Density Matrix Quantum Monte Carlo

Thomas William Rogers

Department of Physics

Imperial College London

MSci Project Report

Supervisors: Prof. Matthew Foulkes & Dr. James Spencer

Assessor: Dr. Alex Thom

Acknowledgements

Thank you to Prof. Matthew Foulkes and Dr. James Spencer who thought of the original idea for this project and provided a great deal of help both in the physics and the computing that resulted in the achievement of these results. Many thanks also to my project-partner Nicholas Blunt, with whom the work in this project is equally shared. Finally, I am grateful for discussions on the application of DMQMC in quantum information theory with Dr. Terry Rudolph and of Tom Poole for proof-reading this report and offering some important changes.

This work made use of the Imperial College High Performance Computing facilities.

Abstract

A new quantum Monte Carlo method based on stochastically sampling the thermal density matrix at finite temperatures has been developed. The method is an adaptation of the recently developed full configuration interaction quantum Monte Carlo[?] (FCIQMC) method and it aims to overcome two of FCIQMCs main restrictions by allowing the calculation of any quantum mechanical observable at finite temperatures. During this report the density matrix quantum Monte Carlo (DMQMC) algorithm is formulated by observing an analogy with FCIQMC. It is then tested on the spin-1/2 antiferromagnetic Heisenberg model on a bipartite lattice.

The DMQMC method produces results that agree with the exact finite temperature FCI energy and exact ground-state staggered magnetisation for the 4×4 lattice. For the larger 6×6 lattice the standard DMQMC algorithm fails. At this point an importance sampling method is introduced, yielding results comparable to accurate Green's function quantum Monte Carlo (GFQMC) estimates. The importance sampling method also enabled simulation of the $4 \times 4 \times 4$ lattice, producing results for the ground-state energy that matched those obtained by FCIQMC.

Finally an avenue into quantum information theory is explored and some key results for the entanglement between two qubits on a spin-1/2 antiferromagnetic Heisenberg ring are reproduced. Additionally, both how the entanglement between qubits changes in a uniformly applied magnetic field and with increasing separation between the two qubits are investigated.

List of Figures

Chapter 1

Introduction

The main aim of current electronic structure methods is to calculate the ground-state properties of many-electron systems. Conventional full-configuration interaction (FCI) calculations quickly become unfeasible because the size of the Hilbert space grows exponentially with the number of electrons. Such calculations based on iterative diagonalisation require the storage of at least as many double-precision numbers as twice the size of the Hilbert space[?] .

Quantum Monte Carlo (QMC) methods can provide accurate simulations of many-electron systems via a stochastic sampling of the many-electron wave function. Such methods are favourable if the number of agents or "walkers" conducting the stochastic sampling is a small fraction of the size of the Hilbert space.

Projector methods[?] such as diffusion Monte Carlo (DMC) rely on the ground-state emerging from the imaginary-time Schrödinger equation in the limit of infinite imaginary-time (see Appx. A). However, because DMC does not enforce the constraint that a Fermionic wave function must be anti-symmetric, they suffer from convergence to the incorrect bosonic ground-state in an event known as the "bosonic catastrophe". In such cases the fixed-node approximation is applied[? ?] , where *a priori* knowledge of the wave-function is used to restrict the propagation of walkers so that they evolve towards an approximately antisymmetric solution.

Until recently QMC methods, such as DMC, could only provide numerically exact solutions for systems of many interacting bosons. The "fermion sign problem", described above, prevented an exact solution for systems of many interacting fermions.

However, recently Booth *et al.*[?] introduced a new approach, known as full configuration-interaction QMC (FCIQMC), in which the antisymmetric property of the many-electron wave function is enforced by working in a discrete space of slater determinants (see Appx. A). The key is the improved efficiency of walker annihilation, which ensures convergence to the fermionic ground-state solution[?] .

FCIQMC has two main restrictions. The first is that it proves difficult to calculate the expectation values of operators that do not commute with the Hamiltonian. The second being that it only allows for calculation of expectation values in the ground-state . This project introduces a method that is similar to FCIQMC, but attempts to overcome both of these restrictions. This is done by a stochastic sampling of the many-electron thermal density matrix as a function of inverse-temperature, rather than sampling the many-electron wave function as a function of imaginary-time. This in theory, allows for the calculation of the expectation values of any operator at finite temperatures.

In Ch. **??** the density matrix quantum Monte Carlo (DMQMC) algorithm is developed and an optimised and highly parallel implementation is discussed. In Ch. **??** DMQMC is applied to the spin-1/2 antiferromagnetic Heisenberg model on two and three dimensional bipartite lattices. The results are compared against exact values from the full diagonalisation of the Hamiltonian, for small systems, and accurate FCIQMC or Green's function QMC (GFQMC) estimates for larger systems. Finally, in Ch. **??** a method of calculating a reduced density matrix for a subsystem of the Heisenberg lattice is developed and this leads into applications in quantum information theory.

Note that a system of units with $\hbar = k_B = 1$ is adopted for all equations in this report.

Chapter 2

The DMQMC algorithm

In this chapter the DMQMC algorithm is formulated using an analogy between the imaginary-time Schrödinger equation and the evolution of the many-electron thermal density matrix as a function of inverse-temperature. Several features of FCIQMC are used in the construction of this algorithm in a way that maintains FCI-quality results. Finally, the main features of an optimised and highly parallel implementation of the DMQMC algorithm are discussed.

2.1 The thermal density matrix

In quantum mechanics the density operator[?] is any operator, ρ, that satisfies three properties:

1. It is Hermitian i.e. $\rho = \rho^\dagger$
2. It is a positive, semi-definite operator, i.e. $\langle\psi|\,\rho\,|\psi\rangle \geq 0 \ \ \forall \ \ |\psi\rangle \in \mathcal{H}$
3. It has unit trace i.e. $\mathrm{Tr}\rho = 1$

As ρ is Hermitian, it can be expanded using the spectral theorem,

$$\rho = \sum_{j=1} p_j \, |\psi_j\rangle \, \langle\psi_j| \, , \tag{2.1}$$

where $\{p_j\}$ are the eigenvalues of ρ and are also the classical probabilities that the system is in the vector state $|\psi_j\rangle$. A density operator is useful for describing and calculating the properties of a mixed state. A mixed state is a statistical ensemble of several quantum states and arises in situations where there is classical uncertainty (which is different to quantum uncertainty).

One major benefit of constructing a density matrix is that it allows for very simple calculations of any quantum mechanical observable, O:

$$\begin{aligned} \langle O \rangle &= \sum_i p_i \langle \psi_i | O | \psi_i \rangle \\ &= \sum_i \langle \psi_i | \rho | \psi_i \rangle \langle \psi_i | O | \psi_i \rangle \\ &= \sum_i \langle \psi_i | \rho O | \psi_i \rangle \\ &= \sum_{i,j} \rho_{ij} O_{ji}. \end{aligned} \tag{2.2}$$

In the last line the matrix representations of the operators ρ and O have been adopted.

This hints that the formulation of a QMC method that can stochastically sample the many-electron density operator, could provide a simple calculation of the expectation value of any quantum mechanical observable. There is no restriction on the operator, O, other than the requirement that it is expressed in the same basis as the density matrix. Eq. **??** is the blue-print used for calculating all estimators of quantum mechanical observables in this section.

An important mixed state is the thermal state[?] , which describes a quantum statistical system at temperature T. In this case, the classical probability distribution is described by the the Boltzmann distribution and the thermal density operator is

$$\rho = \frac{e^{-\beta H}}{Z(\beta)}, \tag{2.3}$$

where H is the Hamiltonian operator, $\beta = 1/T$ is the inverse-temperature in natural units and $Z(\beta) = \mathrm{Tr}(e^{-\beta H})$ is the partition function of the system.

Applying the spectral theorem to the Hamiltonian, Eq. **??** becomes

$$\rho = \sum_j \frac{e^{-\beta E_j}}{Z(\beta)} |E_j\rangle \langle E_j|. \tag{2.4}$$

This reflects the classical uncertainty induced by thermal fluctuations of the energy state that the system is in. The canonical partition function can be expressed as

$$Z(\beta) = \sum_{j=1} e^{-\beta E_j}. \tag{2.5}$$

From now on the convention

$$\rho = e^{-\beta H} \implies \langle O \rangle = \frac{\sum_{i,j} \rho_{ij} O_{ji}}{\sum_i \rho_{ii}} \tag{2.6}$$

is adopted to keep notation simple.

2.2 An analogy with FCIQMC

Using the convention from Eq. **??** it is evident that, in its matrix representation, ρ obeys the symmetrised Bloch equation**?** [i]

$$\frac{\partial \rho_{ij}}{\partial \beta} = -\frac{1}{2} \sum_k H_{ik} \rho_{kj} + \rho_{ik} H_{kj}, \tag{2.7}$$

with the initial condition

$$\boldsymbol{\rho} \left(\beta = 0 \right) = \boldsymbol{I}, \tag{2.8}$$

where $\boldsymbol{I}$ is the identity matrix with the same dimensions as the thermal density matrix $\boldsymbol{\rho}$.

Now as the partition function is simply Tr$\boldsymbol{\rho}$, Eq. **??** provides all of the information required to calculate the normalised thermal density matrix. The Bloch equation (Eq. **??**) is analogous to the imaginary-time Schrödinger equation. The columns of the many-electron density matrix evolving as a function of inverse-temperature, rather than a many-electron wave function evolving in imaginary-time,

$$\frac{\partial \psi_i}{\partial \tau} = -\sum_j H_{ij} \psi_j, \tag{2.9}$$

[i]The symmetrised version of the Bloch equation is used so that there is spawning in two directions rather than one, which makes more sense intuitively.

where ψ_i is the vector representation of the many-electron wave function in a discrete space (e.g. slater determinant space, see Appx. A) and $\tau = it$ is the imaginary-time.

In the true spirit of this analogy, the ground-state many-electron density matrix is projected out in the large β (or zero temperature) limit,

$$\rho(\beta \to \infty) = \lim_{\beta \to \infty} \sum_i e^{-\beta E_i} |E_i\rangle \langle E_i| \approx e^{-\beta E_0} |E_0\rangle \langle E_0| , \tag{2.10}$$

where $|E_0\rangle$ is the ground-state or lowest eigenvector of H and E_0 is the ground-state energy eigenvalue.

This is similar to in FCIQMC, where the ground-state eigenvector, $\boldsymbol{\psi}_0$, of the many-electron Hamiltonian matrix, $\boldsymbol{H}$, is projected out in the long imaginary-time limit? ,

$$\begin{aligned} \boldsymbol{\psi}(\tau \to \infty) &= \lim_{\tau \to \infty} e^{-\boldsymbol{H}\tau} \sum_\alpha v_\alpha(0) \boldsymbol{\psi}_\alpha \\ &= \lim_{\tau \to \infty} \sum_\alpha v_\alpha(0) e^{-E_\alpha \tau} \boldsymbol{\psi}_\alpha \\ &\approx v_0(0) e^{-E_0 \tau} \boldsymbol{\psi}_0, \end{aligned} \tag{2.11}$$

where the initial vector has been expanded in terms of the complete orthonormal set of eigenvectors, $\{\boldsymbol{\psi}_\alpha\}$ of $\boldsymbol{H}$ i.e. $\boldsymbol{\psi}(0) = \sum_\alpha v_\alpha(0) \boldsymbol{\psi}_\alpha$.

So a QMC method based on solving Eq. ?? would evolve towards the ground-state of the many-electron system in a similar way to in FCIQMC, however it would also allow for the density matrix to be sampled at finite temperatures.

The difficult factor of $e^{-\beta E_0}$ in Eq. ?? can be removed by setting $E_0 = 0$. In reality E_0 is usually unknown and so instead one can solve,

$$\frac{\partial \rho_{ij}}{\partial \beta} = \frac{1}{2} \sum_k T_{ik} \rho_{kj} + \rho_{ik} T_{kj}, \tag{2.12}$$

where $T_{ik} = -(H_{ik} - S\delta_{ik})$ is the "update matrix" and S is the energy shift, which can be adjusted carefully to keep normalisation approximately constant? .

2.3 Population dynamics

So drawing upon the analogy between Eq. **??** and Eq. **??**, FCIQMC can be adapted so that rather than stochastically sampling the many-electron wave function, it performs a stochastic sampling of the many-electron thermal density matrix. As in FCIQMC, one considers a collection of markers or "psips"[i] that are distributed over the elements of the density matrix. Now each psip still has a "charge" $q = \pm 1$, but their location is now labelled by two indices (i, j).

The fact that the solution to Eq. **??** is a matrix implies that the psips can now spawn from two ends. One end can spawn in one direction and the other end can spawn in a direction that is perpendicular. The algorithm for the DMQMC can be summarised as follows:

At each inverse-temperature step $\Delta\beta$, loop over the population of psips and allow each end, A and B, of the psip to spawn a "child" at other locations according to the following set of rules:

1. The probability that end A of a psip at (i, j) spawning a child at (k, j) is $\frac{1}{2}\left|T_{ki}\right|\Delta\beta$

2. The probability that end B of a psip at (i, j) spawning a child at (i, k) is $\frac{1}{2}\left|T_{kj}\right|\Delta\beta$

3. If end A of a parent psip with charge q_{parent} at location (i, j) spawns a child at (k, j), the charge of the child is given by $q_{child|A} = \text{sign}(T_{ki})q_{parent}$

4. If end B of a parent psip with charge q_{parent} at location (i, j) spawns a child at (i, k), the charge of the child is given by $q_{child|B} = \text{sign}(T_{kj})q_{parent}$

At the end of each inverse-temperature step, after each psip has spawned as many times as it can, pairs of walkers of opposite charge at the same location annihilate each other and are removed from the simulation.

[i] "psips" stands for "psi particles"[?] but there was a temptation to call them "rhips"

2.4 First order finite-difference approximation

The motivation for this algorithm is that the dynamics satisfy a first-order Euler finite-difference approximation of Eq. **??**. The expected charge, $\bar{q}_{ij}\left(\beta+\Delta\beta\right)$, at location (i,j) at inverse-temperature $\beta+\Delta\beta$ is related to the expected charges $\bar{q}_{kj}\left(\beta\right)$ and $\bar{q}_{ik}\left(\beta\right)$ at inverse-temperature β by

$$\bar{q}_{ij}\left(\beta+\Delta\beta\right)=\bar{q}_{ij}\left(\beta\right)+\frac{1}{2}\sum_{k}\left(T_{ik}\bar{q}_{kj}+\bar{q}_{ik}T_{kj}\right)\Delta\beta. \tag{2.13}$$

The first term on the right-hand side describes the total charge of the walkers that were at (i,j) at inverse-temperature β. The second term describes the spawning of walkers onto (i,j) over the inverse-temperature interval $\Delta\beta$. As this is a first-order Euler finite approximation of Eq. **??**, the distribution of charges is proportional to the density matrix.

The stability[?] of the first order finite-difference approximation in Eq. **??** can be analysed by making a linear perturbation to the density matrix i.e $\boldsymbol{\rho}\rightarrow\boldsymbol{\rho}+\boldsymbol{\epsilon}$. This implies that for the n^{th} β-step,

$$\boldsymbol{\epsilon}_{n+1}=(\boldsymbol{I}+\Delta\beta\boldsymbol{T})\boldsymbol{\epsilon}_{n}. \tag{2.14}$$

Now this is stable if $|\lambda^{i}|\leq 1$, where $\{\lambda^{i}\}$ are the eigenvalues of $(\boldsymbol{I}+\Delta\beta\boldsymbol{T})$. So this leads to the stability condition, $\Delta\beta\leq 2/(E_{max}-S)$, where E_{max} is the maximum energy eigenvalue of the Hamiltonian. The shift, S, can be assumed to be approximately equal to the energy of the system and is most negative at the ground-state. So in the worst case $\Delta\beta$ is limited to

$$0<\Delta\beta\leq\frac{2}{E_{max}-E_{0}}. \tag{2.15}$$

Furthermore, by considering the taylor expansion of the density matrix at inverse-temperature $\beta+\Delta\beta$, it is evident that the local error for a density matrix element is $\mathcal{O}(\Delta\beta^{2})$. Note that statistical errors will also be accrued because Eq. **??** is integrated stochastically. This analysis of the accuracy and the stability was taken into consideration when selecting the $\Delta\beta$ in order to avoid instability or unnecessarily large errors. For all simulations that follow in this report $\Delta\beta=0.1$

was chosen for stability and also as a compromise between accuracy and efficiency.

2.5 Shift update algorithm

The algorithm for updating the shift S was first used by Umrigar[?] in DMC but was also adopted by the inventors of FCIQMC[?]. In this, S is adjusted according to

$$S(\beta + A\Delta\beta) = S(\beta) + \frac{\zeta}{A\Delta\beta} \ln\left(\frac{N_p(\beta + A\Delta\beta)}{N_p(\beta)}\right), \tag{2.16}$$

where A is the number of β-steps between shift updates, ζ is a shift damping parameter and $N_p(\beta)$ is the total number of psips at the inverse-temperature β. During simulations, ζ is chosen carefully to prevent large fluctuations in S. It should also be noted that by averaging the shift over many iterations, it can be used as an estimator for the ground-state energy in a similar way to in FCIQMC. However, this approach was not adopted during this project.

2.6 The spin-1/2 Heisenberg model

In 1926, shortly after the birth of quantum mechanics, Werner Heisenberg[?] and Paul Dirac[?] independently recognised that the new theory could be used to describe the mysterious properties of ferromagnetism. They realised that quantum mechanics implied the existence of an effective interaction, the exchange interaction, between electron spins that is caused by the combined effects of the Pauli exclusion principle and the Coulomb repulsion between them[?].

The Hamiltonian that describes the exchange interactions between a set of quantum mechanical spins (with periodic boundary conditions) is

$$H = -J \sum_{\langle i,j \rangle} S_i S_j + h \sum_i S_i^z, \tag{2.17}$$

where $S = \{S^x, S^y, S^z\} = \frac{1}{2}\{\sigma^x, \sigma^y, \sigma^z\}$ are the quantum spin operators expressed in terms of the Pauli spin matrices, h is a uniform magnetic field in the z-direction and J is the coupling constant. Here it has been assumed that nearest-neighbour interactions are dominant, hence the summation is only over nearest-neighbours.

Furthermore, it is assumed that the system is isotropic so that J is the same between any two neighbouring spins.

The coupling constant J plays a big role in determining the physical properties of the system. If there is no externally applied magnetic field ($h = 0$) then for $J > 0$ the ground-state is always ferromagnetic, where neighbouring spins tend to align themselves in the same direction. On the other hand, for $J < 0$, the ground-state is antiferromagnetic, where neighbouring spins point in opposite directions.

For the Heisenberg model with N spins, H acts on a 2^N dimensional Hilbert space, which is spanned by the orthogonal basis vectors, or spin configurations, $|s_1 s_2 \dots s_N\rangle$. Here, s_n can represent either a spin in the positive z-direction (spin up), $s_n = \uparrow$, or a spin in the negative z-direction (spin down), $s_n = \downarrow$. On implementing DMQMC for the Heisenberg model, bitstrings can be used to represent a given spin-configuration of the system. For example, the configuration for a system of N spins can be represented by a string of N binary digits, with a 1 representing a spin up and 0 a spin down.

The Heisenberg Hamiltonian, Eq. **??** can also be expressed in terms of the spin flip operators $S^{\pm} = S^x \pm iS^y$,

$$H = -J \sum_{\langle i,j \rangle} \left[\frac{1}{2} \left(S_i^+ S_j^- + S_i^- S_j^+ \right) + S_i^z S_j^z \right] + h \sum_i S_i^z. \tag{2.18}$$

This gives a more intuitive feel for the action of the Hamiltonian on a given configuration in the Heisenberg model. Every non-zero off-diagonal element of the Hamiltonian flips a pair of neighbouring spins from $\uparrow\downarrow$ to $\downarrow\uparrow$ and vice versa. So a matrix element, H_{ij}, has a non-zero value of $-J/2$, if the configurations $|i\rangle$ and $|j\rangle$ differ by a spin-flip of neighbouring anti-aligned spins. In other words, the two configurations are a single excitation apart.

For the diagonal elements, H_{ii}, neighbouring spins in the configuration, $|i\rangle$, that are aligned ($\uparrow\uparrow$ or $\downarrow\downarrow$) give a contribution of $-J/4$. Neighbouring spins that are anti-aligned ($\downarrow\uparrow$ or $\uparrow\downarrow$) give a contribution of $J/4$. For a single total spin (M_S) subspace[i], the magnetic field only affects the diagonal elements of the

[i]The current DMQMC implementation only considers a single M_S subspace during a simulation. This is because the code is built upon an FCIQMC code, which only requires simulations

Hamiltonian,

$$\boldsymbol{H}(h) = \boldsymbol{H}(h=0) + \frac{hM_S}{2}\boldsymbol{I}. \tag{2.19}$$

2.7 Implementation

The DMQMC algorithm was implemented in `FORTRAN 90` (see Appdx. C) and was an adaptation of James Spencer's optimised and highly parallel FCIQMC code. As this code adopted the same spawning and death/clone steps as the original FCIQMC method[?] , the DMQMC algorithm has been implemented in a slightly different way to in Sec. ??. To be specific, the effect of the shift is simulated by a death/clone step rather than by changing the spawning probabilities.

In a simulation, the statistics for each β-step are collected from a number of different "β-loops", each of which start with a different random number generator (RNG) seed. In each β-loop the main steps are:

1. Initiate the psip population on the diagonal of the density matrix
2. Update DMQMC estimators and initialise shift
3. Move to next β-step
4. Loop over all psips and attempt to spawn progeny onto two other connected density matrix elements
5. At the same time, attempt to kill or clone each parent psip[i]
6. Annihilate psips with opposite charges that occupy the same density matrix element
7. Update DMQMC estimators and shift (if necessary)
8. Repeat steps 3-7 until the user-defined maximum β has been reached

in a single M_S subspace, the one that contains the ground-state. Unfortunately, there was not enough time during this project to combine all M_S subspaces.

[i]The combination of steps 4 and 5 is equivalent to the spawning step described in Sec. ?? using the update matrix T. This was how the FCIQMC algorithm was originally proposed by Booth *et al.*[?] .

In this implementation the position of a psip within the density matrix is represented by two bitstrings, with one corresponding to row position and the other to column position. The bitstring can also be seen as a representation of the configuration of spins in the Heisenberg model (Sec. **??**). The implementation of the main steps in the DMQMC algorithm is described in more detail below.

2.7.1 Initial condition

The initial distribution of psips must be along the diagonal of the density matrix as this corresponds to a stochastic sampling of the identity matrix, which is the initial condition for the DMQMC algorithm. In order to create a psip on the diagonal, all spins in both bitstrings, that represent a psips position, are initially set down. Then spins are chosen at random and flipped up (in both bitstrings) until the configuration corresponds to the correct total spin value for the current simulation. As both bitstrings are equal, this must correspond to a diagonal element of the density matrix. The process is repeated until the correct number of initial pips have been spawned.

2.7.2 Spawning

The spawning is performed in a way that is very similar to in FCIQMC, but now a psip can attempt to spawn from both of it's "spawning ends", labelled A and B. One spawning end attempts to spawn onto a density matrix element in the same row, whilst the other end attempts to spawn on a density matrix element in the same column. In order to achieve this for the Heisenberg model, a single random excitation[i], is made to the bitstring that corresponds to the current spawning end.

The probability of spawning a psip on this new density matrix element, ρ_{ik},

[i]For the Heisenberg model the spawning corresponding Hamiltonian element is zero if the configurations differ by more than one excitation, so the spawning probability would be zero. An excitation is defined as flipping two adjacent and anti-parallel spins so that the total spin quantum number is conserved

from spawning end A is then given by

$$p_s(i,k|i,j) = \frac{1}{2}\frac{\Delta\beta |H_{jk}|}{P_{gen}(k|j)}, \tag{2.20}$$

where P_{gen} is the probability of generating the single excitation from $|j\rangle$ to $|k\rangle$. Here, the efficiency of the algorithm has been improved by only attempting to spawn to a single connected density matrix element. Hence, the re-weighting of the spawning probability by P_{gen}.

In a similar way, the probability of spawning a psip on a new density matrix element ρ_{kj} from spawning end B is given by

$$p_s(k,j|i,j) = \frac{1}{2}\frac{\Delta\beta |H_{ik}|}{P_{gen}(k|i)}. \tag{2.21}$$

In general, $\lfloor p_s \rfloor$ psips are always spawned and a further psip is spawned if $p_s - \lfloor p_s \rfloor > R$, where $0 \leq R \leq 1$ is a uniformly distributed pseudorandom number[i].

