Computational Physics III: Report 3: Iterative methods for solving System of Linear Equations

Due on June 6, 2019

Nicolas Lesimple

Contents

1. Steepest descent and conjugate gradient methods

The conjugate gradient descent is an optimization algorithm, more precisely a **first-order iterative optimization algorithm**, used to minimize some function by iteratively **moving in the direction of descent as defined by the negative of the gradient.** In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

Gradient descent method is a way to find a local minimum of a function. The way it works is that we start with an **initial guess of the solution** and we take the gradient of the function at that point. We step the solution in the negative direction of the gradient and we repeat the process. The algorithm **will eventually converge where the gradient is zero (which correspond to a local minimum)**. Its brother, the gradient ascent, finds the local maximum nearer the current solution by stepping it towards the positive direction of the gradient. They are both first-order algorithms because **they take only the first derivative of the function.**

The main **difference between the Steepest Gradient descent and the gradient descent is that in conjugate gradient descent, we compute the update for the parameter vector where the following equation $\Theta = \Theta - \eta\Delta\Theta * f(\Theta)$t is typically defined as gradient descent in which the learning rate η is chosen such that it yields maximal gain along the negative gradient direction.** The part of the algorithm that is concerned with determining η each step is called **line search**. To sum up, the gradient descent only cares about descent in the negative gradient direction while the steepest method descents cares about the direction of the largest directional derivative.

In this exercise, **we will implement both method and use it to solve a puzzle problem.** This will illustrate the usefulness of these two methods. In fact, we consider a puzzle that can be solved with **the help of systems of linear equations**. A square board (represented by a matrix which contains non-negative integers) of size N*N contains b_{ij} coins in each cell (i, j), b_{ij} being a non-negative integer. At each step you are allowed to select a particular cell and to take exactly one coin from it and all its edge-sharing neighbors. Your aim is to take all coins from the board meaning that the aim is to set all the values at 0 of the matrix. The order in which this procedure is performed is not important. Below is an example 2-step solution of simple puzzle on a 3*3 board. **The solution can be obtained by solving a system of linear equations. That's why gradient approach method become interesting in this puzzle problem.**

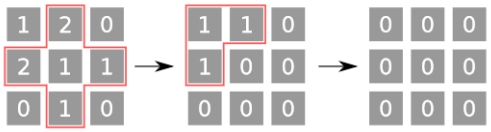

Figure 1: Initial problem and steps to the expected result.

For this particular example, the solution may be represented by the following matrix in Figure 2:

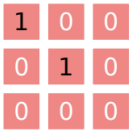

Figure 2: Solution of the Initial problem :

1.

With the following definition, we are able to model the solution by solving a system of linear equations $Ax = b$. The initial number b_{ij} contained in each cell is equal to the number of steps x performed in adjacent cells. It is possible to merge the indices i and j with k :

$$k = (i-1)N + j$$

$$b_{i,j} = x_{i,j} + x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1} \Rightarrow b_k = x_k + x_{k-N} + x_{k+N} + x_{k-1} + x_{k+1} \tag{1}$$

The matrix A can be obtained using a compact notation involving Kronecker products. Note that matrix A has size of $N^2 * N^2$ and depends only on N :

$$A = T \otimes \mathbb{1} + \mathbb{1} \otimes T - \mathbb{1} \otimes \mathbb{1} \tag{2}$$

Thus, we implement a function *puzzleA()* generating A for a given problem size N using kron. Then, with this module, we write a code that solves the above 3*3 example. To do that, we used the matlab build in function to solve linear equations. The code allowing this process is shown in Listing 1.

Listing 1: Code allowing the solving of the puzzle with a linear system :

```
% Question 1 :
N = 3;
A=puzzleA(N);
b=[1;2;0;2;1;1;0;1;0];
x=(A*A)\(A*b);
solution = zeros(N,N);
for i = 1:N
    solution(i,:) = x((i-1)*N+1:i*N);
end
fprintf('The solution of the puzzle is :\n')
disp(round(solution))

function A=puzzleA(N)
    % This function allows the creation of matrix A defined in the
    % question of the problem.
    % ARGS :
    %   - N : Interger corresponding to the size of the problem
    % RETURN :
    %   - A : Matrix A used to define the problem as linear equation
    T=full(gallery('tridiag',N,1,1,1));
    A=kron(T,eye(N))+kron(eye(N),T)-kron(eye(N),eye(N));
end
```

The solution obtained is the one expected described in the question :

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

2.

The steepest descent method and the conjugate gradient method can also be used to solve this linear system as said in the introduction of this exercise. We thus implement both the steepest descent method (*solveSD.m*) and the conjugate gradient method (*solveCG.m*). Both algorithm can be seen in Listing 2 and 3.

