Part II

Implementation and Results

In this part we present the implementation of algorithms presented in the thesis. We have implemented the CCD algorithm and also the CCQMC algorithm for two dimensional quantum dots.
This part has the following structure:

- Chapter 7 is devote to implementation details.
    - Section 7.1 present a general structure of the code and description of all classes used.
    - Section 7.2 present algorithm for Hartree-Fock implementation.
    - Section 7.3 present algorithm for deterministic CCD implementation.
    - In Section 7.4 the CCQMC method implementation is described.
- In Chapter 8 we present results and discussion of what have been done so far.

Chapter 7

Implementation

7.1 General structure of the code

Code has the following classes:

1. *generalspclass*
    - *qdotspbasis* - only for QD
    - *qdotHFbasis* - only for QD
    - *electronGasSPBasis* - only for HEG
2. *CoulombFunctions*
3. *symblock*
4. *ccd*
5. *qstate*
6. *channel*
7. *channelindexpair*
8. *channelset*
    - *qdotchannelset* - only for QD
    - *electrongaschannelset*- only for HEG

Program for QD: The base class *generalspclass* has two derived classes *qdotspbasis* and *qdotHFbasis*. This base class generate objects for the solvers and allow us to switch between different bases and systems of interest. The *qdotspbasis* provides TBME, single particle energies for QD in the HO basis as well as other important quantities such as number of electrons, number of shells, number of states etc. The *qdotHFbasis* provides all these quantities in the Hartree-Fock basis.
Program for HEG: The base class *generalspclass* has derived class *electronGasSP-Basis*. It provide all needed information for regarding the system.

7.2 Implementation of Hartree-Fock solver for QD

Fig. 7.1 present the illustration for the HF algorithm. Here is a bit more detailed description of the algorithm:

- Calculate the one-body $\langle\alpha|\hat{h}|\beta\rangle$ and two-body $\langle\alpha\beta|\hat{v}|\gamma\delta\rangle$ matrix elements.

- Begin iteration procedure:

  1. Start with a guess for coefficient matrix. Usually an identity matrix $C^{(0)}_{i\alpha}$, and density matrix $\rho^{(0)}_{\gamma\delta}$. For $\alpha, \gamma, \delta \in N_{\text{st}}$ and $i \in N_p$ (N_{st} - number of states and N_p - number of particles).
  2. The Hartree-Fock matrix on first iteration is computed as

  $$\hat{h}^{\text{HF}}_{\alpha\beta}(0) = \epsilon_\alpha \delta_{\alpha\beta} + \sum_{\gamma\delta}^{N_{\text{st}}} \rho^{(0)}_{\gamma\delta} \langle\alpha\beta|\hat{v}|\gamma\delta\rangle$$

  3. Diagonalize the Hartree-Fock matrix. Compute $C^{(1)}_{i\alpha}$, $\rho^{(1)}_{\gamma\delta}$ and ϵ^{HF}_i
  4. For every iteration check the convergence:

  $$\frac{\sum_i^{N_p} |\epsilon_i^{(n)} - \epsilon_i^{(n-1)}|}{N_{st}} \leq \textbf{tolerance},$$

  where **tolerance** usialy a small number ($\approx 10^{-6}$). If convergence test fail compute new $\hat{h}^{\text{HF}}_{\alpha\beta}$ matrix and repeat the procedure.

- After the convergence test is passed, compute the HF energy and store the last version of coefficient matrix.

7.2.1 Code structure for HF

The Hartree-Fock algorithm uses the following methods:

- *CalculateSPenergies* to compute the single-particle energies,
- *fillTwoBodyElements* to compute the Coulomb integrals,
- *computeHFmatrix* to set up the Hartree-Fock matrix,
- *computeHartreeFockEnergyDifference* method used to test the convergence.
- *computeHartreeFockEnergy* to compute the ground state energy itself.

The code for algorithm is presented on Fig. 7.2.