2.7.3 Diagonal death/cloning step

The diagonal death/cloning step was performed in a similar way to the original FCIQMC code, but with a contribution from both spawning ends. For either of the spawning ends labelled by $|i\rangle$, the "probability" that the parent psip is killed by that spawning end is

$$p_d(i) = \frac{1}{2}\Delta\beta(H_{ii} - S). \tag{2.22}$$

In the current DMQMC implementation the probabilities of each spawning end are combined for simplicity. So for a density matrix element ρ_{ij}, the probability of a parent psip being killed is

$$p_d(i,j) = p_d(i) + p_d(j) = \left(\frac{H_{ii} + H_{jj}}{2} - S\right)\Delta\beta. \tag{2.23}$$

For $p_d \geq 0$, $\lfloor p_d \rfloor$ psips are always killed and a further psip is killed if $p_d - \lfloor p_d \rfloor > R$,

[i]Random numbers were generated using the `dSFMT` double-precision pseudorandom number generator[?] .

with $0 \leq R \leq 1$ again being a uniformly distributed pseudorandom number. Moreover, in the rare event that $p_d < 0$, then $\lfloor |p_d| \rfloor$ psips are always cloned and a further psip is cloned if $|p_d| - \lfloor |p_d| \rfloor > R$.

2.7.4 Energy estimator

The calculation of the energy estimator is simplified if the Hamiltonian and the thermal density matrix are symmetric, which is the case for the Heisenberg Hamiltonian in Eq. ??. The expression for the estimator can be written

$$\langle H \rangle = \frac{\sum_{i,j} \rho_{ij} H_{ij}}{\sum_i \rho_{ii}}. \tag{2.24}$$

Now the denominator is trivial to calculate – it is the sum of the psip charges, q_{ii}, on the diagonal of the density matrix.

The numerator requires slightly more thought. As highlighted in Sec. ?? off-diagonal elements of the Heisenberg Hamiltonian, H_{ij} are only non-zero if the configurations $|i\rangle$ and $|j\rangle$ differ by a single excitation. In this case the density matrix element with total psip charge q_{ij}, contributes $-Jq_{ij}/2$ to the numerator of the energy estimator.

The two bitstrings that label the density matrix element can be compared, using the a bit-wise `XOR` operation[i]. Half of the sum of the elements of the resultant bitstring is equal to the number of excitations. For example consider the two bitstrings 110100 and 011001. They differ by two excitations - to make them equal the two central spins and also the spins on both ends need to be flipped. In terms of the `XOR` operator this gives

$$110100 \ \texttt{.XOR.} \ 011001 = 101101, \tag{2.25}$$

from which it is clear that the two bitstrings differ by two excitations.

The same bit-wise `XOR` operation can also be used to check whether an element is on the diagonal of the density matrix. In this case the contribution to the

[i]In more than one dimension, one must also test whether the two spins that are flipped between the two bitstrings are neighbours. This can be done by storing a list of neighbouring spins.

numerator of Eq. **??** is $q_{ii}H_{ii}$ where H_{ii} can be determined as described at the end of Sec. **??**

2.7.5 Squared staggered magnetisation estimator

Staggered magnetisation is an order parameter of the antiferromagnetic Heisenberg model described in Sec. **??**. An order parameter is a measure of the degree of order in a system and ranges between zero for total disorder and some maximum non-zero value for complete order. This measure usually changes during a phase transition. For a bipartite antiferromagnetic Heisenberg lattice that can be partitioned into two sublattices a and b such that neighbouring spins are on separate sublattices, the staggered magnetisation is given by[?]

$$m^\dagger = \sqrt{\langle (M^\dagger)^2 \rangle}, \tag{2.26}$$

$$M^\dagger = \frac{1}{N}\sum_{i=1}^{N} \epsilon_i S_i \tag{2.27}$$

$$\langle (M^\dagger)^2 \rangle = \frac{\sum_{i,j} (M^\dagger)^2_{ij}\rho_{ji}}{\sum_i \rho_{ii}} \tag{2.28}$$

Where the co-efficient ϵ_i is $+1$ if the site i is on one sublattice and is -1 if site i is on the other sublattice. For the classical antiferromagnetic ground-state, or the Néel state, $m^\dagger = \frac{1}{2}$.[i]

It is easy to calculate the estimator for the squared staggered magnetisation if $(M^\dagger)^2$ is expressed in the form:

$$(M^\dagger)^2 = \frac{1}{N^2}\sum_{i,j} \epsilon_i\epsilon_j (S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z). \tag{2.29}$$

Now for diagonal elements of $(M^\dagger)^2$, the x and y spin operator terms only contribute for $i = j$. If this is the case then both terms contribute $1/4N$ to the diagonal element. The contribution from the z spin operator term can be deduced from the total number of up spins, $N_\uparrow$, and the total number of up spins

[i]An interesting point is the difference in order for a quantum ground-state and the classical Néel state. When no external magnetic field is applied, the order of the quantum ground-state is zero[?] . This is as a result of intrinsic zero-point quantum spin fluctuations in the ground-state[?]

on sublattice a, $N_\uparrow^a$. Overall the diagonal elements are given by

$$(M^\dagger)^2_{ii} = \frac{(2N_\uparrow^{a,i} - N_\uparrow)^2 + \frac{N}{4}}{N^2}, \tag{2.30}$$

where $N_\uparrow^{a,i}$ is the number of up spins on sublattice a for configuration $|i\rangle$. In the current implementation, this calculation of $N_\uparrow^{a,i}$ is made highly efficient by performing a bit-wise `AND` operation on the bitstring representation of $|i\rangle$ with a boolean mask for subsystem a. The sum of the elements of the resultant bitstring is equal to $N_\uparrow^{a,i}$.

For off-diagonal elements, $(M^\dagger)^2_{ij}$, the contributions from the x and y spin operator terms are non-zero if $|i\rangle$ and $|j\rangle$ differ by one (not necessarily nearest-neighbour) excitation. If this is the case then $(M^\dagger)^2_{ij} = \pm 1/N^2$ depending upon whether the two spins that have been flipped are on the same lattice. Whether they are on the same lattice can be checked very quickly using the boolean mask for system a.

So in a similar way to the energy estimator, a density matrix element only contributes to the numerator of Eq. **??** if there are zero or one excitations between the two bitstrings that label the element. Then the contribution is given by $q_{ij}(M^\dagger)^2_{ij}$, where $(M^\dagger)^2_{ij}$ is found using the above prescription.

2.7.6 Heat capacity estimators

The fundamental equation of thermodynamics for a system of temperature T that is subject to a uniform magnetic field, h, in the z-direction can be written,

$$dE = dQ + dW = TdS - M_z dh, \tag{2.31}$$

where dE is the infinitesimal change in internal energy, dS is the infinitesimal change in entropy for a small flow of heat dQ, and $M_z dh$ is the work done on the system's total magnetic dipole moment $\vec{M}$ by the magnetic field.

The heat capacity, C, can be defined by $dQ = CdT = TdS$. Assuming that the external magnetic field is constant then

$$dE = C_h dT, \tag{2.32}$$

where C_h is now the heat capacity for a constant magnetic field.

Expressing this in terms of β one obtains

$$C_h = -\beta^2 \frac{dE}{d\beta}. \tag{2.33}$$

At this point there is already a route for calculating the heat capacity in a constant magnetic field. DMQMC can easily sample the internal energy, so it is possible to simply differentiate this estimator in order to obtain the finite-temperature spectrum for C_h,

$$\langle C_h \rangle = -\beta^2 \frac{d\langle H \rangle}{d\beta}. \tag{2.34}$$

In practice the data contains a lot of statistical noise, which makes it difficult to accurately approximate the gradient function of $\langle H \rangle$. However, a smoothing spline fit can be approximated for $\langle H \rangle$ in order to smooth out any statistical fluctuations (Appdx. B). From this the derivative in **??** can also be approximated.

Another, perhaps more elegant, method would be to sample C_h directly during the DMQMC simulation. If $\langle H \rangle$ is expanded into an orthonormal basis of energy eigenstates $\{|E_i\rangle\}$,

$$\langle H \rangle = \frac{\mathrm{Tr}(\rho H)}{\mathrm{Tr}\rho} = \frac{\sum_i \langle E_i| \rho H |E_i\rangle}{\sum_i \langle E_i| \rho |E_i\rangle} = \frac{\sum_i \langle E_i| e^{-\beta E_i} E_i |E_i\rangle}{\sum_i \langle E_i| e^{-\beta E_i} |E_i\rangle}, \tag{2.35}$$

it is simple to calculate a value of C_h using only the density matrix and the hamiltonian;

$$\begin{aligned} C_h &= \beta^2 \frac{d}{d\beta}\left[\frac{\sum_i \langle E_i| e^{-\beta E_i} E_i |E_i\rangle}{\sum_i \langle E_i| e^{-\beta E_i} |E_i\rangle}\right] \\ &= \beta^2 \frac{\mathrm{Tr}(\rho H^2)\mathrm{Tr}(\rho) - \mathrm{Tr}(\rho H)^2}{\mathrm{Tr}(\rho)^2} \\ &= \beta^2(\langle H^2 \rangle - \langle H \rangle^2). \end{aligned} \tag{2.36}$$

Where the differentiation is performed using the quotient rule. This is in a form that can be sampled directly during the DMQMC simulation.

It is not immediately obvious whether direct stochastic sampling or a spline fit derivative is best for obtaining $\langle C_h \rangle$, so both will be tested in Sec. **??**.

2.7.7 Other estimators

During the project estimators for spin-spin correlation and squared energy (required for the directly sampled heat capacity estimator) were also implemented. However, there is not enough room in this report to include the details or results for these estimators.

2.7.8 Statistical error calculations

A general DMQMC estimator can be written in the form,

$$O = \frac{x}{y}, \tag{2.37}$$

where x and y are separately averaged over all β-loops. As x and y are not necessarily uncorrelated, the standard error for the estimator was found by propagating the standard errors in the numerator and denominator using

$$\frac{\epsilon_O}{O} = \sqrt{\left(\frac{\epsilon_x}{x}\right)^2 + \left(\frac{\epsilon_y}{y}\right)^2 - 2\frac{\mathrm{Cov}_{xy}}{xy}}, \tag{2.38}$$

where $\{\epsilon_i\}$ are the standard errors and Cov_{xy} is the covariance between x and y. This error analysis was performed in a post-processing step using a script written in `PYTHON` in order to avoid unnecessary calculations during the simulation.

Additionally, it is possible to obtain estimators for ground-state properties. One can average a finite-temperature estimator over consecutive β-steps once the simulation has settled into the ground-state. The Flyvbjerg and Peterson blocking analysis[?] can be used to take into account the correlations between consecutive β-steps when calculating the statistical error. During this project James Spencer's `PYTHON` implementation of this blocking analysis (originally applied to FCIQMC calculations) was employed.

2.8 Summary

In this chapter the DMQMC algorithm has been developed from an analogy with the imaginary-time Schrödinger equation and FCIQMC. In theory it should overcome the main two restrictions of FCIQMC, but may be restrained by the fact that the psips now live in a space that is the size of the Hilbert space of spin configurations squared. Additionally the implementation of the DMQMC algorithm was discussed, along with its application to the Heisenberg model.

Chapter 3

Testing and Results

In this chapter the implementation of DMQMC is tested on some small antiferromagnetic bipartite Heisenberg lattices. The benefits of testing on such systems is that they are sign-problem free? and the ground-state properties have been the subject of extensive study in the literature. All results in this section were simulated in the $M_S = 0$ total spin subspace with an inverse-temperature step of $\Delta\beta J = 0.1$ and in the absence of an external magnetic field ($h = 0$).

3.1 The 4×4 antiferromagnetic Heisenberg Lattice

DMQMC was first tested on the well-studied 4×4 antiferromagnetic Heisenberg lattice. The Hilbert space contains only 12,870 configurations in the $M_S = 0$ subspace, which allows calculation of the exact ground-state properties using direct methods such as FCI.

3.1.1 Energy

The DMQMC finite-temperature energy estimator for the 4×4 lattice is shown in Fig. **??** and Fig. **??**. DMQMC mainly shows a good agreement with the exact FCI finite-temperature energy, which was obtained via a direct Lanczos diagonalisation? . However, there is a slight bias for $0.25 - 1$ βJ. It was later later discovered that this was a problem with accuracy and that the bias disappears if the inverse-temperature step is reduced to $\Delta\beta = 0.01$.

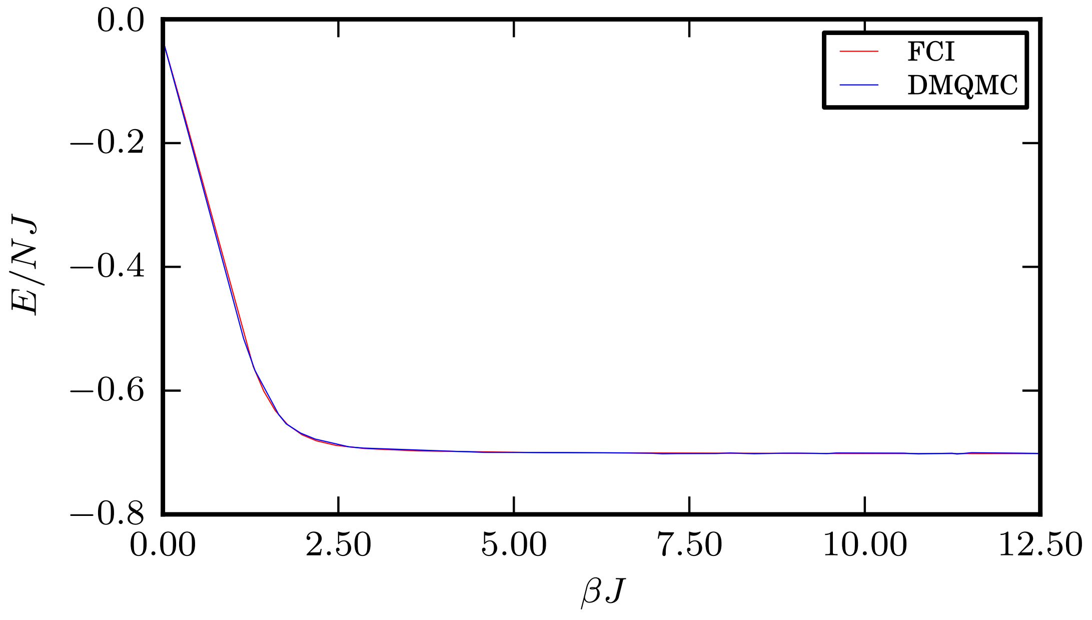


Figure 3.1: The DMQMC finite-temperature energy estimator (blue) and the exact FCI energy (red) for the 4×4 antiferromagnetic Heisenberg model. The DMQMC simulation ran with an initial population of 10^5 psips and for 10^3 β-loops.

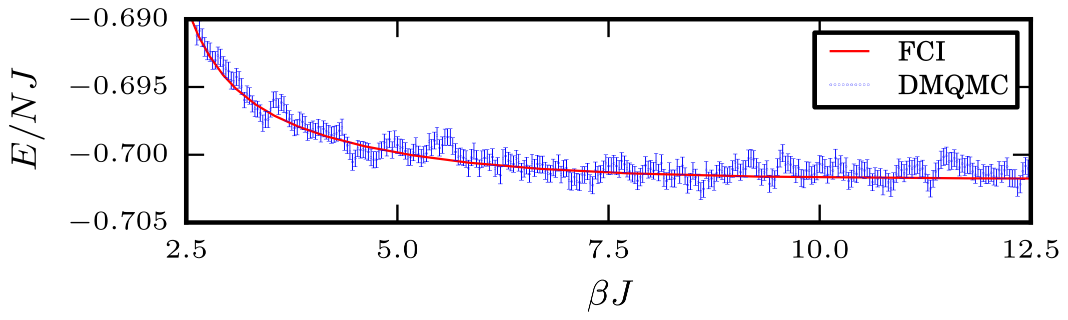


Figure 3.2: The DMQMC finite-temperature energy estimator with statistical errors (blue) and the exact FCI energy (red) for the 4×4 antiferromagnetic Heisenberg model.

A blocking analysis over a single β-loop yields a ground-state energy estimate of $E_0 = -0.7021(3)$, which is within error of the FCI ground-state energy[?] of $E_0 = -0.7018$.

However, the number of psips used was larger than the size of the Hilbert space[i]. So whilst the DMQMC results are reasonably accurate, the finite-temperature properties can be calculated with far less effort using FCI. For example, the DMQMC simulation running on 64 cores (2.8 Ghz) took approximately 3 hours to complete, whereas the FCI calculation took approximately 30 minutes on 2 cores (1.86 GHz). This could prove to be a problem with larger systems where even more psips are required.

3.1.2 Staggered magnetisation

The DMQMC finite-temperature squared staggered magnetisation estimator was also tested on the 4×4 antiferromagnetic Heisenberg model and the results are presented in Fig. **??** and Fig. **??**. A blocking analysis over a single β-loop yields a ground-state squared staggered magnetisation estimate of $\langle (M^\dagger)^2 \rangle_0 = 0.2762(5)$, which agrees (within error) with the exact result of $\langle (M^\dagger)^2 \rangle_0 = 0.2765$ obtained by Dagotto and Moreo[?] using a modified Lanczos method.

[i]But considerably less than the 1.7×10^8 elements of the density matrix

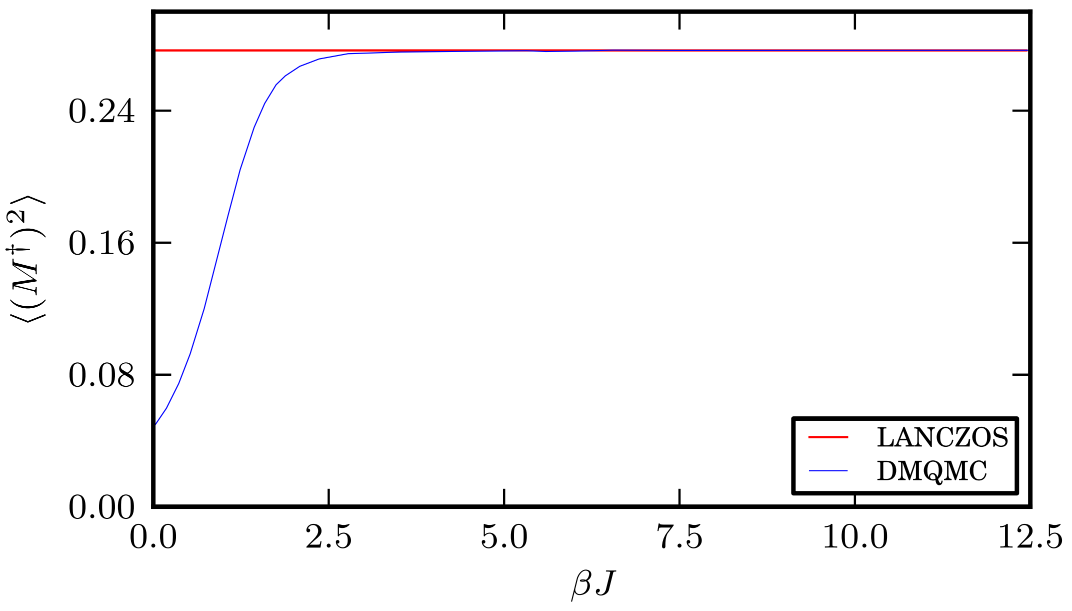


Figure 3.3: The DMQMC finite-temperature squared staggered magnetisation estimator (blue) and the exact modified Lanczos ground-state squared staggered magnetisation (red) for the 4×4 antiferromagnetic Heisenberg model. The DMQMC simulation ran with an initial population of 10^5 psips and for 10^3 β-loops.

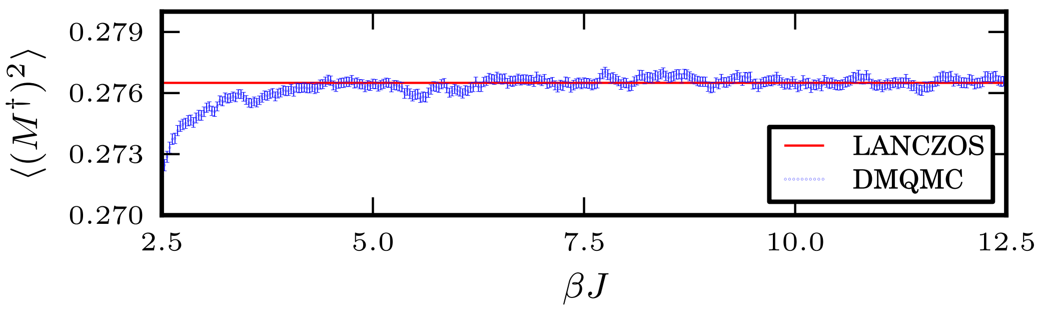


Figure 3.4: The DMQMC finite-temperature squared staggered magnetisation estimator with statistical errors (blue) and the exact modified Lanczos ground-state squared staggered magnetisation (red) for the 4×4 antiferromagnetic Heisenberg model.

3.1.3 Heat capacity

Two methods for calculating the heat capacity in a constant magnetic field were introduced in Sec. ??. One method was to calculate the derivative in Eq. ?? using a cubic spline fit of the finite-temperature energy estimator and the other was to directly sample the heat capacity during the DMQMC simulation. Both were tested on the 4×4 antiferromagnetic Heisenberg lattice with no magnetic field and are shown in Fig. ??.

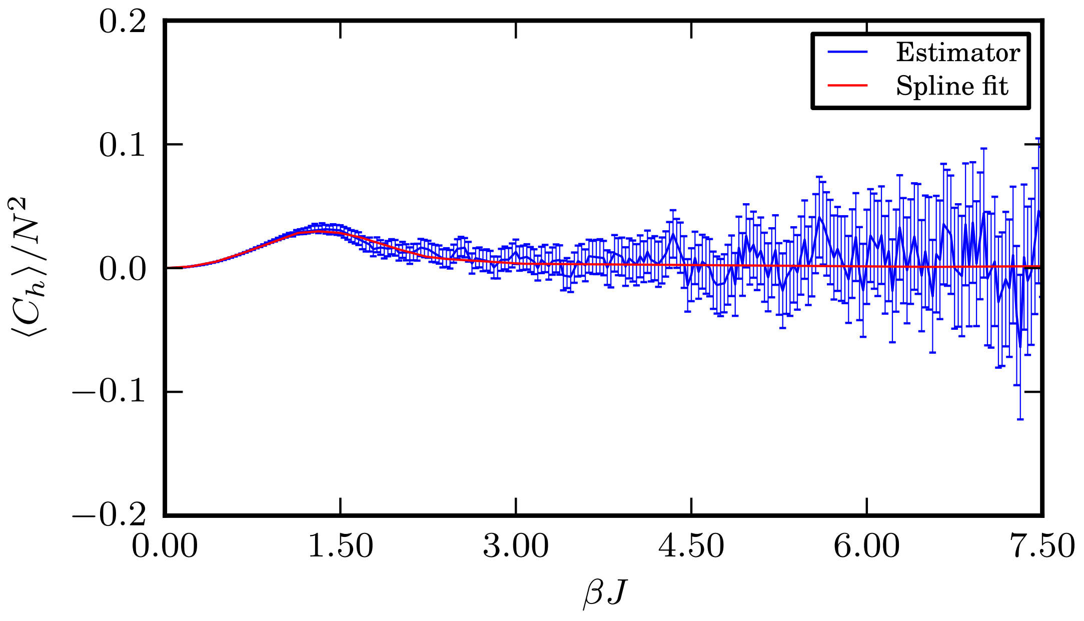


Figure 3.5: The heat capacity estimator calculated by the spline derivative method (red) and by direct stochastic sampling (blue) for a 4×4 antiferromagnetic Heisenberg lattice.

It was found that the spline fit method worked best. The direct stochastic sampling of the heat capacity suffered from huge fluctuations as β grew large. This is due to the β^2 factor in Eq. ??. As β becomes large the fluctuations in $\langle H^2 \rangle - \langle H \rangle^2$ are amplified resulting in heavy statistical noise. This is not the case for the other estimators studied, where the factor is absent. The spline fit on the $\langle H \rangle$ estimator smooths out the statistical fluctuations and removes the dependence on the $\langle H^2 \rangle$ estimator, so performs better. Furthermore, the spline derivative method tends towards the correct value in the large β limit. The heat capacity should be zero at absolute zero otherwise the third law of thermodynamics would be violated

3.2 The 6×6 antiferromagnetic Heisenberg Lattice

The Hilbert space of the 6×6 antiferromagnetic Heisenberg lattice contains a much larger 9.08×10^9 configurations in the $M_S = 0$ subspace. Although it is unfeasible to calculate exact FCI results for this system, it has been well-studied in the literature by methods such as GFQMC.

