FCIQMC Hands-on Session

George Booth

July 31, 2019

1 Part I: Propagating 'walkers'...

In this session, we will attempt to code up the key parts of the FCIQMC method from just an incomplete skeleton code. FCIQMC works in a second-quantized representation, resulting in a discrete and finite number of Slater determinants to sample, rather than the continuous, real-space space of configurations seen to date. In this first part, we will explicitly read in the Hamiltonian matrix for the problem in this basis, and then write an FCIQMC population dynamic code to sparsely and stochastically sample both this Hamiltonian, and a distribution of walkers which represent snapshots of this ground-state. Expectation values such as the energy can then be averaged from these snapshots.

1.1 The basis used in FCIQMC

We can define the Hamiltonian operator in second quantization as

$$\hat{H} = \sum_{pq}^{2M} h_{pq} c_p^\dagger c_q + \frac{1}{2} \sum_{pqrs}^{2M} v_{pqrs} c_p^\dagger c_q^\dagger c_s c_r + E_{NN}, \tag{1}$$

which is defined for a basis of $2M$ one-electron spin-orbitals, $\{\phi_1(\mathbf{r}), \ldots, \phi_p(\mathbf{r}), \ldots, \phi_q(\mathbf{r}), \ldots, \phi_{2M}(\mathbf{r})\}$ (half referring to α spins, and half β spins). The coefficients of the terms are given by

$$h_{pq} = \langle \phi_p(\mathbf{r}) | h | \phi_q(\mathbf{r}) \rangle \tag{2}$$

$$v_{pqrs} = \langle \phi_p(\mathbf{r}_1) \phi_q(\mathbf{r}_2) | r_{12}^{-1} | \phi_r(\mathbf{r}_1) \phi_s(\mathbf{r}_2) \rangle = \langle pq | rs \rangle, \tag{3}$$

where the h operator is a sum of kinetic energy and electron-nuclear repulsion terms. In the limit of a complete basis set, this form of the Hamiltonian is equivalent to the first-quantization representation, but now refers explicitly to orbital degrees of freedom, rather than electrons. A discrete and finite set of explicitly anti-symmetric N-electron functions can then be constructed from this basis, from all ways of occupying N orbitals out of the $2M$ basis functions, where N is the number of electrons in the system. These are the *Slater determinants*, which define the many-body basis of an FCIQMC calculation. However, unlike

the other methods discussed, within FCIQMC it is not necessary to actually evaluate these determinants explicitly in order to sample expectation values.

Question: What is the expression for the number of determinants spanning the wave function for a system of (N_α, N_β) electrons in a space of $2M$ spin-orbitals (in the absence of other (e.g. spatial) symmetry considerations).

In Part I, we are going to hide the specifics of the Hamiltonian construction, so that we can focus on the details of the FCIQMC algorithm more broadly. We can represent a given Slater determinant as $|D_{\mathbf{i}}\rangle$, where $\mathbf{i}$ is a compound index which labels which N out of the $2M$ spin-orbitals are occupied in the configuration. Given the set of all Slater determinants which can be formed, we can evaluate the Hamiltonian in this basis, which we will represent as

$$\langle D_{\mathbf{i}}|\hat{H}|D_{\mathbf{j}}\rangle = H_{\mathbf{ij}}. \tag{4}$$

We will assume initially that we can read in this Hamiltonian matrix, which will then mean that we can avoid the necessity to choose a representation of the orbitals, evaluate the Hamiltonian matrix elements etc. which we will leave to Part II. The ground state many-body problem is then just the task of solving the Schrodinger equation as

$$\hat{H}|\Psi_0\rangle = E_0|\Psi_0\rangle, \tag{5}$$

where E_0 is the lowest energy eigenvalue, and $|\Psi_0\rangle$ is its associated eigenvector. This ground state can be written as a linear combination of *all* Slater determinants in the system, as

$$|\Psi_0\rangle = \sum_{\mathbf{i}} C_{\mathbf{i}}|D_{\mathbf{i}}\rangle. \tag{6}$$

In this part, you will find dumped numpy arrays corresponding to Hamiltonians in this explicit matrix form for two systems of hydrogen rings separated by 1.0Å in a Hartree-Fock orbital representation of a minimal basis set. All the Slater determinants amongst these orbitals are enumerated, and the the matrix elements evaluated for six Hydrogen atoms (`Full_Ham_6H.npy`) and eight Hydrogen atoms (`Full_Ham_8H.npy`). We can then just label Slater determinants by their index in this matrix.