Listing 2: Steepest Descent Method Implementation:

```
function [x, niter] = solveSD(A, b, maxiter)
     % This function solve A*x = b using steepest descent algorithm.
     % It returns the solution and the convergence information.
     % ARGS :
     %  - A : Positive definite NxN matrix
     %  - b : Right-hand side Nx1 column vector
     % RETURN :
     %  - x :  Nx1 solution vector
     %  - conv : Table showing the value of the residual at each step

     % Check if maxiter argument was given. If not, assign default value.
     if nargin < 3
       maxiter = 100000000;
    end

    % Initialisation of the parameters of the SD method
    N = size(A);
    x = eye(N(1),1);
    niter = 1;

    % First initialisation of the varialbles (first iteration)
    r = b - A*x;
    delta = norm(r);%r'*r;
    conv = delta;

    % We continue to iterate until the convergence is reach
    while (delta > 1e-11)&&(maxiter > niter)
        % Here we just apply the well know algorithm
        q = A*r;
        alpha = (r'*r)/(r'*q);
        x = x + alpha*r;
        r = b - A*x;
        delta = norm(r);
        conv = [conv, delta];
        niter = niter + 1;
    end
end
```

Listing 3: Conjugate Gradient Method Implementation:

```
function [x, niter] = solveCG(A, b, maxiter)
    % This function solve A*x = b using Conjugate Gradients method.
    % It returns the solution and the convergence information.
    % ARGS :
    %  - A : Positive definite NxN matrix
    %  - b : Right-hand side Nx1 column vector
    % RETURN :
    %  - x :  Nx1 solution vector
    %  - niter : Number of iterations performed

    % Initialisation of the parameters of the SD method
    tol = 1e-11;
    N = size(A);
    s = eye(N(1),1);

```

```
    % Check if maxiter argument was given. If not, assign default value.
    if nargin < 3
        maxiter = 100000000;
    end

    % First iteration
    x = s;
    r = b - A*s;
    d = r;
    rho = r'*r;
    niter = 0;      % Initialize counter

    % We continue to iterate until the convergence is reach
    while (norm(r) > tol)&&(maxiter > niter)
        % Here we just apply the well know algorithm
        a = A*d;
        alpha = rho/(a'*d); % Calculate the step lenght
        x = x + alpha*d;
        r = r - alpha*a;
        rho_new = r'*r;
        d = r + rho_new/rho * d;
        rho = rho_new;
        niter = niter + 1;
    end
end
```

Tests were conducted on both algorithm (*test_solve.m*) to check the well behavior of both methods. All tests were passed thus our two implemented algorithm are reliable. Thus, the steepest descent method and the conjugate gradient method were successfully implemented and are able to provide relative error of less than 10^{-10}.

3.

We will now apply our two methods on two 10*10 board puzzles that can be found in file boards.mat available on Moodle. This puzzles were given in order to test the two previously implemented algorithms. **Instead of running x = solve(A,b) we decided to use x =** $solve(A^TA, Ab)$**. In fact, the purpose of this replacement is to be sure that we are working on a symmetric positive definite matrix.** In fact, our algorithm works only with that kind of matrix given as input. In this problem, we never checked if the input matrix was completing these characteristics. By doing $A^T * A$ instead of A we ensure that the new matrix of study and thus the problem is symmetric and positive definite. In fact, if A is square and non singular, the solution to $A^T * Ax = A^T * b$ is the solution to Ax = b. If A is not square and Ax = b is over constrained, then it is possible to find solution x that minimizes a least squares problem.

For each of the two puzzles, we found the solution and we plot it in a similar way it was done on this exercise sheet. These results can be seen on Figure 4.

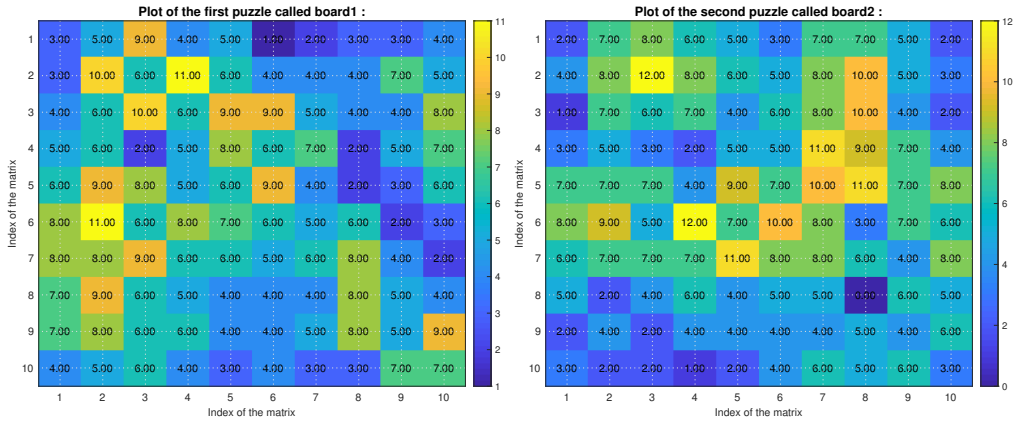

Figure 3: Illustration of the two different board problems we need to solve :