Fig. **??** shows the DMQMQ finite-temperature energy estimator fluctuating wildly after only a short range in β. It was not possible to obtain the ground-state energy. This simulation took roughly 10.5 hours on 64 cores (2.8 Ghz) and although the number of psips was only a fraction of the Hilbert space, DMQMC cannot be regarded as a competitive QMC algorithm unless some major improvements are found.

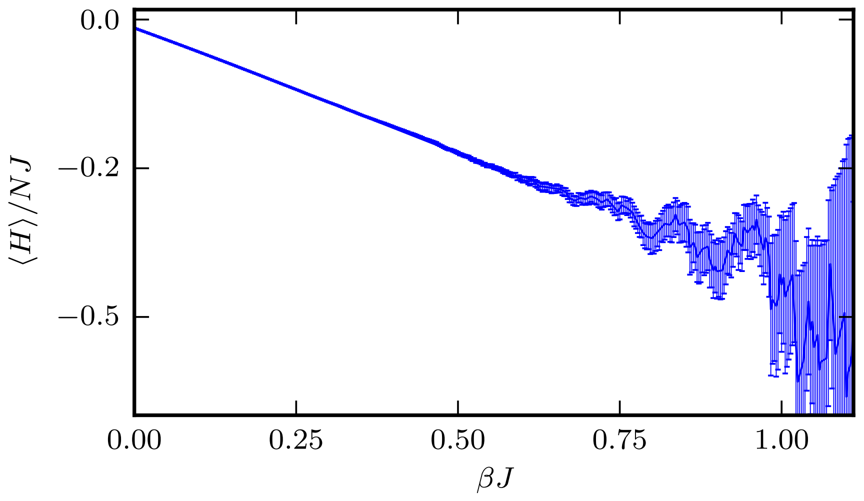


Figure 3.6: The DMQMC finite-temperature energy estimator for the 6×6 antiferromagnetic Heisenberg model. The simulation ran with an initial population of 10^6 psips and for 10^3 β-loops.

3.3 Importance sampling

It was soon discovered that DMQMC was ineffective because the number of psips on the diagonal of the density matrix decays exponentially with inverse-temperature as shown in Fig. **??**. As psips spawn out from the diagonal and start to explore the complete space of density matrix elements, the chance of them spawning back onto the trace becomes very small. For example, psips that carry more than 1 excitation between their bitstring labels can no longer spawn directly back onto the diagonal, but have to do so in a number of steps. Meanwhile, the psips that were originally on the trace are killed off by the clone/death step of the DMQMC algorithm and so the population quickly diminishes.

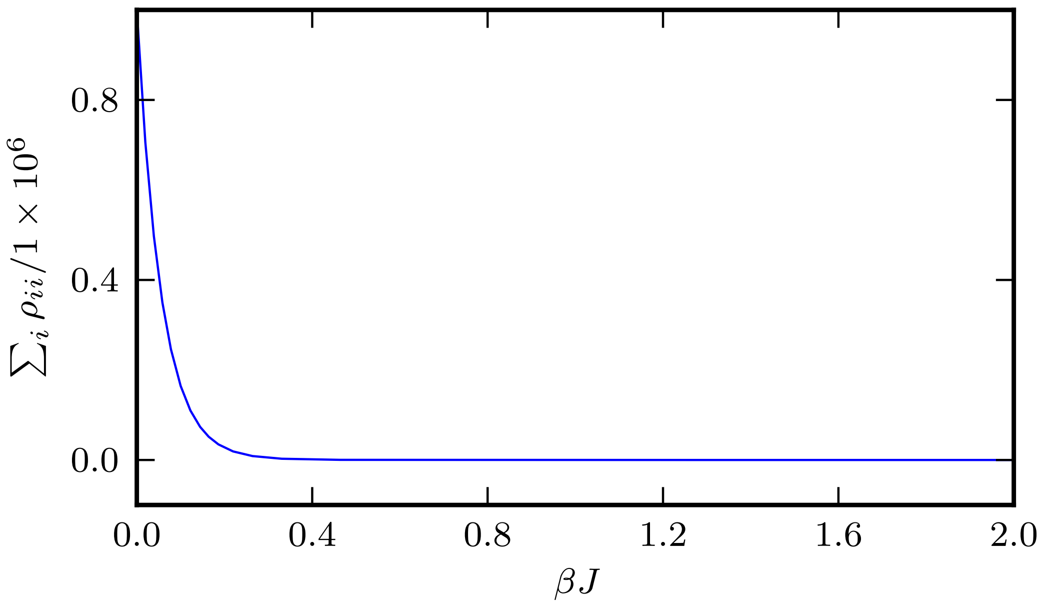


Figure 3.7: Total number of psips on the trace as a function of β.

In order to improve the statistics on the trace at large β, spawning events onto the diagonal can be encouraged in a way that can be undone when calculating expectation value estimators. This can be achieved by increasing the probability of psips spawning onto the diagonal and assigning them a weight to be taken into account when calculating the estimators. More generally, to improve the statistics across all excitations, psips can either be encouraged or discouraged to spawn from one excitation to another and then assigned the relevant weight.

One can think of a scalar function $\mathcal{M}$, which maps an element of $\boldsymbol{\rho}$ to an element of a new matrix $\boldsymbol{\rho}'$, which evolves with improved statistics across all excitations. The scalar function takes as its arguments, the configurations $|i\rangle$ and $|j\rangle$ that represent the position of the element in $\boldsymbol{\rho}$ and the matrix element ρ_{ij} itself. The function compares the two configurations and determines the number of excitations between them and hence scales the matrix element accordingly (equivalent to scaling the number of psips). So $\mathcal{M}$ can be thought of in the following way,

$$\rho'_{ij} = \mathcal{M}(\rho_{ij}, |i\rangle, |j\rangle) = W(|i\rangle, |j\rangle)\rho_{ij}, \tag{3.1}$$

where W is a scalar function that determines the excitation between $|i\rangle$ and $|j\rangle$ and returns the appropriate scaling factor. It is important to note that $W(|i\rangle, |j\rangle) = W(|j\rangle, |i\rangle)$ as $\boldsymbol{\rho}$ is symmetric.

Now the inverse of $\mathcal{M}$ is obvious,

$$\mathcal{M}^{-1}(|i\rangle, |j\rangle, {\rho'}_{ij}) = \frac{1}{W(|i\rangle, |j\rangle)}{\rho'}_{ij} = \rho_{ij} \tag{3.2}$$

To make notation simple define

$$\begin{aligned} \mathcal{M}(\rho_{ij}) &\equiv \mathcal{M}(|i\rangle, |j\rangle, \rho_{ij}) && (3.3)\\ \mathcal{M}^{-1}({\rho'}_{ij}) &\equiv \mathcal{M}^{-1}(|i\rangle, |j\rangle, {\rho'}_{ij}) && (3.4)\\ W_{ij} &\equiv W(|i\rangle, |j\rangle). && (3.5)\end{aligned}$$

Under this transformation the estimator for the expectation value of an operator, O, can be written

$$\begin{aligned} \langle O\rangle = \frac{\sum_{i,j}\rho_{ij}O_{ji}}{\sum_i \rho_{ii}} &= \frac{\sum_{i,j}\mathcal{M}^{-1}({\rho'}_{ij})O_{ji}}{\sum_i \mathcal{M}^{-1}({\rho'}_{ii})} && (3.6)\\ &= \frac{\sum_{i,j}\frac{1}{W_{ij}}{\rho'}_{ij}O_{ji}}{\sum_i \mathcal{M}^{-1}({\rho'}_{ii})} && (3.7)\\ &= W_{ii}\frac{\sum_{i,j}{\rho'}_{ij}\mathcal{M}^{-1}(O_{ji})}{\sum_i {\rho'}_{ii}}. && (3.8)\end{aligned}$$

In the final step $\frac{1}{W_{ij}}O_{ji} = \mathcal{M}^{-1}(O_{ji})$ due to the symmetry of W and the fact that $\boldsymbol{O}$ and $\boldsymbol{\rho}$ are expressed in the same basis. Moreover, W_{ii} has been taken out of the summation because for each i the excitation level is zero and hence W_{ii} is constant.

So under the importance sampling transformation (Eq. **??**), expectation value estimators can be obtained by applying the inverse transformation to the matrix $\boldsymbol{O}$, then taking the expectation value with respect to $\boldsymbol{\rho}'$ and finally multiplying by the weight given to the zeroth excitation. If the weights W_{ij} are chosen carefully then the statistical error on the expectation value can be minimised.

3.3.1 New population dynamics

Given the importance sampling transformation (Eq. **??**) it is possible to deduce a new set of rules for the population dynamics for the evolution of $\boldsymbol{\rho}'$, by transforming Eq. **??**,

$$\begin{aligned} &\mathcal{M}^{-1}({\rho'}_{ij}(\beta+\Delta\beta)) = \\ &\qquad \mathcal{M}^{-1}({\rho'}_{ij}(\beta)) + \frac{1}{2}\sum_m \left(T_{im}\mathcal{M}^{-1}({\rho'}_{mj}(\beta)) + \mathcal{M}^{-1}({\rho'}_{im}(\beta))T_{mj}\right)\Delta\beta. \quad (3.9)\end{aligned}$$

This simplifies to

$${\rho'}_{ij}(\beta+\Delta\beta) = {\rho'}_{ij}(\beta) + \frac{1}{2}\sum_m \left(\frac{W_{ij}}{W_{mj}}T_{im}{\rho'}_{mj}(\beta) + {\rho'}_{im}(\beta)\frac{W_{ij}}{W_{im}}T_{mj}\right)\Delta\beta \quad (3.10)$$

Therefore the rates at which psips spawn has changed so that they spawn from

$$(i,m)\rightarrow(i,j) \quad \text{with probability} \quad \frac{W_{ij}}{2W_{im}}|T_{mj}| \qquad (3.11)$$

and from

$$(m,j)\rightarrow(i,j) \quad \text{with probability} \quad \frac{W_{ij}}{2W_{mj}}|T_{im}|. \qquad (3.12)$$

It is also important to note that when spawning in the opposite direction to in Eq. **??** and Eq. **??** then the inverse weights must be applied.

3.3.2 Choosing the weights

An optimal set of weights can be chosen by studying the number of psips on each excitation level during a standard DMQMC run. Fig. **??** shows an example how the population of psips on each excitation changes as a function of β. In the ground-state some excitation levels are poorly sampled because they are occupied by small numbers of psips.

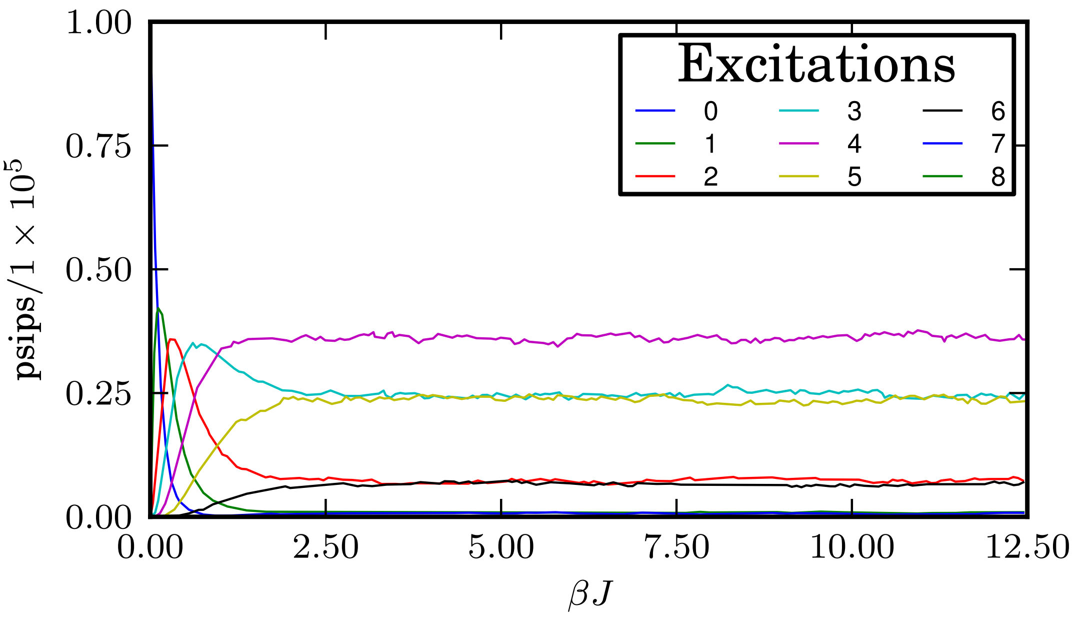


Figure 3.8: The distribution of psips across the different excitation levels for a single β-loop of a DMQMC simulation of a 4×4 antiferromagnetic Heisenberg lattice.

In the ground-state the number of psips on each excitation remains approximately constant. At this point the weights from Sec. **??** can be determined so that the total population of psips is distributed uniformly over the different excitation levels. For instance,

$$W_n = \frac{N_p^{tot}}{N_{ex} N_p^n}, \tag{3.13}$$

where W_n is the weight for the n^{th} excitation, N_p^{tot} is the total number of psips across all excitations, N_{ex} is the total number of excitations and N_p^n is the total number of psips on the n^{th} excitation.

When the importance sampling method was first tested, the weights were applied at the start of the simulation as shown in Fig **??**.

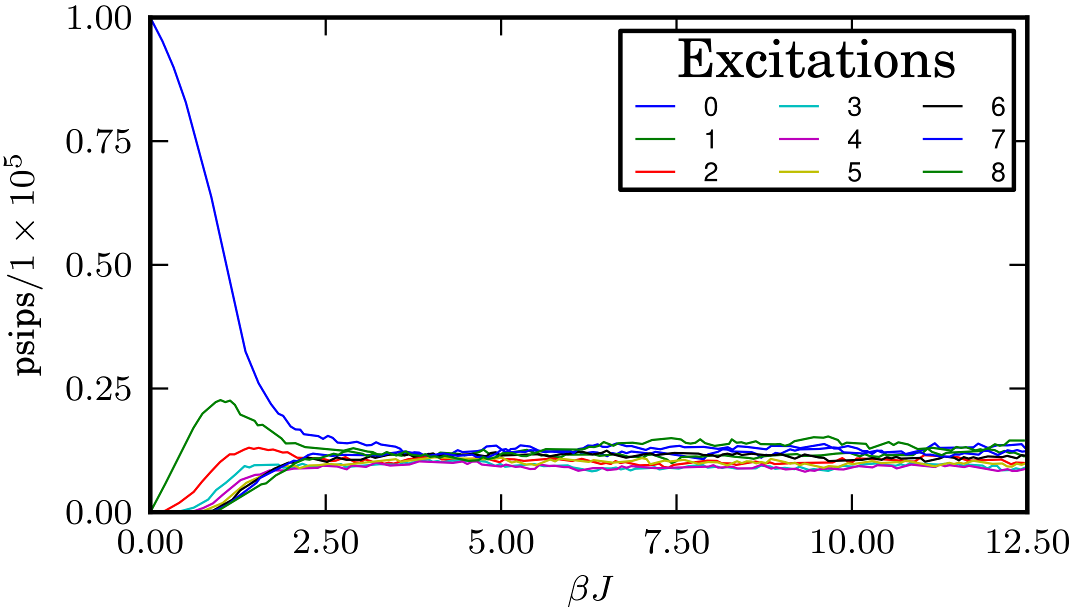


Figure 3.9: The distribution of psips across the different excitation levels for a single β-loop of a DMQMC simulation of a 4×4 antiferromagnetic Heisenberg lattice, with importance sampling weights fully applied at $\beta = 0$.

It is evident that initially spawning events from the diagonal are restricted leading to an effective truncation of the Hilbert space as other excitations are suppressed. This meant that the higher excitations were being under-sampled for the initial evolution, which lead to biased results.

The weights were then introduced gradually, as shown in Fig. **??**, so that in the ground-state ($\beta > \beta_{gs}$) they had reached the intended values. Each weight, W, was set to unity at $\beta = 0$ and for each step of $\Delta\beta$ they were scaled by $W^{\frac{\Delta\beta}{\beta_{gs}}}$ so that for $\beta > \beta_{gs}$ the weights were fully applied. Gradually introducing the weights in this way avoided a discontinuity in the spawning probabilities, whilst also reducing the effective truncation of the Hilbert space. In this way, the psip populations in the pre-ground-state evolution is more similar to the case where no importance sampling is applied (Fig. ??).

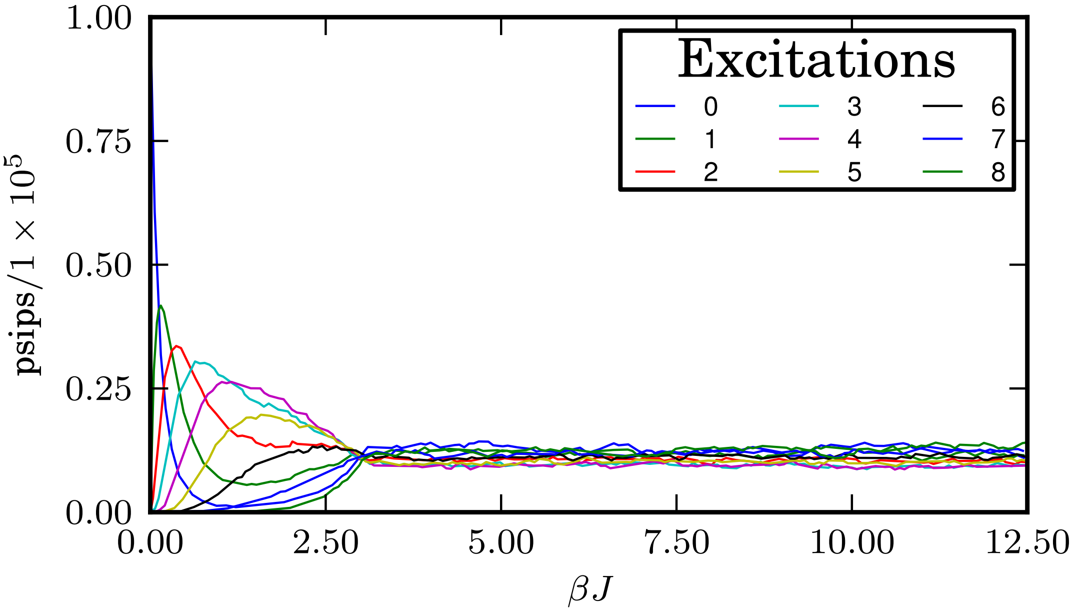


Figure 3.10: The distribution of psips across the different excitation levels for a single β-loop of a DMQMC simulation of a 4×4 antiferromagnetic Heisenberg lattice, with the importance sampling weights introduced gradually.

3.3.3 Importance sampling implementation

This method of importance sampling is simple to implement. When attempting to spawn, as in Eq. ??, the excitation level of the current density matrix element can be found by performing a bit-wise `XOR` between the two bistrings that label it. The number of bits that are set in the output is equal to the number of excitations between the two bitstrings. This operation can also be used to determine the number of excitations between the two bitstrings labelling the density matrix element that a psip is spawning to. It is then easy to apply the correct weight to the spawning probability using Eq. ??.

When adding the contribution from a particular density matrix element to the estimator, the above weighting can be undone easily by calculating the excitation level between the two labelling bit strings (using the bit-wise `XOR` operation) and then dividing by the corresponding weight.

3.4 The 6×6 antiferromagnetic Heisenberg lattice revisited

Now that an importance sampling method has been established, the 6×6 antiferromagnetic Heisenberg model can be re-investigated.

3.4.1 Energy

The importance-sampled DMQMC energy estimator is shown in Fig **??** and with errors in Fig **??**. It shows a dramatic improvement in performance over standard DMQMC and with considerably less sampling. The energy now converges towards a reasonable value for the ground-state energy. For example, a blocking analysis of the DMQMC energy estimator yielded a ground-state energy of $E_0 = -0.6781(5)$, compared to Runge's GFQMC ground-state energy[?] of $E_0 = -0.678871(8)$. An interesting point to note is that in this case the initial number of psips occupy only 1.21×10^{-13} of the elements of the density matrix, so the performance improvement is huge.

3.4.2 Staggered magnetisation

The importance-sampled DMQMC squared staggered magnetisation estimator is shown in Fig **??**. The squared staggered magnetisation estimator converges to the ground-state value of $\langle (M^\dagger)^2 \rangle_0 = 0.2098(2)$, which was found via a blocking analysis of a single β-loop. This agrees with the value of $\langle (M^\dagger)^2 \rangle_0 = 0.20986(8)$ that was calculated by Runge using GFQMC[?] .

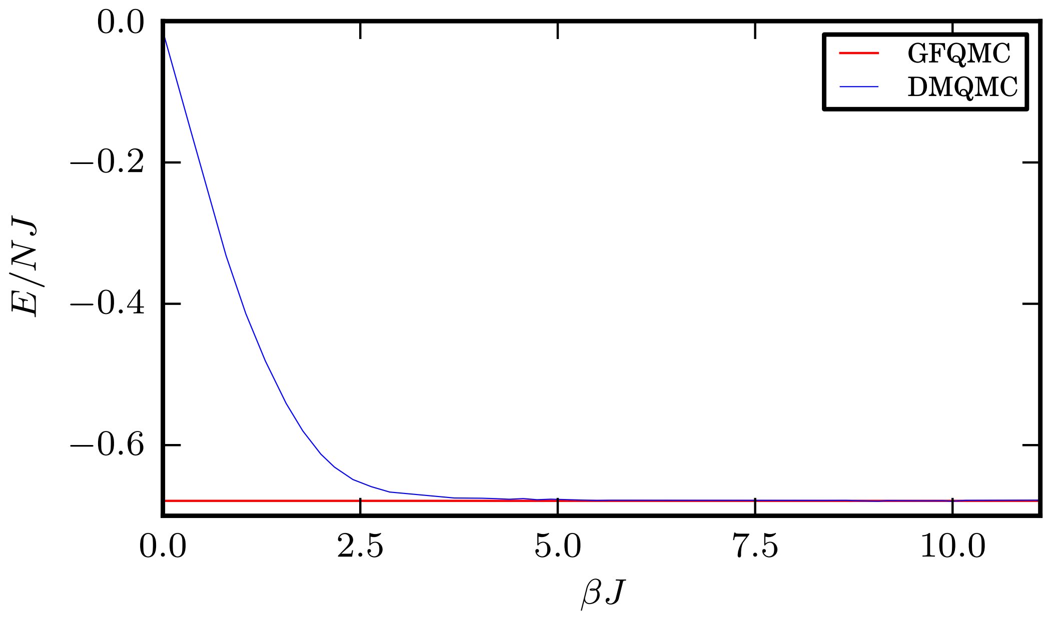


Figure 3.11: The DMQMC finite-temperature energy estimator with importance sampling (blue) and an accurate GFQMC estimate for the ground-state energy (red) for the 6×6 antiferromagnetic Heisenberg model. The simulation ran with an initial population of 1×10^7 psips and for 6 β-loops.

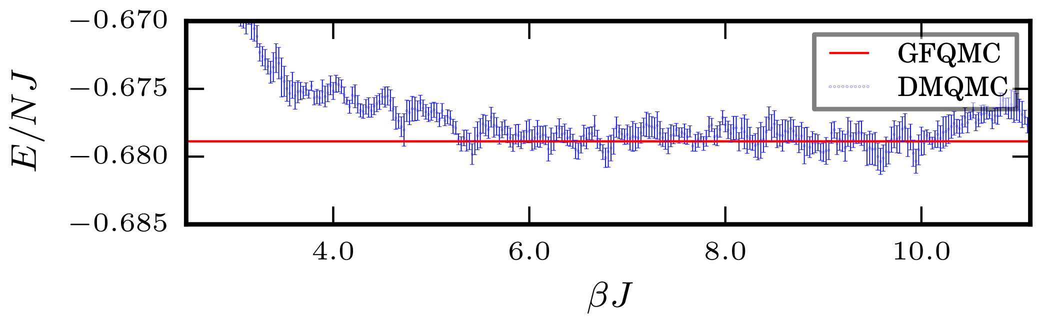


Figure 3.12: The importance-sampled DMQMC finite-temperature energy estimator with statistical errors (blue) and an accurate GFQMC estimate for the ground-state energy (red) for the 4×4 antiferromagnetic Heisenberg model.