1.2 The FCIQMC Algorithm

1.2.1 Wave function representation and compression

The FCIQMC algorithm is designed to find a sparse and stochastically varying representation of the ground state wavefunction shown in Eq. 6, which therefore breaks the *explicitly* factorial scaling of the wave function (though it may still be implicit). We define a snapshot of the wavefunction at a given iteration ($|\tilde{\Psi}(\beta)\rangle$, by analogy with imaginary time) as

$$|\tilde{\Psi}(\beta)\rangle = \sum_{\mathbf{i}}^{N_d} \tilde{C}_{\mathbf{i}}(\beta)|D_{\mathbf{i}}\rangle \tag{7}$$

where quantities with a tilde are denoted as stochastically varying, β refers to the particular iteration, and N_d is the number of determinants 'occupied' in this iteration (i.e. $\tilde{C}_{\mathbf{i}}(\beta) \neq 0$), generally less than the full set of determinants.

There are a number of ways to compress the information in Eq. 6 to Eq. 7 (i.e. $C_{\mathbf{i}} \rightarrow \tilde{C}_{\mathbf{i}}(\beta)$). In early FCIQMC implementations this was done by keeping all determinant weights ($\tilde{C}_{\mathbf{i}}$) as integers, and defining 'walkers' as quantities with a signed unit amplitude. However, this is unnecessarily restrictive, and so we instead allow $\tilde{C}_{\mathbf{i}}(\beta)$ to be a floating-point number. To compress the value and ensure that determinants corresponding to small weights are removed from the wavefunction sum, we define a threshold, χ, which is the minimum absolute weight on any single determinant.

If $|\tilde{C}_{\mathbf{i}}(\beta)| < \chi$, then a *stochastic rounding* step is applied, such that its absolute amplitude is rounded up to χ with probability $\frac{|\tilde{C}_{\mathbf{i}}(\beta)|}{\chi}$, or otherwise rounded down to zero, where the determinant can be entirely removed from the simulation. By ensuring that only *occupied* determinants (i.e. non-zero weight) are explicitly stored in the simulation, along with the stochastic rounding, we can ensure that we have control of the sparsity of the wave function representation in Eq. 7. We then also define a number of 'walkers', which we take to be the L1-norm of the sampled wavefunction, i.e.

$$N_w(\beta) = \sum_{\mathbf{i}} |\tilde{C}_{\mathbf{i}}(\beta)|. \tag{8}$$

1.2.2 Energy expectation values

With this stochastically varying sparse wavefunction, we need to be able to compute expectation values, the most important of which is the energy. We use the 'mixed' or 'projected' energy estimate, as

$$E(\beta) = \frac{\langle \Psi_T | H | \tilde{\Psi}(\beta) \rangle}{\langle \Psi_T | \tilde{\Psi}(\beta) \rangle}, \tag{9}$$

where in these exercises, we will take Ψ_T to be a single, 'reference' Slater determinant, which we denote by the subscript $|D_{\mathbf{0}}\rangle$. We can then denote the energy (confirm for yourself) as

$$E(\beta) = \frac{\sum_{\mathbf{i} \neq \mathbf{0}}^{N_d} H_{\mathbf{0i}} \tilde{C}_{\mathbf{i}}(\beta)}{\tilde{C}_{\mathbf{0}}(\beta)} + H_{\text{ref}}, . \tag{10}$$

where $E_{\text{ref}} = H_{\mathbf{00}}$ is the energy of the reference determinant. It is then possible to average the numerator and denominator of Eq. 10 in order to compute an energy averaged over many iterations. In that way, despite Eq. 7 generally being a poor approximation to Eq. 6 at any one iteration, far better energies can still often be obtained by averaging over this dynamic representation for many iterations.