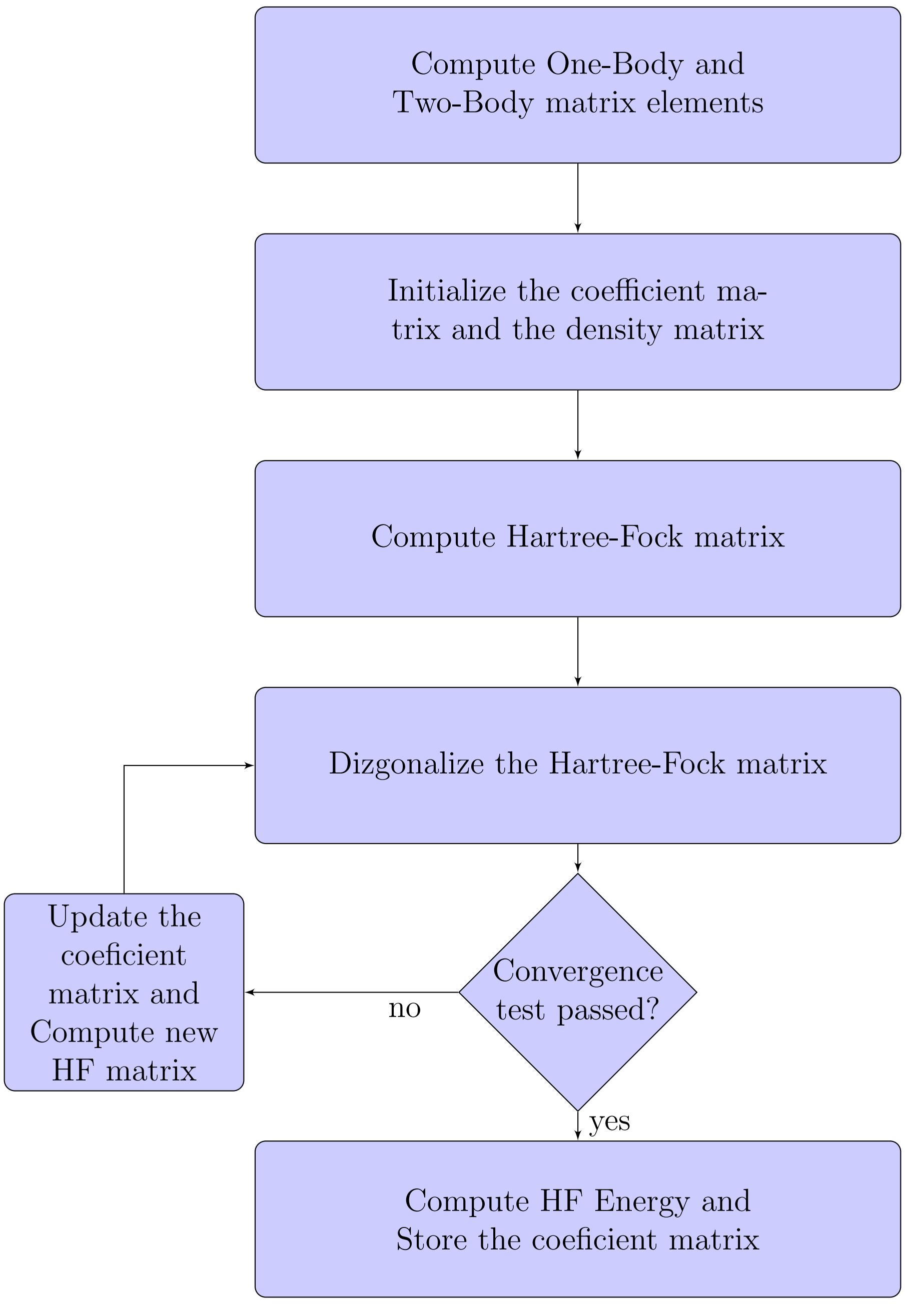

Compute One-Body and Two-Body matrix elements
Initialize the coefficient matrix and the density matrix
Compute Hartree-Fock matrix
Dizgonalize the Hartree-Fock matrix
Convergence test passed?
no
yes
Update the coeficient matrix and Compute new HF matrix
Compute HF Energy and Store the coeficient matrix

Figure 7.1: Flow chart for Hartree-Fock algorithm

```
void qdotHFbasis::applyHartreeFockMethod(){
  int NumberOfStates = m_shells.size();
  arma::mat C(NumberOfStates, NumberOfStates);
  C.eye();
  setCoefficientMatrix (C);
  double difference = 10;
  double epsilon = 10e-8;
  eigval_previous .zeros(NumberOfStates);
  while (epsilon < difference && i < 200){
    arma::mat x_DensityMatrix = computeDensityMatrix();
    computeHFmatrix(x_DensityMatrix);
    arma::eig_sym( eigval , eigvec , m_HF);
    setCoefficientMatrix (eigvec);
    difference = computeHartreeFockEnergyDifference();
    eigval_previous = eigval;
  }
}
```

Figure 7.2: Implementation of Hartree-Fock algorithm

```
void qdotHFbasis::computeHFmatrix(arma::mat DensityMatrix){
  int NumberOfStates = m_shells.size();
  m_HF.zeros(NumberOfStates,NumberOfStates);
  double FockElement = 0;

  for(int i = 0; i < NumberOfStates; i++) {
    for(int j = 0; j < NumberOfStates; j++) {
      for(int k = 0; k < NumberOfStates; k++) {
        for(int l = 0; l < NumberOfStates; l++) {
          FockElement += DensityMatrix(k,l)*TBME(i,k,j,l);
          if (FockElement !=0.0){
          }
        }
      }
      if (i == j) {
        m_HF(i, i) += m_HOEnergies.at(i);
      }
      m_HF(i, j) += FockElement;
      FockElement = 0.0;
    }
  }
}
```

Figure 7.3: Compute HF matrix

7.3 Implementation for CCD

The CCD algorithm is same both for QD and HEG. The equations for CCD amplitudes should be written in in iterative manner and then computed and updated before the convergence is reached.

