Slides from FYS4411 Lectures

Morten Hjorth-Jensen & Gustav R. Jansen

[1]Department of Physics and Center of Mathematics for Applications
University of Oslo, N-0316 Oslo, Norway

Spring 2012

Topics for Weeks 10-15, March 5 - April 15

Slater determinant, minimization and programming strategies

- Many electrons and Slater determinant
- How to implement the Slater determinant
- Minimizing the energy expectation value (Conjugate gradient method)
- Optimization

Project work: Project 1 should be about done before the end of the period. Project 2 will be presented after easter.

Slater determinants

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N, \alpha, \beta, \ldots, \sigma) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) & \ldots & \ldots & \psi_\alpha(\mathbf{r}_N) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \ldots & \ldots & \psi_\beta(\mathbf{r}_N) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \psi_\sigma(\mathbf{r}_1) & \psi_\sigma(\mathbf{r}_2) & \ldots & \ldots & \psi_\gamma(\mathbf{r}_N) \end{vmatrix}, \quad (1)$$

where $\mathbf{r}_i$ stand for the coordinates and spin values of a particle i and $\alpha, \beta, \ldots, \gamma$ are quantum numbers needed to describe remaining quantum numbers.

Slater determinants

The potentially most time-consuming part is the evaluation of the gradient and the Laplacian of an N-particle Slater determinant. We have to differentiate the determinant with respect to all spatial coordinates of all particles. A brute force differentiation would involve $N \cdot d$ evaluations of the entire determinant which would even worsen the already undesirable time scaling, making it $Nd \cdot \mathcal{O}(N^3) \sim \mathcal{O}(d \cdot N^4)$. This poses serious hindrances to the overall efficiency of our code.
The efficiency can be improved however if we move only one electron at the time. The Slater determinant matrix $\mathcal{D}$ is defined by the matrix elements

$$d_{ij} \equiv \phi_j(x_i) \tag{2}$$

where $\phi_j(\mathbf{r}_i)$ is a single particle wave function. The columns correspond to the position of a given particle while the rows stand for the various quantum numbers.

Slater determinants

What we need to realize is that when differentiating a Slater determinant with respect to some given coordinate, only one row of the corresponding Slater matrix is changed. Therefore, by recalculating the whole determinant we risk producing redundant information. The solution turns out to be an algorithm that requires to keep track of the *inverse* of the Slater matrix.
Let the current position in phase space be represented by the $(N \cdot d)$-element vector $\mathbf{r}^{\text{old}}$ and the new suggested position by the vector $\mathbf{r}^{\text{new}}$.
The inverse of $\mathcal{D}$ can be expressed in terms of its cofactors C_{ij} and its determinant $|\mathcal{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\mathcal{D}|} \tag{3}$$

Notice that the interchanged indices indicate that the matrix of cofactors is to be transposed.

Slater determinants

If $\mathcal{D}$ is invertible, then we must obviously have $\mathcal{D}^{-1}\mathcal{D} = \mathbf{1}$, or explicitly in terms of the individual elements of $\mathcal{D}$ and $\mathcal{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik}\, d_{kj}^{-1} = \delta_{ij} \tag{4}$$

Consider the ratio, which we shall call R, between $|\mathcal{D}(\mathbf{r}^{\text{new}})|$ and $|\mathcal{D}(\mathbf{r}^{\text{old}})|$. By definition, each of these determinants can individually be expressed in terms of the ith row of its cofactor matrix

$$R \equiv \frac{|\mathcal{D}(\mathbf{r}^{\text{new}})|}{|\mathcal{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} \tag{5}$$

Slater determinants

Suppose now that we move only one particle at a time, meaning that $\mathbf{r}^{\text{new}}$ differs from $\mathbf{r}^{\text{old}}$ by the position of only one, say the ith, particle. This means that $\mathcal{D}(\mathbf{r}^{\text{new}})$ and $\mathcal{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the ith row. Recall also that the ith row of a cofactor matrix $\mathcal{C}$ is independent of the entries of the ith row of its corresponding matrix $\mathcal{D}$. In this particular case we therefore get that the ith row of $\mathcal{C}(\mathbf{r}^{\text{new}})$ and $\mathcal{C}(\mathbf{r}^{\text{old}})$ must be equal. Explicitly, we have:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\, j \in \{1, \ldots, N\} \tag{6}$$

Slater determinants

Inserting this into the numerator of eq. (5) and using eq. (3) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{7}$$

Slater determinants

Now by eq. (4) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{8}$$

What this means is that in order to get the ratio when only the ith particle has been moved, we only need to calculate the dot product of the vector $\left(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}})\right)$ of single particle wave functions evaluated at this new position with the ith column of the inverse matrix $\mathcal{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $\mathcal{O}(N)$. The only extra thing we need to do is to maintain the inverse matrix $\mathcal{D}^{-1}(\boldsymbol{x}^{\text{old}})$.

Slater determinants

The scheme is also applicable for the calculation of the ratios involving derivatives. It turns out that differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ changes only the ith row of the corresponding Slater matrix.

Slater determinants

The gradient and Laplacian can therefore be calculated as follows:

$$\frac{\boldsymbol{\nabla}_i |\mathcal{D}(\mathbf{r})|}{|\mathcal{D}(\mathbf{r})|} = \sum_{j=1}^{N} \boldsymbol{\nabla}_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \boldsymbol{\nabla}_i \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}) \tag{9}$$

and

$$\frac{\nabla_i^2 |\mathcal{D}(\mathbf{r})|}{|\mathcal{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}) \tag{10}$$

Slater determinants

If the new position $\mathbf{r}^{\text{new}}$ is accepted, then the inverse matrix can by suitably updated by an algorithm having a time scaling of $\mathcal{O}(N^2)$. This algorithm goes as follows. First we update all but the ith column of $\mathcal{D}^{-1}$. For each column $j \neq i$, we first calculate the quantity:

$$S_j = (\mathcal{D}(\mathbf{r}^{\text{new}}) \times \mathcal{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}})\, d_{lj}^{-1}(\mathbf{r}^{\text{old}}) \tag{11}$$

The new elements of the jth column of $\mathcal{D}^{-1}$ are then given by:

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{S_j}{R}\, d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \qquad \begin{array}{l} \forall\ k \in \{1, \ldots, N\} \\ j \neq i \end{array} \tag{12}$$

Slater determinants

Finally the elements of the ith column of $\mathcal{D}^{-1}$ are updated simply as follows:

$$d_{ki}^{-1}(\mathbf{r}^{\text{new}}) = \frac{1}{R}\, d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \forall\ k \in \{1, \ldots, N\} \tag{13}$$

We see from these formulas that the time scaling of an update of $\mathcal{D}^{-1}$ after changing one row of $\mathcal{D}$ is $\mathcal{O}(N^2)$.

