Quantum Monte Carlo: Quest to Get Bigger, Faster, and Cheaper

Thesis by
Michael Todd Feldmann

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

California Institute of Technology
Pasadena, CA
2002
(Defended May 20, 2002)

Preface

This dissertation describes work undertaken between August 1998 and May 2002 in the Materials Simulation Center at the California Institute of Technology under the supervision of Professor William A. Goddard III. This dissertation describes a major portion of the work I did with Dr. Goddard, Dr. Richard P. Muller, and David R. Kent IV developing efficient methods for high accuracy methods for quantum mechanics.

Acknowledgements

The research in this dissertation was the result of many factors coming together in a positive way. It is my pleasure to acknowledge those who have supported me during this work.

I would first like to thank my advisor, Dr. William A. Goddard III. He sold me on the idea of Caltech as a place where exciting science is done. He was correct. The knowledge I take away from the last four years of seeing the *research machine* in motion is priceless.

During this same period Dr. Richard P. Muller acted both as a scientific mentor and motivator. When research was going poorly, Dr. Muller was always the one to come to the table with a smile on his face and get me excited about the work again. I truly thank Dr. Muller for all of his positive words and creative insight.

David "Chip" R. Kent IV was a critical member in making this research become a reality. His insights and *strong will* kept this work on focus and truly made the QMcBeaver software a success. He is one of the brightest people I know and I feel privileged to have worked with him. I have no doubt that the QMcBeaver project will be led by good hands in the future.

The Krell Institute ran the Department of Energy Computational Science Graduate Fellowship (DOE-CSGF) which supported me during my studies at Caltech. Their vision for the future of computional science just about put me in tears as I satisfied their rigorous (and I often thought painful) course requirements. In hindsight, however, accepting the DOE-CSGF has been one of the most pivotal decisions in my young career in computational science. It has changed my focus and vision for the future of this field. I am grateful to both the Krell Institute and the Department of Energy for supporting this excellent program.

The R. J. McElroy foundation also supported my research during the first three years of my graduate work. Their goal of getting rural Iowa liberal arts students on their way to a Ph.D. is a nobel one and I feel privileged to have been selected by them to represent young Iowa scholars.

I saved the most important for last. I would like to thank my wife, Rachel, and our families for their unconditional support. Rachel has learned what it means to be married to a Caltech graduate student who works insane hours, yet she has always been supportive of me completing this doctoral degree. I would also like to thank my family for their guidance and giving me the opportunity to pursue these interests. Without the support of these important people I would not be where I am today.

Abstract

We reexamine some fundamental Quantum Monte Carlo (QMC) algorithms with the goal of making QMC more mainstream and efficient. Two major themes exist: (1) Make QMC faster and cheaper, and (2) Make QMC more robust and easier to use. A fast "on-the-fly" algorithm to extract uncorrelated estimators from serially correlated data on a huge network is presented, DDDA. A very efficient manager-worker algorithm for QMC parallelization is presented, QMC-MW. Reduced expense VMC optimization procedure is presented to better guess initial Jastrow parameter sets for hydrocarbons, GJ. I also examine the formation and decomposition of aminomethanol using a variety of methods including a test of the hydrocarbon GJ set on these oxygen- and nitrogen-containing systems. The QMC program suite QMcBeaver is available from the authors in its entirety while a user's and developer's manual is attached as supplementary material.

Contents

1 Summary **1**

1.1 Future of Supercomputing 3

1.2 Introduction to Quantum Monte Carlo 3

1.3 Efficient Algorithm for "On-the-fly" Error Analysis of Local or Distributed Serially Correlated Data 4

1.4 Manager–Worker-Based Model for the Parallelization of Quantum Monte Carlo on Heterogeneous and Homogeneous Networks 5

1.5 Generic Jastrow Functions for Quantum Monte Carlo Calculations on Hydrocarbons 6

1.6 Aminomethanol Water Elimination: Theoretical Examination 6

1.7 QMcBeaver 7

2 Future of Supercomputing **8**

2.1 Current State of Supercomputing 8

2.2 What is Coming Next? 9

2.3 What Should We Do? 10

2.3.1 How Should We Design Hardware? 10

2.3.2 How Should We Design Software? 11

2.3.3 How Coupled Should Software and Hardware Become? 11

2.4 Mission of This Work 12

3 Introduction to Quantum Monte Carlo 13

3.1 Introduction 13

3.2 Theory 15

3.2.1 Variational Quantum Monte Carlo 16

3.2.2 Diffusion Quantum Monte Carlo 18

3.3 Conclusion 19

4 Efficient Algorithm for "On-the-fly" Error Analysis of Local or Distributed Serially Correlated Data 20

4.1 Introduction 20

4.2 Motivation and Background 21

4.3 Theory 24

4.3.1 Computational Cost of Flyvbjerg-Peterson Algorithm . 26

4.3.2 Dynamic Distributable Decorrelation Algorithm (DDDA) 27

4.3.3 Computational Cost of DDDA 29

4.4 Computational Experiments 31

4.4.1 Variational QMC on One-Dimensional Particle-in-a-Box 31

4.4.2 Finite All-Electron Variational QMC on RDX 35

4.5 Conclusions 40

4.6 Statistic Class Pseudocode 42

4.6.1 Pseudocode for *Statistic.initialize*() 42

4.6.2 Pseudocode for *Statistic.add_Data*(*new_sample*) . . . 44

4.6.3 Pseudocode for *Statistic.addition*(A, B) 44

4.7 Decorrelation Class Pseudocode 45

4.7.1 Pseudocode for *Decorrelation.initialize*() 45

4.7.2 Pseudocode for *Decorrelation.add_Data*(*new_sample*) 45

4.7.3 Pseudocode for *Decorrelation.addition*(A, B) 46

4.8 Simple Example Calculation Pseudocode 50

5 Manager–Worker-Based Model for the Parallelization of Quantum Monte Carlo on Heterogeneous and Homogeneous Networks 52

5.1 Introduction 52

5.2 Theory 54

5.2.1 Pure Iterative Parallelization Algorithm 56

5.2.2 Manager–Worker-Parallelization Algorithm 58

5.2.3 Initialization Catastrophe 61

5.3 Experiment 62

5.3.1 Experiment: Varying Levels of Heterogeneity 63

5.3.2 Experiment: Heterogeneous Network Size 66

5.3.3 Experiment: Homogeneous Network 69

5.3.4 Experiment: Initialization Catastrophe 73

5.4 Conclusion 75

5.5 Pure Iterative Algorithm (QMC-PI) 76

5.6 Manager–Worker Algorithm (QMC-MW) 77

6 Generic Jastrow Functions for Quantum Monte Carlo Calculations on Hydrocarbons 78

6.1 Introduction 78

6.2 Theory 80

6.2.1 Variational Quantum Monte Carlo 81

6.2.2 Diffusion Quantum Monte Carlo 82
6.3 Experiment 83
6.3.1 Motivate Generic Jastrow for Hydrocarbons 83
6.3.2 Experiment: Hydrocarbons Test Set 84
6.3.3 Generic Jastrow for DMC 86
6.3.4 Test Case: 10-Annulene 89
6.4 Conclusion 90

7 Aminomethanol Water Elimination: Theoretical Examination 92

7.1 Introduction 92
7.2 Theory 95
7.3 Experiment 96
7.3.1 Experiment Setup 96
7.3.2 Data 98
7.4 Conclusion 99

8 QMcBeaver 107

List of Figures

4.1 The energy expectation value standard deviation, evaluated with Eq. 4.6, as a function of block size for a VMC "particle-in-a-box" calculation using Method 1 to generate uncorrelated data points. The Flyvbjerg-Peterson algorithm and DDDA yield exactly the same results. The error bars represent one standard deviation in the calculated standard deviation estimator. 34

4.2 The energy expectation value standard deviation, evaluated with Eq. 4.6, versus block size for a VMC "particle-in-a-box" calculation using Method 2 to generate correlated data points. The Flyvbjerg-Peterson algorithm and DDDA yield exactly the same results. The error bars represent one standard deviation in the calculated standard deviation estimator. 35

4.3 The RDX molecule, cyclic $[CH_2\text{-}N(NO_2)]_3$. 36

4.4 The evolution of the energy-standard-deviation-estimator for the ground state of RDX with block size. Shown here are the results for five cases with 62122, 2137179, 6283647, 14566309, and 31163746 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^8 = 256$ is sufficient for this calculation. . 40

4.5 The evolution of the energy-standard-deviation-estimator for the transition state for N-NO2 bond dissociation in RDX with block size. Shown here are the results for five cases with 72899, 1113737, 5284068, 13601739, and 30176694 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^8 = 256$ is sufficient for this calculation. 41

4.6 The evolution with block size of the energy-standard-deviation-estimator for the transition state for concerted symmetric ring decomposition of RDX. Shown here are the results for five cases with 38848, 2110471, 6260482, 14545368, and 31126145 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^{13} = 8192$ is sufficient for this calculation. 42

4.7 The evolution of the standard-deviation-estimate for the energy of the three states of RDX whose results were shown in Fig. 4.4, 4.5, and 4.6. A block size of 2^8 was used for the ground state and the transition state for N-NO2 dissociation while a block size of 2^{13} was used for the symmetric concerted transition state. The energies are in Hartree (1 Hartree = 27.2116 eV). 43

5.1 Time required to complete an 8 processor variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking. The theoretical optimal performance for a given configuration of processors is provided by the curve. 64

5.2 Number of variational QMC steps completed during an 8 processor calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) parallelization algorithms. The pure iterative algorithm always calculates the same number of steps, but the manager–worker algorithm dynamically determines how many steps to take. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking. 66

5.3 Percentage of total calculation time devoted to each component in the pure iterative parallelization algorithm (QMC-PI) during an 8 processor variational QMC calculation of Ne. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking. 67

5.4 Percentage of total calculation time devoted to each component in the manager–worker-parallelization algorithm (QMC-MW) during an 8 processor variational QMC calculation of Ne. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking. 68

5.5 Wall time required to complete a variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms on a heterogeneous linux cluster. The theoretical optimal performance for a given configuration of processors is provided by the line. 69

5.6 Wall time required to complete a variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms on the ASCI Blue Pacific homogeneous supercomputer. The theoretical optimal performance for a given configuration of processors is provided by the line. 70

5.7 Wall time in nonpropagation and non-initialization overhead expenses for QMC-PI and QMC-MW on ASCI Blue Pacific. 71

5.8 Ratio of wall time for QMC-MW/QMC-PI on ASCI Blue Pacific. . . 72

5.9 Efficiency of a variational QMC calculation of RDX as a function of the number of processors used. The calculations were performed using the manager–worker-parallelization algorithm (QMC-MW) on the ASCI-Blue Mountain supercomputer, which has 250 MHz MIPS 10000 processors connected by HIPPI networking. A similar result is produced by the Pure Iterative parallelization algorithm. The data is fit to $\epsilon(N_{Processors}) = a/(a + N_{Processors})$ with $a = 104.203$. 74

6.1 Correlation energy (Hartree) recovered divided by total nuclear charge. 86

6.2 Reduction of the QMC variance for a wavefunction containing a Generic Jastrow compared to a Hartree-Fock wavefunction. 87

6.3 Generic Jastrow correlation functions. $b_{\uparrow,\downarrow} = 3.0$ 88

6.4 Convergence (σ^2 in Hartree^2 units) of methane DMC runs over time for various wavefunctions 89

6.5 Convergence (σ^2 in Hartree^2 units) of acetylene DMC runs over time for various wavefunctions 90

7.1 Mechanism for reaction of formaldehyde and ammonia. 94

7.2 Full mechanism of aminomethanol formation from NH_3 and CH_2O and decomposition to CH_2NH and H_2O. Geometries determined with Jaguar [1] b3lyp/cc-pVTZ. 97

List of Tables

4.1 Comparison of computational costs. Here N is the number of data points analyzed. In evaluating the costs, N is assumed to be a multiple of two. This represents the worst-case scenario. 30

4.2 Total energies (Hartree) for the various calculations on RDX. The HF and DFT [2] results were obtained from Jaguar 4.1 with the 6-31G** basis set [1, 3, 4, 5, 6, 7, 8]. Variational Quantum Monte Carlo based on 3×10^7 points. 39

6.1 Absolute energies (Hartree) for various conformations of 10-annulene methods with and without explicit electron correlation from the Generic Jastrow (basis: cc-pVDZ). 91

7.1 Absolute energies (Hartree) for various methods (basis: cc-pVTZ). *Jaguar, **Molpro, ***QMcBeaver 98

7.2 Absolute energies (Hartree) for various methods (basis: cc-pVTZ++/aug-cc-pVTZ). *Jaguar, **Molpro, ***QMcBeaver 99

7.3 Relative free energies ΔG (kcal/mol) for various methods with cc-pVTZ basis with Jaguar b3lyp/cc-pVTZ(-f) zero point and thermochemical corrections at 2.63E-5 atm and 300K. *Jaguar, **Molpro, ***QMcBeaver 100

7.4 Relative free energies ΔG (kcal/mol) for various methods with cc-pVTZ++/aug-cc-pVTZ basis with Jaguar b3lyp/cc-pVTZ(-f) zero point and thermochemical corrections at 2.63E-5 atm and 300K. *Jaguar, **Molpro, ***QMcBeaver 101

7.5 Relative free energies ΔG (kcal/mol) at 2.63E-5 atm for various temperatures with zero point and thermochemical corrections from Jaguar (b3lyp/cc-pVTZ(-f)) on energetics from MolPro (QCI(T)/cc-pVTZ). . 102

7.6 Relative free energies ΔG (kcal/mol) at 1.32E-8 atm for various temperatures with zero point and thermochemical corrections from Jaguar (b3lyp/cc-pVTZ(-f)) on energetics from MolPro (QCI(T)/cc-pVTZ). . 103

7.7 Geometry for NH_3. 103

7.8 Geometry for CH_2O. 104

7.9 Geometry for $T^{\dagger}_{NH_3+CH_2O}$. 104

7.10 Geometry for $CH_2(OH)NH_2$. 105

7.11 Geometry for $T^{\dagger}_{H_2O+CH_2NH}$. 105

7.12 Geometry for H_2O. 106

7.13 Geometry for CH_2NH. 106

Chapter 1

Summary

Quantum Monte Carlo (QMC) is a very exciting method for calculating electronic structure in chemical systems. QMC can achieve very high accuracy and has several other computationally attractive features. When we first started this type of work, we asked some very basic questions about QMC.

- Why is QMC interesting to others?
- Why is QMC not in more common use?
- What does someone need to use QMC? (software, computational resources, QMC expertise?)
- Is QMC the method of future in computational chemistry and material science?
- What have others done in the field?
- What are others currently focussed on?
- What can I do to help the effort of making QMC practical?

These seem like fairly simple questions yet they led to a surprising richness in research. Currently, QMC is a very exciting area of research with many great minds working hard to make QMC fast, robust, and easy to use (Chapter 3). Many interesting fronts exist in QMC, including higher accuracy methods, released node QMC, QMC calculation of forces, and better single processor computational complexity of QMC to just name a few. To provide the reader with a (by no means exhaustive) list of good references to get a feel for current methods and trends in QMC, many references are included in the bibliography though not explicitely referenced herein.

QMC is a high-accuracy electron structure method, which scales roughly as $O(n^3)$ while competing methods scale as $O(n^6)$ to $O(n!)$ [20]. In addition, QMC is very easy to parallelize, which means you can further reduce the time it takes to compute. Unfortunately, the prefactor on QMC is very large, so it takes too much time to do except on supercomputers for the large majority of interesting chemical and materials problems.

This is where this work begins. QMC is very expensive both in the user's time and in the computational resources it takes to accomplish a QMC job. The goal of this work is to make QMC run on cheaper machines and do it in less time while improving the ease of use for the end user. All of these will bring QMC one step closer to being a commonly used method in computational electronic structure studies.

In presenting this work, I have chosen to make each chapter as self-contained as possible. I hope the reader can read any chapter with only minimal reference to other chapters. This, of course, comes at the expense of being somewhat redundant in the introductory sections of each chapter but I feel most readers will appreciate this as they implement or write their own

QMC related software.

1.1 Future of Supercomputing

Current trends and a projection of what the future of supercomputing will become are examined. Computational modeling is becoming a very important part of basic research and the needs of researchers must be met with economical computing solutions. There are two major sides of this issue. The first is the building of computational infrastructure which provides the most computing power per unit of expense. The second is the development of algorithms and software which can effectively utilize these resources. We also explore the correlated nature of these two points.

1.2 Introduction to Quantum Monte Carlo

A brief background on the popular forms of Quantum Monte Carlo (QMC) are given. The first of these is variation QMC (VMC), and a second popular method is diffusion or fixed node QMC (DMC). VMC and DMC are the two major forms of QMC employed by research scientists and form the basis for other forms of QMC. Any improvements one can make on these base methods will likely have far reaching impact.

1.3 Efficient Algorithm for "On-the-fly" Error Analysis of Local or Distributed Serially Correlated Data

A significant impediment to applying Monte Carlo methods to the computation of physically important systems is the efficient decorrelation of data generated by Markov chains "on-the-fly" and in parallel for the extremely large amount of sampling required to achieve convergence of a given estimator. We describe the Dynamic Distributable Decorrelation Algorithm (DDDA) that eliminates this difficulty by efficiently calculating the true statistical error of an expectation value obtained from serially correlated data. DDDA is an improvement on the Flyvbjerg-Peterson renormalization group method [49], but allowing the statistical error to be evaluated "on-the-fly." This "on-the-fly" determination of statistical quantities allows dynamic termination of Monte Carlo calculations once a specified level of convergence is attained. This is highly desirable, for example, for Quantum Monte Carlo (QMC) calculations where the desired precision might require days or months to compute, but cannot be estimated prior to the calculation.

Furthermore, DDDA allows a very efficient parallel implementation. For the example of predicting the activation energy for decomposition of RDX discussed herein, we estimate that $N_G = 10^{12}$ global data points are required to attain 0.1 kcal/mol precision in the calculated VMC energy estimators. Thus with $M = 10^3$ processing units, the original algorithm requires local processor storage scaling as $O(N)$, $N \approx \frac{N_G}{M}$ or roughly 10^9 numbers which may be difficult to accommodate with local storage and is often very difficult to transfer efficiently between processors. The local processor storage requirement for DDDA scales as $O(log_2(N))$ or roughly 120 doubles for $N_G = 10^{12}$ and $M = 10^3$ with an average update computational complexity for each new sample of $O(1)$. This small amount of data can easily be communicated and combined with data from other processors, making parallel processing very efficient.

1.4 Manager–Worker-Based Model for the Parallelization of Quantum Monte Carlo on Heterogeneous and Homogeneous Networks

A manager–worker-based parallelization algorithm for Quantum Monte Carlo (QMC-MW) is presented and compared to the pure iterative parallelization algorithm, which is in common use. The new manager–worker algorithm performs automatic load balancing, allowing it to perform near the theoretical maximal speed on heterogeneous parallel computers. Furthermore, the new algorithm performs as well as the pure iterative algorithm on homogeneous parallel computers. When combined with the Dynamic Distributable Decorrelation Algorithm (DDDA) [50], the new manager–worker algorithm allows QMC calculations to be terminated when a desired level of convergence is obtained and not when a given number of steps are performed as is the common practice. Additionally, a derivation and experimental verification are given to show that standard QMC implementations are not "perfectly parallel" as is often claimed.

1.5 Generic Jastrow Functions for Quantum Monte Carlo Calculations on Hydrocarbons

A Generic Jastrow (GJ) is examined that can be used for all-electron Quantum Monte Carlo across a large range of small hydrocarbons. This simple GJ captures some of the missing electron correlation of Hartree Fock (HF) theory for Variational Quantum Monte Carlo (VMC) while reducing the E_{Local} variance a substantial amount. This implies Diffusion Quantum Monte Carlo (DMC) may be accomplished with greatly reduced VMC Jastrow optimization expense.

1.6 Aminomethanol Water Elimination: Theoretical Examination

The mechanism for the formation of hexamethylenetetraamine predicts the formation of aminomethanol from the addition of ammonia to formaldehyde. This molecule subsequently undergoes water loss to form methanimine. Aminomethanol is the predicted precursor to interstellar glycine, and is therefore of great interest for laboratory spectroscopic study, which would serve as the basis for observational searches. The height of the water loss barrier is therefore useful in determination of an appropriate experimental approach for spectroscopic characterization of aminomethanol. We have determined the height of this barrier to be 55 kcal/mol at ambient temperatures using QCI(T)/cc-pVTZ. Therefore, spectroscopic characterization of this molecule should be straightforward under typical laboratory conditions.

1.7 QMcBeaver

The software package developed is called QMcBeaver. The user's and developer's manual is added as supplementary material. This is useful for both those developing QMcBeaver and those developing their own QMC package.

Chapter 2

Future of Supercomputing

2.1 Current State of Supercomputing

A current trend in large scale supercomputing [51] is assembling "cheap supercomputers" with commodity components using a Beowolf-type framework. These clusters have proven to be very powerful for high-performance scientific computing applications [52]. Clusters can be constructed as homogeneous supercomputers if the hardware for each node is equivalent, or as heterogeneous supercomputers if various generations of hardware are included.

Another interesting development is the use of loosely coupled, distributed grids of computational resources [53] with components that can even be located in different geographic locations in the world. Such "grids" are upgraded by adding new nodes to the existing grid resulting in continuously upgradable supercomputers, which are inevitably heterogeneous.

2.2 What is Coming Next?

This is by far the most speculative portion of this work. At the same time, I fear this section will date this work worse than any other. *What is Coming Next?* is the question we all wish we knew to answer to. I do want to take a little time to map out what I believe will happen in computing.

The trend of going to larger parallelization is one that I believe is here to stay with the current technology in hardware. The essential parts of a current computer are very inexpensive. We, of course, have no idea of future processing unit technology, but with the current trends, large scale parallelization is here and will only get bigger.

Homogeneous supercomputers will become less and less dominant. Building a huge machine all at once is not generally the best economic model. The demand for a computational resource increases over time and this solution does not scale because the technology used in this machine will become obsolete in a very short amount of time.

Heterogeneous frameworks are the future of supercomputing. A heterogeneous framework is always expandable and the parts of the framework become noneconomical more slowly. Some parts of the framework will become obsolete since the hardware may not be worth the real estate it occupies and the electricity that powers the machine may cost more than the utility the machine gives to the user. We should also note that the current homogeneous machines fit into this model but only as smaller components of a larger heterogeneous framework.

These heterogeneous frameworks will become so large that it will become necessary for loosely coupled interconnects for the majority of the peer to peer communications. This means that algorithms built for very large frameworks need to make small enough tasks be accomplished on these relatively smaller tightly coupled machines or tasks must be able to run efficiently on very loosely coupled networks.

2.3 What Should We Do?

This is yet another speculative section on the proper course of computational science but there are some rather solid statements that can be made. The question *How should we design hardware/software?* is very interesting. The ultimate goal of doing a computation is to obtain a certain result with the least expense. The *certain result* is naturally the result from a given model. This is dependent on the software used yet the level of the calculation can be limited by the hardware. The *least expense* refers to both the expense of *renting* a computational resource and the amount of the user's time the calculation takes. The utility function used will be a combination of computational resource expense and user defined utility as a function of wall clock time to complete the task.

2.3.1 How Should We Design Hardware?

Clearly we want *the most cost effective solution* for building machines that are inexpensive, long lasting, and easy for software engineers to design software for. This is very difficult to accomplish. The cheapest scalable machines to build are heterogeneous frameworks, yet the easiest to build software for is tightly coupled homogeneous machines.

2.3.2 How Should We Design Software?

Software should be designed to run as efficiently as possible on a general framework of computers. This is also difficult to achieve. If we are to efficiently use a heterogeneous network of computers, we need to find algorithms which allow good load balancing. This often comes at the expense of performing worse than the theoretical optimum because of additional bookkeeping in keeping the network load balanced. Also, completely different algorithms may be the *most efficient* on different networks. A slower loosely coupled algorithm may perform poorly compared to a tightly coupled algorithm on a tightly coupled homogeneous network. At the same time, the tightly coupled algorithm will likely perform poorly on a loosely coupled heterogeneous network.

2.3.3 How Coupled Should Software and Hardware Become?

This leads to some interesting ideas. Different algorithms can perform better or worse on different hardware. The different hardware have different assets and liabilities including maintainability, scalability, ease of use, etc. This clearly tells us that the designs of software and hardware are inherently coupled. This coupled nature makes the design of software highly dependent on the frameworks it will be running on.

2.4 Mission of This Work

To efficiently utilize the next generation of supercomputer (heterogeneous cluster or grid), a parallelization algorithm must require little communication between processors and must be able to efficiently use processors that are running at different speeds. We propose a number of algorithms which will allow a particular application, Quantum Monte Carlo (QMC), to run faster and on more general networks of computers. This work was inspired by the authors of our software package, QMcBeaver [54], working on other QMC packages which had some deficiencies we wished to remedy to take QMC from a tightly coupled homogeneous application to a loosely coupled heterogeneous application.

Chapter 3

Introduction to Quantum Monte Carlo

3.1 Introduction

Quantum Monte Carlo (QMC) is becoming a very important member of the electron correlation correction methods in quantum chemistry. Many flavors of QMC exist; Variational (VMC, 3.2.1) and Diffusion (DMC, 3.2.2) Quantum Monte Carlo are two of the more popular methods employed. VMC requires the explicit use of a variational wavefunction, while DMC has the property that it can sample the ground state fixed node solution for a given trial wavefunction.

Experience and tradition have defined a fairly efficient method of obtaining very accurate calculations for molecules and materials using QMC [10, 13, 14, 15, 16, 18, 19, 20, 21, 22]. This protocol follows:

1. Obtain a fair trial wavefunction, Ψ_{Trial}, from some quantum mechanical method, like Density Functional Theory (DFT) or Hartree Fock (*HF*).

2. Guess Jastrow particle-particle correlation functions that have some variational form which maintains the antisymmetry of the total wavefunction. (This may only be a *nearly* antisymmetric wavefunction. Umrigar gives a discussion of this topic [13].)

3. Choose variational parameters such that any Hamiltonian singularities are satisfied with the "cusp condition" in the Jastrow form.

4. Generate an initial "walker(s)" approximately with respect to the particle probability distribution.

5. Equilibrate this "walker(s)" to verify it represents the particle probability distribution.

6. Generate configurations with the Metropolis algorithm in a VMC run.

7. Perturb and evaluate the Jastrow parameters using these configurations. (Repeat this correlated sampling optimization [10] until satisfactory convergence.)

8. Generate (or reuse from a VMC run) initial "walkers" for a DMC run.

9. Equilibrate these "walkers" to verify they represent the proper particle probability distribution.

10. Use the optimized Jastrow for a DMC run to obtain a very accurate result.

Typically the equilibration and generation of the configurations in the VMC and DMC runs are the most expensive parts of this protocol so one would like to minimize the effort in these sections. The main purpose of the

VMC optimization phase is to obtain a good description of the wavefunction. The better this wavefunction is, the quicker the DMC run will converge. This motivates one to optimize the Jastrow very well but not at the expense of marginal returns.

Experience has shown that the VMC Jastrow optimization involves a very difficult objective function. One must reduce the energy and/or variance some but without over-optimizing. The method of correlated sampling is a useful method of optimization yet once it finds a flat region of the objective function (typically a $\sigma^2(E_{Local})$ based objective function), it can falsely encourage over-optimization since it likely reached a point of diminishing returns. Experience has shown that if one can obtain roughly a factor of three reduction in the variance over the HF wavefunction alone, one has done a sufficient job of optimizing and that further optimization may give only marginal returns. Typically, one might spend from 5% to 50% of the one's total effort optimizing the Jastrow in the VMC phase of the calculation.

3.2 Theory

QMC has many flavors, each with certain assets and liabilities. The two particular types of QMC we will examine are VMC (Section 3.2.1) and DMC (Section 3.2.2). These two methods are widely used for production level calculations. Any impact one can make to improve the speed at which one can accomplish these two types of QMC will have far-reaching consequences for many researchers in computational chemistry and materials science.

3.2.1 Variational Quantum Monte Carlo

Variational Quantum Monte Carlo (VMC) is a very simple yet powerful method for examining correlated quantum wavefunctions. If one examines the basic energy expectation integral and reformulates it in terms of an electron probability density, ρ, and a local energy, E_{local}, one finds a very simple description of the energy expectation (3.1). However, this integral can not be solved exactly except for a very few cases. Instead, the integral can be numerically evaluated. This numerical integration is doomed to fail on a regular grid since the dimensionality of the integral can be very high. Instead, the integration can be accomplished with a Monte Carlo algorithm described by Metropolis [55], which can effectively numerically evaluate integrals in many dimensions.

$$\begin{aligned}
\langle E \rangle &= \int \Psi(\vec{x}) \hat{H} \Psi(\vec{x}) dx^{3n} \\
&= \int (\Psi(\vec{x}))^2 \left(\frac{\hat{H}\Psi(\vec{x})}{\Psi(\vec{x})} \right) dx^{3n} \\
&= \int \rho(\vec{x}) E_{local}(\vec{x}) dx^{3n} \qquad (3.1)
\end{aligned}$$

One must now determine what this Ψ should be. Typically, one can use a method like Hartree Fock theory or Density Functional Theory [56, 57, 58, 59, 1, 3, 4, 5, 6, 7, 8] to obtain an antisymmetric wavefunction in a determinant form. These wavefunctions contain no explicit particle correlations other than the Pauli-exclusion of fermions.

This wavefunction is then augmented with a product of symmetric terms which contain the explicit particle correlations. These particle correlation functions will allow each particle to observe the positions of their neighboring particles and will allow additional variational freedom in the wavefunction.

$$\Psi_{Trial} = \Psi_{HF} exp\left(\sum_{i}^{n}\sum_{j<i}^{n} u_{ij}\right) \tag{3.2}$$

To construct the entire trial wavefunction, Ψ_{Trial}, from an HF type initial guess wavefunction, Ψ_{HF}, one uses the following expression (3.2). A Ψ_{Trial} constructed from a DFT type wavefunction is similar.

The building unit of this type of description is a u_{ij} function for particles i and j which are of particle types A and B, respectively (3.3).

$$u_{ij} = \frac{cusp_{AB}r_{ij} + a_{AB}r_{ij}^2 + \cdots}{1 + b_{AB}r_{ij} + c_{AB}r_{ij}^2 + \cdots} \tag{3.3}$$

This particular form of u_{ij} is commonly referred to as the Padé-Jastrow correlation function for finite systems [13] or simply "Jastrow" in this document. We notice that this form contains a $cusp_{AB}$, which removes singularities which arise as two charged particles approach each other. The cusp condition puts a singularity in the kinetic energy part of the two-particle Hamiltonian which exactly removes the singularity in the potential energy part [60].

One must now determine how to optimize the parameters in the u_{ij} functions as well as how many parameters to maintain in the expression. Allowing only the cusp condition parameter in the numerator and the first parameter in the denominator is common practice, though the more parameters one optimizes, the better the result will likely be because of the additional variational freedom. The common optimization procedure is the method of correlated sampling optimization described by Umrigar [10].

3.2.2 Diffusion Quantum Monte Carlo

Examining the time-dependent Schrödinger equation (6.4) in atomic units, we observe that one can make a transformation from real time into imaginary time to produce a diffusion equation (6.6).

$$i\frac{\partial \Psi}{\partial t} = \hat{H}\Psi \tag{3.4}$$

$$t = -i\tau \tag{3.5}$$

$$\frac{\partial \Psi}{\partial \tau} = -\hat{H}\Psi = \left(\frac{1}{2}\nabla^2 - V\right)\Psi \tag{3.6}$$

Expanding Ψ in the eigenstates of the time-independent Schrödinger equation, we observe the following.

$$\Psi = \sum_{i}^{\infty} c_i \phi_i \tag{3.7}$$

Here the ϕ_i's are the eigenstates and the ε_i's are the eigenvalues of the time-independent Schrödinger equation.

$$\hat{H}\phi_i = \varepsilon_i \phi_i \tag{3.8}$$

We can now write the formal solution of the imaginary-time Schrödinger equation (Equation 6.6).

$$\Psi(\tau_1 + \delta\tau) = e^{-\hat{H}\delta\tau}\Psi(\tau_1) \tag{3.9}$$

If the initial $\Psi(\tau_1)$ is expanded in the eigenstates (Equation 3.7), we observe the following.

$$\Psi(\delta\tau) = \sum_{i}^{\infty} c_i e^{-\varepsilon_i \delta\tau} \phi_i \tag{3.10}$$

Therefore, any initial state, which is not orthogonal to the ground state, ϕ_0, will exponentially evolve to the ground state over time.

$$\lim_{\tau \to \infty} \Psi(\tau) = c_0 e^{-\varepsilon_i \tau} \phi_0 \tag{3.11}$$

The end result of this type of method is a sampling of the ground state ϕ_0 distribution with respect to the original Ψ_{Trial} nodes. In practice the results obtained from a fixed node DMC calculation are typically on the same order of accuracy as couple-cluster and higher order methods which come at a much higher expense in many cases (O($n^6 \to n!$))[20].

3.3 Conclusion

Quantum Monte Carlo is a very simple yet powerful method for examining correlated electron structure. VMC and DMC form the basis of this very powerful class of methods and provide a good starting point for improving all QMC based applications.

Chapter 4

Efficient Algorithm for "On-the-fly" Error Analysis of Local or Distributed Serially Correlated Data

4.1 Introduction

Monte Carlo methods are becoming increasingly important in calculating the properties of chemical, biological, and materials systems. An example discussed below shows that using the all-electron Variational Quantum Monte Carlo (VMC) method to calculate the barriers to decomposition of the high-energy material RDX (with 21 atoms and 114 electrons) requires approximately 10^{12} Monte Carlo steps to converge the energy estimator to roughly 0.1 kcal/mol precision.

However, there is a serious difficulty in the practical implementation of such Monte Carlo calculations. The underlying algorithms of Monte Carlo simulations generally involve Markov chains, which produce serially correlated data sets. This means that for the data set D, the value D_{i+j} is highly correlated to D_i for a value of j small compared to the correlation time, K_0.

Flyvbjerg and Peterson described a fairly efficient blocking algorithm for *post-processing error analysis* of serially correlated data on a single processor [49]. However, rather than waiting until after the run is terminated to analyze the precision, it is desirable to *specify in advance* the desired precision after which the program can terminate. This requires the computation of the true variance of serially correlated data, as the Monte Carlo calculation is evolving, "on-the-fly."

We propose a new blocking algorithm, Dynamic Distributable Decorrelation Algorithm (DDDA), which gives the same results as the Flyvbjerg-Peterson algorithm but allows the underlying variance of the serially correlated data to be analyzed "on-the-fly" with negligible additional computational expense. DDDA is ideally suited for parallel computations because only a small amount of data must be communicated between processors to obtain the global results. Furthermore, we present an efficient method for combining results from individual processors in a parallel calculation that allows "on-the-fly" result analysis for parallel calculations.

4.2 Motivation and Background

Although Monte Carlo algorithms are useful for a large range of scientific problems, the convergence to a desired precision often requires very large samplings, making it computationally expensive. In order to reduce the total time to obtain a precise and accurate solution, it is highly desirable to use parallel computing. The availability of low cost clusters and multiple processor computers makes it possible to efficiently parallelize Monte Carlo algorithms, allowing very large samplings to be probed in reasonable time. In order for Monte Carlo algorithms to continue to take full advantage of the advances and availability of massively parallel computers, it is essential that the algorithms evolve to make these methods maximally efficient.

A significant improvement for applying quantum Monte Carlo methods to the computation of chemically important systems was provided by Flyvbjerg and Peterson, who showed that simple blocking of the data (averaging blocks of data together and treating these averages as new data sets) can extract the correct sample variance from a set of serially correlated data [49, 12]. These new "blocked" data points are less correlated than the original data points and are virtually uncorrelated for block sizes larger than the correlation time of the data. Flyvbjerg and Peterson described a fairly efficient blocking algorithm for post-processing error analysis of serially correlated data on a single processor.

Although we are not certain of the historical origins of such data blocking techniques, at least partial credit should be given to Wilson [61], Whitmer [62], and Gottlieb [63]. However, Flyvbjerg and Peterson were the first to formally analyze the technique [49].

We should also note that currently some methods exist which can improve the particular implementation described. If we have some idea of the correlation time, K_0, we can block the data in these size blocks ignoring the smaller block sizes. We will refer to this method of *pre-blocked* data blocking as PB-blocking. What PB-blocking can effectively do is reduce the number of global data points from $N_G = 10^{12}$ to $N_G' = \frac{N_G}{m}$, where m is the predefined block sizes which will be considered the fundamental unit of data.

If one implements the PB-blocking method, several points need to be considered. The correlation time, K_0, is highly dependent on the Monte Carlo time step, dt, we use for the simulation. This implies the user will need to intelligently determine which m to implement.

Blocking the initial data into m sized data blocks initially still requires the user to implement a further blocking algorithm to verify that m was chosen large enough. If m was chosen too small, the Flyvberg-Peterson algorithm can still be implemented on this *pre-blocked* data with correct results. What this does to the overall scaling of the storage is nothing, however. The order of the global data points one needs to store is reduced by a constant factor of m, while the global storage is still $O(N_G)$.

Another unattractive feature of PB-blocking is the additional book-keeping and programming needed to block the raw data. This is really a two-phase algorithm, in which the initial blocking into blocks of size m feeds a Flyvberg-Peterson type algorithm to verify properly uncorrelated data.

We have also found that although the small blocked data (which is still correlated) underestimates the variance, it can still play a role in determining an extrapolation of the variance since these underestimates generally converge to their respective values fairly well. The PB-blocking algorithm essentially throws the small data blocking (blocks smaller than m) away to reduce the storage and communication expenses. This can be useful information when determining the level of correlation which is present at differing block sizes and seeing how this drops off over block size.

The overall reduction of statistics and communication phase for the PB-blocking algorithm still scales as $O(N_G)$. As we probe larger and larger samplings, this can be a prohibitive expense. What is potentially very useful for those who still wish to implement the PB-blocking algorithm is the use of the *pre-blocking* with the DDDA algorithm. Although the large sample scaling will be the same as using the pure DDDA algorithm, this can further reduce the total expense if a good guess m value can be determined. We chose not to implement the *pre-blocking* step since it is additional effort to implement and it gains us so little when we can simply use DDDA which effectively makes the storage and communications expense negligible.

Instead, we aim to improve upon both the PB-blocking and the pure Flyvberg-Peterson algorithm. We wish to simultaneously reduce the amount of user input into the method by eliminating the *pre-blocking* step requiring some level of user expertise to determine m, reduce the global storage, and reduce expenses rigorously to $O(log_2(N_G))$ from $O(N_G)$, and to accomplish a reduction of the global statistics "on-the-fly" with minimal expense.

In this paper, we reformulate the same mathematical results of Flyvbjerg and Peterson to allow efficient decorrelation of serially correlated data "on-the-fly" and in parallel for the extremely large amount of sampling required in Markov chain based calculations such as Quantum Monte Carlo calculations of electronic wavefunctions.

4.3 Theory

Computer simulations of physical systems often involve the calculation of an expectation value, $\langle f \rangle$, with respect to a complicated probability distribution function, $\rho(x)$.

$$\langle f \rangle \equiv \int \rho(x) f(x) dx \tag{4.1}$$

This expression is simple and elegant, but in many physical systems, $\rho(x)$ is too complex for Equation 4.1 to be useful computationally. Commonly, computer simulations involve calculation of the average of samples over some number of Monte Carlo steps (or molecular dynamics times).

$$\bar{f} \equiv \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{4.2}$$

Here x_i is sampled from the distribution $\rho(x)$ using a Monte Carlo or molecular dynamics simulation; i is related to the Monte Carlo step number or molecular dynamics time. Assuming ergodicity, then

$$\langle f \rangle = lim_{n\to\infty} \bar{f} = lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{4.3}$$

Modern computing machines allow the very large samplings required to approach this limit. However, since such sampling is necessarily always finite, $\bar{f}$ will fluctuate requiring the determination of its variance in order to evaluate the results. The variance, σ^2, of $\bar{f}$ can be expressed as

$$\sigma^2(\bar{f}) = \frac{1}{n^2} \sum_{i,j=1}^{n} \left(\langle f(x_i) f(x_j) \rangle - \langle f(x_i) \rangle \langle f(x_j) \rangle \right), \tag{4.4}$$

which is valid for the analysis of correlated data. For uncorrelated data this reduces to the typical variance relation

$$\sigma^2(\bar{f}) = \frac{\langle f^2 \rangle - \langle f \rangle^2}{n} \tag{4.5}$$

Applying Equation 4.5 to systems with correlated data leads to a lower bound of the *true* variance, which is obtained from Equation 4.4.

Flyvbjerg and Peterson show formally that uncorrelated variance estimates can be extracted from serially correlated data by applying a "blocking" transformation. The "blocking" transformation begins by partitioning the serially correlated data into equal-sized, nonoverlapping "blocks" of data. An average is performed over each block to obtain a new reduced set of data. With a sufficiently large block size, the resulting data will be uncorrelated and its variance can be calculated as

$$\sigma^2(\bar{f}) = \frac{1}{blocks}\sum_{i=1}^{blocks}\left(f_i^{block\_ave}\right)^2 - \frac{1}{block^2}\left(\sum_{i=1}^{blocks} f_i^{block\_ave}\right)^2, \tag{4.6}$$

where $f_i^{block\_ave}$ is the average of $f(x_i)$ over block i. In practical terms, the correct block size can be determined by finding the block size after which equation Equation 4.6 plateaus as shown in Figure 4.2.

4.3.1 Computational Cost of Flyvbjerg-Peterson Algorithm

When serially correlated data is collected from a Markov chain-based simulation, the average and variance of the average are the most commonly calculated quantities. The Flyvbjerg-Peterson blocking algorithm requires storing all of the collected data. This has two negative consequences:

- It cannot be performed "on-the-fly,"
- It requires $O(N)$ storage.

If N is assumed to be a power of two and all block sizes are taken to be powers of two, this algorithm requires $5N-3$ floating point additions, $3N-2$

floating point multiplications, and $2N-1$ integer additions to calculate the average and variance of the data with all possible block sizes. Often, the data must be read in from a file to be analyzed, adding an additional slow $O(N)$ operation to the computational cost.

4.3.2 Dynamic Distributable Decorrelation Algorithm (DDDA)

Our new algorithm involves two classes:

Statistic Class

(Pseudocode is listed in Supplement 4.6)

The Statistic class stores the number of samples, running sum of x_i, and running sum of x_i^2 for the data that is entered into it. This allows straightforward calculation of the average (Equation 4.7)

$$\bar{x} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.7}$$

and variance (Equation 4.8)

$$\hat{\sigma}^2(x) = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \frac{1}{n^2}\sum_{i=1}^{n} x_i^{\,2}}{n-1}, \tag{4.8}$$

where n is the number of samples in the Statistic object, and the x_i are the data elements added to the object. This variance estimator only returns the true variance if the data is uncorrelated.

Decorrelation Class

(Pseudocode is listed in Supplement 4.7)

The Decorrelation class stores a vector of Statistics (BlockedDataStatistics), where BlockedDataStatistics[i] corresponds to data that has been partitioned into blocks 2^i long. As new data is collected during a computation, it is added to a Decorrelation object using the *add_Data*(x) function. This function determines when enough data samples exist to create new data blocks and then adds the data from the new blocks to the correct elements of BlockedDataStatistics. An operation is also presented to combine Decorrelation objects generated from independent Markov chains that are produced in parallel computations.

Sample pseudocode for applying the algorithm is presented in Supplement 4.8. This simple code demonstrates the ease of implementation for the new algorithm. This code can be easily modified to include any special features of a particular application including convergence-based termination.

This construction is well suited for parallel computations where multiple, distributed Decorrelation objects will be generated. To combine these objects efficiently, when analyzing the global results for a distributed calculation, it is necessary to efficiently add the data from a number of Decorrelation objects to form a new composite Decorrelation object. The addition operation listed in Supplement 4.7 provides this functionality.

The equations implemented by DDDA are exactly the same as those presented by Flyvbjerg and Peterson. Our implementation, however, is more efficient. This allows new data to be added "on-the-fly" and allowing all current data to be analyzed "on-the-fly" with negligible additional cost. The results obtained from the original Flyvbjerg-Peterson algorithm and DDDA are identical because they are mathematically equivalent.

Summary of Algorithm

The DDDA algorithm is very simple and relies only on the Statistic (4.3.2) and Decorrelation (4.3.2) classes. A Decorrelation object is just an array of Statistics objects. The Decorrelation array element zero corresponds to a block size of one (or block size 2^0), array element one corresponds with a block size of two (or block size 2^1), and array element i corresponds with a block size of 2^i.

When we observe a new sample, we place the sample into the Decorrelation structure at level zero. This gets added to this Statistic and then gets propagated up one level to level one. If a sample is waiting to construct a new block, this new sample and the waiting sample are averaged and added to this level as a new sample. This new composite sample is then recursively sent down the structure constructing new blocks of data. If a sample gets propagated a level and no waiting sample exists to form a new block, this sample becomes the waiting sample which is followed by the termination of this round of updating the Decorrelation structure.

4.3.3 Computational Cost of DDDA

Analysis of DDDA (Table 4.1) shows that it requires $5N - 3$ floating-point additions, $3N - 2$ floating-point multiplications, and $2N - 1$ integer additions, which is identical to the Flyvbjerg-Peterson algorithm. However, in contrast with the Flyvbjerg-Peterson algorithm, DDDA requires storage of only $O(log_2(N))$ numbers and requires no time to read data from a file because the data is added to a Decorrelation object "on-the-fly."

We should note that a storage unit in DDDA is roughly three times as

Expense	Flyvbjerg-Peterson Algorithm	Dynamic Distributable Decorrelation Algorithm (DDDA)
Floating Point Multiplications	$3N-2$	$3N-2$
Floating Point Additions	$5N-3$	$5N-3$
Integer Additions	$2N-1$	$2N-1$
Read-in Data Cost	$O(N)$	None
Storage Cost	$O(N)$	$O(log_2 N)$
"On-the-fly" Calculation	not practical	negligible

Table 4.1: Comparison of computational costs. Here N is the number of data points analyzed. In evaluating the costs, N is assumed to be a multiple of two. This represents the worst-case scenario.

large as the storage unit in the original algorithm. This factor of three results from the class Statistic having three data members. If we assume that the data type used for each data point and the data members of a Statistic object have roughly the same number of bits, the storage cost of DDDA is $3log_2(N)$ which scales as $O(log_2(N))$.

If the correlation time for the data is known approximately, then block sizes significantly larger than this are not required; therefore, it is not necessary for them to be saved or calculated. This reduces the storage cost to $O(1)$. If a sufficiently large block size is not allowed, the calculated variance will be incorrect because the largest data blocks used to calculate the variance will still be correlated. (This is just a special instance where the computational complexity and costs can be managed if the approximate correlation time is known *a priori.*)

To provide an idea of the impact of DDDA, consider the example of predicting the activation energy for decomposition of the RDX molecule discussed below. We estimate that $N_G = 10^{12}$ global data points are required. Thus with $M = 10^3$ processing units, the original algorithm requires local processor storage of $O(N)$, $N \approx \frac{N_G}{M} = 10^9$ numbers, which may be difficult to accommodate on the local memory and may be very difficult to transfer efficiently between processors. In contrast for $N_G = 10^{12}$ and $M = 10^3$, the local processor storage requirement for DDDA is $3log_2(N) = 120$, which is much easier to accommodate than 10^9.

4.4 Computational Experiments

4.4.1 Variational QMC on One-Dimensional Particle-in-a-Box

Details of the calculations

To illustrate DDDA, we consider using Variational Quantum Monte Carlo (VMC) [13] to calculate the energy for a one-dimensional particle-in-a-box of length one. The expected energy of the system is given by Equation 4.9

$$\begin{aligned}\langle E\rangle &= \int_0^1 \Psi_T \hat{H} \Psi_T dx \\ &= \int_0^1 (\Psi_T)^2 \left(\frac{\hat{H}\Psi_T}{\Psi_T}\right) dx \\ &= \int_0^1 p_T(x) E_L(x) dx, \end{aligned} \tag{4.9}$$

where Ψ_T is a normalized, approximate wavefunction, $\hat{H}$ is the Hamiltonian for the system, $E_L(x)$ is the local energy, and $p_T(x)$ is the approximate probability distribution of the particle. Equation 4.9 can be evaluated in two ways:

- One option (Method 1) is to perform a Monte Carlo integral using uniform random numbers to sample $E_L(x)$ with weight $\rho_T(x)$. Because the uniform random numbers are not serially correlated, the sampled values of $\rho_T(x)E_L(x)$ are not serially correlated.

- A second option (Method 2) is to generate points distributed with respect to $\rho_T(x)$ using the Metropolis algorithm [55] and use these points to sample $E_L(x)$. Because the Metropolis algorithm employs a Markov chain, this method will produce serially correlated data.

For our illustration, we chose

$$\Psi_T = \sqrt{30}\left(x - x^2\right) \tag{4.10}$$

This trial wavefunction is a good approximation to the exact ground state wavefunction, $\Psi_{Exact} = \sqrt{2}\sin(\pi x)$. Since the Ψ_T is not an eigenfunction for this system, the local energy will not be constant and the calculated energy expectation value will fluctuate.

Results

DDDA produces the same results as the Flyvbjerg-Peterson algorithm but is a more efficient implementation. The analytic expectation value of the energy is $\langle E \rangle = 5.0$. The uncorrelated estimate of the energy, calculated by Method 1, is $\langle E \rangle = 5.0014(22)$ and the correlated estimate, calculated by Method 2, is $\langle E \rangle = 5.0018(59)$.

The noncorrelated VMC "particle-in-a-box" calculation (Method 1) produces a nearly flat standard deviation estimation for blocks of 2^0 to 2^{12} points (Figure 4.1). This is the expected behavior for noncorrelated data because Equation 4.5 provides a correct prediction of the variance. The poor performance for large block sizes results because they have very few data points leading to less stability in estimating the standard deviation.

The correlated VMC "particle-in-a-box" calculation (Method 2) leads to a nearly monotonic increasing estimate of the standard deviation that levels off for blocks of 2^8 to 2^{12} points (Figure 4.2). The plateau in the standard deviation estimation corresponds to the correct standard deviation of the calculated expectation value. Furthermore, the plateau indicates that blocks of 2^8 points are essentially uncorrelated so that Equation 4.6 provides an appropriate estimate of the variance.

We should note that using Equation 4.5 on correlated data without data blocking yields an estimate of the standard deviation that is much too small. This corresponds to a block size of one (the y-axis intercept) in Figure 4.2. This illustrates the potential dangers in reporting error estimates of data without accounting for the serial correlation that may exist.

We should also note that the error estimates of the correlated and uncor-

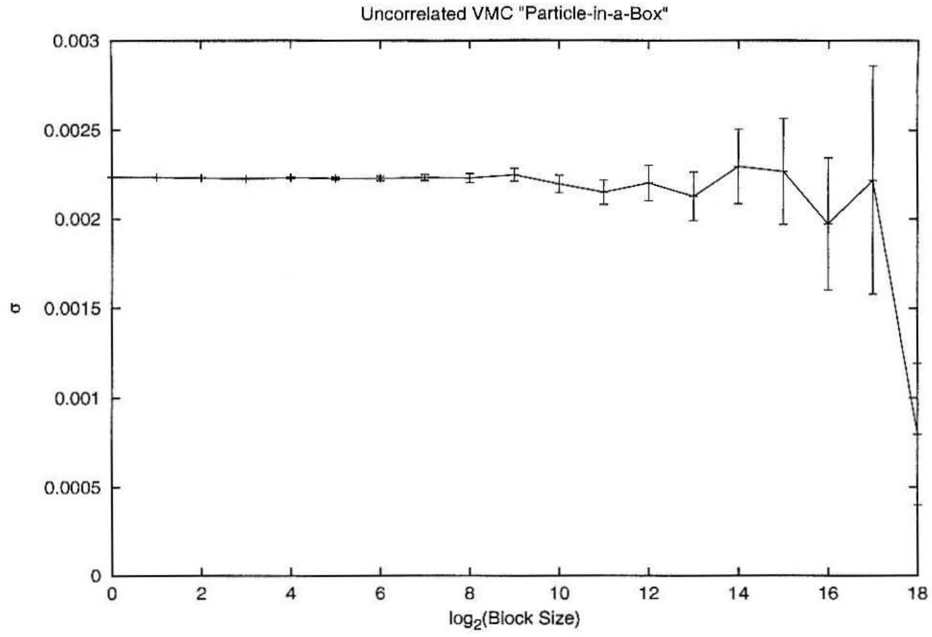


Figure 4.1: The energy expectation value standard deviation, evaluated with Eq. 4.6, as a function of block size for a VMC "particle-in-a-box" calculation using Method 1 to generate uncorrelated data points. The Flyvbjerg-Peterson algorithm and DDDA yield exactly the same results. The error bars represent one standard deviation in the calculated standard deviation estimator.

related "particle-in-a-box" calculations are different. These error estimates illustrate that serially correlated data does not provide as much information as uncorrelated data, resulting in a larger standard deviation for the correlated case than the uncorrelated case when using the same number of samples.