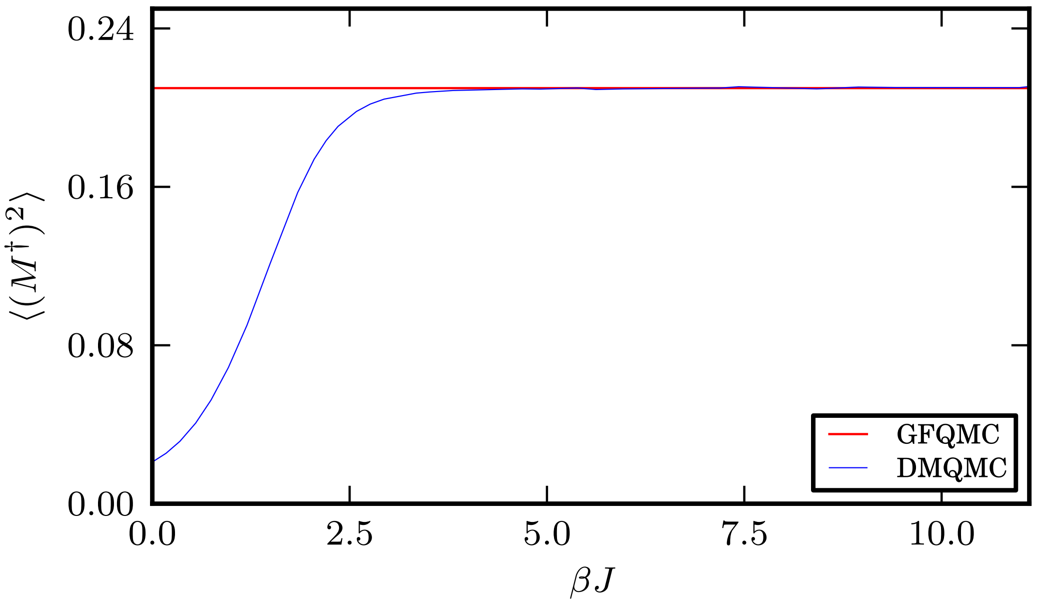


Figure 3.13: The DMQMC finite-temperature squared staggered magnetisation estimator with importance sampling (blue) and an accurate GFQMC estimate for the ground-state squared staggered magnetisation (red) for the 6×6 antiferromagnetic Heisenberg. The simulation ran with an initial population of 10^7 psips and for 6 β-loops.

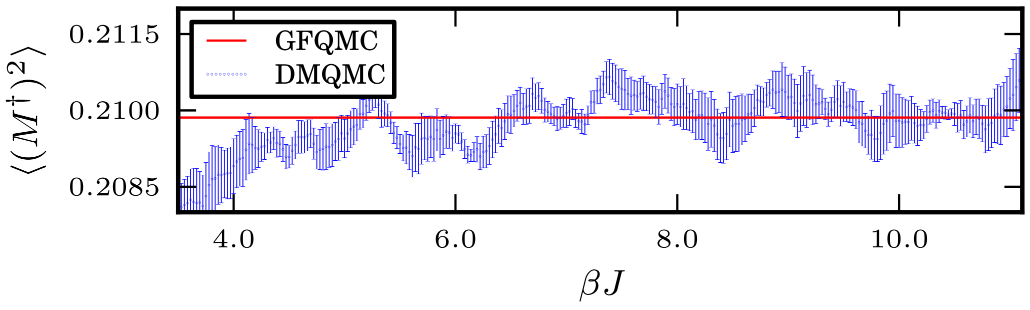


Figure 3.14: The importance-sampled DMQMC finite-temperature squared staggered magnetisation estimator with statistical errors (blue) and an accurate GFQMC estimate for the ground-state square staggered magnetisation (red) for the 6×6 antiferromagnetic Heisenberg model.

3.5 The $4 \times 4 \times 4$ antiferromagnetic Heisenberg lattice

The importance-sampled DMQMC algorithm was also tested on the $4 \times 4 \times 4$ antiferromagnetic Heisenberg lattice and the results are shown in Fig. **??** and Fig. **??**. It was difficult to test the accuracy of the DMQMC method in three dimensions, as the three dimensional antiferromagnetic Heisenberg model features less prominently in the literature. Nevertheless, the DMQMC ground-state energy estimate was compared against an accurate ground-state energy estimate obtained via FCIQMC[?] .

Performing a blocking analysis, on a single β-loop DMQMC simulation gives a ground-state energy of $E_0 = -0.912(6)$, whereas a single run of the FCIQMC algorithm[?] gives a ground-state energy of $E_0 = -0.91509(5)$. The FCIQMC value was obtained using an initial population of 10^6 psips. DMQMC seems a good compromise between accuracy and the ability to obtain finite-temperature estimates.

The Hilbert space for a $4 \times 4 \times 4$ lattice contains roughly 1.83×10^{18} spin configurations and so in this case DMQMC was simulating a density matrix with approximately 1.83×10^{36} elements using only 5×10^5 psips. The statistical errors in Fig. **??** are surprisingly small considering the number of psips used.

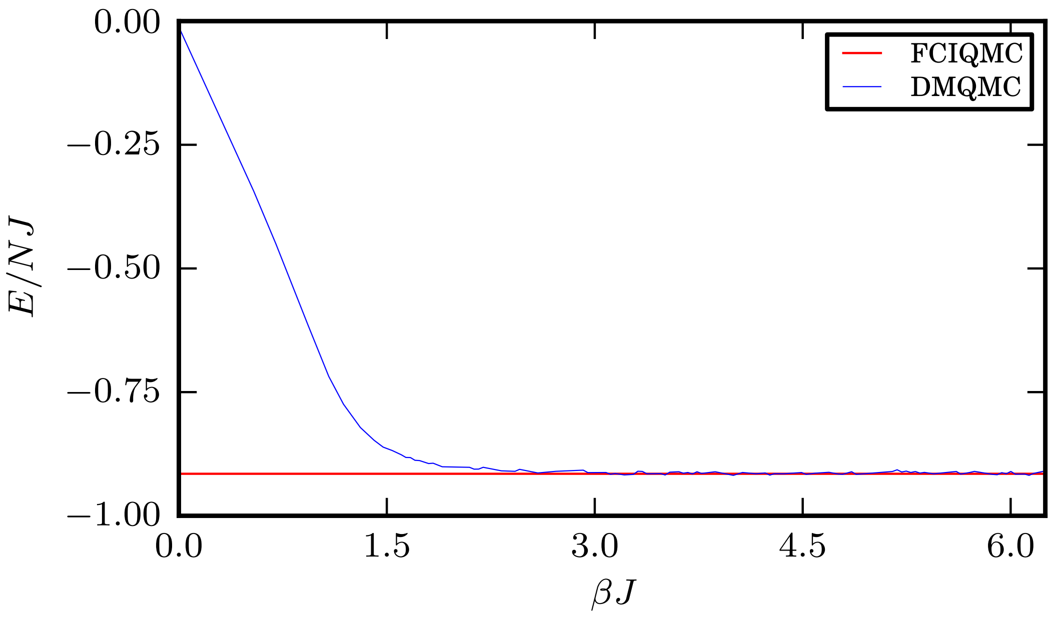


Figure 3.15: The DMQMC finite-temperature energy estimator with importance sampling (blue) and an accurate FCIQMC estimate for the ground-state energy (red) for the $4 \times 4 \times 4$ antiferromagnetic Heisenberg model. The simulation ran with an initial population of 5×10^6 psips and for 43 β-loops.

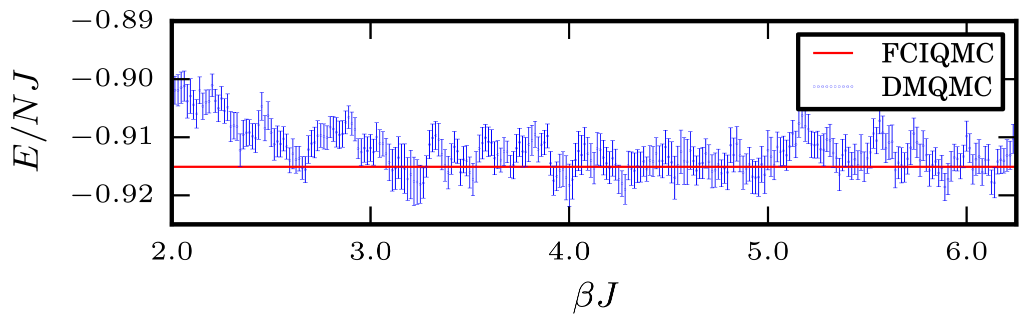


Figure 3.16: The importance-sampled DMQMC finite-temperature energy estimator with statistical errors (blue) and an accurate FCIQMC estimate for the ground-state energy (red) for the $4 \times 4 \times 4$ antiferromagnetic Heisenberg model.

3.6 Summary

The DMQMC implementation was tested on some simple antiferromagnetic Heisenberg models in both two and three dimensions. With the standard algorithm, developed in Ch. **??**, it was found that the initial population of psips was similar to the size of the Hilbert space of configurations and failed for larger systems. It appeared as though the method might not be efficient enough to compete as a QMC method.

However, the reason for this poor performance was soon realised and a suitable importance sampling adaptation to the algorithm was invented, giving dramatically improved results. This new method allowed for accurate finite-temperature calculations of energy and staggered magnetisation for the 6×6 Heisenberg lattice, for which the conventional DMQMC method had failed. Importance sampling also enabled accurate simulation of the $4 \times 4 \times 4$ antiferromagnetic Heisenberg model. In most cases the DMQMC ground-state estimators (obtained via a blocking analysis) were equal to the results in the literature, within error.

Additionally, both versions of the heat capacity estimator were tested on the 4×4 antiferromagnetic Heisenberg lattice. It was found that directly sampling the heat capacity from the DMQMC simulation gave poor results, even with importance sampling. This was because the statistical fluctuations were amplified for large β. This proved the point that, although in DMQMC it is theoretically possible to calculate the expectation value of any quantum mechanical operator, in practice it can prove to be difficult to obtain accurate results. The spline derivative method did provide improved results, but this method requires post-processing of the data and makes it difficult to determine errors.

Chapter 4

Application to Quantum Information

Entanglement is the fundamental inseparability of quantum systems - seen as correlations that emerge in quantum theory and that cannot be described by classical physics. It has been described as “the strangest and least understood facet of quantum theory”[?] and this, in itself, provides a good motivation for the study of quantum entanglement. Furthermore, some future technologies such as quantum computers, quantum teleportation and ion clocks[?] rely on the unique properties of entanglement. A method that can calculate measures of entanglement between qubits or systems of qubits could prove useful in the development of such technologies.

Currently, there are no known scalable simulation methods[?] that are capable of calculating entanglement measures. Entanglement simulations rely on non-scalable algorithms such as density matrix renormalisation group (DMRG) or Lanczos. Other QMC methods do not provide access to the reduced density matrix. However, with DMQMC and the stochastic sampling of the full density matrix, it is possible to obtain the reduced density matrix.

This chapter begins by introducing the two main measures of entanglement (concurrence and Von-Neumann entropy) in quantum information theory. It moves on to discuss how estimators for such methods can be incorporated into the DMQMC algorithm. Finally, the estimator for concurrence is tested against some simple 1D antiferromagnetic Heisenberg rings in the presence and absence of an applied magnetic field.

4.1 Reduced Density Matrix

A major obstacle that prevents QMC simulations of entanglement measures in many-electron quantum systems is the difficulty of obtaining the reduced density matrix. In DMQMC, it is possible to obtain a reduced density matrix for a small subsystem (small enough so that it can be stored and manipulated in computer memory).

Consider two quantum systems A and B that are coupled to form a system C (Fig. **??**) such that

$$\mathcal{H}_C = \mathcal{H}_A \otimes \mathcal{H}_B, \tag{4.1}$$

where $\mathcal{H}_A$, $\mathcal{H}_B$ and $\mathcal{H}_C$ denote the Hilbert spaces of systems A, B and C.

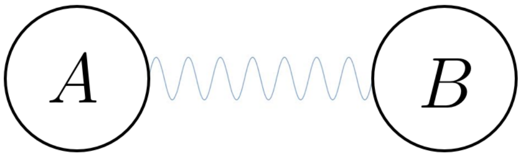


Figure 4.1: Two quantum systems, A and B coupled such that the quantum state of the complete system lives in a Hilbert space that is the tensor product of the Hilbert spaces of A and B.

In the Schmidt basis**?** a quantum state, $|\psi\rangle$, of system C can be expressed in terms of the basis vectors of systems A and B,

$$|\psi\rangle = \sum_n \alpha_n |a_n\rangle \otimes |b_n\rangle . \tag{4.2}$$

With $|a_n\rangle \in \mathcal{H}_A$ and $|b_n\rangle \in \mathcal{H}_B$. Now consider taking a measurement, O_A, on system A without touching B, the expected outcome of this measurement is then,

$$\begin{aligned}
\langle\psi| O_A \otimes \mathbb{1}_B |\psi\rangle &= \langle\psi| \sum_n \alpha_n O_A |a_n\rangle \otimes |b_n\rangle && (4.3)\\
&= \sum_{n,m} \alpha_m^* \alpha_n \langle a_m| O_A |a_n\rangle \otimes \langle b_m|b_n\rangle && (4.4)\\
&= \sum_n |\alpha|^2 \langle a_n| O_A |a_n\rangle && (4.5)\\
&= \mathrm{Tr}_A(\rho_A O_A), && (4.6)
\end{aligned}$$

where $\mathbb{1}_B$ is the identity operator acting on system B, ρ_A is the reduced density operator[i] for system A and Tr_A is the partial trace over the basis of system B. The reduced density operator, ρ_A may be obtained by "tracing out" system B from the projector onto state $|\psi\rangle \in \mathcal{H}_C$ i.e.

$$\begin{aligned}
\rho_C &= |\psi\rangle\langle\psi| = \sum_{n,m} \alpha_n \alpha_m^* |a_n\rangle\langle a_m| \otimes |b_n\rangle\langle b_m| && (4.7)\\
\rho_A &= \mathrm{Tr}_B(\rho_C) = \sum_{n,m} \alpha_n \alpha_m^* |a_n\rangle\langle a_m| \otimes \mathrm{Tr}(|b_n\rangle\langle b_m|) && (4.8)\\
&= \sum_n |\alpha|^2 |a_n\rangle\langle a_n|. && (4.9)
\end{aligned}$$

So in DMQMC, it is also possible to stochastically sample the reduced density matrix for a subsystem of the total system under study. From this it is possible to calculate expectation values for measurement on that subsystem. Moreover, it allows for some interesting measures of quantum entanglement between the subsystems A and B.

4.2 Von-Neumann Entropy

The Von-Neumann Entropy is an extension of the Shannon entropy, from classical information theory into quantum information theory. In classical information

[i]it satisfies the three properties in Sec. ??

theory, the Shannon entropy for a probability distribution $q_1, \ldots, q_n$ is

$$S_C(q_1, \ldots, q_n) = \sum_i q_i \log_2 1/q_i. \tag{4.10}$$

Similarly for a quantum state living in a finite D-dimensional Hilbert space, the Von-Neumann entropy is

$$S_Q(\boldsymbol{\rho}) = -\mathrm{Tr}(\boldsymbol{\rho} \log_2 \boldsymbol{\rho}) = \sum_i p_i \log_2 1/p_i, \tag{4.11}$$

where p_i are the eigenvalues of $\boldsymbol{\rho}$. It describes the purity of the system. If one considers the density matrix in diagonal form, the eigenvalues p_i represent the probability of the system being in some state $|i\rangle$ of the diagonal basis. Hence, for a pure state one eigenvalue is unity, whilst all the others are zero, giving a zero Von-Neumann entropy. However, for a maximally entangled mixed state, with equally likely probabilities of being in any of the states $|i\rangle$ in the diagonal basis, the Von-Neumman entropy is $\log_2 D$.

Therefore, if the Von-Neumann entropy is calculated for the reduced density matrix for system A, in Fig. **??**, it measures the amount of quantum entanglement between systems A and B. For example, if the Von-Neumann entropy is zero, then A is a pure state and so there is no entanglement between A and B, assuming that the overall system C is in a pure state. On the other hand if the Von-Neumann entropy is $\log_2 D$, then subsystem A is in a fully mixed state and A and B must be maximally entangled.

In order to compute the Von-Neumann entropy (Eq. **??**), the subsystem A must be sufficiently small so that the reduced density matrix can be diagonalised directly. The Von-Neumann entropy has not been properly tested due to time restrictions.

4.3 Concurrence and Entanglement of Formation

In 2001, Hill and Wootters? introduced a new measure of entanglement for a two-qubit mixed state. For the case where the total system is mixed, the Von-Neumann entropy is no longer a good measure of quantum entanglement, because both subsystems A and B can have non-zero entropy even if there is no entanglement. The measure, known as the "entanglement of formation" is a generalisation of Von-Neumann entropy. The entanglement of formation of a bipartite quantum system can be defined as "the minimum number of singlets needed to create an ensemble of pure states that represents a mixed state"? . In this case a "singlet" refers to a state with zero total spin.

Another definition of the Von-Neumann entropy, $E(\Phi)$, is the number of singlet pairs, required to required to create a copy of a pure state $|\Phi\rangle$? .The number of singlets required to create a copy of some mixed state that is expressed as an ensemble of pure states, $\rho = \sum_{j=1}^{N} p_j |\Phi_j\rangle \langle\Phi_j|$, can be written

$$N_s = \sum_{j=1}^{N} p_j E(\Phi_j). \tag{4.12}$$

This number depends on the particular ensemble so the minimum over all pure-state decompositions is taken in order to pick out the irreducible entanglement of the mixed system

$$E_f(\rho) = \min \left(\sum_{j}^{N} p_j E(\Phi_j) \right). \tag{4.13}$$

For the special case of a two-qubit mixed state the entanglement of formation can be calculated from a quantity known as "concurrence". For a reduced density matrix, $\boldsymbol{\rho}_A$, where in this case A refers to a subsystem of two qubits, the concurrence is defined as

$$\mathcal{C}(\boldsymbol{\rho}_A) \equiv \max(0, \gamma_1 - \gamma_2 - \gamma_3 - \gamma_4), \tag{4.14}$$

in which $\gamma_1 > \gamma_2 > \gamma_3 > \gamma_4$ are the eigenvalues of the matrix

$$\boldsymbol{R} = \sqrt{\sqrt{\boldsymbol{\rho}_A}\tilde{\boldsymbol{\rho}}_A\sqrt{\boldsymbol{\rho}_A}} \quad \text{with} \quad \tilde{\boldsymbol{\rho}}_A = (\sigma_y \otimes \sigma_y)\boldsymbol{\rho}_A^*(\sigma_y \otimes \sigma_y), \tag{4.15}$$

where σ_y is the Pauli spin matrix for the y-direction.

The concurrence, $\mathcal{C}$, ranges from zero to one and is monotonically related to the entanglement of formation? . In this way the concurrence can also be regarded as a measure of entanglement. The entanglement of formation for two qubits? is given by,

$$\mathcal{E}(\mathcal{C}) = h\left(\frac{1+\sqrt{1-\mathcal{C}^2}}{2}\right) \quad \text{with} \quad h(x) = -x\log_2 x - (1-x)\log_2(1-x). \tag{4.16}$$

The concurrence is easy to compute for the Heisenberg model. If it is assumed that the Hamiltonian is real (as is the case in the Heisenberg model) and therefore that the reduced density matrix is also real, the problem is reduced to finding the moduli of the eigenvalues of

$$\tilde{\boldsymbol{R}} = \boldsymbol{\rho}_A(\sigma_y \otimes \sigma_y). \tag{4.17}$$

As $\tilde{\boldsymbol{R}}$ is only a 4×4 matrix it is trivial to compute the concurrence.

4.4 Implementation of the ground-state entanglement estimators

Whilst it is possible to calculate finite-temperature estimators for both the Von-Neumann entropy and Concurrence, the current implementation can only handle such estimators for the ground-state. This is because the current implementation is based on previous FCIQMC code which only allows simulations for a single M_S subspace. In FCIQMC the ground-state always lies within a known M_S. For example, in a simulation for the antiferromagnetic Heisenberg model with zero applied magnetic field the ground-state is within the $M_S = 0$ subspace. Unfortunately, there was not enough time during this MSci project to adapt the code so that simulations could run in all subspaces simultaneously, which would allow for finite-temperature estimators of entanglement.

In the current implementation the reduced density matrix, for a specified subsystem of spins, is averaged over all $\beta > \beta_{gs}$, where β_{gs} is the critical β beyond which the simulation enters the ground-state. This is to ensure that the reduced density matrix has the highest possible accuracy before computing the entanglement estimators.

In order to obtain the reduced density matrix at a particular $\beta > \beta_{gs}$ the subsystem B is traced out as in Eq. **??**. This was achieved by looping over all elements of the total density matrix with non-zero psip populations and determining whether a particular element contributed towards the reduced density matrix. A simple test (see Fig. **??**) was devised to determine whether this was the case:

1. Define a boolean mask for subsystem B - this is created at the start of the simulation and has bits set to 1 where a spin is to be included in subsystem B.

2. Perform a bit-wise `AND` operation on each of the two bit string ends that label the current density matrix element, with the boolean mask for subsystem B. This sets all bits that represent a spin in subsystem A to 0 and leave those that represent a spin in B unchanged.

3. If the two resultant bit strings are equal then this means that it lies on the diagonal of subsystem B and hence contributes to the reduced density matrix for subsystem A. In this case, the psip charge on this element is added to the corresponding element of the reduced density matrix. If they are not equal then nothing is added to the reduced density matrix.

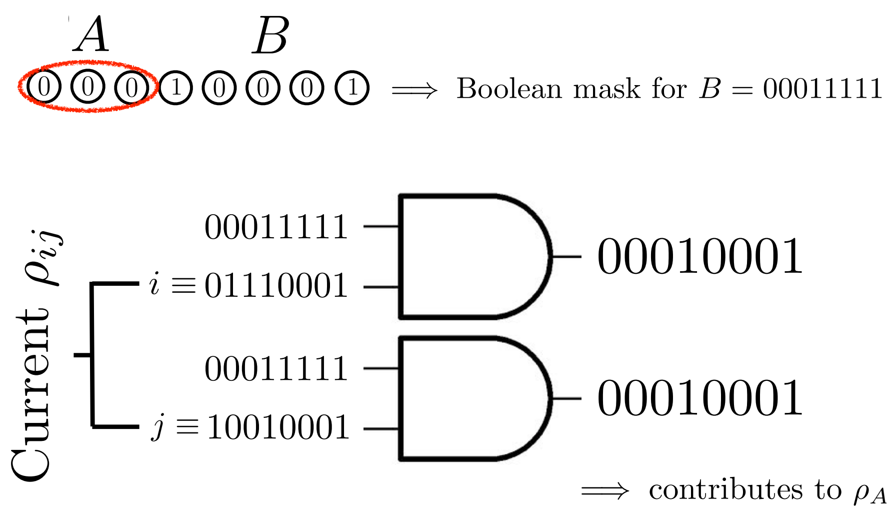


Figure 4.2: Illustration of how a given density matrix element ρ_{ij} is tested to see whether it contributes towards ρ_A for the case that A is 3 neighbouring spins on a chain of length $N = 8$

After obtaining the average reduced density matrix, the Von-Neumann entropy is obtained by directly diagonalising the reduced density matrix[i]. For the concurrence, the matrix $\tilde{R}$ (Eq. ??) is diagonalised[ii]. This whole process is repeated over multiple β-loops to build up statistics and to enable the calculation of errors as in Sec. ??.

4.5 Nearest-neighbour concurrence in Heisenberg rings

The ground-state concurrence for neighbouring spins (qubits), illustrated in Fig. ??, on a bipartite antiferromagnetic Heisenberg ring with N qubits was first calculated by O'Connor and Wooters in 2001[?]. They found that the ground-state concurrence, $\mathcal{C}_{gs}$, is related to the ground-state energy, E_0, in the following way

$$\mathcal{C}_{gs} = -\frac{1}{2}(E_0/N + 1). \tag{4.18}$$

[i]This is achieved using the LAPACK `ssyev` and `dsyev` routines for diagonalising symmetric matrices. These routines use the QR algorithm.