Slater determinants

Thus, to calculate all the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions ($\boldsymbol{\nabla}_i \phi_j(\mathbf{r}_i)$ and $\nabla_i^2 \phi_j(\mathbf{r}_i)$) and the elements of the corresponding inverse Slater matrix ($\mathcal{D}^{-1}(\mathbf{r}_i)$). A calculation of a single derivative is by the above result an $\mathcal{O}(N)$ operation. Since there are $d \cdot N$ derivatives, the time scaling of the total evaluation becomes $\mathcal{O}(d \cdot N^2)$. With an $\mathcal{O}(N^2)$ updating algorithm for the inverse matrix, the total scaling is no worse, which is far better than the brute force approach yielding $\mathcal{O}(d \cdot N^4)$.
Important note: In most cases you end with closed form expressions for the single-particle wave functions. It is then useful to calculate the various derivatives and make separate functions for them.

Slater determinant: Explicit expressions for various Atoms, beryllium

The Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \frac{1}{\sqrt{4!}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

The Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. But we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down.

Slater determinant: Explicit expressions for various Atoms, beryllium

We can rewrite it as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = Det\uparrow(1,2)Det\downarrow(3,4) - Det\uparrow(1,3)Det\downarrow(2,4)$$

$$-Det\uparrow(1,4)Det\downarrow(3,2) + Det\uparrow(2,3)Det\downarrow(1,4) - Det\uparrow(2,4)Det\downarrow(1,3)$$

$$+Det\uparrow(3,4)Det\downarrow(1,2),$$

where we have defined

$$Det\uparrow(1,2) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{vmatrix},$$

and

$$Det\downarrow(3,4) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

The total determinant is still zero!

Slater determinant: Explicit expressions for various Atoms, beryllium

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin.
It can be shown, see for example Moskowitz and Kalos, Int. J. Quantum Chem. **20** (1981) 1107, that for the variational energy we can approximate the Slater determinant as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) \propto Det \uparrow (1,2) Det \downarrow (3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N) \propto Det \uparrow Det \downarrow,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (3 for six-electron QD and 6 in 12-electron QD) and a spin down part involving the electrons with spin down.
This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown that it gives the same expectation value for the energy as the full Slater determinant.

As long as the Hamiltonian is spin independent, the above is correct. Exercise for next week: convince yourself that this is correct.

Slater determinants

We will thus factorize the full determinant $|\mathcal{D}|$ into two smaller ones, where each can be identified with $\uparrow$ and $\downarrow$ respectively:

$$|\mathcal{D}| = |\mathcal{D}|_{\uparrow} \cdot |\mathcal{D}|_{\downarrow} \tag{14}$$

The combined dimensionality of the two smaller determinants equals the dimensionality of the full determinant. Such a factorization is advantageous in that it makes it possible to perform the calculation of the ratio R and the updating of the inverse matrix separately for $|\mathcal{D}|_{\uparrow}$ and $|\mathcal{D}|_{\downarrow}$:

$$\frac{|\mathcal{D}|^{\text{new}}}{|\mathcal{D}|^{\text{old}}} = \frac{|\mathcal{D}|^{\text{new}}_{\uparrow}}{|\mathcal{D}|^{\text{old}}_{\uparrow}} \cdot \frac{|\mathcal{D}|^{\text{new}}_{\downarrow}}{|\mathcal{D}|^{\text{old}}_{\downarrow}} \tag{15}$$

Slater determinants

This reduces the calculation time by a constant factor. The maximal time reduction happens in a system of equal numbers of $\uparrow$ and $\downarrow$ particles, so that the two factorized determinants are half the size of the original one.
Consider the case of moving only one particle at a time which originally had the following time scaling for one transition:

$$\mathcal{O}_R(N) + \mathcal{O}_{\text{inverse}}(N^2) \tag{16}$$

For the factorized determinants one of the two determinants is obviously unaffected by the change so that it cancels from the ratio R.

Slater determinants

Therefore, only one determinant of size $N/2$ is involved in each calculation of R and update of the inverse matrix. The scaling of each transition then becomes:

$$\mathcal{O}_R(N/2) + \mathcal{O}_{\text{inverse}}(N^2/4) \tag{17}$$

and the time scaling when the transitions for all N particles are put together:

$$\mathcal{O}_R(N^2/2) + \mathcal{O}_{\text{inverse}}(N^3/4) \tag{18}$$

which gives the same reduction as in the case of moving all particles at once.

Updating the Slater matrix

Computing the ratios discussed above requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N \times N$ matrix by Gaussian elimination has a complexity of order of $\mathcal{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. We will use the expression

$$d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = \begin{cases} d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{new}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{old}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) & \text{if } j = i \end{cases} \tag{19}$$

Updating the Slater matrix

This equation scales as $O(N^2)$. The evaluation of the determinant of an $N \times N$ matrix by standard Gaussian elimination requires $\mathcal{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

Slater Determinant and VMC

Determining a determinant of an $N \times N$ matrix by standard Gaussian elimination is of the order of $\mathcal{O}(N^3)$ calculations. As there are $N \cdot d$ independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and $N \cdot d$ for the Laplacian (kinetic energy)

With the updating algorithm we need only to invert the Slater determinant matrix once.

This is done by calling standard LU decomposition methods.

How to compute the Slater Determinant

If you choose to implement the above recipe for the computation of the Slater determinant, you need to LU decompose the Slater matrix. This is described in chapter 4 of the lecture notes.

You need to call the function ludcmp in lib.cpp. You need to transfer the Slater matrix and its dimension. You get back an LU decomposed matrix.