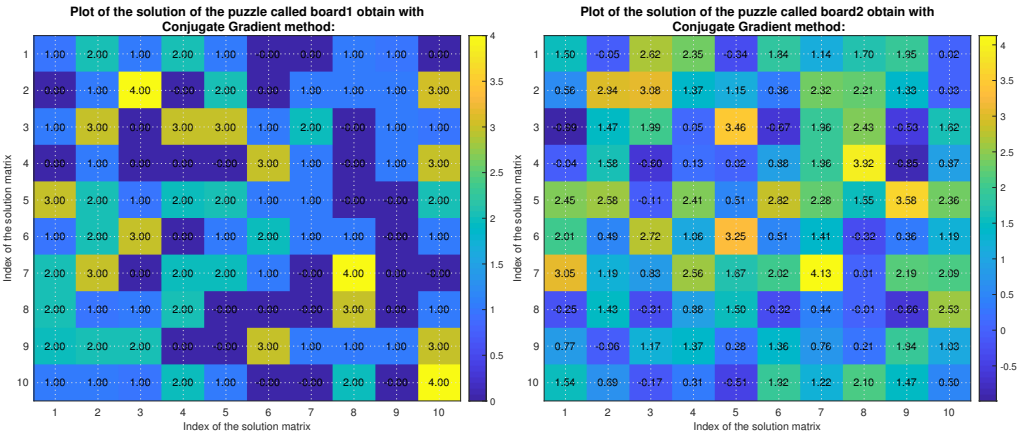

Figure 4: Solution of the two puzzles obtain by using the Conjugate Gradient Descent and the Steepest gradient Descent method. As explained in the report, the coefficient of this matrix are indeed solutions of the system of linear equations (2) but do not correspond to any sensible/reasonable answer for the considered problem.

However, in the results we can notice that the solution to the puzzle corresponding to the board2 has several negative values. These negative values make no sense when you think about the rules of the game. Moreover, **while the first solution is perfectly reasonable, the second one does not make sense for another reason. Indeed, the solution can not have non-integers/negative values because each value describes the number of times a given cell has been selected.** It means that the resolution of the system of linear equations of the second puzzle find an answer to the problem. However, this solution does not correspond to any legitimate solution. This is due to the fact that we did not reduce the set of possible answer our method can give. In fact, we should have add some constraints to force all the coefficient of the solution matrix to be positive. This would become an constraint optimization problem and linear programming could be used to solve it. Some well known techniques could be used using the Lagragian with the dual problem. In opposition, the solution of the first problem make sense and seems to be right or at least, don't seems to be wrong.

The error between the two algorithms with the solution found with Matlab build in function was also computed : error $= ||x_{Matlab} - x_{method}||$. For both boards, it was smaller than 10^{-8}, if the tolerance of the algorithms was set to 10^{-12}.

4.

To analyze and understand the behavior of our two algorithms, we plot the estimated error as a function iteration count for steepest descent and conjugate gradient methods solving the first and second problem of the *boards.mat* examples. These results are displayed in Figure 5.

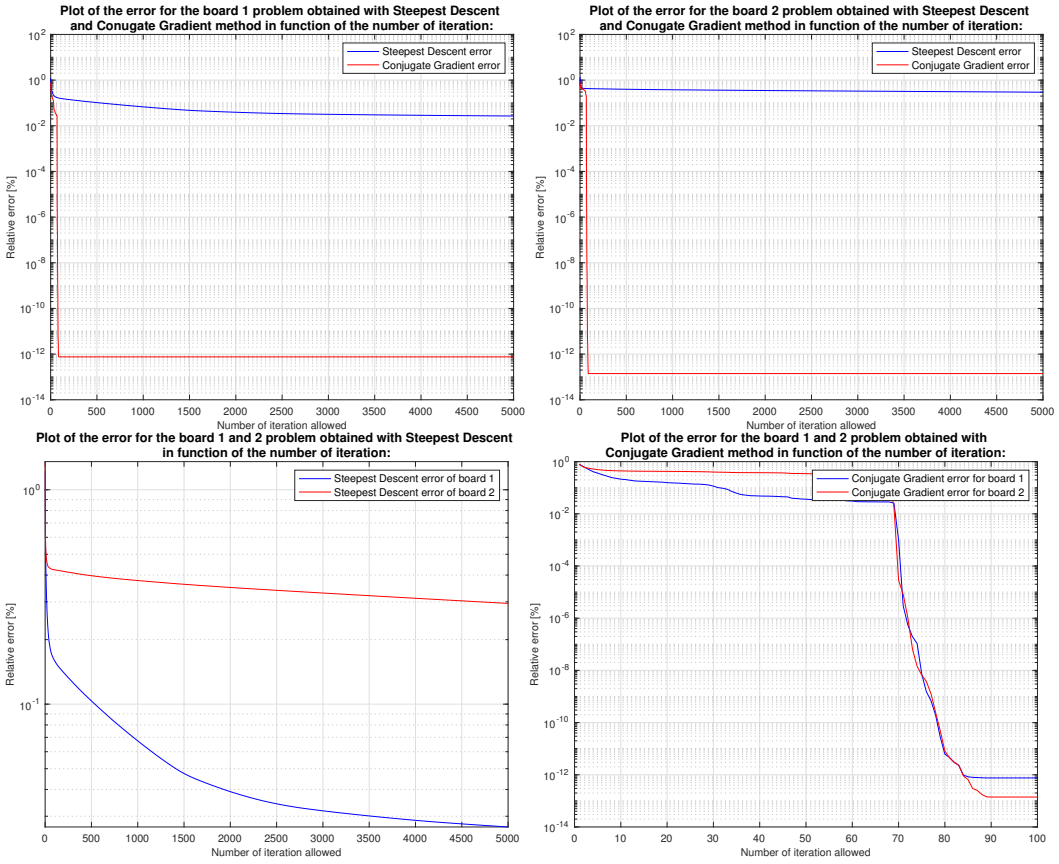

Figure 5: Estimated error as a function iteration count for steepest descent and conjugate gradient methods solving first of the boards.mat example using a semilogy plot.