1.2.3 Propagation

The condition that we want our weights to satisfy in order to extract the exact energy, is that

$$\lim_{\beta\to\infty} \langle \tilde{C}_{\mathbf{i}}(\beta)\rangle_\beta \propto C_{\mathbf{i}}, \tag{11}$$

i.e. that the time-average of the determinant weights is proportional to their exact coefficient. However, obviously we do not want to do this time-averaging explicitly on the determinant weights, otherwise we return to a proliferation of low-weighted amplitudes. Instead, we aim to implement a stochastically realised propagator for the dynamic of the 'walkers' at each iteration, $\tilde{C}_{\mathbf{i}}(\beta)$, and average the expectation values, as in Eq. 10. The master equation for this propagation at a given iteration, β, is

$$\Delta\tilde{C}_{\mathbf{i}}(\beta) = \underbrace{-\tau\left[\sum_{\mathbf{j}\neq\mathbf{i}} H_{\mathbf{ij}}\tilde{C}_{\mathbf{j}}(\beta)\right]}_{\text{spawning step}} \underbrace{-\tau(H_{\mathbf{ii}} - H_{\mathbf{00}} - S)\tilde{C}_{\mathbf{i}}(\beta)}_{\text{death step}}, \tag{12}$$

where S is an energy 'shift', which is dynamically updated during the course of the simulation, and τ is a small timestep for the propagation. This shift is used to stabilise the growth of walkers, such that we can operate at a constant (on average) N_w. In order for this to be achieved, the shift is updated every A iterations (where A is a small number of iterations ~ 10, which we call an 'update cycle' or 'block'), as

$$S(\beta + A\tau) = S(\beta) - \frac{\xi}{A\tau}\ln\frac{N_w(\beta + A\tau)}{N_w(\beta)}, \tag{13}$$

where $N_w(\beta)$ is the number of walkers at the previous update cycle, and $N_w(\beta + A\tau)$ is the number of walkers at the current update cycle, and ξ is a damping term to control the magnitude of the fluctuations in S. Alternatively, if we want to *grow* the number of walkers, to get a better resolution of the wavefunction and smaller statistical errors, then we can work in a 'fixed shift mode', where the shift is fixed (at $\sim$ zero), in order to grow the average number of walkers.

The simulation procedure generally starts in this mode, where the shift is fixed, and the N_w grows, from an initial small weight on just the reference determinant, to explore the space and resolve the wave function. Once a desired number of walkers is reached, the shift is allowed to vary according to Eq. 13, and the walker number stabilizes. Shortly after this (perhaps with an additional small equilibration time), statistics can be accumulated to sample Eq. 10.

Question: Convince yourself that for a stationary walker distribution, the determinant amplitudes satisfy the Schrodinger equation. For a ground-state dynamic, what is the value of S which will do this?

1.3 Implementation

We now consider an algorithm to stochastically implement Eq. 12 via a local Markov chain, which we will implement. This is one choice of a number of close variants which will implement the dynamic, but we pick this one to mirror the skeleton code you will build on in this session. The 'exercises' folder will have some files in, and for Part I, all that is needed is the following files:

- `fciqmc_partI.py`
- `system.py`
- `Full_Ham_6H.npy`
- `Full_Ham_8H.npy`

Of these four files, `Full_Ham_6H.npy` and `Full_Ham_8H.npy` are binary files containing a dumped numpy array of the Hamiltonian in the Slater determinant basis (i.e. $H_{\mathbf{ij}}$) for the 6 and 8 Hydrogen ring systems respectively, as described previously. The determinant basis is ordered such that the first determinant (i.e. the zeroth python indexed entry) refers to the 'reference' determinant. These files can be read in simply to define the Hamiltonian we will be working with for the implementation.

The `system.py` file shouldn't be modified, however, it should read, to understand the naming of the parameters that will be used. It contains two classes:

- `PARAMS`

  This class takes the input parameters for the simulation, and stores them. Read the comments for each parameter stored – they should make sense from the description of the algorithm above. This class also implements a method to update the value of the shift according to Eq. 13.

- `STATS`

  This class stores the stochastic variables which vary (and may be accumulated) during the course of the simulation, such as N_w, N_d, $\tilde{C}_{\mathbf{0}}(\beta)$, and the numerators and denominators of Eq. 10 averaged over various iterations. Again, have a look through these variables, and you should have a good idea what they are referring to. In addition, there are methods for computing/updating the energies and error (both averaged over the simulation, and just the current update cycle), as well as to output the statistics each update cycle to a file called `fciqmc_stats`, which can be plotted during the simulation to observe progress in real-time.

`fciqmc_partI.py` contains the skeleton code which must be modified to implement the FCIQMC algorithm – read it carefully. Initially, the hamiltonian is read in, $H_{\mathbf{00}}$ is found, and used to shift the diagonal of the hamiltonian matrix. The simulation parameters are then set up, along with the object to hold the simulation statistics. Determinants will be

labelled simply by their index in the hamiltonian matrix. A dictionary is set up to hold the N_d determinants occupied by walkers at any one time, and initialized with a small number of walkers on the reference determinant. This dictionary of determinants will take the index of the determinant as a key, and its current amplitude as its value.