LU Decomposition

The LU decomposition method means that we can rewrite this matrix as the product of two matrices **B** and **C** where

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix}.$$

The matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and **A** is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = c_{11}c_{22}\ldots c_{nn}.$$

How should we structure our code?

What do you think is reasonable to split into subtasks defined by classes?

- Single-particle wave functions?
- External potentials?
- Operations on r_{ij} and the correlation function?
- Mathematical operations like the first and second derivative of the trial wave function? How can you split the derivatives into various subtasks?
- Matrix and vector operations?

Your task is to figure out how to structure your code in order to compute the Slater determinant for the six electron dot. This should be compared with the brute force case. Do not include the correlation factor in the first attempt nor the electron-electron repulsion.

A useful piece of code, distances

```
double r_i(double**, int);
//distance between nucleus and electron i
double r_ij(double**, int, int);
//distance between electrons i and j
```

You should also make functions for the single-particle wave functions, their first and second derivatives as well.

The function to set up a determinant

```
//Determinant function
double determinant(double** A, int dim) {
  if (dim == 2)
    return A[0][0]*A[1][1] - A[0][1]*A[1][0];
  double sum = 0;
  for (int i = 0; i < dim; i++) {
    double** sub = new double*[dim-1];
    for (int j = 0; j < i; j++)
      sub[j] = &A[j][1];
    for (int j = i+1; j < dim; j++)
      sub[j-1] = &A[j][1];
    if(i % 2 == 0)
      sum += A[i][0] * determinant(sub,dim-1);
    else
      sum -= A[i][0] * determinant(sub,dim-1);

    delete[] sub;
  }
  return sum;
}
```

Set up the Slater determinant

N is the number of electrons and $N2$ is half the number of electrons.

```
//Slater-determinant
double slater(double** R, double alpha, double N,
    double N2) {
  double** DUp = (double**) matrix(N2,N2,sizeof(
      double));
  double** DDown = (double**) matrix(N2,N2,sizeof(
      double));
  for (int i = 0; i < N2; i++) {
    for (int j = 0; j < N2; j++) {
      DUp[i][j] = phi(j,R,i,alpha);
      DDown[i][j] = phi(j,R,i+N2,alpha);
    }
  }
  //Returns product of spin up and spin down dets
  double det = determinant(DUp,N2)*determinant(
      DDown,N2);
  free_matrix((void**) DUp);
  free_matrix((void**) DDown);
```

Jastrow factor

```
//Jastrow factor
double jastrow(double** R, double beta, double N,
    double N2) {
  double arg = 0;
  for (int i = 1; i < N; i++)
    for (int j = 0; j < i; j++)
      if ((i < N2 && j < N2) || (i >= N2 && j >= N2
          )) {
        double rij = r_ij(R,i,j);
        arg += 0.33333333*rij / (1+beta*rij); //
            same spin
      }
      else {
        double rij = r_ij(R,i,j);
        arg += 1.0*rij / (1+beta*rij); //opposite
            spin
      }
  return exp(arg);
}
```

```
//Check of singularity at R = 0
bool Singularity(double** R, int N) {

  for (int i = 0; i < N; i++)
    if (r_i(R,i) < 1e-10)
      return true;

  for (int i = 0; i < N - 1; i++)
    for (int j = i+1; j < N; j++)
      if (r_ij(R,i,j) < 1e-10)
        return true;
  return false;
}
```

Efficient calculations of wave function ratios

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \widehat{\mathbf{K}}_i \rangle = -\frac{1}{2} \frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle}, \tag{20}$$

$$K_i = -\frac{1}{2} \frac{\nabla_i^2 \Psi}{\Psi}. \tag{21}$$

$$\begin{aligned} \frac{\nabla^2 \Psi}{\Psi} &= \frac{\nabla^2 (\Psi_D \, \Psi_C)}{\Psi_D \, \Psi_C} = \frac{\boldsymbol{\nabla} \cdot [\boldsymbol{\nabla} (\Psi_D \, \Psi_C)]}{\Psi_D \, \Psi_C} = \frac{\boldsymbol{\nabla} \cdot [\Psi_C \boldsymbol{\nabla} \Psi_D + \Psi_D \boldsymbol{\nabla} \Psi_C]}{\Psi_D \, \Psi_C} \\ &= \frac{\boldsymbol{\nabla} \Psi_C \cdot \boldsymbol{\nabla} \Psi_D + \Psi_C \nabla^2 \Psi_D + \boldsymbol{\nabla} \Psi_D \cdot \boldsymbol{\nabla} \Psi_C + \Psi_D \nabla^2 \Psi_C}{\Psi_D \, \Psi_C} \end{aligned} \tag{22}$$

$$\frac{\nabla^2 \Psi}{\Psi} = \frac{\nabla^2 \Psi_D}{\Psi_D} + \frac{\nabla^2 \Psi_C}{\Psi_C} + 2 \frac{\boldsymbol{\nabla} \Psi_D}{\Psi_D} \cdot \frac{\boldsymbol{\nabla} \Psi_C}{\Psi_C} \tag{23}$$

Summing up: Bringing it all together, Local energy

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{a r_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Bringing it all together, Local energy

Using

$$f(r_{ij}) = \frac{ar_{ij}}{1+\beta r_{ij}},$$

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2g(r_{kj})/dr_{kj}^2$ we find that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2}+\sum_{j\neq k}\left(\frac{2a}{r_{kj}(1+\beta r_{kj})^2}-\frac{2a\beta}{(1+\beta r_{kj})^3}\right)$$

Local energy

The gradient and Laplacian can be calculated as follows:

$$\frac{\boldsymbol{\nabla}_i |\mathcal{D}(\mathbf{r})|}{|\mathcal{D}(\mathbf{r})|} = \sum_{j=1}^{N} \boldsymbol{\nabla}_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \boldsymbol{\nabla}_i \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2 |\mathcal{D}(\mathbf{r})|}{|\mathcal{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