As you can see on theses plots, the convergence is achieved for Conjugate gradient algorithm for both problem around iteration 80. On the contrary the convergence is never really reach with the Steepest Descent method for problem 2 if we considered that convergence is achieved when error is less than 10^{-2}. This Steepest Descent convergence was tested for only 5000 iterations. As a really small decreasing trend is observable, we deduce that it should be possible to achieve convergence with a extremely high number of iterations. In addition, with the same convergence criteria, we can conclude that the steepest gradient descent algorithm converged after 5000 iterations for problem 1. However the conclusion stay the same : **It is clear that the conjugate gradient algorithm converge faster than the steepest descent algorithm meaning that the conjugate gradient descent is much more efficient.**

The reason for that is mainly the difference between the two methods explained above about the direction they took to update the step. **In fact, the gradient descent only cares about descent in the negative gradient direction while the steepest method descents cares about the direction of the largest directional derivative.** This is visible with the norm we take to optimize the step. In fact the convergence of the gradient method should take less than N^2 iterations where N is the number of elements in the matrix.

2. N charged particles

In this problem, we consider **N identical charged particles in a 2D harmonic potential. The energy of such a system can be written as a sum of harmonic potential term and repulsive Coulomb interaction between particles** which is :

$$E_{\text{tot}} = E_{\text{harm}} + E_{\text{Coul}}$$
$$E_{\text{harm}} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N) = k \sum_i \mathbf{x}_i^2$$
$$E_{\text{Coul}}(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N) = \sum_{i \neq j} U(|\mathbf{x}_i - \mathbf{x}_j|)$$

Figure 6: Energy of the system

where $x_1, x_2, ..., x_N$ are the coordinates of particles and U(x) is the Coulomb repulsion : $U(x) = \frac{\epsilon\sigma}{x}$. In this exercise, we will consider that $\epsilon = \sigma = k = 1$. The **minima of potential energy** $E(x_1, x_2, ..., x_N)$ (E (x_i) for simplicity) correspond to **equilibrium configurations of the ensemble of particles.**

In this problem, **we will implement the nonlinear conjugate gradient method to find the minima of potential energy surface for an ensemble of N charged particles.** This will allow us to **plot the energy evolution in function of N the number of particles in the system and to find the position of each particle inside the system.**

The nonlinear conjugate gradient method generalizes the conjugate gradient method to nonlinear optimization. Whereas linear conjugate gradient seeks a solution to the linear equation $A^T Ax = A^T b$, the nonlinear conjugate gradient method is generally used to find the **local minimum of a nonlinear function using its gradient** $\nabla_x f$ alone. It works when the **function is approximately quadratic near the minimum, which is the case when the function is twice differentiable at the minimum and the second derivative is non-singular there.** In fact, the basis for a nonlinear conjugate gradient method is to effectively apply the linear conjugate gradient method, where the residual is replaced by the gradient. A model quadratic function is never explicitly formed, so it is always combined with **a line search method**.

1.

In this first question, we need to implement the nonlinear conjugate gradient method for finding the minima of potential energy surface for an ensemble of N charged particles. To do that, **we first need to implement the total energy calculation given a position vector as input.** In Listing 4, you can find our implementation of this calculus taking advantage of the definition of the total energy (which is the sum of the Harmonic and Coulomb energy).

Listing 4: A script which performs the calculus of the energy of the system by calculating and summing the Harmonic and Coulomb corresponding energies:

```
%%%%%% Total Energy Calculus %%%%%%
function e = energy_total(x)
    % This function allows the calculation of the total energy of the
    % system given a position vector.
    % ARGS :
    %   - x : Position vector
    % RETURN :
    %   - e : Float representing the total energy of the system
    % Harmonic Energy Calculus
    E_harm = 0;
    for i=1:length(x)
        E_harm=E_harm+x(i)*x(i);
```

```
    end
    % Coulomb Energy Calculus
    E_coul = 0;
    for i=1:2:length(x)
        for j=1:2:length(x)
            if i~=j
                E_coul=E_coul+1/(norm([x(i:i+1)-x(j:j+1)]));
            end
        end
    end
    % Total Energy Calculus
    e = E_harm + E_coul;
end
```

We used the numerical gradient of $E(x_i)$ for evaluating residuals. To find the minimum of a function in given direction, we used the Newton-Raphson line search method employing the Taylor expansion up to the second order :

$$E(x_i + \alpha \mathbf{d}) = E(x_i) + \alpha * [E'(x_i)]^T * \mathbf{d} + \frac{\alpha^2}{2} * \mathbf{d}^T * (E''(x_i)) * \mathbf{d}$$

Both the vector of gradient $E'(x_i)$ and directional second derivative $d^T E''(x_i)$ have been evaluated numerically. The implementation of the gradient and hessian function using the Newton-Raphson line search method employing the Taylor expansion is shown just below in the Listing 5.