[ii]This is achieved using the LAPACK `sgeev` and `dgeev` routines for diagonalising nonsymmetric matrices. Both routines first use Hessenberg reduction followed by the QR algorithm.

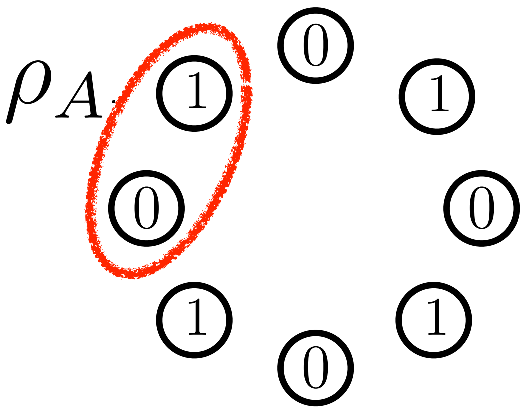


Figure 4.3: Illustration of the subsystem of two qubits, A, on an $N = 8$ antiferromagnetic Heisenberg chain. The system can be regarded as a ring due to the periodic boundary conditions adopted.

To test whether the DMQMC ground-state concurrence estimator was implemented correctly, several Heisenberg rings of differing size were simulated. The results in Tab. 4.1, show that the DMQMC ground-state concurrence estimator was correct within error for bipartite rings up to $N = 10$. Moreover, an $N = 36$ Heisenberg ring was simulated with importance sampling and the ground-state concurrence of $\mathcal{C}_{gs} = 0.3873(8)$ was obtained. This agrees well with the value of $\mathcal{C}_{gs} = 0.38748(4)$ obtained from FCIQMC[?] and Eq. ??.

N	2	4	6	8	10	...	36	∞
[?] $\mathcal{C}_{gs}$	1.000	0.500	0.434	0.412	0.403	...	–	0.386
DMQMC	1.0006(6)	0.5005(4)	0.4342(5)	0.4129(5)	0.4031(4)	...	0.3873(8)	–

Table 4.1: Comparision of the DMQMC ground-state concurrence calculation with the exact values for neighbouring qubits on a simple one dimensional bipartite antiferromagnetic Heisenberg ring of N qubits.

4.6 Concurrence in a uniform magnetic field

Applying a magnetic field to a system of spins changes the structure of the ground-state spin configuration and hence the ground-state entanglement properties of the system. In quantum information it is interesting to look at how the entanglement between qubits changes with a uniformly applied magnetic field. Since the magnetic field can be controlled it is possible to also control the amount of entanglement between certain qubits.

As an example a uniform magnetic field was applied to a Heisenberg antiferromagnetic ring with $N = 8$, the effect of which is shown in Fig. **??**. Within a given M_S subspace, changing the magnetic field shifts the diagonal elements of the Hamiltonian in Eq. **??**. Therefore, although the energy of the ground-state is changing, the ground-state wave function does not. Hence, for a given M_S the concurrence between two neighbouring qubits does not change.

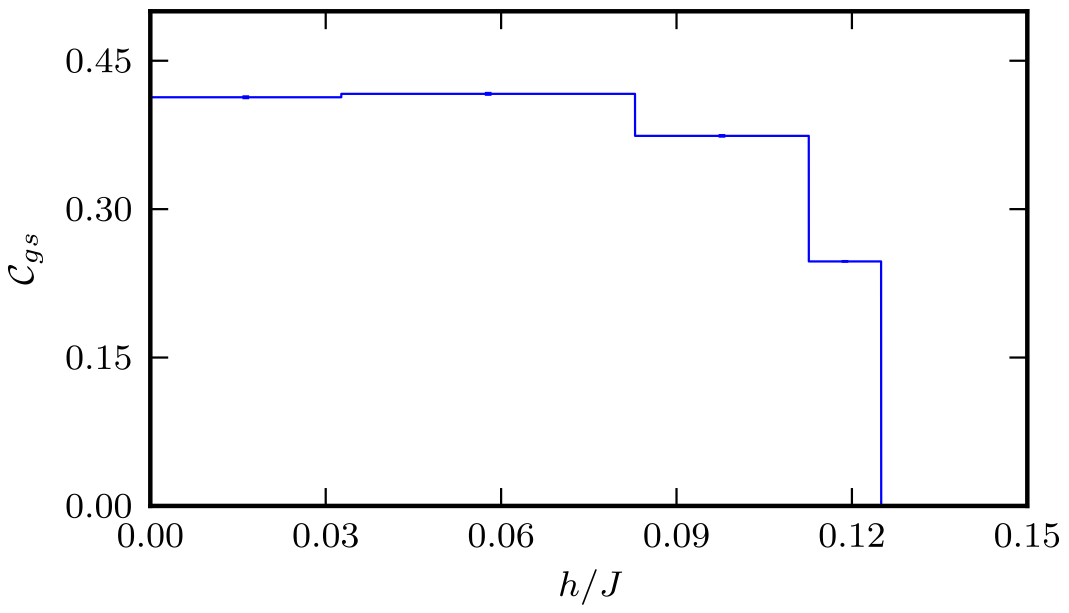


Figure 4.4: The ground-state concurrence, $\mathcal{C}_{gs}$, between two neighbouring qubits on an $N = 8$ antiferromagnetic Heisenberg ring as a function of magnetic field strength in the (z-direction), h.

However, when the lowest energy of a particular M_S subspace is shifted so that it is higher than that of the next M_S, the ground-state wave function changes to one that belongs to the new M_S subspace. As the form of the wave function changes so do the entanglement properties of the system. This explains, why in Fig. **??** there are regions of constant entanglement.

As the magnetic field increases, the amount of entanglement decreases as more spins are forced into alignment and the orientation of neighbouring spins becomes less correlated. Beyond a critical value of the magnetic field, $h/J = 0.125$, the concurrence drops to zero. This is the point where all spins become aligned and quantum correlations between neighbouring qubits disappears. It can be regarded as a quantum phase transition at $T = 0$ due to quantum fluctuations, rather than thermal ones**?** .

4.7 Concurrence with increasing separation

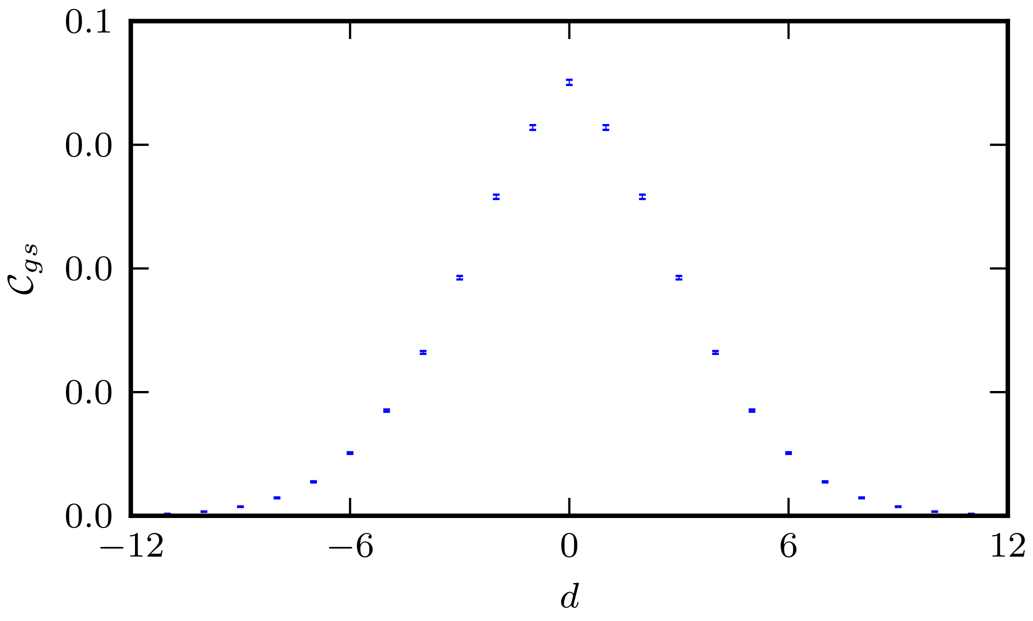
Entanglement length**?** , l_E, is another important property of many-body quantum systems that is relevant to quantum information theorists and experimentalists. It describes how quickly the entanglement between two qubits falls to zero as the separation between them is increased.

In the Heisenberg model it is assumed that spin-spin interactions are short-range and between nearest neighbours. It is therefore difficult to find quantum correlations between spins that are not nearest neighbours. However, there is another critical value of the magnetic field, h_{sym}, for which an antiferromagnetic Heisenberg system is in the symmetric state

$$\psi_{sym} = \frac{1}{\sqrt{N}}(|100\ldots0\rangle + |010\ldots0\rangle + |001\ldots0\rangle + \cdots + |000\ldots1\rangle). \qquad (4.19)$$

In this state it is evident that there is non-zero entanglement between any two qubits. This makes it possible to study how the entanglement between qubits changes as the distance between them is varied. Another benefit of this is that at this critical magnetic field the number of possible spin configurations is small and DMQMC can simulate the system very accurately.

Fig. **??** shows how the concurrence drops off as the distance between the two qubits increases, for an antiferromagnetic Heisenberg ring with $N = 36$. Ignoring the effects near $d = 0$ from the periodic boundary conditions, it can be seen that the concurrence approaches $\mathcal{C}_{gs} \propto e^{-d}$. Hence the entanglement length approaches $l_E \approx 1$. This makes sense as in the Heisenberg model it is approximated that the J-coupling occurs between nearest neighbours only and entanglement between non-nearest neighbours is due to non-direct interactions.



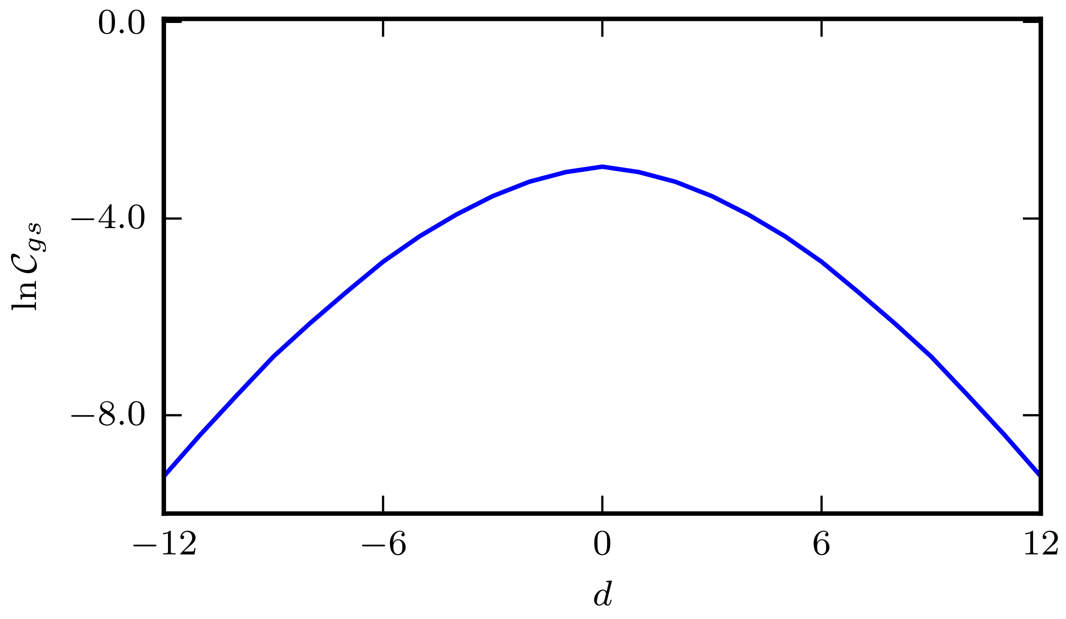


Figure 4.5: Ground-state concurrence with statistical errors (top) and logarithm of ground-state concurrence (bottom) for the $N = 36$ antiferromagnetic Heisenberg ring, with $h/J = 1.37$, as a function of separation between the two qubits. For example, $d = 0$ is nearest neighbours, $d = 1$ is next-nearest neighbours etc. It was difficult to obtain good statistics for $|d| > 10$ as the concurrence was so small.

4.8 Summary

In this chapter two quantum entanglement measures, the Von-Neumann entropy and the concurrence, were introduced. And the integration of these entanglement estimators into the DMQMC algorithm was discussed. The ground-state concurrence estimator was tested on one dimensional antiferromagnetic Heisenberg rings and the results were found to agree very well with the analytic values.

Next the effects of a magnetic field on the entanglement between two nearest-neighbour qubits on an $N = 8$ antiferromagnetic Heisenberg ring were studied. It was found that there were quantum sharp transitions between different levels of entanglement due to quantum phase transitions. Finally, DMQMC was used to study how the entanglement between two qubits falls off as a function of the separation between them. It was found that there was a unit entanglement length, which agrees with the assumption in the Heisenberg model that interactions are only between nearest-neighbours.

Chapter 5

Conclusion

This project sought to overcome the two main restrictions of FCIQMC by stochastically sampling the many-electron thermal density matrix rather than the many-electron wave function. Although sampling the density matrix increases the computational effort because the psips now have to live, die, clone and annihilate in a space that has squared in size, it in theory allows for the calculations of any quantum mechanical observable at finite-temperatures.

The algorithm was developed by studying an analogy between the evolution of the many-electron wave function in imaginary-time and the many-electron thermal density matrix as a function of inverse-temperature. A set of population dynamics, similar to those in FCIQMC were chosen so that they simulate a first-order finite difference approximation that relate the thermal density matrix at one inverse-temperature to the thermal density matrix at the step before.

The DMQMC algorithm was implemented in `FORTRAN90` and built upon a current highly parallel and optimised implementation of FCIQMC. It was decided to test the implementation on some well-studied two and three dimensional antiferromagnetic bipartite Heisenberg lattices because they remove the complication of the fermion sign-problem which plagues some many-electron models. At first, it was found that the DMQMC method was far less efficient than FCI for the 4×4 antiferromagnetic Heisenberg lattice and actually failed for the 6×6 antiferromagnetic Heisenberg lattice.

It was found that the reason for this failure was that the number of psips on the trace decayed exponentially and quickly became undersampled. This was remedied by the introduction of an importance sampling method that looked to increase the sampling of rare spawning events. Under this importance sampling transformation it was found that the number of initial psips could be significantly reduced and that DMQMC could now be used to simulate larger systems for which FCI is unfeasible.

Futhermore, larger systems such as the $4 \times 4 \times 4$ antiferromagnetic Heisenberg lattice could be simulated with a number of psips that was a only a minute fraction of the total number of elements in the thermal density matrix. As expected, DMQMC was found to be less accurate than FCIQMC for calculating ground-state properties with a similar number of psips, but it has the benefit that all of data from the simulation was useful in determining finite-temperature properties. In FCIQMC relevant data can only really be retrieved once the system is in the ground-state in the limit of large imaginary-time.

In theory it is simple to calculate the expectation value of any quantum mechanical observable in DMQMC. However, in practice it was difficult to directly sample the estimator for the heat capacity. This was due to a factor of β^2 in the estimator equation that leads to the amplification of statistical errors as β becomes large. A solution was to calculate the derivative of the energy estimator using a cubic smoothing spline fit. However, this rather inelegant method leads to difficulties in obtaining statistical errors.

The ease of obtaining the reduced density matrix from stochastic sampling of the full thermal density matrix opened up the possibility of calculating entanglement measures and an avenue into quantum information theory. Current QMC methods do not provide access to the reduced density matrix and other methods are non-scalable or can only simulate one dimensional systems (DMRG). So there is need for methods such as DMQMC in quantum information. Two entanglement measures, Von-Neumann entropy and concurrence, were implemented of which the ground-state concurrence estimator was tested on one dimensional antiferromagnetic Heisenberg rings for which analytic values exist.

Some work on the DMQMC implementation is still needed. For example, the current implementation can only simulate systems within a fixed M_S subspace. One needs to find a way of either simulating across all M_S subspaces simultaneously or combining the results from all the different M_S simulations. At this point, it would be possible to obtain the full finite-temperature picture across all M_S values and it would allow for finite-temperature entanglement calculations. Additionally, it seems that the FCIQMC shift update algorithm introduces biases[i] and this problem must be solved.

In conclusion, a new QMC method has been developed. It shares some of the important features of FCIQMC, but almost fully overcomes two of FCIQMC's main restrictions at the expense of efficiency. Some more work is required, but there are signs that it could be an important QMC method for finite-temperature calculations and could be used to tackle some poorly understood aspects of quantum information theory.

Total words: 9997 (via TeXcount 2.3)

[i]There was not enough space in this report to discuss this in full.

References

. G. H. Booth, A. J. W. Thom, and A. Alavi, J. Chem. Phys. **131**, 054106 (2009).

. A. Szabados, P. Jeszenszki, and P. R. Surján, J. Chem. Phys. , 1 (2011).

. P. R. C. Kent, *Techniques and Applications of Quantum Monte Carlo*, Ph.D. thesis, University of Cambridge (1999).

. W. M. C. Foulkes, L. Mitas, R. J. Needs, and G. Rajagopal, Rev. Mod. Phys. **71**, 33 (2001).

. M. Bajdich and L. Mitas, Act. Phys. Slov. **59**, 81 (2009), arXiv:arXiv:1008.2369v1 .

. J. S. Spencer, N. S. Blunt, and W. M. C. Foulkes, J. Chem. Phys. **136**, 054110 (2012).

. C. J. Isham, *Lectures on Quantum Theory. Mathematical and Structural Foundations*, 1st ed. (Imperial College Press, 1995).

. M. Foulkes and J. Spencer, Private communication (2011).

. J. B. Anderson, Physics **1499** (1975), 10.1063/1.431514.

. C. Contaldi, *Computational Physics Lecture Notes* (Imperial College London, 2011).

. C. J. Umrigar, M. P. Nightingale, and K. J. Runge, J. Chem. Phys. **99**, 2865 (1993).

. W. Heisenberg, Zeitschrift fur Physik **38** (1926).

. P. Dirac, Proceedings of the Royal Society of London. **112**, 661 (1926).

. M. Karbach and G. Muller, "Introduction to the Bethe ansatz I," (1998), arXiv:9809162 [cond-mat] .

. M. Saito and M. Matsumoto, in *Monte Carlo and Quasi-Monte Carlo Methods 2008*, edited by P. L' Ecuyer and A. B. Owen (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009) pp. 589–602.

. K. J. Runge, Phys. Rev. B **45**, 7229 (1992).

. J. van den Brink, in *Lecture Notes on Theory of Condensed Matter* (University of Leiden, 2012) pp. 5–28.

. C. L. Henley, in *Lecture notes* (Cornell University, 2007) pp. 15–23.

. H. Flyvbjerg and H. G. Petersen, J. Chem. Phys. **91**, 461 (1989).

. Obtained using J. Spencer's FCI code (2011).

. E. Dagotto and A. Moreo, Phys. Rev. B **38**, 5087 (1988).

. K. J. Runge, Phys. Rev. B **45**, 12292 (1992).

. Obtained using J. Spencer's FCIQMC code (2011).

. T. Rudolph, *Quantum Information Lecture Notes* (Imperial College London, 2011).

. M. B. Hastings, "Measuring Renyi Entanglement Entropy with Quantum Monte Carlo," (2010), arXiv:arXiv:1001.2335v2 .

. A. Ekert and P. L. Knight, Am. J. Phys. **63**, 415 (1994).

. S. Hill and W. Wootters, Phys. Rev. Lett. **78**, 5022 (1997).

. W. Wootters, Quant. Inf. Comp. **1**, 27 (2001).

. K. O'Connor and W. Wootters, Phys. Rev. A **63**, 1 (2001).

. T. Rudolph, Private communication (2012).

. M. Arnesen, S. Bose, and V. Vedral, Phys. Rev. Lett. **87**, 4 (2001).

. M. Born and J. R. Oppenheimer, Ann. Physik **84**, 457 (1927).

. A. Szabo and N. S. Ostlund, *Modern Quantum Chemisty: Introduction to Advanced Electronic Structure Theory* (Dover Publications, 1996).

. B. H. Bransden and C. J. Joachain, *Quantum Mechanics*, 2nd ed. (Prentice Hall, 2000).

. A. J. James, *PhD Thesis*, Ph.D. thesis, Imperial College London (1996).

. A. M. Mazzone and V. Morandi, Comp. Mat. Sci. **38**, 231 (2006).

. S. Sorella, S. Baroni, R. Car, and M. Parrinello, Europhys. Lett. **8**, 663 (1989).

. G. H. Booth and A. Alavi, J. Chem. Phys. **132**, 174104 (2010).

. G. H. Booth, D. Cleland, A. J. W. Thom, and A. Alavi, J. Chem. Phys. **135**, 084104 (2011).

. C. H. Reinsch, Num. Math. **83**.

Appendix A: Basic electronic structure and FCIQMC

This appendix briefly reviews some of the fundamentals in electronic structure theory and projector QMC methods before moving onto a brief outline of the recent FCIQMC method.

The many-electron Schrödinger Equation

For a many-electron problem with N electrons and M nuclei the Hamiltonian, in Hartree atomic units, is

$$\begin{aligned} H = & -\frac{1}{2}\sum_{i=1}^{N}\nabla^2_{\mathbf{r}_i} - \sum_{i=1}^{M}\frac{1}{2M_A}\nabla^2_{\mathbf{r}_A} - \sum_{i=1}^{N}\sum_{A=1}^{M}\frac{Z_A}{|\mathbf{r}_i - \mathbf{d}_A|} \\ & + \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} + \frac{1}{2}\sum_{A=1}^{M}\sum_{B\neq A}^{M}\frac{Z_A Z_B}{|\mathbf{d}_A - \mathbf{d}_B|}, \end{aligned} \tag{1}$$

where M_A is the ratio of the mass of the nucleus A to the mass of an electron; Z_A is the atomic number of nucleus A; $\mathbf{r}_i$ is the position vector of the ith electron; $\mathbf{d}_A$ is the position vector of the Ath nuclei. The first term in Eq. **??** is the kinetic energy operator of the electrons; the second term is the kinetic energy operator of the nuclei; the third term represents the electron-nuclei coulomb attraction; the fourth and fifth terms represent the electron-electron and nuclei-nuclei coulomb repulsions respectively.

The Born-Oppenheimer approximation? assumes that the nuclei are stationary, due to the fact they are much heavier than the electrons. Hence, the second term in Eq. **??** can be neglected and the nuclei-nuclei repulsion term is assumed to be constant. So the time-independent many-electron Schrödinger equation under the Born-Oppenheimer approximation becomes

$$\left[-\frac{1}{2}\sum_{i=1}^{N}\nabla^2_{\mathbf{r}_i}-\sum_{i=1}^{N}\sum_{A=1}^{M}\frac{Z_A}{|\mathbf{r}_i-\mathbf{d}_A|}+\frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\frac{1}{|\mathbf{r}_i-\mathbf{r}_j|}\right]\psi(\mathbf{R})=E\psi(\mathbf{R}). \quad (2)$$

The solution, $\psi(\mathbf{R})$, of Eq. **??** is the many-body wave function for N electrons and is a function of the N-electron position vector $\mathbf{R}=(\mathbf{r}_1,\mathbf{r}_2,\dots,\mathbf{r}_N)$. Physically this can be interpreted as the probability that one electron is in region R_1 and that a second electron is in region R_2 and so on is

$$P(\mathbf{r}_1\in R_1|\mathbf{r}_2\in R_2|\cdots|\mathbf{r}_N\in R_N)=\int_{R_1}d^3\mathbf{r}_1\int_{R_2}d^3\mathbf{r}_2\ \cdots\int_{R_N}d^3\mathbf{r}_N|\psi(\mathbf{R})|^2. \quad (3)$$

The many-electron time-independent Schrödinger proves extremely difficult to solve exactly because the solution is a function of $3N$ variables. Approximate solutions can be obtained by making different assumptions of varying complexity and with this complexity usually comes an increase in the computational resources required.

Hartree-Fock method

In quantum mechanics two electrons are fundamentally indistinguishable from one another. This and the fact that they are fermions lead to a requirement that the many-electron wave function is antisymmetric under interchange of particles;

$$\psi(\dots,\mathbf{x}_i,\dots,\mathbf{x}_j,\dots)=-\psi(\dots,\mathbf{x}_j,\dots,\mathbf{x}_i,\dots). \quad (4)$$

Where $\mathbf{x}_i = \{\mathbf{r}_i, s_i\}$ represents the space and spin co-ordinates of the electron. The property in Eq. **??** ensures that any two electrons do not have the same set of quantum numbers and hence obey the Pauli exclusion principle[?] .