In the box below the main steps for CCD algorithm are presented:

- Compute the MBPT2 Energy
- Set up Initial Amplitudes
- Begin iteration procedure:
  1. Calculate New Amplitudes.
  2. Compute the Correlation Energy.
  3. Compute Energy Difference = $|E^{(n-1)}_{Corr} - E^{(n)}_{Corr}|$.
  4. If Energy Difference > **tolerance**:
     Update Energy and Amplitudes.
  5. Repeat iterations until Energy Difference become less then **tolerance**.
- After the convergence test is passed, compute and store the CCD correlation energy.

The algorithm described above imply the convergence can be reached after a finite number of iterations. However this is not always the case. For some set of parameters one may obtain divergent solutions or at least solutions that require a huge number of iterations. One of the solutions to such problem might be to use some parameter, known as "mixing parameter" to update the amplitudes in the following way:

$$t^n = at^(n) + (1-a)t^{(n-1)}, \tag{7.1}$$

here a $\in (0,1]$ in "mixing parameter", t^{n-1} - amplitudes on the previous iteration and t^n- amplitudes on the current iteration.

Channels for CCD

The straightforward implementation of coupled cluster algorithm is not very efficient. It require both memory and CPU we can't afford using a common modern PC. However one may utilize symmetries in Hamiltonian to reduce both time and memory usage. Let's consider first the two-body matrix elements $\langle pq||rs\rangle$ which can be considered as a transition probability between states r, s and p, q.
For the QD:

Such transition is prohibited if it does not preserve quantum numbers for spin and angular momentum. The preserved quantum numbers are the following:

$$m_r + m_s = m_p + m_q, \tag{7.2}$$
$$s_r + s_s = s_p + s_q. \tag{7.3}$$

For the HEG:
Such transition is prohibited if it does not preserve quantum numbers for spin and wave number. The preserved quantities are the following:

$$\mathbf{k}_r + \mathbf{k}_s = \mathbf{k}_p + \mathbf{k}_q, \tag{7.4}$$
$$s_r + s_s = s_p + s_q. \tag{7.5}$$

Here is no principal difference between these two besides the mentioned above. Further in the text we are going to use single particle states, not a conserved quantities, so the theoretical description is the same for both systems. If this conditions is not met we have $\langle ab||cd\rangle = 0$.
For amplitude equation presented in (5.37) we need matrix elements of the following form:

$$\langle ab||ij\rangle, \tag{7.6}$$
$$\langle kl||ij\rangle, \tag{7.7}$$
$$\langle ab||cd\rangle, \tag{7.8}$$
$$\langle kb||cj\rangle, \tag{7.9}$$
$$\langle kl||cd\rangle. \tag{7.10}$$
$$\tag{7.11}$$

As we have already mentioned indexes $a, b, c, d, ...$ correspond to states which are not occupied in the reference determinant and are called particle states, while indexes $i, j, k, l, ...$ correspond to states which are occupied in the reference determinant and are called hole states. Taking into account the fact that we can possible make only some combinations of two states, namely hole-hole, particle-particle, particle-hole and hole-particle pairs we only are able to set up some specific types of interaction matrices:

$$V_{pppp}, V_{ppph}, V_{pphh}, V_{phhh}, V_{hhpp}, V_{hhhp}, V_{hppp}, V_{hhhh}. \tag{7.12}$$

All together there are eight matrices. Splitting the TBME matrix into parts is not the only thing here, the main idea is to right each block into a "channel" to satisfy the conservation of quantum numbers is the goal. Let's consider how this cam by done.
We start with creating a vectors for combination of two states. On Fig. 7.4 and 7.5 one may see how this is implemented in C++ for QD and HEG respectively. The

code is almost the same, except for the number of quantities to be preserved in the system. We mapping two states into one state which is a sum of the two:

$$|ab\rangle \to |A\rangle\,, \tag{7.13}$$
$$|ij\rangle \to |I\rangle\,, \tag{7.14}$$
$$|ai\rangle \to |AI\rangle\,, \tag{7.15}$$
$$|ia\rangle \to |IA\rangle\,. \tag{7.16}$$
$$\tag{7.17}$$

All new states have quantum numbers which are just sums of the original ones. For example, $|A\rangle$ has quantum numbers $m_A = m_a + m_b$ and $s_A = s_a + s_b$. We loop over all possible sum's for spin and angular momentum, which is just $\{-2, 0, 2\}$ for spin and $\{-2(R-1), \ldots 0, \ldots, 2(R-1)\}$. Here R is shell number. After doing so we have vectors with pairs of states which satisfy the conservation laws. This allow us to store only non-zero matrix elements.