Local energy function

```
double E_local(double** R, double alpha, double
    beta, int N, double** F, double** DinvUp,
                double** DinvDown, int N2, double**
                    detgrad, double** jastgrad) {

  //Kinetic energy
  double kinetic = 0;
  //Determinant part
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < dimension; j++) {

      if (i < N2)
        for (int l = 0; l < N2; l++)
          kinetic -= phi_deriv2(l,R,i,j,alpha)*
              DinvUp[l][i];
      else
        for (int l = 0; l < N2; l++)
          kinetic -= phi_deriv2(l,R,i,j,alpha)*
              DinvDown[l][i-N2];
    }
  }
```

Jastrow part

```
//Jastrow part
double rij,a;
for (int i = 0; i < N; i++) {
  for (int j = 0; j < dimension; j++) {
    kinetic -= jastgrad[i][j]*jastgrad[i][j];
  }
}
for (int i = 0; i < N-1; i++) {
  for (int j = i+1; j < N; j++) {
    if ((j < N2 && i < N2) || (j >= N2 && i >= N2
        ))
      a = 0.33333333;
    else
      a = 1.0;
    rij = r_ij(R,i,j);
    kinetic -= 4*a / (rij*pow(1+beta*rij,3));
  }
}
```

Local energy

```
//"Interference" part
for (int i = 0; i < N; i++) {
  for (int j = 0; j < dimension; j++) {
    kinetic -= 2*detgrad[i][j]*jastgrad[i][j];
  }
}

kinetic *= .5;
```

```
    //Potential energy
    //electron-nucleus potential
    double potential = 0;
    for (int i = 0; i < N; i++)
      potential -= Z / r_i(R,i);

    //electron-electron potential
    for (int i = 0; i < N - 1; i++)
      for (int j = i+1; j < N; j++)
        potential += 1 / r_ij(R,i,j);
    return potential + kinetic;
}
```

Determinant part in quantum force

The gradient for the determinant is

$$\frac{\boldsymbol{\nabla}_i |\mathcal{D}(\mathbf{r})|}{|\mathcal{D}(\mathbf{r})|} = \sum_{j=1}^{N} \boldsymbol{\nabla}_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \boldsymbol{\nabla}_i \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}).$$

Quantum force

```
void calcQF(double** R, double** F, double alpha,
    double beta,
            int N, double** DinvUp, double**
                DinvDown, int N2, double** detgrad,
                 double** jastgrad) {
  double sum;
  //Determinant part
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < dimension; j++) {
      sum = 0;
      if (i < N2)
        for (int l = 0; l < N2; l++)
          sum += phi_deriv(l,R,i,j,alpha)*DinvUp[l
              ][i];
      else
        for (int l = 0; l < N2; l++)
          sum += phi_deriv(l,R,i,j,alpha)*DinvDown[
              l][i-N2];
      detgrad[i][j] = sum;
    }
  }
```

Jastrow gradient in quantum force

We have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} \frac{a r_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. We get for particle k

$$\frac{\nabla_k \Psi_C}{\Psi_C} = \sum_{j\neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

Jastrow part

```
//Jastrow part
double ril,a;
for (int i = 0; i < N; i++) {
  for(int j = 0; j < dimension; j++) {
    sum = 0;
    for (int l = 0; l < N; l++) {
      if (l != i) {
        if ((l < N2 && i < N2) || (l >= N2 && i
            >= N2))
          a = 0.33333333;
        else
          a = 1.0;
```

```
                ril = r_ij(R,i,l);
                sum += (R[i][j]-R[l][j])*a / (ril*pow(1+
                    beta*ril,2));
            }
        }
        jastgrad[i][j] = sum;
    }
  }
  for (int i = 0; i < N; i++)
    for(int j = 0; j < dimension; j++)
      F[i][j] = 2*(detgrad[i][j] + jastgrad[i][j]);
}
```

Metropolis-Hastings part

```
//Initialize positions
double** R = (double**) matrix(N,dimension,sizeof
    (double));
for (int i = 0; i < N; i++)
  for (int j = 0; j < dimension; j++)
    R[i][j] = gaussian_deviate(&idum);


int N2 = N/2; //dimension of Slater matrix
```

Metropolis Hastings part

We need to compute the ratio between wave functions, in particular for the Slater determinants.

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})\, d_{ji}^{-1}(\mathbf{r}^{\text{old}})$$

What this means is that in order to get the ratio when only the ith particle has been moved, we only need to calculate the dot product of the vector $\left(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}})\right)$ of single particle wave functions evaluated at this new position with the ith column of the inverse matrix $\mathcal{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $\mathcal{O}(N)$. The only extra thing we need to do is to maintain the inverse matrix $\mathcal{D}^{-1}(\boldsymbol{x}^{\text{old}})$.

Jastrow factor in Metropolis Hastings

We have

$$\boxed{R_C = \frac{\Psi_C^{\rm new}}{\Psi_C^{\rm cur}} = \frac{e^{U_{new}}}{e^{U_{cur}}} = e^{\Delta U},} \tag{24}$$

where

$$\Delta U = \sum_{i=1}^{k-1} \left(f_{ik}^{\rm new} - f_{ik}^{\rm cur}\right) + \sum_{i=k+1}^{N} \left(f_{ki}^{\rm new} - f_{ki}^{\rm cur}\right) \tag{25}$$

One needs to develop a special algorithm that runs only through the elements of the upper triangular matrix $\boldsymbol{g}$ and have k as an index.

Metropolis-Hastings part

```
//Initialize inverse Slater matrices for spin up
    and spin down
double** DinvUp = (double**) matrix(N2,N2,sizeof(
    double));
double** DinvDown = (double**) matrix(N2,N2,
    sizeof(double));
for (int i = 0; i < N2; i++) {
  for (int j = 0; j < N2; j++) {
    DinvUp[i][j] = phi(j,R,i,alpha);
    DinvDown[i][j] = phi(j,R,i+N2,alpha);
  }
}
inverse(DinvUp,N2);
inverse(DinvDown,N2);
```

Metropolis-Hastings part

```
//Inverse Slater matrix in new position
double** DinvUp_new = (double**) matrix(N2,N2,
    sizeof(double));
double** DinvDown_new = (double**) matrix(N2,N2,
    sizeof(double));
for (int i = 0; i < N2; i++) {
  for (int j = 0; j < N2; j++) {
    DinvUp_new[i][j] = DinvUp[i][j];
    DinvDown_new[i][j] = DinvDown[i][j];
  }
}
```