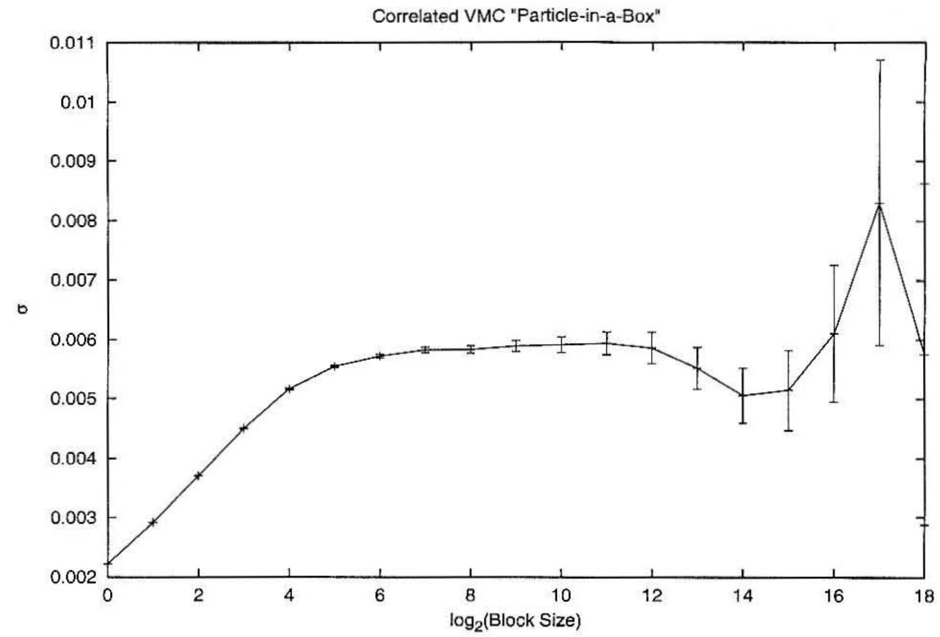


Figure 4.2: The energy expectation value standard deviation, evaluated with Eq. 4.6, versus block size for a VMC "particle-in-a-box" calculation using Method 2 to generate correlated data points. The Flyvbjerg-Peterson algorithm and DDDA yield exactly the same results. The error bars represent one standard deviation in the calculated standard deviation estimator.

4.4.2 Finite All-Electron Variational QMC on RDX

Introduction

For a more realistic test of these algorithms, we consider a practical problem of using Variational Quantum Monte Carlo (VMC) to determine the barrier height for the unimolecular decomposition of the high explosive molecule, RDX (see Fig. 4.3), cyclic $[CH_2 - N(NO_2)]_3$.

The best available DFT calculations [2] indicate an activation barrier of 39.0 kcal for NN dissociation, 39.2 kcal/mol for $HONO$ elimination, and

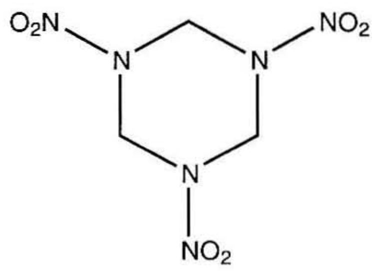


Figure 4.3: The RDX molecule, cyclic $[CH_2\text{-}N(NO_2)]_3$.

59.4 kcal/mol for concerted decomposition. Various choices of high-quality basis sets with various choices of high-quality density functionals (generalized gradient approximations) lead to changes in these activation energies by ~ 5 kcal/mol. These uncertainties pertain to the underlying assumptions about DFT so that we cannot improve this accuracy by just additional computing. Thus with current DFT methodology we are stuck with uncertainties of ~ 5 kcal/mol. In order to properly model combustion processes involving RDX, it would be desirable to know these energies to 0.1 kcal/mol. On the other hand, using various flavors of QMC the calculated energy can be systematically improved. If we have a way to gauge the uncertainty that can be applied while the calculation is underway, then we can continue running until this level of convergence is achieved. Numerous calculations with various flavors of DFT indicate that with good basis sets and good generalized gradient approximations, the geometries generally agree to better than 0.01A with each other and with experiment. Thus a practical procedure would be to determine the geometries (including transitions states) using DFT theory, then start the QMC with this wavefunction and optimize the correlation functions until the results will allow a DMC run to converge to the designed accuracy. To illustrate this process for a general serially correlated Monte

Carlo application, we carried out some initial VMC calculations for RDX.

RDX $= cyclic - [CH_2 - N(NO_2)]_3$ is composed of 21 atoms with 114 electrons, making an all-electron VMC calculation nontrivial but tractable. To demonstrate the robustness of DDDA, we used VMC to calculate the energies of the ground state and two transition state intermediates. In these calculations, we use the structures from the DFT calculations [2].

Details of the VMC Calculations

VMC calculations were performed using QMcBeaver [54], which implements DDDA. Though much work has been done on proper wavefunction optimization techniques [10, 13, 14, 15, 16, 18, 19, 20, 21, 22], we examine a very simple form of the VMC wavefunction as written in Equation 4.11.

$$\begin{aligned} \Psi_{trial} &= \Psi_{HF} J_{Corr} \\ J_{Corr} &= \exp\left(\sum_i \sum_{j<i} u_{i,j}\right) \end{aligned} \tag{4.11}$$

This is the product of a Hartree-Fock wavefunction, (HF), calculated using Jaguar [1, 3], with a Pade-Jastrow correlation function (Equations 4.11, 4.12, 4.13), J_{Corr}. The particular form of the Pade-Jastrow correlation function is in Equation 4.12.

$$u_{AB} = \frac{cusp_{AB} r}{1 + b_{AB} r} \tag{4.12}$$

This was chosen to remove singularities in the local energy while maintaining the structure of the original wavefunction everywhere except where two particles closely approach each other. Thus for the electron-nuclear terms, we set $cusp = -Z$ and $b = 100$. Similarly for the electron-electron terms we use the analytically derived values of $cusp = 0.25$ for same spin and $cusp = 0.50$ for opposite spin electrons. For same spin electrons, we use $b = 100$; while for the opposite spin electrons, we use $b = 3.5$, which we have found works fairly well for the ground state of a number of small molecules containing carbon, hydrogen, oxygen, and nitrogen. These are displayed in Equations 4.13.

$$\begin{aligned}
u_{\uparrow\downarrow} &= \frac{\frac{1}{2}r_{ij}}{1+3.5r_{ij}} \\
u_{\uparrow\uparrow} = u_{\downarrow\downarrow} &= \frac{\frac{1}{4}r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,H} = u_{\downarrow,H} &= \frac{-r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,C} = u_{\downarrow,C} &= \frac{-6r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,N} = u_{\downarrow,N} &= \frac{-7r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,O} = u_{\downarrow,O} &= \frac{-8r_{ij}}{1+100r_{ij}} \qquad (4.13)
\end{aligned}$$

Such correlation functions with fixed parameters provide a crude but "generic" approach to determining a portion of the correlation energy missing in the Hartree-Fock wavefunction.

Of the three calculations, two were run to completion while the third calculation was stopped a fraction of the way through the run and restarted from checkpoints to verify the ease and efficiency with which these new structures allow for checkpointing of the program state variables. The calculations were performed on the ASCI-BLUE Mountain supercomputer at the Los Alamos National Laboratory using 1024 MIPS 10000 processors running at 250 MHz.

Energies for the Jaguar Hartree Fock (HF), Jaguar density functional theory (DFT-B3LYP) [1, 3, 4, 5, 6, 7, 8], and QMcBeaver variational quantum

Species	Hartree Fock	Variational Quantum Monte Carlo
Ground state	-892.491	-893.35(4)
Concerted dissociation	-892.369	-893.29(5)
$N - NO_2$ bond fission	-892.259	-893.20(4)

Table 4.2: Total energies (Hartree) for the various calculations on RDX. The HF and DFT [2] results were obtained from Jaguar 4.1 with the 6-31G** basis set [1, 3, 4, 5, 6, 7, 8]. Variational Quantum Monte Carlo based on 3×10^7 points.

Monte Carlo (VMC) [54] calculations are presented in Table 4.2.

Results

The RDX calculations successfully completed independent of whether they were run to completion or checkpointed and restarted.

Figures 4.4, 4.5, and 4.6 show the evolution of the energy standard deviation estimate as the number of Monte Carlo steps is increased for calculations on the three different RDX species. The standard deviation in the VMC energy expectation value decreases with the number of samples, roughly following the form $\frac{1}{\sqrt{N}}$. Here we see that the plateau in the plot of standard deviation vs. $log_2(block\_size)$ is reached for a block size of roughly 2^8 to 2^{13}.

The dependence of the standard deviation on the number of steps is shown in Fig. 4.7. Based on these results, we estimate that 10^{10} steps are required for 1 kcal/mol uncertainty in the calculated energy estimator, while 10^{12} steps are required for 0.1 kcal/mol uncertainty in the calculated energy estimator.

Another important observation from these calculations is that these crude

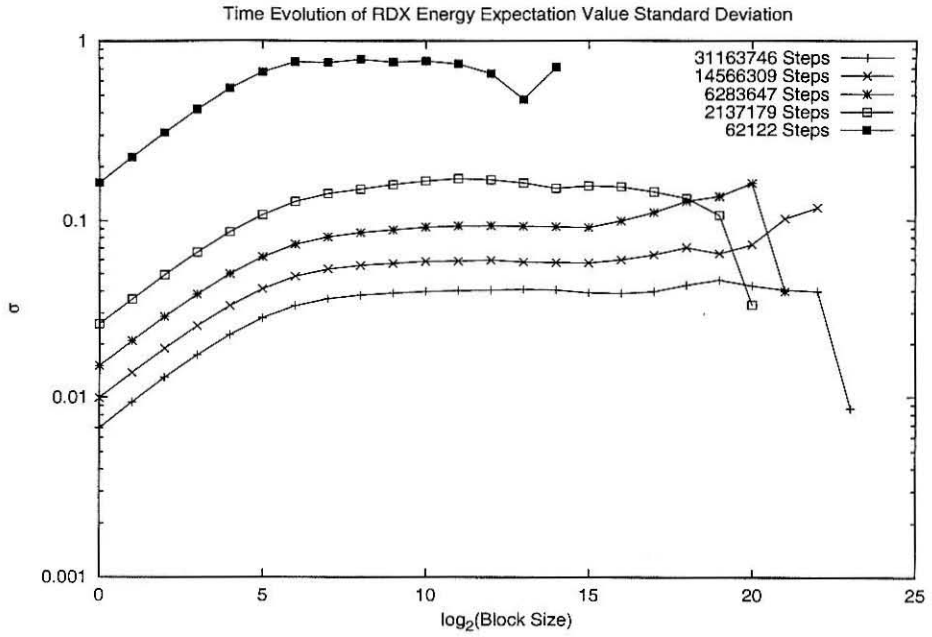


Figure 4.4: The evolution of the energy-standard-deviation-estimator for the ground state of RDX with block size. Shown here are the results for five cases with 62122, 2137179, 6283647, 14566309, and 31163746 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^8 = 256$ is sufficient for this calculation.

"generic" Pade-Jastrow correlation functions appear somewhat effective in improving the Hartree-Fock wavefunctions.

4.5 Conclusions

The primary goal here was to show the robustness and efficiency of DDDA. This method can eliminate computationally expensive I/O operations and reduce the overall storage requirement to $O(log_2(N))$ from $O(N)$. Furthermore, this method allows the variance of the calculated quantity to be evaluated

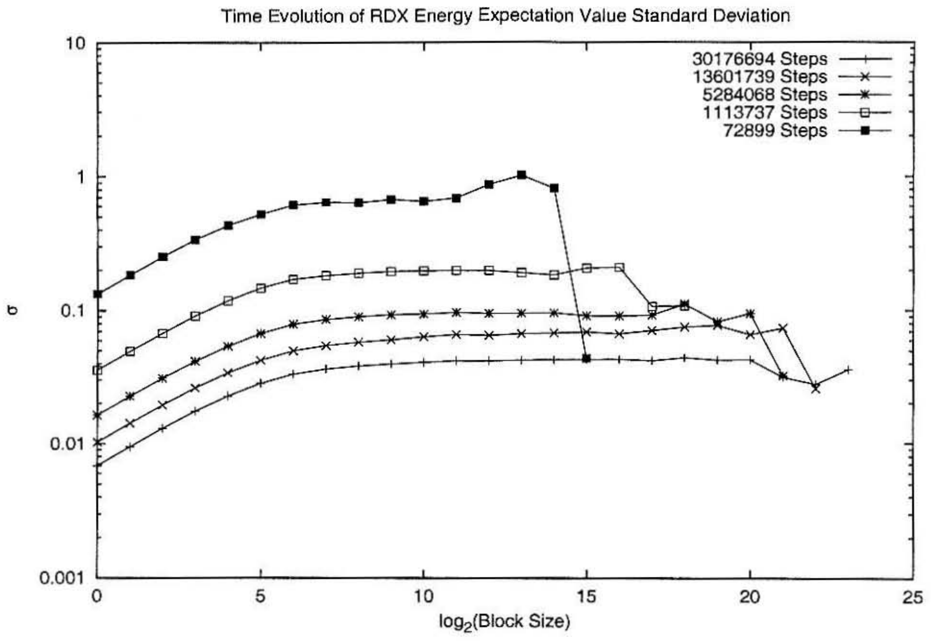


Figure 4.5: The evolution of the energy-standard-deviation-estimator for the transition state for N-NO2 bond dissociation in RDX with block size. Shown here are the results for five cases with 72899, 1113737, 5284068, 13601739, and 30176694 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^8 = 256$ is sufficient for this calculation.

"on-the-fly." This allows a calculation to be terminated when the calculated quantities are converged instead of having to pre-specify the number of simulation steps to be performed.

Variance estimation for parallel simulations is easily and efficiently performed with DDDA. While the Flyvbjerg-Peterson algorithm requires $O(N)$ data points to be communicated over a network connection to evaluate the variance of the global calculation, DDDA requires only $O(1)$ to $O(log_2(N))$ data points to be communicated. This is a great benefit when large amounts of data are generated or when calculations are performed on a "grid" or

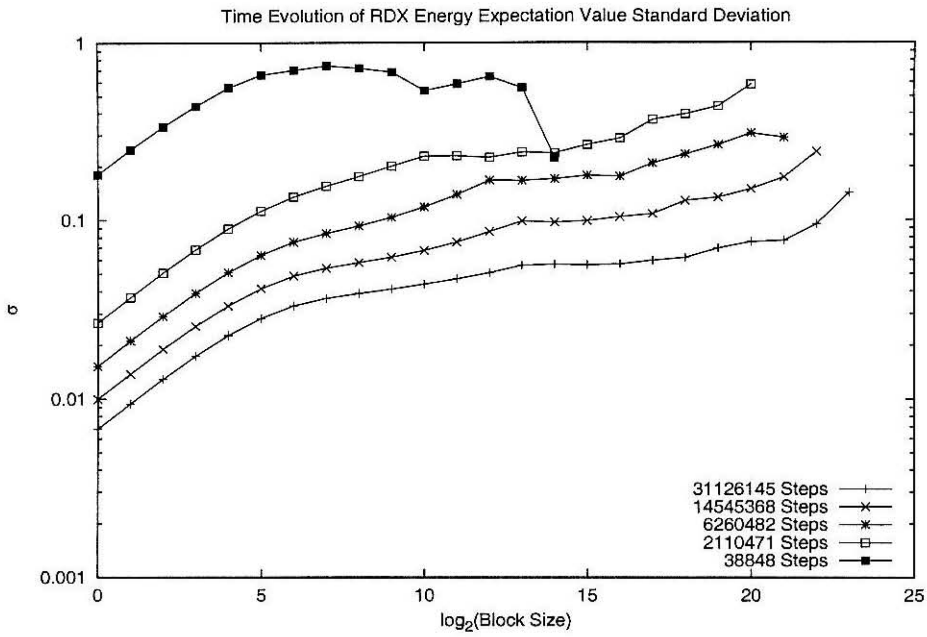


Figure 4.6: The evolution with block size of the energy-standard-deviation-estimator for the transition state for concerted symmetric ring decomposition of RDX. Shown here are the results for five cases with 38848, 2110471, 6260482, 14545368, and 31126145 total QMC steps. The energies are in Hartree (1 Hartree = 27.2116 eV). This shows that a block size of $2^{13} = 8192$ is sufficient for this calculation.

other computational network with potentially limited bandwidth. Furthermore, the Flyvbjerg-Peterson algorithm is typically implemented so that all calculations are performed on one processor. DDDA efficiently partitions the calculation between all available processors.

4.6 Statistic Class Pseudocode

4.6.1 Pseudocode for *Statistic.initialize*()

Statistic.initialize()

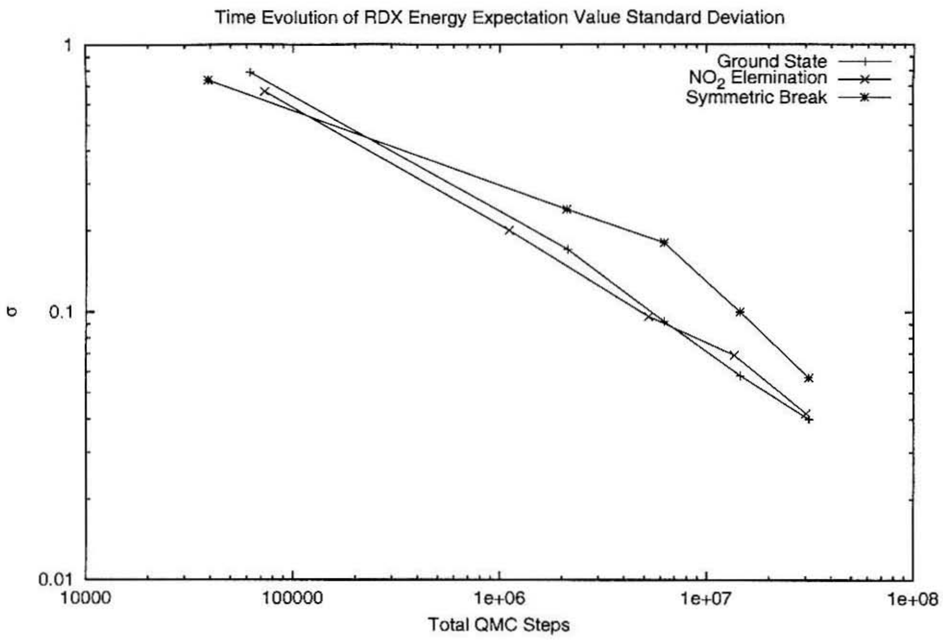


Figure 4.7: The evolution of the standard-deviation-estimate for the energy of the three states of RDX whose results were shown in Fig. 4.4, 4.5, and 4.6. A block size of 2^8 was used for the ground state and the transition state for N-NO2 dissociation while a block size of 2^{13} was used for the symmetric concerted transition state. The energies are in Hartree (1 Hartree = 27.2116 eV).

```
# When a new instance of the Statistic class is created
    initialize its attributes
$NSamples = 0.0$
$Sum = 0.0$
$SumSq = 0.0$
```

4.6.2 Pseudocode for *Statistic.add_Data(new_sample)*

```
Statistic.add_Data(new_sample)
    # Add a new data element to the Statistic object and update
        the object's attributes
    $NSamples = NSamples + 1$
    $Sum = Sum + new\_sample$
    $SumSq = SumSq + new\_sample * new\_sample$
```

4.6.3 Pseudocode for *Statistic.addition(A, B)*

```
Statistic.addition(A, B)
    # Add two Statistics to create a new composite statistic
    # C=A+B (so C is the resulting Statistic object)
    $C = new\ Statistic()$
    $C.NSamples = A.NSamples + B.NSamples$
    $C.Sum = A.Sum + B.Sum$
    $C.SumSq = A.SumSq + B.SumSq$
    $returnC$
```

4.7 Decorrelation Class Pseudocode

4.7.1 Pseudocode for *Decorrelation.initialize*()

```
$Decorrelation.initialize()$ :
    # When a new instance of the Decorrelation class is
        created, initialize its attributes
    $Size = 0$
    $NSamples = 0$
    $BlockedDataStatistics = [new\ Statistic()]$
    $waiting\_sample = [0]$
    $waiting\_sample\_exists = [false]$
```

4.7.2 Pseudocode for *Decorrelation.add_Data*(*new_sample*)

```
$Decorrelation.add\_Data(new\_sample)$ :
    # Add a new data element to the Decorrelation object
        and update the object's attributes
    $NSamples = NSamples + 1$
    # This will dynamically make the Decorrelation arrays
        longer to fit in all the data
    $if\ NSamples >= 2.0^{Size}$ :
        $Size = Size + 1$
        $BlockedDataStatistics =$
            $BlockedDataStatistics.append(newStatistic()$
        $waiting\_sample = waiting\_sample.append(0)$
        $waiting\_sample\_exists = waiting\_sample\_exists.append(false)$
    $BlockedDataStatistics[0].add\_Data(new\_sample)$
    $carry = new\_sample$
    $i = 1$
    $done = false$
    # Propagate the new sample up through the
        BlockedDataStatistics structure
    $while(not\ done):$
        $if\ waiting\_sample\_exists[i]:$
            $new\_sample = (waiting\_sample[i] + carry)/2.0$
            $carry = new\_sample$
            $BlockedDataStatistics[i].addData(new\_sample)$
            $waiting\_sample\_exists[i] = false$
        $else:$
            $waiting\_sample\_exists[i] = true$
            $waiting\_sample[i] = carry$
            $done = true$
        $i = i + 1$
        $if\ i > Size:$
            $done = 1$
```

4.7.3 Pseudocode for *Decorrelation.addition(A, B)*

```
$Decorrelation.addition(A, B):$
    # Add two Decorrelation objects to create a new
    composite Decorrelation object
# C=A+B (so C is the resulting Decorrelation object)
$C = newDecorrelation()$
$C.NSamples = A.NSamples + B.NSamples$
# Make C big enough to hold all the data from A and B
$while\ C.NSamples >= 2.0^{C.Size}:$
    $C.Size = C.Size + 1$
    $C.BlockedDataStatistics =$
        $C.BlockedDataStatistics.append(newStatistic())$
    $C.waiting\_sample = C.waiting\_sample.append(0)$
    $C.waiting\_sample\_exists =$
        $C.waiting\_sample\_exists.append(false)$
$carry\_exists = false$
$carry = 0$
$for\ i\ in\ range(C.Size):$
    $if\ i <= A.Size:$
        $StatA = A.BlockedDataStatistics[i]$
        $waiting\_sampleA = A.waiting\_sample[i]$
        $waiting\_sample\_existsA = A.waiting\_sample\_exists[i]$
    $else:$
        $StatA = new\ Statistic()$
        $waiting\_sampleA = 0$
        $waiting\_sample\_existsA = false$
    $if\ i <= B.Size:$
        $StatB = B.BlockedDataStatistics[i]$
    $waiting\_sampleB = B.waiting\_sample[i]$
    $waiting\_sample\_existsB = B.waiting\_sample\_exists[i]$
$else:$
    $StatB = new\ Statistic()$
    $waiting\_sampleA = 0$
    $waiting\_sample\_existsA = false$
$C.BlockedDataStatistics[i] =$
    $C.BlockedDataStatistics[i].addition(StatA, StatB)$
$if(carry\_exists == true\ \&\ waiting\_sample\_existsA == true\ \&$
    $waiting\_sample\_existsB == true):$
    # We have three samples to handle
    $C.BlockedDataStatistics[i].addData($
        $(waiting\_sampleA + waiting\_sampleB)/2.0)$
    $C.waiting\_sample[i] = carry$
    $C.waiting\_sample\_exists[i] = true$
    $carry\_exists = true$
    $carry = (waiting\_sampleA + waiting\_sampleB)/2.0$
$else\ if(carry\_exists == false\ \&\ waiting\_sample\_existsA == true\ \&$
    $waiting\_sample\_existsB == true):$
    # We have two samples to handle
    $C.BlockedDataStatistics[i].addData($
        $(waiting\_sampleA + waiting\_sampleB)/2.0)$
    $C.waiting\_sample[i] = 0$
    $C.waiting\_sample\_exists[i] = false$
    $carry\_exists = true$
    $carry = (waiting\_sampleA + waiting\_sampleB)/2.0$
$else\ if(carry\_exists == true\ \&\ waiting\_sample\_existsA == false\ \&$
    $waiting\_sample\_existsB == true):$
    # We have two samples to handle
    $C.BlockedDataStatistics[i].addData($
        $(carry + waiting\_sampleB)/2.0)$
    $C.waiting\_sample[i] = 0$
    $C.waiting\_sample\_exists[i] = false$
    $carry\_exists = true$
    $carry = (carry + waiting\_sampleB)/2.0$
$else\ if(carry\_exists == true\ \&\ waiting\_sample\_existsA == true\ \&$
    $waiting\_sample\_existsB == false):$
    # We have two samples to handle
    $C.BlockedDataStatistics[i].addData($
        $(carry + waiting\_sampleA)/2.0)$
    $C.waiting\_sample[i] = 0$
    $C.waiting\_sample\_exists[i] = false$
    $carry\_exists = true$
    $carry = (carry + waiting\_sampleA)/2.0$
$else\ if(carry\_exists == true\ or\ waiting\_sample\_existsA == true\ or$
    $waiting\_sample\_existsB == true):$
    # To get to this code we must only have one sample to handle
    $C.waiting\_sample[i] = carry+$
        $waiting\_sampleA + waiting\_sampleB$
    $C.waiting\_sample\_exists[i] = true$
```

```
            $carry\_exists = false$

            $carry = 0$

        $else:$

            # There are no samples to handle here

            $C.waiting\_sample[i] = 0$

            $C.waiting\_sample\_exists[i] = false$

            $carry\_exists = false$

            $carry = 0$

    $returnC$
```

4.8 Simple Example Calculation Pseudocode

```
$for\ all\ processors:$

    # Initialize the error analysis data structure for each processor

    $LocalErrorAnalysisDataStructure = newDecorrelation()$

    $while\ generating\ new\ data\ points:$

        # Generate new data and add it to the local error

            analysis data structure

        $new\_data = generateNewDataPoint()$

        $LocalErrorAnalysisDataStructure.add\_Data(new\_data)$

    $if\ not\ root\_node:$

        # Send data to the root processor to evaluate the global

            expectation value and Variance

        $send\ LocalErrorAnalysisDataStructure\ to\ root\_node$

    $else:$
```

```
# Generate the global error analysis data structure by
    adding the local error
$GlobalErrorAnalysisDataStructure =$
    $LocalErrorAnalysisDataStructure$
# Analysis data structures form each processor
$for\ processor\ in\ all\ processors\ excluding\ the\ root\ node:$
    $receive(LocalErrorAnalysisDataStructureprocessor)$
    $GlobalErrorAnalysisDataStructure = Decorrelation.add($
        $GlobalErrorAnalysisDataStructure,$
        $LocalErrorAnalysisDataStructureprocessor)$
```

Chapter 5

Manager–Worker-Based Model for the Parallelization of Quantum Monte Carlo on Heterogeneous and Homogeneous Networks

5.1 Introduction

There is currently a great deal of interest in making Quantum Monte Carlo (QMC) methods practical for everyday use by chemists, physicists, and material scientists. Since protocols exist using QMC methods, such as variational QMC, diffusion QMC, and Green's function QMC, to calculate the energy of an atomic or molecular system to within chemical accuracy (< 2 kcal/mol), this makes their everyday application very attractive. High accuracy quantum mechanical methods generally scale very poorly with problem size, typically $O(N^6$ to $N!)$, while QMC scales fairly well, $O(N^3)$, but with a large prefactor. Current research efforts exist to improve QMC's scaling further [45]. Density Functional Theory (DFT) scales well, $O(N^3)$, and could potentially provide highly accurate solutions, but DFT typically has an accuracy of 5 kcal/mol or more with the current generation of functionals and the results cannot be systematically improved.

The primary issue facing the QMC community is that, although QMC scales well with problem size, the prefactor of the method is generally very large, often requiring CPU months to calculate moderately sized systems. The Monte Carlo nature of QMC allows it to be easily parallelized, thus, reducing the prefactor, with respect to the wall clock.

Application of QMC to physically interesting systems almost always requires the use of supercomputers to enable calculations to complete in a reasonable amount of time. Currently, however, supercomputing resources are very expensive and can be difficult to gain access to. To make QMC more useful for an average practitioner, algorithms must become more efficient, and/or large inexpensive supercomputers must be produced.

A current trend in large scale supercomputing [51] is assembling "cheap supercomputers" with commodity components using a Beowolf-type framework. These clusters have proven to be very powerful for high-performance scientific computing applications [52]. Clusters can be constructed as homogeneous supercomputers if the hardware for each node is equivalent or as heterogeneous supercomputers if various generations of hardware are included.

Another interesting development is the use of loosely coupled, distributed grids of computational resources [53] with components that can even reside in different geographic locations in the world. Such "grids" are upgraded by adding new compute nodes to the existing grid resulting in continuously upgradable supercomputers, which are inevitably heterogeneous.

To efficiently utilize the next generation of supercomputer (heterogeneous cluster or grid), a parallelization algorithm must require little communication between processors and must be able to efficiently use processors that are running at different speeds. We propose a manager–worker-parallelization algorithm for QMC (QMC-MW) that is designed for just such systems. This algorithm is compared against the pure iterative parallelization algorithm (QMC-PI), which is most commonly used in QMC implementations [64, 65, 66].

5.2 Theory

Because QMC is a Monte Carlo method and thus stochastic in nature, it is one of the easiest algorithms to parallelize and can be scaled to large numbers of processors. In a parallel calculation, an independent QMC calculation is performed on each processor, and the resulting statistics from all the processors are combined to produce the global result.

QMC calculations can typically be broken into two major computationally expensive phases: initialization and statistics gathering. Points distributed with respect to a complicated probability distribution, in this case the square of the wavefunction amplitude, are required during a QMC calculation. In efficient implementations, this is almost always done using the Metropolis algorithm [55].

The first points generated by the Metropolis algorithm are not generated with respect to the desired probability distribution so they must be discarded. Additionally, points generated for diffusion QMC and Green's function QMC must be discarded if there are significant excited state contributions which have not yet decayed. This represents the initialization phase. Once the algorithm begins to generate points with respect to the desired distribution, the points are said to be "equilibrated" and can be used to generate valid statistical information for the QMC calculation. This represents the statistics gathering phase and is the phase where useful data is generated.

To obtain statistically independent data, each processor, in a parallel calculation, must perform its own initialization procedure which is the same length as the initialization procedure on a single processor. When large numbers of processors are used, the fraction of the time devoted to initializing the calculation can be very large and will eventually limit the number of processors that can be effectively used in parallel (Section 5.2.3).

Sections 5.2.1 and 5.2.2 theoretically analyze the pure iterative (QMC-PI) and manager–worker (QMC-MW) parallelization algorithms for QMC. The analyses assume that an $O(\log_2(N_{processors}))$ method, where $N_{processors}$ is the total number of processors, is used to gather the statistical data from all processors and return it to the root processor [50]. To simplify analysis of the algorithms, the analysis is performed for variational QMC (VMC) with the same number of walkers on each processor, but it is possible to extend the results to other QMC methods.

5.2.1 Pure Iterative Parallelization Algorithm

The pure iterative parallelization algorithm (QMC-PI) is the most commonly implemented parallelization algorithm for QMC (Algorithm 5.5) [64, 65, 66]. This algorithm has its origins on homogeneous parallel machines and simply allocates an equal fraction of the total work to each processor. The processors execute their required tasks and percolate the resultant statistics to the root node once every processor has finished its work.

In this algorithm, the number of QMC steps taken by each processor during the statistics gathering phase, $Steps_{PI,i}$, is equal to the total number of QMC steps taken for the calculation, $Steps^{RequiredTotal}$, divided by the total number of processors, $N_{Processors}$.

$$Steps_{PI,i} = \frac{Steps^{RequiredTotal}}{N_{Processors}} \tag{5.1}$$

The number of QMC steps required to initialize each walker during the initialization, $Steps^{Initialize}$, is taken to be a constant. An optimally efficient initialization algorithm would determine how many QMC steps are required to equilibrate each walker, but in current practice, each walker is generally equilibrated for the same number of steps.

The wall clock time required for a QMC calculation using the QMC-PI algorithm, t_{PI}, can be expressed as

$$t_{PI} = t_{PI,i}^{Initialize} + t_{PI,i}^{Propagate} + t_{PI,i}^{Synchronize} + t_{PI}^{Communicate}, \tag{5.2}$$

where $t_{PI,i}^{Initialize}$ is the time required to initialize the calculation on processor i, $t_{PI,i}^{Propagate}$ is the time used in gathering useful statistics on processor i, $t_{PI,i}^{Synchronize}$ is the amount of time processor i has to wait for other processors to complete their tasks, and $t_{PI}^{Communicate}$ is the wall clock time required to communicate all results to the root node. These components can be expressed in terms of quantities that can be measured for each processor and the network connecting them.

$$t_{PI,i}^{Initialize} = N_w(t_i^{GenerateWalker} + Steps^{Initialize} t_i^{QMC}) \tag{5.3}$$

$$t_{PI,i}^{Propagate} = \left(\frac{Steps^{RequiredTotal}}{N_{Processors}}\right) t_i^{QMC} \tag{5.4}$$

$$t_{PI}^{Communicate} = \log_2(N_{Processors})(t^{Latency} + \beta L) \tag{5.5}$$

Here N_w is the number of walkers per processor, $t_i^{GenerateWalker}$ is the time required to construct a walker on processor i, t_i^{QMC} is the time required for a QMC step on processor i, $t^{Latency}$ is the latency of the network, β is the inverse bandwidth of the network, and L is the amount of data being transmitted between pairs of processors when data is percolated to the root node.

The way this algorithm is constructed, all processors must wait for the slowest processor to complete all of its tasks before the program can terminate. Therefore $t_{PI,slowest}^{Synchronize} = 0$, and the wall clock time to complete the QMC-PI calculation is

$$t_{PI} = t_{PI,slowest}^{Initialize} + t_{PI,slowest}^{Propagate} + t_{PI}^{Communicate}. \tag{5.6}$$

Furthermore,

$$t_{PI,i}^{Synchronize} = (t_{PI,slowest}^{Initialize} + t_{PI,slowest}^{Propagate}) - (t_{PI,i}^{Initialize} + t_{PI,i}^{Propagate}). \tag{5.7}$$

Similarly, the total time required for a QMC calculation using the QMC-PI algorithm, T_{PI}, can be expressed as

$$T_{PI} = T_{PI}^{Initialize} + T_{PI}^{Propagate} + T_{PI}^{Synchronize} + T_{PI}^{Communicate}, \tag{5.8}$$

where $T_{PI}^{Initialize}$ is the total time required to initialize the calculation, $T_{PI}^{Propagate}$ is the total time used in gathering useful statistics, $T_{PI}^{Synchronize}$ is the total time used in synchronizing the processors, and $T_{PI}^{Communicate}$ is the total time used to communicate all results to the root node. These components can be expressed in terms of quantities that can be measured for each processor and the network connecting them.

$$T_{PI}^{Initialize} = \sum_{i}^{N_{Processors}} t_{PI,i}^{Initialize} \tag{5.9}$$

$$T_{PI}^{Propagate} = \sum_{i}^{N_{Processors}} t_{PI,i}^{Propagate} \tag{5.10}$$

$$T_{PI}^{Synchronize} = \sum_{i}^{N_{Processors}} t_{PI,i}^{Synchronize} \tag{5.11}$$

$$T_{PI}^{Communicate} = (N_{Processors} - 1)(t^{Latency} + \beta L) \tag{5.12}$$

5.2.2 Manager–Worker-Parallelization Algorithm

The manager–worker paradigm (QMC-MW) offers an entirely new method for performing parallel QMC calculations (Algorithm 5.6). This algorithm makes the root node a "manager" and all of the other nodes "workers." The worker nodes compute until they receive a command from the manager node. The command either tells the worker to 1) percolate its results to the manager node and continue working or 2) percolate its results to the manager node and terminate. The manager periodically collects the statistics that have been calculated. If the statistics are sufficiently converged, the manager commands the workers to send all their data and terminate; otherwise, the manager will do some of its own work and repeat the process again later. Unlike QMC-PI, QMC-MW dynamically determines how much work each processor performs. This allows faster processors to do more work so the calculation is automatically load balanced.

The wall clock time required to perform a QMC-MW calculation can be broken into the same terms as were used for a QMC-PI calculation (Equation 5.3).

$$t_{MW} = t_{MW,i}^{Initialize} + t_{MW,i}^{Propagate} + t_{MW,i}^{Synchronize} + t_{MW,i}^{Communicate} \tag{5.13}$$

Because MW dynamically determines how many steps are performed by each processor, each of the constituent terms has a more complicated form than in QMC-PI. Allowing $\hat{\tau}$ to be the minimum wall clock needed to achieve convergence on a given network and τ to be the approximate wall clock time during the run, one can easily derive the following expressions. Once τ becomes $\hat{\tau}$ the QMC-MW algorithm will terminate.

$$t_{MW,i}^{Initialize} = N_w t_i^{GenerateWalker} + Steps_{MW,i}^{Initialize}(\hat{\tau}) t_i^{QMC} \tag{5.14}$$

$$t_{MW,i}^{Propagate} = Steps_{MW,i}^{Propagate}(\hat{\tau}) t_i^{QMC} \tag{5.15}$$

$$\begin{aligned} t_{MW,i}^{Communicate} = & \left\lceil \frac{Steps_{MW,0}^{Total}(\hat{\tau})}{N_w Steps^{Reduce}} \right\rceil \log_2(N_{Processors})(t^{Latency} + \beta L) + \\ & \left\lceil \frac{Steps_{MW,i}^{Total}(\hat{\tau})}{N_w Steps^{Poll}} \right\rceil t_i^{Poll} \end{aligned} \tag{5.16}$$

$$t_{MW,i}^{Synchronize} \leq N_w Steps^{Poll} t_{slowest}^{Poll} \left\lceil \frac{Steps_{MW,0}^{Total}(\hat{\tau})}{N_w Steps^{Reduce}} \right\rceil, \tag{5.17}$$

where

$$\begin{aligned}\tau &= t_{MW} - \left\lceil \frac{Steps_{MW,0}^{Total}(\tau)}{N_w Steps^{Reduce}} \right\rceil \log_2(N_{Processors})(t^{Latency} + \beta L) \quad (5.18)\\ &\approx t_{MW} - t_{MW,i}^{Synchronize} - t_{MW,i}^{Communicate}\\ &= t_{MW,i}^{Initialize} + t_{MW,i}^{Propagate}\end{aligned}$$

$$Steps_{MW,i}^{Total}(\tau) = \left\lceil \frac{\tau}{N_w t_i^{QMC}} \right\rceil, \tag{5.19}$$

$$Steps_{MW,i}^{Initialize}(\tau) = \min(Steps_{MW,i}^{Total}(\tau), N_w Steps_{Initialize}), \tag{5.20}$$

$$Steps_{MW,i}^{Propagate}(\tau) = Steps_{MW,i}^{Total}(\tau) - Steps_{MW,i}^{Initialize}(\tau), \tag{5.21}$$

and

$$\hat{\tau} = \min \tau \ni \left\{ \begin{array}{c} \sum_i^{N_{Processors}} Steps_{MW,i}^{Propagate}(\tau) \geq Steps^{RequiredTotal} \\ \tau/(Steps^{Reduce} t_0^{QMC}) \in \mathcal{I} \end{array} \right. . \tag{5.22}$$

$Steps^{RequiredTotal}$ is the minimum number of steps that are required to obtain the desired level of convergence, $Steps^{Poll}$ is the number of QMC steps that take place on a worker processor between checking for a message from the manager, and $Steps^{Reduce}$ is the number of QMC steps that take place on the manager processor between sending commands to the workers. Unlike t_{PI}, t_{MW} cannot be simply expressed in terms of individual processor speeds.

The total time required for the MW algorithm, T_{MW}, can be expressed as

$$T_{MW} = T_{MW}^{Initialize} + T_{MW}^{Propagate} + T_{MW}^{Synchronize} + T_{MW}^{Communicate}, \tag{5.23}$$

which contains the same components as Equation 5.9.

$$T_{MW}^{Initialize} = \sum_i^{N_{Processors}} t_{MW,i}^{Initialize} \tag{5.24}$$

$$T_{MW}^{Propagate} = \sum_{i}^{N_{Processors}} t_{MW,i}^{Propagate} \tag{5.25}$$

$$T_{MW}^{Synchronize} = \sum_{i}^{N_{Processors}} t_{MW,i}^{Synchronize} \tag{5.26}$$

$$\begin{aligned} T_{MW}^{Communicate} = & \left\lceil \frac{Steps_{MW,0}^{Total}}{N_w Steps^{Reduce}} \right\rceil (N_{Processors} - 1)(t^{Latency} + \beta L) + \\ & \sum_{i}^{N_{Processors}} \left\lceil \frac{Steps_{MW,i}^{Total}(\hat{\tau})}{N_w Steps^{Poll}} \right\rceil t_i^{Poll} \end{aligned} \tag{5.27}$$

5.2.3 Initialization Catastrophe

QMC algorithms are described as being "embarrassingly parallel" and linearly scaling with respect to the number of processors used. These statements are true for a large fraction of Monte Carlo calculations but are *not* true for QMC calculations which employ the Metropolis algorithm [55]. To obtain independent statistical data from each processor, at least one independent Markov chain must be initialized on each processor (Section 5.2). This gives an initialization cost, $T^{Initialize}$, which scales as $O(N_{Processors})$. The time devoted to generating useful statistical data during the calculation, $T^{Propagate}$, scales as $O(1)$ because a given number of independent Monte Carlo samples are required to obtain a desired statistical accuracy no matter how many processors are used. From this, the efficiency, or fraction of the total calculation time devoted to useful work, ϵ is

$$\epsilon = \frac{T^{Propagate}}{T^{Initialize} + T^{Propagate} + T^{Synchronize} + T^{Communicate}} \tag{5.28}$$

$$\approx \frac{O(1)}{O(N_{Processors}) + O(1)}. \tag{5.29}$$

This result clearly demonstrates that QMC calculations using the Metropolis algorithm as described above are *not* linearly scaling for large numbers of processors as is often claimed. This results from the initialization of the Metropolis algorithm and not the parallelization algorithm used.

We should note that different initialization schemes exist which could potentially reduce the expense of the equilibration phase. If it takes longer to get a guess walker to become equilibrated than it takes to trust that a particular walker has been moved to an uncorrelated configuration from some previously equilibrated configuration, one could make QMC a two-phase algorithm with an initial phase on a single processor which makes uncorrelated configurations from a single equilibrated configuration to start a full QMC run on. This could also be done on multiple processors in a broadcast tree manner where each new uncorrelated configuration seeds a new branch of the tree to generate configurations. Trivially, however, we note that these and any algorithm, which requires the generation of an uncorrelated set of walkers requires computational effort which grows linearly with the number of total global walkers. Therefore, we will continue to analyze the current method of generating each walker from a guess configuration which we manually equilibrate since the total computational complexity will be the same.

5.3 Experiment

Computational experiments comparing QMC-PI and QMC-MW parallelization algorithms were performed using *QMcBeaver* [54, 50], a finite all-electron QMC software package we developed. Variational QMC was chosen as the particular QMC flavor to allow direct comparison with the theoretical results in Section 5.2.

QMcBeaver percolates statistical results from all nodes to the root node using the Dynamic Distributable Decorrelation Algorithm (DDDA) [50] and the *MPI_Reduce* command from MPI [67]. This combination provides an $O(\log_2(N_{Processors}))$ method for gathering the statistical data from all processors, decorrelating the statistical data, and returning it to the root node.

The time spent initializing, propagating, synchronizing, and communicating during a calculation was obtained from timers inserted into the relevant sections of *QMcBeaver*. During a parallel calculation, each node has its own set of timers which provide information on how that particular processor is performing. At the completion of a calculation, the results from all processors are combined to yield the total CPU time devoted to each class of task.

5.3.1 Experiment: Varying Levels of Heterogeneity

For this experiment, a combination of Intel Pentium Pro 200 MHz and Intel Pentium III 866 MHz computers connected with a 100 Mb/sec network was used. The total number of processors was kept constant at 8, but the number of each type of processor was varied over the whole range. This setup provided a series of 8 processor parallel computers with a spectrum of heterogeneous configurations. For our calculations with the current version of *QMcBeaver*, the Pentium III is roughly 4.4 times faster than the Pentium Pro at performing QMcBeaver on these test systems.

The Ne atom was the particular chemical system the computational experiments were performed on. A Hartree-Fock/TZV [1] wavefunction calculated using GAMESS [68, 56] was used as the trial wavefunction. For the parallelization algorithms, the following values were used: $Steps^{RequiredTotal} = 2.5 \times 10^6$, $Steps^{Initialize} = 1 \times 10^3$, $Steps^{Poll} = 1$, $Steps^{Reduce} = 1 \times 10^3$, and

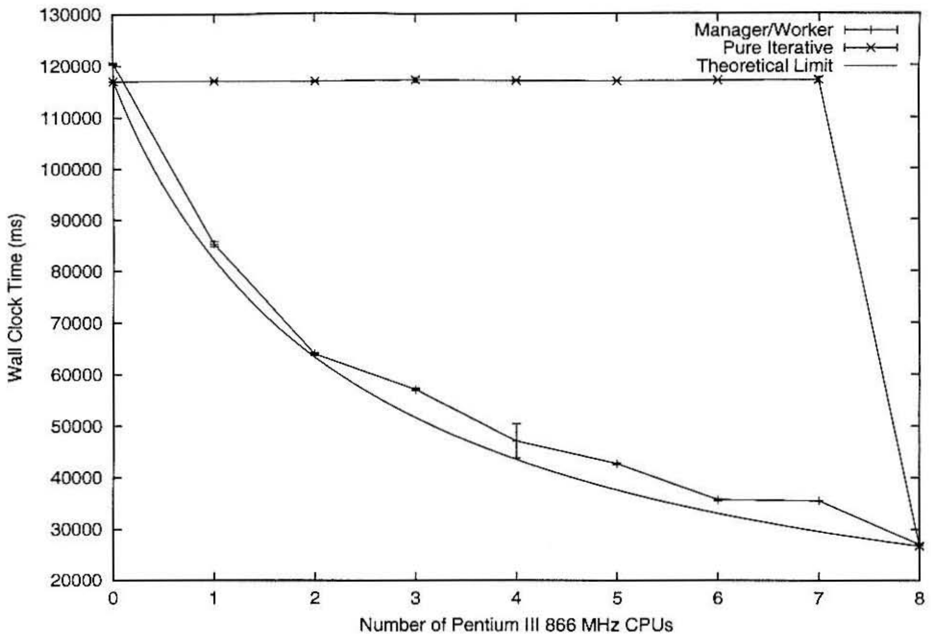


Figure 5.1: Time required to complete an 8 processor variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking. The theoretical optimal performance for a given configuration of processors is provided by the curve.

$N_w = 2$.

The time required to complete the QMC calculation for the QMC-PI and QMC-MW parallelization algorithms is shown in Figure 5.1. Each data point was calculated five times and averaged to provide statistically relevant data.

One should note $Steps^{RequiredTotal}$ is not known before a calculation. Therefore, the QMC-MW model here is very representative of a real-world implementation with its dynamic termination. However, allowing someone using the QMC-PI method to know exactly $Steps^{RequiredTotal}$ before a calculation begins is the best-case scenario. Typically, one either under or overestimates $Steps^{RequiredTotal}$ when using the QMC-PI method. If one overestimates $Steps^{RequiredTotal}$ some amount of computational resources will be wasted over converging the calculation. If one underestimates $Steps^{RequiredTotal}$, the job must be resubmitted to the queue with its last checkpoint state file. Both cases waste the user's time and/or computational resources.

The time required for the QMC-PI algorithm to complete is determined by the slowest processor. When between 1 and 8 Pentium Pro processors are used, the calculation takes the same time as when 8 Pentium Pro processors are used; yet, when 8 Pentium III processors are used (homogeneous network), the calculation completes much faster. This matches the behavior predicted by Equation 5.6. This figure also shows that MW performs near the theoretical speed limit for each of the heterogeneous configurations. This is a result of the dynamic load balancing inherent in QMC-MW.

The total number of QMC steps performed during a calculation is shown in Figure 5.2. The QMC-PI method executes the same number of steps regardless of the particular network because the number of steps performed by each processor is determined *a priori*. On the other hand, QMC-MW executes a different number of steps for each network configuration. This results from the dynamic determination of the number of steps performed by each processor. The total number of steps is always greater than or equal to the number of steps needed to obtain a desired precision, $Steps^{RequiredTotal}$.

Figures 5.3 and 5.4 break the total calculation time down into its constituent components (Equations 5.8 and 5.23). QMC-MW spends essentially all of its time initializing walkers or generating useful QMC data. Synchronization and communication costs are minimal. On the other hand, QMC-PI devotes a huge portion of the total calculation time to synchronizing proces-

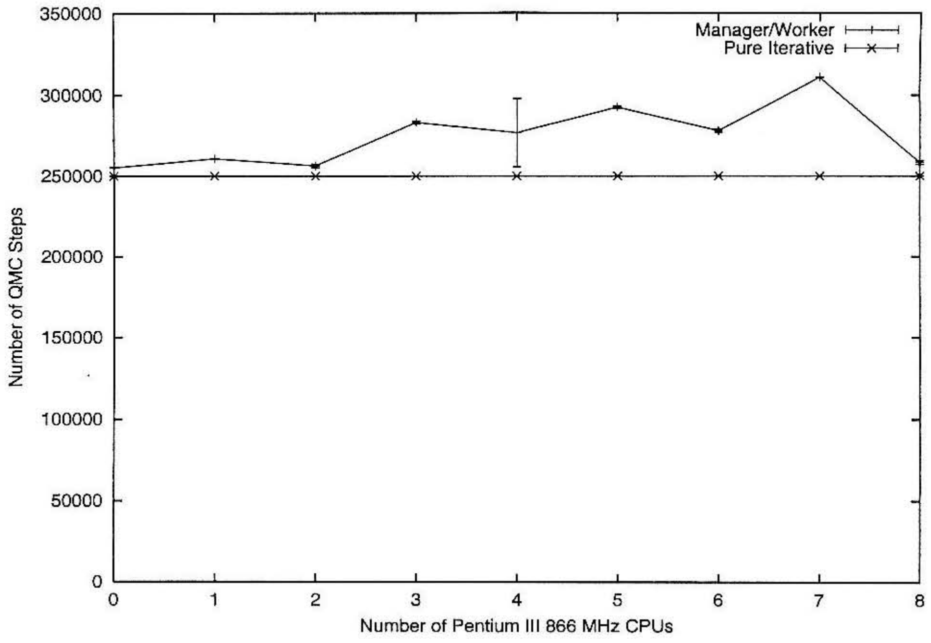


Figure 5.2: Number of variational QMC steps completed during an 8 processor calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) parallelization algorithms. The pure iterative algorithm always calculates the same number of steps, but the manager–worker algorithm dynamically determines how many steps to take. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking.

sors on heterogeneous networks. This is very inefficient and wasteful.