Having such vectors we may build up all needed matrices. Let's consider how this is done on Fig. 7.6. For every matrix we need $(V_{pppp}, V_{pphh}, V_{hhpp}, V_{hhhh})$ we pick two vectors with the indexes corresponding to needed combination of states and store the matrix element into one of blocks. This is done for all channels that have been created when we set up the vectors. We do not do this for matrices like V_{phhh} due to the reasons that will be discussed below in this section.

Mapping two states into a single one has even more advantages, we now may use the matrix-matrix multiplications to compute the correlation energy and amplitudes. Here is a brief explanation of how it works. Let's take for example the second leaner term [1] in (5.37):

$$\sum_{cd} \langle ab|v|cd\rangle\, t_{cd}^{ij} \to V_{ab}^{cd} \times T_{cd}^{ij} \to V_A^C \times T_C^I \to (VT)_A^I, \tag{7.18}$$

which is just an ordinary matrix-matrix multiplication.
However not all terms in (5.37) can be tackled as easy as this one. For example, the third leaner term:

$$\sum_{kc} \langle kb|v|cj\rangle\, t_{ik}^{ac} \to V_{kb}^{cj} \times T_{ik}^{ac}, \tag{7.19}$$

cannot be straightforward implemented as matrix-matrix multiplication. In order to do this we need to perform a following transformation:

$$\sum_{kc} \langle kb|v|cj\rangle\, t_{ik}^{ac} \to \sum_{kc} \langle kc|\tilde{v}|jb\rangle\, \tilde{t}_{ai}^{kc} \to \tilde{V}_{kc}^{jb} \times \tilde{T}_{ai}^{kc} \to \tilde{V} \times \tilde{T}. \tag{7.20}$$

[1]The name linear is used to denote that we have only one amplitude to multiply with interaction matrix element. Terms with two amplitudes are considered quadratic.

```
void qdotchannelset::setUpChannels(generalSPclass * qsystem){
qsys = qsystem;
int Mlmax = 2*(qsys->getShellsStochastic()-1);
int Smax = 2;
for(int Ml = -Mlmax; Ml <= Mlmax; Ml++){
  for(int S = -Smax; S <= Smax; S = S + 2){
    ChannelVariety.emplace_back(channel());
    for(int i = 0; i < qsys->getFermiLevel(); i++){
      for(int j = 0; j < qsys->getFermiLevel(); j++){
        qstate *QuantumState = qsys->sumState(i,j);
        if(Ml == QuantumState->m() && S == QuantumState->s() && i != j){
          ChannelVariety.back().m_HoleHoleVec.emplace_back(channelindexpair());
          ChannelVariety.back().m_HoleHoleVec.back().set(i, j);
        } delete QuantumState;
      }
    }
    for(int a = qsys->getFermiLevel(); a < qsys->getStatesStochastic(); a++){
      for(int b = qsys->getFermiLevel(); b < qsys->getStatesStochastic(); b++){
        qstate *QuantumState = qsys->sumState(a,b);
        if(Ml == QuantumState->m() && S == QuantumState->s() && a != b){
          ChannelVariety.back(). m_ParticleParticleVec .emplace_back(channelindexpair());
          ChannelVariety.back(). m_ParticleParticleVec .back().set(a, b);
        } delete QuantumState;
      }
    }
    for(int c = qsys->getFermiLevel(); c < qsys->getStatesStochastic(); c++){
      for(int k = 0; k < qsys->getFermiLevel(); k++){
        qstate *QuantumState = qsys->sumState(c,k);
        if(Ml == QuantumState->m() && S == QuantumState->s() && c != k){
          ChannelVariety.back().m_ParticleHoleVec.emplace_back(channelindexpair());
          ChannelVariety.back().m_ParticleHoleVec.back().set(c, k);
        } delete QuantumState;
      }
    }
    for(int l = 0; l < qsys->getFermiLevel(); l++){
      for(int d = qsys->getFermiLevel(); d < qsys->getStatesStochastic(); d++){
        qstate *QuantumState = qsys->sumState(l,d);
        if(Ml == QuantumState->m() && S == QuantumState->s() && l != d){
          ChannelVariety.back().m_HoleParticleVec.emplace_back(channelindexpair());
          ChannelVariety.back().m_HoleParticleVec.back().set(l, d);
        } delete QuantumState;
      }
    }
  }
}
}
```