We summarize the algorithm to be implemented below to update this walker list each iteration, and must be used to complete the skeleton code. Every `pass` statement in the skeleton code denotes one/a few lines of code which need to be added to perform a task to complete the algorithm below:

For iter in iterations:
    For $\mathbf{j}$ in N_d occupied determinants:

1. **Accumulate:** Update contibution to the energy expectation values of Eq. 10 for this iteration.

2. **Compression:** Stochastically round low-weighted determinant amplitudes, according to the parameter χ, either by removing the determinant and skipping the rest of the loop, or scaling up the absolute value of the amplitude to χ. How this is done is described in Sec. 1.2.1.

3. **Spawning:** Perform a stochastic realization of the first term of Eq. 12.

    - We first need to choose an integer number of 'spawning' attempts, n_s, from the current determinant, $|D_\mathbf{j}\rangle$. We can choose n_s to be proportional to the absolute amplitude on the determinant, $\tilde{C}_\mathbf{j}(\beta)$ (e.g. the nearest integer).
    - For each of the n_s attempts, we then want to stochastically choose another determinant in the space, ensuring that it is not equal to $\mathbf{j}$. We label this determinant $\mathbf{i}$, and calculate the normalized probability of having selecting it, $p_{\text{gen}}(\mathbf{i}|\mathbf{j})$.
    - We then compute the spawning weight onto determinant $\mathbf{i}$, based on the current state of $\mathbf{j}$. This is done as

$$p_{\text{spawn}}(\mathbf{j} \to \mathbf{i}) = -\tau \frac{H_{\mathbf{ij}} \tilde{C}_\mathbf{j}}{p_{\text{gen}}(\mathbf{i}|\mathbf{j}) n_s}. \tag{14}$$

    - This spawned weight should be put (or added if already there) into the spawned walker dictionary, which is used to store all the spawned walkers each iteration, and will be used to augment the amplitudes of the main walker list later.

    Convince yourself, that what you are doing is a stochastic realization of the first term of Eq. 12, where instead of taking all elements of the sum, you are instead choosing n_s at random. As a note, it may be worth ensuring that n_s is at least 1, to ensure that all determinants get the chance to spawn.

4. **Death:** This step realizes the second term in Eq. 12. Modify the amplitudes of the determinant $\mathbf{j}$ by
$$\Delta\tilde{C}_{\mathbf{j}} = -\tau(H_{\mathbf{jj}} - H_{\mathbf{00}} - S)\tilde{C}_{\mathbf{j}}(\beta). \quad (15)$$
This can be done directly in the main walker dictionary.

5. **Annihilation:** Loop through the list of walkers (determinant weight) which has been spawned this iteration, and use it to augment the amplitudes in the main walker list. This may mean adding the new weight to determinants which are already occupied, or introducing new occupied determinants to the main list.

6. Finally, the energy expectation is averaged / written out, and the shift varied (if the target number of walkers has been reached) every A iterations.

The challenge is now to complete this code, and use it to run an FCIQMC calculation on the Hydrogen ring systems, to see if you can get agreement with the exact diagonalization energies ($-3.23747E_h$ and $-4.17546E_h$ for the 6 and 8 Hydrogen systems respectively). The output file, `fciqmc_stats` should output the iteration number, S, N_w, $E(\beta)$, $\langle E\rangle$, Error in $\langle E\rangle$, N_d and $\tilde{C}_{\mathbf{0}}(\beta)$ every A iterations, which can be plotted in real-time with gnuplot. Note that at convergence, $S + E_{\text{ref}}$, as well as $\langle E\rangle$ and $E(\beta)$ should all fluctuate randomly about the exact energy.

If you have difficulty, please ask for help. There are also worked solutions for what should be implemented in the 'solutions' folder, but please try to work through the code yourself and ideally get it working *before* looking at them.

1.4 Further questions

- Ensure you can get the exact energies. How does the simulation change with initial parameters? Specifically, change the target N_w, χ (the determinant threshold weight) and τ (the timestep), and see how this changes the profile of the number of walkers, number of occupied determinants, and accuracy/efficiency of the ground-state energy. Note in particular how you can use τ to control the rate of growth of walkers (in fixed shift mode), or magnitude of the energy fluctuations (in variable shift mode).

- Why would we store the walker lists as a dictionary (as opposed to e.g. a list)?