5.3.2 Experiment: Heterogeneous Network Size

The Ne atom was the particular chemical system the computational experiments were performed on. A Hartree-Fock/TZV [1] wavefunction calculated using GAMESS [68, 56] was used as the trial function. The network of machines used was a heterogeneous cluster of linux boxes. A unit of five machines goes as follows. Three different sized networks were ex-

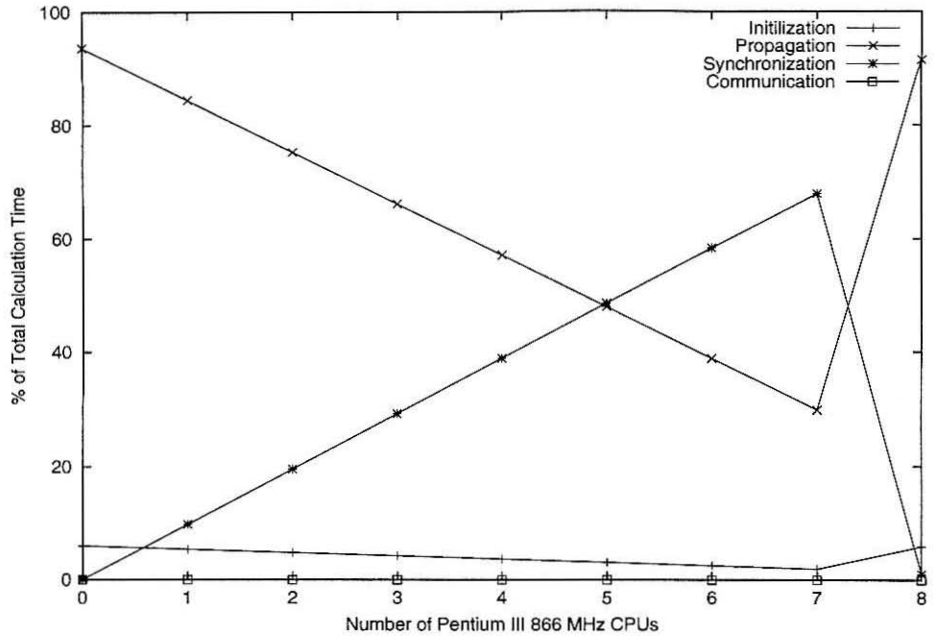


Figure 5.3: Percentage of total calculation time devoted to each component in the pure iterative parallelization algorithm (QMC-PI) during an 8 processor variational QMC calculation of Ne. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking.

amined each with either one, two, or four of each of these respective processors. For the parallelization algorithms, the following values were used: $Steps^{RequiredTotal} = 2.5\times10^6$, $Steps^{Initialize} = 1\times10^3$, $Steps^{Poll} = 1$, $Steps^{Reduce} = 1 \times 10^3$, and $N_w = 2$.

- Intel Pentium Pro 200 MHz
- Intel Pentium II 450 MHz
- Intel Pentium III Xeon 550MHz
- Intel Pentium III 600 MHz

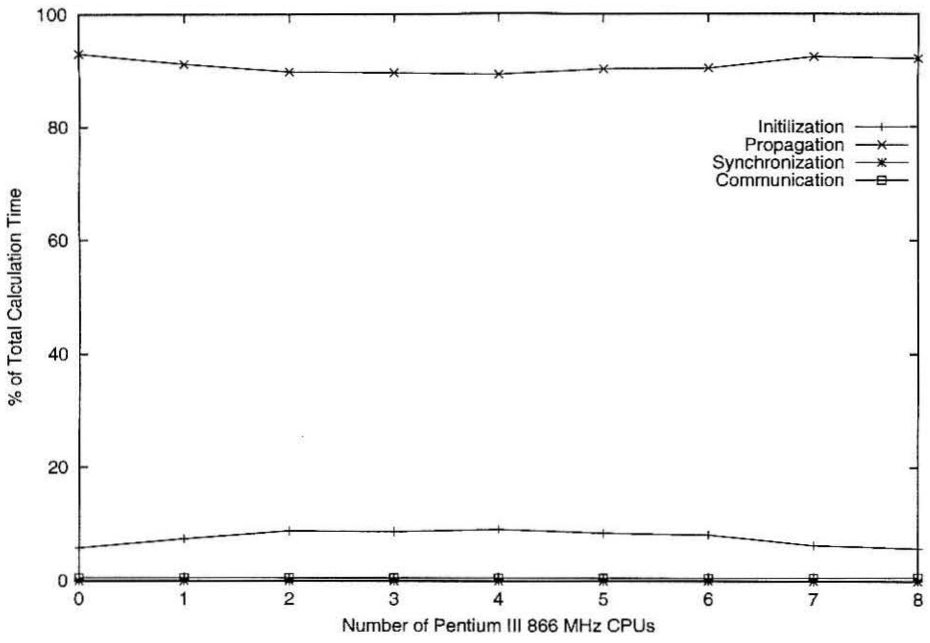


Figure 5.4: Percentage of total calculation time devoted to each component in the manager–worker-parallelization algorithm (QMC-MW) during an 8 processor variational QMC calculation of Ne. The 8 processors are a mixture of Pentium Pro 200 MHz and Pentium III 866 MHz Intel processors connected by 100 Mb/s networking.

- Intel Pentium III 866 MHz

Implementing QMC-PI and QMC-MW exactly as was done in Section 5.3.1 for this network we observe the results in Figure 5.5. This shows that even as the network size increases, the QMC-MW model does an excellent job of running near the theoretical optimal time for this network. However, the QMC-PI method struggles to compete.

We, of course, could improve the efficiency of the QMC-PI method if we knew the machine was devoted to our QMC-PI program and we had previously bench-marked the QMC job on each machine. However, this would require the effort of bench-marking, trusting that the machine is truly de-

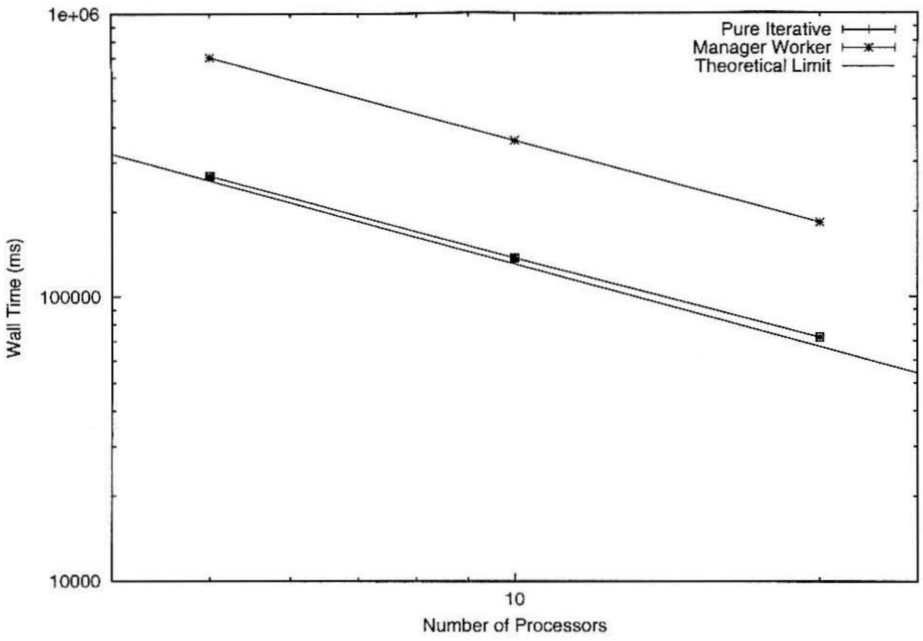


Figure 5.5: Wall time required to complete a variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms on a heterogeneous linux cluster. The theoretical optimal performance for a given configuration of processors is provided by the line.

voted to our task, and the extra bookkeeping needed to match up the number of tasks with each machine's predicted effectiveness. This all could be accomplished with no assumptions on the network by simply implementing the QMC-MW method which already pushes the boundary of perfect efficiency.

5.3.3 Experiment: Homogeneous Network

The QMC-PI algorithm was originally designed to work on homogeneous supercomputers with fast communication while the QMC-MW algorithm was designed to work on heterogeneous supercomputers with slow communication. To test the QMC-MW algorithm on the QMC-PI algorithm's native

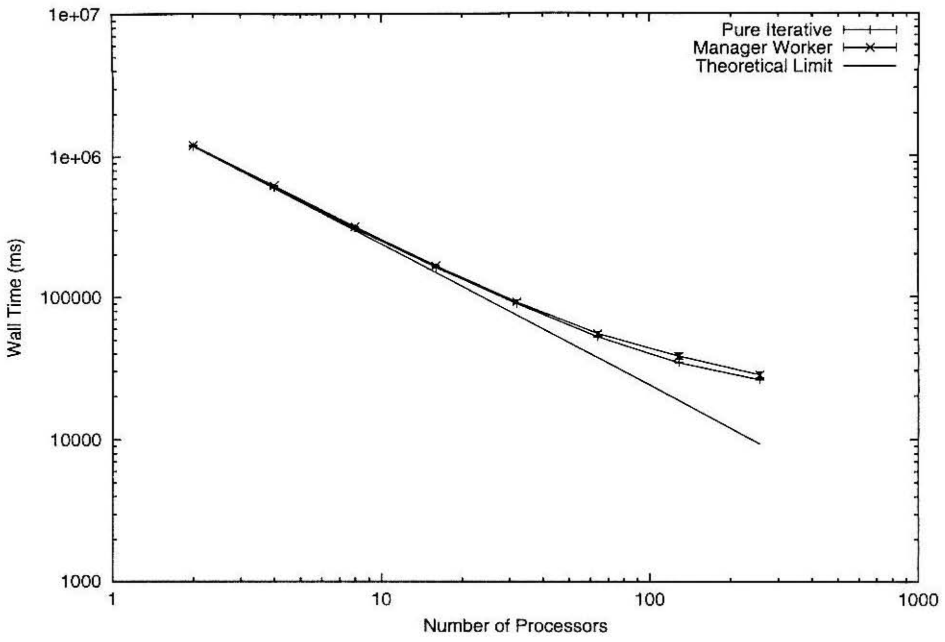


Figure 5.6: Wall time required to complete a variational QMC calculation of Ne using the manager–worker (QMC-MW) and pure iterative (QMC-PI) algorithms on the ASCI Blue Pacific homogeneous supercomputer. The theoretical optimal performance for a given configuration of processors is provided by the line.

architecture, a QMC scaling calculation (Figure 5.6) was performed on the ASCI-Blue Pacific supercomputer at Lawrence Livermore National Laboratory. This machine is a homogeneous supercomputer composed of 332 MHz PowerPC 604e processors connected by HIPPI networking.

Ne atom was the particular chemical system the computational experiments were performed on. A Hartree-Fock/TZV [1] wavefunction calculated using GAMESS [68, 56] was used as the trial function. For the parallelization algorithms, the following values were used: $Steps^{RequiredTotal} = 1 \times 10^6$, $Steps^{Initialize} = 2 \times 10^3$, $Steps^{Poll} = 1$, $Steps^{Reduce} = 1 \times 10^3$, and $N_w = 2$.

Figure 5.6 shows that the QMC-MW and QMC-PI algorithms perform

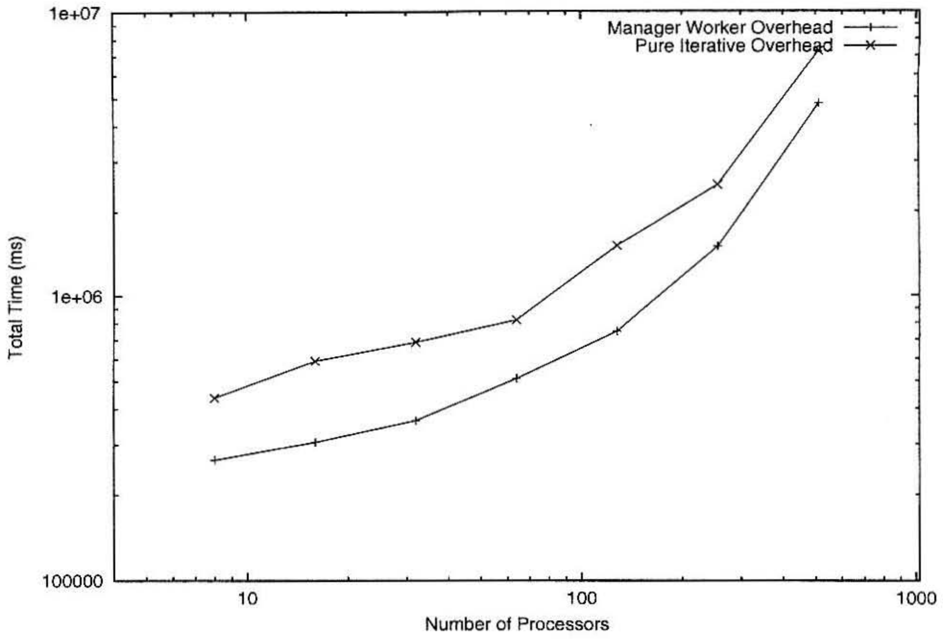


Figure 5.7: Wall time in nonpropagation and non-initialization overhead expenses for QMC-PI and QMC-MW on ASCI Blue Pacific.

nearly identically on Blue Pacific. The QMC-MW calculation is consistently slightly slower than the QMC-PI algorithm because the QMC-MW calculation performed more QMC steps. This results because the QMC-PI calculation performs a predetermined number of steps while the QMC-MW calculation performs at least a predetermined number of steps. The discrepancy can be reduced by decreasing $Steps^{Reduce}$.

Two useful figures show how these two methods really differ. Observing the overhead expense (all nonpropagation or initialization clock time) for running both methods we observe that the QMC-PI actually has a slightly higher overhead expense than QMC-MW in Figure 5.7. (The growth of both of these for large numbers of processors is relic of the *Initialization Catastrophe* 5.3.4.)

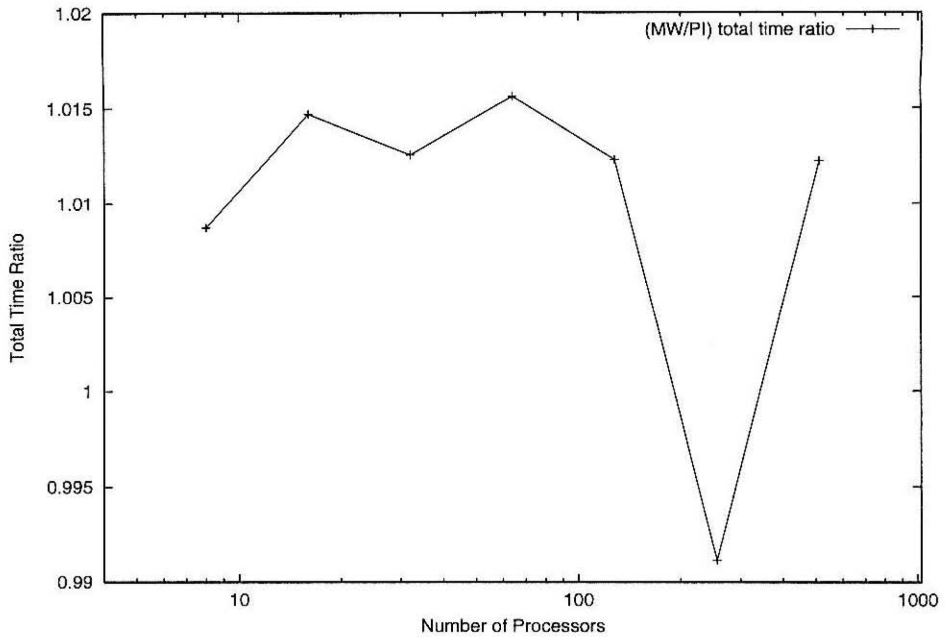


Figure 5.8: Ratio of wall time for QMC-MW/QMC-PI on ASCI Blue Pacific.

If one observes the total computational resources used over a given time and takes a ratio of the two methods total run time, we observe (Figure 5.8) that both methods use roughly the same amount of resources. Since they are within a couple of percent of each other, they can be considered to take roughly the same time and expense on this homogeneous machine.

To resolve the seemingly contradictory results from Figures 5.7 and 5.8, we must remember than the QMC-MW method may actually do more QMC steps than the QMC-PI in these experiments. This shows that even if one can exactly guess the correct number of QMC steps needed to converge a given QMC-PI run, both QMC-PI and QMC-MW perform roughly the same with respect to wall clock. However, in reality, rarely does the user know how many steps they should require and the QMC-PI will perform poorly compared to this idealized result whereas the QMC-MW will always perform near this level since it dynamically determines convergence and termination.

Both algorithms do not perform near the linear scaling limit for large numbers of processors. This is a result of the initialization catastrophe discussed in Sections 5.2.3 and 5.3.4.

5.3.4 Experiment: Initialization Catastrophe

To demonstrate the "initialization catastrophe" described in Section 5.2.3, a scaling experiment was performed on the ASCI-Blue Mountain supercomputer at Los Alamos National Laboratory (Figure 5.9). This machine is a homogeneous supercomputer composed of MIPS 10000 processors running at 250 MHz connected by HIPPI networking. Variational QMC calculations of RDX, cyclic-$[CH_2NNO_2]$, using the QMC-MW algorithm with $Steps^{RequiredTotal} = 1\times10^5$, $Steps^{Initialize} = 1\times10^3$, $Steps^{Poll} = 1$, $Steps^{Reduce} = 1 \times 10^2$, and $N_w = 1$ were performed. Jaguar 4.0 [1] was used to generate a HF/6-31G** trial wavefunction.

The efficiency of the scaling experiments were calculated using Equation 5.28, and the results were fit to

$$\epsilon = \frac{a}{a + N_{Processors}} \tag{5.30}$$

with $a = 104.203$. The efficiency at 2048 processors is better than the value predicted from the fit equation. This is an artifact of the QMC-MW algorithm which resulted from this calculation taking significantly more steps than $Steps^{RequiredTotal}$. Decreasing the value of $Steps^{Reduce}$ would reduce this problem.

The excellent fit of the data to Equation 5.30 clearly shows that QMC calculations using the Metropolis algorithm are *not* linearly scaling for large

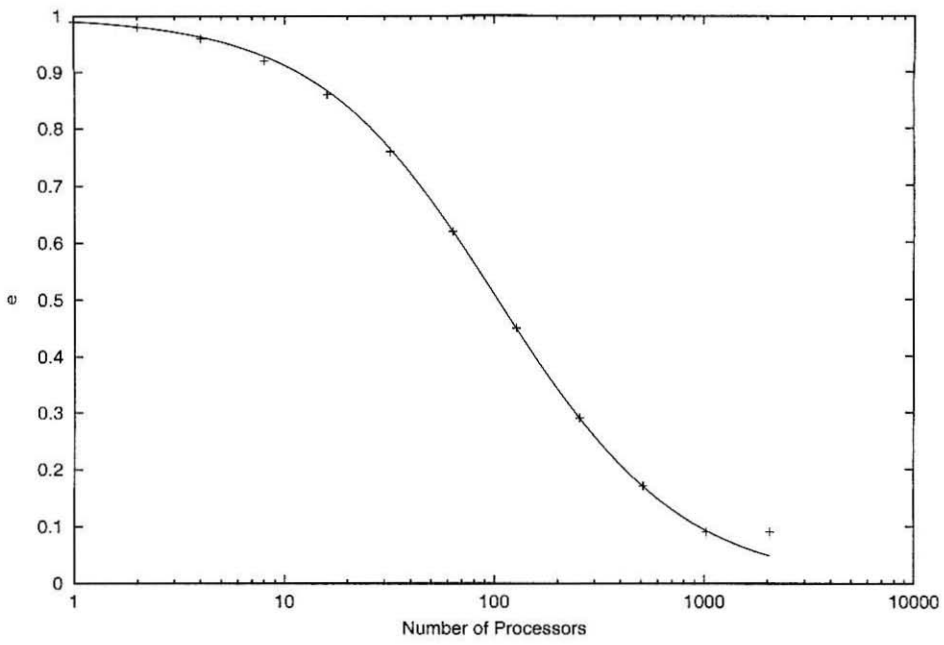


Figure 5.9: Efficiency of a variational QMC calculation of RDX as a function of the number of processors used. The calculations were performed using the manager–worker-parallelization algorithm (QMC-MW) on the ASCI-Blue Mountain supercomputer, which has 250 MHz MIPS 10000 processors connected by HIPPI networking. A similar result is produced by the Pure Iterative parallelization algorithm. The data is fit to $\epsilon(N_{Processors}) = a/(a + N_{Processors})$ with $a = 104.203$.

numbers of processors. This result holds true for both QMC-MW and QMC-PI because it results from the initialization of the Metropolis algorithm and not the parallelization of the statistics gathering propagation phase. Furthermore, longer statistics gathering calculations have better efficiencies and thus better scaling than short statistics gathering calculations. This can be seen by examining Equation 5.28.

5.4 Conclusion

The new QMC manager–worker-parallelization algorithm clearly outperforms the commonly used Pure Iterative parallelization algorithm on heterogeneous parallel computers and performs near the theoretical speed limit. Furthermore, both algorithms perform essentially equally well on a homogeneous supercomputer with high speed networking.

When combined with DDDA, QMC-MW is able to determine, "on-the-fly," how well a calculation is converging, allowing convergence-based termination. This is opposed to the standard practice of having QMC calculations run for a predefined number of steps. If the predefined number of steps is too long, computer time is wasted, and if too short, the job will not have the required convergence and must be resubmitted to the queue lengthening the total time for the calculation to complete. Additionally, specifying a calculation precision (2 kcal/mol for example) is more natural for the applications user than specifying a number of QMC steps.

QMC-MW allows *very* low cost QMC specific parallel computers to be built. These machines can use commodity processors, commodity networking, and no hard disks. Because the algorithm efficiently handles loosely coupled heterogeneous machines, such a computer is continuously upgradeable and can have new nodes added as resources become available. This greatly reduces the cost of the resources the average practitioner needs access to, bringing QMC closer to becoming a mainstream method.

It is possible to use QMC-PI on a heterogeneous computer with good efficiency if the speed of each processor is known. Determining and effectively using this information can be a great deal of work. If the user has little or inaccurate information about the computer, this approach will fail. QMC-MW overcomes these shortfalls with no work or input on the users part. Also, when new nodes are added to the computer, QMC-MW can immediately take advantage of them while the modified QMC-PI must have benchmark information recorded before they can be efficiently used. The benefits and displayed ease of implementation of QMC-MW clearly outweigh those of QMC-PI supporting its adoption as the method of choice for making QMC parallel.

The prediction and verification of the initialization catastrophe clearly highlights the need for efficient initialization schemes if QMC is to be scaled to tens of thousands or more processors. Producing such algorithms must be a focus of future work.

5.5 Pure Iterative Algorithm (QMC-PI)

```
for $Processor_i$; $i = 0$ to $N_{Processors} - 1$
    $Steps_{PI,i} = Steps^{RequiredTotal} / N_{Processors}$
    Generate $N_w$ walkers
    for $Steps^{Initilize}$ steps
        Equilibrate walkers
    for $Steps_{PI,i}$ steps
        Generate QMC statistics
    Percolate statistics to $Processor_0$
```

5.6 Manager–Worker Algorithm (QMC-MW)

```
for $Processor_i$; $i = 0$ to $N_{Processors} - 1$
    $done = false$
    $counter = 0$
    Generate $N_w$ walkers

    while not $done$:
        if $counter < Steps^{Initialize}$:
            Equilibrate all local walkers 1 step
        else:
            Propagate all local walkers 1 step and collect QMC statistics

        if $i = 0$:
            if statistics are converged:
                $done = true$
                Tell workers to percolate statistics to $Processor_0$ and
                set $done = true$
            else if $counter \mod Steps^{Reduce} = 0$:
                Tell workers to percolate statistics to $Processor_0$
        else:
            if $counter \mod Steps^{Poll} = 0$:
                Check for commands from the manager and
                execute the commands.

    $counter = counter + 1$
```

Chapter 6

Generic Jastrow Functions for Quantum Monte Carlo Calculations on Hydrocarbons

6.1 Introduction

Quantum Monte Carlo (QMC) is becoming a very important member of the electron correlation correction methods in quantum chemistry. Many flavors of QMC exist while Variational (VMC, 6.2.1) and Diffusion (DMC, 6.2.2) Quantum Monte Carlo are two of the more popular methods employed. VMC requires the explicit use of a variational wavefunction while DMC has the property that it can sample the ground state fixed node solution for a given trial wavefunction.

Experience and tradition have defined a fairly efficient method of obtaining very accurate calculations for molecules and materials using QMC [10, 13, 14, 15, 16, 18, 19, 20, 21, 22]. This protocol follows:

1. Obtain a fair trial wavefunction, Ψ_{Trial}, from some quantum mechanical method, like Density Functional Theory (DFT) or Hartree Fock (HF).
2. Guess Jastrow particle-particle correlation functions that have some variational form which maintains the antisymmetry of the total wavefunction. (This may only be a *nearly* antisymmetric wavefunction. Umrigar gives a discussion of this topic [13].)
3. Choose variational parameters such that any Hamiltonian singularities are satisfied with the "cusp condition" in the Jastrow form.
4. Generate an initial "walker(s)" approximately with respect to the particle probability distribution.
5. Equilibrate this "walker(s)" to verify it represents the particle probability distribution.
6. Generate configurations with the Metropolis algorithm in a VMC run.
7. Perturb and evaluate the Jastrow parameters using these configurations. (Repeat this correlated sampling optimization [10] until satisfactory convergence.)
8. Generate (or reuse from a VMC run) initial "walkers" for a DMC run.
9. Equilibrate these "walkers" to verify they represent the proper particle probability distribution.
10. Use the optimized Jastrow for a DMC run to obtain a very accurate result.

Typically the equilibration and generation of the configurations in the VMC and DMC runs are the most expensive parts of this protocol so one would like to minimize the effort in these sections. The main purpose of the VMC optimization phase is to obtain a good description of the wavefunction. The better this wavefunction is, the quicker the DMC run will converge. This motivates a very well optimized Jastrow but not at the expense of marginal returns.

Experience has shown that the VMC Jastrow optimization involves a very difficult objective function. What one should generally try to do is reduce the energy and/or variance a fair amount but not try to overoptimize. The method of correlated sampling is a useful method of optimization yet once it finds a flat region of the objective function (typically a $\sigma^2(E_{Local})$ based objective function), it can falsely encourage you to overoptimize since you have already likely reached a point of diminishing returns. Experience has shown that if you can get roughly a factor of three reduction in the variance over the HF wavefunction alone, you have done a fair job of optimizing and that further optimization may give only marginal returns. Typically, one might spend 5% to 50% of the one's total effort optimizing the Jastrow in the VMC phase of the calculation.

6.2 Theory

QMC has many flavors each with certain assets and liabilities. The two particular types of QMC we will examine are VMC (Section 6.2.1) and DMC (Section 6.2.2). These two methods are widely used and in general use for production level calculations. Any impact we can make to improve the speed at which one can accomplish these two types of QMC will have far-reaching consequences for many researchers in computational chemistry and materials science.

6.2.1 Variational Quantum Monte Carlo

Variational Quantum Monte Carlo (VMC) is a very simple yet powerful method for examining correlated quantum wavefunctions. If one examines the basic energy expectation integral and reformulates it in terms of an electron probability density, ρ, and a local energy, E_{local}, one finds a very simple description of the energy expectation (6.1).

$$\begin{aligned} \langle E \rangle &= \int \Psi(\vec{x}) \hat{H} \Psi(\vec{x}) dx^{3n} \\ &= \int (\Psi(\vec{x}))^2 \left(\frac{\hat{H}\Psi(\vec{x})}{\Psi(\vec{x})} \right) dx^{3n} \\ &= \int \rho(\vec{x}) E_{local}(\vec{x}) dx^{3n} \end{aligned} \tag{6.1}$$

We must now determine what this Ψ should be. Typically, we can use a method like Hartree Fock theory or Density Functional Theory [68, 56, 57, 58, 59, 1, 3, 4, 5, 6, 7, 8] to obtain an antisymmetric wavefunction in a determinant form.

This wavefunction is then augmented with a product of symmetric terms which contain the explicit particle correlations. These particle correlation functions will allow each particle to observe the positions of their neighboring particles and will allow addition variational freedom in the wavefunction.

To construct the entire trial wavefunction, Ψ_{Trial}, from a HF type type initial guess wavefunction, Ψ_{HF}, involves the use of the following expression

(6.2). A Ψ_{Trial} constructed from a DFT type wavefunction is similar.

$$\Psi_{Trial} = \Psi_{HF} e^{\sum_i^n \sum_{j<i}^n u_{ij}} \tag{6.2}$$

The building unit of this type of description is a u_{ij} function for particles i and j which are of particle types A and B, respectively (6.3).

$$u_{ij} = \frac{cusp_{AB} r_{ij} + a_{AB} r_{ij}^2 + \cdots}{1 + b_{AB} r_{ij} + c_{AB} r_{ij}^2 + \cdots} \tag{6.3}$$

This particular form of u_{ij} is commonly referred to as the Padé-Jastrow for finite systems [13]. $cusp_{AB}$ removes singularities which arise as two charged particles approach each other.

We must now determine how to optimize the parameters in the u_{ij} functions as well as determining how many parameters to maintain in the expression. Allowing only the cusp condition parameter in the numerator and the first parameter in the denominator is common practice, though the more parameters we optimizes the better the result will likely be with the additional variational freedom. The common optimization procedure is the method of correlated sampling optimization described by Umrigar [10].

6.2.2 Diffusion Quantum Monte Carlo

Examining the time-dependent Schrödinger equation (6.4) in atomic units we observe that one can make a transformation from real time into imaginary time to produce a diffusion equation (6.6).

$$i\frac{\partial \Psi}{\partial t} = \hat{H}\Psi \tag{6.4}$$

$$t = -i\tau \tag{6.5}$$

$$\frac{\partial \Psi}{\partial \tau} = -\hat{H}\Psi = \left(\frac{1}{2}\nabla^2 - V\right)\Psi \tag{6.6}$$

Techniques exist which allow one to sample the ground state with respect to the original Ψ_{Trial} nodes. Many excellent in-depth descriptions of this method exist [20, 11, 34, 48].

6.3 Experiment

6.3.1 Motivate Generic Jastrow for Hydrocarbons

A significant part of the computational expense from taking a task from conception to completion using QMC is to optimize the Jastrow parameters with correlated sampled VMC. This resulting optimized wavefunction is then generally a good starting wavefunction for DMC. We would like to minimize the time spent in these expensive parts of the program to make the QMC method faster and cheaper.

Breaking with traditional methods, we searched for physically motivated parameters for the Jastrows. The dominant part of the electron correlation we wish to regain is thought to be in the spatially similar electrons which do not actively avoid each other. The parallel spin electrons do avoid each other by being described by a determinant which goes to zero as two parallel spin electrons approach each other. Experience has also shown that the most fruitful particle-particle interactions will likely be in opposite spin electrons which are not correlated with the determinant description of the wavefunction.

For the other particle-particle interactions, we have found the original wavefunction does a fair job of describing their interactions and that any $b_{\uparrow\downarrow}$ relaxation will not need too large of a relaxation of the $b_{\uparrow,\uparrow}$,$b_{\downarrow,\downarrow}$,$b_{\uparrow,H}$,$b_{\downarrow,H}$,$b_{\uparrow,C}$, and $b_{\downarrow,C}$ parameters in a hydrocarbon type molecule (6.3.2). Therefore, a b parameter of 100 was chosen for these Jastrow parameters which allows the cusp condition to satisfy the removal of the singularities but makes the u functions have a very short range effect. The opposite spin electrons will have a free $b_{\uparrow\downarrow}$ parameter which we will examine in the computational experiments.

$$u_{\uparrow\downarrow} = \frac{\frac{1}{2}r_{ij}}{1 + b_{\uparrow\downarrow}r_{ij}} \tag{6.7}$$

$$u_{\uparrow\uparrow} = u_{\downarrow\downarrow} = \frac{\frac{1}{4}r_{ij}}{1 + 100r_{ij}} \tag{6.8}$$

$$u_{\uparrow,H} = u_{\downarrow,H} = \frac{-r_{ij}}{1 + 100r_{ij}} \tag{6.9}$$

$$u_{\uparrow,C} = u_{\downarrow,C} = \frac{-6r_{ij}}{1 + 100r_{ij}} \tag{6.10}$$

6.3.2 Experiment: Hydrocarbons Test Set

Several types of hydrocarbons are examined. Simple single-bonded systems, double-bonded, triple-bonded, and π - conjugated systems are examined. A complete list follows:

- benzene

- trans-butadiene
- cis-butadiene
- ethylene
- ethane
- allene
- acetylene
- methane

Optimal geometries and wavefunctions were obtained using HF/6-31G** theory and the Jaguar quantum mechanical program suite [1]. The use of HF wavefunctions is attractive since it gives a variational bound on the energy expectation which we must strive to improve. This is very useful in determining if a set of GJ parameters is doing a good job of describing the system. (We will refer to a Hartree Fock wavefunction with the Generic Jastrow type correlation function as a HF-GJ type wavefunction.)

The results of varying the $b_{\uparrow\downarrow}$ parameter for these simple hydrocarbons are shown in the following figures (Figures 6.1 and 6.2). Figure 6.1 shows the correlation energy gained with the use of the HF-GJ over the HF energy. This result is scaled by the total charge on the nuclei to give a consistent correlation energy gained per carbon and hydrogen and a consistent minimum for all hydrocarbons in the range of 2.0 to 4.0.

Figure 6.2 shows the remaining variance after implementing the GJ. This is the ratio of the variance in the energy estimator from a pure HF only type wavefunction over the variance in the energy estimator of the HF-GJ type

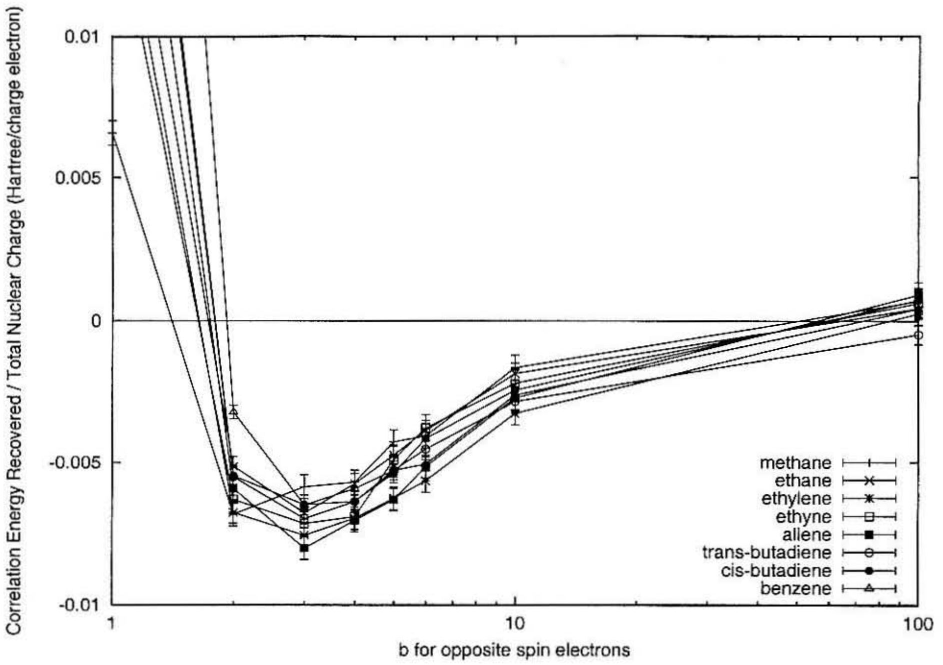


Figure 6.1: Correlation energy (Hartree) recovered divided by total nuclear charge.

wavefunction for various values of the $b_{\uparrow\downarrow}$ parameter. We observe a dramatic reduction of the variance for the VMC runs and a consistent minimum again in the range of 2.0 to 4.0.

The resulting Generic Jastrow for these systems is the form given in equations (Equations 6.7, 6.8, 6.9, and 6.10) with $b_{\uparrow\downarrow} = 3.0$. These functions are plotted in Figure 6.3. What we notice is that all the correlation functions have a fairly short range while the opposite spin correlation function has a longer range.

6.3.3 Generic Jastrow for DMC

To demonstrate the utility of the GJ parameters for DMC, we ran methane and acetylene with various Jastrow parameters. A HF only type wavefunction

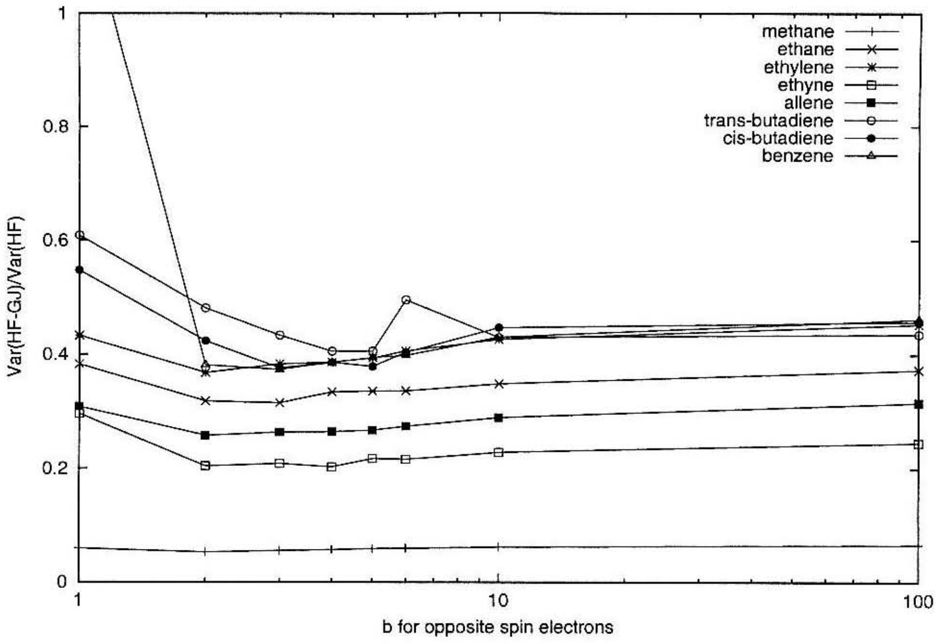


Figure 6.2: Reduction of the QMC variance for a wavefunction containing a Generic Jastrow compared to a Hartree-Fock wavefunction.

was used in a DMC run and was extremely unstable. The other four sets of parameters resulted in stable DMC runs. Figure 6.4 shows that the GJ does an outstanding job of reducing the variance in the DMC calculation while the other $b_{\uparrow\downarrow}$ do an inferior job. We also notice that the VMC variance optimized wavefunction does not even match the GJ performance. The optimization procedure used was a variance optimization as described by Umrigar [10] with 4000 statistically independent configurations. This implies that for this methane wavefunction that the optimization procedure actually resulted in a slightly worse wavefunction than the GJ wavefunction. This is possible since the objective function being optimized and gradients on the objective function are inherently inaccurate because of a finite VMC sampling in the correlated sampling procedure.

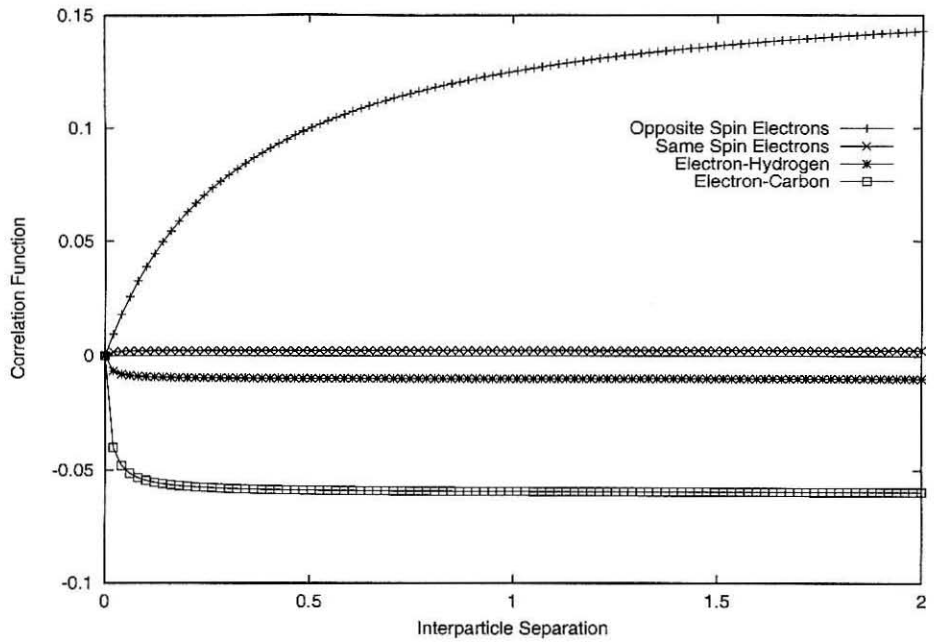


Figure 6.3: Generic Jastrow correlation functions. $b_{\uparrow,\downarrow} = 3.0$

Figure 6.5 shows that the GJ again does a great job at reducing the variance in the DMC calculation and nearly matches the VMC variance optimized wavefunction. In this case the optimized parameters did result in the nearly best DMC variance yet its improvement over the GJ parameter set was negligible.

What this shows is the Generic Jastrow does a very good job at accelerating the convergence of a DMC run. It appears that the generic form proposed for all hydrocarbons is very near the optimal for these methane and acetylene test cases.

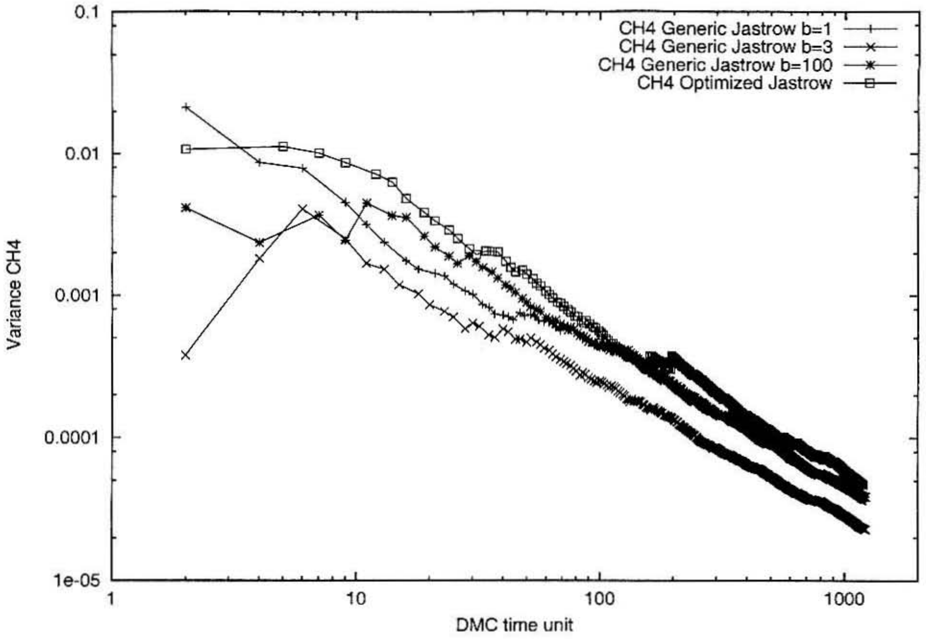


Figure 6.4: Convergence (σ^2 in Hartree^2 units) of methane DMC runs over time for various wavefunctions

6.3.4 Test Case: 10-Annulene

To test the transferability of these generic Jastrow parameters, we examined two different conformations of 10-Annulene studied by Scheafer's group [69]. This molecule has some interesting electron correlation issues and provides a slightly larger yet interesting test case for the proposed Generic Jastrow.

The HF wavefunction as well as the HF-GJ type wavefunction are examined. The HF energies are from Jaguar[1] while the VMC(HF-GJ) results are from QMcBeaver[54].

Comparing the results from Table 6.1 with the results in basic hydrocarbons in Figure 6.1, we notice a consistent correlation energy gain per atom.

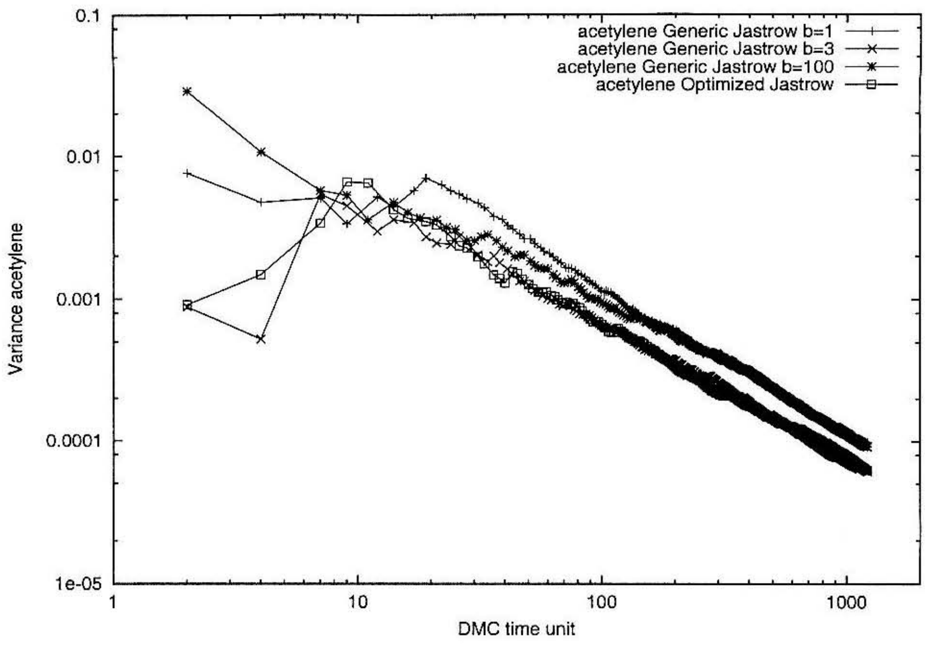


Figure 6.5: Convergence (σ^2 in Hartree^2 units) of acetylene DMC runs over time for various wavefunctions

6.4 Conclusion

The idea of using a GJ parameter set is fairly crude, yet we show that the VMC energy and variance are reduced over a purely HF wavefunction in each case. The clearest impact of this work is on the initial guess Jastrow used in the VMC optimization procedure. Using a generic Jastrow as an initial guess allows these generally good parameter sets to be further optimized to meet the user's needs.

If further work shows these generic Jastrows provide good enough starting points for DMC, the VMC optimization procedure's expense may not only be reduced but may be eliminated. This is a significant fraction of the computational expense which may be saved.

conformation	E_{HF}	$E_{VMC,GenericJastrow}$	ΔE	$\frac{\Delta E}{totalZ}$
napth	-383.07	-383.46 $\pm$ 0.03	0.39	0.0056
twist	-383.06	-383.48 $\pm$ 0.03	0.42	0.0060

Table 6.1: Absolute energies (Hartree) for various conformations of 10-annulene methods with and without explicit electron correlation from the Generic Jastrow (basis: cc-pVDZ).

We grant that for all-electron calculations the majority of the correlation gotten by these methods is core electron correlation and will be very transferable between similar species. This is clearly seen in the various types of hydrocarbons including the larger test case of 10-annulene. This is important information for these types of QMC calculations which have proven to be very difficult in the past.

This work supports further studies to find trends in optimal Jastrow parameters. A database may be formed which could allow for good initial guess Jastrow parameters for QMC calculations.

Chapter 7

Aminomethanol Water Elimination: Theoretical Examination

7.1 Introduction

Presently there are over one hundred known interstellar molecules, the great majority of which are organic [70]. Theoretical models of grain surface chemistry predict precursors to the more complex compounds, such as simple alcohols and aminoalcohols [71, 72, 73]. Many potential grain surface reaction pathways are eliminated by the conditions imposed on these models, greatly simplifying the possible products of grain synthesis and eliminating the possibility for much larger organics to form on the grain surfaces. Gas-phase theoretical models of the chemistry in hot protostellar cores involving the products of grain surface reactions are therefore required to explain the formation of substantially larger organics under interstellar conditions.

In these models, the temperature of the so-called *hot cores* ($\sim$300 K) near young stars leads to thermal evaporation of simple molecules, such as alcohols and aminoalcohols, from the grain surface. These molecules can then undergo gas-phase reactions to form more complex species such as amino acids, sugars, and other biologically important molecules.

The recent detection of glycolaldehyde ($CHOCH_2OH$), the simplest sugar, in the hot core Sagitarrius B2(N-LMH) [74] has confirmed the need for further experimental and observational investigation of these models. One proposed pathway involves both grain surface and hot core gas-phase chemistry for the formation of amino acids. In this pathway, the protonated forms of aminomethanol (NH_2CH_2OH) and aminoethanol ($NH_2CH_2CH_2OH$) react with formic acid ($HCOOH$) to yield the protonated forms of glycine and alanine, respectively [72]. However, laboratory and observational data supporting the presence of these aminoalcohols remains incomplete. Therefore, the first step in the evaluation of this model is the complete spectroscopic characterization of aminomethanol and aminoethanol in order to search for them astronomically.

Aminoethanol is commercially available and the gas-phase species is easily attainable. Its laboratory characterization has been completed [75, 76], and observational searches are underway. In contrast, aminomethanol has not been isolated, and little is known about the stability of this molecule. It is proposed to form from the addition of ammonia to formaldehyde, and the energy barrier for this reaction is calculated (MP2/6-311++G**) to be 34.1 kcal mol^{-1} [77]. However, the hexamethylenetetraamine formation mechanism shows that aminomethanol (1) forms upon the addition of ammonia to formaldehyde in aqueous solution (Figure 7.1, [78]). Aminomethanol then

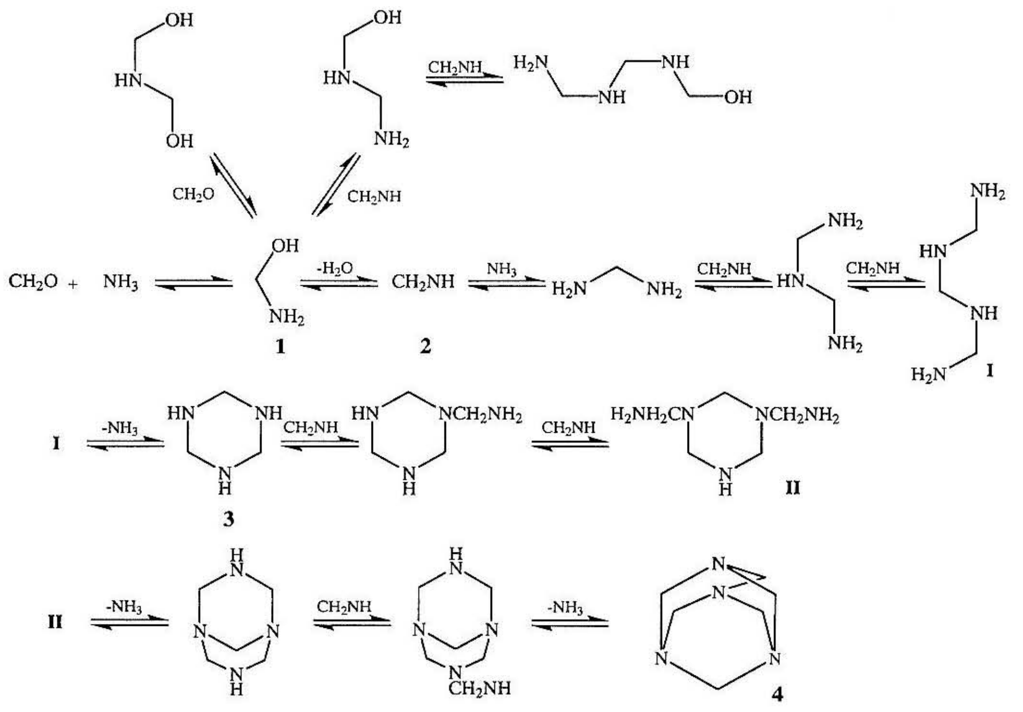


Figure 7.1: Mechanism for reaction of formaldehyde and ammonia.

undergoes a water loss and converts to the highly reactive species methanimine (2). Further reactions lead to the formation of a stable intermediate cyclotrimethylenetriamine (3), with eventual conversion to the stable end product hexamethylenetetraamine (4).

Laboratory exploration of the gas-phase reaction of formaldehyde and ammonia is the most promising route for production of aminomethanol. Reaction of this molecule with other species can be minimized by quenching the formaldehyde + ammonia reaction in a molecular beam. Characterization of aminomethanol is therefore possible if the barrier to the loss of water and conversion to methanimine is sufficiently high. The major route for destruction of aminomethanol in the interstellar medium is also through this water loss and conversion to methanimine. Therefore, determination of this barrier height will indicate the feasibility of laboratory production of aminomethanol as well as this molecule's stability in a hot core environment.

7.2 Theory

The most simple methods one can employ for examining the quantum properties of small molecules are Hartree Fock Theory (HF) and Density Functional Theory (DFT: B3LYP, BLYP, BP86). These are implemented using the Jaguar [1] package. Corrections to HF theory methods can take many forms, though we will focus on HF, MP2, MP4, CCSD, CCSD(T), and QCI(T) implemented in the MolPro package [57, 58, 59, 79, 80, 81, 81, 82].