Figure 7.4: Setting up channels for QD

```
void electrongaschannelset ::setUpChannels(generalSPclass * qsystem){
qsys = qsystem;
int Nmax = qsys->getShellsStochastic() - 1;
int Smax = 2;
for(int Nx = -Nmax; Nx <= Nmax; Nx++){
  for(int Ny = -Nmax; Ny <= Nmax; Ny++){
    for(int Nz = -Nmax; Nz <= Nmax; Nz++){
      for(int S = -Smax; S <= Smax; S = S + 2){

        ChannelVariety.emplace_back(channel());
        for(int i = 0; i < qsys->getFermiLevel(); i++){
          for(int j = 0; j < qsys->getFermiLevel(); j++){
            if(Nx == qsys->sumState(i,j).nx() && Ny == qsys->sumState(i,j).ny() &&
                Nz == qsys->sumState(i,j).nz() && S == qsys->sumState(i,j).s() && i
                != j){
            ChannelVariety.back().m_HoleHoleVec.emplace_back(channelindexpair());
            ChannelVariety.back().m_HoleHoleVec.back().set(i, j);
            }
          }
        }
        for(int a = qsys->getFermiLevel(); a < qsys->getStatesStochastic(); a++){
          for(int b = qsys->getFermiLevel(); b < qsys->getStatesStochastic(); b++){
            if(Nx == qsys->sumState(a,b).nx() && Ny == qsys->sumState(a,b).ny()
                && Nz == qsys->sumState(a,b).nz() && S ==
                qsys->sumState(a,b).s() && a != b){
            ChannelVariety.back(). m_ParticleParticleVec .emplace_back(channelindexpair());
            ChannelVariety.back(). m_ParticleParticleVec .back().set(a, b);
            }
          }
        }
      }
    }
  }
}
```

Figure 7.5: Setting up channels for HEG.

```
void ccd :: setUpInterractionMatrixBlocks (){
  for(channel onechannel : ChannelVariety){
    m_ppppVBlock.emplace_back(symblock(onechannel.m_ParticleParticleVec.size(),
        onechannel. m_ParticleParticleVec . size ()));
    for(unsigned int ab = 0; ab < onechannel. m_ParticleParticleVec . size (); ab++){
      channelindexpair AB = onechannel.m_ParticleParticleVec.at(ab);
      for(unsigned int cd = 0; cd < onechannel. m_ParticleParticleVec . size (); cd++){
        channelindexpair CD = onechannel.m_ParticleParticleVec.at(cd);
        m_ppppVBlock.back().setElement((int)ab, (int)cd, qsys->TBME(AB.first(),
            AB.second(), CD.first(), CD.second())) ;
      }
    }
    m_hhhhVBlock.emplace_back(symblock(onechannel.m_HoleHoleVec.size(),
        onechannel.m_HoleHoleVec.size()));
    for(unsigned int kl = 0; kl < onechannel.m_HoleHoleVec.size(); kl++){
      channelindexpair KL = onechannel.m_HoleHoleVec.at(kl);
      for(unsigned int ij = 0; ij < onechannel.m_HoleHoleVec.size(); ij++){
        channelindexpair IJ = onechannel.m_HoleHoleVec.at(ij);
        m_hhhhVBlock.back().setElement((int)kl,(int) ij , qsys->TBME(KL.first(),
            KL.second(), IJ. first (), IJ.second()));
      }
    }

    m_hhppVBlock.emplace_back(symblock(onechannel.m_HoleHoleVec.size(),
        onechannel.m_ParticleParticleVec. size ()));
    for(unsigned int kl = 0; kl < onechannel.m_HoleHoleVec.size(); kl++){
      channelindexpair KL = onechannel.m_HoleHoleVec.at(kl);
      for(unsigned int cd = 0; cd < onechannel. m_ParticleParticleVec . size (); cd++){
        channelindexpair CD = onechannel.m_ParticleParticleVec.at(cd);
        m_hhppVBlock.back().setElement((int)kl,(int)cd, qsys->TBME(KL.first(),
            KL.second(), CD.first(), CD.second())) ;
      }
    }
    m_pphhVBlock.emplace_back(symblock(onechannel.m_ParticleParticleVec.size(),
        onechannel.m_HoleHoleVec.size()));
    for(unsigned int kl = 0; kl < onechannel.m_HoleHoleVec.size(); kl++){
      channelindexpair KL = onechannel.m_HoleHoleVec.at(kl);
      for(unsigned int cd = 0; cd < onechannel. m_ParticleParticleVec . size (); cd++){
        channelindexpair CD = onechannel.m_ParticleParticleVec.at(cd);
        m_pphhVBlock.back().setElement((int)cd,(int) kl , qsys->TBME(CD.first(),
            CD.second(), KL.first(), KL.second())) ;
      }
    }
  }
}
```