Listing 5: A script which performs the calculus of the gradient and hessian with the Newton-Raphson line search method employing the Taylor expansion :

```
%%%%%%% Gradient Calculus %%%%%%%
function grad_value=grad(x)
    % This function allows the calculus of the Gradient of the function
    % using line search and Taylor expansion.
    % ARGS :
    %   - x : Position vector
    % RETURN :
    %   - grad_value : Float corresponding to the value of the gradient
    %   corresponding to the position vector given as input.
    grad_value=zeros(size(x));
    h=1e-7;
    for i=1:length(x)
        ei=zeros(size(x));
        ei(i)=1;
        grad_value(i)=energy_total(x+ei*h)-energy_total(x);
    end
    grad_value=grad_value/(h);
end



%%%%%%% Hessian Calculus %%%%%%%
function H=hessian(x,d)
    % This function allows the calculus of the Hessian using line search
    % and Taylor expansion.
    % ARGS :
    %   - x : Position vector
    %   - d : Number representing to point on which we want to calculate
    %   the Hessian.
    % RETURN :
```

```
    %   - H : Float corresponding to the value of the hessian.
    d=d/norm(d);
    h=1e-4;
    H=(energy_total(x+d*h)-2*energy_total(x)+energy_total(x-d*h))/h^2;
end
```

Then, thanks to these different implementations, we were able to implement the non-linear conjugate gradient descent. The code allowing this process is shown below in Listing 6. We apply the algorithm saw in class which is the one display in Figure 7. In addition, we added an optimizing step: when we found a local minimum in the energy landscape, we decide to update the position vector in the direction of the gradient corresponding to this local minima. Moreover, we calculate and save the energy level corresponding to the minimal energy configuration given by the vector position for each system of N = 1, ..., 12 particles.

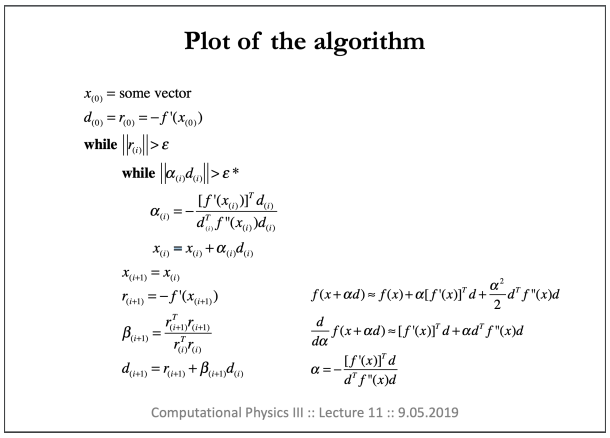

Figure 7: Non Linear Conjugate Gradient Algorithm

Listing 6: A script which performs the nonlinear conjugate gradient descent allowing the finding of the minimal energy configuration of the N particles composing the system :

```
%%%%% Non Linear conjugate gradient method %%%%%

% Declaration of the parameters of our algorithm
Number_particles_max=12;
E=zeros(Number_particles_max,1);
x=cell(Number_particles_max,1);
tol = 1e-4;
tol_inside = 1e-10;

% Non linear method to find the configuration to have a min energy
for N=2:Number_particles_max

    % First declaration per particles of the several parameters of the
    % simulation
    x{N}=rand(2*N,1); % Random initialisation for the position vector
    r=-grad(x{N}); d=r; alpha=1;
    b=0;

    while norm(r)>tol
        while norm(alpha*d)>tol_inside
            hess=hessian(x{N},d); % Hessian calculation
            if hess==0
```

```
                x{N}=x{N}-1e-4*grad(x{N})/norm(grad(x{N})); % Special case
                % if the hessian is 0
            end
            % Update of the variable as we are computing an iterative method
            alpha=(-grad(x{N})'*d)/(d'*hess*d);
            x_after=x{N}+alpha*d;
            if energy_total(x_after)>energy_total(x{N}) % If with find a local min
            % , we will search in this direction
                % This conditions allows a faster computation as it
                % optimize the search direction
                x{N}=x{N}-1e-4*grad(x{N})/norm(grad(x{N}));
                b=1;
                break
            else
                x{N}=x_after; % Classic update of x_i
            end
        end

    % Update of the variable as we are computing an iterative method
    rtmp=-grad(x{N});
    beta=rtmp'*rtmp/(r'*r);
    r=rtmp;
    d=r+beta*d;
    end

    % Calculus and storage of the min energy of the system corresponding to
    % the configuration we found with our algorithm
    E(N)=energy_total(x{N});

    if N == Number_particles_max
        fprintf('The energy for each configuration (N=1...N_max) is display below :\n')
        disp(E)
    end
end
```

2.