Quantum Monte Carlo (QMC) is another family of methods which have proven themselves to be very powerful for obtaining very accurate electronic structure energies. The two flavors employed in this paper are Variational QMC (VMC) and Diffusion QMC (DMC). These will be implemented with the QMcBeaver [54] package to test the "Generic Jastrow" (GJ) parameter set.

For QMC one must pick a Jastrow form and variationally determine the parameters in the correlation functions. The Generic Jastrow for hydrocarbons is used in this study with the following form (7.1)

$$
\begin{aligned}
u_{\uparrow\downarrow} &= \frac{\frac{1}{2}r_{ij}}{1+3.0r_{ij}} \\
u_{\uparrow\uparrow} = u_{\downarrow\downarrow} &= \frac{\frac{1}{4}r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,H} = u_{\downarrow,H} &= \frac{-r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,C} = u_{\downarrow,C} &= \frac{-6r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,N} = u_{\downarrow,N} &= \frac{-7r_{ij}}{1+100r_{ij}} \\
u_{\uparrow,O} = u_{\downarrow,O} &= \frac{-8r_{ij}}{1+100r_{ij}}
\end{aligned} \tag{7.1}
$$

This set of "Generic Jastrows" is very similar to the hydrocarbon GJ set. This work aims at examining the validity of this generic set of parameters to a larger body of simple molecules.

7.3 Experiment

7.3.1 Experiment Setup

Full geometry optimizations and transition state searches were completed using b3lyp/cc-pVTZ level of theory with Jaguar, as experience has shown that this level determines geometries well. These geometries were then fixed and single point energy calculations were completed using a variety of methods. Thermodynamic calculations were not able to be completed at this level of theory with Jaguar because the basis contained "f" type functions. Transition states were verified with the analytic Hessian calculations. These geometries are in Figure 7.2.

We used Jaguar to do full thermodynamic calculations using b3lyp/cc-pVTZ(-f). This basis is very similar to the full cc-pVTZ used to obtain

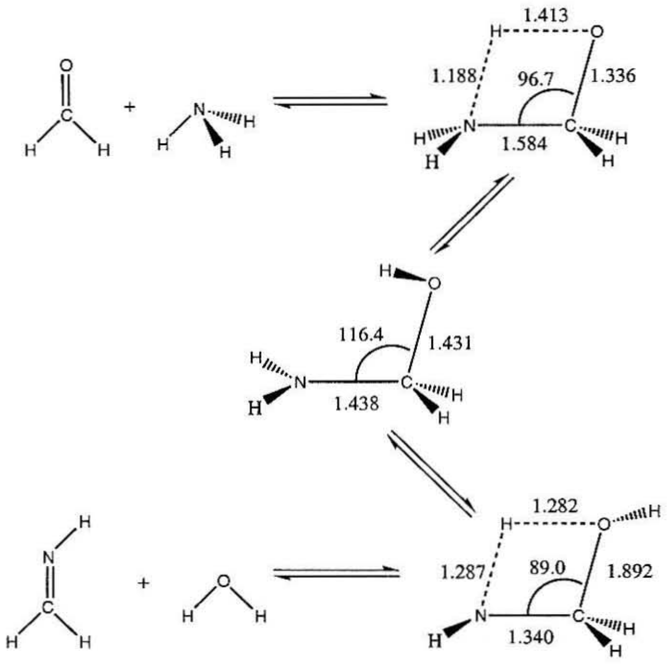


Figure 7.2: Full mechanism of aminomethanol formation from NH_3 and CH_2O and decomposition to CH_2NH and H_2O. Geometries determined with Jaguar [1] b3lyp/cc-pVTZ.

the electronic energy but the "f" type orbitals are removed since Jaguar is unable to analytically take derivatives of these functions. These calculations provided a zero point energy correction to the electronic energy as well as free energy corrections.

A larger *cc-pVTZ++* [1] and *aug-cc-pVTZ* [57] basis sets were used to determine the importance of diffuse functions in this mechanism. This larger basis set is ideal to use since it may describe the lower electron density regions better, particularly in the transition states. This basis set, however, was too large to run for all methods on our current computational resources. Where possible, the energies for this basis are given.

The b3lyp/cc-pVTZ(-f) thermodynamic corrections were used through-

method	NH_3	CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O	CH_2NH
b3lyp*	-114.5494	-56.5847	-171.0856	-171.1537	-171.0634	-76.4599	-94.6694
blyp*	-114.5228	-56.5569	-171.0313	-171.0936	-171.0139	-76.4413	-94.6325
bp86*	-114.5472	-56.5827	-171.0916	-171.1513	-171.0717	-76.4596	-94.6655
HF*	-113.9120	-56.2178	-170.0584	-170.1464	-170.0266	-76.0569	-94.0677
HF**	-113.9120	-56.2177	-170.0585	-170.1464	-170.0266	-76.0568	-94.0677
MP2**	-114.3070	-56.4529	-170.7153	-170.7841	-170.6896	-76.3186	-94.4386
MP4**	-114.3367	-56.4734	-170.7624	-170.8314	-170.7382	-76.3330	-94.4733
CCSD**	-114.3173	-56.4655	-170.7309	-170.8052	-170.7034	-76.3245	-94.4560
CCSD(T)**	-114.3337	-56.4732	-170.7584	-170.8292	-170.7336	-76.3322	-94.4725
QCI(T)**	-114.3343	-56.4733	-170.7593	-170.8297	-170.7347	-76.3323	-94.4729
VMC (HF-GJ)***	NA	NA	NA	-170.2987 $\pm$0.0055	-170.181 $\pm$0.014	-76.1803 $\pm$0.0055	-94.2183 $\pm$0.0047

Table 7.1: Absolute energies (Hartree) for various methods (basis: cc-pVTZ). *Jaguar, **Molpro, ***QMcBeaver

out for the other methods with the other basis sets.

7.3.2 Data

Electronic energies were obtained with various methods using the cc-pVTZ basis. These results are found in Table 7.1.

To verify the absence of diffuse functions was a valid assumption to make these calculations less expensive several methods are given with the *cc-pVTZ++* [1] and *aug-cc-pVTZ* [57] basis sets.

These electronic energies in Tables 7.1 and 7.2 are corrected with the zero point and thermochemical corrections at 300K and 2.63E-5 atm in Tables 7.3 and 7.4.

The free energies based on QCI(T)/cc-pVTZ base energies are given in Tables 7.5 and 7.6 for various temperatures. These allow for comparison of kinetics for different temperatures and pressures.

method	NH_3	CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O	CH_2NH
b3lyp*	-114.5520	-56.5887	-171.0922	-171.1594	-171.0698	-76.4660	-94.6719
blyp*	-114.5259	-56.5619	-171.0390	-171.1009	-171.0214	-76.4489	-94.6355
bp86*	-114.5497	-56.5868	-171.0983	-171.1576	-171.0779	-76.4658	-94.6680
HF*	-113.9140	-56.2201	-170.0633	-170.1503	-170.0313	-76.0600	-94.0696
HF**	-113.9142	-56.2202	NA	NA	NA	-76.0603	-94.0697
MP2**	-114.3161	-56.4605	NA	NA	NA	-76.3290	-94.4466
MP4**	-114.3460	-56.4810	NA	NA	NA	-76.3437	NA
CCSD**	-114.3254	-56.4722	NA	NA	NA	-76.3337	NA
CCSD(T)**	-114.3427	-56.4806	NA	NA	NA	-76.3423	NA
QCI(T)**	-114.3433	-56.4807	NA	NA	NA	-76.3426	NA

Table 7.2: Absolute energies (Hartree) for various methods (basis: cc-pVTZ++/aug-cc-pVTZ). *Jaguar, **Molpro, ***QMcBeaver

7.4 Conclusion

All methods in Table 7.3 are in fair agreement excluding the $[H_2O \ \& \ CH_2NH]$ for VMC(HF-GJ). This can be observed in Table 7.1, where the two smaller molecules gain more correlation energy relative to molecular size than the two larger molecules.

The QMC jobs show that the *Generic Jastrow* regains some of the missing correlation energy in the HF description. The *Generic Jastrows* found for hydrocarbons is transferable to these electronically similar molecules for obtaining *some* of the correlation. At the same time, the results obtained from the pure VMC calculations are of little value when compared to the other high-level methods. This supports the use of *Generic Jastrows* for these types of systems for initial guess parameter sets which regain some of the missing correlation but does not support the use of this type of parameter set for final VMC calculations.

The verification of the formation mechanism provides little new insight.

method	NH_3 & CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O & CH_2NH
b3lyp*	-8.47	39.25	0	51.74	-5.33
blyp*	-12.06	35.56	0	45.04	-8.25
bp86*	-7.29	33.96	0	45.01	-4.21
HF*	-10.28	51.72	0	70.30	-6.91
HF**	-10.28	51.67	0	70.28	-6.91
MP2**	-5.58	39.67	0	54.40	-3.75
MP4**	-7.39	39.76	0	53.55	-4.88
CCSD**	-6.62	43.17	0	59.01	-5.09
CCSD(T)**	-6.73	40.91	0	55.07	-5.28
QCI(T)**	-6.84	40.72	0	54.69	-5.24
VMC(HF-GJ)***	NA	NA	0	68.94 ± 9.4	-83.32 ± 5.7

Table 7.3: Relative free energies ΔG (kcal/mol) for various methods with cc-pVTZ basis with Jaguar b3lyp/cc-pVTZ(-f) zero point and thermochemical corrections at 2.63E-5 atm and 300K. *Jaguar, **Molpro, ***QMcBeaver

The results obtained from the traditional higher level methods provide similar results to those obtained in previous work [77].

The barrier to elimination of water is 55 kcal/mol at ambient temperatures, indicating that the conversion to methanimine is highly unfavored under typical laboratory conditions. Therefore, loss of aminomethanol through this and other pathways can be virtually eliminated by minimizing reactions with other species in a molecular beam experiment. Spectroscopic characterization of aminomethanol should therefore be a straightforward process.

In addition, these results indicate that aminomethanol could indeed be a stable species in hot core environments, which are typically near ambient temperatures. Aminomethanol is predicted to be at densities similar to those for observed alcohols in hot cores. Once the laboratory characterization is

method	NH_3 & CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O & CH_2NH
b3lyp*	-8.99	38.70	0	51.34	-7.08
blyp*	-12.49	35.36	0	44.96	-10.30
bp86*	-7.47	33.75	0	45.09	-5.65
HF*	-10.57	51.05	0	69.76	-7.66

Table 7.4: Relative free energies ΔG (kcal/mol) for various methods with cc-pVTZ++/aug-cc-pVTZ basis with Jaguar b3lyp/cc-pVTZ(-f) zero point and thermochemical corrections at 2.63E-5 atm and 300K. *Jaguar, **Molpro, ***QMcBeaver

complete, aminomethanol will therefore be an ideal target for observational searches.

temp	NH_3 & CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O & CH_2NH
0	8.84	40.58	0	54.74	10.14
100	4.37	40.59	0	54.71	5.76
200	-1.12	40.62	0	54.69	0.38
300	-6.84	40.72	0	54.69	-5.24
400	-12.63	40.87	0	54.69	-10.94
500	-18.43	41.07	0	54.69	-16.64
600	-24.21	41.30	0	54.69	-22.33
700	-29.95	41.55	0	54.69	-28.00
800	-35.66	41.82	0	54.69	-33.63
900	-41.34	42.11	0	54.69	-39.23
1000	-46.98	42.41	0	54.70	-44.80
1100	-52.59	42.71	0	54.71	-50.34

Table 7.5: Relative free energies ΔG (kcal/mol) at 2.63E-5 atm for various temperatures with zero point and thermochemical corrections from Jaguar (b3lyp/cc-pVTZ(-f)) on energetics from MolPro (QCI(T)/cc-pVTZ).

temp	NH_3 & CH_2O	$T^{\dagger}_{NH_3+CH_2O}$	$CH_2(OH)NH_2$	$T^{\dagger}_{H_2O+CH_2NH}$	H_2O & CH_2NH
0	8.84	40.58	0	54.74	10.14
100	2.86	40.59	0	54.71	4.25
200	-4.14	40.62	0	54.69	-2.64
300	-11.37	40.72	0	54.69	-9.77
400	-18.67	40.87	0	54.69	-16.98
500	-25.98	41.07	0	54.69	-24.19
600	-33.27	41.30	0	54.69	-31.39
700	-40.53	41.55	0	54.69	-38.57
800	-47.75	41.82	0	54.69	-45.71
900	-54.93	42.11	0	54.69	-52.82
1000	-62.08	42.41	0	54.70	-59.91
1100	-69.20	42.71	0	54.71	-66.96

Table 7.6: Relative free energies ΔG (kcal/mol) at 1.32E-8 atm for various temperatures with zero point and thermochemical corrections from Jaguar (b3lyp/cc-pVTZ(-f)) on energetics from MolPro (QCI(T)/cc-pVTZ).

atom	x	y	z
N	0.0000000000000	0.0000000000000	0.0000000000000
H	0.0000000000000	0.0000000000000	1.0141884635000
H	0.9722262947221	0.0000000000000	-0.2877117192551
H	-0.3866564451307	-0.8920318164661	-0.2877117192551

Table 7.7: Geometry for NH_3.

atom	x	y	z
O	0.0000000000000	0.0000000000000	0.0000000000000
C	0.0000000000000	0.0000000000000	1.1998577370000
H	0.9369230569270	0.0000000000000	1.7885795046292
H	-0.9369230569270	0.0000000000000	1.7885795046292

Table 7.8: Geometry for CH_2O.

atom	x	y	z
N	0.0000000000000	0.0000000000000	0.0000000000000
C	0.0000000000000	0.0000000000000	1.5843876532000
O	1.3268837112519	0.0000000000000	1.7410676408991
H	-0.3691539435124	-0.8336901130763	-0.4490956442016
H	-0.3631605001333	0.8366888213139	-0.4483032152738
H	-0.5594636703232	0.8984852223094	1.9001385714186
H	-0.5591343872703	-0.8994665053243	1.8988541274420
H	1.1378517202149	-0.0018988097578	0.3412417016124

Table 7.9: Geometry for $T^{\dagger}_{NH_3+CH_2O}$.

atom	x	y	z
N	0.0000000000000	0.0000000000000	0.0000000000000
C	0.0000000000000	0.0000000000000	1.4380669284000
O	1.2820747992013	0.0000000000000	2.0740189979613
H	0.5657837377401	-0.7566646507275	-0.3638314440938
H	0.3496459707302	0.8701754653024	-0.3813217278531
H	-0.5942959853476	0.8530557152119	1.7775757584241
H	-0.4769046620119	-0.9139223538174	1.7883090361815
H	1.6911105284372	0.8608638484430	1.9389453497317

Table 7.10: Geometry for $CH_2(OH)NH_2$.

atom	x	y	z
O	0.0000000000000	0.0000000000000	0.0000000000000
H	0.0000000000000	0.0000000000000	1.2824478194000
N	1.0290406695545	0.0000000000000	2.0554527257564
C	1.6155874630923	-0.3760963838510	0.9109052611784
H	1.2502580267367	0.9677093577925	2.2725286174727
H	2.3722912575875	0.2198389043536	0.4068337595816
H	1.6216229675463	-1.4327932210215	0.6660865869843
H	-0.4382033110516	-0.7395829315353	-0.4448633375480

Table 7.11: Geometry for $T^{\dagger}_{H_2O+CH_2NH}$.

atom	x	y	z
O	0.0000000000000	0.0000000000000	0.0000000000000
H	0.0000000000000	0.0000000000000	0.9616229062000
H	0.9304177983672	0.0000000000000	-0.2429842262576

Table 7.12: Geometry for H_2O.

atom	x	y	z
N	0.0000000000000	0.0000000000000	0.0000000000000
C	0.0000000000000	0.0000000000000	1.2640707739000
H	0.9536983299391	0.0000000000000	-0.3660720458042
H	0.8975274435092	0.0002654608506	1.8902959237014
H	-0.9530880382316	0.0000011614524	1.7921749554776

Table 7.13: Geometry for CH_2NH.

Chapter 8

QMcBeaver

I contemplated adding the entire QMcBeaver source code to this thesis until I came to the harsh realization that it was several hundred pages single spaced. The actual source can be obtained by contacting the William A. Goddard group or by searching online. We are currently attempting to get a *gnu* public license, but at the time of this writing it is not secure and no devoted url exists for its distribution. Hopefully, in the near future this will be accomplished.

I did include the current version of the user's and developer's manual. It is attached as a supplement to the thesis. It will serve both the developers of QMcBeaver and those developing their own QMC package well. QMcBeaver is still very much an academic code and many parts need serious engineering to become optimally efficient. At the same time, this version of QMcBeaver has some novel features and provides a good framework from which to extend. We hope those who obtain QMcBeaver will find it provides insight on developing better distributed algorithms as well as better QMC codes.

Bibliography

[1] Murco N. Ringnalda, Jean-Marc Langlois, Robert B. Murphy, Burnham H. Greeley, Christian Cortis, Thomas V. Russo, Bryan Marten, Robert E. Donnelly, Jr., W. Thomas Pollard, Yixiang Cao, Richard P. Muller, Daniel T. Mainz, Julie R. Wright, Gregory H. Miller, William A. Goddard III, and Richard A. Friesner. Jaguar v4.0, 2001.

[2] Chakraborty, R. P. Muller, S. Dasgupta, and W.A. Goddard III. The mechanism for unimolecular decomposition of RDX (1,3,5-trinitro-1,3,5-triazine), an ab initio study. *Journal of Physcial Chemistry*, 104:2261–2272, 2000.

[3] Burnham H. Greeley, Thomas V. Russo, Daniel T. Mainz, Richard A. Friesner, William A. Goddard III, Robert E. Donnelly, Jr., and Murco N. Ringnalda. New pseudospectral algorithms for electronic structure calculations: Length-scale separation and analytical two-electron integral calculations. *Journal of Chemical Physics*, 101:4028, 1994.

[4] J. C. Slater. *The Self-Consistent Field for Molecules and Solids.* McGraw-Hill, New York, 1974.

[5] Axel D. Becke. Density functional thermochemistry III: The role of exact exchange. *Journal of Chemical Physics*, 98:5648, 1993.

[6] Axel D. Becke. Density-functional exchange-energy approximation with correct asymptotic behavior. *Physical Review A*, 38:3098, 1988.

[7] S. H. Vosko, L. Wilk, and M. Nusair. Accurate spin-dependent electron liquid correlation energies for local spin density calculations: A critical analysis. *Canadian Journal of Physics*, 58:1200, 1980.

[8] C. Lee, W. Yang, and R. G. Parr. Development of the Colle-Salvetti correlation energy formula into a functional of the electron density. *Physical Review B*, 37:785, 1988.

[9] J. B. Anderson, C. A. Traynor, and B. M. Boghosian. Quantum-chemistry by random-walk-exact treatment of many-electron systems. *Journal of Chemical Physics*, 95(10):7418–7425, 1991.

[10] C. J. Umrigar, K. G. Wilson, and J. W. Wilkins. Optimized trial wavefunctions for Quantum Monte Carlo calculations. *Physical Review Letters*, 60(17):1719–1722, 1988.

[11] Lubos Mitas. Diffusion Monte Carlo. In M. P. Nightingale and C. J. Umrigar, editors, *Quantum Monte Carlo Methods in Physics and Chemistry*, volume 525 of *Nato Science Series C: Mathematical and Physical Sciences*, pages 247–261, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.

[12] C. J. Umrigar. Basics, Quantum Monte Carlo and statistical mechanics. In M. P. Nightingale and C. J. Umrigar, editors, *Quantum Monte Carlo Methods in Physics and Chemistry*, volume 525 of *Nato Science Series C: Mathematical and Physical Sciences*, pages 1–36, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.

[13] C. J. Umrigar. Variational Monte Carlo basics and applications to atoms and molecules. In M. P. Nightingale and C. J. Umrigar, editors, *Quantum Monte Carlo Methods in Physics and Chemistry*, volume 525 of *Nato Science Series C: Mathematical and Physical Sciences*, pages 129–160, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.

[14] L. Mitas and J. C. Grossman. Quantum Monte Carlo for electronic structure of clusters and solids. *Abstracts of Papers of the American Chemical Society*, 211(1):21–COMP, 1996.

[15] L. Mitas. Electronic structure calculations by quantum monte carlo methods. *Physica B*, 237:318–320, 1997.

[16] J. C. Grossman and L. Mitas. High accuracy molecular heats of formation and reaction barriers: Essential role of electron correlation. *Physical Review Letters*, 79(22):4353–4356, 1997.

[17] J. C. Grossman and L. Mitas. Quantum Monte Carlo as a high-accuracy method for treating chemical reactions. *Abstracts of Papers of the American Chemical Society*, 213(2):171–PHYS, 1997.

[18] W. M. C. Foulkes, L. Mitas, R. J. Needs, and G. Rajagopal. Quantum Monte Carlo simulations of solids. *Reviews of Modern Physics*, 73(1):33–83, 2001.

[19] Y. Kwon, D. M. Ceperley, and R. M. Martin. Transient-estimate Monte Carlo in the two-dimensional electron gas. *Physical Review B*, 53(11):7376–7382, 1996.

[20] David M. Ceperley and Lubos Mitas. Quantum Monte Carlo methods in chemistry. In I. Prigogine and Stuart A. Rice, editors, *Advances in Chemical Physics*, volume XCIII. John Wiley and Sons, Inc., 1996.

[21] S. Fahy, X. W. Wang, and S. G. Louie. Variational Quantum Monte Carlo nonlocal pseudopotential approach to solids-formulation and application to diamond, graphite, and silicon. *Physical Review B*, 42(6):3503–3522, 1990.

[22] S. Fahy, X. W. Wang, and S. G. Louie. Variational Quantum Monte Carlo nonlocal pseudopotential approach to solids-cohesive and structural-properties of diamond. *Physical Review Letters*, 61(14):1631–1634, 1988.

[23] Malvin H. Kalos and Francesco Pederiva. Fermion Monte Carlo. In M. P. Nightingale and C. J. Umrigar, editors, *Quantum Monte Carlo Methods in Physics and Chemistry*, volume 525 of *Nato Science Series C: Mathematical and Physical Sciences*, pages 263–286, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.

[24] M. H. Kalos and F. Pederiva. Exact Monte Carlo method for continuum fermion systems. *Physical Review Letters*, 85(17):3547–3551, 2000.

[25] M. H. Kalos and F. Pederiva. Fermion Monte Carlo for continuum systems. *Physica A*, 279(1-4):236–243, 2000.

[26] M. H. Kalos. Exact Monte Carlo for few-fermion systems. *Journal of Statistical Physics*, 63(5-6):1269–1281, 1991.

[27] M. H. Kalos. Monte Carlo methods for the many-fermion problem. *Physica A*, 124(1-3):427–427, 1984.

[28] M. H. Kalos. The Green-Function Monte Carlo method. *Bulletin of the American Physical Society*, 25(7):725–725, 1980.

[29] M. H. Kalos. Monte Carlo methods in quantum many-body problems. *Nuclear Physics A*, 328(1-2):153–168, 1979.

[30] B. J. Alder, K. J. Runge, and R. T. Scalettar. Variational Monte Carlo study of an interacting electron-phonon model. *Physical Review Letters*, 79(16):3022–3025, 1997.

[31] J. B. Anderson. An exact Quantum Monte Carlo calculation of the helium-helium intermolecular potential. ii. *Journal of Chemical Physics*, 115(10):4546–4548, 2001.

[32] J. B. Anderson. Quantum Monte Carlo: Direct calculation of corrections to trial wave functions and their energies. *Journal of Chemical Physics*, 112(22):9699–9702, 2000.

[33] J. B. Anderson. Quantum Monte Carlo. From a few electrons to a few thousand. *Abstracts of Papers of the American Chemical Society*, 216(2):U776–U776, 1998.

[34] J. B. Anderson. Fixed-node Quantum Monte Carlo. *International Reviews in Physical Chemistry*, 14(1):85–112, 1995.

[35] D. Bressanini and P. J. Reynolds. Spatial-partitioning-based acceleration for variational Monte Carlo. *Journal of Chemical Physics*, 111(14):6180–6189, 1999.

[36] D. Bressanini and P. J. Reynolds. Between classical and Quantum Monte Carlo methods: "variational" QMC. *Monte Carlo Methods in Chemical Physics*, 105:37–64, 1999.

[37] C. Chakravarty, M. C. Gordillo, and D. M. Ceperley. A comparison of the efficiency of fourier-and discrete time-path integral Monte Carlo. *Journal of Chemical Physics*, 109(6):2123–2134, 1998.

[38] C. Filippi and C. J. Umrigar. Correlated sampling in Quantum Monte Carlo: A route to forces. *Physical Review B*, 61(24):R16291–R16294, 2000.

[39] C. Filippi and S. Fahy. Optimal orbitals from energy fluctuations in correlated wave functions. *Journal of Chemical Physics*, 112(8):3523–3531, 2000.

[40] 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.

[41] A. Luchow and J. B. Anderson. Monte Carlo methods in electronic structures for large systems. *Annual Review of Physical Chemistry*, 51:501–526, 2000.

[42] L. Mitas. Electronic structure by Quantum Monte Carlo: Atoms, molecules and solids. *Computer Physics Communications*, 96(2-3):107–117, 1996.

[43] F. Pederiva and M. H. Kalos. Fermion Monte Carlo. *Computer Physics Communications*, 122(SI):440–445, 1999.

[44] T. Torelli and L. Mitas. Recent developments in the Quantum Monte Carlo method: Evaluation of interatomic forces. *Progress of Theoretical Physics Supplement*, (138):78–83, 2000.

[45] A. J. Williamson, R. Q. Hood, and J. C. Grossman. Linear-scaling Quantum Monte Carlo calculations. *Physical Review Letters*, 8724(24):6406–+, 2001.

[46] A. J. Williamson, S. D. Kenny, G. Rajagopal, A. J. James, R. J. Needs, L. M. Fraser, W. M. C. Foulkes, and P. Maccullum. Optimized wave functions for Quantum Monte Carlo studies of atoms and solids. *Physical Review B*, 53(15):9640–9648, 1996.

[47] F. H. Zong and D. M. Ceperley. Path integral Monte Carlo calculation of electronic forces. *Physical Review E*, 58(4):5123–5130, 1998.

[48] 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.

[49] H. Flyvberg and H. Peterson. Error estimates on averages of correlated data. *Journal of Chemical Physics*, 91:461–466, 1989.

[50] M.T. Feldmann, D.R. Kent IV, R.P. Muller, and W.A. Goddard III. Efficient algorithm for "on-the-fly" error analysis of local or distributed serially-correlated data. *Journal of Chemical Physics*, submitted, 2002.

[51] O. Yaser. New trends in high performance computing. *Parallel Computing*, 27:1–2, 2001.

[52] Y. Deng and A. Korobka. The performance of a supercomputer built with commodity components. *Parallel Computing*, 27:91–108, 2001.

[53] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15:200–222, 2001.

[54] M.T. Feldmann and D.R. Kent IV. QM^{c}Beaver v2002.01.09 ©, 2001.

[55] 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.

[56] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. Gamess, a package of ab initio programs, version 2000, 2000.

[57] R. D. Amos, A. Bernhardsson, A. Berning, P. Celani, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, C. Hampel, G. Hetzer, P. J. Knowles, T. Korona, R. Lindh, A. W. Lloyd, S. J. McNicholas, F. R. Manby, W. Meyer, M. E. Mura, A. Nicklass, P. Palmieri, R. Pitzer, G. Rauhut, M. Schütz, U. Schumann, H. Stoll, A. J. Stone, R. Tarroni, T. Thorsteinsson, and H.-J. Werner. Molpro, a package of ab initio programs designed by H.-J. Werner and P. J. Knowles, version 2000.1, 2000.

[58] Roland Lindh. Molpro modual: SEWARD (gaussian integral code), 2000.

[59] W. Meyer and H.-J. Werner. Molpro modual: RHF-SCF, 2000.

[60] T. Kato. On the eigenfunctions of many-particle systems in quantum mechanics. *Communication on Pure Applied Mathematics*, 10:151–177, 1957.

[61] K. Wilson. Recent developments in guage theories, 1979.

[62] C. Whitmer. Over-relaxation methods for monte-carlo simulations of quadratic and multiquadratic actions. *Physical Review D*, 29:306–311, 1984.

[63] S. Gottlieb, P. Mackenzie, H. Thacker, and D. Weingarten. Hadronic coupling-constants in lattice gauge-theory. *Nuclear Physics B*, 263:704–730, 1986.

[64] R. P. Muller, M. T. Feldmann, R. N. Barnett, B. L. Hammond, P. J. Reynold, L. Terray, and W. A. Lester Jr. California Institute of Technology Material Simulation Center parallel QMAGIC, version 1.1.0p, 2000.

[65] R. Needs, G. Rajagopal, M. D. Towler, P. R. C. Kent, and A.Williamson. CASINO, the Cambridge Quantum Monte Carlo code, version 1.1.0, 2000.

[66] L. Smith and P. Kent. Development and performance of mixed OpenMP/MPI Quantum Monte Carlo code. *Concurrency: Practice and Experience*, 12:1121–1129, 2000.

[67] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core.* The MIT Press, Cambridge, Massachusetts, second edition, 1998.

[68] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14:1347–1363, 1993.

[69] Rollin A. King, T. Daniel Crawford, John F. Stanton, and Henry F. Schaefer III. Conformations of [10]annulene: More bad news for density functional theory and second-order perturbation theory. *Journal of the American Chemical Society*, 121:10788–10793, 1999.

[70] Ohishi M. Observations of hot cores. In E.F. van Dishoeck, editor, *IAU Symposium No. 178: Molecules in Astrophysics*, pages 61 – 74. Kluwer; Dordrecht, 1997.

[71] Charnley S. Interstellar alcohols. *Astrophysics Journal*, 448:232, 1995.

[72] Charnley S. Interstellar organic chemistry. In *The Bridge Between the Big Bang and Biology.* Consiglio Nazionale delle Ricerche, Italy, 1999.

[73] S. Charnley. On the nature of interstellar organic chemistry. In C. B. Cosmovici, S. Bowyer, and D. Werthimer, editors, *Astronomical and Biochemical Origins and the Search for Life in the Universe*, page 89. Editrice Compositori; Bologna, 1997.

[74] J. M. Hollis, F. J. Lovas, and P. R. Jewell. Interstellar glycolaldehyde: The first sugar. *Astrophysics Journal*, 540:L107–L110, 2000.

[75] R. E. Penn and R. F. Curl. Microwave spectrum of 2-aminoethanol: Structural effects of the hydrogen bond. *Journal Chemical Physics*, 53:651 – 658, 1971.

[76] S. L. Widicus, B. J. Drouin, K. A. Dyl, and G. A. Blake. Title in prep. *in prep.*, 2002.

[77] Minyaev R. M. and Lepin E. A. Gradient line reaction path of ammonia addition to formaldehyde. *Mendeleev Communications*, 5:189–191, 1997.

[78] A. T. Nielsen, D. W. Moore, M. D. Ogan, and R. L. Atkins. Structure and chemistry of the aldehyde ammonias .3. formaldehyde-ammonia reaction - 1,3,5-hexahydrotriazine. *Journal Organic Chemistry*, 44:1678 – 1684, 1979.

[79] C. Hampel, H.-J. Werner, M. Deegan, and P. J. Knowles. Molpro modual: MP2, 2000.

[80] C. Hampel, H.-J. Werner, M. Deegan, and P. J. Knowles. Molpro modual: MP4, 2000.

[81] C. Hampel, H.-J. Werner, M. Deegan, and P. J. Knowles. Molpro modual: CCSD, 2000.

[82] C. Hampel, H.-J. Werner, M. Deegan, and P. J. Knowles. Molpro modual: QCI, 2000.

QMcBeaver Reference Manual

Generated by Doxygen 1.2.15

Wed May 1 11:34:37 2002

Contents

1 QMcBeaver Hierarchical Index 1

2 QMcBeaver Compound Index 3

3 QMcBeaver Class Documentation 8

4 QMcBeaver File Documentation 164

1 QMcBeaver Hierarchical Index

1.1 QMcBeaver Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

- **Array1D< T >** 8
- **Array2D< T >** 11
- **Array3D< T >** 15
- **Array4D< T >** 18
- **Complex** 23
- **Exception** 37
  - **XMLParseException** 161
- **FunctionR1toR1** 40
  - **CubicSpline** 28
    - **CubicSplineWithGeometricProgressionGrid** 32
  - **Polynomial** 45
    - **QMCPolynomial** 108
- **ParameterScorePair** 43
- **QMCBasisFunction** 49
- **QMCBasisFunctionCoefficients** 53

QMCCopyright **56**

QMCCorrelatedSamplingVMCOptimization **56**

QMCCorrelationFunction **57**

  FixedCuspPadeCorrelationFunction **38**

  PadeCorrelationFunction **42**

  ZeroCorrelationFunction **162**

QMCCorrelationFunctionFactory **58**

QMCCorrelationFunctionParameters **59**

QMCDerivativeProperties **64**
QMCflags

QMCFunctions **67**

QMCInitializeWalker **72**

  QMCMikesJackedWalkerInitialization **95**

QMCInitializeWalkerFactory **73**
QMCInput

QMCJastrow **74**

QMCJastrowElectronElectron **77**

QMCJastrowElectronNuclear **80**

QMCJastrowParameters **83**

QMCLineSearchStepLengthSelectionAlgorithm **90**

  QMCMikesBracketingStepLengthSelector **94**

QMCLineSearchStepLengthSelectionFactory **91**

QMCManager **92**

QMCMolecule **97**

QMCObjectiveFunction **99**

QMCObjectiveFunctionResult **102**

QMCOptimizationAlgorithm 106
  CKGeneticAlgorithm1 21
  QMCLineSearch 88
    QMCSteepestDescent 129
QMCOptimizationFactory 107
QMCPotential_Energy 113
QMCproperties 114
QMCproperty 116
QMCReadAndEvaluateConfigs 119
QMCrun 121
QMCSlater 124
QMCstatistic 127
QMCStopwatches 131
QMCwalker 134
QMCWavefunction 139
SortedParameterScorePairList 142
Stopwatch 144
StringManipulation 146
XMLElement 150

2 QMcBeaver Compound Index

2.1 QMcBeaver Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Array1D< T > (A 1-dimensional template for making arrays) 8
Array2D< T > (A 2-dimensional template for making arrays) 11

Array3D< T > (A 3-dimensional template for making arrays) 15

Array4D< T > (A 4-dimensional template for making arrays) 18

CKGeneticAlgorithm1 (A moderately greedy genetic algorithm for trying to globally optimize a function dreamed up by David Randall (Chip) Kent IV) 21

Complex (An implementation of a complex number with the associated basic functions) 23

CubicSpline (A 1-dimensional ($R^1 \to R^1$) cubic spline interpolation) 28

CubicSplineWithGeometricProgressionGrid (A 1-dimensional ($R^1 \to R^1$) cubic spline interpolation with a grid that is assumed to be spaced according to a geometric relationship for faster evaluation) 32

Exception (An Exception is thrown when an error occurs) 37

FixedCuspPadeCorrelationFunction (Correlation function which uses a Pade expansion to describe particle-particle interactions) 38

FunctionR1toR1 (An interface for a function from $R^1 \to R^1$) 40

PadeCorrelationFunction (Correlation function which uses a Pade expansion to describe particle-particle interactions) 42

ParameterScorePair (A container which holds a set of parameters and an associated scalar score value) 43

Polynomial (A one dimensional real polynomial) 45

QMCBasisFunction (This class stores all of the parameters that a gaussian basis set is constructed from for a MOLECULE) 49

QMCBasisFunctionCoefficients (This class stores all of the parameters that a gaussian basis set is constructed from for an ATOM) 53

QMCCopyright (Central localtion for all copyright information relevant to QMcBeaver) 56

QMCCorrelatedSamplingVMCOptimization (Optimize the parameters in a variational QMC (VMC) calculation using the correlated sampling method) 56

QMCCorrelationFunction (Interface for a parameterized function describing the interaction of two particles) 57

QMCCorrelationFunctionFactory (Object factory which returns the correct QMCCorrelationFunction (p. 57) when a string keyword describing the correlation function is provided) 58

QMCCorrelationFunctionParameters (This is a collection of parameters and related functions which describe the interaction of two particles of specific types) 59

QMCDerivativeProperties (All of the calculated quantities and properties that are derived from quantities and properties evaluated during a calculation) 64

QMCFunctions (This class calculates the value of the wavefunction, it's first two derivatives, and any other properties which are calculated from the wavefunction (local energy, etc.)) 67

QMCInitializeWalker (Interface to algorithms which generate new walkers for a QMC calculation) 72

QMCInitializeWalkerFactory (Object factory which returns the correct QMCInitialize walker when a string keyword describing the correlation function is provided) 73

QMCJastrow (This class calculates the value of the Jastrow function and it's first two derivatives) 74

QMCJastrowElectronElectron (This class calculates the value of the electron-electron part of the Jastrow function and it's first two derivatives) 77

QMCJastrowElectronNuclear (This class calculates the value of the electron-nuclear part of the Jastrow function and it's first two derivatives) 80

QMCJastrowParameters (This class contains all of the parameters and corelation functons from which the Jastrow function is composed) 83

QMCLineSearch (Abstract implementation of a line search numerical optimization algorithm) 88

QMCLineSearchStepLengthSelectionAlgorithm (Interface to algorithms which determine the proper step length to use during a line search optimization (**QMCLineSearch** (p. 88))) **90**

QMCLineSearchStepLengthSelectionFactory (Object factory which returns the correct **QMCLineSearchStepLength-SelectionAlgorithm** (p. 90) when a string keyword describing the correlation function is provided) **91**

QMCManager (Controls the major sections of a QMC calculation) **92**

QMCMikesBracketingStepLengthSelector (Algorithm to determine the step length for a line search optimization developed by Michael Todd Feldmann) **94**

QMCMikesJackedWalkerInitialization (This is the algorithm made to initialize walkers) **95**

QMCMolecule (Describes a particular molecular geometry) **97**

QMCObjectiveFunction (Objective function optimized during a variational QMC (VMC) calculation to find the optimal wavefunction parameters) **99**

QMCObjectiveFunctionResult (Results from the evaluation of an objective function during a QMC calculation) **102**

QMCOptimizationAlgorithm (Interface for numerical optimization algorithms) **106**

QMCOptimizationFactory (Object factory which returns the correct **QMCOptimizationAlgorithm** (p. 106) specified in the calculation input data) **107**

QMCPolynomial (An extension of **Polynomial** (p. 45) which adds QMC specific functionality) **108**

QMCPotential_Energy (The potential energy of the system) **113**

QMCproperties (All of the quantities and properties evaluated during a calculation) **114**

QMCproperty (All of the statistical information used in calculating a quantity or property during a calculation) **116**

QMCReadAndEvaluateConfigs (Calculates properties (**QMCproperties** (p. 114)) from walkers and related data saved to a file during a QMC calculation) **119**

QMCrun (Collection of walkers (**QMCwalker** (p. 134)) with the functionality to do the basic operations from which a QMC algorithm is built) **121**

QMCSlater (A Slater determinant describing like spin electrons from a 3N dimensional wavefunction) **124**

QMCstatistic (Statistical information on a set of data) **127**

QMCSteepestDescent (Steepest descent line search numerical optimization algorithm) **129**

QMCStopwatches (A collection of **Stopwatch** (p. 144) objects used to record information relevant to the timing of a QMC calculation) **131**

QMCwalker (An instantaneous snapshot of all 3N electronic corrdinates for a system) **134**

QMCWavefunction (The coefficients and parameters describing the trial wavefunction for the system) **139**

SortedParameterScorePairList (A sorted list of **Parameter-ScorePair** (p. 43) objects where the objects are ordered in an increasing order) **142**

Stopwatch (An accurate software stopwatch) **144**

StringManipulation (A set of functions to manipulate strings) **146**

XMLElement (XMLElement is a representation of an XML object) **150**

XMLParseException (An XMLParseException is thrown when an error occures while parsing an XML stream) **161**

ZeroCorrelationFunction (Correlation function which describes noninteracting particles) **162**

3 QMcBeaver Class Documentation

3.1 Array1D< T > Class Template Reference

A 1-dimensional template for making arrays.

Public Methods

- int **dim1** ()

  Gets the number of elements in the array's first dimension.

- int **size** ()

  Gets the total number of elements in the array.

- T * **array** ()

  Gets a pointer to an array containing the array elements.

- void **allocate** (int i)

  Allocates memory for the array.

- void **deallocate** ()

  Deallocates memory for the array.

- void **operator=** (const Array1D &rhs)

  Sets two arrays equal.

- void **operator=** (const T C)

  Sets all of the elements in an array equal to the same value.

- T **operator** * (const Array1D &rhs)

  Returns the dot product of two arrays.

- Array1D **operator** * (const double rhs)

  Returns the product of an array and a double.

- Array1D **operator+** (const Array1D &rhs)

  Returns the sum of two arrays.

- Array1D **operator-** (const Array1D &rhs)

  Returns the difference of two arrays.

- void **operator** *= (const T C)

  Sets this array equal to itself times a scalar value.

- void **operator/=** (const T C)

  Sets this array equal to itself divided by a scalar value.

- **Array1D** ()

  Creates an array.

- **Array1D** (int i)

  Creates an array and allocates memory.

- **Array1D** (const Array1D &rhs)

  Creates an array and sets it equal to another array.

- **~Array1D** ()

  Destroy's the array and cleans up the memory.

- T & **operator()** (int i)

  Accesses element `(i)` *of the array.*

Friends

- ostream & **operator<<** (ostream &strm, const Array1D< T > &rhs)

  Prints the array to a stream.

3.1.1 Detailed Description

template<class T> class Array1D< T >

A 1-dimensional template for making arrays.

All of the memory allocation and deallocation details are dealt with by the class.

Definition at line 26 of file Array1D.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 template<class T> Array1D< T >::Array1D (int *i*) `[inline]`

Creates an array and allocates memory.

Parameters:

i size of the array's first dimension.

Definition at line 256 of file Array1D.h.

3.1.2.2 template<class T> Array1D< T >::Array1D (const Array1D< T > & *rhs*) `[inline]`

Creates an array and sets it equal to another array.

Parameters:

rhs array to set this array equal to.

Definition at line 265 of file Array1D.h.

3.1.3 Member Function Documentation

3.1.3.1 template<class T> void Array1D< T >::allocate (int *i*) `[inline]`

Allocates memory for the array.

Parameters:

i size of the array's first dimension.

Definition at line 69 of file Array1D.h.

Referenced by Array1D< QMCBasisFunctionCoefficients >::Array1D(), QMCJastrowParameters::getParameters(), Array1D< QMCBasisFunction-Coefficients >::operator=(), QMCReadAndEvaluateConfigs::rootCalculate-Properties(), QMCJastrowParameters::setParameterVector(), and QMCRead-AndEvaluateConfigs::workerCalculateProperties().

3.1.3.2 template<class T> T* Array1D< T >::array () `[inline]`

Gets a pointer to an array containing the array elements.

The ordering of this array is NOT specified.

Definition at line 61 of file Array1D.h.

Referenced by QMCCorrelatedSamplingVMCOptimization::optimize().

3.1.3.3 template<class T> int Array1D< T >::dim1 () `[inline]`

Gets the number of elements in the array's first dimension.

Returns:

number of elements in the array's first dimension.

Definition at line 48 of file Array1D.h.

Referenced by QMCJastrowElectronNuclear::evaluate(), QMCJastrowParameters::getParameters(), QMCPolynomial::hasNonNegativeZeroes(), PadeCorrelationFunction::initializeParameters(), FixedCuspPadeCorrelationFunction::initializeParameters(), CubicSpline::initializeWithDerivativeValues(), CubicSpline::initializeWithFunctionValues(), QMCLineSearch::optimize(), QMCCorrelatedSamplingVMCOptimization::optimize(), QMCCorrelationFunctionParameters::setParameters(), QMCJastrowParameters::setParameterVector(), and QMCReadAndEvaluateConfigs::workerCalculateProperties().

3.1.3.4 template<class T> int Array1D< T >::size () `[inline]`

Gets the total number of elements in the array.

Returns:

total number of elements in the array.

Definition at line 55 of file Array1D.h.

3.2 Array2D< T > Class Template Reference

A 2-dimensional template for making arrays.

Public Methods

- int **dim1** ()

  Gets the number of elements in the array's first dimension.

- int **dim2** ()

  Gets the number of elements in the array's second dimension.

- int **size** ()

  Gets the total number of elements in the array.

- T * **array** ()

  Gets a pointer to an array containing the array elements.

- void **allocate** (int i, int j)

  Allocates memory for the array.

- void **deallocate** ()

  Deallocates memory for the array.

- void **operator=** (const Array2D &rhs)

  Sets two arrays equal.

- void **operator=** (const T C)

  Sets all of the elements in an array equal to the same value.

- Array2D **operator** * (const Array2D &rhs)

  Returns the matrix product of two arrays.

- Array2D **operator** * (const T C)

  Returns the product of an array and a scalar.

- void **operator** *= (const T C)

  Sets this array equal to itself times a scalar value.

- void **operator/=** (const T C)

  Sets this array equal to itself divided by a scalar value.

- **Array2D** ()

  Creates an array.

- **Array2D** (int i, int j)

  Creates an array and allocates memory.

- **Array2D** (const Array2D< T > &rhs)

  Creates an array and sets it equal to another array.

- **~Array2D** ()

  Destroy's the array and cleans up the memory.

- T & **operator()** (int i, int j)

  Accesses element `(i,j)` *of the array.*

Friends

- ostream & **operator<<** (ostream &strm, const Array2D< T > &rhs)

  Prints the array to a stream.

3.2.1 Detailed Description

template<class T> class Array2D< T >

A 2-dimensional template for making arrays.

All of the memory allocation and deallocation details are dealt with by the class.

Definition at line 27 of file Array2D.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 template<class T> Array2D< T >::Array2D (int *i*, int *j*) `[inline]`

Creates an array and allocates memory.

Parameters:
- *i* size of the array's first dimension.
- *j* size of the array's second dimension.

Definition at line 242 of file Array2D.h.

3.2.2.2 template<class T> Array2D< T >::Array2D (const Array2D< T > & *rhs*) `[inline]`

Creates an array and sets it equal to another array.

Parameters:
- *rhs* array to set this array equal to.

Definition at line 251 of file Array2D.h.

3.2.3 Member Function Documentation

3.2.3.1 template<class T> void Array2D< T >::allocate (int *i*, int *j*) `[inline]`

Allocates memory for the array.

Parameters:
- *i* size of the array's first dimension.
- *j* size of the array's second dimension.

Definition at line 84 of file Array2D.h.

Referenced by Array2D< CubicSplineWithGeometricProgressionGrid >::Array2D(), and Array2D< CubicSplineWithGeometricProgressionGrid >::operator=().

3.2.3.2 template<class T> T* Array2D< T >::array () `[inline]`

Gets a pointer to an array containing the array elements.

The ordering of this array is NOT specified.

Definition at line 75 of file Array2D.h.

3.2.3.3 template<class T> int Array2D< T >::dim1 () `[inline]`

Gets the number of elements in the array's first dimension.

Returns:
number of elements in the array's first dimension.

Definition at line 55 of file Array2D.h.

Referenced by QMCJastrowElectronNuclear::evaluate(), QMCJastrowElectron-Electron::evaluate(), and QMCJastrow::evaluate().

3.2.3.4 template<class T> int Array2D< T >::dim2 () `[inline]`

Gets the number of elements in the array's second dimension.

Returns:
number of elements in the array's second dimension.

Definition at line 62 of file Array2D.h.

Referenced by QMCJastrow::evaluate().

3.2.3.5 template<class T> int Array2D< T >::size () `[inline]`

Gets the total number of elements in the array.

Returns:
total number of elements in the array.

Definition at line 69 of file Array2D.h.

3.3 Array3D< T > Class Template Reference

A 3-dimensional template for making arrays.

Public Methods

- int **dim1** ()

  Gets the number of elements in the array's first dimension.

- int **dim2** ()

  Gets the number of elements in the array's second dimension.

- int **dim3** ()

  Gets the number of elements in the array's third dimension.

- int **size** ()

  Gets the total number of elements in the array.

- T * **array** ()

  Gets a pointer to an array containing the array elements.

- void **allocate** (int i, int j, int k)

  Allocates memory for the array.

- void **deallocate** ()

  Deallocates memory for the array.

- void **operator=** (const Array3D &rhs)

  Sets two arrays equal.

- **Array3D** ()

  Creates an array.

- **Array3D** (int i, int j, int k)

  Creates an array and allocates memory.

- **Array3D** (const Array3D< T > &rhs)

  Creates an array and sets it equal to another array.

- **~Array3D** ()

  Destroy's the array and cleans up the memory.

- T & **operator()** (int i, int j, int k)

    Accesses element `(i,j,k)` *of the array.*

3.3.1 Detailed Description

template<class T> class Array3D< T >

A 3-dimensional template for making arrays.

All of the memory allocation and deallocation details are dealt with by the class.

Definition at line 23 of file Array3D.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 template<class T> Array3D< T >::Array3D (int *i*, int *j*, int *k*) `[inline]`

Creates an array and allocates memory.

Parameters:

i size of the array's first dimension.

j size of the array's second dimension.

k size of the array's third dimension.

Definition at line 175 of file Array3D.h.

3.3.2.2 template<class T> Array3D< T >::Array3D (const Array3D< T > & *rhs*) `[inline]`

Creates an array and sets it equal to another array.

Parameters:

rhs array to set this array equal to.

Definition at line 185 of file Array3D.h.

3.3.3 Member Function Documentation

3.3.3.1 template<class T> void Array3D< T >::allocate (int *i*, int *j*, int *k*) `[inline]`

Allocates memory for the array.

Parameters:

i size of the array's first dimension.

j size of the array's second dimension.

k size of the array's third dimension.

Definition at line 97 of file Array3D.h.

Referenced by Array3D< double >::Array3D(), and Array3D< double >::operator=().

3.3.3.2 template<class T> T* Array3D< T >::array () `[inline]`

Gets a pointer to an array containing the array elements.

The ordering of this array is NOT specified.

Definition at line 87 of file Array3D.h.

3.3.3.3 template<class T> int Array3D< T >::dim1 () `[inline]`

Gets the number of elements in the array's first dimension.

Returns:

number of elements in the array's first dimension.

Definition at line 60 of file Array3D.h.

3.3.3.4 template<class T> int Array3D< T >::dim2 () `[inline]`

Gets the number of elements in the array's second dimension.

Returns:

number of elements in the array's second dimension.

Definition at line 67 of file Array3D.h.

3.3.3.5 template<class T> int Array3D< T >::dim3 () `[inline]`

Gets the number of elements in the array's third dimension.

Returns:

number of elements in the array's third dimension.

Definition at line 74 of file Array3D.h.

3.3.3.6 template<class T> int Array3D< T >::size () `[inline]`

Gets the total number of elements in the array.

Returns:

total number of elements in the array.

Definition at line 81 of file Array3D.h.

3.4 Array4D< T > Class Template Reference

A 4-dimensional template for making arrays.

Public Methods

- int **dim1** ()

  Gets the number of elements in the array's first dimension.

- int **dim2** ()

  Gets the number of elements in the array's second dimension.

- int **dim3** ()

  Gets the number of elements in the array's third dimension.

- int **dim4** ()

  Gets the number of elements in the array's fourth dimension.

- int **size** ()

  Gets the total number of elements in the array.

- T * **array** ()

  Gets a pointer to an array containing the array elements.

- void **allocate** (int i, int j, int k, int l)

  Allocates memory for the array.

- void **deallocate** ()

  Deallocates memory for the array.

- void **operator=** (const Array4D &rhs)

  Sets two arrays equal.

- **Array4D** ()
  Creates an array.
- **Array4D** (int i, int j, int k, int l)
  Creates an array and allocates memory.
- **Array4D** (const Array4D &rhs)
  Creates an array and sets it equal to another array.
- **~Array4D** ()
  Destroy's the array and cleans up the memory.
- T & **operator()** (int i, int j, int k, int l)
  Accesses element `(i,j,k,l)` *of the array.*

3.4.1 Detailed Description

template<class T> class Array4D< T >

A 4-dimensional template for making arrays.

All of the memory allocation and deallocation details are dealt with by the class.

Definition at line 23 of file Array4D.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 template<class T> Array4D< T >::Array4D (int *i*, int *j*, int *k*, int *l*) `[inline]`

Creates an array and allocates memory.