- Suggest how you could improve the accuracy of the energy estimator?

- Can you converge the energy of the 8 Hydrogen system with an arbitrarily small number of walkers? What problems might be face in trying to do so?

- Do the errorbars computed for the averaged energy estimator look to be over- or under-estimated (remember that the true value should be inside the errorbars $\sim 60\%$ of the time). Have a look at how the errorbar is computed in `system.py`, and suggest two reasons why this is not an optimal estimate of the random error. How could we treat the errors more rigorously?

- In the current implementation, identify the explicitly exponentially scaling parts of this algorithm as it is.

2 Part II: Making it practical...

The challenge set in Part I was somewhat artificial. While it demonstrated the main aspects of the FCIQMC algorithm, it required the entire Hamiltonian matrix to be enumerated and stored, which is unfeasible for all but small systems (look at how much disk space the two `Full_Ham_6H.npy` and `Full_Ham_8H.npy` files take up, as you add two more orbitals). To make the approach practical, it has to at least remove the explicitly exponentially scaling parts. This will require us to generate the $H_{\mathbf{ij}}$ hamiltonian matrix elements on-the-fly when needed, by computing the expectation value of Eq. 1 between two Slater determinants. The information required to specify this expectation value is encoded within the h_{pq}, v_{pqrs} and E_{NN} values, which define the Hamiltonian of the system. This is also a far more efficient way to go, since the hamiltonian matrix is also in fact incredible sparse. While its size grows exponentially, the number of non-zero elements grows only quartically in the worse case, as only single and double electron replacements have a non-zero hamiltonian value between them.

Question: From the form of Eq. 1, explain why only single and double electron replacement determinants have non-zero hamiltonian matrix elements between them.

Question: Consider which permutations of the indices p, q, r and s will result in equivalent values in the two-electron integral tensor v_{pqrs}. What about constraints imposed by the spin label of each orbital?

2.1 Slater-Condon Rules

The Slater-Condon rules are the equations to compute the Hamiltonian matrix elements between two Slater determinants, $|D_{\mathbf{i}}\rangle$ and $|D_{\mathbf{j}}\rangle$, given the integrals h_{pq}, v_{pqrs} and E_{NN}. We divide these into 3 classes, the diagonal ($|D_{\mathbf{i}}\rangle = |D_{\mathbf{j}}\rangle$), determinants separated by a single electron substitution (i.e. in $|D_{\mathbf{i}}\rangle$, we annihilate the electron in orbital p, and create an electron in orbital a in order to get $|D_{\mathbf{j}}\rangle$, denoted $p \rightarrow a$), or determinants separated by a double electron substitution (i.e. $p \rightarrow a$ and $q \rightarrow b$). These expectation values are given below, and can be derived by correctly accounting for the commutation relations of the operators in Eq. 1 (which can equivalently be cast as enforcing the antisymmetry of the determinant states).

2.1.1 Diagonal hamiltonian matrix elements, $|D_{\mathbf{i}}\rangle = |D_{\mathbf{j}}\rangle$

$$H_{\mathbf{ii}} = E_{NN} + \sum_{p}^{N} h_{pp} + \sum_{p}^{N} \sum_{q>p}^{N} \langle pq|pq\rangle - \langle pq|qp\rangle, \tag{16}$$

where the summations denote running over the orbital indices occupied in the determinant $|D_{\mathbf{i}}\rangle$.

2.1.2 Single electron substitutions, $p \rightarrow a$

$$H_{\mathbf{ij}} = h_{pa} + \sum_{q}^{N} \langle pq|aq\rangle - \langle pq|qa\rangle \tag{17}$$

2.1.3 Double electron substitutions, $p \rightarrow a$ and $q \rightarrow b$

$$H_{\mathbf{ij}} = \langle pq|ab\rangle - \langle pq|ba\rangle \tag{18}$$

2.1.4 Parity considerations

Something should be nagging at you. For instance, if we were to consider instead a double substitution of $p \rightarrow b$ and $q \rightarrow a$, rather than $p \rightarrow a$ and $q \rightarrow b$, then according to above, the Hamiltonian matrix element would change sign. However, the determinant that you would end up at, would be the same! The resolution to this apparent paradox comes from the fact that all Slater determinant configurations must be defined with respect to a consistent ordering of the orbitals. If the ordering of the orbitals changes, then this would be equivalent to changing columns in the Slater determinant, which would change the sign of the basis functions. This arises from the implicit antisymmetry of our configurational basis functions. Therefore, the determinants that we excite to via the electron substitutions must end up with a consistent orbital ordering. If the excitations change the ordering of the occupied indices, then pairwise permutations need to be applied to end up with a consistent orbital ordering, noting that for each permutation of neighbouring electrons results in a sign-change. This can be considered equivalent to pairwise exchange of creation operators acting on the true vacuum state, which from the commutation relations must change the sign of the state. The overall 'parity' of the excitation is +1 if the number of permutations is even, or -1 if it is odd.