Figure 7.6: Setting up interaction matrix

```
for(int l = 0; l < qsys->getFermiLevel(); l++){
  for(int d = qsys->getFermiLevel(); d < qsys->getStatesStochastic(); d++){
    qstate QuantumState = qsys->substractState(l,d);
    if(Ml == QuantumState.m() && S == QuantumState.s() && l != d){
      ChannelVariety.back().m_HoleMinusParticleVec.emplace_back(channelindexpair());
      ChannelVariety.back().m_HoleMinusParticleVec.back().set(l, d);
    } //delete QuantumState;
  }
}

for(int c = qsys->getFermiLevel(); c < qsys->getStatesStochastic(); c++){
          for(int k = 0; k < qsys->getFermiLevel(); k++){
            qstate QuantumState = qsys->substractState(c,k);
            if(Ml == QuantumState.m() && S == QuantumState.s() && c != k){
              ChannelVariety.back().m_ParticleMinusHoleVec.emplace_back(channelindexpair());
              ChannelVariety.back().m_ParticleMinusHoleVec.back().set(c, k);
            } //delete QuantumState;
          }
}
```

Figure 7.7: Setting up channels for aligned clusters.

It is important to take into account all preserved quantities while we make such permutation of indexes. The new product should obey the same conservation laws as the original one. For the term under consideration that means the following transformation:

$$k + b = c + j = a + c = i + k \Rightarrow k - c = j - b = a - i = k - c. \tag{7.21}$$

Such transformation require a new block interaction matrix. In order to set it up we need to set up the vectors with indexes that correspond the transformation described in (7.21). As one can see we transformed "hole plus particle" indexes into "hole minus particle". On Fig. 7.7 we first set up the vectors with index pairs corresponding to such states. After this is done we can set up the corresponding interaction matrix itself the same way as we did it before.

Other terms in the amplitude equitation can be considered in a similar manner. For the first quadratic term does not require any transformation and can be expressed as a product of three matrices:

$$\sum_{klcd} \langle kl|v|cd\rangle\, t_{ij}^{cd} t_{kl}^{ab} \to \tilde{V}_{kl}^{cd} \times \tilde{T}_{ij}^{kl} \times \tilde{T}_{kl}^{ab}. \tag{7.22}$$

Other terms are a bit more complicated, but are based on the same technique:

$$\sum_{klcd} \langle kl||cd\rangle\, t^{ik}_{ac} t^{jl}_{bd} \to \tilde{T}^{ck}_{ai} \times \tilde{V}^{ld}_{ck} \times \tilde{T}^{dj}_{ld} \tag{7.23}$$

$$\sum_{klcd} \langle kl||cd\rangle\, t^{dc}_{ik} t^{ab}_{lj} \to \tilde{T}^{abj}_{l} \times \tilde{V}^{l}_{kcd} \times \tilde{T}^{kcd}_{i}, \tag{7.24}$$

$$\sum_{klcd} \langle kl||cd\rangle\, t^{ac}_{lk} t^{db}_{ij} \to \tilde{T}^{a}_{klc} \times \tilde{V}^{klc}_{d} \times \tilde{T}^{d}_{bij}. \tag{7.25}$$

Transformation for (7.23) is trivial, but (7.24) and (7.25) need some comments. For the two last quadratic terms in (5.37) we need to map three states into a single one. In order to do this we need to set up new vectors and interaction matrix blocks. On Fig. 7.8 we present a method to store indexes needed for such transformation - three indexes are stored into one "hole plus hole minus particle" state and one into a single "particle" state. Transformation is made in a similar manner as it has been made in (7.21) for leaner term, the only difference is that now we need to change the number of indexes not only their positions. However all quantum numbers should be preserved according to the same rules as before.
After this transformation we use matrix multiplication for each term and compute amplitudes on the next iteration.

7.4 CCQMC implementation

In Chapter on Monte Carlo 6 we have discussed many different Monte Carlo methods. The section 6.3.2 is devoted to CCQMC algorithm. In this section we present some details how the alrgorithm has been implemented in the C++. Our program aims to simulate equation (6.25). This is done by *applyCCQMC* method of the *ccqmc* class. The main idea of the method is to move from the iterative scheme to the numerical sampling of the expectation value of the Hamiltonian. In order to do so we need first to sample the wave function and then sample action of the Hamiltonian. This sampling procedure is presented on Fig. 7.9. Sampling of the wave function can be described as several independent sampling processes inside one Monte Carlo iteration. This comes from the fact that we are dealing with cluster of different sizes while doing so. As soon as we only implement the method for the HEG we do not have single excitations and would only deal with doubles. The cluster sizes we have to sample can be limited to cluster size one, two and three. All can be sampled independently and then combined on the annihilation step. However such limitation also means we are not going to have triples in the simulation, as soon as we only limit ourselves to doubles and the combined clusters can only produce the quadrupole excitations, which we do not store anyway as we want to compare our CCQMC with the deterministic solver for CCD.