The Hartree-Fock approximation[?] [?] [?] is the simplest theory that correctly implements the anti-symmetric properties of the many-electron wave function. The simplest wave function with the required antisymmetry is given by the slater determinant

$$D(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_1(\mathbf{x}_2) & \ldots & \psi_1(\mathbf{x}_N) \\ \psi_2(\mathbf{x}_1) & \psi_2(\mathbf{x}_2) & \ldots & \psi_2(\mathbf{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ \psi_N(\mathbf{x}_1) & \psi_N(\mathbf{x}_2) & \ldots & \psi_N(\mathbf{x}_N) \end{vmatrix}. \qquad (5)$$

The simple antisymmetric slater determinant, Eq. **??**, is a starting point for finding the ground state. It can be used as a variational trial function and optimised by minimising the expectation value of the hamiltonian, H with respect to each of the orbitals $\psi_i(\mathbf{r})$. This leads to a set of N self-consistent Hartree-Fock equations describing the single electron orbitals. The solutions to these behave as though each electron is subjected to the mean-field created by all of the other electrons.

This single determinant method includes the effects of the antisymmetry property of the many-electron wave function, Eq. **??**, but neglects the electronic correlation arising from the electron-electron Coulomb repulsion. Correlation energies are only a small fraction of the total energy but are still important when considering binding energies[?] . This correlation energy can be accounted for by using a linear combination of slater determinants. However, this leads to another problem as a very large number of determinants are required to accurately describe a many-electron wave function. For a full configuration-interaction calculation the number of determinants required is seen to scale exponentially with system size[?] . This is a big problem for computational resources and other, more efficient methods must be considered to calculate ground state wave functions and energies.[i]

[i]One such popular method is Density Functional Theory (DFT). This has favourable $N^2 - N^3$ scaling, but often fails for strongly correlated systems and non-local correlation phenomena

Diffusion Monte Carlo and the Fermion Sign Problem

A starting point for all "projector" quantum Monte Carlo methods[?] is to transform the time-dependent Schrödinger equation into imaginary time, $\tau = it$. The so-called imaginary-time Schrödinger equation is then,

$$\frac{\partial \psi}{\partial \tau} = -H\psi, \tag{6}$$

where ψ is the time-dependent many-electron wave function. A formal solution to the imaginary-time Schrödinger equation can be written,

$$\psi(\tau_1 + \delta\tau) = e^{-H\delta\tau}\psi(\tau_1). \tag{7}$$

The state ψ evolves in imaginary time over a duration of $\delta\tau$. If the initial state, $\psi(\tau_1)$, is expanded as a linear combination of ordered energy eigenstates, ϕ_i, with $\epsilon_0 < \epsilon_1 < \cdots < \epsilon_\infty$, then

$$\psi(\delta\tau) = \sum_{i=0}^{\infty} c_i e^{-\epsilon_i \delta\tau}\phi_i. \tag{8}$$

From this it is then obvious that any initial state, ψ, that is not orthogonal to the ground-state, ϕ_0, will evolve to the ground state in the long-time limit,

$$\lim_{\tau\to\infty} \psi = c_0 e^{-\epsilon_0\delta\tau}\phi_0. \tag{9}$$

The ground-state is therefore projected out. The aim of all Monte Carlo projector methods, such as Diffusion Monte Carlo (DMC)[?] , is to perform a stochastic long-time integration of Eq. **??** in order to retrieve the ground-state.

If the hamiltonian in Eq. **??** is expanded in terms of the kinetic energy and potential terms[i], the imaginary-time Schrödinger equation takes on a form that

such as van der Waals interactions[?] .

[i]An adjustable constant energy offset, E_T, is introduced in order to keep Eq. **??** finite.

is analogous to the diffusion equation:

$$-\frac{\partial}{\partial\tau}\psi(\mathbf{R},\tau) = \left[-\frac{1}{2}\sum_{i=1}^{N}\nabla^2_{\mathbf{r}_i} + V(\mathbf{R}) - E_T\right]\psi(\mathbf{R},\tau). \tag{10}$$

Here, $\psi(\mathbf{R},\tau)$ may be interpreted as the density of diffusing particles and $(V(\mathbf{R})-E_T)$ as a rate term describing a potential-dependent change in the particle density. In the Monte Carlo literature these particles are known as "walkers".

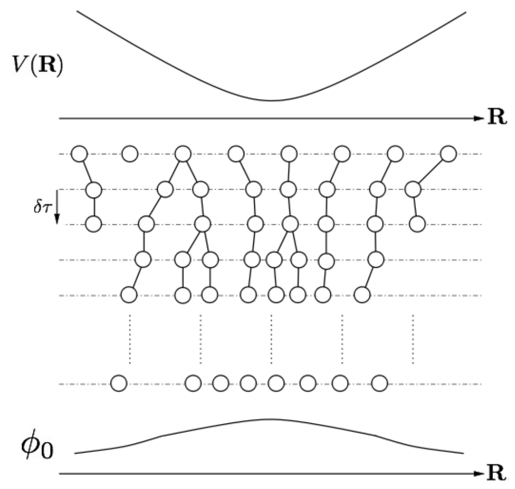


Figure 1: The diffusion of walkers through **R**-space. An initially even distribution of walkers is propagated through imaginary time, τ. Under the influence of the potential $V(\mathbf{R})$, the distribution eventually evolves into what is representative of the ground state many-electron wave function $\psi(\mathbf{R})$.

It is well known that the only major obstacle preventing the exact solution of the many-electron Schrödinger equation via stochastic methods, such as DMC, is the fermion sign problem. This problem arises from the antisymmetry of many-electron wave functions, Eq. **??**. As the Schrödinger equation becomes a diffusion equation in imaginary time, Eq. **??**, its lowest energy solution is generally symmetric and nodeless, which directly disagrees with fermion antisymmetry. Stochastic propagation leads directly to this undesired solution in a process known as the "boson catastrophe"[?] .

One way of trying to prevent the boson catastrophe is the fixed-node approximation[?][?]. In this, a trial function is used to predict positive and negative regions of the wave function, thus defining a nodal surface. If a walker attempts to cross a node then it is rejected, hence constraining the propagation to disjoint areas of propagation. This procedure would be exact if the applied nodal boundaries coincided with the exact nodal hypersurface of the ground-state electronic wave function[?]. It has in fact proven very difficult to determine accurate fixed-node surfaces for large systems.

Full Configuration-Interaction Quantum Monte Carlo

In 2009 Booth, Alavi and Thom[?] introduced a new quantum Monte Carlo method for the simulation of correlated many-electron systems in full configuration-interaction spaces. The new method is designed to simulate the many-electron imaginary-time Schrödinger using a set of walkers that inhabit Slater determinant space, $\{D_i\}$, and evolve under the guidance of a simple set of rules; spawning, cloning, death and annihilation. The inventors of FCIQMC note three key elements of the algorithm;

1. As with DMC a long-time integration of the imaginary-time Schrödinger equation is performed, however it is achieved in a space of Slater determinants
2. In a similar way to DMC the instantaneous wave function is represented by walkers instead of amplitude coefficients, allowing the description of the FCI wave function stochastically, without storing all amplitudes simultaneously
3. Walkers carry a positive or negative sign which allows for annihilation, where two walkers of opposite sign are destroyed if they coincide on the same determinant

In order to satisfy these three key elements a long-time integration must be performed on an analogue of Eq. **??** that is expressed in a Slater determinant basis. A set of coupled first-order differential equations for the FCI coefficients are derived? :

$$-\frac{dC_i}{d\tau} = (K_{ii} - S)C_i + \sum_{j \neq i} K_{ij} C_j, \tag{11}$$

where

$$K_{ij} \equiv \langle D_i | K | D_j \rangle = \langle D_i | H | D_j \rangle - E_{HF}\delta_{ij}, \tag{12}$$

where S is an adjustable energy offset[i], C_j is the coefficient of the jth determinant in the Slater determinant expansion of the wave function (CI coefficients) and E_{HF} is the Hartree-Fock energy.

The population dynamics algorithm? ? ? simulates the set of differential equations, Eq. **??**. The algorithm consists of three steps, which are performed at each time step of length $\delta\tau$:

1. Spawning: Each walker located on D_i attempts to spawn another walker on a connected determinant D_j with probability

$$P_s(D_j|D_i) = \frac{\delta\tau |K_{ij}|}{P_{gen}(D_j|D_i)}, \tag{13}$$

   where $P_{gen}(D_j|D_i)$ is the calculated probability of generating D_j[ii]. The sign of the spawned walker is the same as the parent if $K_{ij} < 0$ and is opposite otherwise.

2. Diagonal death/cloning: For each parent (pre-existing walker) compute

$$p_d(D_i) = \delta\tau(K_{ii} - S). \tag{14}$$

   If $p_d > 0$ the walker dies (immediately removed from simulation) with probability p_d otherwise it is cloned with probability $|p_d|$.

3. Annihilation: run over all remaining walkers and annihilate all opposite-sign pairs that occupy the same point in antisymmetric Slater determinant

[i]Similar to in DMC this energy offset is introduced to keep the population under control
[ii]Booth *et al.*? describe how to calculate P_{gen}

space.

Such an algorithm converges on the exact fermonic ground-state of the Hamiltonian in the Slater determinant basis? . So in this way, it prevents convergence to a bosonic solution, which is a major problem in other QMC methods. Moreover, FCIQMC simulations do not require any *a priori* information and hence removes the hassle of trying to calculate a nodal structure as is inherent in DMC.

Appendix B: C_h spline derivative estimator

This appendix includes more detail on the calculation of the heat capacity in a uniform magnetic field using the derivative of the DMQMC finite-temperature energy estimator.

Smoothing spline fit

In Sec. **??** it was found that the heat capacity in a uniform magnetic field could be expressed as

$$C_h = -\beta^2 \frac{d\langle H \rangle}{d\beta}. \tag{15}$$

The problem here is that the $\langle H \rangle$ data contains statistical noise and it thus proves difficult to obtain an accurate approximation of the gradient function. However, it is possible to fit a smooth curve to the data using a smoothing spline**?** . A spline is essentially a piece-wise polynomial. The data is split into regions and each region is approximated by a polynomial, usually a cubic.

For a sequence of measurements M_i that can be modelled as $M_i = F(x_i)$, where $\{x_i | x_1 < x_2 < \cdots < x_N, x_i \in \mathcal{Z}\}$ and F is some function, the smoothing spline estimate $\bar{F}$ of F is defined as the function that minimises

$$\sum_{i=1}^{n}(M_i - \bar{F}(x_i))^2 + \eta \int_{x_1}^{x_N} \bar{F}''(x)^2 dx. \tag{16}$$

Here $\eta \geq 0$ is the smoothing parameter which controls the smoothing characteristics $\bar{F}$. For $\eta \to 0$ $\bar{F}$ becomes an interpolating spline and for $\eta \to \infty$ $\bar{F}$ converges to a linear least squares estimate.

`Python` has a smoothing spline implementation, `splev`, in the `scipy.interpolate` sub-package. In this the smoothing parameter is chosen so that it is between $N - \sqrt{2N} < \eta < N + \sqrt{2N}$ and the best fit of the data can be found by trial and error. The `splev` function can also return derivatives of $\bar{F}$, which proved useful for calculating the C_h.

So during this investigation the DMQMC spline derivative estimate of C_h was determined in a post-processing step. This step was inelegant as it relied on some trial and error and it was difficult to obtain errors.

Appendix C: Code

This appendix presents some of the main components of the DMQMC implementation in `FORTRAN 90` that were developed during this project. The `dmqmc` module in Code 1 contains the main outline of the algorithm as presented in Sec. **??**. `dmqmc_procedures.f90` and `dmqmc_estimators.f90` in Code 2 and Code 3 contain some important general procedures and the implementation of the estimators respectively.

Code 1: dmqmc.f90

```
module dmqmc

! Main loop for performing DMQMC calculations, similar to the
! file fciqmc.f90 which carries out the main loop in FCIQMC
! calculations.

use fciqmc_data
use proc_pointers
implicit none

contains

   subroutine do_dmqmc()

      ! Run DMQMC calculation. We run from a beta=0 to a value of beta
      ! specified by the user and then repeat this main loop beta_loops
      ! times, to accumulate statistics for each value for beta.

      use parallel
      use annihilation, only: direct_annihilation
      use basis, only: basis_length, bit_lookup, nbasis
```

```fortran
        use death, only: stochastic_death
        use determinants, only: det_info, alloc_det_info, dealloc_det_info
        use dmqmc_procedures, only: random_distribution_heisenberg
        use dmqmc_procedures, only: update_sampling_weights, output_and_alter_weights
        use dmqmc_estimators, only: update_dmqmc_estimators, call_dmqmc_estimators
        use dmqmc_estimators, only: call_rdm_procedures
        use excitations, only: excit, get_excitation_level
        use fciqmc_common
        use fciqmc_restart, only: dump_restart
        use interact, only: fciqmc_interact
        use system, only: nel
        use calc, only: seed, doing_dmqmc_calc, dmqmc_energy
        use calc, only: dmqmc_staggered_magnetisation, dmqmc_energy_squared
        use dSFMT_interface, only: dSFMT_init
        use utils, only: int_fmt

        integer :: idet, ireport, icycle, iparticle, iteration
        integer :: beta_cycle
        integer(lint) :: nparticles_old(sampling_size)
        integer(lint) :: nattempts, nparticles_start_report
        type(det_info) :: cdet1, cdet2
        integer :: nspawned, ndeath
        type(excit) :: connection
        integer :: spawning_end
        logical :: soft_exit
        real :: t1, t2

        ! Allocate det_info components. We need two cdet objects
        ! for each 'end' which may be spawned from in the DMQMC algorithm.
        call alloc_det_info(cdet1, .false.)
        call alloc_det_info(cdet2, .false.)

        ! Main DMQMC loop.
        if (parent) call write_fciqmc_report_header()
        ! Initialise timer.
        call cpu_time(t1)

        initial_shift = shift
        ! When we accumulate data throughout a run, we are actually accumulating
        ! results from the psips distribution from the previous iteration.
        ! For example, in the first iteration, the trace calculated will be that
        ! of the initial distribution, which corresponds to beta=0. Hence, in the
        ! output we subtract one from the iteration number, and run for one more
        ! report loop, asimplemented in the line of code below.
```

```fortran
    nreport = nreport+1

    do beta_cycle = 1, beta_loops
        ! Reset the current position in the spawning array to be the
        ! slot preceding the first slot.
        spawning_head = spawning_block_start
        tot_walkers = 0
        shift = initial_shift
        nparticles = 0
        if (allocated(reduced_density_matrix)) reduced_density_matrix = 0
        if (dmqmc_vary_weights) dmqmc_accumulated_probs = 1.0_p
        if (dmqmc_find_weights) excit_distribution = 0
        vary_shift = .false.

        ! Need to place psips randomly along the diagonal at the
        ! start of every iteration. Pick orbitals randomly, each
        ! with equal probability, so that when electrons are placed
        ! on these orbitals they will have the correct spin and symmetry.
        call dmqmc_initial_distribution_ptr()

        call direct_annihilation()

        if (beta_cycle .ne. 1) then
            ! Reset the random number generator with seed = seed + nprocs
            seed = seed + nprocs
            call dSFMT_init(seed + iproc)
            if (parent) then
                write (6,'(a32,i7)') &
                    " # Resetting beta... Beta loop =", beta_cycle
                write (6,'(a52,'//int_fmt(seed,1)//',a1)') &
                    " # Resetting random number generator with a seed of:", seed, "."
            end if
        end if

        nparticles_old = nint(D0_population)

        do ireport = 1, nreport
            ! Zero report cycle quantities.
            rspawn = 0.0_p
            trace = 0
            estimator_numerators = 0
            if (calculate_excit_distribution) excit_distribution = 0
            nparticles_start_report = nparticles_old(1)

            do icycle = 1, ncycles
                spawning_head = spawning_block_start
                iteration = (ireport-1)*ncycles + icycle

                ! Number of spawning attempts that will be made.
                ! Each particle and each end gets to attempt to
                ! spawn onto a connected determinant and a chance
                ! to die/clone.
                nattempts = 4*nparticles(1)

                ! Reset death counter
                ndeath = 0

                do idet = 1, tot_walkers ! loop over walkers/dets

                    ! f points to the bitstring that is spawning, f2 to the
                    ! other bit string.
                    cdet1%f => walker_dets(:basis_length,idet)
                    cdet1%f2 => walker_dets((basis_length+1):(2*basis_length),idet)
                    cdet2%f => walker_dets((basis_length+1):(2*basis_length),idet)
                    cdet2%f2 => walker_dets(:basis_length,idet)

                    ! Decode and store the the relevant information for
                    ! both bitstrings. Both of these bitstrings are required
                    ! to refer to the correct element in the density matrix.
                    call decoder_ptr(cdet1%f, cdet1)
                    call decoder_ptr(cdet2%f, cdet2)

                    ! Call wrapper function which calls all requested estimators
                    ! to be updated, and also always updates the trace separately.

                    if (icycle == 1) call call_dmqmc_estimators(idet, iteration)

                    do iparticle = 1, abs(walker_population(1,idet))
                        ! Spawn from the first end.
                        spawning_end = 1
                        ! Attempt to spawn.
                        call spawner_ptr(cdet1, walker_population(1,idet), nspawned, connection)
                        ! Spawn if attempt was successful.
                        if (nspawned /= 0) then
                            call create_spawned_particle_dm_ptr(cdet1%f, cdet2%f, connection, nspawned,
                                  spawning_end)
                        end if

```

```fortran
                     ! Now attempt to spawn from the second end.
                     spawning_end = 2
                     call spawner_ptr(cdet2, walker_population(1,idet), nspawned, connection)
                     if (nspawned /= 0) then
                        call create_spawned_particle_dm_ptr(cdet2%f, cdet1%f, connection, nspawned, spawning_end)
                     end if
                  end do

                  ! Clone or die.
                  ! We have contirbutions to the clone/death step from both ends of the
                  ! current walker. We do both of these at once by using walker_data(1,idet)
                  ! which, when running a DMQMC algorithm, stores the average of the two diagonal
                  ! elements corresponding to the two indicies of the density matrix (the two ends).
                  call stochastic_death(walker_data(1,idet), walker_population(1,idet), nparticles(1), ndeath)
               end do

               ! Add the spawning rate (for the processor) to the running
               ! total.
               rspawn = rspawn + spawning_rate(ndeath, nattempts)

               ! Perform the annihilation step where the spawned walker list is merged with
               ! the main walker list, and walkers of opposite sign on the same sites are
               ! annihilated.
               call direct_annihilation()

               ! If doing importance sampling *and* varying the weights of the trial function, call a routine
               ! to update these weights and alter the number of psips on each excitation level accordingly.
               if (dmqmc_vary_weights .and. iteration <= finish_varying_weights) call update_sampling_weights()

            end do

            ! If averaging the shift to use in future beta loops, add contirubtion from this report.
            if (average_shift_until > 0) shift_profile(ireport) = shift_profile(ireport) + shift

            ! Update the shift and desired thermal quantites.
            call update_dmqmc_estimators(nparticles_old, ireport)

            call cpu_time(t2)

```

```
                ! t1 was the time at the previous iteration, t2 the current time.
                ! t2-t1 is thus the time taken by this report loop.
                if (parent) call write_fciqmc_report(ireport, nparticles_start_report, t2-t1)

                ! cpu_time outputs an elapsed time, so update the reference timer.
                t1 = t2

                call fciqmc_interact(soft_exit)
                if (soft_exit) exit

            end do

            if (soft_exit) exit

            ! If have just finished last beta loop of accumulating the shift, then perform
            ! the averaging and set average_shift_until to -1. This tells the shift update
            ! algorithm to use the values for shift stored in shift_profile.
            if (beta_cycle == average_shift_until) then
                shift_profile = shift_profile/average_shift_until
                average_shift_until = -1
            end if

            ! Calculate and output all requested estimators based on the reduced dnesity matrix.
            if (doing_reduced_dm) call call_rdm_procedures()
            ! Calculate and output new weights based on the psip distirubtion in the previous loop.
            if (dmqmc_find_weights) call output_and_alter_weights()
        end do

        if (parent) then
            call write_fciqmc_final(ireport)
            write (6,'()')
        end if

        call load_balancing_report()

        if (dump_restart_file) call dump_restart(mc_cycles_done+ncycles*nreport, nparticles_old(1))

        call dealloc_det_info(cdet1, .false.)
        call dealloc_det_info(cdet2, .false.)

    end subroutine do_dmqmc

end module dmqmc
```