Metropolis-Hastings part

```
//Gradients of determinant and and Jastrow factor
double** detgrad = (double**) matrix(N,dimension,
    sizeof(double));
double** jastgrad = (double**) matrix(N,dimension
    ,sizeof(double));
double** detgrad_new = (double**) matrix(N,
    dimension,sizeof(double));
double** jastgrad_new = (double**) matrix(N,
    dimension,sizeof(double));

//Initialize quantum force
double** F = (double**) matrix(N,dimension,sizeof
    (double));
calcQF(R,F,alpha,beta,N,DinvUp,DinvDown,N2,
    detgrad,jastgrad);
```

Metropolis-Hastings part

```
double EL; //Local energy
double sqrtdt = sqrt(delta_t);
double D = .5; //diffusion constant
//For Metropolis-Hastings algo:
double** R_new = (double**) matrix(N,dimension,
    sizeof(double));
double** F_new = (double**) matrix(N,dimension,
    sizeof(double));
double greensratio; // Ratio between Green's
    functions
double detratio; //Ratio between Slater
    determinants
double jastratio; //Ratio between Jastrow factors
double rold,rnew,a;
double alphaderiv,betaderiv;
```

Metropolis-Hastings part, inside Monte Carlo loop

```
//Ratio between Slater determinants
if (i < N2) {
  detratio = 0;
  for (int l = 0; l < N2; l++)
    detratio += phi(l,R_new,i,alpha) * DinvUp
        [l][i];
}
else {
  detratio = 0;
  for (int l = 0; l < N2; l++)
    detratio += phi(l,R_new,i,alpha) *
        DinvDown[l][i-N2];
}
```

Metropolis-Hastings part

```
//Inverse Slater matrix in new position
if (i < N2) { //Spinn up
  for (int j = 0; j < N2; j++) {
    if (j != i) {
      Sj = 0;
      for (int l = 0; l < N2; l++) {
        Sj += phi(l,R_new,i,alpha) * DinvUp[l
            ][j];
      }
      for (int l = 0; l < N2; l++)
        DinvUp_new[l][j] = DinvUp[l][j] - Sj
            * DinvUp[l][i] / detratio;
    }
  }
  for (int l = 0; l < N2; l++)
    DinvUp_new[l][i] = DinvUp[l][i] /
        detratio;
}
```

Metropolis-Hastings part

```
else { //Spinn-ned
  for (int j = 0; j < N2; j++) {
    if (j != i-N2) {
      Sj = 0;
      for (int l = 0; l < N2; l++) {
        Sj += phi(l,R_new,i,alpha) * DinvDown
            [l][j];
      }
      for (int l = 0; l < N2; l++)
        DinvDown_new[l][j] = DinvDown[l][j] -
            Sj * DinvDown[l][i-N2] /
            detratio;
    }
  }
  for (int l = 0; l < N2; l++)
    DinvDown_new[l][i-N2] = DinvDown[l][i-N2]
        / detratio;
}
```

Jastrow ratio

```
//Ratio between Jastrow factors
jastratio = 0;
for (int l = 0; l < N; l++) {
  if (l != i) {
    if ((l < N2 && i < N2) || (l >= N2 && i
        >= N2))
      a = 0.33333333;
    else
      a = 1.0;
    rold = r_ij(R,l,i);
    rnew = r_ij(R_new,l,i);
    jastratio += a * (rnew/(1+beta*rnew) -
        rold/(1+beta*rold));
  }
}
jastratio = exp(jastratio);
```

Green's functions

```
//quantum force in new position
calcQF(R_new,F_new,alpha,beta,N,DinvUp_new,
    DinvDown_new,N2,detgrad_new,jastgrad_new)
    ;

//Ratio between Green's functions
greensratio = 0;
for (int ii = 0; ii < N; ii++)
  for (int j = 0; j < 3; j++)
    greensratio += .5*(F_new[ii][j]+F[ii][j])
        * (.5*D*delta_t*(F[ii][j]-F_new[ii][
        j]) + R[ii][j] - R_new[ii][j]);
greensratio = exp(greensratio);
```

Metropolis Hastings test

```
//Metropolis-Hastings-test
if (ran2(&idum) < greensratio*detratio*
    detratio*jastratio*jastratio) {
  //Accept move abd update invers Slater
      matrix
  if (i < N2)
    for (int l = 0; l < N2; l++)
      for (int m = 0; m < N2; m++)
        DinvUp[l][m] = DinvUp_new[l][m];
  else
    for (int l = 0; l < N2; l++)
      for (int m = 0; m < N2; m++)
        DinvDown[l][m] = DinvDown_new[l][m];
```

```
        //Update position, quantum force and
            gradients
        for (int ii = 0; ii < N; ii++) {
          for (int j = 0; j < 3; j++) {
            R[ii][j] = R_new[ii][j];
            F[ii][j] = F_new[ii][j];
            detgrad[ii][j] = detgrad_new[ii][j];
            jastgrad[ii][j] = jastgrad_new[ii][j];
    .......
} //End loop of electron that has been moved
```

Proof for updating algorithm of the Slater matrix

As a starting point we may consider that each time a new position is suggested in the Metropolis algorithm, a row of the current Slater matrix experiences some kind of perturbation. Hence, the Slater matrix with its orbitals evaluated at the new position equals the old Slater matrix plus a perturbation matrix,

$$d_{jk}(\boldsymbol{x}^{\boldsymbol{new}}) = d_{jk}(\boldsymbol{x}^{\boldsymbol{old}}) + \Delta_{jk}\,, \tag{26}$$

where

$$\Delta_{jk} = \delta_{ik}[\phi_j(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_j(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{old}})] = \delta_{ik}(\Delta\phi)_j\,. \tag{27}$$

Proof for updating algorithm of the Slater matrix

Computing the inverse of the transposed matrix we arrive to

$$d_{kj}(\boldsymbol{x}^{\boldsymbol{new}})^{-1} = [d_{kj}(\boldsymbol{x}^{\boldsymbol{old}}) + \Delta_{kj}]^{-1}. \tag{28}$$