From an algorithmic perspective, we can consider the following approach to compute the parity of an excitation where $p \rightarrow a$ in a determinant $|D_{\mathbf{i}}\rangle$, where $\mathbf{i}$ defines an ordered list of the occupied orbitals (including p) in the original determinant.

1. Replace occupied orbital p by a in the (ordered) list defining $\mathbf{i}$.
2. Exchange the indices of the orbital a with the orbital indices directly above or below it, in order to return the list to an ordered state.
3. Count the number of neighbour pairwise permutations of the occupied indices this required, P.
4. This new list defines the ordered, occupied orbital list $\mathbf{j}$ for the excited determinant $|D_{\mathbf{j}}\rangle$.
5. The *parity* of this excitation is given by $(-1)^P$.

For double electron excitations, then this ordering process is performed twice, once for the first electron replacement and once for the second, and the overall permutation is the sum of the number of permutations require for each reordering, with the overall parity as $(-1)^{P_1+P_2}$. Finally, in order to compute the Hamiltonian matrix elements for an excitation, this parity should be multiplied by the result of the Slater-Condon rules above, to ensure that this anti-symmetry of the determinants is correctly taken into account (not needed for diagonal elements).

2.2 Initiator Approximation

The other adaptation we will try to implement in this part is the *initiator* approximation. As you may have seen in Part I, as you increase the size of the system, it is not always possible to converge the energy with arbitrarily small numbers of walkers. Each system has a specific number of walkers which must be exceeded in order to ensure convergence to the correct energy. An adaptation to the algorithm is often used, which can dramatically improve the rate of convergence to this exact limit. An additional rule is introduced to the dynamic, which is hard to formulate mathematically:

- At the annihilation stage, if spawned weight is being transferred onto a previously *unoccupied* determinant, then this can only proceed if the *parent* determinant has an absolute weight which exceeds some user-defined threshold, n_{init}.

This approximation is a dynamic approximation, as since the weights of the determinants fluctuate, the restrictions on the spawning change over iterations, in a way that mitigates much of the error introduced in any single iteration.

The 'hand-waving' justification for the constraint is that the sign-problem manifests itself in FCIQMC from the difficulty in establishing a *sign-structure* of the determinant amplitudes in Eq. 7. If this were a trivial sign-structure, then an arbitrarily small number of walkers should be sufficient to simulate the system in an unbiased fashion. In the presence of a sign problem, the correct sign-structure of the determinants only emerges through annihilation events, which in turn depend on the number of walkers. We denote determinants who do not have a correctly established sign-structure as 'sign-incoherent', and errors exponentially quickly propagate through the system if sign-incoherent determinants spawn to unoccupied determinants (precluding annihilation).

Therefore, we use the proxy of the determinant amplitude to determine whether a determinant is likely to be sign-coherent with its neighbours (i.e. if its weight is above n_{init} or not). If so, then we allow it to propagate its amplitude (and hence sign-structure) to neighbouring unoccupied determinants via spawning events - if not, then it is constrained from doing so, in case it propagates incorrectly signed weights. Note that as the walker number increases, more determinants will exceed this threshold, and hence will have no constraints on their dynamic, and so the algorithm naturally returns to the 'exact' unbiased algorithm.

2.3 Implementation

Since we are not enumerating all determinants, we can no longer use the index of the hamiltonian matrix element to uniquely represent a Slater determinant as we did in Part I. Instead, we can represent a determinant by an ordered list of integers giving the indices of its N occupied spin-orbitals[1]. Another new functionality required is to randomly excite a determinant to another one, via either a single or double electron substitution, and to calculate a normalized probability of having done so (rather than just randomly picking determinant indices in Part I). The single excitation requires selection of a single occupied spin-orbital, and single unoccupied spin-orbital, while the double excitation requires the selection of a unique pair of occupied and unoccupied spin-orbitals. This can get quite involved if you want to include many different symmetry selection rules in this, or if you want to have a custom importance sampled distribution for the picking, but we will keep it simple and ignore these considerations.

