and *estimateTimeToFinish* can be used to guess how much time remains before a calculation completes. This function is particularly useful in conjunction with a queue command such as *qstat*, since you can submit a job requesting only the amount of time necessary, perhaps improving the run priority in the queue.

```
1  #!/usr/bin/perl
2  use POSIX;
3
4  sub areComparable {
5
6  # This function is used by the code to see if two calculations can be compared.
7  # The script will generate output comparing each result against all other results,
8  # which add up to quite a few comparisons, most of which are actually meaningless.
9  # So if they're meaningless, then return 0. You'll probably want to edit this function
10 # to choose your own comparisons.
11 #
12 # The input is from summary.pl, where each a key is created for each calculation:
13 # my $key = "$refE&$dt&$numbf&$numjw&$nw&$numci&$numor&$oepi&$short";
14 #
15     my ( $one, $two ) = @_;
16     my @od = split /&/, $one;
17     my @td = split /&/, $two;
18
19     return 0
20       if (
21         $od[0] == $td[0] ||    #compare energies
22         $od[1] != $td[1] ||    #compare dt
23         $od[4] != $td[4] ||    #compare num walkers
24         $od[7] != $td[7]
25       );                       #compare oepi
26
27     #make sure the jastrows are comparable
28     return 0 if ( $od[3] =~ /44/ && $td[3] !~ /44/ );
29     return 0 if ( $od[3] !~ /44/ && $td[3] =~ /44/ );
30
31 #the files are named something like awt0p2, so extract the letter after the 0 (or 4),
32 #p in this case, and make sure they match
33     my $oType = "";
34     my $tType = "";
35     $oType = $1 if ( $od[8] =~ /t\d(\w)/ );
36     $tType = $1 if ( $td[8] =~ /t\d(\w)/ );
37
38     #this probably needs to be turned off for atomization energies
39     return 0 if ( $oType ne $tType );
40
41     #make sure the last number in the file matches
```

```
    #This only makes a difference if we didn't average over the results.
    my $oLast = "";
    my $tLast = "";
    $oLast = $1 if ( $od[8] =~ /([\d\.]+)$/ );
    $tLast = $1 if ( $td[8] =~ /([\d\.]+)$/ );

    #return 0 if($oLast ne $tLast);

    return 1;
}

#alphabet first, numerical second
sub a1n2 {
    my @adata = split /&/, $a;
    my @bdata = split /&/, $b;
    $bdata[1] <=> $adata[1];
    if ( $adata[0] eq $bdata[0] ) {
        if ( $adata[3] eq $bdata[3] ) {
            $bdata[1] cmp $adata[1];
        }
        else {
            $adata[3] <=> $bdata[3];
        }
    }
    else {
        $bdata[0] cmp $adata[0];
    }
}

sub a2n3 {
    my @adata = split /&/, $a;
    my @bdata = split /&/, $b;
    if ( $adata[0] eq $bdata[0] ) {
        if ( $adata[5] eq $bdata[5] ) {

            #sort by opt iter
            $bdata[1] <=> $adata[1];
        }
        else {

            #sort by reference energy
            $adata[5] cmp $bdata[5];
        }
    }
    else {

        #Sort by jastrow type (e.g. s, t, UC, etc)
        $bdata[0] cmp $adata[0];
    }
}

sub byenergy {
    my @adata = split /&/, $a;
    my @bdata = split /&/, $b;
    if ( $adata[0] != $bdata[0] ) {
        $bdata[0] <=> $adata[0];
    }
    else {
        $bdata[1] <=> $adata[1];
    }
}

sub bydt {
    my @adata = split /&/, $a;
    my @bdata = split /&/, $b;
    if ( $adata[1] != $bdata[1] ) {