The evaluation of the right hand side (rhs) term above is carried out by applying the identity $(A+B)^{-1} = A^{-1} - (A+B)^{-1}BA^{-1}$. In compact notation it yields

$$\begin{aligned}
[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{new}})]^{-1} &= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}}) + \Delta^T]^{-1} \\
&= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}})]^{-1} - [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}}) + \Delta^T]^{-1}\Delta^T[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}})]^{-1} \\
&= [\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}})]^{-1} - \underbrace{[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{new}})]^{-1}}_{\text{By Eq.28}}\Delta^T[\boldsymbol{D}^T(\boldsymbol{x}^{\boldsymbol{old}})]^{-1}.
\end{aligned}$$

Proof for updating algorithm of the Slater matrix

Using index notation, the last result may be expanded by

$$
\begin{aligned}
d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) &= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \sum_{l}\sum_{m} d_{km}^{-1}(\boldsymbol{x}^{\boldsymbol{new}})\Delta_{ml}^{T} d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
&= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \sum_{l}\sum_{m} d_{km}^{-1}(\boldsymbol{x}^{\boldsymbol{new}})\Delta_{lm} d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{cur}}) \\
&= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \sum_{l}\sum_{m} d_{km}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) \underbrace{\delta_{im}(\Delta\phi)_l}_{\text{By Eq. } 27} d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
&= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) \sum_{l=1}^{N} (\Delta\phi)_l d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
&= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) \sum_{l=1}^{N} \underbrace{[\phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}})]}_{\text{By Eq.} 27} D_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}).
\end{aligned}
$$

Proof for updating algorithm of the Slater matrix

Using

$$\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) = \frac{adj\boldsymbol{D}}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{old}})|} \quad \text{and} \quad \boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = \frac{adj\boldsymbol{D}}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{new}})|},$$

and dividing these two equations we get

$$\frac{\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{\boldsymbol{D}^{-1}(\boldsymbol{x}^{\boldsymbol{new}})} = \frac{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{new}})|}{|\boldsymbol{D}(\boldsymbol{x}^{\boldsymbol{old}})|} = R \Rightarrow d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R}.$$

Therefore,

$$d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} [\phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) - \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}})] d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}),$$

Proof for updating algorithm of the Slater matrix

or

$$
\begin{aligned}
d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \quad & - \quad \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{new}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
& + \quad \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} \phi_l(\boldsymbol{r}_{\boldsymbol{i}}^{\boldsymbol{old}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
= d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \quad & - \quad \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{new}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) \\
& + \quad \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{old}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}).
\end{aligned}
$$

Proof for updating algorithm of the Slater matrix

In this equation, the first line becomes zero for $j = i$ and the second for $j \neq i$. Therefore, the update of the inverse for the new Slater matrix is given by

$$d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{new}}) = \begin{cases} d_{kj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) - \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{new}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\boldsymbol{x}^{\boldsymbol{old}})}{R} \sum_{l=1}^{N} d_{il}(\boldsymbol{x}^{\boldsymbol{old}}) d_{lj}^{-1}(\boldsymbol{x}^{\boldsymbol{old}}) & \text{if } j = i \end{cases}$$

Newton's method

If we consider finding the minimum of a function f using Newton's method, that is search for a zero of the gradient of a function. Near a point x_i we have to second order

$$f(\hat{\mathbf{x}}) = f(\hat{\mathbf{x}}_i) + (\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\nabla f(\hat{\mathbf{x}}_i) + \frac{1}{2}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)$$

giving

$$\nabla f(\hat{\mathbf{x}}) = \nabla f(\hat{\mathbf{x}}_i) + \hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i).$$

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point (here the exact result)

$$\hat{\mathbf{x}} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}\nabla f(\hat{\mathbf{x}}_i).$$

Subtracting this equation from that of $\hat{\mathbf{x}}_{i+1}$ we have

$$\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}(\nabla f(\hat{\mathbf{x}}_{i+1}) - \nabla f(\hat{\mathbf{x}}_i)).$$

Conjugate gradient (CG) method

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}.$$

In the iterative process we end up with a problem like

$$\hat{\mathbf{r}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}},$$

where $\hat{\mathbf{r}}$ is the so-called residual or error in the iterative process.

Conjugate gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T \hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T \hat{\mathbf{b}},$$

with the constraint that the matrix $\hat{\mathbf{A}}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{\mathbf{A}}$, which is called the Hessian, is given by the second-derivative of the variance. This quantity is always positive definite. If we vary the energy, the Hessian may not always be positive definite.

Conjugate gradient method

In the CG method we define so-called conjugate directions and two vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ are said to be conjugate if

$$\hat{\mathbf{s}}^T \hat{\mathbf{A}} \hat{\mathbf{t}} = 0.$$

The philosophy of the CG method is to perform searches in various conjugate directions of our vectors $\hat{\mathbf{x}}_i$ obeying the above criterion, namely

$$\hat{\mathbf{x}}_i^T \hat{\mathbf{A}} \hat{\mathbf{x}}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product.

Being conjugate is a symmetric relation: if $\hat{\mathbf{s}}$ is conjugate to $\hat{\mathbf{t}}$, then $\hat{\mathbf{t}}$ is conjugate to $\hat{\mathbf{s}}$.

Conjugate gradient method

An example is given by the eigenvectors of the matrix

$$\hat{\mathbf{v}}_i^T \hat{\mathbf{A}} \hat{\mathbf{v}}_j = \lambda \hat{\mathbf{v}}_i^T \hat{\mathbf{v}}_j,$$

which is zero unless $i = j$.