We want to visualize the results of our algorithm and thus being able to understand and analyze if our results make sense. To do that, **we needed to experiment with the process to find the values of the parameters allowing us to optimize the coordinates efficiently enough.** In fact, we observe that **depending on our convergence criteria and initial configuration both unstable and non-converged configuration can be found.** We thus succeeded to find appropriate parameters. Specifically, we took times to tune the two different tolerance used in our implementation. Thanks to that, we were able to obtain the following results for N = 2,...,9 displayed in Figure 8 and Figure 9.

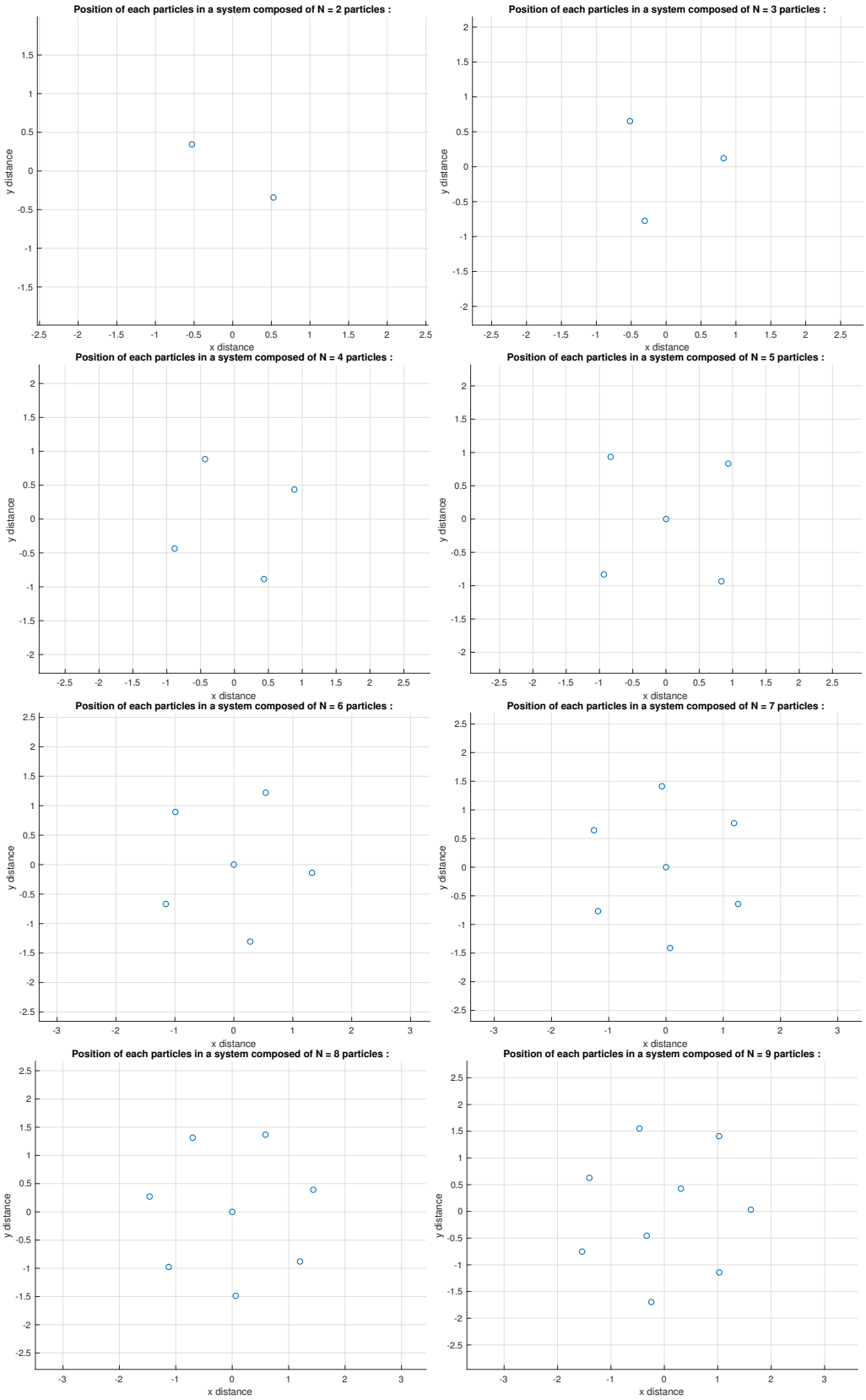

Figure 8: Position of the particles composing a system of N (N = 2, ..., 9) particles after the minimization of the total energy contained in each systems.