```

```
109         #compare dt
110         $bdata[1] <=> $adata[1];
111     }
112     else {
113 
114         #compare energies
115         $bdata[0] <=> $adata[0];
116     }
117 }
118 
119 sub gcf {
120     my ( $x, $y ) = @_;
121     ( $x, $y ) = ( $y, $x % $y ) while $y;
122     return $x;
123 }
124 
125 sub getEnergyWError {
126     my ( $nrg, $err ) = @_;
127     my $str = "";
128     if ( abs($err) == 0 ) {
129         $str = "$nrg";
130     }
131     else {
132         my $d = 1 - int( floor( log($err) / log(10.0) ) );
133         my $energy = floor( $nrg * pow( 10.0, $d ) + 0.5 ) / pow( 10.0, $d );
134         $str = sprintf "%.*f", $d, $energy;
135         my $error = floor( $err * pow( 10.0, $d ) + 0.5 );
136         $str = "$str($error)";
137     }
138 
139 #printf("nrg=%10.5f err=%10.5f d=%3i energy=%20f str=%s\n",$nrg,$err,$d,$energy,$str);
140     return $str;
141 }
142 
143 sub getFormula {
144     my ( $a, $b, $c, $orbFilter ) = @_;
145     my $am = $c;
146     my $bm = $b;
147     my $cm = $a;
148 
149     my $factor = 100;
150     while ( $factor != 1 ) {
151         $factor = gcf( $am, $cm );
152 
153         #print "gcf($ar,$cr) = $factor\n";
154         $am /= $factor;
155         $cm /= $factor;
156     }
157 
158     if (   $am == int($am)
159         && $cm == int($cm)
160         && $am < 10
161         && $cm < 10 )
162     {
163 
164         #return (0,0) if($ar*$cd[6] != $cr*$ad[6] &&
165         # $arbFilter);
166         #print "($a, 0, $c) => ($am, 0, $cm)\n";
167         return ( $am, 0, $cm );
168     }
169 
170     my $maxF = 6;
171     for ( $am = 1 ; $am <= $maxF ; $am += 1 ) {
172         for ( $bm = 1 ; $bm <= $maxF ; $bm += 1 ) {
173             for ( $cm = 1 ; $cm <= $maxF ; $cm += 1 ) {
174                 if ( $am * $a + $bm * $b == $cm * $c ) {
175 
```

```
                    #print "($a, $b, $c) => ($am, $bm, $cm)\n";
                    return ( $am, $bm, $cm );
                }
            }
        }
    }

    return ( 0, 0, 0 );
}

sub getFileAge {
    my ( $file, $abstime ) = @_;
    my $curTime = qx! date +%s !;
    my $data    = `/bin/ls -lh --time-style=+%s $file`;
    my @list    = split / +/, $data;
    $outSize = $list[4];
    my $outModTime = $curTime - $list[5];

    return $outModTime if ( $abstime == 1 );
    $char = " ";

    if ( $outModTime > 3600 ) {
        $outModTime /= 3600;
        $char = "h";
        if ( $outModTime > 24 ) {
            $outModTime /= 24;
            $char = "d";
        }
    }
    if ( $char eq " " ) {
        $outModTime = sprintf "%5.0f $char", $outModTime;
    }
    else {
        $outModTime = sprintf "%5.1f $char", $outModTime;
    }

    #$outModTime .= sprintf " %3s", $list[5];
    #$outModTime .= sprintf " %2s", $list[6];
    #$outModTime .= sprintf " %5s", $list[7];
    return $outModTime;
}

sub estimateTimeToFinish {
    my ( $outfile, $time ) = @_;
    return 0 if ( !( -e $outfile ) );
    my $base = substr( $outfile, 0, -4 );
    @newsteps = `grep "new steps" $outfile`;

    my $equilSteps = 0;
    my $totalSteps = 0;
    if ( $#newsteps < 0 ) {
        open( CKMFFILE, "${base}.ckmf" );
        while (<CKMFFILE>) {
            if ( $_ =~ m/^\s*max_time_steps\s*$/ ) {
                $_ = <CKMFFILE>;
                chomp;
                my @line = split /[ ]+/;
                $totalSteps += $line[1];
            }
            if ( $_ =~ m/^\s*equilibration_steps\s*$/ ) {
                $_ = <CKMFFILE>;
                chomp;
                my @line = split /[ ]+/;
                $equilSteps = $line[1];

                #$totalSteps += $line[1];
            }
```

```
243         }
244     }
245     else {
246         $totalSteps = ( split /\s+/, $newsteps[$#newsteps] )[12];
247     }
248 
249     @itertime = `grep "Average iterations per hour:" $outfile`;
250     my $curIter = ( split /\s+/, `tail -n 1 ${base}.qmc` )[1];
251     $curIter += $equilSteps if ( $curIter <= 0 );
252 
253     my $itersPerHour = 0;
254     if ( $#itertime < 0 && $time != 0 ) {
255         $itersPerHour = $curIter / $time;
256         $itersPerHour *= 3600;
257     }
258     elsif ( $#itertime >= 0 ) {
259         my $shift = $#itertime;
260 
261 #the correlated sampling phase runs faster per iteration, and we assume that we're currently
262 #in the longer phase, so we want to look back 2 iterations
263         $shift -= 1 if ( $shift > 0 );
264         $itersPerHour = ( split /\s+/, $itertime[$shift] )[4];
265     }
266     else {
267         return "0:0";
268     }
269     return "0" if ( $itersPerHour == 0 );
270     my $est = ( $totalSteps - $curIter ) / $itersPerHour;
271     my $hrs = int($est);
272     my $mns = int( ( $est - $hrs ) * 60.0 + 0.5 );
273     if ( $mns < 10 ) {
274         $mns = "0$mns";
275     }
276 
277 #print "$est => hrs = $hrs mns = $mns\n";
278 #print "totalSteps = $totalSteps curiter = $curIter itertime = $itersPerHour est = $est\n";
279     return "${hrs}:${mns}";
280 }
281 
282 sub getOPTHeader {
283     return "IDUE3L";
284 }
285 
286 sub getCKMFHeader {
287     my $header = sprintf "%69s%5s\n", "", " CUUN";
288 
289     $header .= sprintf "%-30s %2s 0 %3s %11s %1s %-7s %-7s %6s %-15s %8s %8s\n",
290       "Name", "  ", "NW", "EQ/Steps", "e", "dt", "nci:nbf",
291       getOPTHeader(),
292       "HF", "Age", "Size";
293     return $header;
294 }
295 
296 sub getCKMFSummary {
297     my ($ckmf) = @_;
298 
299     $base = substr( $ckmf, 0, -5 );
300     my $dirname = `dirname $base`;
301     chomp($dirname);
302     my $shortbase = `basename $base`;
303 
304     if ( $dirname eq "." ) {