Finally, it is necessary to read in a Hamiltonian specified by the integrals v_{pqrs}, h_{pq} and E_{NN}, as well as specifying the system parameters such as number of electrons. This will be done with four new files with more skeleton code for the Part II exercise. These are:

- `det_ops.py`
- `fciqmc_partII.py`
- `FCIDUMP.6H`
- `FCIDUMP.8H`

These files are needed, in addition to the `system.py` file from Part I, which does not need to be modified. Have a look at them to get an idea for the tasks at hand.

The `FCIDUMP` files define the system and hamiltonian, as given in Eq. 1, in an 'industry-standard', human-readable format which is available directly from many open-source electronic-structure programs, including `MOLPRO`, `PySCF`, `VASP` and `Psi4` as well as others. The header to this file defines information about the system (electron number, spin, symmetry information), while the rest of the file defines the values of the integrals. These files define the same 6 and 8 electron Hydrogen test systems as Part I.

The new `det_ops.py` file contains a class, `HAM`, which will store the information required to specify the system, and store the integrals defining the Hamiltonian. In addition, it will also store an input parameter, `p_single`, which will be used to define the probability of randomly generating a single, rather than double-electron excitation in the random excitation generator. Finally, it also defines the reference determinant, in list form, which is given by `ref_det`. The hamiltonian matrix elements are stored in `self.nn` (the nuclear repulsion energy, a scalar), `self.h1` (the one-electron terms, a $2M \times 2M$ array), and `self.h2` (the

[1] A technical point: Since keys in a dictionary in python must be immutable, we will use the 'string' representation of the list to specify a determinant in walker dictionaries, which can be constructed with the `det_str = repr(det)` function. To turn it back from a string representation to a list to extract the information about the orbitals, we can use `det = ast.literal_eval(det_str)`.

two-electron terms, a $2M \times 2M \times 2M \times 2M$ array). Note that it is a convention (called chemical notation) to store the two-electron integrals as

$$v_{pqrs} = \langle pq|rs\rangle = \texttt{self.h2[p, r, q, s]}. \tag{19}$$

I fully expect people to get caught out by this (everyone is)! The `HAM` class also has a method, `read_in_fcidump`, which will read in the specified `FCIDUMP` file for you, and store all the system information appropriately. There are however, several other tasks required to complete the code, which involve completing the functions denoted by `pass` statements in the file.

2.3.1 Tasks

There is no strict order in which the tasks need to be completed, but I suggest the following order to complete the functions required in `det_ops.py`.

1. Implement the Slater-Condon rules of Sec. 2.1.

   Look carefully at the `HAM` class specification, and the docstring of `slater_condon` where it should be implemented. After having written it, it should be tested in isolation with some unit tests. These unit tests have already been written for you at the end of the file, and can be run by simply running the `det_ops.py` file directly in isolation from the rest of the code. They will test various diagonal, single and double excitation matrix elements and check that your code is working. Ensure that these tests are correctly passing before moving on.

2. Complete the `elec_exchange_ops` function.

   This function takes a list of occupied spin-orbitals, where all entries in the list are ordered, apart from (potentially) one. This would represent the state of a determinant after having performed a single electron substitution $p \rightarrow a$, but before the permutations required to compute the parity. The index in the list of this replaced spin-orbital label (a) is also given as an argument. The aim is to compute and return the number of nearest-neighbour, pairwise permutations which would be required to put the occupied orbital list back into an ordered state (sometimes called a 'lineup' operation). This is the main component of computing the 'parity' of an excitation, as detailed in Sec 2.1.4. See the docstring of the function to be completed for more information.

   Again, a set of unit tests have been devised at the end of the file to test the implementation is correct before moving on.

3. Code up a random excitation generator in the `excit_gen` function.

   This functions wants to take in a determinant as an ordered list of occupied spin-orbitals, and return a randomly selected single- or double-electron excitation of this determinant. A single-excitation should be selected to be excited to with a probability given by the parameter `self.p_single`. The function should return:

- The excited determinant as an ordered list of N occupied spin-orbitals (`excited_det`).
- The excitation matrix. This is a list of length 2, with each element being a tuple of the orbitals which are changed in the excitation. See docstring in the `slater_condon` function for an example of this. (`excit_mat`).
- The parity of the excitation. Note that this will be computed by calling the function `elec_exchange_ops`. It will be called once for a single excitation, and twice for a double excitation. Remember to ensure that you reorder the determinant list after each substitution.
- The normalized probability of creating the excitation (`prob`).