Conjugate gradient method

Assume now that we have a symmetric positive-definite matrix $\hat{\mathbf{A}}$ of size $n \times n$. At each iteration $i + 1$ we obtain the conjugate direction of a vector

$$\hat{\mathbf{x}}_{i+1} = \hat{\mathbf{x}}_i + \alpha_i \hat{\mathbf{p}}_i .$$

We assume that $\hat{\mathbf{p}}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{\mathbf{p}}_i$ form a basis of R^n and we can expand the solution $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ in this basis, namely

$$\hat{\mathbf{x}} = \sum_{i=1}^{n} \alpha_i \hat{\mathbf{p}}_i .$$

Conjugate gradient method

The coefficients are given by

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i = \mathbf{b}.$$

Multiplying with $\hat{\mathbf{p}}_k^T$ from the left gives

$$\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{x}} = \sum_{i=1}^{n} \alpha_i \hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{p}}_i = \hat{\mathbf{p}}_k^T \hat{\mathbf{b}},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{\mathbf{p}}_k^T \hat{\mathbf{b}}}{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{p}}_k}$$

Conjugate gradient method and iterations

If we choose the conjugate vectors $\hat{\mathbf{p}}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{\mathbf{x}}$. So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where n is so large that the direct method would take too much time.
We denote the initial guess for $\hat{\mathbf{x}}$ as $\hat{\mathbf{x}}_0$. We can assume without loss of generality that

$$\hat{\mathbf{x}}_0 = 0,$$

or consider the system

$$\hat{\mathbf{A}}\hat{\mathbf{z}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_0,$$

instead.

Conjugate gradient method

Important, one can show that the solution $\hat{\mathbf{x}}$ is also the unique minimizer of the quadratic form

$$f(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T \hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T \hat{\mathbf{x}}, \quad \hat{\mathbf{x}} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{\mathbf{p}}_1$ to be the gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_0$, which equals

$$\hat{\mathbf{A}}\hat{\mathbf{x}}_0 - \hat{\mathbf{b}},$$

and $\hat{\mathbf{x}}_0 = 0$ it is equal $-\hat{\mathbf{b}}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Conjugate gradient method

Let $\hat{\mathbf{r}}_k$ be the residual at the k-th step:

$$\hat{\mathbf{r}}_k = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k .$$

Note that $\hat{\mathbf{r}}_k$ is the negative gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_k$, so the gradient descent method would be to move in the direction $\hat{\mathbf{r}}_k$. Here, we insist that the directions $\hat{\mathbf{p}}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{\mathbf{r}}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{\mathbf{p}}_{k+1} = \hat{\mathbf{r}}_k - \frac{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}} \hat{\mathbf{r}}_k}{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}} \hat{\mathbf{p}}_k} \hat{\mathbf{p}}_k .$$

Conjugate gradient method

We can also compute the residual iteratively as

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_{k+1},$$

which equals

$$\hat{\mathbf{b}} - \hat{\mathbf{A}}(\hat{\mathbf{x}}_k + \alpha_k \hat{\mathbf{p}}_k),$$

or

$$(\hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k) - \alpha_k \hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

which gives

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{r}}_k - \hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

Codes from numerical recipes

The codes are taken from chapter 10.7 of Numerical recipes. We use the functions *dfpmin* and *lnsrch*. You can load down the package of programs from the webpage of the course, see under project 1. The package is called *NRcgm*107.*tar*.*gz* and contains the files *dfmin*.*c*, *lnsrch*.*c*, *nrutil*.*c* and *nrutil*.*h*. These codes are written in C.

```
void dfpmin(double p[], int n, double gtol, int *iter, double *fret,
double(*func)(double []), void (*dfunc)(double [], double []))
```

What you have to provide

The input to *dfpmin*

```
void dfpmin(double p[], int n, double gtol, int *iter, double *fret,
double(*func)(double []), void (*dfunc)(double [], double []))
```

is

- The starting vector *p* of length *n*
- The function *func* on which minimization is done
- The function *dfunc* where the gradient i calculated
- The convergence requirement for zeroing the gradient *gtol*.

It returns in *p* the location of the minimum, the number of iterations and the minimum value of the function under study *fret*.

Simple example and demonstration

For the harmonic oscillator in one-dimension with a trial wave function and probability

$$\psi_T(x) = e^{-\alpha^2 x^2} \qquad , P_T(x)dx = \frac{e^{-2\alpha^2 x^2}dx}{\int dx e^{-2\alpha^2 x^2}}$$

with α as the variational parameter. We have the following local energy

$$E_L[\alpha] = \alpha^2 + x^2\left(\frac{1}{2} - 2\alpha^2\right),$$

which results in the expectation value

$$\langle E_L[\alpha]\rangle = \frac{1}{2}\alpha^2 + \frac{1}{8\alpha^2}$$

Simple example and demonstration

The derivative of the energy with respect to α gives

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \alpha - \frac{1}{4\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = 1 + \frac{3}{4\alpha^4}$$

The condition

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 0,$$

gives the optimal $\alpha = 1/\sqrt{2}$.

Simple example and demonstration

In general we end up computing the expectation value of the energy in terms of some parameters $\alpha = \{\alpha_0, \alpha_1, \ldots, \alpha_n\}$) and we search for a minimum in parameter space. This leads to an energy minimization problem.
The elements of the gradient are (*Ei* is the first derivative wrt to the variational parameter α_i)

$$\bar{E}_i = \left\langle \frac{\psi_i}{\psi} E_{\mathrm{L}} + \frac{H\psi_i}{\psi} - 2\bar{E}\frac{\psi_i}{\psi} \right\rangle \tag{29}$$

$$= 2\left\langle \frac{\psi_i}{\psi}(E_{\mathrm{L}} - \bar{E}) \right\rangle \quad (\text{by Hermiticity}). \tag{30}$$

For our simple model we get the same expression for the first derivative (check it!).

Simple example and demonstration

Taking the second derivative the Hessian is

$$\begin{aligned}\bar{E}_{ij} &= 2\left[\left\langle\left(\frac{\psi_{ij}}{\psi}+\frac{\psi_i\psi_j}{\psi^2}\right)(E_{\rm L}-\bar{E})\right\rangle\right. \\ &\quad \left.-\left\langle\frac{\psi_i}{\psi}\right\rangle\bar{E}_j-\left\langle\frac{\psi_j}{\psi}\right\rangle\bar{E}_i+\left\langle\frac{\psi_i}{\psi}E_{{\rm L},j}\right\rangle\right]. \qquad (31)\end{aligned}$$

Note that our conjugate gradient approach does need the Hessian! Check again that the simple models gives the same second derivative with the above expression.