Before starting the simulation there are no excips in the space of excited determinants, only those on reference. In the FCIQMC or DMC it is common to begin

```
for(int Ml = -Mlmax; Ml <= Mlmax; Ml++){
    for(int S = -Smax; S <= Smax; S++){
      ChannelVariety2.emplace_back(channel());
      for(int a = 0; a < qsys->getStatesStochastic(); a++){
        qstate  OneQS = qsys->oneState(a);
        if(Ml == OneQS.m() && S == OneQS.s()){
          for(int i = 0; i < qsys->getFermiLevel(); i++){
            for(int j = 0; j < qsys->getFermiLevel(); j++){
              for(int b = qsys->getFermiLevel(); b < qsys->getStatesStochastic();
                  b++){
          qstate QuantumState = qsys->sumSubstractState(i,j,b);
          if( QuantumState.m() == OneQS.m()
          && QuantumState.s() == OneQS.s()){
          ChannelVariety2.back().m_HolePlusHoleMinusParticleVec.emplace_back(channelindexpair());
          ChannelVariety2.back().m_HolePlusHoleMinusParticleVec.back().setThree(i, j,
              b);
          ChannelVariety2.back().m_ParticleVec.emplace_back(channelindexpair());
          ChannelVariety2.back().m_ParticleVec.back().setOne(a);
                }
              }
            }
          }
        }
      }
    }
}
```

Figure 7.8: Setting up channels for rotated clusters.

with just a single walker on the $|D_0\rangle$, however this is not the case for CCQMC, we have to start with a number of walkers, usually in range form 100 to 1000. After the simulation begins the population of excitor space start to grow.

A single iteration after some time is then goes af follows:

- Function *sampleReference* select clusters of size zero, which is the reference determinant, functions *sampleClusterSizeOne* and *sampleClusterSizeTwo* select clusters of size one, which is a double excitation of the reference and cluster of size two respectively.
- All exitors that have been spawned or killed are then stored in the *TemporaryStorage* vector and the corresponding excips are stored in vector *TemporaryStoragePopulation.*
- In the end of one MC iteration this two vectors are used to update *StorageCumulativeDeterminants* vector and *StorageCumulativePopulation.* These two contain the total number of excips and excitors after the time $n\delta\tau$, here n is number of iterations done so far.
- After this all temporary vectors are cleared and the number of excips in the excitor space is updated. This value is used on next iteration to calculate the number of sampling attempts. There are many different schemes of sampling, in this simulation we used the following: $N_{attempts}(\tau) = N_0(\tau-1) + N_{ex}(\tau-1)$, here $N_{attempts}(\tau)$ is number of sampling attempts on current step, $N_0(\tau-1)$ is reference population on the previous step and $_{ex}(\tau-1)$ is excitor space population on previous step.

In the box below we present a short description of what is done here:

- Set up parameters: $\delta\tau$, maximum number of iterations, N_0.
- Sample Cluster Size Zero N_0 times before the iterations start to create initial population in excitor space.
  1. Sample Cluster Size Zero N_0 times.
  2. Sample Cluster Size Zero $\frac{2}{3}N_{ex}$ times.
  3. Sample Cluster Size Zero $\frac{1}{3}N_{ex}$ times.
  4. Add population of excips on current iteration to the cumulative population.
  5. Update τ as $\tau + \delta\tau$.
- End the simulation if maximum number of iterations have been reached.