Code 2: dmqmc_procedures.f90

```fortran
module dmqmc_procedures

use const
implicit none

contains

   subroutine init_dmqmc()

       use basis, only: basis_length, total_basis_length, bit_lookup, basis_lookup
       use calc, only: doing_dmqmc_calc, dmqmc_calc_type, dmqmc_energy, dmqmc_energy_squared
       use calc, only: dmqmc_staggered_magnetisation, dmqmc_correlation
       use checking, only: check_allocate, check_deallocate
       use fciqmc_data, only: trace, energy_index, energy_squared_index, correlation_index
       use fciqmc_data, only: staggered_mag_index, estimator_numerators, subsystem_A_size
       use fciqmc_data, only: subsystem_A_mask, subsystem_B_mask, subsystem_A_bit_positions
       use fciqmc_data, only: subsystem_A_list, dmqmc_factor, number_dmqmc_estimators, ncycles
       use fciqmc_data, only: reduced_density_matrix, doing_reduced_dm, tau, dmqmc_weighted_sampling
       use fciqmc_data, only: correlation_mask, correlation_sites, half_density_matrix
       use fciqmc_data, only: dmqmc_sampling_probs, dmqmc_accumulated_probs, flip_spin_matrix
       use fciqmc_data, only: doing_concurrence, calculate_excit_distribution, excit_distribution
       use fciqmc_data, only: nreport, average_shift_until, shift_profile, dmqmc_vary_weights
       use fciqmc_data, only: finish_varying_weights, weight_altering_factors, dmqmc_find_weights
       use parallel, only: parent
       use system, only: system_type, heisenberg, nsites, max_number_excitations

       integer :: ierr
       integer :: i, ipos, basis_find, bit_position, bit_element

       number_dmqmc_estimators = 0
       trace = 0

       if (doing_dmqmc_calc(dmqmc_energy)) then
          number_dmqmc_estimators = number_dmqmc_estimators + 1
          energy_index = number_dmqmc_estimators
       end if
       if (doing_dmqmc_calc(dmqmc_energy_squared)) then
          number_dmqmc_estimators = number_dmqmc_estimators + 1
          energy_squared_index = number_dmqmc_estimators
       end if
       if (doing_dmqmc_calc(dmqmc_correlation)) then
          number_dmqmc_estimators = number_dmqmc_estimators + 1
          correlation_index = number_dmqmc_estimators
```

```
            allocate(correlation_mask(1:basis_length), stat=ierr)
            call check_allocate('correlation_mask',basis_length,ierr)
            correlation_mask = 0
            do i = 1, 2
                bit_position = bit_lookup(1,correlation_sites(i))
                bit_element = bit_lookup(2,correlation_sites(i))
                correlation_mask(bit_element) = ibset(correlation_mask(bit_element), bit_position)
            end do
        end if
        if (doing_dmqmc_calc(dmqmc_staggered_magnetisation)) then
            number_dmqmc_estimators = number_dmqmc_estimators + 1
            staggered_mag_index = number_dmqmc_estimators
        end if

        allocate(estimator_numerators(1:number_dmqmc_estimators), stat=ierr)
        call check_allocate('estimator_numerators',number_dmqmc_estimators,ierr)
        estimator_numerators = 0

        if (calculate_excit_distribution .or. dmqmc_find_weights) then
            allocate(excit_distribution(0:max_number_excitations), stat=ierr)
            call check_allocate('excit_distribution',max_number_excitations+1,ierr)
            excit_distribution = 0.0_p
        end if

        if (average_shift_until > 0) then
            allocate(shift_profile(1:nreport+1), stat=ierr)
            call check_allocate('shift_profile',nreport+1,ierr)
            shift_profile = 0.0_p
        end if

        ! In DMQMC we want the spawning probabilities to have an extra factor of a half,
        ! because we spawn from two different ends with half probability. To avoid having
        ! to multiply by an extra variable in every spawning routine to account for this, we
        ! multiply the time step by 0.5 instead, then correct this in the death step (see below).
        tau = tau*0.5_p
        ! Set dmqmc_factor to 2 so that when probabilities in death.f90 are multiplied
        ! by this factor it cancels the factor of 0.5 introduced into the timestep in DMQMC.
        ! Every system uses the same death routine, so this factor only needs to be added once.
        ! This factor is also used in updated the shift, where the true tau is needed.
        dmqmc_factor = 2.0_p

        if (dmqmc_weighted_sampling) then
            ! dmqmc_sampling_probs stores the factors by which probabilities are to
            ! be reduced when spawning away from the diagonal. The trial function required
```

```fortran
            ! from these probabilities, for use in importance sampling, is actually that of
            ! the accumulated factors, ie, if dmqmc_sampling_probs = (a, b, c, ...) then
            ! dmqmc_accumulated_factors = (1, a, ab, abc, ...). This is the array which we
            ! need to create and store. dmqmc_sampling_probs is no longer needed and so can
            ! be deallocated. Also, the user may have only input factors for the first few
            ! excitation levels, but we need to store factors for all levels, as done below.
            if (.not.allocated(dmqmc_sampling_probs)) then
               allocate(dmqmc_sampling_probs(1:max_number_excitations), stat=ierr)
               call check_allocate('dmqmc_sampling_probs',max_number_excitations,ierr)
               dmqmc_sampling_probs = 1.0_p
            end if
            if (half_density_matrix) dmqmc_sampling_probs(1) = dmqmc_sampling_probs(1)*2.0_p
            allocate(dmqmc_accumulated_probs(0:max_number_excitations), stat=ierr)
            call check_allocate('dmqmc_accumulated_probs',max_number_excitations+1,ierr)
            dmqmc_accumulated_probs(0) = 1.0_p
            do i = 1, size(dmqmc_sampling_probs)
               dmqmc_accumulated_probs(i) = dmqmc_accumulated_probs(i-1)*dmqmc_sampling_probs(i)
            end do
            dmqmc_accumulated_probs(size(dmqmc_sampling_probs)+1:max_number_excitations) = &
                                 dmqmc_accumulated_probs(size(dmqmc_sampling_probs))
            if (dmqmc_vary_weights) then
               ! Allocate an array to store the factors by which the weights will change each
               ! iteration.
               allocate(weight_altering_factors(0:max_number_excitations), stat=ierr)
               call check_allocate('weight_altering_factors',max_number_excitations+1,ierr)
               weight_altering_factors = dble(dmqmc_accumulated_probs)**(1/dble(finish_varying_weights))
               ! If varying the weights, start the accumulated probabilties as all 1.0
               ! initially, and then alter them gradually later.
               dmqmc_accumulated_probs = 1.0_p
            end if
         else
            allocate(dmqmc_accumulated_probs(0:max_number_excitations), stat=ierr)
            call check_allocate('dmqmc_accumulated_probs',max_number_excitations+1,ierr)
            dmqmc_accumulated_probs = 1.0_p
            if (half_density_matrix) dmqmc_accumulated_probs(1:max_number_excitations) &
                                  = 2.0_p*dmqmc_accumulated_probs(1:max_number_excitations)
         end if

         ! If doing a reduced density matrix calculation, then allocate and define the
         ! bit masks that have 1's at the positions referring to either subsystems A or B.
         if (doing_reduced_dm) then
            subsystem_A_size = ubound(subsystem_A_list,1)
            allocate(subsystem_A_mask(1:basis_length), stat=ierr)
            call check_allocate('subsystem_A_mask',basis_length,ierr)
            allocate(subsystem_B_mask(1:basis_length), stat=ierr)
            call check_allocate('subsystem_B_mask',basis_length,ierr)
            allocate(subsystem_A_bit_positions(subsystem_A_size,2), stat=ierr)
            call check_allocate('subsystem_A_bit_positions',2*subsystem_A_size,ierr)
            subsystem_A_mask = 0
            subsystem_B_mask = 0
            subsystem_A_bit_positions = 0
            ! For the Heisenberg model only currently.
            if (system_type==heisenberg) then
                do i = 1, subsystem_A_size
                    bit_position = bit_lookup(1,subsystem_A_list(i))
                    bit_element = bit_lookup(2,subsystem_A_list(i))
                    subsystem_A_mask(bit_element) = ibset(subsystem_A_mask(bit_element), bit_position)
                    subsystem_A_bit_positions(i,1) = bit_position
                    subsystem_A_bit_positions(i,2) = bit_element
                end do
                subsystem_B_mask = subsystem_A_mask
                ! We cannot just flip the mask for system A to get that for system B, because
                ! there the trailing bits on the end don't refer to anything and should be
                ! set to 0. So, first set these to 1 and then flip all the bits.
                do ipos = 0, i0_end
                    basis_find = basis_lookup(ipos, basis_length)
                    if (basis_find == 0) then
                        subsystem_B_mask(basis_length) = ibset(subsystem_B_mask(basis_length),ipos)
                    end if
                end do
                subsystem_B_mask = not(subsystem_B_mask)
            end if
            if (subsystem_A_size <= int(nsites/2)) then
                ! In this case, for an ms = 0 subspace (as the ground state of the Heisenberg model
                ! will be) then any combination of spins can occur in the subsystem, from all spins
                ! down to all spins up. Hence the total size of the reduced density matrix will be
                ! 2**(number of spins in subsystem A).
                allocate(reduced_density_matrix(2**subsystem_A_size,2**subsystem_A_size), stat=ierr)
                call check_allocate('reduced_density_matrix', 2**(2*subsystem_A_size),ierr)
                reduced_density_matrix = 0
            else if (subsystem_A_size == nsites) then
                allocate(reduced_density_matrix(2**subsystem_A_size,2**subsystem_A_size), stat=ierr)
                call check_allocate('reduced_density_matrix', 2**(2*subsystem_A_size),ierr)
                reduced_density_matrix = 0
            end if
        end if

        ! If doing concurrence calculation then construct and store the 4x4 flip spin matrix i.e.
        ! \sigma_y \otimes \sigma_y

        if (doing_concurrence) then
            allocate(flip_spin_matrix(4,4), stat=ierr)
            call check_allocate('flip_spin_matrix', 16,ierr)
            flip_spin_matrix = 0._p
            flip_spin_matrix(1,4) = -1._p
            flip_spin_matrix(4,1) = -1._p
            flip_spin_matrix(3,2) = 1._p
            flip_spin_matrix(2,3) = 1._p
        end if

    end subroutine init_dmqmc

    subroutine random_distribution_heisenberg()

       ! For the Heisenberg model only. Distribute the initial number of psips
       ! along the main diagonal. Each diagonal element should be chosen
       ! with the same probability.

       ! Currently this creates psips with Ms = ms_in only.

       ! If we have number of sites = nsites,
       ! and total spin value = ms_in,
       ! then number of up spins is equal to up_spins = (ms_in + nsites)/2.

       ! Start from state with all spins down, then choose the above number of
       ! spins to flip up with equal probability.

       use basis, only: nbasis, basis_length, bit_lookup
       use calc, only: ms_in
       use dSFMT_interface, only: genrand_real2
       use fciqmc_data, only: D0_population
       use parallel
       use system, only: nsites

       integer :: i, up_spins, rand_basis, bits_set
       integer :: bit_element, bit_position, npsips
       integer(i0) :: f(basis_length)
       real(dp) :: rand_num

       up_spins = (ms_in+nsites)/2
       npsips = int(D0_population/nprocs)
       ! If initial number of psips does not split evenly between all processors,
       ! add the leftover psips to the first processors in order.
       if (D0_population-(nprocs*int(D0_population/nprocs)) > iproc) npsips = npsips+1

       do i = 1, npsips

          ! Start with all spins down.
          f = 0
          bits_set = 0

          do
              ! If half the spins are now flipped up, we have our basis
              ! function fully created, so exit the loop.
              if (bits_set==up_spins) exit
              ! Choose a random spin to flip.
              rand_num = genrand_real2()
              rand_basis = ceiling(rand_num*nbasis)
              ! Find the corresponding positions for this spin.
              bit_position = bit_lookup(1,rand_basis)
              bit_element = bit_lookup(2,rand_basis)
              if (.not. btest(f(bit_element),bit_position)) then
                  ! If not flipped up, flip the spin up.
                  f(bit_element) = ibset(f(bit_element),bit_position)
                  bits_set = bits_set + 1
              end if
          end do

          ! Now call a routine to add the corresponding diagonal element to
          ! the spawned walkers list.
          call create_particle(f,f,1)

       end do

    end subroutine random_distribution_heisenberg

    subroutine create_particle(f1,f2,nspawn)

       ! Create a psip on a diagonal element of the density
       ! matrix by adding it to the spawned walkers list. This
       ! list can then be sorted correctly by the direct_annihilation
       ! routine

       ! In:
       ! f_new: Bit string representation of index of the diagonal
       ! element upon which a new psip shall be placed.
```

```

        use hashing
        use basis, only: basis_length, total_basis_length
        use fciqmc_data, only: spawned_walkers, spawning_head, spawned_pop
        use parallel

        integer(i0), intent(in) :: f1(basis_length), f2(basis_length)
        integer, intent(in) :: nspawn
        integer(i0) :: f_new(total_basis_length)
#ifndef PARALLEL
        integer, parameter :: iproc_spawn = 0
#else
        integer :: iproc_spawn
#endif

        ! Create the bitstring of the psip.
        f_new = 0
        f_new(:basis_length) = f1
        f_new((basis_length+1):(total_basis_length)) = f2

#ifdef PARALLEL
        ! Need to determine which processor the spawned walker should be sent to.
        iproc_spawn = modulo(murmurhash_bit_string(f_new, &
                              (total_basis_length)), nprocs)
#endif

        ! Move to the next position in the spawning array.
        spawning_head(iproc_spawn) = spawning_head(iproc_spawn) + 1

        ! Set info in spawning array.
        ! Zero it as not all fields are set.
        spawned_walkers(:,spawning_head(iproc_spawn)) = 0
        ! indices 1 to total_basis_length store the bitstring.
        spawned_walkers(:(2*basis_length),spawning_head(iproc_spawn)) = f_new
        ! The final index stores the number of psips created.
        spawned_walkers((2*basis_length)+1,spawning_head(iproc_spawn)) = nspawn

    end subroutine create_particle

    subroutine decode_dm_bitstring(f, index1, index2)

        ! This function maps an input DMQMC bitstring to two indices
        ! giving the corresponding position of the bitstring in the reduced
        ! density matrix.
```

```fortran

        use basis, only: total_basis_length, basis_length
        use fciqmc_data, only: subsystem_A_bit_positions, subsystem_A_bit_positions
        use fciqmc_data, only: subsystem_A_size

        integer(i0), intent(in) :: f(total_basis_length)
        integer(i0), intent(out) :: index1, index2
        integer :: i

        ! Start from all bits down, so that we can flip bits up one by one.
        index1 = 0
        index2 = 0

        ! Loop over all the sites in the sublattice considered for the reduced density matrix.
        do i = 1, subsystem_A_size
           ! If the spin is up, flip the corresponding bit in the first index up.
           if (btest(f(subsystem_A_bit_positions(i,2)),subsystem_A_bit_positions(i,1))) &
              index1 = ibset(index1,i-1)
           ! Similarly for the second index, by looking at the second end of the bitstring.
           if (btest(f(subsystem_A_bit_positions(i,2)+basis_length),subsystem_A_bit_positions(i,1))) &
              index2 = ibset(index2,i-1)
        end do

        ! The process above maps to numbers between 0 and 2^subsystem_A_size-1, but the smallest
        ! and largest of the reduced density matrix are one more than these, so add one...
        index1 = index1+1
        index2 = index2+1

    end subroutine decode_dm_bitstring

    subroutine update_sampling_weights()

        ! This routine updates the values of the weights used in importance sampling. It also
        ! removes or adds psips from the various levels accordingly.

        ! In:
        ! iteration: The current iteration in the beta loop.

        use annihilation, only: remove_unoccupied_dets
        use basis, only: basis_length, total_basis_length
        use excitations, only: get_excitation_level
        use fciqmc_data, only: dmqmc_accumulated_probs, finish_varying_weights
        use fciqmc_data, only: weight_altering_factors, tot_walkers, walker_dets, walker_population
        use fciqmc_data, only: nparticles
        use dSFMT_interface, only: genrand_real2

        integer :: idet, excit_level, nspawn, sign_factor, old_population
        real(p) :: new_factor
        real(dp) :: rand_num, prob

        ! Alter weights for the next iteration.
        dmqmc_accumulated_probs = dble(dmqmc_accumulated_probs)*weight_altering_factors

        ! When the weights for an excitation level are increased by a factor, the number
        ! of psips on that level has to decrease by the same factor, else the wavefunction
        ! which the psips represent will not be the correct importance sampled wavefunction
        ! for the new weights. The code below loops over every psips and destorys (or creates)
        ! it with the appropriate probability.
        do idet = 1, tot_walkers
            excit_level = get_excitation_level(walker_dets(1:basis_length,idet),&
                    walker_dets(basis_length+1:total_basis_length,idet))
            old_population = abs(walker_population(1,idet))
            rand_num = genrand_real2()
            ! If weight_altering_factors(excit_level) > 1, need to kill psips.
            ! If weight_altering_factors(excit_level) < 1, need to create psips.
            prob = abs(1.0_dp - weight_altering_factors(excit_level)**(-1))*old_population
            nspawn = int(prob)
            prob = prob - nspawn
            if (rand_num < prob) nspawn = nspawn + 1
            if (weight_altering_factors(excit_level) > 1.0_dp) then
                sign_factor = -1
            else
                sign_factor = +1
            end if
            nspawn = sign(nspawn,walker_population(1,idet)*sign_factor)
            ! Update the population on this determinant.
            walker_population(1,idet) = walker_population(1,idet) + nspawn
            ! Update the total number of walkers
            nparticles(1) = nparticles(1) - old_population + abs(walker_population(1,idet))
        end do

        ! Call the annihilation routine to update the main walker list, as some
        ! sites will now have no psips on and so need removing from the simulation.
        call remove_unoccupied_dets()

    end subroutine update_sampling_weights

    subroutine output_and_alter_weights()
```

```

        ! This routine will alter and output the sampling weights used in importance
        ! sampling. It uses the excitation distribution, calculated on the beta loop
        ! which has just finished, and finds the weights needed so that each excitation
        ! level will have roughly equal numbers of psips in the next loop. For example,
        ! to find the weights of psips on the 1st excitation level, divide the number of
        ! psips on the 1st excitation level by the number on the 0th level, then multiply
        ! the old sampling weight by this number to give the new weight. This can be used
        ! when the weights are being introduced gradually each beta loop, too. The weights
        ! are output and can then be used in future DMQMC runs.

        use fciqmc_data, only: dmqmc_sampling_probs, dmqmc_accumulated_probs
        use fciqmc_data, only: excit_distribution, finish_varying_weights
        use fciqmc_data, only: dmqmc_vary_weights, weight_altering_factors
        use parallel
        use system, only: max_number_excitations

        integer :: i, ierr
#ifdef PARALLEL
        real(p) :: merged_excit_dist(max_number_excitations)
        call mpi_allreduce(excit_distribution, merged_excit_dist, max_number_excitations, &
           MPI_REAL8, MPI_SUM, MPI_COMM_WORLD, ierr)

        excit_distribution = merged_excit_dist
#endif

        ! It is assumed that there is an even maximum number of excitations.
        do i = 1, (max_number_excitations/2)
           ! Don't include levels where there are very few psips accumulated.
           if (excit_distribution(i-1) > 10.0_p .and. excit_distribution(i) > 10.0_p) then
              ! Alter the sampling weights using the relevant excitation distribution.
              dmqmc_sampling_probs(i) = dmqmc_sampling_probs(i)*&
                 (excit_distribution(i)/excit_distribution(i-1))
              dmqmc_sampling_probs(max_number_excitations+1-i) = dmqmc_sampling_probs(i)**(-1)
           end if
        end do

        ! Recalculate dmqmc_accumulated_probs with the new weights.
        do i = 1, max_number_excitations
           dmqmc_accumulated_probs(i) = dmqmc_accumulated_probs(i-1)*dmqmc_sampling_probs(i)
        end do

        ! If dmqmc_vary_weights is true then the weights are to be introduced gradually at the
        ! start of each beta loop. This required redefining weight_altering_factors to coincide
```

```fortran
      ! with the new sampling weights.
      if (dmqmc_vary_weights) then
         weight_altering_factors = dble(dmqmc_accumulated_probs)**(1/dble(finish_varying_weights))
         ! Reset the weights for the next loop.
         dmqmc_accumulated_probs = 1.0_p
      end if

      if (parent) then
         ! Print out weights in a form which can be copied into an input file.
         write(6, '(a31,2X)', advance = 'no') ' # Importance sampling weights:'
         do i = 1, max_number_excitations
            write (6, '(es12.4,2X)', advance = 'no') dmqmc_sampling_probs(i)
         end do
         write (6, '()', advance = 'yes')
      end if

   end subroutine output_and_alter_weights

end module dmqmc_procedures
```

Code 3: dmqmc_estimators.f90

```fortran
module dmqmc_estimators

use const

implicit none

contains

   subroutine update_dmqmc_estimators(ntot_particles_old, ireport)

        ! Update the shift and average the shift and estimators

        ! Should be called every report loop in an DMQMC calculation.

        ! Inout:
        ! ntot_particles_old: total number (across all processors) of
        ! particles in the simulation at end of the previous report loop.
        ! Returns the current total number of particles for use in the
        ! next report loop.

        use calc, only: doing_dmqmc_calc, dmqmc_energy, dmqmc_staggered_magnetisation
        use calc, only: dmqmc_energy_squared
        use checking, only: check_allocate
        use energy_evaluation, only: update_shift
        use fciqmc_data, only: nparticles, sampling_size, target_particles, rspawn
        use fciqmc_data, only: shift, vary_shift, nreport
        use fciqmc_data, only: estimator_numerators, number_dmqmc_estimators
        use fciqmc_data, only: nreport, ncycles, trace, calculate_excit_distribution
        use fciqmc_data, only: excit_distribution, average_shift_until, shift_profile
        use parallel

        integer(lint), intent(inout) :: ntot_particles_old(sampling_size)
        integer, intent(in) :: ireport
        integer(lint) :: ntot_particles(sampling_size)

#ifdef PARALLEL
        real(dp), allocatable :: ir(:)
        real(dp), allocatable :: ir_sum(:)
        integer :: ierr, array_size

        array_size = sampling_size+2+number_dmqmc_estimators
        if (calculate_excit_distribution) array_size = array_size + size(excit_distribution)