Parameters:

i size of the array's first dimension.

j size of the array's second dimension.

k size of the array's third dimension.

l size of the array's fourth dimension.

Definition at line 205 of file Array4D.h.

References Array4D< T >::allocate().

3.4.2.2 template<class T> Array4D< T >::Array4D (const Array4D< T > & *rhs*) `[inline]`

Creates an array and sets it equal to another array.

Parameters:
 rhs array to set this array equal to.

Definition at line 215 of file Array4D.h.

References Array4D< T >::allocate(), Array4D< T >::n_1, Array4D< T >::n_2, Array4D< T >::n_3, Array4D< T >::n_4, and Array4D< T >::pArray.

3.4.3 Member Function Documentation

3.4.3.1 template<class T> void Array4D< T >::allocate (int *i*, int *j*, int *k*, int *l*) `[inline]`

Allocates memory for the array.

Parameters:
 i size of the array's first dimension.
 j size of the array's second dimension.
 k size of the array's third dimension.
 l size of the array's fourth dimension.

Definition at line 112 of file Array4D.h.

References Array4D< T >::deallocate().

Referenced by Array4D< T >::Array4D(), and Array4D< T >::operator=().

3.4.3.2 template<class T> T* Array4D< T >::array () `[inline]`

Gets a pointer to an array containing the array elements.

The ordering of this array is NOT specified.

Definition at line 101 of file Array4D.h.

3.4.3.3 template<class T> int Array4D< T >::dim1 () `[inline]`

Gets the number of elements in the array's first dimension.

Returns:
 number of elements in the array's first dimension.

Definition at line 67 of file Array4D.h.

3.4.3.4 template<class T> int Array4D< T >::dim2 () `[inline]`

Gets the number of elements in the array's second dimension.

Returns:
number of elements in the array's second dimension.

Definition at line 74 of file Array4D.h.

3.4.3.5 template<class T> int Array4D< T >::dim3 () `[inline]`

Gets the number of elements in the array's third dimension.

Returns:
number of elements in the array's third dimension.

Definition at line 81 of file Array4D.h.

3.4.3.6 template<class T> int Array4D< T >::dim4 () `[inline]`

Gets the number of elements in the array's fourth dimension.

Returns:
number of elements in the array's fourth dimension.

Definition at line 88 of file Array4D.h.

3.4.3.7 template<class T> int Array4D< T >::size () `[inline]`

Gets the total number of elements in the array.

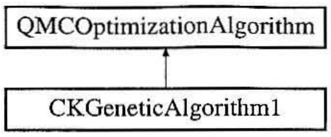
Returns:
total number of elements in the array.

Definition at line 95 of file Array4D.h.

3.5 CKGeneticAlgorithm1 Class Reference

A moderately greedy genetic algorithm for trying to globally optimize a function dreamed up by David Randall (Chip) Kent IV.

Inheritance diagram for CKGeneticAlgorithm1::



Public Methods

- **CKGeneticAlgorithm1** (**QMCObjectiveFunction** *function, int populationsize, double mutationrate, double distributionwidth)

  Constructs and inializes this optimization algorithm.

- **Array1D**< double > **optimize** (**Array1D**< double > &initialGuess)

  Optimize the function starting from the provided initial guess parameters.

3.5.1 Detailed Description

A moderately greedy genetic algorithm for trying to globally optimize a function dreamed up by David Randall (Chip) Kent IV.

As is standard in the field, optimization means minimization.

Mutation is accomplished by adding a N-dimensional gaussian random variable to the population member.

The amount of each parent contributed to a child is determined by a uniform random variable.

A linear probability distribution is used to select which population member will be a parent. The best members have better probabilities of being selected.

Definition at line 38 of file CKGeneticAlgorithm1.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 CKGeneticAlgorithm1::CKGeneticAlgorithm1 (QMCObjectiveFunction * *function*, int *populationsize*, double *mutationrate*, double *distributionwidth*)

Constructs and inializes this optimization algorithm.

Parameters:

function function to optimize.

populationsize number of members in the population used to optimize the function. This is a positive number.

mutationrate a positive number describing how much mutation is introduced into the population. Larger numbers correspond to more mutation.

distributionwidth a positive number describing how far the initial population members spread from the initial guess.

Definition at line 15 of file CKGeneticAlgorithm1.cpp.

3.5.3 Member Function Documentation

3.5.3.1 Array1D< double > CKGeneticAlgorithm1::optimize (Array1D< double > & *initialGuess*) `[virtual]`

Optimize the function starting from the provided initial guess parameters.

Parameters:

initialGuess initial guess parameters for the optimization.

Returns:

optimized parameters.

Implements **QMCOptimizationAlgorithm** (p. 107).

Definition at line 136 of file CKGeneticAlgorithm1.cpp.

References SortedParameterScorePairList::get(), ParameterScorePair::getParameters(), and ParameterScorePair::getScore().

3.6 Complex Class Reference

An implementation of a complex number with the associated basic functions.

Public Methods

- **Complex** ()

  Creates an object and initializes it to (0,0).

- **Complex** (double re, double im)

  Creates and initializes this object.

- **Complex** (const Complex &rhs)

  Creates an new instance of this object which is equal to another instance.

- double **real** ()

Real part of this number.

- double **imaginary** ()

  Imaginary part of this number.

- void **operator=** (const Complex &rhs)

  Sets two complex numbers equal.

- void **operator=** (const double &rhs)

  Sets a complex number and a real number equal.

- Complex **operator+** (const Complex &rhs)

  Adds two complex numbers.

- Complex **operator+** (const double &rhs)

  Adds a complex and a real number.

- Complex **operator-** (const Complex &rhs)

  Subtracts two complex numbers.

- Complex **operator-** (const double &rhs)

  Subtracts a complex and a real number.

- Complex **operator** * (const Complex &rhs)

  Multiplies two complex number.

- Complex **operator** * (const double &rhs)

  Multiplies a complex and a real number.

- Complex **operator/** (const Complex &rhs)

  Divides two complex numbers.

- Complex **conjugate** ()

  Calculates the complex conjugate of this number.

- double **abs** ()

  Calculates the magniutde of this complex number.

- Complex **squareroot** ()

  Calculates the square root of this complex number.

Friends

- ostream & **operator<<** (ostream &strm, Complex &c)
  Write the number to an output stream.

3.6.1 Detailed Description

An implementation of a complex number with the associated basic functions.

Definition at line 23 of file Complex.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Complex::Complex (double *re*, double *im*)

Creates and initializes this object.

Parameters:
re real part of this number.
im imaginary part of this number.

Definition at line 21 of file Complex.cpp.

3.6.2.2 Complex::Complex (const Complex & *rhs*)

Creates an new instance of this object which is equal to another instance.

Parameters:
rhs object this new object will be set equal to.

Definition at line 27 of file Complex.cpp.

3.6.3 Member Function Documentation

3.6.3.1 double Complex::abs ()

Calculates the magniutde of this complex number.

$$c.abs() = \sqrt{(c.re()^2 + c.im()^2)}$$

Returns:
magnitude of this complex number.

Definition at line 146 of file Complex.cpp.

3.6.3.2 Complex Complex::conjugate ()

Calculates the complex conjugate of this number.

Returns:
complex conjugate of this number.

Definition at line 136 of file Complex.cpp.

References im, and re.

3.6.3.3 double Complex::imaginary ()

Imaginary part of this number.

Returns:
imaginary part of this number.

Definition at line 37 of file Complex.cpp.

3.6.3.4 Complex Complex::operator * (const double & *rhs*)

Multiplies a complex and a real number.

Returns:
product of the arguments.

Definition at line 104 of file Complex.cpp.

References im, and re.

3.6.3.5 Complex Complex::operator * (const Complex & *rhs*)

Multiplies two complex number.

Returns:
product of the arguments.

Definition at line 94 of file Complex.cpp.

References im, and re.

3.6.3.6 Complex Complex::operator+ (const double & *rhs*)

Adds a complex and a real number.

Returns:
sum of the arguments.

Definition at line 64 of file Complex.cpp.

References im, and re.

3.6.3.7 Complex Complex::operator+ (const Complex & *rhs*)

Adds two complex numbers.

Returns:
sum of the arguments.

Definition at line 54 of file Complex.cpp.

References im, and re.

3.6.3.8 Complex Complex::operator- (const double & *rhs*)

Subtracts a complex and a real number.

Returns:
difference of the arguments.

Definition at line 84 of file Complex.cpp.

References im, and re.

3.6.3.9 Complex Complex::operator- (const Complex & *rhs*)

Subtracts two complex numbers.

Returns:
difference of the arguments.

Definition at line 74 of file Complex.cpp.

References im, and re.

3.6.3.10 Complex Complex::operator/ (const Complex & *rhs*)

Divides two complex numbers.

Returns:
result of the division.

Definition at line 114 of file Complex.cpp.

References im, and re.

3.6.3.11 void Complex::operator= (const double & *rhs*)

Sets a complex number and a real number equal.

@rhs number to set this one equal to.

Definition at line 48 of file Complex.cpp.

3.6.3.12 void Complex::operator= (const Complex & *rhs*)

Sets two complex numbers equal.

@rhs number to set this one equal to.

Definition at line 42 of file Complex.cpp.

References im, and re.

3.6.3.13 double Complex::real ()

Real part of this number.

Returns:
real part of this number.

Definition at line 32 of file Complex.cpp.

3.6.3.14 Complex Complex::squareroot ()

Calculates the square root of this complex number.

Returns:
square root of this complex number.

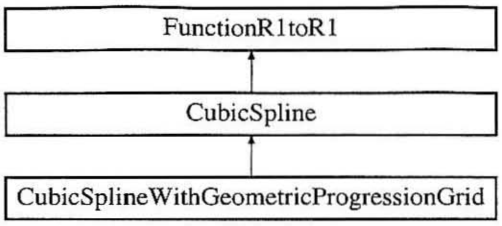
Definition at line 174 of file Complex.cpp.

References im, and re.

3.7 CubicSpline Class Reference

A 1-dimensional ($\mathbf{R}^1 \to \mathbf{R}^1$) cubic spline interpolation.

Inheritance diagram for CubicSpline::



Public Methods

- **CubicSpline** ()

  Creates an instance of this class.

- void **operator=** (const CubicSpline &rhs)

  Sets two CubicSpline objects equal.

- void **initializeWithFunctionValues** (**Array1D**< double > &xInput, **Array1D**< double > &yInput, double yPrimeFirst, double yPrimeLast)

  Initializes the spline with the function values at given points plus the derivative values at the end points.

- void **initializeWithDerivativeValues** (**Array1D**< double > &xInput, **Array1D**< double > &yPrimeInput, double yFirst)

  Initializes the spline with the derivative values at given points plus the function value at the first point.

- void **evaluate** (double x)

  Evaluates the function at x.

- double **getFunctionValue** ()

  Gets the function value at the last evaluated point.

- double **getFirstDerivativeValue** ()

  Gets the function's first deriviate at the last evaluated point.

- double **getSecondDerivativeValue** ()

  Gets the function's second deriviative at the last evaluated point.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

Protected Methods

- void **evaluate** (double x, int index)

  Evaluate the function at x when the index of the box of the domain containing x is known.

3.7.1 Detailed Description

A 1-dimensional ($\mathbf{R}^1 \rightarrow \mathbf{R}^1$) cubic spline interpolation.

Definition at line 30 of file CubicSpline.h.

3.7.2 Member Function Documentation

3.7.2.1 void CubicSpline::evaluate (double *x*, int *index*) `[protected]`

Evaluate the function at x when the index of the box of the domain containing x is known.

Parameters:

x point to evaluate the function.

index index of the box of the domain containing `x`.

Definition at line 404 of file CubicSpline.cpp.

3.7.2.2 void CubicSpline::evaluate (double *x*) `[virtual]`

Evaluates the function at x.

Parameters:

x point to evaluate the function.

Implements **FunctionR1toR1** (p. 41).

Reimplemented in **CubicSplineWithGeometricProgressionGrid** (p. 34).

Definition at line 375 of file CubicSpline.cpp.

Referenced by CubicSplineWithGeometricProgressionGrid::evaluate().

3.7.2.3 double CubicSpline::getFirstDerivativeValue () `[virtual]`

Gets the function's first deriviate at the last evaluated point.

Returns:
function's deriviative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 439 of file CubicSpline.cpp.

3.7.2.4 double CubicSpline::getFunctionValue () `[virtual]`

Gets the function value at the last evaluated point.

Returns:
function value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 434 of file CubicSpline.cpp.

3.7.2.5 double CubicSpline::getSecondDerivativeValue () `[virtual]`

Gets the function's second deriviative at the last evaluated point.

Returns:
function's second derivative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 444 of file CubicSpline.cpp.

3.7.2.6 void CubicSpline::initializeWithDerivativeValues (Array1D< double > & *xInput*, Array1D< double > & *yPrimeInput*, double *yFirst*)

Initializes the spline with the derivative values at given points plus the function value at the first point.

Parameters:
xInput x values of the given points.

yPrimeInput derivative values of the given points.

yFirst function value at the first point.

Definition at line 165 of file CubicSpline.cpp.

References Array1D< double >::allocate(), and Array1D< T >::dim1().

3.7.2.7 void CubicSpline::initializeWithFunctionValues (Array1D< double > & *xInput*, Array1D< double > & *yInput*, double *yPrimeFirst*, double *yPrimeLast*)

Initializes the spline with the function values at given points plus the derivative values at the end points.

Parameters:

xInput x values of the given points.

yInput y values of the given points.

yPrimeFirst derivative value at the first point.

yPrimeLast derivative value at the last point.

Definition at line 37 of file CubicSpline.cpp.

References Array1D< double >::allocate(), and Array1D< T >::dim1().

3.7.2.8 void CubicSpline::operator= (const CubicSpline & *rhs*)

Sets two CubicSpline objects equal.

Parameters:

rhs object to set this object equal to

Definition at line 18 of file CubicSpline.cpp.

References a0_list, a1_list, a2_list, a3_list, ddfddx, dfdx, f, n, x_list, y_list, yp0, yp_list, and ypend.

Referenced by CubicSplineWithGeometricProgressionGrid::operator=().

3.7.2.9 void CubicSpline::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:

strm XML stream

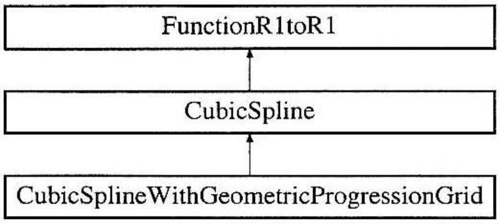
Definition at line 449 of file CubicSpline.cpp.

References Array1D< double >::dim1().

3.8 CubicSplineWithGeometricProgressionGrid Class Reference

A 1-dimensional ($\mathbf{R}^1 \to \mathbf{R}^1$) cubic spline interpolation with a grid that is assumed to be spaced according to a geometric relationship for faster evaluation.

Inheritance diagram for CubicSplineWithGeometricProgressionGrid::



Public Methods

- **CubicSplineWithGeometricProgressionGrid** ()

  Constructs an uninitialized spline.

- void **setGridParameters** (double beta, double x0)

  Sets the value for β and x_0 used in generating this grid.

- void **evaluate** (double x)

  Evaluates the function at x.

- void **operator=** (const CubicSplineWithGeometricProgressionGrid &rhs)

  Sets two CubicSplineWithGeometricProgressionGrid objects equal.

- void **initializeWithFunctionValues** (**Array1D**< double > &xInput, **Array1D**< double > &yInput, double yPrimeFirst, double yPrimeLast)

  Initializes the spline with the function values at given points plus the derivative values at the end points.

- void **initializeWithDerivativeValues** (**Array1D**< double > &xInput, **Array1D**< double > &yPrimeInput, double yFirst)

  Initializes the spline with the derivative values at given points plus the function value at the first point.

- double **getFunctionValue** ()

  Gets the function value at the last evaluated point.

- double **getFirstDerivativeValue** ()

  Gets the function's first deriviate at the last evaluated point.

- double **getSecondDerivativeValue** ()

  Gets the function's second deriviative at the last evaluated point.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

Protected Methods

- void **evaluate** (double x, int index)

  Evaluate the function at x when the index of the box of the domain containing x is known.

3.8.1 Detailed Description

A 1-dimensional ($\mathbf{R}^1 \rightarrow \mathbf{R}^1$) cubic spline interpolation with a grid that is assumed to be spaced according to a geometric relationship for faster evaluation.

$$x_{i+1} = \beta x_i$$

β is a user provided parameter and x_0 is set equal to the first datum used to initialize the spline.

Definition at line 30 of file CubicSplineWithGeometricProgressionGrid.h.

3.8.2 Member Function Documentation

3.8.2.1 void CubicSpline::evaluate (double x, int *index*) `[protected, inherited]`

Evaluate the function at x when the index of the box of the domain containing x is known.

Parameters:

x point to evaluate the function.

index index of the box of the domain containing x.

Definition at line 404 of file CubicSpline.cpp.

3.8.2.2 void CubicSplineWithGeometricProgressionGrid::evaluate (double x) `[virtual]`

Evaluates the function at x.

Parameters:

x point to evaluate the function.

Reimplemented from **CubicSpline** (p. 30).

Definition at line 29 of file CubicSplineWithGeometricProgressionGrid.cpp.

References CubicSpline::evaluate().

3.8.2.3 double CubicSpline::getFirstDerivativeValue () `[virtual, inherited]`

Gets the function's first deriviate at the last evaluated point.

Returns:

function's deriviative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 439 of file CubicSpline.cpp.

3.8.2.4 double CubicSpline::getFunctionValue () `[virtual, inherited]`

Gets the function value at the last evaluated point.

Returns:

function value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 434 of file CubicSpline.cpp.

3.8.2.5 double CubicSpline::getSecondDerivativeValue () `[virtual, inherited]`

Gets the function's second deriviative at the last evaluated point.

Returns:

function's second derivative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 444 of file CubicSpline.cpp.

3.8.2.6 void CubicSpline::initializeWithDerivativeValues (Array1D< double > & *xInput*, Array1D< double > & *yPrimeInput*, double *yFirst*) `[inherited]`

Initializes the spline with the derivative values at given points plus the function value at the first point.

Parameters:

xInput x values of the given points.

yPrimeInput derivative values of the given points.

yFirst function value at the first point.

Definition at line 165 of file CubicSpline.cpp.

References Array1D< double >::allocate(), and Array1D< T >::dim1().

3.8.2.7 void CubicSpline::initializeWithFunctionValues (Array1D< double > & *xInput*, Array1D< double > & *yInput*, double *yPrimeFirst*, double *yPrimeLast*) `[inherited]`

Initializes the spline with the function values at given points plus the derivative values at the end points.

Parameters:

xInput x values of the given points.

yInput y values of the given points.

yPrimeFirst derivative value at the first point.

yPrimeLast derivative value at the last point.

Definition at line 37 of file CubicSpline.cpp.

References Array1D< double >::allocate(), and Array1D< T >::dim1().

3.8.2.8 void CubicSplineWithGeometricProgressionGrid::operator= (const CubicSplineWithGeometricProgressionGrid & *rhs*)

Sets two CubicSplineWithGeometricProgressionGrid objects equal.

Parameters:

rhs object to set this object equal to

Definition at line 38 of file CubicSplineWithGeometricProgressionGrid.cpp.

References beta, CubicSpline::operator=(), and x0.

3.8.2.9 void CubicSplineWithGeometricProgressionGrid::setGridParameters (double *beta*, double *x0*)

Sets the value for β and x_0 used in generating this grid.

$$x_{i+1} = \beta x_i$$

Parameters:

beta the parameter used in generating the grid $x_{i+1} = \beta x_i$.

x0 the first point in the grid $x_{i+1} = \beta x_i$.

Definition at line 22 of file CubicSplineWithGeometricProgressionGrid.cpp.

3.8.2.10 void CubicSpline::toXML (ostream & *strm*) `[inherited]`

Writes the state of this object to an XML stream.

Parameters:

strm XML stream

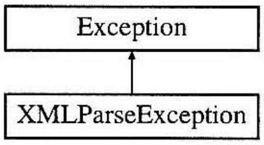
Definition at line 449 of file CubicSpline.cpp.

References Array1D< double >::dim1().

3.9 Exception Class Reference

An Exception is thrown when an error occurs.

Inheritance diagram for Exception::



Public Methods

- **Exception** ()

  Creates an exception.

- **Exception** (string message)

Creates an exception.

- void **setMessage** (string message)

  Sets the error message for the exception.

- string **getMessage** ()

  Gets the error message for the exception.

3.9.1 Detailed Description

An Exception is thrown when an error occurs.

This can be extended to deal with special types of errors.

Definition at line 23 of file Exception.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Exception::Exception (string *message*)

Creates an exception.

Parameters:

message A message describing what went wrong.

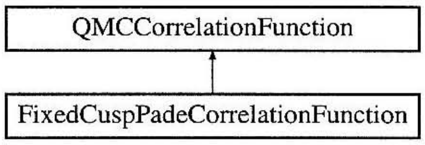
Definition at line 19 of file Exception.cpp.

References setMessage().

3.10 FixedCuspPadeCorrelationFunction Class Reference

Correlation function which uses a Pade expansion to describe particle-particle interactions.

Inheritance diagram for FixedCuspPadeCorrelationFunction::



Public Methods

- void **initializeParameters** (**Array1D**< int > &BeginningIndexOfParameterType, **Array1D**< double > &Parameters, **Array1D**< int > &BeginningIndexOfConstantType, **Array1D**< double > &Constants)

  Initializes the correlation function with a specified set of parameters.

- void **evaluate** (double r)

  Evaluates the correlation function and it's first two derivatives at r.

- bool **isSingular** ()

  Returns true if the correlation function has a singularity in the domain $r \geq 0$, and false otherwise.

- double **getFunctionValue** ()

  Gets the value of the correlation function for the last evaluated r.

- double **getFirstDerivativeValue** ()

  Gets the value of the first derivative of the correlation function for the last evaluated r.

- double **getSecondDerivativeValue** ()

  Gets the value of the second derivative of the correlation function for the last evaluated r.

3.10.1 Detailed Description

Correlation function which uses a Pade expansion to describe particle-particle interactions.

The cusp condition is a fixed constant, and all other parameters will be adjusted during an optimization.

Definition at line 26 of file FixedCuspPadeCorrelationFunction.h.

3.10.2 Member Function Documentation

3.10.2.1 void FixedCuspPadeCorrelationFunction::initializeParameters (Array1D< int > & *BeginningIndexOfParameterType*, Array1D< double > & *Parameters*, Array1D< int > & *BeginningIndexOfConstantType*, Array1D< double > & *Constants*) `[virtual]`

Initializes the correlation function with a specified set of parameters.

This must be called every time the parameters are changed.

Implements **QMCCorrelationFunction** (p. 58).

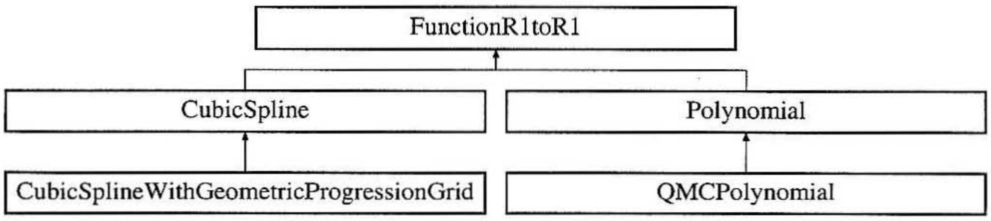
Definition at line 15 of file FixedCuspPadeCorrelationFunction.cpp.

References Array1D< T >::dim1(), and Polynomial::initialize().

3.11 FunctionR1toR1 Class Reference

An interface for a function from $\mathbf{R}^1 \rightarrow \mathbf{R}^1$.

Inheritance diagram for FunctionR1toR1::



Public Methods

- virtual ~**FunctionR1toR1** ()
  Virtual destructor.
- virtual void **evaluate** (double x)=0
  Evaluates the function at x.
- virtual double **getFunctionValue** ()=0
  Gets the function value at the last evaluated point.
- virtual double **getFirstDerivativeValue** ()=0
  Gets the function's first deriviate at the last evaluated point.
- virtual double **getSecondDerivativeValue** ()=0
  Gets the function's second deriviative at the last evaluated point.

3.11.1 Detailed Description

An interface for a function from $\mathbf{R}^1 \rightarrow \mathbf{R}^1$.

Definition at line 24 of file FunctionR1toR1.h.

3.11.2 Member Function Documentation

3.11.2.1 virtual void FunctionR1toR1::evaluate (double *x*) `[pure virtual]`

Evaluates the function at x.

Parameters:
x point to evaluate the function.

Implemented in **CubicSpline** (p. 30), **CubicSplineWithGeometricProgressionGrid** (p. 34), and **Polynomial** (p. 47).

3.11.2.2 virtual double FunctionR1toR1::getFirstDerivativeValue () `[pure virtual]`

Gets the function's first deriviate at the last evaluated point.

Returns:
function's deriviative value.

Implemented in **CubicSpline** (p. 30), and **Polynomial** (p. 47).

3.11.2.3 virtual double FunctionR1toR1::getFunctionValue () `[pure virtual]`

Gets the function value at the last evaluated point.

Returns:
function value.

Implemented in **CubicSpline** (p. 31), and **Polynomial** (p. 48).

3.11.2.4 virtual double FunctionR1toR1::getSecondDerivativeValue () `[pure virtual]`

Gets the function's second deriviative at the last evaluated point.

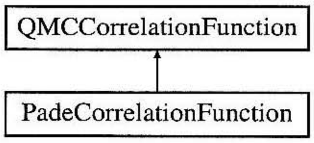
Returns:
function's second derivative value.

Implemented in **CubicSpline** (p. 31), and **Polynomial** (p. 48).

3.12 PadeCorrelationFunction Class Reference

Correlation function which uses a Pade expansion to describe particle-particle interactions.

Inheritance diagram for PadeCorrelationFunction::



Public Methods

- void **initializeParameters** (**Array1D**< int > &BeginningIndexOfParameterType, **Array1D**< double > &Parameters, **Array1D**< int > &BeginningIndexOfConstantType, **Array1D**< double > &Constants)

  Initializes the correlation function with a specified set of parameters.

- void **evaluate** (double r)

  Evaluates the correlation function and it's first two derivatives at r.

- bool **isSingular** ()

  Returns true if the correlation function has a singularity in the domain $r \geq 0$, and false otherwise.

- double **getFunctionValue** ()

  Gets the value of the correlation function for the last evaluated r.

- double **getFirstDerivativeValue** ()

  Gets the value of the first derivative of the correlation function for the last evaluated r.

- double **getSecondDerivativeValue** ()

  Gets the value of the second derivative of the correlation function for the last evaluated r.

3.12.1 Detailed Description

Correlation function which uses a Pade expansion to describe particle-particle interactions.

All parameters will be adjusted during an optimization.

Definition at line 26 of file PadeCorrelationFunction.h.

3.12.2 Member Function Documentation

3.12.2.1 void PadeCorrelationFunction::initializeParameters (Array1D< int > & *BeginningIndexOfParameterType*, Array1D< double > & *Parameters*, Array1D< int > & *BeginningIndexOfConstantType*, Array1D< double > & *Constants*) `[virtual]`

Initializes the correlation function with a specified set of parameters.

This must be called every time the parameters are changed.

Implements **QMCCorrelationFunction** (p. 58).

Definition at line 15 of file PadeCorrelationFunction.cpp.

References Array1D< T >::dim1(), and Polynomial::initialize().

3.13 ParameterScorePair Class Reference

A container which holds a set of parameters and an associated scalar score value.

Public Methods

- **ParameterScorePair** ()

  Creates an uninitialized instance of this class with no allocated memory.

- **ParameterScorePair** (double score, **Array1D**< double > &parameters)

  Creates an uninitialized instance of this class and sets the score and parameter values.

- **ParameterScorePair** (const ParameterScorePair &PSP)

  Creates an instance of this class which is equal to another instance.

- double **getScore** ()

  Gets the score.

- **Array1D**< double > * **getParameters** ()

  Gets the parameters.

- void **operator=** (const ParameterScorePair &rhs)

  Set two ParameterScorePair objects equal.

- bool **operator<** (ParameterScorePair &PSP)

  An operator which orders ParameterScorePair objects based on their scores.

3.13.1 Detailed Description

A container which holds a set of parameters and an associated scalar score value.

Definition at line 23 of file ParameterScorePair.h.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 ParameterScorePair::ParameterScorePair (double *score*, Array1D< double > & *parameters*)

Creates an uninitialized instance of this class and sets the score and parameter values.

Parameters:

score Score.

parameters Parameters.

Definition at line 19 of file ParameterScorePair.cpp.

3.13.3 Member Function Documentation

3.13.3.1 Array1D< double > * ParameterScorePair::getParameters ()

Gets the parameters.

Returns:

paramters.

Definition at line 57 of file ParameterScorePair.cpp.

Referenced by CKGeneticAlgorithm1::optimize().

3.13.3.2 double ParameterScorePair::getScore ()

Gets the score.

Returns:

score.

Definition at line 52 of file ParameterScorePair.cpp.

Referenced by CKGeneticAlgorithm1::optimize().

3.13.3.3 void ParameterScorePair::operator= (const ParameterScorePair & *rhs*)

Set two ParameterScorePair objects equal.

Parameters:

rhs object to set this object equal to.

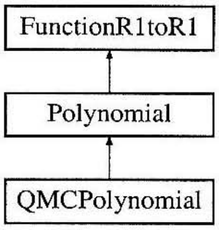
Definition at line 31 of file ParameterScorePair.cpp.

References Parameters, and Score.

3.14 Polynomial Class Reference

A one dimensional real polynomial.

Inheritance diagram for Polynomial::



Public Methods

- **Polynomial** ()

  Constructs an uninitialized instance of this class.

- **Polynomial** (**Array1D**< double > &coeffs)

  Constructs and initializes an intance of this class.

- void **initialize** (**Array1D**< double > &coeffs)

  Initializes this object.

- void **evaluate** (double x)

  Evaluates the function at x.

- double **getFunctionValue** ()

  Gets the function value at the last evaluated point.

- double **getFirstDerivativeValue** ()

  Gets the function's first deriviate at the last evaluated point.

- double **getSecondDerivativeValue** ()

  Gets the function's second deriviative at the last evaluated point.

- **Array1D< Complex > getRoots** ()

  Gets the roots of the polynomial.

Protected Methods

- int **getNumberCoefficients** ()

  Gets the number of coefficients in the polynomial.

- double **getCoefficient** (int i)

  Gets the ith coefficient of the polynomial.

3.14.1 Detailed Description

A one dimensional real polynomial.

$$P(x) = \sum_{i=0}^{n} c_i x^i$$

Definition at line 28 of file Polynomial.h.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 Polynomial::Polynomial (Array1D< double > & *coeffs*)

Constructs and initializes an intance of this class.

Parameters:

coeffs set of polynomial coefficients to use for the polynomial.

Definition at line 20 of file Polynomial.cpp.

References initialize().

3.14.3 Member Function Documentation

3.14.3.1 void Polynomial::evaluate (double *x*) [virtual]

Evaluates the function at x.

Parameters:
x point to evaluate the function.

Implements **FunctionR1toR1** (p. 41).

Definition at line 61 of file Polynomial.cpp.

Referenced by PadeCorrelationFunction::evaluate(), FixedCuspPadeCorrelationFunction::evaluate(), getFirstDerivativeValue(), getFunctionValue(), and getSecondDerivativeValue().

3.14.3.2 double Polynomial::getCoefficient (int *i*) [protected]

Gets the ith coefficient of the polynomial.

Where the polynomial is defined such that

$$P(x) = \sum_{i=0}^{n} c_i x^i$$

where n is the order of the polynomial and c_i is the ith coefficient.

Parameters:
i index of the coefficient to return.

Returns:
ith coefficient of the polynomial.

Definition at line 122 of file Polynomial.cpp.

3.14.3.3 double Polynomial::getFirstDerivativeValue () [virtual]

Gets the function's first deriviate at the last evaluated point.

Returns:
function's deriviative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 97 of file Polynomial.cpp.

References evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPadeCorrelationFunction::evaluate().

3.14.3.4 double Polynomial::getFunctionValue () [virtual]

Gets the function value at the last evaluated point.

Returns:
function value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 87 of file Polynomial.cpp.

References evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPadeCorrelationFunction::evaluate().

3.14.3.5 int Polynomial::getNumberCoefficients () [protected]

Gets the number of coefficients in the polynomial.

This is one larger than the order of the polynomial.

Returns:
number of coefficients in the polynomial.

Definition at line 117 of file Polynomial.cpp.

References Array1D< double >::dim1().

3.14.3.6 Array1D< Complex > Polynomial::getRoots ()

Gets the roots of the polynomial.

Returns:
roots of the polynomial.

Exceptions:
Exception (p. 37) if problems were encountered during the root calculation.

Definition at line 127 of file Polynomial.cpp.

References Array1D< double >::dim1().

Referenced by QMCPolynomial::hasNonNegativeZeroes().

3.14.3.7 double Polynomial::getSecondDerivativeValue () [virtual]

Gets the function's second deriviative at the last evaluated point.

Returns:
function's second derivative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 107 of file Polynomial.cpp.

References evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPadeCorrelationFunction::evaluate().

3.14.3.8 void Polynomial::initialize (Array1D< double > & *coeffs*)

Initializes this object.

Parameters:
coeffs set of polynomial coefficients to use for the polynomial.

Definition at line 39 of file Polynomial.cpp.

References Array1D< double >::allocate(), and Array1D< double >::dim1().

Referenced by PadeCorrelationFunction::initializeParameters(), FixedCuspPadeCorrelationFunction::initializeParameters(), and Polynomial().

3.15 QMCBasisFunction Class Reference

This class stores all of the parameters that a gaussian basis set is constructed from for a MOLECULE.

Public Methods

- **QMCBasisFunction** ()

  Creates an instance of the class.

- void **initialize** (QMCflags *flags, **QMCMolecule** *molecule)

  Initializes the class with data input to control the calculation and provide the molecular geometry.

- double **getPsi** (int whichBF, **Array2D**< double > &X, int elNumber)

  Calculates the value of a basis function.

- **Array1D**< double > **getGradPsi** (int whichBF, **Array2D**< double > &X, int elNumber)

  Calculates the gradient of a basis function.

- double **getLaplacianPsi** (int whichBF, **Array2D**< double > &X, int elNumber)

  Calculates the laplacian of a basis function.

- void **operator=** (const QMCBasisFunction &rhs)

  Sets two QMCBasisFunctions objects equal.

- void **read** (string runfile)

  Loads the state of the object from a file.

- int **getNumberBasisFunctions** (int i)

  Returns how many basis functions are located on a specific atom.

Friends

- istream & **operator>>** (istream &strm, QMCBasisFunction &rhs)

  Loads the state of the object from an input stream.

- ostream & **operator<<** (ostream &strm, QMCBasisFunction &rhs)

  Writes the state of the object to an output stream.

3.15.1 Detailed Description

This class stores all of the parameters that a gaussian basis set is constructed from for a MOLECULE.

This contains a QMCBasisFunctionCoefficent for each atom type.

Definition at line 36 of file QMCBasisFunction.h.

3.15.2 Member Function Documentation

3.15.2.1 Array1D< double > QMCBasisFunction::getGradPsi (int *whichBF*, Array2D< double > & *X*, int *elNumber*)

Calculates the gradient of a basis function.

Parameters:

whichBF which basis function to evaluate

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

elNumber which electron in X to calculate the basis function for

Returns:
basis function gradient value

Definition at line 361 of file QMCBasisFunction.cpp.

References QMCMolecule::Atom_Positions.

3.15.2.2 double QMCBasisFunction::getLaplacianPsi (int *whichBF*, Array2D< double > & *X*, int *elNumber*)

Calculates the laplacian of a basis function.

Parameters:
whichBF which basis function to evaluate
X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix
elNumber which electron in X to calculate the basis function for

Returns:
basis function laplacian value

Definition at line 377 of file QMCBasisFunction.cpp.

References QMCMolecule::Atom_Positions.

3.15.2.3 int QMCBasisFunction::getNumberBasisFunctions (int *i*)

Returns how many basis functions are located on a specific atom.

This can probably be depricated once we have a good initialization scheme and not MikesJacked one.

Parameters:
i index of atom

Returns:
number of basis functions on the atom

Definition at line 128 of file QMCBasisFunction.cpp.

References QMCBasisFunctionCoefficients::getNumberBasisFunctions().

3.15.2.4 double QMCBasisFunction::getPsi (int *whichBF*, Array2D< double > & *X*, int *elNumber*)

Calculates the value of a basis function.

Parameters:
whichBF which basis function to evaluate

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

elNumber which electron in X to calculate the basis function for

Returns:
basis function value

Definition at line 346 of file QMCBasisFunction.cpp.

References QMCMolecule::Atom_Positions.

3.15.2.5 void QMCBasisFunction::initialize (QMCflags * *flags*, QMCMolecule * *molecule*)

Initializes the class with data input to control the calculation and provide the molecular geometry.

Parameters:

flags input control information

molecule information about the specific molecule

Definition at line 19 of file QMCBasisFunction.cpp.

References Array1D< double >::allocate().

3.15.2.6 void QMCBasisFunction::operator= (const QMCBasisFunction & *rhs*)

Sets two QMCBasisFunctions objects equal.

Parameters:

rhs object to set this object equal to

Definition at line 116 of file QMCBasisFunction.cpp.

References BFCoeffs, BFLookupTable, flags, Molecule, N_BasisFunctions, Splines, use_splines, and Xcalc.

3.15.2.7 void QMCBasisFunction::read (string *runfile*)

Loads the state of the object from a file.

Parameters:

runfile file to load

Definition at line 175 of file QMCBasisFunction.cpp.

3.16 QMCBasisFunctionCoefficients Class Reference

This class stores all of the parameters that a gaussian basis set is constructed from for an ATOM.

Public Methods

- **QMCBasisFunctionCoefficients** ()

  Creates an instance of the class.

- int **getNumberBasisFunctions** ()

  Gets the number of basis functions.

- void **operator=** (const QMCBasisFunctionCoefficients &rhs)

  Sets two QMCBasisFunctionCoefficients objects equal.

- void **read** (string runfile)

  Loads the state of the object from a file.

Public Attributes

- **Array3D**< double > **Coeffs**

  Array containing the parameters for the basis functions where Coeffs[bf #][Gaussian #][0=exp,1=contract].

- **Array2D**< int > **xyz_powers**

  Array containing the k,l,m parameters which indicate the "angular momentum state" of the basis function (bf $= x^k y^l z^m *$ *RadialFunction(r)) where xyz[bf #][0=k,1=l,2=m].*

- **Array1D**< int > **N_Gauss**

  Array containing the number of gaussians that need to be contracted for the radial portion of the basis function (bf $= x^k y^l z^m *$*RadialFunction(r)) where N_Gauss[bf #].*

- **Array1D**< string > **Type**

  Array containing the type of the basis function where Type[bf #].

Friends

- istream & **operator>>** (istream &strm, QMCBasisFunctionCoefficients &rhs)

  Loads the state of the object from an input stream.

- ostream & **operator<<** (ostream &strm, QMCBasisFunctionCoefficients &rhs)

  Writes the state of the object to an output stream.

3.16.1 Detailed Description

This class stores all of the parameters that a gaussian basis set is constructed from for an ATOM.

For example, a gaussian basis function is

$$Gbf(x,y,z) = x^k y^l z^m \sum_{i=0}^{Ngaussians-1} a_i e^{-b_i r^2}$$

where k,l,m are determined by the type of basis function, a_i is the contraction parameter, and b_i is the exponential parameter. The particular contraction parameter is chosen so that the basis function is normalized. This is slightly different than what is common with linear algebra quantum mechanics programs. The contraction parameters used here can be obtained using the contraction and exponential parameters and k,l,m from a linear algebra basis file. You will have to look up the formula for doing this.

This reads in basis function coefficients in the following format...

```
AtomLabel  Number_of_orbitals Maximum_Gaussians

Ngaussians  Type
exp_param   contraction_param
...         ...

Ngaussians  Type
exp_param   contraction_param
...         ...

etc...
```

Definition at line 49 of file QMCBasisFunctionCoefficients.h.

3.16.2 Member Function Documentation

3.16.2.1 int QMCBasisFunctionCoefficients::getNumberBasisFunctions ()

Gets the number of basis functions.

Returns:
number of basis functions

Definition at line 20 of file QMCBasisFunctionCoefficients.cpp.

Referenced by QMCBasisFunction::getNumberBasisFunctions().

3.16.2.2 void QMCBasisFunctionCoefficients::operator= (const QMCBasisFunctionCoefficients & *rhs*)

Sets two QMCBasisFunctionCoefficients objects equal.

Parameters:
rhs object to set this object equal to

Definition at line 25 of file QMCBasisFunctionCoefficients.cpp.

References Coeffs, Label, Max_Gaussians, N_Gauss, N_Orbitals, Type, and xyz_powers.

3.16.3 Member Data Documentation

3.16.3.1 Array1D<string> QMCBasisFunctionCoefficients::Type

Array containing the type of the basis function where Type[bf #].

The type is a string representation of the "angular momentum state." For example, "px", "dxy", and "fxxx" are all types of basis functions.

Definition at line 101 of file QMCBasisFunctionCoefficients.h.

Referenced by operator=().

3.16.3.2 Array2D<int> QMCBasisFunctionCoefficients::xyz_powers

Array containing the k,l,m parameters which indicate the "angular momentum state" of the basis function $(bf = x^k y^l z^m * RadialFunction(r))$ where xyz[bf #][0=k,1=l,2=m].

For example, a "px" orbital would have $(k, l, m) = (1, 0, 0)$.

Definition at line 83 of file QMCBasisFunctionCoefficients.h.

Referenced by operator=().

3.17 QMCCopyright Class Reference

Central localtion for all copyright information relevant to QMcBeaver.

Friends

- ostream & **operator**<< (ostream &strm, QMCCopyright &rhs)

  Writes the copyright information to a stream in a human readable format.

3.17.1 Detailed Description

Central localtion for all copyright information relevant to QMcBeaver.

Definition at line 25 of file QMCCopyright.h.

3.18 QMCCorrelatedSamplingVMCOptimization Class Reference

Optimize the parameters in a variational QMC (VMC) calculation using the correlated sampling method.

Static Public Methods

- void **optimize** (QMCInput *input)

  Optimizes the parameters in a variational QMC (VMC) calculation using the correlated sampling method.

3.18.1 Detailed Description

Optimize the parameters in a variational QMC (VMC) calculation using the correlated sampling method.

Definition at line 26 of file QMCCorrelatedSamplingVMCOptimization.h.

3.18.2 Member Function Documentation

3.18.2.1 void QMCCorrelatedSamplingVMCOptimization::optimize (QMCInput * *input*) [static]

Optimizes the parameters in a variational QMC (VMC) calculation using the correlated sampling method.

Parameters:
input data input to control the calculation.

Definition at line 15 of file QMCCorrelatedSamplingVMCOptimization.cpp.

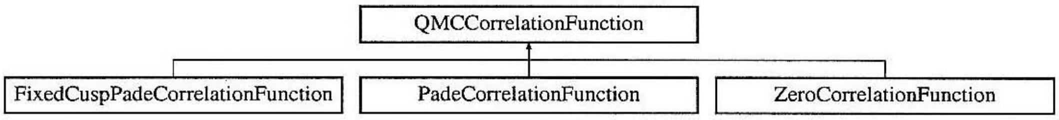
References Array1D< T >::array(), Array1D< T >::dim1(), QMCObjective-Function::initialize(), QMCOptimizationFactory::optimizationAlgorithm-Factory(), QMCOptimizationAlgorithm::optimize(), and QMCReadAnd-EvaluateConfigs::workerCalculateProperties().

Referenced by QMCManager::optimize().

3.19 QMCCorrelationFunction Class Reference

Interface for a parameterized function describing the interaction of two particles.

Inheritance diagram for QMCCorrelationFunction::



Public Methods

- virtual ~**QMCCorrelationFunction** ()

  Virtual destructor.

- virtual void **initializeParameters** (**Array1D**< int > &Beginning-IndexOfParameterType, **Array1D**< double > &Parameters, **Array1D**< int > &BeginningIndexOfConstantType, **Array1D**< double > &Con-stants)=0

  Initializes the correlation function with a specified set of parameters.

- virtual bool **isSingular** ()=0

  Returns true if the correlation function has a singularity in the domain $r \geq 0$, and false otherwise.

- virtual void **evaluate** (double r)=0

  Evaluates the correlation function and it's first two derivatives at r.

- virtual double **getFunctionValue** ()=0

  Gets the value of the correlation function for the last evaluated r.

- virtual double **getFirstDerivativeValue** ()=0

  Gets the value of the first derivative of the correlation function for the last evaluated r.

- virtual double **getSecondDerivativeValue** ()=0

  Gets the value of the second derivative of the correlation function for the last evaluated r.

3.19.1 Detailed Description

Interface for a parameterized function describing the interaction of two particles.

The trial wavefunction for QMC is $\Psi_{QMC} = \Psi_{Trial} J$ where $J = exp(\sum u_{i,j}(r_{i,j}))$. $u_{ij}(r_{ij})$ are the QMCCorrelationFunctions describing the interactions of particles i and j.

Definition at line 27 of file QMCCorrelationFunction.h.

3.19.2 Member Function Documentation

3.19.2.1 virtual void QMCCorrelationFunction::initializeParameters (Array1D< int > & *BeginningIndexOfParameterType*, Array1D< double > & *Parameters*, Array1D< int > & *BeginningIndexOfConstantType*, Array1D< double > & *Constants*) `[pure virtual]`

Initializes the correlation function with a specified set of parameters.

This must be called every time the parameters are changed.

Implemented in **FixedCuspPadeCorrelationFunction** (p. 39), **PadeCorrelationFunction** (p. 43), and **ZeroCorrelationFunction** (p. 164).

3.20 QMCCorrelationFunctionFactory Class Reference

Object factory which returns the correct **QMCCorrelationFunction** (p. 57) when a string keyword describing the correlation function is provided.

Static Public Methods

- **QMCCorrelationFunction** ∗ **correlationFunctionFactory** (string &Type)

  Returns the correct **QMCCorrelationFunction** (p. 57) *when a string keyword describing the correlation function is provided.*

3.20.1 Detailed Description

Object factory which returns the correct **QMCCorrelationFunction** (p. 57) when a string keyword describing the correlation function is provided.

Definition at line 31 of file QMCCorrelationFunctionFactory.h.

3.21 QMCCorrelationFunctionParameters Class Reference

This is a collection of parameters and related functions which describe the interaction of two particles of specific types.

Public Methods

- **QMCCorrelationFunctionParameters** ()

  Creates an instance of the class.

- **QMCCorrelationFunctionParameters** (const QMCCorrelationFunctionParameters &rhs)

  Creates an instance of the class that is identical to another instance of the class.

- ~**QMCCorrelationFunctionParameters** ()

  Deallocates all of the memory used by the object and prepares it to be destroyed.

- **Array1D**< double > **getParameters** ()

  Gets the parameters describing the particle-particle interactions.

- string **getParticle1Type** ()

  Gets the first particle type in a particle1-particle2 interaction described by this object.

- string **getParticle2Type** ()

Gets the second particle type in a particle1-particle2 interaction described by this object.

- int **getTotalNumberOfParameters** ()

  Gets the total number of parameters used to describe the particle-particle interaction.

- **QMCCorrelationFunction** * **getCorrelationFunction** ()

  Gets the parameterized **QMCCorrelationFunction** (p. 57) *used in* **QMC-Jastrow** (p. 74) *to describe the particular particle-particle interaction when calculating the Jastrow function.*

- void **setParameters** (**Array1D**< double > &params)

  Sets the parameters describing the particle-particle interaction.

- void **setParticle1Type** (string val)

  Sets the type of particle1 for the particular particle-particle interaction described by this object.

- void **setParticle2Type** (string val)

  Sets the type of particle2 for the particular particle-particle interaction described by this object.

- bool **isSingular** ()

  Returns true if the parameterized correlation function described by this object is singular on the positive real axis and false otherwise.

- void **operator=** (const QMCCorrelationFunctionParameters &rhs)

  Sets two QMCCorrelationFunctionParameters objects equal.

- void **read** (istream &strm)

  Loads the state of the object from an input stream.

Friends

- ostream & **operator<<** (ostream &strm, QMCCorrelationFunction-Parameters &rhs)

  Writes the state of the object to an output stream.

3.21.1 Detailed Description

This is a collection of parameters and related functions which describe the interaction of two particles of specific types.

For example, an instance of this class could hold the information describing the interaction of an up spin electron and a hydrogen nucleus or two down spin electrons.

The interactions are parameterized in terms of "parameters" and "constants." "parameters" are modified during optimizations, and "constants" are not.

Definition at line 36 of file QMCCorrelationFunctionParameters.h.

3.21.2 Constructor & Destructor Documentation

3.21.2.1 QMCCorrelationFunctionParameters::QMCCorrelationFunctionParameters (const QMCCorrelationFunctionParameters & *rhs*)

Creates an instance of the class that is identical to another instance of the class.

Parameters:
rhs object to copy

Definition at line 250 of file QMCCorrelationFunctionParameters.cpp.

3.21.3 Member Function Documentation

3.21.3.1 QMCCorrelationFunction * QMCCorrelationFunctionParameters::getCorrelationFunction ()

Gets the parameterized **QMCCorrelationFunction** (p. 57) used in **QMCJastrow** (p. 74) to describe the particular particle-particle interaction when calculating the Jastrow function.

Returns:
function describing **getParticle1Type()** (p. 62)-**getParticle2Type()** (p. 62) interactions

Definition at line 307 of file QMCCorrelationFunctionParameters.cpp.

Referenced by QMCJastrowElectronElectron::evaluate().

3.21.3.2 Array1D< double > QMCCorrelationFunctionParameters::getParameters ()

Gets the parameters describing the particle-particle interactions.

Returns:
parameters describing particle-particle interactions.

Definition at line 15 of file QMCCorrelationFunctionParameters.cpp.

Referenced by QMCJastrowParameters::getParameters().

3.21.3.3 string QMCCorrelationFunctionParameters::getParticle1Type ()

Gets the first particle type in a particle1-particle2 interaction described by this object.

Returns:
particle type

Definition at line 257 of file QMCCorrelationFunctionParameters.cpp.

Referenced by QMCJastrowParameters::read().

3.21.3.4 string QMCCorrelationFunctionParameters::getParticle2Type ()

Gets the second particle type in a particle1-particle2 interaction described by this object.

Returns:
particle type

Definition at line 262 of file QMCCorrelationFunctionParameters.cpp.

Referenced by QMCJastrowParameters::read().

3.21.3.5 int QMCCorrelationFunctionParameters::getTotalNumberOfParameters ()

Gets the total number of parameters used to describe the particle-particle interaction.

Returns:
total number of parameters

Definition at line 267 of file QMCCorrelationFunctionParameters.cpp.

Referenced by QMCJastrowParameters::getParameters(), and QMCJastrowParameters::setParameterVector().

3.21.3.6 bool QMCCorrelationFunctionParameters::isSingular ()

Returns true if the parameterized correlation function described by this object is singular on the positive real axis and false otherwise.

Returns:
true if the current parameterization of the correlation function is singular on the positive real axis and false otherwise

Definition at line 326 of file QMCCorrelationFunctionParameters.cpp.

References QMCCorrelationFunction::isSingular().

3.21.3.7 void QMCCorrelationFunctionParameters::operator= (const QMCCorrelationFunctionParameters & *rhs*)

Sets two QMCCorrelationFunctionParameters objects equal.

Parameters:
rhs object to set this object eqal to

Definition at line 20 of file QMCCorrelationFunctionParameters.cpp.

References BeginningIndexOfConstantType, BeginningIndexOfParameterType, Constants, CorrelationFunctionType, NumberOfConstants, NumberOfConstantTypes, NumberOfParameters, NumberOfParameterTypes, Parameters, ParticleTypes, TotalNumberOfConstants, and TotalNumberOfParameters.

3.21.3.8 void QMCCorrelationFunctionParameters::read (istream & *strm*)

Loads the state of the object from an input stream.

Parameters:
strm input stream

Definition at line 54 of file QMCCorrelationFunctionParameters.cpp.

References Array1D< double >::allocate(), Array1D< int >::allocate(), Array1D< string >::allocate(), Array1D< double >::deallocate(), Array1D< int >::deallocate(), and StringManipulation::toFirstUpperRestLower().

Referenced by QMCJastrowParameters::read().

3.21.3.9 void QMCCorrelationFunctionParameters::setParameters (Array1D< double > & *params*)

Sets the parameters describing the particle-particle interaction.