305 
306     }
307     else {
308         my $nextbase = `basename $dirname`;
309         chomp($nextbase);
```

```
310         $shortbase = "$nextbase/$shortbase";
311     }
312
313     chomp($shortbase);
314     open( CKMFFILE, "$ckmf" );
315
316     while ( <CKMFFILE> !~ /&flags/ ) { }
317     $rt      = "";
318     $numbf   = 0;
319     $numci   = 0;
320     $hfe     = "";
321     $nw      = 0;
322     $dt      = 0;
323     $steps   = 0;
324     $eqsteps = 0;
325     $iseed   = 0;
326     $oepi    = 0;
327
328     $opt   = 0;
329     $optl  = 0;
330     $optci = 0;
331     $opt3  = 0;
332     $optUD = 0;
333     $optUU = 0;
334     $optNE = 0;
335     while (<CKMFFILE>) {
336
337         if ( $_ =~ m/^\s*run_type\s*$/ ) {
338             $_ = <CKMFFILE>;
339             chomp;
340             my @line = split /[ ]+/;
341             $rt = $line[1];
342         }
343         if ( $_ =~ m/^\s*energy\s*$/ ) {
344             $_ = <CKMFFILE>;
345             chomp;
346             my @line = split /[ ]+/;
347             $hfe = $line[1];
348         }
349         if ( $_ =~ m/^\s*nbasisfunc\s*$/ ) {
350             $_ = <CKMFFILE>;
351             chomp;
352             my @line = split /[ ]+/;
353             $numbf = $line[1];
354         }
355         if ( $_ =~ m/^\s*ndeterminants\s*$/ ) {
356             $_ = <CKMFFILE>;
357             chomp;
358             my @line = split /[ ]+/;
359             $numci = $line[1];
360         }
361         if ( $_ =~ m/^\s*dt\s*$/ ) {
362             $_ = <CKMFFILE>;
363             chomp;
364             my @line = split /[ ]+/;
365             $dt = $line[1];
366         }
367         if ( $_ =~ m/^\s*one_e_per_iter\s*$/ ) {
368             $_ = <CKMFFILE>;
369             chomp;
370             my @line = split /[ ]+/;
371             $oepi = $line[1];
372         }
373         if ( $_ =~ m/^\s*number_of_walkers\s*$/ ) {
374             $_ = <CKMFFILE>;
375             chomp;
376             my @line = split /[ ]+/;
```

```
377             $nw = $line[1];
378         }
379         if ( $_ =~ m/^\s*optimize_Psi\s*$/ ) {
380             $_ = <CKMFFILE>;
381             chomp;
382             my @line = split /[ ]+/;
383             $opt = $line[1];
384         }
385         if ( $_ =~ m/^\s*optimize_L\s*$/ ) {
386             $_ = <CKMFFILE>;
387             chomp;
388             my @line = split /[ ]+/;
389             $optl = $line[1];
390         }
391         if ( $_ =~ m/^\s*optimize_EE_Jastrows\s*$/ ) {
392             $_ = <CKMFFILE>;
393             chomp;
394             my @line = split /[ ]+/;
395             $optUU = $line[1];
396             $optUD = $line[1];
397         }
398         if ( $_ =~ m/^\s*optimize_EN_Jastrows\s*$/ ) {
399             $_ = <CKMFFILE>;
400             chomp;
401             my @line = split /[ ]+/;
402             $optNE = $line[1];
403         }
404         if ( $_ =~ m/^\s*optimize_UD_Jastrows\s*$/ ) {
405             $_ = <CKMFFILE>;
406             chomp;
407             my @line = split /[ ]+/;
408             $optUD = $line[1];
409         }
410         if ( $_ =~ m/^\s*optimize_UU_Jastrows\s*$/ ) {
411             $_ = <CKMFFILE>;
412             chomp;
413             my @line = split /[ ]+/;
414             $optUU = $line[1];
415         }
416         if ( $_ =~ m/^\s*optimize_CI\s*$/ ) {
417             $_ = <CKMFFILE>;
418             chomp;
419             my @line = split /[ ]+/;
420             $optci = $line[1];
421         }
422         if ( $_ =~ m/^\s*optimize_NEE_Jastrows\s*$/ ) {
423             $_ = <CKMFFILE>;
424             chomp;
425             my @line = split /[ ]+/;
426             $opt3 = $line[1];
427         }
428         if ( $_ =~ m/^\s*max_time_steps\s*$/ ) {
429             $_ = <CKMFFILE>;
430             chomp;
431             my @line = split /[ ]+/;
432             $steps = $line[1];
433         }
434         if ( $_ =~ m/^\s*equilibration_steps\s*$/ ) {
435             $_ = <CKMFFILE>;
436             chomp;
437             my @line = split /[ ]+/;
438             $eqsteps = $line[1];
439         }
440         if ( $_ =~ m/^\s*iseed\s*$/ ) {
441             $_ = <CKMFFILE>;
442             chomp;
443             my @line = split /[ ]+/;
```

```
444             $iseed = $line[1];
445         }
446
447         #any other interesting parameters?
448         #if($ARGV[0] != "" && $_ =~ m/$ARGV[0]/ && $_ !~ /\#/){
449         #    $name = $_;
450         #    chomp $name;
451         #    $_ = <CKMFFILE>;
452         #    chomp;
453         #    my @line = split/[ ]+/;
454         #    $val = $line[1];
455         #    $searchdata .= sprintf "%20s: %30s = %30s\n", "", $name, $val;
456         #}
457         if ( $_ =~ m/&geometry$/ ) {
458             last;
459         }
460     }
461
462     if ( $rt eq "variational" ) {
463         $rt = "v";
464     }
465     elsif ( $rt = "diffusion" ) {
466         $rt = "d";
467     }
468
469     my $outModTime = "";
470     my $outSize    = "";
471     my $ovData     = "";
472     my $failed     = 0;
473
474     if ( -e "$base.out" ) {
475         $outModTime = getFileAge( "$base.out", 0 );
476     }
477
478     if ( -e "$base.out" && $opt ) {
479         $data = `grep failed $base.out`;
480         $failed = 1 if ( length($data) > 0 );
481
482         if ( $failed == 1 ) {
483             $outModTime .= "*";
484         }
485         else {
486             $outModTime .= " ";
487         }
488
489         @list   = `grep "Objective Value" $base.out`;
490         $ovData = $list[$#list];
491         chomp($ovData);
492