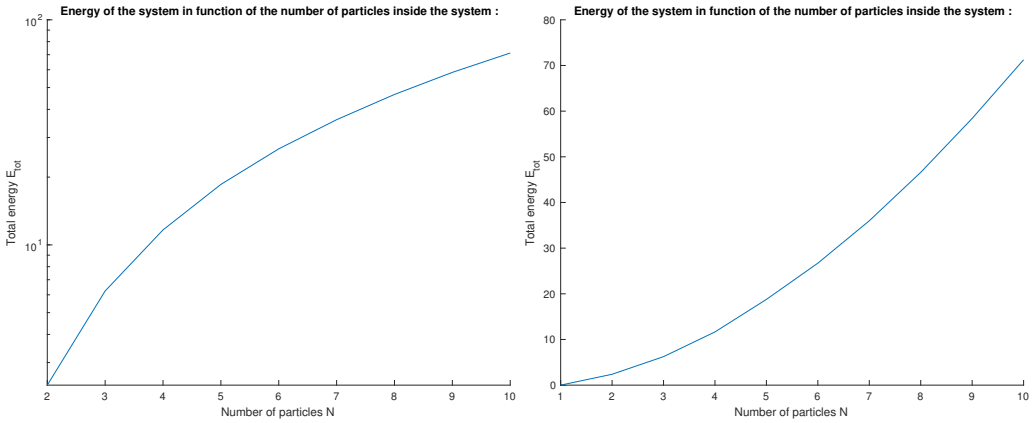

Figure 9: Energy of the system in function of the number of particles inside the system. The left one is in log scale, the right one is in classic scale.

We were thus able to visualize the most stable atomic configuration for each system composed of a different number of particles thanks to our non-linear conjugate gradient implementation. In fact, the Energy corresponding to each system is summing up in the following table :

Number of particles in the system	E_{min}
1	0
2	2.3811
3	6.2403
4	11.6544
5	18.5603
6	26.7117
7	35.9909
8	46.5922
9	58.4053
10	71.1940

Figure 10: Lowest energy configurations for larger N = 1, ..., 10.

3.

The lowest energy configurations for N = 5, ..., 10 are sum up in the previous table in Figure 10. In addition in Figure 9, the evolution of the ground states can be visualized in the function of N. **As in the previous task, we had to perform multiple runs starting from different initial configurations to be sure that the minimum in the energy landscape was found. We thus, as before for smaller N, be able to each time find the ground-state configuration with minimal energy E_{min}.** For these larger systems, the optimal geometry (which means the physical position of each particle into the system) was also displayed in Figure 8.

To conclude, thanks to the non-linear conjugate gradient method, we were able to find the lowest energy configurations which are the most stable atomic configuration of our system. For each system, we were able to visualize the optimal geometry and the energy value of each ground states configurations. Our results make sense as for N = 2, the geometry is a line, for N = 3 a triangle, for N = 4 a square, for N = 5 a cross (square with one particle in the center), for N = 6 a pentagon with one in the middle and for 7 and more, we always have a kind hexagon structure with some particles in the center.

All these geometric shapes are known to be the usual structure of low energy states as by definition these shapes are minimizing the total energy in the system.

3. Over-defined system of linear equations

In mathematics, a system of equations is considered **overdetermined if there are more equations than unknowns. An overdetermined system is almost always inconsistent (it has no solution) when constructed with random coefficients.** However, an overdetermined system will have solutions in some cases, for example, if some equation occurs several times in the system, or if some equations are linear combinations of the others.

The terminology can be described in terms of the concept of constraint counting. Each unknown can be seen as an available degree of freedom. Each equation introduced into the system can be viewed as a constraint that restricts one degree of freedom. Therefore, the critical case occurs when the number of equations and the number of free variables are equal. For every variable giving a degree of freedom, there exists a corresponding constraint. The overdetermined case occurs when the system has been overconstrained that is when the equations outnumber the unknowns.

In this problem we consider the following over-defined system of linear equations :

$$\begin{vmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{vmatrix} * \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} 1 \\ 1.2 \\ 1.1 \end{vmatrix}$$

1.

We want to minimize the norm of residual $r(x) = ||A * x–b||$ **by computing the Penrose inverse of the matrix A. To do that we use Matlab command *pinv*** : pinv is the pseudoinverse command that produces a matrix X of the same dimensions as A' so that A*X*A = A, X*A*X = X, and A*X and X*A are Hermitian. In fact, the Penrose inverse A^+ of the matrix A is defined as follows:

$$A^+ = (A^* A)^{-1} A^*$$

The computation is based on SVD(A) and any singular values less than a tolerance are treated as zero. **The minimization of the residual was achieved with the following command** $x = pinv(A) * b$**.** With that the solution of the system that minimizes this norm was found :

$$x_{sol} = \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} 1 \\ 0.05 \end{vmatrix}$$

2.

To illustrate that **our previous solution indeed minimizes the norm of residual r(x), we plot its magnitude in the vicinity of the solution x.** For this purpose, we use the Matlab command *contour()*. The resulting plot is observable on Figure 11 just below.