Parameters:

params new set of parameters

Definition at line 272 of file QMCCorrelationFunctionParameters.cpp.

References Array1D< double >::dim1(), and Array1D< T >::dim1().

Referenced by QMCJastrowParameters::setParameterVector().

3.22 QMCDerivativeProperties Class Reference

All of the calculated quantities and properties that are derived from quantities and properties evaluated during a calculation.

Public Methods

- **QMCDerivativeProperties** (**QMCproperties** *properties, double dt)

  Creates and initializes an instance of this class.

- double **getEffectiveTimeStep** ()

  Gets the effective time step for the calculation.

- double **getEffectiveTimeStepVariance** ()

  Gets the variance of the calculated effective time step for the calculation.

- double **getEffectiveTimeStepStandardDeviation** ()

  Gets the standard deviation of the calculated effective time step for the calculation.

- double **getVirialRatio** ()

  Gets the virial ratio for the calculation.

- double **getVirialRatioVariance** ()

  Gets the variance of the calculated virial ratio for the calculation.

- double **getVirialRatioStandardDeviation** ()

  Gets the standard deviation of the calculated virial ratio for the calculation.

Friends

- ostream & **operator<<** (ostream &strm, QMCDerivativeProperties &rhs)

  Formats and prints the properties to a stream in human readable fromat.

3.22.1 Detailed Description

All of the calculated quantities and properties that are derived from quantities and properties evaluated during a calculation.

Definition at line 23 of file QMCDerivativeProperties.h.

3.22.2 Constructor & Destructor Documentation

3.22.2.1 QMCDerivativeProperties::QMCDerivativeProperties (QMCproperties * *properties*, double *dt*)

Creates and initializes an instance of this class.

Parameters:

properties calculated properties for the system.

dt time step for the calculation.

Definition at line 16 of file QMCDerivativeProperties.cpp.

3.22.3 Member Function Documentation

3.22.3.1 double QMCDerivativeProperties::getEffectiveTimeStep ()

Gets the effective time step for the calculation.

Returns:

effective time step for the calculation.

Definition at line 23 of file QMCDerivativeProperties.cpp.

References QMCproperties::distanceMovedAccepted, QMCproperties::distanceMovedTrial, and QMCproperty::getAverage().

3.22.3.2 double QMCDerivativeProperties::getEffectiveTimeStep-StandardDeviation ()

Gets the standard deviation of the calculated effective time step for the calculation.

Returns:
standard deviation of the effective time step for the calculation.

Definition at line 51 of file QMCDerivativeProperties.cpp.

References getEffectiveTimeStepVariance().

3.22.3.3 double QMCDerivativeProperties::getEffectiveTimeStep-Variance ()

Gets the variance of the calculated effective time step for the calculation.

Returns:
variance of the effective time step for the calculation.

Definition at line 31 of file QMCDerivativeProperties.cpp.

References QMCproperties::distanceMovedAccepted, QMCproperties::distance-MovedTrial, QMCproperty::getAverage(), and QMCproperty::getVariance().

Referenced by getEffectiveTimeStepStandardDeviation().

3.22.3.4 double QMCDerivativeProperties::getVirialRatio ()

Gets the virial ratio for the calculation.

The virial ratio is $-\langle V\rangle / \langle T\rangle$ where $\langle V\rangle$ is the expectation value of the potential energy and $\langle T\rangle$ is the expectation value of the kinetic energy.

Returns:
virial ratio.

Definition at line 56 of file QMCDerivativeProperties.cpp.

References QMCproperty::getAverage(), QMCproperties::kineticEnergy, and QMCproperties::potentialEnergy.

3.22.3.5 double QMCDerivativeProperties::getVirialRatio-StandardDeviation ()

Gets the standard deviation of the calculated virial ratio for the calculation.

Returns:
standard deviation of the virial ratio.

Definition at line 81 of file QMCDerivativeProperties.cpp.

References getVirialRatioVariance().

3.22.3.6 double QMCDerivativeProperties::getVirialRatioVariance ()

Gets the variance of the calculated virial ratio for the calculation.

Returns:
variance of the virial ratio.

Definition at line 64 of file QMCDerivativeProperties.cpp.

References QMCproperty::getAverage(), QMCproperty::getVariance(), QMCproperties::kineticEnergy, and QMCproperties::potentialEnergy.

Referenced by getVirialRatioStandardDeviation().

3.23 QMCFunctions Class Reference

This class calculates the value of the wavefunction, it's first two derivatives, and any other properties which are calculated from the wavefunction (local energy, etc.).

Public Methods

- **QMCFunctions** ()

  Creates a new instance of the class.

- **QMCFunctions** (QMCInput *input)

  Creates a new instance of the class and initializes it with the data controling the QMC calculation.

- **QMCFunctions** (const QMCFunctions &rhs)

  Creates a new instance of the class that is identical to another instance of QMCFunctions.

- void **initialize** (QMCInput *input)

  Initializes the object with the data controling the QMC calculation.

- void **evaluate** (**Array2D**< double > &X)

Evaluates all of the calculated properties at X.

- double **getPsi** ()

  Gets the value of the wavefunction at the last evaluated electronic configuration.

- double **getLocalEnergy** ()

  Gets the local energy at the last evaluated electronic configuration.

- double **getKineticEnergy** ()

  Gets the kinetic energy at the last evaluated electronic configuration.

- double **getPotentialEnergy** ()

  Gets the potential energy at the last evaluated electronic configuration.

- **Array2D**< double > * **getGradPsiRatio** ()

  Gets the ratio of the wavefunction gradient to the wavefunction value at the last evaluated electronic configuration.

- **Array2D**< double > * **getModifiedGradPsiRatio** ()

  Gets a modified version of the ratio of the wavefunction gradient to the wavefunction value at the last evaluated electronic configuration.

- bool **isSingular** ()

  Returns true if the last evaluated electronic configuration gives a singular Slater matrix and false otherwise.

- void **operator=** (const QMCFunctions &rhs)

  Sets two QMCFunctions objects equal.

- void **writeCorrelatedSamplingConfiguration** (ostream &strm)

  Writes the state of this object to a stream for use in correlated sampling calculations.

3.23.1 Detailed Description

This class calculates the value of the wavefunction, it's first two derivatives, and any other properties which are calculated from the wavefunction (local energy, etc.).

The wavefunction is assumed to be of the form

$$\Psi_{QMC} = D_{\uparrow} D_{\downarrow} J$$

where

$$J = exp(\sum u_{i,j}(r_{i,j}))$$

is a Jastrow type correlation function, and $D_\uparrow$ and $D_\downarrow$ are Slater determinants for the up and down electrons respectively.

Definition at line 45 of file QMCFunctions.h.

3.23.2 Constructor & Destructor Documentation

3.23.2.1 QMCFunctions::QMCFunctions (QMCInput * *input*)

Creates a new instance of the class and initializes it with the data controling the QMC calculation.

Parameters:

input input data for the calculation

Definition at line 19 of file QMCFunctions.cpp.

References initialize().

3.23.2.2 QMCFunctions::QMCFunctions (const QMCFunctions & *rhs*)

Creates a new instance of the class that is identical to another instance of QMCFunctions.

Parameters:

rhs object to make a copy of

Definition at line 24 of file QMCFunctions.cpp.

3.23.3 Member Function Documentation

3.23.3.1 void QMCFunctions::evaluate (Array2D< double > & *X*)

Evaluates all of the calculated properties at X.

Parameters:

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 60 of file QMCFunctions.cpp.

References QMCPotential_Energy::evaluate(), QMCJastrow::evaluate(), and QMCSlater::evaluate().

3.23.3.2 Array2D< double > * QMCFunctions::getGradPsiRatio ()

Gets the ratio of the wavefunction gradient to the wavefunction value at the last evaluated electronic configuration.

This is also known as the quantum force.

Returns:
wavefunction gradient ratio (quantum force)

Definition at line 290 of file QMCFunctions.cpp.

3.23.3.3 double QMCFunctions::getKineticEnergy ()

Gets the kinetic energy at the last evaluated electronic configuration.

Returns:
kinetic energy.

Definition at line 280 of file QMCFunctions.cpp.

3.23.3.4 double QMCFunctions::getLocalEnergy ()

Gets the local energy at the last evaluated electronic configuration.

Returns:
local energy

Definition at line 275 of file QMCFunctions.cpp.

Referenced by QMCwalker::toXML().

3.23.3.5 Array2D< double > * QMCFunctions::getModifiedGradPsiRatio ()

Gets a modified version of the ratio of the wavefunction gradient to the wavefunction value at the last evaluated electronic configuration.

The modifications typically help deal with singularities near nodes, and the particular type of modification can be selected. This is also known as the modified quantum force.

Returns:
modified wavefunction gradient ratio (modified quantum force)

Definition at line 295 of file QMCFunctions.cpp.

3.23.3.6 double QMCFunctions::getPotentialEnergy ()

Gets the potential energy at the last evaluated electronic configuration.

Returns:
potential energy.

Definition at line 285 of file QMCFunctions.cpp.

References QMCPotential_Energy::getEnergy().

3.23.3.7 double QMCFunctions::getPsi ()

Gets the value of the wavefunction at the last evaluated electronic configuration.

The returned value is not normalized to one.

Returns:
wavefunction value

Definition at line 270 of file QMCFunctions.cpp.

3.23.3.8 void QMCFunctions::initialize (QMCInput * *input*)

Initializes the object with the data controling the QMC calculation.

Parameters:
input input data for the calculation

Definition at line 46 of file QMCFunctions.cpp.

References Array2D< double >::allocate(), QMCJastrow::initialize(), QMCPotential_Energy::initialize(), and QMCSlater::initialize().

Referenced by QMCwalker::initialize(), and QMCFunctions().

3.23.3.9 bool QMCFunctions::isSingular ()

Returns true if the last evaluated electronic configuration gives a singular Slater matrix and false otherwise.

Returns:
true if the Slater matrix is singular and false otherwise

Definition at line 335 of file QMCFunctions.cpp.

References QMCSlater::isSingular().

Referenced by QMCwalker::isSingular().

3.23.3.10 void QMCFunctions::operator= (const QMCFunctions & *rhs*)

Sets two QMCFunctions objects equal.

Parameters:
rhs object to set this object equal to

Definition at line 29 of file QMCFunctions.cpp.

References Alpha, Beta, E_Local, Grad_PsiRatio, Input, Jastrow, Laplacian_-PsiRatio, Modified_Grad_PsiRatio, PE, and Psi.

3.23.3.11 void QMCFunctions::writeCorrelatedSamplingConfiguration (ostream & *strm*)

Writes the state of this object to a stream for use in correlated sampling calculations.

Parameters:
strm output stream

Definition at line 300 of file QMCFunctions.cpp.

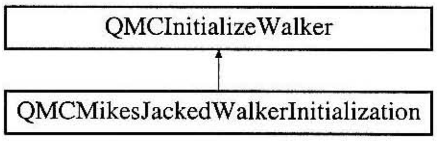
References QMCPotential_Energy::getEnergy(), QMCSlater::getGradPsiRatio(), QMCJastrow::getJastrow(), and QMCSlater::getLaplacianPsiRatio().

Referenced by QMCwalker::writeCorrelatedSamplingConfiguration().

3.24 QMCInitializeWalker Class Reference

Interface to algorithms which generate new walkers for a QMC calculation.

Inheritance diagram for QMCInitializeWalker::



Public Methods

- virtual ~**QMCInitializeWalker** ()
  Virtual destructor.

- virtual **Array2D**< double > **initializeWalkerPosition** ()=0

  Generates a new walker.

3.24.1 Detailed Description

Interface to algorithms which generate new walkers for a QMC calculation.

A good algorithm will generate walkers which require little time for the Metropolis algorithm to be equilibrated.

Definition at line 25 of file QMCInitializeWalker.h.

3.24.2 Member Function Documentation

3.24.2.1 virtual Array2D<double> QMCInitializeWalker::initializeWalkerPosition () `[pure virtual]`

Generates a new walker.

Returns:

new walker configuration represented by a $N \times 3$ matrix

Implemented in **QMCMikesJackedWalkerInitialization** (p. 96).

Referenced by QMCwalker::initializeWalkerPosition().

3.25 QMCInitializeWalkerFactory Class Reference

Object factory which returns the correct QMCInitialize walker when a string keyword describing the correlation function is provided.

Static Public Methods

- **QMCInitializeWalker** * **initializeWalkerFactory** (QMCInput *input, string &type)

  Returns the correct **QMCInitializeWalker** (p. 72) *when a string keyword describing the initialization method is provided.*

3.25.1 Detailed Description

Object factory which returns the correct QMCInitialize walker when a string keyword describing the correlation function is provided.

Definition at line 28 of file QMCInitializeWalkerFactory.h.

3.25.2 Member Function Documentation

3.25.2.1 QMCInitializeWalker * QMCInitializeWalkerFactory::initializeWalkerFactory (QMCInput * *input*, string & *type*) [static]

Returns the correct **QMCInitializeWalker** (p. 72) when a string keyword describing the initialization method is provided.

Parameters:

input input input data for the calculation

type string describing which initialization algorithm to choose

Returns:

the selected **QMCInitializeWalker** (p. 72) method.

Definition at line 16 of file QMCInitializeWalkerFactory.cpp.

Referenced by QMCwalker::initializeWalkerPosition().

3.26 QMCJastrow Class Reference

This class calculates the value of the Jastrow function and it's first two derivatives.

Public Methods

- void **initialize** (QMCInput *input)

  Initializes the class with the data controling the calculation.

- void **evaluate** (**Array2D**< double > &X)

  Evaluates the Jastrow function and it's derivatives at X using the **QMCJastrowParameters** (p. 83) *stored in the QMCInput class.*

- void **evaluate** (**QMCJastrowParameters** &JP, **Array2D**< double > &X)

  Evaluates the Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

- double **getJastrow** ()

  Gets the value of the Jastrow function for the last evaluated electronic configuration and parameter set.

- double **getLnJastrow** ()

  Gets the value of the natural log of the Jastrow function for the last evaluated electronic configuration and parameter set.

- **Array2D**< double > * **getGradientLnJastrow** ()

  Gets the gradient of the natural log of the Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

- double **getLaplacianLnJastrow** ()

  Gets the laplacian of the natural log of the Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

3.26.1 Detailed Description

This class calculates the value of the Jastrow function and it's first two derivatives.

The wavefunction is assumed to be of the form

$$\Psi_{QMC} = \Psi_{Trial} J$$

where Ψ_{Trial} is a wavefunction calculated using a standard QM method and

$$J = exp(\sum u_{i,j}(r_{i,j}))$$

is a Jastrow type correlation function. $u_{ij}(r_{ij})$ are **QMCCorrelationFunction** (p. 57) describing the interactions of particles i and j.

Definition at line 46 of file QMCJastrow.h.

3.26.2 Member Function Documentation

3.26.2.1 void QMCJastrow::evaluate (QMCJastrowParameters & *JP*, Array2D< double > & *X*)

Evaluates the Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

Parameters:

JP Jastrow parameters to use during the evaluation

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 48 of file QMCJastrow.cpp.

References Array2D< double >::allocate(), Array2D< T >::dim1(), Array2D< T >::dim2(), QMCJastrowElectronElectron::evaluate(), QMCJastrowElectronNuclear::evaluate(), QMCJastrowElectronElectron::getGradientLnJastrow(), QMCJastrowElectronNuclear::getGradientLnJastrow(), QMCJastrowElectronElectron::getLaplacianLnJastrow(), QMCJastrowElectronNuclear::getLaplacianLnJastrow(), QMCJastrowElectronElectron::getLnJastrow(), and QMCJastrowElectronNuclear::getLnJastrow().

3.26.2.2 void QMCJastrow::evaluate (Array2D< double > & *X*)

Evaluates the Jastrow function and it's derivatives at X using the **QMCJastrowParameters** (p. 83) stored in the QMCInput class.

Parameters:

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 43 of file QMCJastrow.cpp.

Referenced by QMCFunctions::evaluate().

3.26.2.3 Array2D< double > * QMCJastrow::getGradientLnJastrow ()

Gets the gradient of the natural log of the Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$

Returns:

gradient natural log of the Jastrow function ($\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$)

Definition at line 33 of file QMCJastrow.cpp.

3.26.2.4 double QMCJastrow::getJastrow ()

Gets the value of the Jastrow function for the last evaluated electronic configuration and parameter set.

$J = exp(\sum u_{i,j}(r_{i,j}))$

Returns:

Jastrow function value ($J = exp(\sum u_{i,j}(r_{i,j}))$).

Definition at line 23 of file QMCJastrow.cpp.

Referenced by QMCFunctions::writeCorrelatedSamplingConfiguration().

3.26.2.5 double QMCJastrow::getLaplacianLnJastrow ()

Gets the laplacian of the natural log of the Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$

Returns:
gradient natural log of the Jastrow function ($\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$)

Definition at line 38 of file QMCJastrow.cpp.

3.26.2.6 double QMCJastrow::getLnJastrow ()

Gets the value of the natural log of the Jastrow function for the last evaluated electronic configuration and parameter set.

$\ln(J) = \sum u_{i,j}(r_{i,j})$

Returns:
natural log of the Jastrow function ($\ln(J) = \sum u_{i,j}(r_{i,j})$)

Definition at line 28 of file QMCJastrow.cpp.

3.26.2.7 void QMCJastrow::initialize (QMCInput * *input*)

Initializes the class with the data controling the calculation.

Parameters:
input input data for the calculation

Definition at line 15 of file QMCJastrow.cpp.

References QMCJastrowElectronElectron::initialize(), and QMCJastrow-ElectronNuclear::initialize().

Referenced by QMCReadAndEvaluateConfigs::initialize(), and QMCFunctions::initialize().

3.27 QMCJastrowElectronElectron Class Reference

This class calculates the value of the electron-electron part of the Jastrow function and it's first two derivatives.

Public Methods

- void **initialize** (QMCInput *input)

  Initializes the class with the data controling the calculation.

- void **evaluate** (**QMCJastrowParameters** &JP, **Array2D**< double > &X)

  Evaluates the electron-electron Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

- double **getLnJastrow** ()

  Gets the value of the natural log of the electron-electron Jastrow function for the last evaluated electronic configuration and parameter set.

- **Array2D**< double > * **getGradientLnJastrow** ()

  Gets the gradient of the natural log of the electron-electron Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

- double **getLaplacianLnJastrow** ()

  Gets the laplacian of the natural log of the electron-electron Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

3.27.1 Detailed Description

This class calculates the value of the electron-electron part of the Jastrow function and it's first two derivatives.

The wavefunction is assumed to be of the form

$$\Psi_{QMC} = \Psi_{Trial} J$$

where Ψ_{Trial} is a wavefunction calculated using a standard QM method and

$$J = exp(\sum u_{i,j}(r_{i,j}))$$

is a Jastrow type correlation function. $u_{ij}(r_{ij})$ are **QMCCorrelationFunction** (p. 57) describing the interactions of particles i and j. The sum can be broken up into electron-electron and electron-nuclear components.

Definition at line 41 of file QMCJastrowElectronElectron.h.

3.27.2 Member Function Documentation

3.27.2.1 void QMCJastrowElectronElectron::evaluate (QMCJastrowParameters & *JP*, Array2D< double > & *X*)

Evaluates the electron-electron Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

Parameters:

JP Jastrow parameters to use during the evaluation

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 60 of file QMCJastrowElectronElectron.cpp.

References Array2D< double >::allocate(), Array2D< T >::dim1(), QMCCorrelationFunction::evaluate(), QMCCorrelationFunction-Parameters::getCorrelationFunction(), QMCJastrowParameters::getElectron-DownElectronDownParameters(), QMCJastrowParameters::getElectron-UpElectronDownParameters(), QMCJastrowParameters::getElectronUp-ElectronUpParameters(), QMCCorrelationFunction::getFirstDerivative-Value(), QMCCorrelationFunction::getFunctionValue(), and QMCCorrelation-Function::getSecondDerivativeValue().

Referenced by QMCJastrow::evaluate().

3.27.2.2 Array2D< double > * QMCJastrowElectronElectron::get-GradientLnJastrow ()

Gets the gradient of the natural log of the electron-electron Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$

Returns:

gradient natural log of the electron-electron Jastrow function ($\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$)

Definition at line 50 of file QMCJastrowElectronElectron.cpp.

Referenced by QMCJastrow::evaluate().

3.27.2.3 double QMCJastrowElectronElectron::getLaplacianLn-Jastrow ()

Gets the laplacian of the natural log of the electron-electron Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$

Returns:
gradient natural log of the electron-electron Jastrow function ($\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$)

Definition at line 45 of file QMCJastrowElectronElectron.cpp.

Referenced by QMCJastrow::evaluate().

3.27.2.4 double QMCJastrowElectronElectron::getLnJastrow ()

Gets the value of the natural log of the electron-electron Jastrow function for the last evaluated electronic configuration and parameter set.

$\ln(J) = \sum u_{i,j}(r_{i,j})$

Returns:
natural log of the electron-electron Jastrow function ($\ln(J) = \sum u_{i,j}(r_{i,j})$)

Definition at line 55 of file QMCJastrowElectronElectron.cpp.

Referenced by QMCJastrow::evaluate().

3.27.2.5 void QMCJastrowElectronElectron::initialize (QMCInput * *input*)

Initializes the class with the data controling the calculation.

Parameters:
input input data for the calculation

Definition at line 15 of file QMCJastrowElectronElectron.cpp.

Referenced by QMCJastrow::initialize().

3.28 QMCJastrowElectronNuclear Class Reference

This class calculates the value of the electron-nuclear part of the Jastrow function and it's first two derivatives.

Public Methods

- void **initialize** (QMCInput *input)
  Initializes the class with the data controling the calculation.

- void **evaluate** (**QMCJastrowParameters** &JP, **Array2D**< double > &X)

  Evaluates the electron-nuclear Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

- double **getLnJastrow** ()

  Gets the value of the natural log of the electron-nuclear Jastrow function for the last evaluated electronic configuration and parameter set.

- **Array2D**< double > * **getGradientLnJastrow** ()

  Gets the gradient of the natural log of the electron-nuclear Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

- double **getLaplacianLnJastrow** ()

  Gets the laplacian of the natural log of the electron-nuclear Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

3.28.1 Detailed Description

This class calculates the value of the electron-nuclear part of the Jastrow function and it's first two derivatives.

The wavefunction is assumed to be of the form

$$\Psi_{QMC} = \Psi_{Trial} J$$

where Ψ_{Trial} is a wavefunction calculated using a standard QM method and

$$J = exp(\sum u_{i,j}(r_{i,j}))$$

is a Jastrow type correlation function. $u_{ij}(r_{ij})$ are **QMCCorrelationFunction** (p. 57) describing the interactions of particles i and j. The sum can be broken up into electron-electron and electron-nuclear components.

Definition at line 44 of file QMCJastrowElectronNuclear.h.

3.28.2 Member Function Documentation

3.28.2.1 void QMCJastrowElectronNuclear::evaluate (QMCJastrowParameters & *JP*, Array2D< double > & *X*)

Evaluates the electron-nuclear Jastrow function and it's derivatives at X using a given set of **QMCJastrowParameters** (p. 83).

Parameters:

JP Jastrow parameters to use during the evaluation

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 62 of file QMCJastrowElectronNuclear.cpp.

References Array2D< double >::allocate(), Array1D< T >::dim1(), Array2D< T >::dim1(), QMCCorrelationFunction::evaluate(), QMCJastrowParameters::getElectronDownNuclearParameters(), QMCJastrowParameters::getElectronUpNuclearParameters(), QMCCorrelationFunction::getFirstDerivativeValue(), QMCCorrelationFunction::getFunctionValue(), QMCJastrowParameters::getNucleiTypes(), and QMCCorrelationFunction::getSecondDerivativeValue().

Referenced by QMCJastrow::evaluate().

3.28.2.2 Array2D< double > * QMCJastrowElectronNuclear::getGradientLnJastrow ()

Gets the gradient of the natural log of the electron-nuclear Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$

Returns:

gradient natural log of the electron-nuclear Jastrow function ($\nabla \ln(J) = \nabla \sum u_{i,j}(r_{i,j})$)

Definition at line 52 of file QMCJastrowElectronNuclear.cpp.

Referenced by QMCJastrow::evaluate().

3.28.2.3 double QMCJastrowElectronNuclear::getLaplacianLnJastrow ()

Gets the laplacian of the natural log of the electron-nuclear Jastrow function with respect to the cartesian electronic coordinates for the last evaluated electronic configuration and parameter set.

$\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$

Returns:

gradient natural log of the electron-nuclear Jastrow function ($\nabla^2 \ln(J) = \nabla^2 \sum u_{i,j}(r_{i,j})$)

Definition at line 47 of file QMCJastrowElectronNuclear.cpp.

Referenced by QMCJastrow::evaluate().

3.28.2.4 double QMCJastrowElectronNuclear::getLnJastrow ()

Gets the value of the natural log of the electron-nuclear Jastrow function for the last evaluated electronic configuration and parameter set.

$\ln(J) = \sum u_{i,j}(r_{i,j})$

Returns:
natural log of the electron-nuclear Jastrow function ($\ln(J) = \sum u_{i,j}(r_{i,j})$)

Definition at line 57 of file QMCJastrowElectronNuclear.cpp.

Referenced by QMCJastrow::evaluate().

3.28.2.5 void QMCJastrowElectronNuclear::initialize (QMCInput * *input*)

Initializes the class with the data controling the calculation.

Parameters:
input input data for the calculation

Definition at line 15 of file QMCJastrowElectronNuclear.cpp.

Referenced by QMCJastrow::initialize().

3.29 QMCJastrowParameters Class Reference

This class contains all of the parameters and corelation functons from which the Jastrow function is composed.

Public Methods

- **QMCJastrowParameters** ()
  Creates an instance of the class.
- **QMCJastrowParameters** (const QMCJastrowParameters &rhs)
  Creates an instance of the class that is identical to another instance of the class.
- void **setParameterVector** (**Array1D**< double > &params)
  Sets the parameters describing the particle-particle interactions.
- **Array1D**< double > **getParameters** ()
  Gets the parameters describing the particle-particle interactions.

- **QMCCorrelationFunctionParameters** $*$ **getElectronUpElectronDownParameters** ()

  Gets the **QMCCorrelationFunctionParameters** (p. 59) *describing up-down electron interactions.*

- **QMCCorrelationFunctionParameters** $*$ **getElectronUpElectronUpParameters** ()

  Gets the **QMCCorrelationFunctionParameters** (p. 59) *describing up-up electron interactions.*

- **QMCCorrelationFunctionParameters** $*$ **getElectronDownElectronDownParameters** ()

  Gets the **QMCCorrelationFunctionParameters** (p. 59) *describing down-down electron interactions.*

- **Array1D**< **QMCCorrelationFunctionParameters** > $*$ **getElectronUpNuclearParameters** ()

  Gets an array of **QMCCorrelationFunctionParameters** (p. 59) *describing up electron-nuclear interactions.*

- **Array1D**< **QMCCorrelationFunctionParameters** > $*$ **getElectronDownNuclearParameters** ()

  Gets an array of **QMCCorrelationFunctionParameters** (p. 59) *describing down electron-nuclear interactions.*

- **Array1D**< string > $*$ **getNucleiTypes** ()

  Gets an array which is a list of all the different types of nuclei in the molecule being calculated.

- void **operator=** (const QMCJastrowParameters &rhs)

  Sets two QMCJastrowParameters objects equal.

- void **read** (**Array1D**< string > &nucleitypes, bool linkparams, int nelup, int neldn, string runfile)

  Loads the state of the object from a file.

Friends

- ostream & **operator<<** (ostream &strm, QMCJastrowParameters &rhs)

  Writes the state of the object to an output stream.

3.29.1 Detailed Description

This class contains all of the parameters and corelation functons from which the Jastrow function is composed.

The wavefunction is assumed to be of the form

$$\Psi_{QMC} = \Psi_{Trial} J$$

where Ψ_{Trial} is a wavefunction calculated using a standard QM method and

$$J = exp(\sum u_{i,j}(r_{i,j}))$$

is a Jastrow type correlation function. $u_{ij}(r_{ij})$ are **QMCCorrelationFunction** (p. 57) describing the interactions of particles i and j. The correlation functions are parameterized to allow optimization. This class contains the functions and their specific parameterizations. The interactions are parameterized in terms of "parameters" and "constants." "parameters" are modified during optimizations, and "constants" are not.

Definition at line 48 of file QMCJastrowParameters.h.

3.29.2 Constructor & Destructor Documentation

3.29.2.1 QMCJastrowParameters::QMCJastrowParameters (const QMCJastrowParameters & *rhs*)

Creates an instance of the class that is identical to another instance of the class.

Parameters:
rhs object to copy

Definition at line 614 of file QMCJastrowParameters.cpp.

3.29.3 Member Function Documentation

3.29.3.1 QMCCorrelationFunctionParameters * QMCJastrowParameters::getElectronDownElectronDownParameters ()

Gets the **QMCCorrelationFunctionParameters** (p. 59) describing down-down electron interactions.

Returns:
down-down electron interaction parameters

Definition at line 588 of file QMCJastrowParameters.cpp.

Referenced by QMCJastrowElectronElectron::evaluate().

3.29.3.2 Array1D< QMCCorrelationFunctionParameters > * QMCJastrowParameters::getElectronDownNuclearParameters ()

Gets an array of **QMCCorrelationFunctionParameters** (p. 59) describing down electron-nuclear interactions.

Returns:
down electron-nuclear interaction parameters

Definition at line 600 of file QMCJastrowParameters.cpp.

Referenced by QMCJastrowElectronNuclear::evaluate().

3.29.3.3 QMCCorrelationFunctionParameters * QMCJastrowParameters::getElectronUpElectronDownParameters ()

Gets the **QMCCorrelationFunctionParameters** (p. 59) describing up-down electron interactions.

Returns:
up-down electron interaction parameters

Definition at line 576 of file QMCJastrowParameters.cpp.

Referenced by QMCJastrowElectronElectron::evaluate().

3.29.3.4 QMCCorrelationFunctionParameters * QMCJastrowParameters::getElectronUpElectronUpParameters ()

Gets the **QMCCorrelationFunctionParameters** (p. 59) describing up-up electron interactions.

Returns:
up-up electron interaction parameters

Definition at line 582 of file QMCJastrowParameters.cpp.

Referenced by QMCJastrowElectronElectron::evaluate().

3.29.3.5 Array1D< QMCCorrelationFunctionParameters > * QMCJastrowParameters::getElectronUpNuclearParameters ()

Gets an array of **QMCCorrelationFunctionParameters** (p. 59) describing up electron-nuclear interactions.

Returns:
up electron-nuclear interaction parameters

Definition at line 594 of file QMCJastrowParameters.cpp.

Referenced by QMCJastrowElectronNuclear::evaluate().

3.29.3.6 Array1D< double > QMCJastrowParameters::getParameters ()

Gets the parameters describing the particle-particle interactions.

Returns:

parameters describing particle-particle interactions.

Definition at line 206 of file QMCJastrowParameters.cpp.

References Array1D< T >::allocate(), Array1D< T >::dim1(), Array1D< QMCCorrelationFunctionParameters >::dim1(), QMCCorrelationFunctionParameters::getParameters(), and QMCCorrelationFunctionParameters::getTotalNumberOfParameters().

3.29.3.7 void QMCJastrowParameters::operator= (const QMCJastrowParameters & *rhs*)

Sets two QMCJastrowParameters objects equal.

Parameters:

rhs object to set this object eqal to

Definition at line 15 of file QMCJastrowParameters.cpp.

References EdnEdn, EdnNuclear, EquivalentElectronUpDownParams, EupEdn, EupEup, EupNuclear, NucleiTypes, NumberOfElectronsDown, NumberOfElectronsUp, and NumberOfParameters.

3.29.3.8 void QMCJastrowParameters::read (Array1D< string > & *nucleitypes*, bool *linkparams*, int *nelup*, int *neldn*, string *runfile*)

Loads the state of the object from a file.

Parameters:

nucleitypes list of the different kinds of nuclei

linkparams true if nuclear-electron interactions are strictly the same and false otherwise

nelup number of up spin electrons

neldn numer of down spin electrons

runfile name of the file to be loaded

Definition at line 385 of file QMCJastrowParameters.cpp.

References Array1D< QMCCorrelationFunctionParameters >::allocate(), Array1D< QMCCorrelationFunctionParameters >::dim1(), Array1D< string >::dim1(), QMCCorrelationFunctionParameters::getParticle1Type(), QMCCorrelationFunctionParameters::getParticle2Type(), QMCCorrelationFunctionParameters::read(), QMCCorrelationFunctionParameters::setParticle1Type(), and QMCCorrelationFunctionParameters::setParticle2Type().

3.29.3.9 void QMCJastrowParameters::setParameterVector (Array1D< double > & *params*)

Sets the parameters describing the particle-particle interactions.

Parameters:

params new set of parameters

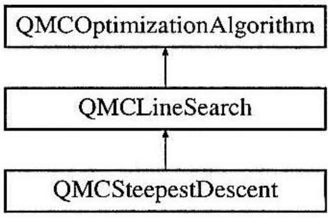
Definition at line 29 of file QMCJastrowParameters.cpp.

References Array1D< T >::allocate(), Array1D< QMCCorrelationFunctionParameters >::dim1(), Array1D< T >::dim1(), QMCCorrelationFunctionParameters::getTotalNumberOfParameters(), QMCCorrelationFunctionParameters::setParameters(), QMCCorrelationFunctionParameters::setParticle1Type(), and QMCCorrelationFunctionParameters::setParticle2Type().

3.30 QMCLineSearch Class Reference

Abstract implementation of a line search numerical optimization algorithm.

Inheritance diagram for QMCLineSearch::



Public Methods

- **QMCLineSearch** (**QMCObjectiveFunction** *function, **QMCLine-**

SearchStepLengthSelectionAlgorithm *stepAlg, int maxSteps, double tol)

Constructs and initializes an instance of this class.

- virtual **~QMCLineSearch** ()

  Virtual destructor.

- **Array1D**< double > **optimize** (**Array1D**< double > &initialGuess)

  Optimize the function starting from the provided initial guess parameters.

Protected Methods

- **QMCObjectiveFunction** * **getObjectiveFunction** ()

  Gets the objective function for the calculation.

3.30.1 Detailed Description

Abstract implementation of a line search numerical optimization algorithm.

As is standard in the field, the optimization is a minimization.

Definition at line 27 of file QMCLineSearch.h.

3.30.2 Constructor & Destructor Documentation

3.30.2.1 QMCLineSearch::QMCLineSearch (QMCObjectiveFunction * *function*, QMCLineSearchStepLengthSelectionAlgorithm * *stepAlg*, int *maxSteps*, double *tol*)

Constructs and initializes an instance of this class.

Parameters:

function objective function to optimize.

stepAlg algorithm to use claculate the step length.

maxSteps maximum number of steps to be performed during the line search.

tol tolerance to converge the solution to. Calculation is converged when $\left|1 - \frac{f(x_{i+1})}{f(x_i)}\right| < tol$.

Definition at line 15 of file QMCLineSearch.cpp.

3.30.3 Member Function Documentation

3.30.3.1 Array1D< double > QMCLineSearch::optimize (Array1D< double > & *initialGuess*) [virtual]

Optimize the function starting from the provided initial guess parameters.

Parameters:
initialGuess initial guess parameters for the optimization.

Returns:
optimized parameters.

Implements **QMCOptimizationAlgorithm** (p. 107).

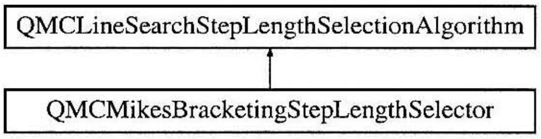
Definition at line 31 of file QMCLineSearch.cpp.

References Array1D< T >::dim1(), QMCObjectiveFunction::evaluate(), and QMCObjectiveFunctionResult::getScore().

3.31 QMCLineSearchStepLengthSelectionAlgorithm Class Reference

Interface to algorithms which determine the proper step length to use during a line search optimization (**QMCLineSearch** (p. 88)).

Inheritance diagram for QMCLineSearchStepLengthSelectionAlgorithm::


QMCLineSearchStepLengthSelectionAlgorithm
QMCMikesBracketingStepLengthSelector

Public Methods

- virtual ~**QMCLineSearchStepLengthSelectionAlgorithm** ()

  Virtual destructor.

- virtual double **stepLength** (**QMCObjectiveFunction** *function, **Array1D**< double > &position, **Array1D**< double > &searchDirection)=0

  Calculates the step length to use when performing a line search optimization.

3.31.1 Detailed Description

Interface to algorithms which determine the proper step length to use during a line search optimization (**QMCLineSearch** (p. 88)).

Definition at line 23 of file QMCLineSearchStepLengthSelectionAlgorithm.h.

3.31.2 Member Function Documentation

3.31.2.1 virtual double QMCLineSearchStepLengthSelectionAlgorithm::stepLength (QMCObjectiveFunction * *function*, Array1D< double > & *position*, Array1D< double > & *searchDirection*) `[pure virtual]`

Calculates the step length to use when performing a line search optimization.

Parameters:
- ***function*** objective function being optimized.
- ***position*** current location of the optimization.
- ***searchDirection*** direction to optimize along.

Implemented in **QMCMikesBracketingStepLengthSelector** (p. 95).

3.32 QMCLineSearchStepLengthSelectionFactory Class Reference

Object factory which returns the correct **QMCLineSearchStepLengthSelectionAlgorithm** (p. 90) when a string keyword describing the correlation function is provided.

Static Public Methods

- **QMCLineSearchStepLengthSelectionAlgorithm** * **factory** (string &Type)

  Returns the correct **QMCLineSearchStepLengthSelectionAlgorithm** (p. 90) *when a string keyword describing the correlation function is provided.*

3.32.1 Detailed Description

Object factory which returns the correct **QMCLineSearchStepLengthSelectionAlgorithm** (p. 90) when a string keyword describing the correlation function is provided.

Definition at line 29 of file QMCLineSearchStepLengthSelectionFactory.h.

3.33 QMCManager Class Reference

Controls the major sections of a QMC calculation.

Public Methods

- **QMCManager** ()

  Creates an uninitialized instance of this class.

- **~QMCManager** ()

  Destroys this object, cleans up the memory, and closes all open streams.

- void **initialize** (int argc, char **argv)

  Initializes this object and loads the input data for the calculation.

- void **finalize** ()

  Prepares the calculation to terminate.

- void **run** ()

  Performs a QMC calculation.

- void **optimize** ()

  Optimizes the parameters in a variational QMC (VMC) calculation using the correlated sampling method.

- void **zeroOut** ()

  Zeroes out all of the statistical data calculated by this object.

- void **writeRestart** ()

  Writes the restart file for the calculation.

- void **writeTimingData** (ostream &strm)

  Writes the timing data to a stream.

- QMCInput * **getInputData** ()

  Gets the input data for the calculation.

- ostream * **getResultsOutputStream** ()

  Gets the stream for outputting results from a calculation.

Friends

- ostream & **operator**<< (ostream &strm, QMCManager &rhs)

  Writes the current QMC results calculated by this object to an output stream in a human readable format.

3.33.1 Detailed Description

Controls the major sections of a QMC calculation.

This allows a QMC calculation to be run and parameters to be optimized.

Definition at line 41 of file QMCManager.h.

3.33.2 Member Function Documentation

3.33.2.1 QMCInput * QMCManager::getInputData ()

Gets the input data for the calculation.

Returns:
input data for the calculation.

Definition at line 769 of file QMCManager.cpp.

3.33.2.2 ostream * QMCManager::getResultsOutputStream ()

Gets the stream for outputting results from a calculation.

Returns:
output stream for results.

Definition at line 775 of file QMCManager.cpp.

3.33.2.3 void QMCManager::initialize (int *argc*, char ** *argv*)

Initializes this object and loads the input data for the calculation.

Parameters:
argc number of command line arguments.
argv command line arguments.

Definition at line 25 of file QMCManager.cpp.

References QMCrun::initialize().

3.33.2.4 void QMCManager::run ()

Performs a QMC calculation.

The specifics of the calculation are prescribed in the input.

Definition at line 241 of file QMCManager.cpp.

References QMCStopwatches::getInitializationStopwatch(), QMCStopwatches::getPropagationStopwatch(), QMCrun::getProperties(), Stopwatch::start(), QMCrun::step(), Stopwatch::stop(), QMCrun::writeCorrelatedSamplingConfigurations(), QMCrun::writeEnergies(), QMCrun::zeroOut(), and QMCproperties::zeroOut().

3.33.2.5 void QMCManager::writeTimingData (ostream & *strm*)

Writes the timing data to a stream.

This is only valid after `finalize` is called and only on the root node.

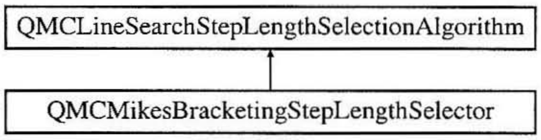
Parameters:
strm stream to write timing information to.

Definition at line 541 of file QMCManager.cpp.

3.34 QMCMikesBracketingStepLengthSelector Class Reference

Algorithm to determine the step length for a line search optimization developed by Michael Todd Feldmann.

Inheritance diagram for QMCMikesBracketingStepLengthSelector::


QMCLineSearchStepLengthSelectionAlgorithm
QMCMikesBracketingStepLengthSelector

Public Methods

- double **stepLength** (**QMCObjectiveFunction** *function, **Array1D**< double > &position, **Array1D**< double > &searchDirection)

  Calculates the step length to use when performing a line search optimization.

3.34.1 Detailed Description

Algorithm to determine the step length for a line search optimization developed by Michael Todd Feldmann.

This algorithm is purely huristic and does not insure the Wolfe conditions or other such properties. Again, much work could be done to do this part of a line search better.

Definition at line 29 of file QMCMikesBracketingStepLengthSelector.h.

3.34.2 Member Function Documentation

3.34.2.1 double QMCMikesBracketingStepLengthSelector::stepLength (QMCObjectiveFunction * *function*, Array1D< double > & *position*, Array1D< double > & *searchDirection*) [virtual]

Calculates the step length to use when performing a line search optimization.

Parameters:

function objective function being optimized.

position current location of the optimization.

searchDirection direction to optimize along.

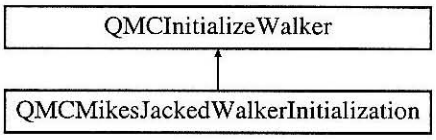
Implements **QMCLineSearchStepLengthSelectionAlgorithm** (p. 91).

Definition at line 15 of file QMCMikesBracketingStepLengthSelector.cpp.

3.35 QMCMikesJackedWalkerInitialization Class Reference

This is the algorithm made to initialize walkers.

Inheritance diagram for QMCMikesJackedWalkerInitialization::



Public Methods

- **QMCMikesJackedWalkerInitialization** (QMCInput *input)

Create an instance of the clas and initializes it.

- **Array2D**< double > **initializeWalkerPosition** ()

  Generates a new walker.

3.35.1 Detailed Description

This is the algorithm made to initialize walkers.

It is based on figuring out how many electrons should be on each atom followed by putting them in a gaussian around the atom. This is by far a method which needs a serious overhaul. This was a quick fix to initializing the walkers and the ideas are borrowed from CASINO. This method of initializing is probably very inefficient. This goes without mentioning how ugly the code is. This is a great place for further future work. A huge dent will likely be made on the "Initialization Catastrophe" problem here.

Definition at line 35 of file QMCMikesJackedWalkerInitialization.h.

3.35.2 Constructor & Destructor Documentation

3.35.2.1 QMCMikesJackedWalkerInitialization::QMCMikesJackedWalkerInitialization (QMCInput * *input*)

Create an instance of the clas and initializes it.

Parameters:
  input input data for the calculation

Definition at line 19 of file QMCMikesJackedWalkerInitialization.cpp.

3.35.3 Member Function Documentation

3.35.3.1 Array2D< double > QMCMikesJackedWalkerInitialization::initializeWalkerPosition () `[virtual]`

Generates a new walker.

Returns:
  new walker configuration represented by a $N \times 3$ matrix

Implements **QMCInitializeWalker** (p. 73).

Definition at line 24 of file QMCMikesJackedWalkerInitialization.cpp.

3.36 QMCMolecule Class Reference

Describes a particular molecular geometry.

Public Methods

- **QMCMolecule** ()

  Creates an instance of the class.

- void **initialize** (int nAtoms)

  Initializes the object.

- int **getNumberAtoms** ()

  Gets the number of atoms in the molecule.

- QMCMolecule **operator=** (const QMCMolecule &rhs)

  Sets two QMCMolecule objects equal.

- void **read** (string runfile)

  Loads the state of the object from a file.

Public Attributes

- **Array1D**< string > **Atom_Labels**

  Array containing the labels for the atoms.

- **Array2D**< double > **Atom_Positions**

  Array containing the 3-dimensional cartesian positions for the atoms.

- **Array1D**< int > **Z**

  Array containing the nuclear charges for the atoms.

- **Array1D**< string > **NucleiTypes**

  Array containing all of the different atom labels used in the molecule.

Friends

- istream & **operator>>** (istream &strm, QMCMolecule &rhs)

  Loads the state of the object from an input stream.

- ostream & **operator<<** (ostream &strm, QMCMolecule &rhs)
  Writes the state of the object to an output stream.

3.36.1 Detailed Description

Describes a particular molecular geometry.

The geometry is defined by 3-dimensional cartesian coordinates for each atom, with specified charges and types.

Definition at line 34 of file QMCMolecule.h.

3.36.2 Member Function Documentation

3.36.2.1 int QMCMolecule::getNumberAtoms ()

Gets the number of atoms in the molecule.

Returns:
number of atoms in the molecule.

Definition at line 25 of file QMCMolecule.cpp.

3.36.2.2 void QMCMolecule::initialize (int *nAtoms*)

Initializes the object.

Parameters:
nAtoms number of atoms in the molecule.

Definition at line 20 of file QMCMolecule.cpp.

3.36.2.3 QMCMolecule QMCMolecule::operator= (const QMC-Molecule & *rhs*)

Sets two QMCMolecule objects equal.

Parameters:
rhs object to set this object equal to.

Definition at line 31 of file QMCMolecule.cpp.

References Atom_Labels, Atom_Positions, Natoms, and Z.

3.36.2.4 void QMCMolecule::read (string *runfile*)

Loads the state of the object from a file.

Parameters:
runfile file to load the object state from.

Definition at line 60 of file QMCMolecule.cpp.

References Array1D< string >::allocate(), Atom_Labels, Array1D< string >::dim1(), and NucleiTypes.

3.36.3 Member Data Documentation

3.36.3.1 Array1D<string> QMCMolecule::Atom_Labels

Array containing the labels for the atoms.

The ith element is the label for the ith atom.

Definition at line 66 of file QMCMolecule.h.

Referenced by operator=(), and read().

3.36.3.2 Array2D<double> QMCMolecule::Atom_Positions

Array containing the 3-dimensional cartesian positions for the atoms.

The ith element is the position for the ith atom.

Definition at line 74 of file QMCMolecule.h.

Referenced by QMCBasisFunction::getGradPsi(), QMCBasisFunction::get-LaplacianPsi(), QMCBasisFunction::getPsi(), and operator=().

3.36.3.3 Array1D<int> QMCMolecule::Z

Array containing the nuclear charges for the atoms.

The ith element is the charge for the ith atom.

Definition at line 82 of file QMCMolecule.h.

Referenced by operator=().

3.37 QMCObjectiveFunction Class Reference

Objective function optimized during a variational QMC (VMC) calculation to find the optimal wavefunction parameters.

Public Methods

- void **initialize** (QMCInput *input)

  Initializes this object.

- **QMCObjectiveFunctionResult evaluate** (**Array1D**< double > &params)

  Evaluates and returns the result of the objective function evaluated with a single set of parameters.

- **Array1D**< **QMCObjectiveFunctionResult** > **evaluate** (**Array1D**< **Array1D**< double > > &params)

  Evaluates and returns the result of the objective function evaluated with multiple single sets of parameters.

- **Array1D**< double > **grad** (**Array1D**< double > &params)

  Evaluates and returns the gradient of the objective function for one set of parameters.

- **Array1D**< **Array1D**< double > > **grad** (**Array1D**< **Array1D**< double > > &params)

  Evaluates and returns the gradient of the objective function for multiple sets of parameters.

3.37.1 Detailed Description

Objective function optimized during a variational QMC (VMC) calculation to find the optimal wavefunction parameters.

As is standard in the field of numerical optimization, optimization means minimization. The particular form of the objective function is determined by parameters in the input file.

Definition at line 35 of file QMCObjectiveFunction.h.

3.37.2 Member Function Documentation

3.37.2.1 Array1D< QMCObjectiveFunctionResult > QMCObjectiveFunction::evaluate (Array1D< Array1D< double > > & *params*)

Evaluates and returns the result of the objective function evaluated with multiple single sets of parameters.

Parameters:

params sets of parameters to evaluate the objective function with.

Returns:

results of the objective function evaluations. The index of the input parameters corresponds to the index of the returned values.

Definition at line 21 of file QMCObjectiveFunction.cpp.

References QMCReadAndEvaluateConfigs::rootCalculateProperties().

3.37.2.2 QMCObjectiveFunctionResult QMCObjectiveFunction::evaluate (Array1D< double > & *params*)

Evaluates and returns the result of the objective function evaluated with a single set of parameters.

Parameters:

params set of parameters to evaluate the objective function with.

Returns:

result of the objective function evaluation.

Definition at line 46 of file QMCObjectiveFunction.cpp.

Referenced by QMCLineSearch::optimize().

3.37.2.3 Array1D< Array1D< double > > QMCObjectiveFunction::grad (Array1D< Array1D< double > > & *params*)

Evaluates and returns the gradient of the objective function for multiple sets of parameters.

Parameters:

params sets of parameters to evaluate the gradient with.

Returns:

gradients of the objective function. The index of the input parameters corresponds to the index of the returned values.

Definition at line 60 of file QMCObjectiveFunction.cpp.

3.37.2.4 Array1D< double > QMCObjectiveFunction::grad (Array1D< double > & *params*)

Evaluates and returns the gradient of the objective function for one set of parameters.

Parameters:
params sets of parameters to evaluate the gradient with.

Returns:
gradient of the objective function.

Definition at line 71 of file QMCObjectiveFunction.cpp.

3.37.2.5 void QMCObjectiveFunction::initialize (QMCInput * *input*)

Initializes this object.

This must be called before any other functions in this object are called.

Parameters:
input input data for the calculation

Definition at line 15 of file QMCObjectiveFunction.cpp.

References QMCReadAndEvaluateConfigs::initialize().

Referenced by QMCCorrelatedSamplingVMCOptimization::optimize().

3.38 QMCObjectiveFunctionResult Class Reference

Results from the evaluation of an objective function during a QMC calculation.

Public Methods

- **QMCObjectiveFunctionResult** ()
  Creates a new uninitialized instance of this class.
- **QMCObjectiveFunctionResult** (QMCInput *input, double energyAve, double energyVar, double logWeightAve, double logWeightVar)
  Creates and initializes a new instance of this class.
- **QMCObjectiveFunctionResult** (QMCObjectiveFunctionResult &rhs)

Creates a new instance of this class and makes it equivalent to another instance of this class.

- double **getLogWeightsAve** ()

  Gets the average value of the natural log of the statistical weights for the configurations used in this function evaluation.

- double **getLogWeightsVar** ()

  Gets the variance of the natural log of the statistical weights for the configurations used in this function evaluation.

- double **getEnergyAve** ()

  Gets the calculated average energy value.

- double **getEnergyVar** ()

  Gets the calculated energy variance.

- double **getScore** ()

  Gets a score for this function evaluation.

- double **getDerivativeScore** ()

  Gets a score for this function evaluation that is to be used in calculating the derivative in a numerical optimization.

- void **operator=** (QMCObjectiveFunctionResult &rhs)

  Sets two QMCObjectiveFunctionResult objects equal.

Friends

- ostream & **operator<<** (ostream &strm, const QMCObjectiveFunction-Result &rhs)

  Prints the contents of this object in a human readable format.

3.38.1 Detailed Description

Results from the evaluation of an objective function during a QMC calculation.

These results can then be used for numerical optimization or other functions.

Definition at line 28 of file QMCObjectiveFunctionResult.h.

3.38.2 Constructor & Destructor Documentation

3.38.2.1 QMCObjectiveFunctionResult::QMCObjectiveFunctionResult (QMCInput * *input*, double *energyAve*, double *energyVar*, double *logWeightAve*, double *logWeightVar*)

Creates and initializes a new instance of this class.