493         @list = split /[ =]+/, $ovData;
494         $ovData = "";
495         if ( $#list == 9 ) {
496             $ovData .= sprintf "%2i",      $list[1];
497             $ovData .= sprintf " %15.10f", $list[5];
498             $ovData .= sprintf " %8.5f",   $list[7];
499             $ovData .= sprintf " %10s",    $list[9];
500         }
501
502         @newsteps = `grep "new steps" $base.out`;
503         my $curSteps = $steps;
504         if ( $#newsteps >= 0 ) {
505             $curSteps = ( split /\s+/, $newsteps[$#newsteps] )[12];
506         }
507         $ovData .= sprintf " %10s", $curSteps;
508     }
509
510     $steps_str = "";
```

```
511     if ( $steps >= 1000 * 1000 * 1000 ) {
512         $steps /= int( 1000 * 1000 * 1000 );
513         $steps_str = sprintf "%2.1fB", ${steps};
514     }
515     elsif ( $steps >= 1000 * 1000 ) {
516         $steps /= int( 1000 * 1000 );
517         $steps_str = sprintf "%2.1fM", ${steps};
518     }
519     elsif ( $steps >= 1000 ) {
520         $steps /= int(1000);
521         $steps_str = sprintf "%2.1fK", ${steps};
522     }
523     else {
524         $steps_str = "$steps";
525     }
526 
527     $eqsteps_str = "";
528     if ( $eqsteps >= 1000 * 1000 * 1000 ) {
529         $eqsteps /= int( 1000 * 1000 * 1000 );
530         $eqsteps_str = sprintf "%2.1fB", ${eqsteps};
531     }
532     elsif ( $eqsteps >= 1000 * 1000 ) {
533         $eqsteps /= int( 1000 * 1000 );
534         $eqsteps_str = sprintf "%2.1fM", ${eqsteps};
535     }
536     elsif ( $eqsteps >= 1000 ) {
537         $eqsteps /= int(1000);
538         $eqsteps_str = sprintf "%2.1fK", ${eqsteps};
539     }
540     else {
541         $eqsteps_str = "$eqsteps";
542     }
543 
544     my $oneliner = "";
545     $oneliner .=
546       sprintf
547       "%-30s %2s %1i %3i %5s/%-5s %1s %-7s %3i:%-3s %1i%1i%1i%1i%1i%1i %-15s",
548       $shortbase, $rt,          $opt,
549       $nw,        $eqsteps_str, $steps_str,
550       $oepi,      $dt,
551       ${numci}, ${numbf},
552       $optci, $optUD, $optUU, $optNE, $opt3, $optl, $hfe;
553     $oneliner .= sprintf " %10s", $outModTime;
554     $oneliner .= sprintf " %7s",  $outSize;
555     $oneliner .= sprintf " %50s", $ovData;
556     if ( $iseed != 0 ) {
557         $oneliner .= sprintf " iseed = $iseed";
558     }
559     $oneliner .= sprintf "\n";
560 
561     close(CKMFFILE);
562     return $oneliner;
563 }
564 
565 sub getEnergies {
566     my ( $filename, $energies ) = @_;
567     open( FILE, "$filename" );
568 
569     $more = 1;
570     while (<FILE>) {
571         $sampleclock = ( split /[ ]+/ )[8]
572           if (/Average microseconds per sample per num initial walkers/);
573         $sampleVar = ( split /[ ]+/ )[3]
574           if ( /Sample variance/ && $sampleVar == 0 );
575 
576         #this is to avoid processing warnings
577         next if ( $_ =~ /[=:]/ && $_ !~ /Results/ );
```

```
578
579         chomp;
580         @data = split /[ ]+/;
581
582         #this is the number of data elements per line
583         #it can have the letter 'e' or 'E' since scientific notation uses them
584         if ( $#data >= 8 && $_ !~ /[A-DF-Za-df-z]+/ && $more ) {
585             $counter++;
586             $iteration = $data[1];
587             $eavg      = $data[2];
588             $estd      = $data[3];
589             if ( abs($eavg) > 1e-10 ) {
590                 push( @$energies, $eavg );
591             }
592         }
593         elsif (/Results/) {
594
595             #$more = 0;
596         }
597     }
598     close(FILE);
599 }
600
601 # this function will fill in the files array with
602 # all files that have the extension ext. there are
603 # a few known directories it will not descend into
604 sub getFileList {
605     my ( $ext, $files ) = @_;
606
607 #this will scan through all the subdirectories in the $files array looking for $ext files
608     my $clean = 0;
609     my $loops = 0;
610     while ( $clean == 0 ) {
611         $loops++;
612         $clean = 1;
613         my @newfiles;
614
615         for ( my $index = 0 ; $index <= $#$files ; $index++ ) {
616             my $cur = ${@$files}[$index];
617             chomp($cur);
618
619 #there are some obvious directories we don't need to search.
620 #we also don't look in folders that end in 'hide', unless it was specified on the command line
621             if (   -d $cur
622                 && $cur !~ /src$/
623                 && $cur !~ /bin$/
624                 && $cur !~ /include$/
625                 && ( $cur !~ /hide$/ || $loops <= 1 ) )
626             {
627                 my @list = `ls $cur`;
628                 foreach $item (@list) {
629
630           #we have a directory in the list, so we're going to need to loop again
631                     $clean = 0;
632                     chomp($item);
633                     if ( $cur eq "." ) {
634                         push( @newfiles, "$item" );
635                     }
636                     else {
637                         push( @newfiles, "$cur/$item" );
638                     }
639                 }
640             }
641             elsif ($cur =~ /$ext$/
642                 && $cur !~ /.step[\d]+./
643                 && $cur !~ /.opt[\d]+./ )
644             {
```

```
645
646                 #turn all // in file paths to just one /
647                 $cur =~ s/\/\//\//;
648                 push( @newfiles, $cur );
649             }
650         }
651         @$files = @newfiles;
652
653         if ( $loops > 8 ) {
654             print "Stopping recursion at $loops.\n";
655         }
656     }
657 }
658 1;
```