```

```
       allocate(ir(1:array_size), stat=ierr)
       call check_allocate('ir',array_size,ierr)
       allocate(ir_sum(1:array_size), stat=ierr)
       call check_allocate('ir_sum',array_size,ierr)

       ! Need to sum the number of particles and other quantites over all processors.
       ir(1:sampling_size) = nparticles
       ir(sampling_size+1) = rspawn
       ir(sampling_size+2) = trace
       ir(sampling_size+3:sampling_size+2+number_dmqmc_estimators) = estimator_numerators
       if (calculate_excit_distribution) ir(sampling_size+3+number_dmqmc_estimators:array_size) = excit_distribution
       ! Merge the lists from each processor together.
       call mpi_allreduce(ir, ir_sum, size(ir), MPI_REAL8, MPI_SUM, MPI_COMM_WORLD, ierr)
       ntot_particles = nint(ir_sum(1:sampling_size),lint)
       rspawn = ir_sum(sampling_size+1)
       trace = nint(ir_sum(sampling_size+2))
       estimator_numerators = ir_sum(sampling_size+3:sampling_size+2+number_dmqmc_estimators)
       if (calculate_excit_distribution) excit_distribution = ir_sum(sampling_size+3+number_dmqmc_estimators:array_size)
#else
       ntot_particles = nparticles
#endif

       ! If average_shift_until = -1 then it means that the shift should be updated to
       ! use the values of shift stored in shift_profile. Otherwise, use the standard update
       ! routine.
       if (average_shift_until == -1) then
           if (ireport < nreport) shift = shift_profile(ireport+1)
       else
           if (vary_shift) call update_shift(ntot_particles_old(1), ntot_particles(1), ncycles)
           if (nparticles(1) > target_particles .and. .not.vary_shift) vary_shift = .true.
       end if

       ntot_particles_old = ntot_particles
       rspawn = rspawn/(ncycles*nprocs)

   end subroutine update_dmqmc_estimators

   subroutine call_dmqmc_estimators(idet, iteration)

      ! This function calls the processes to update the estimators which
      ! have been requested by the user to be calculated.
      ! First, calculate the excitation level between the two bitsrtings
```

```fortran
        ! corresponding to the the two ends. Then add the contribution from
        ! the current density matrix element to the trace, which is always
        ! calculated. Then call other estimators, as required.

        ! In:
        ! idet: Current position in the main bitstring list.
        ! iteration: current Monte Carlo cycle.

        use basis, only: basis_length, total_basis_length
        use calc, only: doing_dmqmc_calc, dmqmc_energy, dmqmc_staggered_magnetisation
        use calc, only: dmqmc_energy_squared, dmqmc_correlation
        use excitations, only: get_excitation, excit
        use fciqmc_data, only: walker_dets, walker_population, trace, doing_reduced_dm
        use fciqmc_data, only: dmqmc_accumulated_probs, start_averaging, dmqmc_find_weights
        use fciqmc_data, only: calculate_excit_distribution, excit_distribution
        use proc_pointers, only: update_dmqmc_energy_ptr, update_dmqmc_stag_mag_ptr
        use proc_pointers, only: update_dmqmc_energy_squared_ptr, update_dmqmc_correlation_ptr

        integer, intent(in) :: idet, iteration
        type(excit) :: excitation
        real(p) :: unweighted_walker_pop

        ! Get excitation.
        excitation = get_excitation(walker_dets(:basis_length,idet), &
                       walker_dets((1+basis_length):total_basis_length,idet))

        ! When performing importance sampling the result is that certain excitation
        ! levels have smaller psips populations than the true density matrix by some
        ! factor. In these cases, we want to multiply the psip population by this factor
        ! to calculate the contribution from these excitation levels correctly.

        ! In the case of no importance sampling, unweighted_walker_pop = walker_population(1,idet)
        ! and so this change can be ignored.
        unweighted_walker_pop = walker_population(1,idet)*dmqmc_accumulated_probs(excitation%nexcit)

        ! If diagonal element, add to the trace.
        if (excitation%nexcit == 0) trace = trace + walker_population(1,idet)
        ! See which estimators are to be calculated, and call the corresponding procedures.
        ! Energy
        if (doing_dmqmc_calc(dmqmc_energy)) call update_dmqmc_energy_ptr&
              &(idet, excitation, unweighted_walker_pop)
        ! Energy squared
        if (doing_dmqmc_calc(dmqmc_energy_squared)) call update_dmqmc_energy_squared_ptr&
              &(idet, excitation, unweighted_walker_pop)
```

```
    ! Spin-spin correlation function
    if (doing_dmqmc_calc(dmqmc_correlation)) call update_dmqmc_correlation_ptr&
          &(idet, excitation, unweighted_walker_pop)
    ! Staggered magnetisation
    if (doing_dmqmc_calc(dmqmc_staggered_magnetisation)) call update_dmqmc_stag_mag_ptr&
          &(idet, excitation, unweighted_walker_pop)
    ! Reduced density matrix
    if (doing_reduced_dm .and. iteration > start_averaging) &
          call update_reduced_density_matrix_heisenberg(idet, unweighted_walker_pop)
    ! Excitation distribution
    if (calculate_excit_distribution) excit_distribution(excitation%nexcit) = &
          excit_distribution(excitation%nexcit) + abs(walker_population(1,idet))
    ! Excitation_distribtuion for calculating importance sampling weights
    if (dmqmc_find_weights .and. iteration > start_averaging) excit_distribution(excitation%nexcit) = &
          excit_distribution(excitation%nexcit) + abs(walker_population(1,idet))

end subroutine call_dmqmc_estimators

subroutine dmqmc_energy_heisenberg(idet, excitation, walker_pop)

    ! For the Heisenberg model only.
    ! Add the contribution from the current density matrix element
    ! to the thermal energy estimate.

    ! In:
    ! idet: Current position in the main bitstring (density matrix) list.
    ! excitation: excit type variable which stores information on
    ! the excitation between the two bitstring ends, corresponding
    ! to the two labels for the density matrix element.
    ! walker_pop: number of particles on the current density matrix
    ! element.

    use basis, only: basis_length, total_basis_length
    use basis, only: bit_lookup
    use excitations, only: excit
    use fciqmc_data, only: walker_dets
    use fciqmc_data, only: walker_data, H00
    use fciqmc_data, only: estimator_numerators, energy_index
    use hubbard_real, only: connected_orbs
    use system, only: J_coupling

    integer, intent(in) :: idet
    type(excit), intent(in) :: excitation
    real(p) , intent(in) :: walker_pop
```

```fortran
        integer :: bit_element, bit_position

        ! If no excitation, we have a diagonal element, so add elements
        ! which involve the diagonal element of the Hamiltonian.
        if (excitation%nexcit == 0) then
           estimator_numerators(energy_index) = estimator_numerators(energy_index) + &
                                  (walker_data(1,idet)+H00)*walker_pop
        else if (excitation%nexcit == 1) then
        ! If not a diagonal element, but only a single excitation, then the corresponding
        ! Hamiltonian element may be non-zero. Calculate if the flipped spins are
        ! neighbours on the lattice, and if so, add the contirbution from this site.
           bit_position = bit_lookup(1,excitation%from_orb(1))
           bit_element = bit_lookup(2,excitation%from_orb(1))
           if (btest(connected_orbs(bit_element, excitation%to_orb(1)), bit_position)) &
                estimator_numerators(energy_index) = estimator_numerators(energy_index) - &
                                 (2.0*J_coupling*walker_pop)
        end if

    end subroutine dmqmc_energy_heisenberg

    subroutine dmqmc_energy_squared_heisenberg(idet, excitation, walker_pop)

        ! For the Heisenberg model only.
        ! Add the contribution from the current density matrix element
        ! to the thermal energy squared estimate.

        ! In:
        ! idet: Current position in the main bitstring (density matrix) list.
        ! excitation: excit type variable which stores information on
        ! the excitation between the two bitstring ends, corresponding
        ! to the two labels for the density matrix element.
        ! walker_pop: number of particles on the current density matrix
        ! element.

        use basis, only: basis_length, total_basis_length
        use basis, only: bit_lookup
        use excitations, only: excit
        use fciqmc_data, only: walker_dets
        use fciqmc_data, only: walker_data, H00
        use fciqmc_data, only: estimator_numerators, energy_squared_index
        use hubbard_real, only: connected_orbs, next_nearest_orbs
        use system, only: J_coupling, nbonds

        integer, intent(in) :: idet
        type(excit), intent(in) :: excitation
        real(p) , intent(in) :: walker_pop
        integer :: bit_element1, bit_position1, bit_element2, bit_position2
        real(p) :: sum_H1_H2, J_coupling_squared

        sum_H1_H2 = 0
        J_coupling_squared = J_coupling**2

        if (excitation%nexcit == 0) then
            ! If there are 0 excitations then either nothing happens twice, or we
            ! flip the same pair of spins twice. The Hamiltonian element for doing nothing
            ! is just the diagonal element. For each possible pairs of spins which can be
            ! flipped, there is a mtarix element of -2*J_coupling, so we just need to count
            ! the number of such pairs, which can be found simply from the diagonal element.

            sum_H1_H2 = (walker_data(1,idet)+H00)**2
            sum_H1_H2 = sum_H1_H2 + 2.0*J_coupling_squared*nbonds + 2.0*J_coupling*(walker_data(1,idet)+H00)

        else if (excitation%nexcit == 1) then
            ! If there is only one excitation (2 spins flipped) then the contribution to H^2
            ! depend on the positions of the spins relative to one another.
            ! If the the spins are nearest neighbors then we could either do nothing and then
            ! flip the pair, or flip the pair and then do nothing.
            ! If next nearest neighbors and there is only one two-bond path to get from one
            ! spin to the other, we first flip the pair on the first bond, then flip the pair
            ! on the second bond. This flipping can only be done in exactly one order, not both -
            ! the two spins which change are opposite, so the middle spin will initially only
            ! be the same as one or the other spin. This is nice, because we don't have check
            ! which way up the intermeddiate spin is - there will always be one order which contributes.
            ! If there are two such paths, then this could happen by either paths, but again, the two
            ! intermeddiate spins will only allow one order of spin flipping for each path, no
            ! matter which way up they are, so we only need to check if there are two possible paths.

            if (next_nearest_orbs(excitation%from_orb(1),excitation%to_orb(1)) /= 0) then
                ! Contribution for next-nearest neighbors.
                sum_H1_H2 = 4.0*J_coupling_squared*next_nearest_orbs(excitation%from_orb(1),excitation%to_orb(1))
            end if
            ! Contributions for nearest neighbors.
            ! Note, for certain lattices, such as the triangular lattice, two spins can be both
            ! nearest neighbors *and* next-nearest neighbors. Therefore, it is necessary in general
            ! to check for both situations.
            bit_position1 = bit_lookup(1,excitation%from_orb(1))
            bit_element1 = bit_lookup(2,excitation%from_orb(1))
            if (btest(connected_orbs(bit_element1, excitation%to_orb(1)), bit_position1)) &
```

```
                sum_H1_H2 = sum_H1_H2 - 4.0*J_coupling*(walker_data(1,idet)+H00)

    else if (excitation%nexcit == 2) then
        ! If there are two excitations (4 spins flipped) then, once again, the contribution
        ! to the thermal energy squared will depend on the positions of the spins. If there
        ! are two pairs of spins flipped which are separated then there is one way
        ! for this to happen - by flipping one pair, and then the other (this also requires
        ! that the two spins within each neighboring pair are opposite, as ever for the
        ! Heisenberg model). These two flips can happen in either order.
        ! In some cases the spins may be such that we may pair the spins in more than one way.
        ! For example, if the four spins are in a square shape, or for a 4-by-4 Heisenberg
        ! model, the spins could be connected across the whole lattice, forming a ring due
        ! to the periodic boundaries. In these cases it may be possible to perform the spin
        ! flips by pairing them in either of two ways. To account for this possibility we
        ! have to try and pair the spins in both ways, so we always check both if statements
        ! below. Again, once these pairings have been chosen, the flips can be performed in
        ! either order.

        bit_position1 = bit_lookup(1,excitation%from_orb(1))
        bit_element1 = bit_lookup(2,excitation%from_orb(1))
        bit_position2 = bit_lookup(1,excitation%from_orb(2))
        bit_element2 = bit_lookup(2,excitation%from_orb(2))
        if (btest(connected_orbs(bit_element1, excitation%to_orb(1)), bit_position1) .and. &
        btest(connected_orbs(bit_element2, excitation%to_orb(2)), bit_position2)) &
            sum_H1_H2 = 8.0*J_coupling_squared
        if (btest(connected_orbs(bit_element1, excitation%to_orb(2)), bit_position1) .and. &
        btest(connected_orbs(bit_element2, excitation%to_orb(1)), bit_position2)) &
            sum_H1_H2 = sum_H1_H2 + 8.0*J_coupling_squared

    end if

    estimator_numerators(energy_squared_index) = estimator_numerators(energy_squared_index) + &
            sum_H1_H2*walker_pop

end subroutine dmqmc_energy_squared_heisenberg

subroutine dmqmc_energy_hub_real(idet, excitation, walker_pop)

    ! For the Heisenberg model only.
    ! Add the contribution from the current density matrix element
    ! to the thermal energy estimate.

    ! In:
    ! idet: Current position in the main bitstring (density matrix) list.
```

```
        ! excitation: excit type variable which stores information on
        ! the excitation between the two bitstring ends, corresponding
        ! to the two labels for the density matrix element.
        ! walker_pop: number of particles on the current density matrix
        ! element.

        use basis, only: basis_length, total_basis_length
        use excitations, only: excit
        use fciqmc_data, only: walker_dets
        use fciqmc_data, only: walker_data, H00
        use fciqmc_data, only: estimator_numerators, energy_index
        use hamiltonian, only: slater_condon1_hub_real

        integer, intent(in) :: idet
        type(excit), intent(in) :: excitation
        real(p) , intent(in) :: walker_pop
        real(p) :: hmatel

        ! If no excitation, we have a diagonal element, so add elements
        ! which involve the diagonal element of the Hamiltonian.
        if (excitation%nexcit == 0) then
            estimator_numerators(energy_index) = estimator_numerators(energy_index) + &
                                    (walker_data(1,idet)+H00)*walker_pop
        else if (excitation%nexcit == 1) then
        ! If not a diagonal element, but only a single excitation, then the corresponding
        ! Hamiltonian element may be non-zero. Calculate if the flipped spins are
        ! neighbours on the lattice, and if so, add the contirbution from this site.
            hmatel = slater_condon1_hub_real(excitation%from_orb(1), excitation%to_orb(1), excitation%perm)
            estimator_numerators(energy_index) = estimator_numerators(energy_index) + &
                                    (hmatel*walker_pop)
        end if

    end subroutine dmqmc_energy_hub_real

    subroutine dmqmc_correlation_function_heisenberg(idet, excitation, walker_pop)

        ! For the Heisenberg model only.
        ! Add the contribution from the current density matrix element
        ! to the thermal spin correlation function estimator.

        ! In:
        ! idet: Current position in the main bitstring (density matrix) list.
        ! excitation: excit type variable which stores information on
        ! the excitation between the two bitstring ends, corresponding
        ! to the two labels for the density matrix element.
        ! walker_pop: number of particles on the current density matrix
        ! element.

        use basis, only: basis_length, total_basis_length
        use basis, only: bit_lookup
        use bit_utils, only: count_set_bits
        use excitations, only: excit
        use fciqmc_data, only: walker_dets
        use fciqmc_data, only: walker_data, H00, correlation_mask
        use fciqmc_data, only: estimator_numerators, correlation_index
        use hubbard_real, only: connected_orbs
        use system, only: J_coupling

        integer, intent(in) :: idet
        type(excit), intent(in) :: excitation
        real(p) , intent(in) :: walker_pop
        integer(i0) :: f(basis_length)
        integer :: bit_element1, bit_position1, bit_element2, bit_position2
        integer :: sign_factor

        ! If no excitation, we want the diagonal element of the correlation function operator.
        if (excitation%nexcit == 0) then
            ! If the two spis i and j are the same, the matrix element is +1/4.
            ! If they are different, the matrix element is -1/4.
            ! So we want sign_factor to be +1 in the former case, and -1 in the latter case.
            ! f as calculated below will have 0's at sites other than i and j, and the same values
            ! as walker_dets at i and j. Hence, if f has two 1's or no 1's, we want
            ! sign_factor = +1. Else if we have one 1, we want sign_factor = -1.
            f = iand(walker_dets(:basis_length,idet), correlation_mask)
            ! Count if we have zero, one or two 1's.
            sign_factor = sum(count_set_bits(f))
            ! The operation below will map 0 and 2 to +1, and will map 1 to -1, as is easily checked.
            sign_factor = (mod(sign_factor+1,2)*2)-1
            ! Hence sign_factor can be used to find the matrix element, as used below.
            estimator_numerators(correlation_index) = estimator_numerators(correlation_index) + &
                                   (sign_factor*(walker_pop/4))
        else if (excitation%nexcit == 1) then
        ! If not a diagonal element, but only a single excitation, then the corresponding
        ! matrix element will be 1/2 if and only if the two sites which are flipped are
        ! sites i and j, else it will be 0. We assume that excitations will only be set if
        ! i and j are opposite (else they could not be flipped, for ms=0).
            bit_position1 = bit_lookup(1,excitation%from_orb(1))
            bit_element1 = bit_lookup(2,excitation%from_orb(1))
```

```fortran
            bit_position2 = bit_lookup(1,excitation%to_orb(1))
            bit_element2 = bit_lookup(2,excitation%to_orb(1))
            if (btest(correlation_mask(bit_element1), bit_position1).and.btest(correlation_mask(bit_element2),
                 bit_position2)) &
                 estimator_numerators(correlation_index) = estimator_numerators(correlation_index) + &
                                 (walker_pop/2)
        end if

    end subroutine dmqmc_correlation_function_heisenberg

    subroutine dmqmc_stag_mag_heisenberg(idet, excitation, walker_pop)

        ! For the Heisenberg model only.
        ! Add the contribution from the current density matrix element
        ! to the thermal staggered magnetisation estimate.

        ! In:
        ! idet: Current position in the main bitstring (density matrix) list.
        ! excitation: excit type variable which stores information on
        ! the excitation between the two bitstring ends, corresponding
        ! to the two labels for the density matrix element.
        ! walker_pop: number of particles on the current density matrix
        ! element.

        use basis, only: basis_length, total_basis_length, bit_lookup
        use bit_utils, only: count_set_bits
        use determinants, only: lattice_mask
        use excitations, only: excit
        use fciqmc_data, only: walker_dets
        use fciqmc_data, only: estimator_numerators, staggered_mag_index
        use system, only: nel, nsites

        integer, intent(in) :: idet
        type(excit), intent(in) :: excitation
        real(p) , intent(in) :: walker_pop
        integer :: bit_element1, bit_position1, bit_element2, bit_position2
        integer(i0) :: f(basis_length)
        integer :: n_up_plus
        integer :: total_sum

        total_sum = 0

        ! This is for the staggered magnetisation squared. This is given by
        ! M^2 = M_x^2 + M_y^2 + M_z^2
        ! where M_x = \sum{i}(-1)^i S_i^x, etc... and S_i^x is the spin operator
        ! at site i, in the x direction.
        if (excitation%nexcit == 0) then
            ! Need to calculate the number of spins up on sublattice 1:
            ! N_u(+) - Number up on + sublattice (where (-1)^i above is +1)
            ! Note the number of down spins on a sublattice is easily obtained from
            ! N_u(+) since there are N/2 spins on each - and the number of spins up on
            ! a different sublattice is easily obtained since there are nel spins
            ! up in total. Hence the matrix element will be written only in terms
            ! of the number of up spins on sublattice 1, to save computation.
            f = iand(walker_dets(:basis_length,idet), lattice_mask)
            n_up_plus = sum(count_set_bits(f))
            ! Below, the term in brackets and middle term come from the z component (the
            ! z operator is diagonal) and one nsites/4 factor comes from the x operator,
            ! the other nsites/4 factor from the y operator.
            total_sum = (2*n_up_plus-nel)**2 + (nsites/2)
        else if (excitation%nexcit == 1) then
            ! Off-diagonal elements from the y and z operators. For the pair of spins
            ! that are flipped, if they are on the same sublattice, we get a factor of
            ! 1, or if on different sublattices, a factor of -1.
            bit_position1 = bit_lookup(1,excitation%from_orb(1))
            bit_element1 = bit_lookup(2,excitation%from_orb(1))
            bit_position2 = bit_lookup(1,excitation%to_orb(1))
            bit_element2 = bit_lookup(2,excitation%to_orb(1))
            if (btest(lattice_mask(bit_element1), bit_position1)) total_sum = total_sum+1
            if (btest(lattice_mask(bit_element2), bit_position2)) total_sum = total_sum+1
            ! The operation below will map 0 and 2 to +1, and will map 1 to -1, as is easily checked.
            ! We want this - if both or no spins on this sublattice, then both on same sublattice
            ! either way, so plus one. Else they are on different sublattices, so we want a factor
            ! of -1, as we get.
            total_sum = (mod(total_sum+1,2)*2)-1
        end if

        estimator_numerators(staggered_mag_index) = estimator_numerators(staggered_mag_index) + &
                                 (real(total_sum)/real(nsites**2))*walker_pop

    end subroutine dmqmc_stag_mag_heisenberg

    subroutine update_reduced_density_matrix_heisenberg(idet, walker_pop)

        ! Add a contribution from the current walker to the reduced density
        ! matrix estimator, which is produced by 'tracing out' a given set of
        ! spins.

        ! Applicable only to the Heisenberg model.

        ! This procedure takes the two determinants bitstrings for the current
        ! walker and, if the two bitstrings of the B subsystem are identical,
        ! adds the walker population to the corresponding reduced density matrix
        ! element.

        ! In:
        ! idet: Current position in the main bitstring (density matrix) list.
        ! walker_pop: number of particles on the current density matrix
        ! element.

        use basis, only: basis_length, total_basis_length
        use dmqmc_procedures, only: decode_dm_bitstring
        use fciqmc_data, only: subsystem_B_mask, reduced_density_matrix
        use fciqmc_data, only: walker_dets, walker_population

        integer, intent(in) :: idet
        real(p), intent(in) :: walker_pop
        integer(i0) :: f1(basis_length), f2(basis_length)
        integer(i0) :: end1, end2
        ! Apply the mask for the B subsystem to set all sites in the A subsystem to 0.
        f1 = iand(subsystem_B_mask,walker_dets(:basis_length,idet))
        f2 = iand(subsystem_B_mask,walker_dets(basis_length+1:total_basis_length,idet))

        ! Once this is done, check if the resulting bitstring (which can only possibly
        ! have 1's in the B subsystem) are identical. If they are, then this psip gives
        ! a contibution to the reduced density matrix for subsystem A. This is because we
        ! get the reduced density matrix for A by 'tracing out' over B, which in practice
        ! means only keeping matrix elements that are on the diagonal for subsystem B.
        if (sum(abs(f1-f2)) == 0) then
            ! We need to assign positions in the reduced density matrix for this walker,
            ! so call a function which maps the possible subsystem A spin configurations
            ! to indices.
            call decode_dm_bitstring(walker_dets(:,idet),end1,end2)
            reduced_density_matrix(end1,end2) = reduced_density_matrix(end1,end2) + walker_pop
        end if

    end subroutine update_reduced_density_matrix_heisenberg

    subroutine call_rdm_procedures()

         ! Wrapper for calling relevant reduced density matrix procedures

```

```
        use fciqmc_data, only: reduced_density_matrix, subsystem_A_size
        use fciqmc_data, only: doing_von_neumann_entropy, doing_concurrence
        use parallel

        real(p) :: trace_rdm
        integer :: i, j
        ! If in paralell then merge the reduced density matrix onto one processor
#ifdef PARALLEL

        real(dp) :: dm(2**subsystem_A_size,2**subsystem_A_size)
        real(dp) :: dm_sum(2**subsystem_A_size,2**subsystem_A_size)
        integer :: ierr

        dm = reduced_density_matrix

        call mpi_allreduce(dm, dm_sum, size(dm), MPI_REAL8, MPI_SUM, MPI_COMM_WORLD, ierr)

        reduced_density_matrix = dm_sum

#endif

        trace_rdm = 0.0_p

        if (parent) then
            ! Force the reduced desnity matrix to be symmetric by averaging the upper and lower triangles
            do i = 1, ubound(reduced_density_matrix,1)
                do j = 1, i-1
                    reduced_density_matrix(i,j) = 0.5_p*(reduced_density_matrix(i,j) +&
                            reduced_density_matrix(j,i))
                    reduced_density_matrix(j,i) = reduced_density_matrix(i,j)
                end do
                ! Add current contirbution to the trace.
                trace_rdm = trace_rdm + reduced_density_matrix(i,i)
            end do

            ! Call the routines to calculate the desired quantities.
            if (doing_von_neumann_entropy) call calculate_vn_entropy()
            if (doing_concurrence) call calculate_concurrence()

            write (6,'(1x,a12,1X,f22.12)') "# RDM trace=", trace_rdm
        end if


    end subroutine call_rdm_procedures

    subroutine calculate_vn_entropy()

        ! Calculate the Von Neumann Entropy. Use lapack to calculate the
        ! eigenvalues {\lambda_j} of the reduced density matrix.
        ! Then VN Entropy S = -\sum_j\lambda_j\log_2{\lambda_j}

        ! Need to paralellise for large subsystems and introduce test to
        ! check whether diagonalisation should be performed in serial or
        ! paralell.

        use checking, only: check_allocate, check_deallocate
        use fciqmc_data, only: subsystem_A_size, reduced_density_matrix

        integer :: i, j, rdm_size
        integer :: info, ierr, lwork
        real(p), allocatable :: work(:)
        real(p) :: eigv(2**subsystem_A_size)
        real(p) :: vn_entropy

        rdm_size = 2**subsystem_A_size
        vn_entropy = 0._p

        ! Find the optimal size of the workspace.
        allocate(work(1), stat=ierr)
        call check_allocate('work',1,ierr)
#ifdef SINGLE_PRECISION
        call ssyev('N', 'U', rdm_size, reduced_density_matrix, rdm_size, eigv, work, -1, info)
#else
        call dsyev('N', 'U', rdm_size, reduced_density_matrix, rdm_size, eigv, work, -1, info)
#endif
        lwork = nint(work(1))
        deallocate(work)
        call check_deallocate('work',ierr)

        ! Now perform the diagonalisation.
        allocate(work(lwork), stat=ierr)
        call check_allocate('work',lwork,ierr)

#ifdef SINGLE_PRECISION
        call ssyev('N', 'U', rdm_size, reduced_density_matrix, rdm_size, eigv, work, lwork, info)
#else
        call dsyev('N', 'U', rdm_size, reduced_density_matrix, rdm_size, eigv, work, lwork, info)
#endif
```

```
        do i = 1, ubound(eigv,1)
            vn_entropy = vn_entropy - eigv(i)*(log(eigv(i))/log(2.0_p))
        end do
        write (6,'(1x,a36,1X,f22.12)') "# Unnormalised Von-Neumann Entropy= ", vn_entropy

    end subroutine calculate_vn_entropy

    subroutine calculate_concurrence()

        ! Calculate the concurrence of a qubit. For a reduced density matrix \rho,
        ! the concurrence, C = max(0, \lamda_1 - \lambda_2 - \lambda_3 -\lambda_4) where
        ! \lambda_i are the eigenvalues of the matrix, R = \sqrt{\sqrt{\rho}\~{\rho}\sqrt{\rho}}.
        ! Where \~\rho = {\sigma_y \otimes \sigma_y} \rho^{\ast} {\sigma_y \otimes \sigma_y}.
        ! \lambda_1 > ... > \lambda_4

        ! This can be simplified to finding the square root of the eigenvalues \{\lambda_i\} of \rho\~{\rho}
        ! and in the case where \rho is a real, symmetric matrix then we can further simplify the problem
        ! to finding the eigenvalues of R = \rho \sigma_y \otimes \sigma_y.

        ! Below we have named {\sigma_y \otimes \sigma_y} flip_spin_matrix as in the literature.

        use checking, only: check_allocate, check_deallocate
        use fciqmc_data, only: subsystem_A_size, reduced_density_matrix, flip_spin_matrix
        use utils, only: print_matrix
        integer :: i,j
        integer :: info, ierr, lwork
        real(p), allocatable :: work(:)
        real(p) :: reigv(4), ieigv(4)
        real(p) :: concurrence
        real(p) :: rdm_spin_flip(4,4), rdm_spin_flip_tmp(4,4), VL(4,4), VR(4,4)

        ! Make rdm_spin_flip_tmp because sgeev and dgeev delete input matrix
        rdm_spin_flip_tmp = matmul(reduced_density_matrix, flip_spin_matrix)
        rdm_spin_flip = rdm_spin_flip_tmp

        ! Find the optimal size of the workspace.
        allocate(work(1), stat=ierr)
        call check_allocate('work',1,ierr)
#ifdef SINGLE_PRECISION
        call sgeev('N', 'N', 4, rdm_spin_flip_tmp, 4, reigv, ieigv, VL, 1, VR, 1, work, -1, info)
#else
        call dgeev('N', 'N', 4, rdm_spin_flip_tmp, 4, reigv, ieigv, VL, 1, VR, 1, work, -1, info)
#endif
        lwork = nint(work(1))
        deallocate(work)
        call check_deallocate('work',ierr)

        ! Now perform the diagonalisation.
        allocate(work(lwork), stat=ierr)
        call check_allocate('work',lwork,ierr)
#ifdef SINGLE_PRECISION
        call sgeev('N', 'N', 4, rdm_spin_flip, 4, reigv, ieigv, VL, 1, VR, 1, work, lwork, info)
#else
        call dgeev('N', 'N', 4, rdm_spin_flip, 4, reigv, ieigv, VL, 1, VR, 1, work, lwork, info)
#endif
        ! Calculate the concurrence. Take abs of eigenvalues so that this is equivelant to sqauring
        ! and then square-rooting
        concurrence = 2._p*maxval(abs(reigv)) - sum(abs(reigv))
        concurrence = max(0._p, concurrence)
        write (6,'(1x,a28,1X,f22.12)') "# Unnormalised concurrence= ", concurrence

    end subroutine calculate_concurrence

end module dmqmc_estimators
```