Simple example and demonstration

We can also minimize the variance. In our simple model the variance is

$$\sigma^2[\alpha] = \frac{1}{2}\alpha^4 - \frac{1}{4} + \frac{1}{32\alpha^4},$$

with first derivative

$$\frac{d\sigma^2[\alpha]}{d\alpha} = 2\alpha^3 - \frac{1}{8\alpha^5}$$

and a second derivative which is always positive

$$\frac{d^2\sigma^2[\alpha]}{d\alpha^2} = 6\alpha^2 + \frac{5}{8\alpha^6}$$

Conjugate gradient method, our case

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point (here the exact result)

$$\hat{\mathbf{x}} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1} \nabla f(\hat{\mathbf{x}}_i).$$

Subtracting this equation from that of $\hat{\mathbf{x}}_{i+1}$ we have

$$\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1} (\nabla f(\hat{\mathbf{x}}_{i+1}) - \nabla f(\hat{\mathbf{x}}_i)).$$

Simple example and demonstration

In our case f can be either the energy or the variance. If we choose the energy then we have

$$\hat{\alpha}_{i+1} - \hat{\alpha}_i = \hat{\mathbf{A}}^{-1}(\nabla E(\hat{\alpha}_{i+1}) - \nabla E(\hat{\alpha}_i)).$$

In the simple model gradient and the Hessian $\hat{\mathbf{A}}$ are

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \alpha - \frac{1}{4\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\hat{\mathbf{A}} = \frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = 1 + \frac{3}{4\alpha^4}$$

Simple example and demonstration

We get then

$$\alpha_{i+1} = \frac{4}{3}\alpha_i - \frac{\alpha_i^4}{3\alpha_{i+1}^3},$$

which can be rewritten as

$$\alpha_{i+1}^4 - \frac{4}{3}\alpha_i\alpha_{i+1}^4 + \frac{1}{3}\alpha_i^4.$$

Our code does however not need the value of the Hessian since it produces an estimate of the Hessian.

Simple example and code (model.cpp on webpage)

```
#include "nrutil.h"
using namespace std;
//     Here we define various functions called by the main program

double E_function(double *x);
void   dE_function(double *x, double *g);
void   dfpmin(double p[], int n, double gtol, int *iter, double *fret,
    double(*func)(double []), void (*dfunc)(double [], double []));
//   Main function begins here
int main()
{
     int n, iter;
     double gtol, fret;
     double alpha;
     n = 1;
     cout << "Read in guess for alpha" << endl;
     cin >> alpha;
```

Simple example and code (model.cpp on webpage)

```
//   reserve space in memory for vectors containing the variational
//   parameters
     double *p = new double [2];
     gtol = 1.0e-5;
//   now call dfmin and compute the minimum
     p[1] = alpha;
     dfpmin(p, n, gtol, &iter, &fret,&E_function,&dE_function);
     cout << "Value of energy minimum = " << fret << endl;
     cout << "Number of iterations = " << iter << endl;
     cout << "Value of alpha at minimu = " << p[1] << endl;
      delete [] p;
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the Energy function
double E_function(double x[])
{
  double value = x[1]*x[1]*0.5+1.0/(8*x[1]*x[1]);
  return value;
} // end of function to evaluate
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the derivative of the energy
void dE_function(double x[], double g[])
{
  g[1] = x[1]-1.0/(4*x[1]*x[1]*x[1]);
} // end of function to evaluate
```

Using the conjugate gradient method

- Start your program with calling the CGM method (function *dfpmin*).
- This function needs the function for the expectation value of the local energy and the derivative of the local energy. Change the functions *func* and *dfunc* in the codes below.
- Your function *func* is now the Metropolis part with a call to the local energy function. For every call to the function *func* I used 1000 Monte Carlo cycles for the trial wave function

$$\Psi_T(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}$$

- This gave me an expectation value for the energy which is returned by the function *func*.
- When I call the local energy I also compute the first derivative of the expectaction value of the local energy

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 2\left\langle \frac{\psi_i}{\psi}(E_L[\alpha] - \langle E_L[\alpha]\rangle)\right\rangle .$$

Using the conjugate gradient method

The expectation value for the local energy of the Helium atom with a simple Slater determinant is given by

$$\langle E_L \rangle = \alpha^2 - 2\alpha \left(Z - \frac{5}{16} \right)$$

You should test your numerical derivative with the derivative of the last expression, that is

$$\frac{d\langle E_L[\alpha] \rangle}{d\alpha} = 2\alpha - 2\left(Z - \frac{5}{16} \right).$$

Simple example and code (model.cpp on webpage)

```
#include "nrutil.h"
using namespace std;
//     Here we define various functions called by the main program

double E_function(double *x);
void   dE_function(double *x, double *g);
void   dfpmin(double p[], int n, double gtol, int *iter, double *fret,
    double(*func)(double []), void (*dfunc)(double [], double []));
//   Main function begins here
int main()
{
     int n, iter;
     double gtol, fret;
     double alpha;
     n = 1;
     cout << "Read in guess for alpha" << endl;
     cin >> alpha;
```

Simple example and code (model.cpp on webpage)

```
//   reserve space in memory for vectors containing the variational
//   parameters
     double *p = new double [2];
     gtol = 1.0e-5;
//   now call dfmin and compute the minimum
     p[1] = alpha;
     dfpmin(p, n, gtol, &iter, &fret,&E_function,&dE_function);
     cout << "Value of energy minimum = " << fret << endl;
     cout << "Number of iterations = " << iter << endl;
     cout << "Value of alpha at minimu = " << p[1] << endl;
      delete [] p;
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the Energy function
double E_function(double x[])
{

//  Change here by calling your Metropolis function which
//  returns the local energy

  double value = x[1]*x[1]*0.5+1.0/(8*x[1]*x[1]);



  return value;
} // end of function to evaluate
```

You need to change this function so that you call the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$.

Simple example and code (model.cpp on webpage)

```
//  this function defines the derivative of the energy
void dE_function(double x[], double g[])
{

//  Change here by calling your Metropolis function.
//  I compute both the local energy and its derivative for every call

  g[1] = x[1]-1.0/(4*x[1]*x[1]*x[1]);
} // end of function to evaluate
```

You need to change this function so that you call the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$. When I compute the local energy I also compute its derivative. After roughly 10-20 iterations I got a converged result in terms of α.