Bibliography

[1] C. J. Umrigar, M. P. Nightingale, and K. J. Runge. A diffusion monte carlo algorithm with very small time-step errors. *The Journal of Chemical Physics*, 99(4):2865–2890, 1993.

[2] P. J. Reynolds, D. M. Ceperley, B. J. Alder, and W. A. Lester. Fixed-node Quantum Monte Carlo for molecules. *Journal of Chemical Physics*, 77(11):5593–5603, 1982.

[3] D. M. Ceperley and B. J. Alder. Quantum Monte Carlo. *Science*, 231(4738):555–560, 1986.

[4] Amos G Anderson, Dan Fisher, Mike Feldmann, and David R Kent. Qmcbeaver, 2009. URL: `http://qmcbeaver.sourceforge.net`.

[5] M. N. Ringnalda, J.-M. Langlois, R. B. Murphy, B. H. Greeley, C. Cortis, T. V. Russo, B. Marten, R. E. Donnelly, Jr., W. T. Pollard, Y. Cao, R. P. Muller, D. T. Mainz, J. R. Wright, G. H. Miller, W. A. Goddard III, and R. A. Friesner. Jaguar v7.5, 2008.

[6] Michael W. Schmidt, Kim K. Baldridge, Jerry A. Boatz, Steven T. Elbert, Mark S. Gordon, Jan H. Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A. Nguyen, Shujun Su, Theresa L. Windus, Michel Dupuis, and John A. Montgomery Jr. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.

[7] Brett M. Bode and Mark S. Gordon. Macmolplt: a graphical user interface for gamess. *Journal of Molecular Graphics and Modelling*, 16(3):133 – 138, 1998.

[8] Matt Pharr, editor. *GPU Gems 2*. Addison-Wesley, 2005.

[9] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proc. of ACM/IEEE Superc. Conf.*, page 47, 2004.

[10] J. Georgii and R. Westermann. A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics*, 30(3), 2006.

[11] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision fem simulations with gpus. In *Proc. of ASIM 2005*, pages 139–144, 2005.

[12] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. of ACM/IEEE Superc. Conf.*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[13] nVidia. Compute unified device architecture. http://developer.nvidia.com/cuda.

[14] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *Proc. of ACM/IEEE Superc. Conf.*, Washington, DC, USA, 2006. IEEE Computer Society.

[15] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. *PACT'05*, pages 185–196, 2005.

[16] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 133–137, 2004.

[17] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[18] J. D. Hall, N. A. Carr, and J. C. Hart. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois, 2003.