Parameters:

input data input to control the calculation.

energyAve calculated energy value

energyVar calculated energy variance

logWeightAve average value of the natural log of the statistical weights of the configurations.

logWeightVar variance in the above quantity.

Definition at line 19 of file QMCObjectiveFunctionResult.cpp.

3.38.2.2 QMCObjectiveFunctionResult::QMCObjectiveFunctionResult (QMCObjectiveFunctionResult & *rhs*)

Creates a new instance of this class and makes it equivalent to another instance of this class.

Parameters:

rhs object to set this equal to.

Definition at line 36 of file QMCObjectiveFunctionResult.cpp.

3.38.3 Member Function Documentation

3.38.3.1 double QMCObjectiveFunctionResult::getDerivativeScore ()

Gets a score for this function evaluation that is to be used in calculating the derivative in a numerical optimization.

The algorithm used for arriving at this score is determined by the input data. The convergence of a numerical optimization can be modified by changing the score functions.

Returns:

score for the derivative evaluation.

Definition at line 67 of file QMCObjectiveFunctionResult.cpp.

3.38.3.2 double QMCObjectiveFunctionResult::getEnergyAve ()

Gets the calculated average energy value.

Returns:
calculated average energy value.

Definition at line 52 of file QMCObjectiveFunctionResult.cpp.

3.38.3.3 double QMCObjectiveFunctionResult::getEnergyVar ()

Gets the calculated energy variance.

Returns:
calculated energy variance.

Definition at line 57 of file QMCObjectiveFunctionResult.cpp.

3.38.3.4 double QMCObjectiveFunctionResult::getLogWeightsAve ()

Gets the average value of the natural log of the statistical weights for the configurations used in this function evaluation.

Returns:
average value of the natural log of the statistical weights.

Definition at line 42 of file QMCObjectiveFunctionResult.cpp.

3.38.3.5 double QMCObjectiveFunctionResult::getLogWeightsVar ()

Gets the variance of the natural log of the statistical weights for the configurations used in this function evaluation.

Returns:
variance of the natural log of the statistical weights.

Definition at line 47 of file QMCObjectiveFunctionResult.cpp.

3.38.3.6 double QMCObjectiveFunctionResult::getScore ()

Gets a score for this function evaluation.

Better scores have lower values. The algorithm used for arriving at the scoris is determined by the input data. The convergence of a numerical optimization can be modified by changing the score functions.

Returns:
score for the function evaluation.

Definition at line 62 of file QMCObjectiveFunctionResult.cpp.

Referenced by QMCLineSearch::optimize().

3.38.3.7 void QMCObjectiveFunctionResult::operator= (QMCObjectiveFunctionResult & *rhs*)

Sets two QMCObjectiveFunctionResult objects equal.

Parameters:
rhs object to set this object equal to.

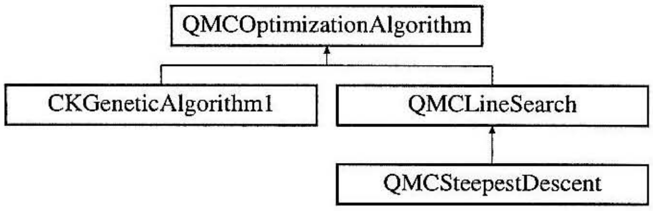
Definition at line 144 of file QMCObjectiveFunctionResult.cpp.

References energy_ave, energy_var, Input, log_weights_ave, log_weights_var, score, and score_for_derivative.

3.39 QMCOptimizationAlgorithm Class Reference

Interface for numerical optimization algorithms.

Inheritance diagram for QMCOptimizationAlgorithm::



Public Methods

- virtual ~**QMCOptimizationAlgorithm** ()

  Virtual destructor.

- virtual **Array1D**< double > **optimize** (**Array1D**< double > &initialGuess)=0

  Optimize the function starting from the provided initial guess parameters.

3.39.1 Detailed Description

Interface for numerical optimization algorithms.

Definition at line 22 of file QMCOptimizationAlgorithm.h.

3.39.2 Member Function Documentation

3.39.2.1 virtual Array1D<double> QMCOptimizationAlgorithm::optimize (Array1D< double > & *initialGuess*) `[pure virtual]`

Optimize the function starting from the provided initial guess parameters.

Parameters:
initialGuess initial guess parameters for the optimization.

Returns:
optimized parameters.

Implemented in **CKGeneticAlgorithm1** (p. 23), and **QMCLineSearch** (p. 90).

Referenced by QMCCorrelatedSamplingVMCOptimization::optimize().

3.40 QMCOptimizationFactory Class Reference

Object factory which returns the correct **QMCOptimizationAlgorithm** (p. 106) specified in the calculation input data.

Static Public Methods

- **QMCOptimizationAlgorithm** ∗ **optimizationAlgorithmFactory** (**QMCObjectiveFunction** &objFunc, QMCInput ∗input)

  Returns the correct **QMCOptimizationAlgorithm** (p. 106) *specified in the calculation input data.*

3.40.1 Detailed Description

Object factory which returns the correct **QMCOptimizationAlgorithm** (p. 106) specified in the calculation input data.

Optimization assumed to mean minimization, as is standard in the field.

Definition at line 30 of file QMCOptimizationFactory.h.

3.40.2 Member Function Documentation

3.40.2.1 QMCOptimizationAlgorithm * QMCOptimizationFactory::optimizationAlgorithmFactory (QMCObjectiveFunction & *objFunc*, QMCInput * *input*) [static]

Returns the correct **QMCOptimizationAlgorithm** (p. 106) specified in the calculation input data.

Parameters:

objFunc object function to optimize.

input input data to control the calculation.

Definition at line 16 of file QMCOptimizationFactory.cpp.

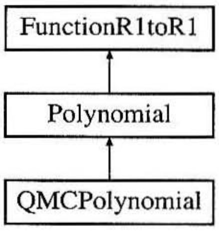
References QMCLineSearchStepLengthSelectionFactory::factory().

Referenced by QMCCorrelatedSamplingVMCOptimization::optimize().

3.41 QMCPolynomial Class Reference

An extension of **Polynomial** (p. 45) which adds QMC specific functionality.

Inheritance diagram for QMCPolynomial::



Public Methods

- **QMCPolynomial** ()

  Constructs an uninitialized instance of this class.

- **QMCPolynomial** (**Array1D**< double > &coeffs)

  Constructs and initializes an intance of this class.

- bool **hasNonNegativeZeroes** ()

  Determines if this polynomial has any non-negative real zeroes.

- void **initialize** (**Array1D**< double > &coeffs)

  Initializes this object.

- void **evaluate** (double x)

  Evaluates the function at x.

- double **getFunctionValue** ()

  Gets the function value at the last evaluated point.

- double **getFirstDerivativeValue** ()

  Gets the function's first deriviate at the last evaluated point.

- double **getSecondDerivativeValue** ()

  Gets the function's second deriviative at the last evaluated point.

- **Array1D**< **Complex** > **getRoots** ()

  Gets the roots of the polynomial.

Protected Methods

- int **getNumberCoefficients** ()

  Gets the number of coefficients in the polynomial.

- double **getCoefficient** (int i)

  Gets the ith coefficient of the polynomial.

3.41.1 Detailed Description

An extension of **Polynomial** (p. 45) which adds QMC specific functionality.

Definition at line 22 of file QMCPolynomial.h.

3.41.2 Constructor & Destructor Documentation

3.41.2.1 QMCPolynomial::QMCPolynomial (Array1D< double > & *coeffs*)

Constructs and initializes an intance of this class.

Parameters:

coeffs set of polynomial coefficients to use for the polynomial.

Definition at line 19 of file QMCPolynomial.cpp.

3.41.3 Member Function Documentation

3.41.3.1 void Polynomial::evaluate (double x) [virtual, inherited]

Evaluates the function at x.

Parameters:
x point to evaluate the function.

Implements **FunctionR1toR1** (p. 41).

Definition at line 61 of file Polynomial.cpp.

Referenced by PadeCorrelationFunction::evaluate(), FixedCuspPadeCorrelationFunction::evaluate(), Polynomial::getFirstDerivativeValue(), Polynomial::getFunctionValue(), and Polynomial::getSecondDerivativeValue().

3.41.3.2 double Polynomial::getCoefficient (int i) [protected, inherited]

Gets the ith coefficient of the polynomial.

Where the polynomial is defined such that

$$P(x) = \sum_{i=0}^{n} c_i x^i$$

where n is the order of the polynomial and c_i is the ith coefficient.

Parameters:
i index of the coefficient to return.

Returns:
ith coefficient of the polynomial.

Definition at line 122 of file Polynomial.cpp.

3.41.3.3 double Polynomial::getFirstDerivativeValue () [virtual, inherited]

Gets the function's first deriviate at the last evaluated point.

Returns:
function's deriviative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 97 of file Polynomial.cpp.

References Polynomial::evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPadeCorrelationFunction::evaluate().

3.41.3.4 double Polynomial::getFunctionValue () [virtual, inherited]

Gets the function value at the last evaluated point.

Returns:
function value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 87 of file Polynomial.cpp.

References Polynomial::evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPadeCorrelationFunction::evaluate().

3.41.3.5 int Polynomial::getNumberCoefficients () [protected, inherited]

Gets the number of coefficients in the polynomial.

This is one larger than the order of the polynomial.

Returns:
number of coefficients in the polynomial.

Definition at line 117 of file Polynomial.cpp.

References Array1D< double >::dim1().

3.41.3.6 Array1D< Complex > Polynomial::getRoots () [inherited]

Gets the roots of the polynomial.

Returns:
roots of the polynomial.

Exceptions:
Exception (p. 37) if problems were encounted during the root calculation.

Definition at line 127 of file Polynomial.cpp.

References Array1D< double >::dim1().

Referenced by hasNonNegativeZeroes().

3.41.3.7 double Polynomial::getSecondDerivativeValue () `[virtual, inherited]`

Gets the function's second deriviative at the last evaluated point.

Returns:
function's second derivative value.

Implements **FunctionR1toR1** (p. 41).

Definition at line 107 of file Polynomial.cpp.

References Polynomial::evaluate().

Referenced by PadeCorrelationFunction::evaluate(), and FixedCuspPade-CorrelationFunction::evaluate().

3.41.3.8 bool QMCPolynomial::hasNonNegativeZeroes ()

Determines if this polynomial has any non-negative real zeroes.

Returns:
`true` if the polynomial has a non-negative real zeros and `false` otherwise.

Exceptions:
Exception (p. 37) if problems were encounted during the calculation.

Definition at line 23 of file QMCPolynomial.cpp.

References Array1D< T >::dim1(), and Polynomial::getRoots().

Referenced by PadeCorrelationFunction::isSingular(), and FixedCuspPade-CorrelationFunction::isSingular().

3.41.3.9 void Polynomial::initialize (Array1D< double > & *coeffs*) `[inherited]`

Initializes this object.

Parameters:
coeffs set of polynomial coefficients to use for the polynomial.

Definition at line 39 of file Polynomial.cpp.

References Array1D< double >::allocate(), and Array1D< double >::dim1().

Referenced by PadeCorrelationFunction::initializeParameters(), FixedCuspPadeCorrelationFunction::initializeParameters(), and Polynomial::Polynomial().

3.42 QMCPotential_Energy Class Reference

The potential energy of the system.

Public Methods

- **QMCPotential_Energy** ()

  Creates an instance of the class.

- void **initialize** (QMCInput *input)

  Initialize the object.

- void **evaluate** (**Array2D**< double > &X)

  Evaluates the potential energy for the given electronic configuration.

- double **getEnergy** ()

  Gets the potential energy of the last configuration evaluated.

- void **operator=** (const QMCPotential_Energy &rhs)

  Sets two QMCPotential_Energy objects equal.

3.42.1 Detailed Description

The potential energy of the system.

Definition at line 29 of file QMCPotential_Energy.h.

3.42.2 Member Function Documentation

3.42.2.1 void QMCPotential_Energy::evaluate (Array2D< double > & *X*)

Evaluates the potential energy for the given electronic configuration.

Parameters:

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 35 of file QMCPotential_Energy.cpp.

Referenced by QMCFunctions::evaluate().

3.42.2.2 void QMCPotential_Energy::initialize (QMCInput * *input*)

Initialize the object.

Parameters:
input data input to control the calculation

Definition at line 29 of file QMCPotential_Energy.cpp.

Referenced by QMCFunctions::initialize().

3.42.2.3 void QMCPotential_Energy::operator= (const QMCPotential_Energy & *rhs*)

Sets two QMCPotential_Energy objects equal.

Parameters:
rhs object to set this object equal to

Definition at line 19 of file QMCPotential_Energy.cpp.

References Energy_total, Input, P_ee, P_en, and P_nn.

3.43 QMCproperties Class Reference

All of the quantities and properties evaluated during a calculation.

Public Methods

- **QMCproperties** ()

  Creates a zeroed out instance of the class and generates the MPI types if they have not been done.

- void **zeroOut** ()

  Sets all of the data in the object to zero.

- QMCproperties **operator+** (QMCproperties &rhs)

  Returns the sum of two QMCproperties.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

- void **readXML** (istream &strm)

  Loads the state of this object from an XML stream.

Public Attributes

- **QMCproperty energy**

  Total energy of the system.

- **QMCproperty kineticEnergy**

  Kinetic energy of the system.

- **QMCproperty potentialEnergy**

  Potential energy of the system.

- **QMCproperty logWeights**

  Log of the weights on the walkers.

- **QMCproperty acceptanceProbability**

  Probability a trial move is accepted.

- **QMCproperty distanceMovedAccepted**

  Average distance an accepted move travels.

- **QMCproperty distanceMovedTrial**

  Average distance for a trial move.

Static Public Attributes

- MPI_Datatype **MPI_TYPE**

  The MPI data type for a QMCproperties.

- MPI_Op **MPI_REDUCE**

  The MPI operation for performing MPI_Reduce on QMCproperties.

Friends

- ostream & **operator<<** (ostream &strm, QMCproperties &rhs)
  Formats and prints the properties to a stream.

3.43.1 Detailed Description

All of the quantities and properties evaluated during a calculation.

Definition at line 32 of file QMCproperties.h.

3.43.2 Member Function Documentation

3.43.2.1 void QMCproperties::readXML (istream & *strm*)

Loads the state of this object from an XML stream.

Parameters:
strm XML stream

Definition at line 102 of file QMCproperties.cpp.

References acceptanceProbability, distanceMovedAccepted, distance-MovedTrial, energy, kineticEnergy, logWeights, potentialEnergy, and QMCproperty::readXML().

Referenced by QMCrun::readXML().

3.43.2.2 void QMCproperties::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:
strm XML stream

Definition at line 58 of file QMCproperties.cpp.

References acceptanceProbability, distanceMovedAccepted, distance-MovedTrial, energy, kineticEnergy, logWeights, potentialEnergy, and QMCproperty::toXML().

Referenced by QMCrun::toXML().

3.44 QMCproperty Class Reference

All of the statistical information used in calculating a quantity or property during a calculation.

Public Methods

- **QMCproperty** ()

  Creates a zeroed out instance of the class and generates the MPI types if they have not been done.

- void **zeroOut** ()

  Sets all of the data in the object to zero.

- void **newSample** (double s, double weight)

  Adds a new data sample to the object.

- long **getNumberSamples** ()

  Gets the number of data samples entered into the object.

- double **getAverage** ()

  Gets the average of the data entered into the object.

- double **getVariance** ()

  Gets the variance of the data entered into the object.

- double **getSeriallyCorrelatedVariance** ()

  Gets the serially correlated variance of the data entered into the object.

- double **getStandardDeviation** ()

  Gets the standard deviation of the data entered into the object.

- double **getSeriallyCorrelatedStandardDeviation** ()

  Gets the serially correlated standard deviation of the data entered into the object.

- QMCproperty **operator+** (QMCproperty &rhs)

  Returns the sum of two **QMCproperties** (p. 114).

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

- void **readXML** (istream &strm)

  Loads the state of this object from an XML stream.

Static Public Attributes

- MPI_Datatype **MPI_TYPE**

  The MPI data type for a QMCproperty.

- MPI_Op **MPI_REDUCE**

  The MPI operation for performing MPI_Reduce on **QMCproperties** (p. 114).

Friends

- ostream & **operator<<** (ostream &strm, QMCproperty &rhs)

  Formats and prints the property to a stream.

3.44.1 Detailed Description

All of the statistical information used in calculating a quantity or property during a calculation.

Definition at line 40 of file QMCproperty.h.

3.44.2 Member Function Documentation

3.44.2.1 void QMCproperty::newSample (double *s*, double *weight*)

Adds a new data sample to the object.

Parameters:

s new sample data

weight statistical weight of the sample

Definition at line 94 of file QMCproperty.cpp.

References QMCstatistic::newSample().

Referenced by QMCwalker::calculateObservables().

3.44.2.2 void QMCproperty::readXML (istream & *strm*)

Loads the state of this object from an XML stream.

Parameters:

strm XML stream

Definition at line 299 of file QMCproperty.cpp.

References QMCstatistic::readXML().

Referenced by QMCproperties::readXML().

3.44.2.3 void QMCproperty::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:
strm XML stream

Definition at line 250 of file QMCproperty.cpp.

References QMCstatistic::toXML().

Referenced by QMCproperties::toXML().

3.45 QMCReadAndEvaluateConfigs Class Reference

Calculates properties (**QMCproperties** (p. 114)) from walkers and related data saved to a file during a QMC calculation.

Public Methods

- **QMCReadAndEvaluateConfigs** ()

  Creates an instance of the class.

- **QMCReadAndEvaluateConfigs** (QMCInput *input)

  Creates an instance of the class and initializes it.

- void **initialize** (QMCInput *input)

  Initializes the object.

- void **rootCalculateProperties** (**Array1D**< **Array1D**< double > > &params, **Array1D**< **QMCproperties** > &properties)

  *Calculates properties (***QMCproperties** (p. 114)*) for different parameter sets from walkers and related data saved to a file during a QMC calculation.*

- void **workerCalculateProperties** ()

  *Calculates properties (***QMCproperties** (p. 114)*) for different parameter sets from walkers and related data saved to a file during a QMC calculation.*

3.45.1 Detailed Description

Calculates properties (**QMCproperties** (p. 114)) from walkers and related data saved to a file during a QMC calculation.

Definition at line 36 of file QMCReadAndEvaluateConfigs.h.

3.45.2 Constructor & Destructor Documentation

3.45.2.1 QMCReadAndEvaluateConfigs::QMCReadAndEvaluateConfigs (QMCInput * *input*)

Creates an instance of the class and initializes it.

Parameters:
input data input to control the calculation.

Definition at line 19 of file QMCReadAndEvaluateConfigs.cpp.

References initialize().

3.45.3 Member Function Documentation

3.45.3.1 void QMCReadAndEvaluateConfigs::initialize (QMCInput * *input*)

Initializes the object.

Parameters:
input data input to control the calculation.

Definition at line 24 of file QMCReadAndEvaluateConfigs.cpp.

References Array2D< double >::allocate(), and QMCJastrow::initialize().

Referenced by QMCObjectiveFunction::initialize(), and QMCReadAndEvaluateConfigs().

3.45.3.2 void QMCReadAndEvaluateConfigs::rootCalculateProperties (Array1D< Array1D< double > > & *params*, Array1D< QMCproperties > & *properties*)

Calculates properties (**QMCproperties** (p. 114)) for different parameter sets from walkers and related data saved to a file during a QMC calculation.

This function is called only by the root node. The non-root nodes should call **workerCalculateProperties**() (p. 121).

Parameters:

params array of parameters which parameterize the wavefunction.

properties properties calculated from params and the saved configurations.

Definition at line 87 of file QMCReadAndEvaluateConfigs.cpp.

References Array1D< T >::allocate().

Referenced by QMCObjectiveFunction::evaluate().

3.45.3.3 void QMCReadAndEvaluateConfigs::workerCalculateProperties ()

Calculates properties (**QMCproperties** (p. 114)) for different parameter sets from walkers and related data saved to a file during a QMC calculation.

This function is called only by the non-root nodes. The root node should call rootCalculateProperties(params, properties).

Definition at line 155 of file QMCReadAndEvaluateConfigs.cpp.

References Array1D< T >::allocate(), and Array1D< T >::dim1().

Referenced by QMCCorrelatedSamplingVMCOptimization::optimize().

3.46 QMCrun Class Reference

Collection of walkers (**QMCwalker** (p. 134)) with the functionality to do the basic operations from which a QMC algorithm is built.

Public Methods

- **QMCrun** ()

  Creates an uninitialized instance of this class.

- void **initialize** (QMCInput *input)

  Initializes this object.

- void **zeroOut** ()

  Sets all of the data in the object to zero.

- void **step** ()

  Propagate the QMC calculation one time step forward.

- **QMCproperties** * **getProperties** ()

Gets the statistics for the properties that have been calculated.

- double **getWeights** ()

  Gets the total statistical weights for all the current living walkers.

- int **getNumberOfWalkers** ()

  Gets the current number of walkers.

- void **randomlyInitializeWalkers** ()

  Generates all of the walkers by initializing the electronic configurations for the walkers using an algorithm from **QMCInitializeWalkerFactory** (p. 73).

- void **writeEnergies** (ostream &strm)

  Writes the energies of all the walkers to a stream.

- void **writeCorrelatedSamplingConfigurations** (ostream &strm)

  Writes the state of this group of walkers to a stream in a format that is suitable for correlated sampling calculations.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

- void **readXML** (istream &strm)

  Reads the state of this object from an XML stream.

3.46.1 Detailed Description

Collection of walkers (**QMCwalker** (p. 134)) with the functionality to do the basic operations from which a QMC algorithm is built.

Definition at line 30 of file QMCrun.h.

3.46.2 Member Function Documentation

3.46.2.1 int QMCrun::getNumberOfWalkers ()

Gets the current number of walkers.

Returns:
number of walkers.

Definition at line 355 of file QMCrun.cpp.

3.46.2.2 QMCproperties * QMCrun::getProperties ()

Gets the statistics for the properties that have been calculated.

Returns:
statistics for the properties that have been calculated.

Definition at line 350 of file QMCrun.cpp.

Referenced by QMCManager::run().

3.46.2.3 double QMCrun::getWeights ()

Gets the total statistical weights for all the current living walkers.

Returns:
total weights for current walkers.

Definition at line 303 of file QMCrun.cpp.

3.46.2.4 void QMCrun::initialize (QMCInput * *input*)

Initializes this object.

Parameters:
input input data for the calculation

Definition at line 69 of file QMCrun.cpp.

References QMCproperties::zeroOut().

Referenced by QMCManager::initialize().

3.46.2.5 void QMCrun::readXML (istream & *strm*)

Reads the state of this object from an XML stream.

Parameters:
strm XML stream

Definition at line 332 of file QMCrun.cpp.

References QMCwalker::initialize(), QMCwalker::readXML(), and QMCproperties::readXML().

3.46.2.6 void QMCrun::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:

strm XML stream

Definition at line 316 of file QMCrun.cpp.

References QMCproperties::toXML().

3.46.2.7 void QMCrun::writeCorrelatedSamplingConfigurations (ostream & *strm*)

Writes the state of this group of walkers to a stream in a format that is suitable for correlated sampling calculations.

This writes out more information than `toXML` so that parts of the wavefunction do not have to be reevaluated every time properties are calculated using correlated sampling.

Parameters:

strm stream to write correlated sampling information to.

Definition at line 149 of file QMCrun.cpp.

Referenced by QMCManager::run().

3.46.2.8 void QMCrun::writeEnergies (ostream & *strm*)

Writes the energies of all the walkers to a stream.

Parameters:

strm stream to write energies to.

Definition at line 139 of file QMCrun.cpp.

Referenced by QMCManager::run().

3.47 QMCSlater Class Reference

A Slater determinant describing like spin electrons from a 3N dimensional wavefunction.

Public Methods

- void **initialize** (QMCInput *input, int startEl, int stopEl)

Initializes the class and sets which region of the $3N$ dimensional electronic configuration corresponds to electrons in this Slater determinant.

- void **evaluate** (**Array2D**< double > &X)

  Evaluates the slater determinant and it's first two derivatives at X.

- double **getPsi** ()

  Gets the value of the Slater determinant for the last evaluated electronic configuration.

- **Array2D**< double > * **getGradPsiRatio** ()

  Gets the ratio of the Slater determinant gradient over the Slater determinant for the last evaluated electronic configuration.

- double **getLaplacianPsiRatio** ()

  Gets the ratio of the Slater determinant laplacian over the Slater determinant for the last evaluated electronic configuration.

- bool **isSingular** ()

  Returns true if the Slater determinant is singular and false otherwise.

- void **operator=** (const QMCSlater &rhs)

  Sets two QMCSlater objects equal.

3.47.1 Detailed Description

A Slater determinant describing like spin electrons from a 3N dimensional wave-function.

This class allows the function, it's gradient, and it's laplacian to be calculated.

Definition at line 33 of file QMCSlater.h.

3.47.2 Member Function Documentation

3.47.2.1 void QMCSlater::evaluate (Array2D< double > & *X*)

Evaluates the slater determinant and it's first two derivatives at X.

Parameters:

X $3N$ dimensional configuration of electrons represented by a $N \times 3$ matrix

Definition at line 66 of file QMCSlater.cpp.

References isSingular().

Referenced by QMCFunctions::evaluate().

3.47.2.2 Array2D< double > * QMCSlater::getGradPsiRatio ()

Gets the ratio of the Slater determinant gradient over the Slater determinant for the last evaluated electronic configuration.

This value does not depend on the normalization of the Slater determinant.

Definition at line 223 of file QMCSlater.cpp.

Referenced by QMCFunctions::writeCorrelatedSamplingConfiguration().

3.47.2.3 double QMCSlater::getLaplacianPsiRatio ()

Gets the ratio of the Slater determinant laplacian over the Slater determinant for the last evaluated electronic configuration.

This value does not depend on the normalization of the Slater determinant.

Definition at line 218 of file QMCSlater.cpp.

Referenced by QMCFunctions::writeCorrelatedSamplingConfiguration().

3.47.2.4 double QMCSlater::getPsi ()

Gets the value of the Slater determinant for the last evaluated electronic configuration.

The returned value is not normalized to one. Assuming the basis functions ued to make the determinant are normalized, this value can be normalized by dividing it by $\sqrt{M!}$, where M is the number of electrons in this determinant.

Definition at line 213 of file QMCSlater.cpp.

3.47.2.5 void QMCSlater::initialize (QMCInput * *input*, int *startEl*, int *stopEl*)

Initializes the class and sets which region of the $3N$ dimensional electronic configuration corresponds to electrons in this Slater determinant.

It is assumed that all electrons in a determinant are grouped together in the configuration.

Parameters:

input input data for the calculation

startEl first particle in this determinant.

stopEl last particle in this determinant.

Definition at line 38 of file QMCSlater.cpp.

Referenced by QMCFunctions::initialize().

3.47.2.6 void QMCSlater::operator= (const QMCSlater & *rhs*)

Sets two QMCSlater objects equal.

Parameters:
rhs object to set this object equal to

Definition at line 15 of file QMCSlater.cpp.

References BF, D, Array2D< double >::dim1(), Grad_PsiRatio, Input, Laplacian_PsiRatio, Psi, Singular, Start, Stop, and WF.

3.48 QMCstatistic Class Reference

Statistical information on a set of data.

Public Methods

- **QMCstatistic** ()

  Creates a zeroed out instance of the class and generates the MPI type if it has not been done.

- void **zeroOut** ()

  Sets all of the data in the object to zero.

- long **getNumberSamples** ()

  Gets the number of data samples entered into the object.

- double **getAverage** ()

  Gets the average of the data entered into the object.

- double **getVariance** ()

  Gets the variance of the data entered into the object.

- double **getStandardDeviation** ()

  Gets the standard deviation of the data entered into the object.

- void **newSample** (double s, double weight)

  Adds a new data sample to the object.

- QMCstatistic **operator+** (const QMCstatistic &rhs)

Returns the sum of two QMCstatistics.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

- void **readXML** (istream &strm)

  Loads the state of this object from an XML stream.

Static Public Attributes

- MPI_Datatype **MPI_TYPE**

  The MPI data type for a QMCstatistic.

- MPI_Op **MPI_REDUCE**

  The MPI operation for performing MPI_Reduce on QMCstatistics.

Friends

- ostream & **operator<<** (ostream &strm, QMCstatistic &rhs)

  Formats and prints the statistic to a stream.

3.48.1 Detailed Description

Statistical information on a set of data.

Definition at line 31 of file QMCstatistic.h.

3.48.2 Member Function Documentation

3.48.2.1 void QMCstatistic::newSample (double *s*, double *weight*)

Adds a new data sample to the object.

Parameters:

s new sample data

weight statistical weight of the sample

Definition at line 58 of file QMCstatistic.cpp.

Referenced by QMCproperty::newSample(), and QMCproperty::operator+().

3.48.2.2 void QMCstatistic::readXML (istream & *strm*)

Loads the state of this object from an XML stream.

Parameters:
strm XML stream

Definition at line 97 of file QMCstatistic.cpp.

Referenced by QMCproperty::readXML().

3.48.2.3 void QMCstatistic::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:
strm XML stream

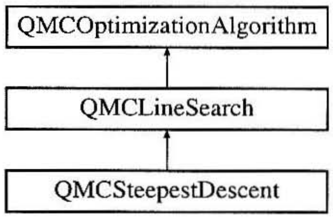
Definition at line 76 of file QMCstatistic.cpp.

Referenced by QMCproperty::toXML().

3.49 QMCSteepestDescent Class Reference

Steepest descent line search numerical optimization algorithm.

Inheritance diagram for QMCSteepestDescent::



Public Methods

- **QMCSteepestDescent** (**QMCObjectiveFunction** *function, **QMCLineSearchStepLengthSelectionAlgorithm** *stepAlg, int maxSteps, double tol)

    Constructs and initializes an instance of this class.

- **Array1D**< double > **optimize** (**Array1D**< double > &initialGuess)

    Optimize the function starting from the provided initial guess parameters.

Protected Methods

- **QMCObjectiveFunction ∗ getObjectiveFunction** ()

  Gets the objective function for the calculation.

3.49.1 Detailed Description

Steepest descent line search numerical optimization algorithm.

As is standard in the field, the optimization is a minimization.

Definition at line 23 of file QMCSteepestDescent.h.

3.49.2 Constructor & Destructor Documentation

3.49.2.1 QMCSteepestDescent::QMCSteepestDescent (QMCObjectiveFunction ∗ *function*, QMCLineSearchStepLengthSelectionAlgorithm ∗ *stepAlg*, int *maxSteps*, double *tol*)

Constructs and initializes an instance of this class.

Parameters:

function objective function to optimize.

stepAlg algorithm to use in determining the line search step length.

maxSteps maximum number of steps to be performed during the line search.

tol tolerance to converge the solution to. Calculation is converged when $\left|1 - \frac{f(x_{i+1})}{f(x_i)}\right| < tol$.

Definition at line 15 of file QMCSteepestDescent.cpp.

3.49.3 Member Function Documentation

3.49.3.1 Array1D< double > QMCLineSearch::optimize (Array1D< double > & *initialGuess*) `[virtual, inherited]`

Optimize the function starting from the provided initial guess parameters.

Parameters:

initialGuess initial guess parameters for the optimization.

Returns:

optimized parameters.

Implements **QMCOptimizationAlgorithm** (p. 107).

Definition at line 31 of file QMCLineSearch.cpp.

References Array1D< T >::dim1(), QMCObjectiveFunction::evaluate(), and QMCObjectiveFunctionResult::getScore().

3.50 QMCStopwatches Class Reference

A collection of **Stopwatch** (p. 144) objects used to record information relevant to the timing of a QMC calculation.

Public Methods

- **QMCStopwatches** ()

  Creates a new instance of this class with all timers stopped.

- void **stop** ()

  Stops all stopwatches in this object which are running.

- void **reset** ()

  Resets all stopwatches in this object and leaves the stopwatches stopped.

- **Stopwatch** * **getInitializationStopwatch** ()

  Gets the stopwatch which times the initialization of the calculation.

- **Stopwatch** * **getPropagationStopwatch** ()

  Gets the stopwatch which times the useful propagation of walkers.

- **Stopwatch** * **getSendCommandStopwatch** ()

  Gets the stopwatch which times the sending of commands between processors.

- **Stopwatch** * **getGatherPropertiesStopwatch** ()

  Gets the stopwatch which times the gathering of **QMCproperties** (p. 114) *from all processors.*

- **Stopwatch** * **getCommunicationSynchronizationStopwatch** ()

  Gets the stopwatch which times the synchronization of all the processors.

- **Stopwatch** * **getCommandPollingStopwatch** ()

  Gets the stopwatch which times how long is devoted to seeing if a processor has a command waiting for it.

- **Stopwatch** * **getOptimizationStopwatch** ()

Gets the stopwatch which times the VMC optimization.

- **Stopwatch * getTotalTimeStopwatch** ()

  Gets the stopwatch which records the total time of the calculation.

- QMCStopwatches **operator+** (QMCStopwatches &rhs)

  Returns a QMCStopwatches which is the sum of two QMCStopwatches objects.

Static Public Attributes

- MPI_Datatype **MPI_TYPE**

  The MPI data type for a QMCStopwatches.

- MPI_Op **MPI_REDUCE**

  The MPI operation for performing MPI_Reduce on QMCStopwatches objects.

Friends

- ostream & **operator<<** (ostream &strm, QMCStopwatches &rhs)

  Writes the timing results of this class to a human readable stream.

3.50.1 Detailed Description

A collection of **Stopwatch** (p. 144) objects used to record information relevant to the timing of a QMC calculation.

Definition at line 29 of file QMCStopwatches.h.

3.50.2 Member Function Documentation

3.50.2.1 Stopwatch * QMCStopwatches::getCommandPollingStopwatch ()

Gets the stopwatch which times how long is devoted to seeing if a processor has a command waiting for it.

Returns:

the stopwatch.

Definition at line 78 of file QMCStopwatches.cpp.

3.50.2.2 Stopwatch * QMCStopwatches::getCommunication-SynchronizationStopwatch ()

Gets the stopwatch which times the synchronization of all the processors.

Returns:
the stopwatch.

Definition at line 73 of file QMCStopwatches.cpp.

3.50.2.3 Stopwatch * QMCStopwatches::getGatherProperties-Stopwatch ()

Gets the stopwatch which times the gathering of **QMCproperties** (p. 114) from all processors.

Returns:
the stopwatch.

Definition at line 68 of file QMCStopwatches.cpp.

3.50.2.4 Stopwatch * QMCStopwatches::getOptimizationStopwatch ()

Gets the stopwatch which times the VMC optimization.

Returns:
the stopwatch.

Definition at line 83 of file QMCStopwatches.cpp.

Referenced by QMCManager::optimize().

3.50.2.5 Stopwatch * QMCStopwatches::getPropagationStopwatch ()

Gets the stopwatch which times the useful propagation of walkers.

The time required to initialize the walkers is not included.

Returns:
the stopwatch.

Definition at line 58 of file QMCStopwatches.cpp.

Referenced by QMCManager::run().

3.50.2.6 Stopwatch * QMCStopwatches::getSendCommandStopwatch ()

Gets the stopwatch which times the sending of commands between processors.

Returns:
the stopwatch.

Definition at line 63 of file QMCStopwatches.cpp.

3.50.2.7 Stopwatch * QMCStopwatches::getTotalTimeStopwatch ()

Gets the stopwatch which records the total time of the calculation.

Returns:
the stopwatch.

Definition at line 88 of file QMCStopwatches.cpp.

Referenced by QMCManager::QMCManager().

3.51 QMCwalker Class Reference

An instantaneous snapshot of all 3N electronic corrdinates for a system.

Public Methods

- **QMCwalker** ()

  Creates a new uninitialized instance of this class.

- **QMCwalker** (const QMCwalker &rhs)

  Creates a new instance of this class and makes it equivalent to another instance of this class.

- **~QMCwalker** ()

  Deallocates the memory allocated by this object.

- void **initialize** (QMCInput *input)

  Initializes and allocates memory for the walker.

- void **initializeWalkerPosition** ()

  Initializes the electronic configuration for this walker using an algorithm from **QMCInitializeWalkerFactory** (p. 73).

- void **propagateWalker** ()

  Proposes a trial walker move and accepts or rejects it.

- void **calculateObservables** (**QMCproperties** &props)

  Calculates the observables for this walker and adds them to the input **QM-Cproperties** (p. 114).

- void **operator=** (const QMCwalker &rhs)

  Sets two QMCwalker objects equal.

- double **getWeight** ()

  Gets the weight for this walker.

- void **setWeight** (double val)

  Sets the weight for this walker.

- bool **isSingular** ()

  Determines if the trial wavefunction is singular for this walker.

- void **toXML** (ostream &strm)

  Writes the state of this object to an XML stream.

- void **readXML** (istream &strm)

  Loads the state of this object from an XML stream.

- void **writeCorrelatedSamplingConfiguration** (ostream &strm)

  Writes the state of this walker to a stream in a format that is suitable for correlated sampling.

- double **getLocalEnergyEstimator** ()

  Gets the value of the local energy estimator for this walker.

3.51.1 Detailed Description

An instantaneous snapshot of all 3N electronic corrdinates for a system.

This is the same as the "walker" or "psip" discussed in QMC literature.

Definition at line 29 of file QMCwalker.h.

3.51.2 Constructor & Destructor Documentation

3.51.2.1 QMCwalker::QMCwalker (const QMCwalker & *rhs*)

Creates a new instance of this class and makes it equivalent to another instance of this class.

Parameters:

rhs object to set this equal to.

Definition at line 24 of file QMCwalker.cpp.

3.51.3 Member Function Documentation

3.51.3.1 void QMCwalker::calculateObservables (QMCproperties & *props*)

Calculates the observables for this walker and adds them to the input **QM-Cproperties** (p. 114).

Parameters:

props properties to which this walkers current observable values are added.

Definition at line 650 of file QMCwalker.cpp.

References QMCproperties::acceptanceProbability, QMCproperties::distance-MovedAccepted, QMCproperties::distanceMovedTrial, QMCproperties::energy, getWeight(), QMCproperties::kineticEnergy, QMCproperties::logWeights, QMCproperty::newSample(), and QMCproperties::potentialEnergy.

Referenced by propagateWalker().

3.51.3.2 double QMCwalker::getWeight ()

Gets the weight for this walker.

Returns:

weight for this walker.

Definition at line 587 of file QMCwalker.cpp.

Referenced by calculateObservables(), and toXML().

3.51.3.3 void QMCwalker::initialize (QMCInput * *input*)

Initializes and allocates memory for the walker.

The electronic configuration for the walker is not set. To do this `initialize-WalkerPosition` must be used to generate a new walker, or `read` must be used to read this walkers state from a stream.

Parameters:
input data input to control the calculation.

Definition at line 470 of file QMCwalker.cpp.

References Array2D< double >::allocate(), and QMCFunctions::initialize().

Referenced by QMCrun::randomlyInitializeWalkers(), and QMCrun::read-XML().

3.51.3.4 void QMCwalker::initializeWalkerPosition ()

Initializes the electronic configuration for this walker using an algorithm from **QMCInitializeWalkerFactory** (p. 73).

If a singular walker is generated, upto 100 configurations are generated until one is not singular.

Definition at line 557 of file QMCwalker.cpp.

References QMCInitializeWalkerFactory::initializeWalkerFactory(), QMCInitializeWalker::initializeWalkerPosition(), and isSingular().

Referenced by propagateWalker(), and QMCrun::randomlyInitializeWalkers().

3.51.3.5 bool QMCwalker::isSingular ()

Determines if the trial wavefunction is singular for this walker.

Returns:
`true` if the trial wavefunction is singular for this walker, and `false` otherwise.

Definition at line 678 of file QMCwalker.cpp.

References QMCFunctions::isSingular().

Referenced by initializeWalkerPosition(), and propagateWalker().

3.51.3.6 void QMCwalker::operator= (const QMCwalker & *rhs*)

Sets two QMCwalker objects equal.

Parameters:
rhs object to set this object equal to.

Definition at line 45 of file QMCwalker.cpp.

References AcceptanceProbability, age, distanceMovedAccepted, dR2, Input, kineticEnergy, localEnergy, move_accepted, potentialEnergy, QMF, R, and weight.

3.51.3.7 void QMCwalker::readXML (istream & *strm*)

Loads the state of this object from an XML stream.

The input stream must be formatted exactly like the output from `toXML` because it is not intelligent.

Parameters:
  strm XML stream

Definition at line 524 of file QMCwalker.cpp.

Referenced by QMCrun::readXML().

3.51.3.8 void QMCwalker::setWeight (double *val*)

Sets the weight for this walker.

Parameters:
  val value to set the weight equal to.

Definition at line 592 of file QMCwalker.cpp.

3.51.3.9 void QMCwalker::toXML (ostream & *strm*)

Writes the state of this object to an XML stream.

Parameters:
  strm XML stream

Definition at line 503 of file QMCwalker.cpp.

References QMCFunctions::getLocalEnergy(), and getWeight().

3.51.3.10 void QMCwalker::writeCorrelatedSamplingConfiguration (ostream & *strm*)

Writes the state of this walker to a stream in a format that is suitable for correlated sampling.

This writes out more information than `toXML` so that parts of the wavefunction do not have to be reevaluated every time properties are calculated using correlated sampling.

Parameters:

strm stream to write correlated sampling information to.

Definition at line 483 of file QMCwalker.cpp.

References Array2D< double >::dim1(), and QMCFunctions::writeCorrelated-SamplingConfiguration().

3.52 QMCWavefunction Class Reference

The coefficients and parameters describing the trial wavefunction for the system.

Public Methods

- **QMCWavefunction** ()

  Creates an instance of the class.

- int **getNumberOrbitals** ()

  Gets the number of orbitals.

- int **getNumberBasisFunctions** ()

  Gets the number of basis functions.

- int **getNumberAlphaElectrons** ()

  Gets the number of α spin electrons.

- int **getNumberBetaElectrons** ()

  Gets the number of β spin electrons.

- int **getNumberElectrons** ()

  Gets the total number of electrons.

- QMCWavefunction **operator=** (const QMCWavefunction &rhs)

  Sets two QMCWavefunction objects equal.

- void **read** (int numberOrbitals, int numberBasisFunctions, string run-file)

  Loads the state of the object from a file.

Public Attributes

- **Array2D**< double > **Coeffs**

  Array containing the coefficients used to construct the orbitals.

- **Array1D**< int > **AlphaOccupation**

  Array which indicates how many α spin electron are in each orbital for the wavefunction.

- **Array1D**< int > **BetaOccupation**

  Array which indicates how many β spin electron are in each orbital for the wavefunction.

Friends

- istream & **operator>>** (istream &strm, QMCWavefunction &rhs)

  Loads the state of the object from an input stream.

- ostream & **operator<<** (ostream &strm, QMCWavefunction &rhs)

  Writes the state of the object to an output stream.

3.52.1 Detailed Description

The coefficients and parameters describing the trial wavefunction for the system.

These are the coefficients for a wavefunction obtained through standard means (HF, DFT, etc.).

Definition at line 33 of file QMCWavefunction.h.

3.52.2 Member Function Documentation

3.52.2.1 int QMCWavefunction::getNumberAlphaElectrons ()

Gets the number of α spin electrons.

Returns:
number of α spin electrons.

Definition at line 34 of file QMCWavefunction.cpp.

3.52.2.2 int QMCWavefunction::getNumberBasisFunctions ()

Gets the number of basis functions.

Returns:
number of basis functions.

Definition at line 29 of file QMCWavefunction.cpp.

3.52.2.3 int QMCWavefunction::getNumberBetaElectrons ()

Gets the number of β spin electrons.

Returns:
number of β spin electrons.

Definition at line 39 of file QMCWavefunction.cpp.

3.52.2.4 int QMCWavefunction::getNumberElectrons ()

Gets the total number of electrons.

Returns:
total number of electrons.

Definition at line 44 of file QMCWavefunction.cpp.

3.52.2.5 int QMCWavefunction::getNumberOrbitals ()

Gets the number of orbitals.

Returns:
number of orbitals.

Definition at line 24 of file QMCWavefunction.cpp.

3.52.2.6 QMCWavefunction QMCWavefunction::operator= (const QMCWavefunction & *rhs*)

Sets two QMCWavefunction objects equal.

Parameters:
rhs object to set this object equal to.

Definition at line 50 of file QMCWavefunction.cpp.

References AlphaOccupation, BetaOccupation, Coeffs, Nalpha, Nbasisfunc, Nbeta, Nelectrons, and Norbitals.

3.52.2.7 void QMCWavefunction::read (int *numberOrbitals*, int *numberBasisFunctions*, string *runfile*)

Loads the state of the object from a file.

Parameters:

numberOrbitals number of orbitals.

numberBasisFunctions number of basis functions.

runfile file to load the object state from.

Definition at line 93 of file QMCWavefunction.cpp.

3.52.3 Member Data Documentation

3.52.3.1 Array2D<double> QMCWavefunction::Coeffs

Array containing the coefficients used to construct the orbitals.

For example, orbitals are constructed so that

$$Orbital_i(x,y,z) = \sum_{j=0}^{NumberBasisFunctions-1} Coeffs_{i,j} BasisFunction_j(x,y,z)$$

where the the $BasisFunction_j(x,y,z)$ are from **QMCBasisFunction** (p. 49). It is assumed that the ordering of the coefficients is the same as the basisfunctions in the input file.

Definition at line 108 of file QMCWavefunction.h.

Referenced by operator=().

3.53 SortedParameterScorePairList Class Reference

A sorted list of **ParameterScorePair** (p. 43) objects where the objects are ordered in an increasing order.

Public Methods

- **SortedParameterScorePairList** ()

  Creates an empty instance of this class.

- **SortedParameterScorePairList** (SortedParameterScorePairList &SPSL)

  Createsan instance of this class which is equal to another instance.

- int **size** ()

  Gets the number of elements in this list.

- void **add** (const **ParameterScorePair** &PSP)

  Adds a new **ParameterScorePair** (p. 43) *to this list.*

- **ParameterScorePair get** (int i)

  Gets the ith element.

- void **clear** ()

  Remove all elements from this list.

- void **operator=** (const SortedParameterScorePairList &SPSL)

  Sets two objects equal to one another.

3.53.1 Detailed Description

A sorted list of **ParameterScorePair** (p. 43) objects where the objects are ordered in an increasing order.

Definition at line 27 of file SortedParameterScorePairList.h.

3.53.2 Constructor & Destructor Documentation

3.53.2.1 SortedParameterScorePairList::SortedParameterScore-PairList (SortedParameterScorePairList & *SPSL*)

Createsan instance of this class which is equal to another instance.

Parameters:

SPSL this object to which this one will be made equal.

Definition at line 19 of file SortedParameterScorePairList.cpp.

References PSPList.

3.53.3 Member Function Documentation

3.53.3.1 void SortedParameterScorePairList::add (const Parameter-ScorePair & *PSP*)

Adds a new **ParameterScorePair** (p. 43) to this list.

Parameters:
PSP new element to add to this list.

Definition at line 29 of file SortedParameterScorePairList.cpp.

3.53.3.2 ParameterScorePair SortedParameterScorePairList::get (int *i*)

Gets the ith element.

Parameters:
i index of the element to return.

Returns:
the ith element of the list.

Definition at line 35 of file SortedParameterScorePairList.cpp.

Referenced by CKGeneticAlgorithm1::optimize().

3.53.3.3 void SortedParameterScorePairList::operator= (const SortedParameterScorePairList & *SPSL*)

Sets two objects equal to one another.

Parameters:
SPSL object to set this object equal to.

Definition at line 60 of file SortedParameterScorePairList.cpp.

References PSPList.

3.53.3.4 int SortedParameterScorePairList::size ()

Gets the number of elements in this list.

Returns:
number of elements in this list.

Definition at line 24 of file SortedParameterScorePairList.cpp.

3.54 Stopwatch Class Reference

An accurate software stopwatch.

Public Methods

- **Stopwatch** ()

  Creates an instance of the stopwatch that is zeroed and not running.

- void **reset** ()

  Resets and stops the stopwatch.

- void **start** ()

  Starts the stopwatch.

- void **stop** ()

  Stops the stopwatch.

- long **timeMS** ()

  Gets the time in milliseconds.

- bool **isRunning** ()

  Returns true if the stopwatch is running and false otherwise.

- string **toString** ()

  Gets the time formatted as a string.

- Stopwatch **operator+** (Stopwatch &rhs)

  Returns a stopwatch which contains the total time from two stopwatch objects.

Static Public Attributes

- MPI_Datatype **MPI_TYPE**

  The MPI data type for a Stopwatch.

- MPI_Op **MPI_REDUCE**

  The MPI operation for performing MPI_Reduce on Stopwatch objects.

Friends

- ostream & **operator<<** (ostream &strm, Stopwatch &watch)

  Formats and prints the time to a stream.

3.54.1 Detailed Description

An accurate software stopwatch.

Definition at line 31 of file Stopwatch.h.

3.55 StringManipulation Class Reference

A set of functions to manipulate strings.

Static Public Methods

- string **toAllUpper** (string &s)
  Converts a string to all upper case.
- string **toAllLower** (string &s)
  Converts a string to all lower case.
- string **toFirstUpperRestLower** (string &s)
  Capitalizes the first letter and lowers all others in a string.
- char **toUpperChar** (char c)
  Makes a character upper case.
- char **toLowerChar** (char c)
  Makes a character lower case.
- string **intToString** (int i)
  Returns a string representation of an integer.
- string **intToHexString** (int i)
  Returns a hexadecimal string representation of an integer.
- string **doubleToString** (double d)
  Returns a string representation of a double.
- int **stringToInt** (string &s)
  Returns an int representation of a string.
- int **hexstringToInt** (string &s)
  Returns an representation of a hexadecimal string.
- double **stringToDouble** (string &s)

Returns an double representation of a string.

3.55.1 Detailed Description

A set of functions to manipulate strings.

Definition at line 26 of file StringManipulation.h.

3.55.2 Member Function Documentation

3.55.2.1 string StringManipulation::doubleToString (double *d*) `[static]`

Returns a string representation of a double.

Parameters:
d a double.

Definition at line 163 of file StringManipulation.cpp.

Referenced by XMLElement::setAttribute().

3.55.2.2 int StringManipulation::hexstringToInt (string & *s*) `[static]`

Returns an representation of a hexadecimal string.

Parameters:
s a string.

Definition at line 183 of file StringManipulation.cpp.

3.55.2.3 string StringManipulation::intToHexString (int *i*) `[static]`

Returns a hexadecimal string representation of an integer.

Parameters:
i an integer.

Definition at line 151 of file StringManipulation.cpp.

3.55.2.4 string StringManipulation::intToString (int *i*) `[static]`

Returns a string representation of an integer.

Parameters:
i an integer.

Definition at line 139 of file StringManipulation.cpp.

Referenced by XMLElement::setAttribute(), and XMLParseException::XMLParseException().

3.55.2.5 double StringManipulation::stringToDouble (string & *s*) `[static]`

Returns an double representation of a string.

Parameters:
s a string.

Definition at line 191 of file StringManipulation.cpp.

Referenced by XMLElement::getDoubleAttribute().

3.55.2.6 int StringManipulation::stringToInt (string & *s*) `[static]`

Returns an int representation of a string.

Parameters:
s a string.

Definition at line 175 of file StringManipulation.cpp.

Referenced by XMLElement::getIntAttribute().

3.55.2.7 string StringManipulation::toAllLower (string & *s*) `[static]`

Converts a string to all lower case.

Parameters:
s a string

Definition at line 31 of file StringManipulation.cpp.

References toLowerChar().

3.55.2.8 string StringManipulation::toAllUpper (string & *s*) `[static]`

Converts a string to all upper case.

Parameters:
  s a string

Definition at line 16 of file StringManipulation.cpp.

References toUpperChar().

3.55.2.9 string StringManipulation::toFirstUpperRestLower (string & *s*) `[static]`

Capitalizes the first letter and lowers all others in a string.

Parameters:
  s a string

Definition at line 47 of file StringManipulation.cpp.

References toLowerChar(), and toUpperChar().

Referenced by QMCCorrelationFunctionParameters::read().

3.55.2.10 char StringManipulation::toLowerChar (char *c*) `[static]`

Makes a character lower case.

Parameters:
  c a character

Definition at line 105 of file StringManipulation.cpp.

Referenced by toAllLower(), and toFirstUpperRestLower().

3.55.2.11 char StringManipulation::toUpperChar (char *c*) `[static]`

Makes a character upper case.

Parameters:
  c a character

Definition at line 70 of file StringManipulation.cpp.

Referenced by toAllUpper(), and toFirstUpperRestLower().