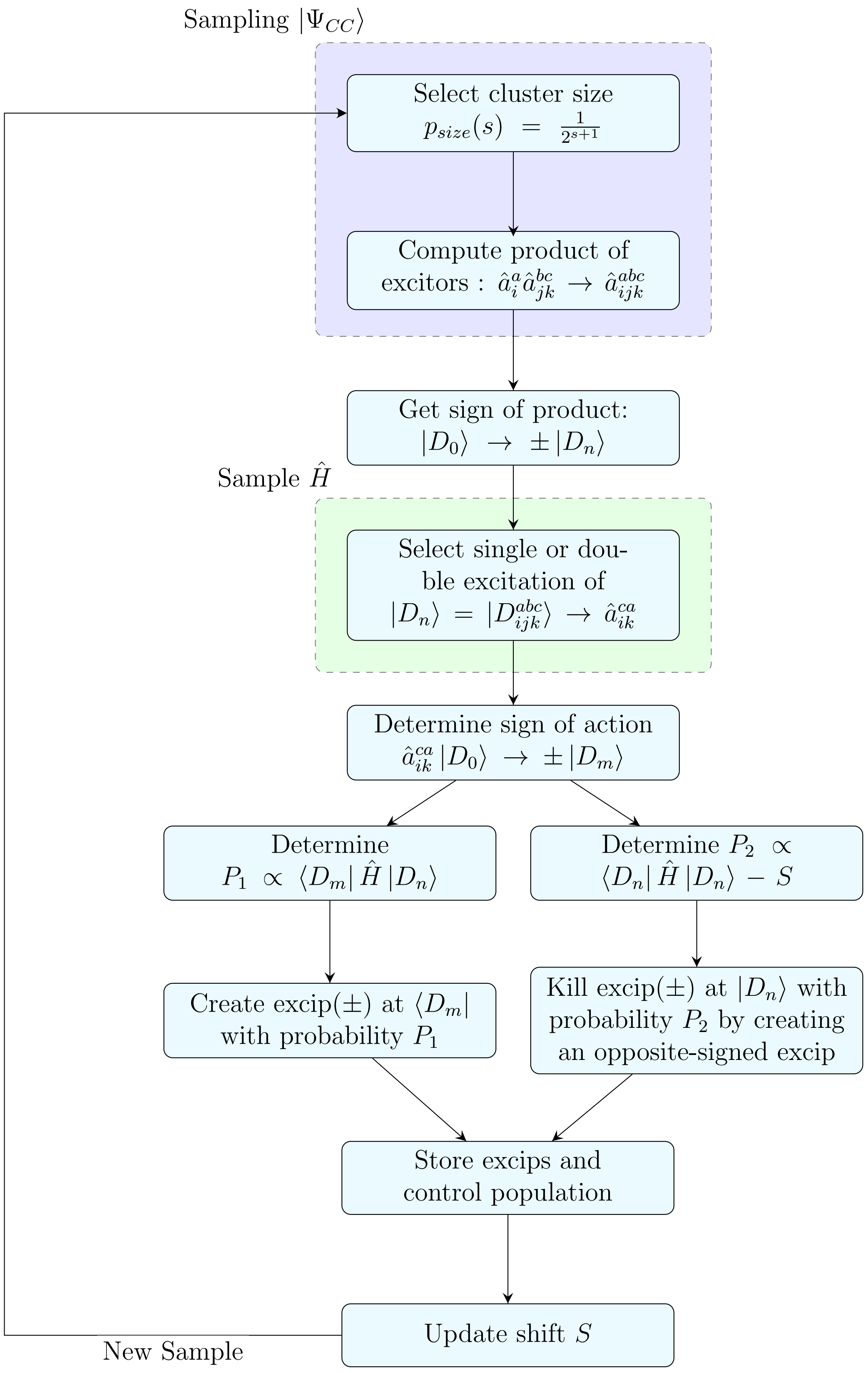
Here we need to mention sign of excip to be killed or spawned. Killing process does not cause any difficulties because we just create excip with the opposite sing.

However the spawning process is rather complicated. First we consider the excips spawned from reference. In order to determine the sign we need to take into account the following:

- sign of H_{n0}.
- possible sign change if the from reordering of the creation and annihilation operators. This comes from the fact that we store determinants in an ordered way, so that any excited determinant $|D_{ij}^{ab}\rangle$ should satisfy the following rule $a < b$ and $i < j$. If it is not the case we reorder operators and should change the sign of excip is the number of permutation needed for the reordering is odd.

For the spawning from other determinants we should consider some additional possible sources of sign change:

- sign of H_{nm}.
- possible sign change if the from reordering of the creation and annihilation operators when collapsing selected clusters to a single determinant.
- possible sign change due to action on reference, both for the determinants in "bra" and "ket" parts.
- sing of cluster amplitude.
- possible sign change after acting with randomly chosen excitation on reference (sign of excitor itself).



Sampling $|\Psi_{CC}\rangle$
Select cluster size $p_{size}(s) = \frac{1}{2^{s+1}}$
Compute product of excitors : $\hat{a}_i^a \hat{a}_{jk}^{bc} \rightarrow \hat{a}_{ijk}^{abc}$
Get sign of product: $|D_0\rangle \rightarrow \pm |D_n\rangle$
Sample $\hat{H}$
Select single or double excitation of $|D_n\rangle = |D_{ijk}^{abc}\rangle \rightarrow \hat{a}_{ik}^{ca}$
Determine sign of action $\hat{a}_{ik}^{ca} |D_0\rangle \rightarrow \pm |D_m\rangle$
Determine $P_1 \propto \langle D_m| \hat{H} |D_n\rangle$
Determine $P_2 \propto \langle D_n| \hat{H} |D_n\rangle - S$
Create excip($\pm$) at $\langle D_m|$ with probability P_1
Kill excip($\pm$) at $|D_n\rangle$ with probability P_2 by creating an opposite-signed excip
Store excips and control population
Update shift S
New Sample

Figure 7.9: Flow chart for CCQMC sampling.

Chapter 8

Results and discussion

8.1 The CCD results

As we have already mentioned the HEG is a very convenient system to study various many-body methods. In particular we have found many other researches who have implemented the CCD approximation for such system. That allow us to test our solution against results obtained in many other works on the topic. In particular we have compared our results with some other students from our department, namely with Miller [22], Hansen [16] and [14]. Table 8.1 present the comparison of the results. As one can see we manage to reproduce their results up to a chosen **tolerance** for CCD (in this case 10^{-6}).