[19] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[20] Á. Moravánszky. Dense matrix algebra on the gpu, 2003. NovodeX AG.

[21] S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware, 2001.

[22] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 102–111, 2003.

[23] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. In *Proc. of ACM/IEEE Superc. Conf.*, volume 52, pages 654–663, 2005.

[24] P. A. Heng J. Q. Wang, T. T. Wong and C. S. Leung. Discrete wavelet transform on gpu. In *ACM Workshop on GPGPU*, 2004.

[25] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 112–119, 2003.

[26] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[27] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *Course Notes*, 2004.

[28] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *Proc. of ACM SIGGRAPH*, 2004.

[29] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh.* AK Peters, Ltd., 2004.

[30] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087, 1953.

[31] C. J. Umrigar, M. P. Nightingale, and K. J. Runge. A diffusion Monte Carlo algorithm with very small time-step errors. *Journal of Chemical Physics*, 99(4):2865–2890, 1993.

[32] AJ Williamson, R.Q. Hood, and JC Grossman. Linear-Scaling Quantum Monte Carlo Calculations. *Physical Review Letters*, 87(24):246406, 2001.

[33] A. Aspuru-Guzik, R. Salomon-Ferrer, B. Austin, and W.A. Lester. A sparse algorithm for the evaluation of the local energy in quantum Monte Carlo. *Journal of Computational Chemistry*, 26(7):708–715, 2005.

[34] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[35] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. of ACM/IEEE Superc. Conf.*, 1998.

[36] S. Huzinaga, E. Miyoshi, and M. Sekiya. A method of generating an effective orbital set for configuration interaction calculations. *The Journal of Chemical Physics*, 100(2):1435–1439, 1994.

[37] C. Schwartz. Experiment and theory in computations of the he atom ground state. *Int. J. of Mod. Phys.*, 15(4):877–888, 2006.

[38] David E. Woon and Jr. Thom H. Dunning. Gaussian basis sets for use in correlated molecular calculations. v. core-valence basis sets for boron through neon. *The Journal of Chemical Physics*, 103(11):4572–4585, 1995.

[39] Attila G. Csaszar, Wesley D. Allen, and Henry F. Schaefer III. In pursuit of the ab initio limit for conformational energy prototypes. *Journal of Chemical Physics*, 108(23):9751–9764, 1998.

[40] K. E. Hillesland and A. Lastra. Gpu floating-point paranoia. In GP^2, pages C–8, 2004.

[41] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Com. of the ACM*, 8(1):40–40, 1965.

[42] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[43] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley, 1997.

[44] John Michael McNamee. A comparison of methods for accurate summation. *SIGSAM Bull.*, 38(1):1–7, 2004.

[45] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[46] A. Aspuru-Guzik, O. El Akramine, J.C. Grossman, and W.A. Lester Jr. Quantum Monte Carlo for electronic excitations of free-base porphyrin. *Journal of Chemical Physics*, 120(7):3049–3050, 2004.

[47] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[48] Frank W. Bobrowicz and William A. Goddard, III. The self-consistent field equations for generalized valence bond and open-shell Hartree-Fock wave functions. In Henry F. Schaefer, III, editor, *Methods of Electronic Structure Theory*, volume 3 of *Modern Theoretical Chemistry*, page 79. Plenum Press, New York, 1977.

[49] N. D. Drummond, M. D. Towler, and R. J. Needs. Jastrow correlation factor for atoms, molecules, and solids. *Phys. Rev. B*, 70(23):235119, Dec 2004.

[50] Julien Toulouse and C. J. Umrigar. Optimization of quantum monte carlo wave functions by energy minimization. *The Journal of Chemical Physics*, 126(8):084102, 2007.

[51] Roland Assaraf, Michel Caffarel, and Anatole Khelif. Diffusion monte carlo methods with a fixed number of walkers. *Phys. Rev. E*, 61(4):4566–4575, Apr 2000.

[52] Michael W. Schmidt and Mark S. Gordon. The construction and interpretation of mcscf wavefunctions. *Annual Review of Physical Chemistry*, 49(1):233–266, 1998.

[53] Jean-Marc Langlois, Terumasa Yamasaki, Richard P. Muller, and William A. III Goddard. Rule-based trial wave functions for generalized valence bond theory. *The Journal of Physical Chemistry*, 98(51):13498–13505, 1994.

[54] Robert B. Murphy, Richard A. Friesner, Murco N. Ringnalda, and William A. Goddard III. Pseudospectral contracted configuration interaction from a generalized valence bond reference. *The Journal of Chemical Physics*, 101(4):2986–2994, 1994.

[55] Amos G. Anderson, William A. Goddard III, and Peter Schrder. Quantum monte carlo on graphical processing units. *Computer Physics Communications*, 177(3):298 – 306, 2007.

[56] David Feller. The role of databases in support of computational chemistry calculations. *Journal of Computational Chemistry*, 17(13):1571–1586, 1996.

[57] Karen L. Schuchardt, Brett T. Didier, Todd Elsethagen, Lisong Sun, Vidhya Gurumoorthi, Jared Chase, Jun Li, and Theresa L. Windus. Basis set exchange: A community database for computational sciences. *Journal of Chemical Information and Modeling*, 47(3):1045–1052, 2007.