3.56 XMLElement Class Reference

XMLElement is a representation of an XML object.

Public Methods

- **XMLElement** ()

  Creates and initializes a new XML element.

- **XMLElement** (map< string, string > *entities)

  Creates and initializes a new XML element.

- **XMLElement** (bool skipLeadingWhitespace)

  Creates and initializes a new XML element.

- **XMLElement** (map< string, string > *entities, bool skipLeading-Whitespace)

  Creates and initializes a new XML element.

- int **countChildren** ()

  Returns the number of child elements of the element.

- void **addChild** (XMLElement &child)

  Adds a child element.

- void **setAttribute** (string &name, string &value)

  Adds or modifies an attribute.

- void **setAttribute** (string &name, int value)

  Adds or modifies an attribute.

- void **setAttribute** (string &name, double value)

  Adds or modifies an attribute.

- void **parse** (string &file)

  Reads one XML element from a file and parses it.

- void **parse** (istream &reader)

  Reads one XML element from a stream and parses it.

- void **removeChild** (XMLElement &child)

  Removes a child element.

- list< XMLElement > * **getChildren** ()

  Returns the child elements as a Vector.

- string **getStringAttribute** (string &name)

  Returns an attribute of the element.

- string **getStringAttribute** (string &name, string &defaultValue)

  Returns an attribute of the element.

- int **getIntAttribute** (string &name)

  Returns an attribute of the element.

- int **getIntAttribute** (string &name, int defaultValue)

  Returns an attribute of the element.

- double **getDoubleAttribute** (string &name)

  Returns an attribute of the element.

- double **getDoubleAttribute** (string &name, double defaultValue)

  Returns an attribute of the element.

- bool **getBooleanAttribute** (string &name, string &trueValue, string &falseValue, bool defaultValue)

  Returns an attribute of the element.

- void **removeAttribute** (string &name)

  Removes an attribute.

- void **setContent** (string &content)

  Changes the content string.

- string **getContent** ()

  Returns the PCDATA content of the object.

- string **getName** ()

  Returns the name of the element.

- void **setName** (string &name)

  Changes the name of the element.

- int **getLineNr** ()

Returns the line number in the source data on which the element is found.

- void **singleLineWriter** (ostream &writer)

  Writes the XML element to an output stream as a single line.

- void **write** (string &file)

  Writes the XML element to a file using a pretty format.

- void **prettyWriter** (ostream &writer)

  Writes the XML element to an output stream using a pretty format.

- void **operator=** (XMLElement &rhs)

  Sets two objects equal to one another.

- bool **operator==** (XMLElement &rhs)

  Determines if two objects equal to one another.

3.56.1 Detailed Description

XMLElement is a representation of an XML object.

The object is able to parse and write XML code.

Definition at line 32 of file XMLElement.h.

3.56.2 Constructor & Destructor Documentation

3.56.2.1 XMLElement::XMLElement ()

Creates and initializes a new XML element.

A basic entity ("&", etc.) conversion table is used and leading whitespace is not skipped.

Definition at line 15 of file XMLElement.cpp.

3.56.2.2 XMLElement::XMLElement (map< string, string > * *entities*)

Creates and initializes a new XML element.

A basic entity ("&", etc.) conversion table and the provided entity conversion table are used and leading whitespace is not skipped.

Parameters:

entities The entity conversion table.

Definition at line 21 of file XMLElement.cpp.

3.56.2.3 XMLElement::XMLElement (bool *skipLeadingWhitespace*)

Creates and initializes a new XML element.

A basic entity ("&", etc.) conversion table is used and skipping of leading whitespace is controled by `skipLeadingWhitespace`.

Parameters:

skipLeadingWhitespace `true` if leading and trailing whitespace in PC-DATA content has to be removed.

Definition at line 27 of file XMLElement.cpp.

3.56.2.4 XMLElement::XMLElement (map< string, string > * *entities*, bool *skipLeadingWhitespace*)

Creates and initializes a new XML element.

A basic entity ("&", etc.) conversion table and the provided entity conversion table are used and leading whitespace is controled by `skipLeadingWhitespace`.

Parameters:

entities The entity conversion table.

skipLeadingWhitespace `true` if leading and trailing whitespace in PC-DATA content has to be removed.

Definition at line 33 of file XMLElement.cpp.

3.56.3 Member Function Documentation

3.56.3.1 void XMLElement::addChild (XMLElement & *child*)

Adds a child element.

Parameters:

child The child element to add.

Definition at line 93 of file XMLElement.cpp.

3.56.3.2 int XMLElement::countChildren ()

Returns the number of child elements of the element.

Returns:
number of child elements.

Definition at line 88 of file XMLElement.cpp.

3.56.3.3 bool XMLElement::getBooleanAttribute (string & *name*, string & *trueValue*, string & *falseValue*, bool *defaultValue*)

Returns an attribute of the element.

If the attribute doesn't exist, `defaultValue` is returned. If the value of the attribute is equal to `trueValue`, `true` is returned. If the value of the attribute is equal to `falseValue`, `false` is returned. If the value doesn't match `trueValue` or `falseValue`, an exception is thrown.

Parameters:
name The name of the attribute.

trueValue The value associated with `true`.

falseValue The value associated with `true`.

defaultValue Value to use if the attribute is missing.

Returns:
The value of the attribute.

Exceptions:
XMLParseException (p. 161) If the value doesn't match `trueValue` or `falseValue`.

Definition at line 274 of file XMLElement.cpp.

3.56.3.4 list< XMLElement > * XMLElement::getChildren ()

Returns the child elements as a Vector.

It is safe to modify this Vector.

Returns:
The child elements of this element.

Definition at line 202 of file XMLElement.cpp.

3.56.3.5 string XMLElement::getContent ()

Returns the PCDATA content of the object.

If there is no such content, an empty string is returned.

Returns:
PCDATA content.

Definition at line 182 of file XMLElement.cpp.

3.56.3.6 double XMLElement::getDoubleAttribute (string & *name*, double *defaultValue*)

Returns an attribute of the element.

If the attribute doesn't exist, `defaultValue` is returned.

Parameters:
name The name of the attribute.
defaultValue Key to use if the attribute is missing.

Returns:
The value of the attribute.

Definition at line 258 of file XMLElement.cpp.

References StringManipulation::stringToDouble().

3.56.3.7 double XMLElement::getDoubleAttribute (string & *name*)

Returns an attribute of the element.

If the attribute doesn't exist, `0.0` is returned.

Parameters:
name The name of the attribute.

Returns:
The value of the attribute.

Definition at line 252 of file XMLElement.cpp.

3.56.3.8 int XMLElement::getIntAttribute (string & *name*, int *defaultValue*)

Returns an attribute of the element.

If the attribute doesn't exist, `defaultValue` is returned.

Parameters:
name The name of the attribute.

defaultValue Key to use if the attribute is missing.

Returns:
The value of the attribute.

Definition at line 236 of file XMLElement.cpp.

References StringManipulation::stringToInt().

3.56.3.9 int XMLElement::getIntAttribute (string & *name*)

Returns an attribute of the element.

If the attribute doesn't exist, `0` is returned.

Parameters:
name The name of the attribute.

Returns:
The value of the attribute.

Definition at line 230 of file XMLElement.cpp.

3.56.3.10 int XMLElement::getLineNr ()

Returns the line number in the source data on which the element is found.

This method returns `0` there is no associated source data.

Returns:
Line number in the source data on which the element is found.

Definition at line 197 of file XMLElement.cpp.

3.56.3.11 string XMLElement::getName ()

Returns the name of the element.

Returns:
name of the element.

Definition at line 187 of file XMLElement.cpp.

3.56.3.12 string XMLElement::getStringAttribute (string & *name*, string & *defaultValue*)

Returns an attribute of the element.

If the attribute doesn't exist, `defaultValue` is returned.

Parameters:
name The name of the attribute.

defaultValue Key to use if the attribute is missing.

Returns:
The value of the attribute.

Definition at line 216 of file XMLElement.cpp.

3.56.3.13 string XMLElement::getStringAttribute (string & *name*)

Returns an attribute of the element.

If the attribute doesn't exist, an empty string is returned.

Parameters:
name The name of the attribute.

Returns:
The value of the attribute.

Definition at line 209 of file XMLElement.cpp.

3.56.3.14 void XMLElement::operator= (XMLElement & *rhs*)

Sets two objects equal to one another.

Parameters:
rhs object to set this object equal to.

Definition at line 1037 of file XMLElement.cpp.

References attributes, children, contents, entities, ignoreWhitespace, lineNr, and name.

3.56.3.15 bool XMLElement::operator== (XMLElement & *rhs*)

Determines if two objects equal to one another.

Parameters:
rhs object to determine if this one is equal to.

Returns:
`true` if both objects are equal and `false` otherwise.

Definition at line 1063 of file XMLElement.cpp.

References attributes, children, contents, and name.

3.56.3.16 void XMLElement::parse (istream & *reader*)

Reads one XML element from a stream and parses it.

Parameters:
reader The stream from which to retrieve the XML data.

Exceptions:
XMLParseException (p. 161) If an error occured while parsing the read data.

Definition at line 124 of file XMLElement.cpp.

References parse().

3.56.3.17 void XMLElement::parse (string & *file*)

Reads one XML element from a file and parses it.

Parameters:
file The file from which to retrieve the XML data.

Exceptions:
XMLParseException (p. 161) If an error occured while parsing the read data.

Definition at line 116 of file XMLElement.cpp.

Referenced by parse().

3.56.3.18 void XMLElement::prettyWriter (ostream & *writer*)

Writes the XML element to an output stream using a pretty format.

Parameters:
writer The stream to write the XML data to.

Definition at line 929 of file XMLElement.cpp.

Referenced by write().

3.56.3.19 void XMLElement::removeAttribute (string & *name*)

Removes an attribute.

Parameters:

name The name of the attribute.

Definition at line 172 of file XMLElement.cpp.

3.56.3.20 void XMLElement::removeChild (XMLElement & *child*)

Removes a child element.

Parameters:

child The child element to remove.

Definition at line 159 of file XMLElement.cpp.

3.56.3.21 void XMLElement::setAttribute (string & *name*, double *value*)

Adds or modifies an attribute.

Parameters:

name The name of the attribute.

value The value of the attribute.

Definition at line 111 of file XMLElement.cpp.

References StringManipulation::doubleToString().

3.56.3.22 void XMLElement::setAttribute (string & *name*, int *value*)

Adds or modifies an attribute.

Parameters:

name The name of the attribute.

value The value of the attribute.

Definition at line 105 of file XMLElement.cpp.

References StringManipulation::intToString().

3.56.3.23 void XMLElement::setAttribute (string & *name*, string & *value*)

Adds or modifies an attribute.

Parameters:

name The name of the attribute.

value The value of the attribute.

Definition at line 99 of file XMLElement.cpp.

3.56.3.24 void XMLElement::setContent (string & *content*)

Changes the content string.

Parameters:

content The new content string.

Definition at line 177 of file XMLElement.cpp.

3.56.3.25 void XMLElement::setName (string & *name*)

Changes the name of the element.

Parameters:

name The new name.

Definition at line 192 of file XMLElement.cpp.

3.56.3.26 void XMLElement::singleLineWriter (ostream & *writer*)

Writes the XML element to an output stream as a single line.

Parameters:

writer The stream to write the XML data to.

Definition at line 858 of file XMLElement.cpp.

3.56.3.27 void XMLElement::write (string & *file*)

Writes the XML element to a file using a pretty format.

Parameters:

file The file to write the XML data to.

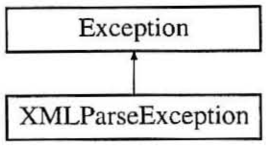
Definition at line 922 of file XMLElement.cpp.

References prettyWriter().

3.57 XMLParseException Class Reference

An XMLParseException is thrown when an error occures while parsing an XML stream.

Inheritance diagram for XMLParseException::



Public Methods

- **XMLParseException** (string name, string message)

  Creates an exception.

- **XMLParseException** (string name, int lineNr, string message)

  Creates an exception.

- int **getLineNr** ()

  Where the error occurred, or `NO_LINE` *if the line number is unknown.*

- void **setMessage** (string message)

  Sets the error message for the exception.

- string **getMessage** ()

  Gets the error message for the exception.

Static Public Attributes

- int **NO_LINE** = -1

  Indicates that no line number has been associated with this exception.

3.57.1 Detailed Description

An XMLParseException is thrown when an error occures while parsing an XML stream.

Definition at line 28 of file XMLParseException.h.

3.57.2 Constructor & Destructor Documentation

3.57.2.1 XMLParseException::XMLParseException (string *name*, string *message*)

Creates an exception.

Parameters:

name The name of the element where the error is located.

message A message describing what went wrong.

Definition at line 18 of file XMLParseException.cpp.

References NO_LINE, and Exception::setMessage().

3.57.2.2 XMLParseException::XMLParseException (string *name*, int *lineNr*, string *message*)

Creates an exception.

Parameters:

name The name of the element where the error is located.

lineNr The number of the line in the input.

message A message describing what went wrong.

Definition at line 29 of file XMLParseException.cpp.

References StringManipulation::intToString(), and Exception::setMessage().

3.57.3 Member Function Documentation

3.57.3.1 int XMLParseException::getLineNr ()

Where the error occurred, or `NO_LINE` if the line number is unknown.

Returns:

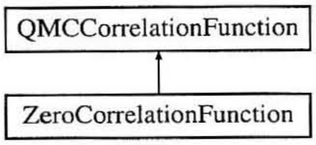
Line number where the error occurred.

Definition at line 41 of file XMLParseException.cpp.

3.58 ZeroCorrelationFunction Class Reference

Correlation function which describes noninteracting particles.

Inheritance diagram for ZeroCorrelationFunction::



Public Methods

- void **initializeParameters** (**Array1D**< int > &BeginningIndexOfParameterType, **Array1D**< double > &Parameters, **Array1D**< int > &BeginningIndexOfConstantType, **Array1D**< double > &Constants)

  Initializes the correlation function with a specified set of parameters.

- void **evaluate** (double **r**)

  Evaluates the correlation function and it's first two derivatives at r.

- bool **isSingular** ()

  Returns true if the correlation function has a singularity in the domain $r \geq 0$, and false otherwise.

- double **getFunctionValue** ()

  Gets the value of the correlation function for the last evaluated r.

- double **getFirstDerivativeValue** ()

  Gets the value of the first derivative of the correlation function for the last evaluated r.

- double **getSecondDerivativeValue** ()

  Gets the value of the second derivative of the correlation function for the last evaluated r.

3.58.1 Detailed Description

Correlation function which describes noninteracting particles.

Definition at line 24 of file ZeroCorrelationFunction.h.

3.58.2 Member Function Documentation

3.58.2.1 void ZeroCorrelationFunction::initializeParameters (Array1D< int > & *BeginningIndexOfParameterType*, Array1D<

double > & ***Parameters*****, Array1D< int > &** ***BeginningIndexOfConstantType*****, Array1D< double > &** ***Constants*****)** `[virtual]`

Initializes the correlation function with a specified set of parameters.

This must be called every time the parameters are changed.

Implements **QMCCorrelationFunction** (p. 58).

Definition at line 16 of file ZeroCorrelationFunction.cpp.

4 QMcBeaver File Documentation

4.1 ckfastfunctions.h File Reference

This is a fast function library originally intended to speed up QMcBeaver a Quantum Monte Carlo program.

Functions

- double **fastPower** (double x, int n)

  Fast power function for use when the exponent is a small integer.

4.1.1 Detailed Description

This is a fast function library originally intended to speed up QMcBeaver a Quantum Monte Carlo program.

Definition in file **ckfastfunctions.h**.

4.1.2 Function Documentation

4.1.2.1 double fastPower (double x, int n)

Fast power function for use when the exponent is a small integer.

Parameters:

x base

n exponent

Returns:

x^n

Definition at line 15 of file ckfastfunctions.cpp.

4.2 LU.h File Reference

Library of matrix functions which involve LU decompositions.

Functions

- void **ludcmp** (**Array2D**< double > &a, int *indx, double *d, bool *calcOK)

  LU decomposition using the algorithm in numerical recipes for a dense matrix.

- void **lubksb** (**Array2D**< double > &a, int *indx, **Array1D**< double > &b)

  LU backsubstitution using the algorithm in numerical recipes for a dense matrix.

- double **determinant** (**Array2D**< double > a, bool *calcOK)

  Calculates a determinant of a matrix using a dense LU solver.

- **Array2D**< double > **inverse** (**Array2D**< double > a, bool *calcOK)

  Calculates the inverse of a matrix using a dense LU solver.

- void **determinant_and_inverse** (**Array2D**< double > a, **Array2D**< double > &inv, double &det, bool *calcOK)

  Calculates the inverse and determinant of a matrix using a dense LU solver.

- void **linearsolver** (**Array2D**< double > &a, **Array1D**< double > &b, bool *calcOK)

  Solves a system of linear equations using a dense LU solver.

4.2.1 Detailed Description

Library of matrix functions which involve LU decompositions.

Definition in file **LU.h**.

4.2.2 Function Documentation

4.2.2.1 double determinant (Array2D< double > *a*, bool * *calcOK*)

Calculates a determinant of a matrix using a dense LU solver.

This method scales as $O(\frac{1}{3}N^3)$.

Parameters:

a a $N \times N$ matrix

calcOK returns false if the calculation is singular and true otherwise

Returns:

the determinant of a

Definition at line 115 of file LU.cpp.

4.2.2.2 void determinant_and_inverse (Array2D< double > *a*, Array2D< double > & *inv*, double & *det*, bool * *calcOK*)

Calculates the inverse and determinant of a matrix using a dense LU solver.

This method scales as $O(1N^3)$.

Parameters:

a a $N \times N$ matrix

inv inverse of a is returned here

det determinant of a is returned here

calcOK returns false if the calculation is singular and true otherwise

Definition at line 161 of file LU.cpp.

4.2.2.3 Array2D<double> inverse (Array2D< double > *a*, bool * *calcOK*)

Calculates the inverse of a matrix using a dense LU solver.

This method scales as $O(1N^3)$.

Parameters:

a a $N \times N$ matrix

calcOK returns false if the calculation is singular and true otherwise

Returns:

the inverse of a

Definition at line 134 of file LU.cpp.

4.2.2.4 void linearsolver (Array2D< double > & *a*, Array1D< double > & *b*, bool * *calcOK*)

Solves a system of linear equations using a dense LU solver.

this method scales as $O(\frac{1}{3}N^3)$.

Parameters:

a a $N \times N$ matrix. This matrix is destroyed in the calculation.

b the N dimensional right hand side to solve for. Result is returned here and the original values are destroyed.

calcOK returns false if the calculation is singular and true otherwise

Definition at line 191 of file LU.cpp.

4.2.2.5 void lubksb (Array2D< double > & *a*, int * *indx*, Array1D< double > & *b*)

LU backsubstitution using the algorithm in numerical recipes for a dense matrix.

Parameters:

a the LU decomposition of a matrix produced by ludcmp

indx a N dimensional array which records the row permutation from partial pivoting generated by ludcmp

b the N dimensional array right hand side of the system of equations to solve

Definition at line 90 of file LU.cpp.

4.2.2.6 void ludcmp (Array2D< double > & *a*, int * *indx*, double * *d*, bool * *calcOK*)

LU decomposition using the algorithm in numerical recipes for a dense matrix.

Parameters:

a a $N \times N$ matrix which is destroyed during the operation. The resulting LU decompositon is placed here.

indx a N dimensional array which records the row permutation from partial pivoting.

d used to give det(a) the correct sign

calcOK returns false if the calculation is singular and true otherwise

Definition at line 23 of file LU.cpp.

4.3 mfrandom.h File Reference

Library of functions for generating random numbers.

Functions

- double **gasdev** (long *idum)

  Generates a gaussian distributed random number with unit variance using the gasdev algorithm from numerical recipes.

- double **ran1** (long *idum)

  Generates a uniform random number on [0, 1] *using the ran1 algorithm from numerical recipes.*

4.3.1 Detailed Description

Library of functions for generating random numbers.

Definition in file **mfrandom.h**.

4.3.2 Function Documentation

4.3.2.1 double gasdev (long * *idum*)

Generates a gaussian distributed random number with unit variance using the gasdev algorithm from numerical recipes.

Parameters:
idum random number seed

Returns:
gaussian random number with unit variance

Definition at line 16 of file mfrandom.cpp.

4.3.2.2 double ran1 (long * *idum*)

Generates a uniform random number on $[0, 1]$ using the ran1 algorithm from numerical recipes.

Parameters:
idum random number seed

Returns:
uniform random number on $[0, 1]$.

Definition at line 55 of file mfrandom.cpp.

Index

~Array1D
    Array1D, 14
~Array2D
    Array2D, 17
~Array3D
    Array3D, 20
~Array4D
    Array4D, 24
~FunctionR1toR1
    FunctionR1toR1, 45
~QMCCorrelationFunction
    QMCCorrelationFunction, 62
~QMCCorrelationFunctionParameters
    QMCCorrelationFunction-
        Parameters, 64
~QMCInitializeWalker
    QMCInitializeWalker, 77
~QMCLineSearch
    QMCLineSearch, 94
~QMCLineSearchStepLengthSelectionAlgorithm
    QMCLineSearchStepLength-
        SelectionAlgorithm, 95
~QMCManager
    QMCManager, 97
~QMCOptimizationAlgorithm
    QMCOptimizationAlgorithm,
        111
~QMCwalker
    QMCwalker, 139

abs
    Complex, 30
acceptanceProbability
    QMCproperties, 120
add
    SortedParameterScorePair-
        List, 148
addChild
    XMLElement, 158
allocate
    Array1D, 15
    Array2D, 18
    Array3D, 21
    Array4D, 25
AlphaOccupation
    QMCWavefunction, 145
array
    Array1D, 15
    Array2D, 19
    Array3D, 22
    Array4D, 25
Array1D, 13
    ~Array1D, 14
    allocate, 15
    array, 15
    Array1D, 14, 15
    deallocate, 13
    dim1, 15
    operator *, 13
    operator *=, 13
    operator(), 14
    operator+, 13
    operator-, 13
    operator/=, 14
    operator<<, 14
    operator=, 13
    size, 16
Array2D, 16
    ~Array2D, 17
    allocate, 18
    array, 19
    Array2D, 17, 18
    deallocate, 17
    dim1, 19
    dim2, 19
    operator *, 17
    operator *=, 17
    operator(), 17
    operator/=, 17
    operator<<, 17
    operator=, 17
    size, 19
Array3D, 20
    ~Array3D, 20
    allocate, 21
    array, 22

Array3D, 20, 21
deallocate, 20
dim1, 22
dim2, 22
dim3, 22
operator(), 21
operator=, 20
size, 22
Array4D, 23
~Array4D, 24
allocate, 25
array, 25
Array4D, 24
deallocate, 23
dim1, 25
dim2, 25
dim3, 26
dim4, 26
operator(), 24
operator=, 23
size, 26
Atom_Labels
QMCMolecule, 104
Atom_Positions
QMCMolecule, 104

BetaOccupation
QMCWavefunction, 145

calculateObservables
QMCwalker, 141
ckfastfunctions.h, 169
fastPower, 169
CKGeneticAlgorithm1
CKGeneticAlgorithm1, 27
CKGeneticAlgorithm1, 26
CKGeneticAlgorithm1, 27
optimize, 28
clear
SortedParameterScorePair-List, 148
Coeffs
QMCBasisFunction-Coefficients, 58
QMCWavefunction, 147
Complex, 28
abs, 30
Complex, 28, 30
conjugate, 30
imaginary, 31
operator *, 31
operator+, 31, 32
operator-, 32
operator/, 32
operator<<, 30
operator=, 32, 33
real, 33
squareroot, 33
conjugate
Complex, 30
correlationFunctionFactory
QMCCorrelationFunction-Factory, 64
countChildren
XMLElement, 158
CubicSpline
CubicSpline, 34
CubicSpline, 33
evaluate, 35
getFirstDerivativeValue, 35
getFunctionValue, 36
getSecondDerivativeValue, 36
initializeWithDerivativeValues, 36
initializeWithFunctionValues, 36
operator=, 37
toXML, 37
CubicSplineWithGeometricProgressionGrid
CubicSplineWithGeometric-ProgressionGrid, 38
CubicSplineWithGeometric-ProgressionGrid, 37
evaluate, 39
getFirstDerivativeValue, 40
getFunctionValue, 40
getSecondDerivativeValue, 40
initializeWithDerivativeValues, 40
initializeWithFunctionValues, 41
operator=, 41

setGridParameters, 41
toXML, 42

deallocate
Array1D, 13
Array2D, 17
Array3D, 20
Array4D, 23
determinant
LU.h, 170
determinant_and_inverse
LU.h, 171
dim1
Array1D, 15
Array2D, 19
Array3D, 22
Array4D, 25
dim2
Array2D, 19
Array3D, 22
Array4D, 25
dim3
Array3D, 22
Array4D, 26
dim4
Array4D, 26
distanceMovedAccepted
QMCproperties, 120
distanceMovedTrial
QMCproperties, 120
doubleToString
StringManipulation, 152

energy
QMCproperties, 120
evaluate
CubicSpline, 35
CubicSplineWithGeometric-ProgressionGrid, 39
FixedCuspPadeCorrelation-Function, 44
FunctionR1toR1, 46
PadeCorrelationFunction, 47
Polynomial, 52
QMCCorrelationFunction, 63
QMCFunctions, 74
QMCJastrow, 80, 81
QMCJastrowElectronElectron, 84
QMCJastrowElectronNuclear, 86
QMCObjectiveFunction, 105, 106
QMCPolynomial, 115
QMCPotential_Energy, 118
QMCSlater, 130
ZeroCorrelationFunction, 168
Exception, 42
Exception, 42, 43
getMessage, 43
setMessage, 43

factory
QMCLineSearchStepLength-SelectionFactory, 96
fastPower
ckfastfunctions.h, 169
finalize
QMCManager, 97
FixedCuspPadeCorrelationFunction
evaluate, 44
getFirstDerivativeValue, 44
getFunctionValue, 44
getSecondDerivativeValue, 44
isSingular, 44
FixedCuspPadeCorrelation-Function, 43
initializeParameters, 44
FunctionR1toR1
~FunctionR1toR1, 45
FunctionR1toR1, 45
evaluate, 46
getFirstDerivativeValue, 46
getFunctionValue, 46
getSecondDerivativeValue, 46

gasdev
mfrandom.h, 173
get
SortedParameterScorePair-List, 149
getAverage

QMCproperty, 122
QMCstatistic, 132
getBooleanAttribute
XMLElement, 159
getChildren
XMLElement, 159
getCoefficient
Polynomial, 52
QMCPolynomial, 115
getCommandPollingStopwatch
QMCStopwatches, 137
getCommunicationSynchronizationStopwatch
QMCStopwatches, 137
getContent
XMLElement, 159
getCorrelationFunction
QMCCorrelationFunction-Parameters, 66
getDerivativeScore
QMCObjectiveFunction-Result, 109
getDoubleAttribute
XMLElement, 160
getEffectiveTimeStep
QMCDerivativeProperties, 70
getEffectiveTimeStepStandardDeviation
QMCDerivativeProperties, 70
getEffectiveTimeStepVariance
QMCDerivativeProperties, 71
getElectronDownElectronDownParameters
QMCJastrowParameters, 90
getElectronDownNuclearParameters
QMCJastrowParameters, 90
getElectronUpElectronDownParameters
QMCJastrowParameters, 91
getElectronUpElectronUpParameters
QMCJastrowParameters, 91
getElectronUpNuclearParameters
QMCJastrowParameters, 91
getEnergy
QMCPotential_Energy, 118
getEnergyAve
QMCObjectiveFunction-Result, 109
getEnergyVar
QMCObjectiveFunction-Result, 110
getFirstDerivativeValue
CubicSpline, 35
CubicSplineWithGeometric-ProgressionGrid, 40
FixedCuspPadeCorrelation-Function, 44
FunctionR1toR1, 46
PadeCorrelationFunction, 47
Polynomial, 52
QMCCorrelationFunction, 63
QMCPolynomial, 115
ZeroCorrelationFunction, 168
getFunctionValue
CubicSpline, 36
CubicSplineWithGeometric-ProgressionGrid, 40
FixedCuspPadeCorrelation-Function, 44
FunctionR1toR1, 46
PadeCorrelationFunction, 47
Polynomial, 52
QMCCorrelationFunction, 63
QMCPolynomial, 116
ZeroCorrelationFunction, 168
getGatherPropertiesStopwatch
QMCStopwatches, 138
getGradientLnJastrow
QMCJastrow, 81
QMCJastrowElectronElectron, 84
QMCJastrowElectronNuclear, 87
getGradPsi
QMCBasisFunction, 55
getGradPsiRatio
QMCFunctions, 74
QMCSlater, 131
getInitializationStopwatch
QMCStopwatches, 136
getInputData
QMCManager, 98
getIntAttribute
XMLElement, 160, 161
getJastrow

QMCJastrow, 81
getKineticEnergy
QMCFunctions, 75
getLaplacianLnJastrow
QMCJastrow, 81
QMCJastrowElectronElectron, 84
QMCJastrowElectronNuclear, 87
getLaplacianPsi
QMCBasisFunction, 56
getLaplacianPsiRatio
QMCSlater, 131
getLineNr
XMLElement, 161
XMLParseException, 167
getLnJastrow
QMCJastrow, 82
QMCJastrowElectronElectron, 85
QMCJastrowElectronNuclear, 87
getLocalEnergy
QMCFunctions, 75
getLocalEnergyEstimator
QMCwalker, 140
getLogWeightsAve
QMCObjectiveFunction-Result, 110
getLogWeightsVar
QMCObjectiveFunction-Result, 110
getMessage
Exception, 43
XMLParseException, 166
getModifiedGradPsiRatio
QMCFunctions, 75
getName
XMLElement, 161
getNucleiTypes
QMCJastrowParameters, 89
getNumberAlphaElectrons
QMCWavefunction, 145
getNumberAtoms
QMCMolecule, 103
getNumberBasisFunctions
QMCBasisFunction, 56
QMCBasisFunction-Coefficients, 60
QMCWavefunction, 145
getNumberBetaElectrons
QMCWavefunction, 146
getNumberCoefficients
Polynomial, 53
QMCPolynomial, 116
getNumberElectrons
QMCWavefunction, 146
getNumberOfWalkers
QMCrun, 127
getNumberOrbitals
QMCWavefunction, 146
getNumberSamples
QMCproperty, 122
QMCstatistic, 132
getObjectiveFunction
QMCLineSearch, 94
QMCSteepestDescent, 135
getOptimizationStopwatch
QMCStopwatches, 138
getParameters
ParameterScorePair, 49
QMCCorrelationFunction-Parameters, 66
QMCJastrowParameters, 92
getParticle1Type
QMCCorrelationFunction-Parameters, 67
getParticle2Type
QMCCorrelationFunction-Parameters, 67
getPotentialEnergy
QMCFunctions, 75
getPropagationStopwatch
QMCStopwatches, 138
getProperties
QMCrun, 127
getPsi
QMCBasisFunction, 56
QMCFunctions, 76
QMCSlater, 131
getResultsOutputStream
QMCManager, 98

getRoots
  Polynomial, 53
  QMCPolynomial, 116
getScore
  ParameterScorePair, 49
  QMCObjectiveFunction-
    Result, 110
getSecondDerivativeValue
  CubicSpline, 36
  CubicSplineWithGeometric-
    ProgressionGrid, 40
  FixedCuspPadeCorrelation-
    Function, 44
  FunctionR1toR1, 46
  PadeCorrelationFunction, 47
  Polynomial, 53
  QMCCorrelationFunction, 63
  QMCPolynomial, 117
  ZeroCorrelationFunction, 168
getSendCommandStopwatch
  QMCStopwatches, 138
getSeriallyCorrelatedStandardDeviation
  QMCproperty, 122
getSeriallyCorrelatedVariance
  QMCproperty, 122
getStandardDeviation
  QMCproperty, 122
  QMCstatistic, 132
getStringAttribute
  XMLElement, 161, 162
getTotalNumberOfParameters
  QMCCorrelationFunction-
    Parameters, 67
getTotalTimeStopwatch
  QMCStopwatches, 139
getVariance
  QMCproperty, 122
  QMCstatistic, 132
getVirialRatio
  QMCDerivativeProperties, 71
getVirialRatioStandardDeviation
  QMCDerivativeProperties, 71
getVirialRatioVariance
  QMCDerivativeProperties, 72
getWeight
  QMCwalker, 141
getWeights
  QMCrun, 128
grad
  QMCObjectiveFunction, 106

hasNonNegativeZeroes
  QMCPolynomial, 117
hexstringToInt
  StringManipulation, 152

imaginary
  Complex, 31
initialize
  Polynomial, 54
  QMCBasisFunction, 57
  QMCFunctions, 76
  QMCJastrow, 82
  QMCJastrowElectronElectron,
    85
  QMCJastrowElectronNuclear,
    88
  QMCManager, 98
  QMCMolecule, 103
  QMCObjectiveFunction, 107
  QMCPolynomial, 117
  QMCPotential_Energy, 119
  QMCReadAndEvaluate-
    Configs, 125
  QMCrun, 128
  QMCSlater, 131
  QMCwalker, 141
initializeParameters
  FixedCuspPadeCorrelation-
    Function, 44
  PadeCorrelationFunction, 48
  QMCCorrelationFunction, 63
  ZeroCorrelationFunction, 168
initializeWalkerFactory
  QMCInitializeWalkerFactory,
    79
initializeWalkerPosition
  QMCInitializeWalker, 78
  QMCMikesJackedWalker-
    Initialization, 101
  QMCwalker, 142
initializeWithDerivativeValues

CubicSpline, 36
CubicSplineWithGeometric-
ProgressionGrid, 40
initializeWithFunctionValues
CubicSpline, 36
CubicSplineWithGeometric-
ProgressionGrid, 41
intToHexString
StringManipulation, 152
intToString
StringManipulation, 152
inverse
LU.h, 171
isRunning
Stopwatch, 150
isSingular
FixedCuspPadeCorrelation-
Function, 44
PadeCorrelationFunction, 47
QMCCorrelationFunction, 62
QMCCorrelationFunction-
Parameters, 67
QMCFunctions, 76
QMCSlater, 130
QMCwalker, 142
ZeroCorrelationFunction, 168

kineticEnergy
QMCproperties, 120

linearsolver
LU.h, 171
logWeights
QMCproperties, 120
LU.h, 170
determinant, 170
determinant_and_inverse, 171
inverse, 171
linearsolver, 171
lubksb, 172
ludcmp, 172
lubksb
LU.h, 172
ludcmp
LU.h, 172

mfrandom.h, 172
gasdev, 173
ran1, 173
MPI_REDUCE
QMCproperties, 120
QMCproperty, 123
QMCstatistic, 133
QMCStopwatches, 137
Stopwatch, 150
MPI_TYPE
QMCproperties, 120
QMCproperty, 123
QMCstatistic, 133
QMCStopwatches, 137
Stopwatch, 150

N_Gauss
QMCBasisFunction-
Coefficients, 58
newSample
QMCproperty, 123
QMCstatistic, 133
NO_LINE
XMLParseException, 166
NucleiTypes
QMCMolecule, 102

operator *
Array1D, 13
Array2D, 17
Complex, 31
operator *=
Array1D, 13
Array2D, 17
operator()
Array1D, 14
Array2D, 17
Array3D, 21
Array4D, 24
operator+
Array1D, 13
Complex, 31, 32
QMCproperties, 119
QMCproperty, 122
QMCstatistic, 132
QMCStopwatches, 137
Stopwatch, 150

operator-
  Array1D, 13
  Complex, 32
operator/
  Complex, 32
operator/=
  Array1D, 14
  Array2D, 17
operator<
  ParameterScorePair, 49
operator<<
  Array1D, 14
  Array2D, 17
  Complex, 30
  QMCBasisFunction, 55
  QMCBasisFunction-
    Coefficients, 59
  QMCCopyright, 61
  QMCCorrelationFunction-
    Parameters, 65
  QMCDerivativeProperties, 70
  QMCJastrowParameters, 89
  QMCManager, 98
  QMCMolecule, 103
  QMCObjectiveFunction-
    Result, 108
  QMCproperties, 121
  QMCproperty, 123
  QMCstatistic, 133
  QMCStopwatches, 137
  QMCWavefunction, 145
  Stopwatch, 150
operator=
  Array1D, 13
  Array2D, 17
  Array3D, 20
  Array4D, 23
  Complex, 32, 33
  CubicSpline, 37
  CubicSplineWithGeometric-
    ProgressionGrid, 41
  ParameterScorePair, 50
  QMCBasisFunction, 57
  QMCBasisFunction-
    Coefficients, 60
  QMCCorrelationFunction-
    Parameters, 68
  QMCFunctions, 76
  QMCJastrowParameters, 92
  QMCMolecule, 103
  QMCObjectiveFunction-
    Result, 111
  QMCPotential_Energy, 119
  QMCSlater, 132
  QMCwalker, 142
  QMCWavefunction, 146
  SortedParameterScorePair-
    List, 149
  XMLElement, 162
operator==
  XMLElement, 162
operator>>
  QMCBasisFunction, 55
  QMCBasisFunction-
    Coefficients, 59
  QMCMolecule, 102
  QMCWavefunction, 145
optimizationAlgorithmFactory
  QMCOptimizationFactory,
    113
optimize
  CKGeneticAlgorithm1, 28
  QMCCorrelatedSampling-
    VMCOptimization, 62
  QMCLineSearch, 95
  QMCManager, 97
  QMCOptimizationAlgorithm,
    112
  QMCSteepestDescent, 135

PadeCorrelationFunction
  evaluate, 47
  getFirstDerivativeValue, 47
  getFunctionValue, 47
  getSecondDerivativeValue, 47
  isSingular, 47
PadeCorrelationFunction, 47
  initializeParameters, 48
ParameterScorePair
  operator<, 49
  ParameterScorePair, 48, 49

ParameterScorePair, 48
 getParameters, 49
 getScore, 49
 operator=, 50
 ParameterScorePair, 49
parse
 XMLElement, 163
Polynomial, 50
 evaluate, 52
 getCoefficient, 52
 getFirstDerivativeValue, 52
 getFunctionValue, 52
 getNumberCoefficients, 53
 getRoots, 53
 getSecondDerivativeValue, 53
 initialize, 54
 Polynomial, 50, 51
potentialEnergy
 QMCproperties, 120
prettyWriter
 XMLElement, 163
propagateWalker
 QMCwalker, 140

QMCBasisFunction
 operator<<, 55
 operator>>, 55
 QMCBasisFunction, 54
QMCBasisFunction, 54
 getGradPsi, 55
 getLaplacianPsi, 56
 getNumberBasisFunctions, 56
 getPsi, 56
 initialize, 57
 operator=, 57
 read, 57
QMCBasisFunctionCoefficients
 Coeffs, 58
 N_Gauss, 58
 operator<<, 59
 operator>>, 59
 QMCBasisFunction-
 Coefficients, 58
 read, 58
QMCBasisFunctionCoefficients, 58
 getNumberBasisFunctions, 60
 operator=, 60
 Type, 60
 xyz_powers, 60
QMCCopyright, 61
 operator<<, 61
QMCCorrelatedSampling-
 VMCOptimization, 61
 optimize, 62
QMCCorrelationFunction
 ~QMCCorrelationFunction,
 62
 evaluate, 63
 getFirstDerivativeValue, 63
 getFunctionValue, 63
 getSecondDerivativeValue, 63
 isSingular, 62
QMCCorrelationFunction, 62
 initializeParameters, 63
QMCCorrelationFunctionFactory
 correlationFunctionFactory, 64
QMCCorrelationFunctionFactory,
 63
QMCCorrelationFunctionParameters
 ~QMCCorrelationFunction-
 Parameters, 64
 operator<<, 65
 QMCCorrelationFunction-
 Parameters, 64, 66
 setParticle1Type, 65
 setParticle2Type, 65
QMCCorrelationFunction-
 Parameters, 64
 getCorrelationFunction, 66
 getParameters, 66
 getParticle1Type, 67
 getParticle2Type, 67
 getTotalNumberOfParame-
 ters, 67
 isSingular, 67
 operator=, 68
 QMCCorrelationFunctionPa-
 rameters, 66
 read, 68
 setParameters, 68
QMCDerivativeProperties
 operator<<, 70

QMCDerivativeProperties, 70
QMCDerivativeProperties, 69
getEffectiveTimeStep, 70
getEffectiveTimeStepStandardDeviation, 70
getEffectiveTimeStepVariance, 71
getVirialRatio, 71
getVirialRatioStandardDeviation, 71
getVirialRatioVariance, 72
QMCDerivativeProperties, 70
QMCFunctions, 72
evaluate, 74
getGradPsiRatio, 74
getKineticEnergy, 75
getLocalEnergy, 75
getModifiedGradPsiRatio, 75
getPotentialEnergy, 75
getPsi, 76
initialize, 76
isSingular, 76
operator=, 76
QMCFunctions, 72, 74
writeCorrelatedSamplingConfiguration, 77
QMCInitializeWalker
~QMCInitializeWalker, 77
QMCInitializeWalker, 77
initializeWalkerPosition, 78
QMCInitializeWalkerFactory, 78
initializeWalkerFactory, 79
QMCJastrow, 79
evaluate, 80, 81
getGradientLnJastrow, 81
getJastrow, 81
getLaplacianLnJastrow, 81
getLnJastrow, 82
initialize, 82
QMCJastrowElectronElectron, 82
evaluate, 84
getGradientLnJastrow, 84
getLaplacianLnJastrow, 84
getLnJastrow, 85
initialize, 85
QMCJastrowElectronNuclear, 85
evaluate, 86
getGradientLnJastrow, 87
getLaplacianLnJastrow, 87
getLnJastrow, 87
initialize, 88
QMCJastrowParameters
getNucleiTypes, 89
operator<<, 89
QMCJastrowParameters, 88, 90
QMCJastrowParameters, 88
getElectronDownElectronDownParameters, 90
getElectronDownNuclearParameters, 90
getElectronUpElectronDownParameters, 91
getElectronUpElectronUpParameters, 91
getElectronUpNuclearParameters, 91
getParameters, 92
operator=, 92
QMCJastrowParameters, 90
read, 92
setParameterVector, 93
QMCLineSearch
~QMCLineSearch, 94
getObjectiveFunction, 94
QMCLineSearch, 94
QMCLineSearch, 93
optimize, 95
QMCLineSearch, 94
QMCLineSearchStepLengthSelectionAlgorithm
~QMCLineSearchStepLengthSelectionAlgorithm, 95
QMCLineSearchStepLengthSelectionAlgorithm, 95
stepLength, 96
QMCLineSearchStepLengthSelectionFactory
factory, 96
QMCLineSearchStepLengthSelectionFactory, 96
QMCManager, 97
~QMCManager, 97
finalize, 97

getInputData, 98
getResultsOutputStream, 98
initialize, 98
operator<<, 98
optimize, 97
QMCManager, 97
run, 98
writeRestart, 97
writeTimingData, 99
zeroOut, 97
QMCMikesBracketingStepLength-
Selector, 99
stepLength, 100
QMCMikesJackedWalkerInitialization
QMCMikesJackedWalker-
Initialization, 101
QMCMikesJackedWalker-
Initialization, 100
initializeWalkerPosition, 101
QMCMikesJackedWalkerIni-
tialization, 101
QMCMolecule, 102
Atom_Labels, 104
Atom_Positions, 104
getNumberAtoms, 103
initialize, 103
NucleiTypes, 102
operator<<, 103
operator=, 103
operator>>, 102
QMCMolecule, 102
read, 103
Z, 104
QMCObjectiveFunction, 104
evaluate, 105, 106
grad, 106
initialize, 107
QMCObjectiveFunctionResult
operator<<, 108
QMCObjectiveFunction-
Result, 107, 109
QMCObjectiveFunctionResult, 107
getDerivativeScore, 109
getEnergyAve, 109
getEnergyVar, 110
getLogWeightsAve, 110
getLogWeightsVar, 110
getScore, 110
operator=, 111
QMCObjectiveFunctionRe-
sult, 109
QMCOptimizationAlgorithm
~QMCOptimization-
Algorithm, 111
QMCOptimizationAlgorithm, 111
optimize, 112
QMCOptimizationFactory, 112
optimizationAlgorithmFac-
tory, 113
QMCPolynomial, 113
evaluate, 115
getCoefficient, 115
getFirstDerivativeValue, 115
getFunctionValue, 116
getNumberCoefficients, 116
getRoots, 116
getSecondDerivativeValue, 117
hasNonNegativeZeroes, 117
initialize, 117
QMCPolynomial, 113, 114
QMCPotential_Energy, 118
evaluate, 118
getEnergy, 118
initialize, 119
operator=, 119
QMCPotential_Energy, 118
QMCproperties, 119
acceptanceProbability, 120
distanceMovedAccepted, 120
distanceMovedTrial, 120
energy, 120
kineticEnergy, 120
logWeights, 120
MPI_REDUCE, 120
MPI_TYPE, 120
operator+, 119
operator<<, 121
potentialEnergy, 120
QMCproperties, 119
readXML, 121
toXML, 121
zeroOut, 119

QMCproperty, 121
getAverage, 122
getNumberSamples, 122
getSeriallyCorrelated-StandardDeviation, 122
getSeriallyCorrelatedVariance, 122
getStandardDeviation, 122
getVariance, 122
MPI_REDUCE, 123
MPI_TYPE, 123
newSample, 123
operator+, 122
operator<<, 123
QMCproperty, 122
readXML, 123
toXML, 124
zeroOut, 122
QMCReadAndEvaluateConfigs
QMCReadAndEvaluate-Configs, 124, 125
QMCReadAndEvaluateConfigs, 124
initialize, 125
QMCReadAndEvaluateCon-figs, 125
rootCalculateProperties, 125
workerCalculateProperties, 126
QMCrun, 126
getNumberOfWalkers, 127
getProperties, 127
getWeights, 128
initialize, 128
QMCrun, 126
randomlyInitializeWalkers, 127
readXML, 128
step, 126
toXML, 128
writeCorrelatedSamplingCon-figurations, 129
writeEnergies, 129
zeroOut, 126
QMCSlater, 129
evaluate, 130
getGradPsiRatio, 131
getLaplacianPsiRatio, 131
getPsi, 131
initialize, 131
isSingular, 130
operator=, 132
QMCstatistic, 132
getAverage, 132
getNumberSamples, 132
getStandardDeviation, 132
getVariance, 132
MPI_REDUCE, 133
MPI_TYPE, 133
newSample, 133
operator+, 132
operator<<, 133
QMCstatistic, 132
readXML, 133
toXML, 134
zeroOut, 132
QMCSteepestDescent
getObjectiveFunction, 135
QMCSteepestDescent, 135
QMCSteepestDescent, 134
optimize, 135
QMCSteepestDescent, 135
QMCStopwatches, 136
getCommandPollingStop-watch, 137
getCommunicationSynchro-nizationStopwatch, 137
getGatherPropertiesStop-watch, 138
getInitializationStopwatch, 136
getOptimizationStopwatch, 138
getPropagationStopwatch, 138
getSendCommandStopwatch, 138
getTotalTimeStopwatch, 139
MPI_REDUCE, 137
MPI_TYPE, 137
operator+, 137
operator<<, 137
QMCStopwatches, 136

reset, 136
stop, 136
QMCwalker, 139
~QMCwalker, 139
calculateObservables, 141
getLocalEnergyEstimator, 140
getWeight, 141
initialize, 141
initializeWalkerPosition, 142
isSingular, 142
operator=, 142
propagateWalker, 140
QMCwalker, 139, 141
readXML, 143
setWeight, 143
toXML, 143
writeCorrelatedSamplingConfiguration, 143
QMCWavefunction, 144
AlphaOccupation, 145
BetaOccupation, 145
Coeffs, 147
getNumberAlphaElectrons, 145
getNumberBasisFunctions, 145
getNumberBetaElectrons, 146
getNumberElectrons, 146
getNumberOrbitals, 146
operator<<, 145
operator=, 146
operator>>, 145
QMCWavefunction, 144
read, 146

ran1
mfrandom.h, 173
randomlyInitializeWalkers
QMCrun, 127
read
QMCBasisFunction, 57
QMCBasisFunctionCoefficients, 58
QMCCorrelationFunctionParameters, 68
QMCJastrowParameters, 92
QMCMolecule, 103
QMCWavefunction, 146
readXML
QMCproperties, 121
QMCproperty, 123
QMCrun, 128
QMCstatistic, 133
QMCwalker, 143
real
Complex, 33
removeAttribute
XMLElement, 163
removeChild
XMLElement, 164
reset
QMCStopwatches, 136
Stopwatch, 150
rootCalculateProperties
QMCReadAndEvaluateConfigs, 125
run
QMCManager, 98

setAttribute
XMLElement, 164
setContent
XMLElement, 165
setGridParameters
CubicSplineWithGeometricProgressionGrid, 41
setMessage
Exception, 43
XMLParseException, 166
setName
XMLElement, 165
setParameters
QMCCorrelationFunctionParameters, 68
setParameterVector
QMCJastrowParameters, 93
setParticle1Type
QMCCorrelationFunctionParameters, 65
setParticle2Type
QMCCorrelationFunctionParameters, 65
setWeight

QMCwalker, 143
singleLineWriter
XMLElement, 165
size
Array1D, 16
Array2D, 19
Array3D, 22
Array4D, 26
SortedParameterScorePair-
List, 149
SortedParameterScorePairList
clear, 148
SortedParameterScorePair-
List, 147, 148
SortedParameterScorePairList, 147
add, 148
get, 149
operator=, 149
size, 149
SortedParameterScore-
PairList, 148
squareroot
Complex, 33
start
Stopwatch, 150
step
QMCrun, 126
stepLength
QMCLineSearchStepLength-
SelectionAlgorithm, 96
QMCMikesBracketingStep-
LengthSelector, 100
stop
QMCStopwatches, 136
Stopwatch, 150
Stopwatch, 149
isRunning, 150
MPI_REDUCE, 150
MPI_TYPE, 150
operator+, 150
operator<<, 150
reset, 150
start, 150
stop, 150
Stopwatch, 150
timeMS, 150
toString, 150
StringManipulation, 151
doubleToString, 152
hexstringToInt, 152
intToHexString, 152
intToString, 152
stringToDouble, 153
stringToInt, 153
toAllLower, 153
toAllUpper, 153
toFirstUpperRestLower, 154
toLowerChar, 154
toUpperChar, 154
stringToDouble
StringManipulation, 153
stringToInt
StringManipulation, 153

timeMS
Stopwatch, 150
toAllLower
StringManipulation, 153
toAllUpper
StringManipulation, 153
toFirstUpperRestLower
StringManipulation, 154
toLowerChar
StringManipulation, 154
toString
Stopwatch, 150
toUpperChar
StringManipulation, 154
toXML
CubicSpline, 37
CubicSplineWithGeometric-
ProgressionGrid, 42
QMCproperties, 121
QMCproperty, 124
QMCrun, 128
QMCstatistic, 134
QMCwalker, 143
Type
QMCBasisFunction-
Coefficients, 60

workerCalculateProperties

QMCReadAndEvaluate-
Configs, 126
write
XMLElement, 165
writeCorrelatedSamplingConfiguration
QMCFunctions, 77
QMCwalker, 143
writeCorrelatedSamplingConfigurations
QMCrun, 129
writeEnergies
QMCrun, 129
writeRestart
QMCManager, 97
writeTimingData
QMCManager, 99

XMLElement, 155
addChild, 158
countChildren, 158
getBooleanAttribute, 159
getChildren, 159
getContent, 159
getDoubleAttribute, 160
getIntAttribute, 160, 161
getLineNr, 161
getName, 161
getStringAttribute, 161, 162
operator=, 162
operator==, 162
parse, 163
prettyWriter, 163
removeAttribute, 163
removeChild, 164
setAttribute, 164
setContent, 165
setName, 165
singleLineWriter, 165
write, 165
XMLElement, 157, 158
XMLParseException
getMessage, 166
NO_LINE, 166
setMessage, 166
XMLParseException, 167
XMLParseException, 166
getLineNr, 167
XMLParseException, 167
xyz_powers
QMCBasisFunction-
Coefficients, 60

Z
QMCMolecule, 104
ZeroCorrelationFunction
evaluate, 168
getFirstDerivativeValue, 168
getFunctionValue, 168
getSecondDerivativeValue, 168
isSingular, 168
ZeroCorrelationFunction, 167
initializeParameters, 168
zeroOut
QMCManager, 97
QMCproperties, 119
QMCproperty, 122
QMCrun, 126
QMCstatistic, 132