In order to test this function, we can plot a histogram of the following quantity for each determinant generated ($|D_{\mathbf{j}}\rangle$) from a test determinant ($|D_{\mathbf{i}}\rangle$):

$$X_{\mathbf{j}} = \frac{n_{\text{gen}}(\mathbf{j}|\mathbf{i})}{N_{\text{att}} \times p_{\text{gen}}(\mathbf{j}|\mathbf{i})}, \tag{20}$$

where N_{att} is the number of excited determinants generated from $|D_{\mathbf{i}}\rangle$, $n_{\text{gen}}(\mathbf{j}|\mathbf{i})$ is the number of times that $|D_{\mathbf{j}}\rangle$ is randomly generated by the algorithm, and $p_{\text{gen}}(\mathbf{j}|\mathbf{i})$ is the normalized probability of the excitation computed by the function. If the probability of excitation is normalized, and proportional to the frequency it is generated, the the histogram over all generated determinants should approach 1 for all excited determinants in the limit that $N_{\text{att}} \to \infty$. Justify to yourself that this is should be the case.

The unit test should be set up to test this, with an arbitrary initial determinant to excite from. Check that your histogram looks 'flat', and that it becomes flatter as increasingly equal to 1.0 for all excitations as N_{att} increases. Finally, also work out the expression for the expected number of unique excitations overall for the excitation generation algorithm used, and ensure that this the same as the number of unique excitations found (which is written out).

4. Complete the `calc_excit_mat_parity` function.

   At certain points in the algorithm (for instance when computing the contributions to the energy estimator), we need to compute the parity and excitation matrix between two given determinants (rather than this being computed at the same time as a random excitation). In this case, we need to complete the `calc_excit_mat_parity` function, which will read in two determinants, and compute the excitation matrix between then (as defined previously), giving a list of length two, of tuples indicating the orbital indices which have been excited *from* the first determinant, and *to* the second determinant.

   In order to test this function, we can randomly generate excitations using the `excit_gen` function, and then compute the parity and excitation matrix again independently using the `calc_excit_mat_parity` function, and check they give the same results. If the

results are not the same, then it will not indicate which function the error is in, but at least with agreement you can be sure that the two approaches are consistent. Note that there is a subtlety in this test, in that the excitation matrix and parity are not unique (see section 2.1.4). We can swap the indices in the excited 'from' orbitals, and/or the excited 'to' orbitals, as long as there is a corresponding change in the parity.

Finally, we need to make changes to the `fciqmc_partII.py` file. This is very similar to the code written in `fciqmc_partI.py`, with some key differences.

1. A `HAM` object is created to read in the system from the desired `FCIDUMP` file, rather than reading in the full Hamiltonian matrix.
2. Determinants are now specified by a list of occupied orbitals, rather than their index.
3. Matrix elements and random excitation generation is now performed via reference to the appropriate functions in the `HAM` object.

Other than this, the structure is almost identical to the (completed) code from Part I. The only task to complete here, will be to add some lines to implement the *initiator* approximation, described in section 2.2. The `pass` statement indicates where this should be added. If the `sim_params.init_thresh` value is `None`, then we can assume that the initiator approximation is not used. Otherwise, this variable will hold the value of n_{init}.

With all of these tasks, feel free to ask for help if you are stuck. If needed, there are worked solutions in the solutions folder, but try to complete the code / ask first!

2.4 Tests and Questions

- With the completed code from Part II, we can compare to calculations on the same system with the completed code of Part I. The code in Part II has removed all explicitly exponentially scaling steps from the algorithm used in Part I. Which is faster, and why?
- What affect does the initiator approximation have on the dynamics of the calculation? Are you able to converge the energy of the 8 hydrogen ring system with fewer walkers? Note that other parameters (e.g. timestep) may also need to change to get a reasonable simulation.
- With the new excitation generator, there is an input parameter which denotes the probability of choosing to create a single rather than double excitation. What do you think is likely to be the optimal value for this?
- The code as it is does not exploit spin polarization symmetry (i.e. the fact that alpha and beta electrons have different spin). How could this be incorporated into the algorithm, and where do you think you would then find efficiency gains? What other (scientific) reasons might you have for ensuring that spin symmetry is explicitly considered?
- List 5 ways that you might improve the efficiency of the algorithm.