[58] Kirk A. Peterson and Jr. Thom H. Dunning. Accurate correlation consistent basis sets for molecular core–valence correlation effects: The second row atoms al–ar, and the first row atoms b–ne revisited. *The Journal of Chemical Physics*, 117(23):10548–10560, 2002.

[59] Piotr Piecuch, Stanislaw A. Kucharski, Karol Kowalski, and Monika Musial. Efficient computer implementation of the renormalized coupled-cluster methods: The r-ccsd[t], r-ccsd(t), cr-ccsd[t], and cr-ccsd(t) approaches. *Computer Physics Communications*, 149(2):71 – 96, 2002.

[60] Yan Zhao and Donald G. Truhlar. Density functionals with broad applicability in chemistry. *Accounts of Chemical Research*, 41(2):157–167, 2008.

[61] Daniel R. Fisher, David R. Kent IV, Michael T. Feldmann, and William A. Goddard III. An optimized initialization algorithm to ensure accuracy in quantum monte carlo calculations. *Journal of Computational Chemistry*, 29(14):2335–2343, 2008.

[62] A. Ma, M. D. Towler, N. D. Drummond, and R. J. Needs. Scheme for adding electron–nucleus cusps to gaussian orbitals. *The Journal of Chemical Physics*, 122(22):224322, 2005.

[63] Michael T. Feldmann, Julian C. Cummings, David R. Kent IV, Richard P. Muller, and William A. Goddard III. Manager-worker-based model for the parallelization of quantum monte carlo on heterogeneous and homogeneous networks. *Journal of Computational Chemistry*, 29(1):8–16, 2008.

[64] David R. Kent IV, Richard P. Muller, Amos G. Anderson, William A. Goddard III, and Michael T. Feldmann. Efficient algorithm for “on-the-fly” error analysis of local or

distributed serially correlated data. *Journal of Computational Chemistry*, 28(14):2309–2316, 2007.

[65] Per Jensen and P. R. Bunker. The potential surface and stretching frequencies of $\tilde{X}\ ^3b_1$ methylene (ch_2) determined from experiment using the morse oscillator-rigid bender internal dynamics hamiltonian. *The Journal of Chemical Physics*, 89(3):1327–1332, 1988.

[66] J. P. Gu, G. Hirsch, R. J. Buenker, M. Brumm, G. Osmann, P. R. Bunker, and P. Jensen. A theoretical study of the absorption spectrum of singlet ch2. *Journal of Molecular Structure*, 517-518:247 – 264, 2000.

[67] Tibor Furtenbacher, Gbor Czak, Brian T. Sutcliffe, Attila G. Csszr, and Viktor Szalay. The methylene saga continues: Stretching fundamentals and zero-point energy of ch2. *Journal of Molecular Structure*, 780-781:283 – 294, 2006. Spectroscopic and Theoretical Determination of Molecular Properties - Spectroscopic and Theoretical Determination of Molecular Properties: - A Collection of Invited Papers in Honor of Dr. Jean Demaison.

[68] Hrvoje Petek, David J. Nesbitt, David C. Darwin, Peter R. Ogilby, C. Bradley Moore, and D. A. Ramsay. Analysis of ch_2 $\tilde{a}\ ^1a_1$ (1,0,0) and (0,0,1) coriolis-coupled states, $\tilde{a}\ ^1a_1$–$\tilde{X}\ ^3b_1$ spin–orbit coupling, and the equilibrium structure of ch_2 $\tilde{a}\ ^1a_1$ state. *The Journal of Chemical Physics*, 91(11):6566–6578, 1989.

[69] A Kalemos, TH Dunning, A Mavridis, and JF Harrison. CH2 revisited. *CANADIAN JOURNAL OF CHEMISTRY-REVUE CANADIENNE DE CHIMIE*, 82(6):684–693, JUN 2004.

[70] W. M. Flicker, O. A. Mosher, and A. Kuppermann. Singlet –¿ triplet transitions in methyl-substituted ethylenes. *Chemical Physics Letters*, 36(1):56 – 60, 1975.

[71] D. F. Evans. 347. magnetic perturbation of singlettriplet transitions. part iv. unsaturated compounds. *Journal of the Chemical Society*, pages 1735–1745, 1960.

[72] O. El Akramine, A. C. Kollias, and Jr. W. A. Lester. Quantum monte carlo study of singlet–triplet transition in ethylene. *The Journal of Chemical Physics*, 119(3):1483–1488, 2003.

[73] Friedemann Schautz and Claudia Filippi. Optimized jastrow–slater wave functions for ground and excited states: Application to the lowest states of ethene. *The Journal of Chemical Physics*, 120(23):10931–10941, 2004.

[74] Minh Tho Nguyen, Myrna H. Matus, William A. Lester, and David A. Dixon. Heats of formation of triplet ethylene, ethylidene, and acetylene. *The Journal of Physical Chemistry A*, 112(10):2082–2087, 2008.

[75] Bernhard Gemein and Sigrid D. Peyerimhoff. Radiationless transitions between the first excited triplet state and the singlet ground state in ethylene: A theoretical study. *The Journal of Physical Chemistry*, 100(50):19257–19267, 1996.

[76] Shlomit Jacobi and Roi Baer. The well-tempered auxiliary-field monte carlo. *The Journal of Chemical Physics*, 120(1):43–50, 2004.

[77] R McDiarmid. On the electronic spectra of small linear polyenes. *Advances in Chemical Physics*, pages 177–214, 1999.

[78] Weston Thatcher Borden and Ernest R. Davidson. The importance of including dynamic electron correlation in ab initio calculations. *Accounts of Chemical Research*, 29(2):67–75, 1996.

[79] Fei Qi, Osman Sorkhabi, and Arthur G. Suits. Evidence of triplet ethylene produced from photodissociation of ethylene sulfide. *The Journal of Chemical Physics*, 112(24):10707–10710, 2000.

[80] L.V. Gurvich, I. V. Veyts, and C. B Alcock. *Thermodynamic Properties of Individual Substances*. Hemisphere Pub. Co., 4 edition, 1989.

[81] K.N. Frenkel, M; Marsh, R.C. Wilhoit, G.J. Kabo, and G.N. Roganov. Thermodynamics of organic compounds in the gas state. *International Journal of Chemical Kinetics*, 28(7):553–554, 1996.

[82] David Feller and David A. Dixon. Extended benchmark studies of coupled cluster theory through triple excitations. *The Journal of Chemical Physics*, 115(8):3484–3496, 2001.

[83] C. J. Umrigar, K. G. Wilson, and J. W. Wilkins. Optimized trial wave-functions for quantum monte carlo calculations. *Physical Review Letters*, 60(17):1719–1722, 1988.

[84] Julien Toulouse and C. J. Umrigar. Full optimization of jastrow–slater wave functions with application to the first-row atoms and homonuclear diatomic molecules. *The Journal of Chemical Physics*, 128(17):174101, 2008.

[85] C. J. Huang, C. J. Umrigar, and M. P. Nightingale. Accuracy of electronic wave functions in quantum monte carlo: The effect of high-order correlations. *Journal of Chemical Physics*, 107(8):3007–3013, 1997.

[86] V. E. Bondybey. Electronic structure and bonding of be2. *Chemical Physics Letters*, 109(5):436 – 441, 1984.

[87] J. A. W. Harkless and K. K. Irikura. Multi-determinant trial functions in the determination of the dissociation energy of the beryllium dimer: Quantum monte carlo study. *International Journal of Quantum Chemistry*, 106(11):2373–2378, 2006.

[88] G. A. Petersson and William A. Shirley. The beryllium dimer potential. *Chemical Physics Letters*, 160(5-6):494 – 501, 1989.

[89] Robert J. Gdanitz. Accurately solving the electronic schrdinger equation of atoms and molecules using explicitly correlated (r12-)mr-ci.: The ground state of beryllium dimer (be2). *Chemical Physics Letters*, 312(5-6):578 – 584, 1999.

[90] Jan M. L. Martin. The ground-state spectroscopic constants of be2 revisited. *Chemical Physics Letters*, 303(3-4):399 – 407, 1999.

[91] Vl.G. Tyuterev, S. Tashkun, P. Jensen, A. Barbe, and T. Cours. Determination of the effective ground state potential energy function of ozone from high-resolution infrared spectra. *Journal of Molecular Spectroscopy*, 198(1):57 – 76, 1999.

[92] Apostolos Kalemos and Aristides Mavridis. Electronic structure and bonding of ozone. *The Journal of Chemical Physics*, 129(5):054312, 2008.

[93] Don W. Arnold, Cangshan Xu, Eun H. Kim, and Daniel M. Neumark. Study of low-lying electronic states of ozone by anion photoelectron spectroscopy of o[sup -][sub 3]. *The Journal of Chemical Physics*, 101(2):912–922, 1994.

[94] Sabine F. Deppe, Uwe Wachsmuth, Bernd Abel, Martina Bittererová, Sergy Yu. Grebenshchikov, Rüdiger Siebert, and Reinhard Schinke. Resonance spectrum and dissociation dynamics of ozone in the [sup 3]b[sub 2] electronically excited state: Experiment and theory. *The Journal of Chemical Physics*, 121(11):5191–5200, 2004.

[95] J. Berkowitz, J. P. Greene, H. Cho, and B. Ruščić. Photoionization mass spectrometric studies of sih[sub n] (n=1–4). *The Journal of Chemical Physics*, 86(3):1235–1248, 1987.

[96] Jeffrey C. Grossman. Benchmark quantum monte carlo calculations. *The Journal of Chemical Physics*, 117(4):1434–1440, 2002.

[97] James L. Blue, Isabel Beichl, and Francis Sullivan. Faster monte carlo simulations. *Phys. Rev. E*, 51(2):R867–R868, Feb 1995.

[98] Abhijit Chatterjee and Dionisios G. Vlachos. An overview of spatial microscopic and accelerated kinetic monte carlo methods. *Journal of Computer-Aided Materials Design*, 14(2):253–308, 2007.