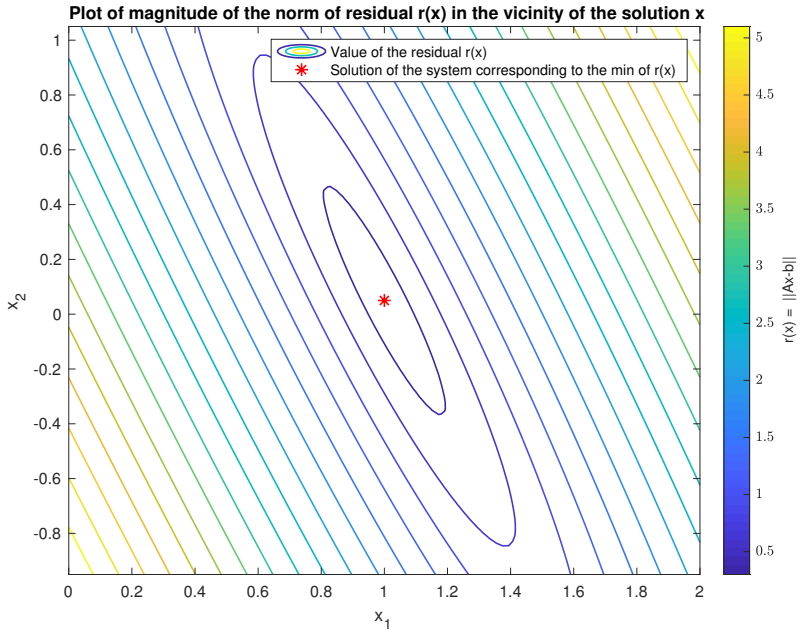

Figure 11: Contour plot of the magnitude of the norm of the residual in the vicinity of the solution x.

As you can see, **the found minimum corresponds to the minimum of the magnitude of the norm. In fact, we are able to see with the color bar that the smallest values of the norm of the residual are located in the vicinity of the founded minimum.** Thus, this result illustrates well that our previous solution indeed minimizes the norm of residual r(x).

4. Image compression using SVD

In linear algebra, **the singular-value decomposition (SVD) is a factorization of a real or complex matrix.** It is the generalization of the eigen-decomposition of a positive semi-definite normal matrix to any $m \times n$ matrix via an extension of the polar decomposition. It has many useful applications in signal processing and statistics.

Formally, the singular-value decomposition of an $m \times n$ real or complex matrix $\mathbf{M}$ is a factorization of the form $\mathbf{U\Sigma V}^*$, where $\mathbf{U}$ is an $m \times m$ real or complex unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and $\mathbf{V}$ is an $n \times n$ real or complex unitary matrix. The diagonal entries Σ_i of Σ are known as the singular values of $\mathbf{M}$. The columns of $\mathbf{U}$ and the columns of $\mathbf{V}$ are called the left-singular vectors and right-singular vectors of $\mathbf{M}$, respectively.

Applications that employ the SVD include computing the pseudoinverse, least squares fitting of data, multivariable control, matrix approximation, and determining the rank, range and null space of a matrix.

In this exercise, **we will focus on the usage of the singular value decomposition for image compression.** The idea is to perform the singular value decomposition of the matrix **representing the image and store only a subset of selected significant singular values and corresponding vectors. Then, the approximate image can then be easily reconstructed using this reduced amount of data.**

1.

The first step that needed to be done was the downloading of the photograph of Emmy Noether available on Moodle and the loading of it into Matlab using *imshow()*. After having used the function *double()* to translate the image into double, Matlab singular value decomposition transformation was performed on the image. The resulting singular values are ploted on a logarithmic scale in Figure 12.

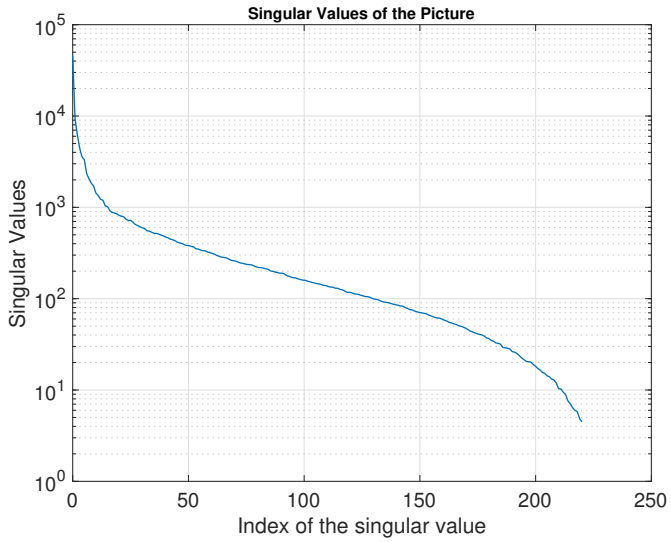

Figure 12: Singular values of of picture of interest with the Y axis as logarithmic scale.

In addition to this information, **the rank of the matrix has been found. In linear algebra, the rank of a matrix A is the dimension of the vector space generated (or spanned) by its columns.** This corresponds to the maximal number of linearly independent columns of A. This, in turn, is identical to the dimension of the space spanned by its rows. Rank is thus a measure of the "nondegenerateness" of the system of linear equations and linear transformation encoded by A. **In our case the rank of the image is 220, which correspond to the number of columns (the width) of the image. This means that the set of all these vectors is full-rank.**

2.

We now want to compress our image using SVD algorithm. To do that, we first approximate the original image by a lower-rank matrix of rank N. On Figure 13, the original image, as well as the approximated images for the number of singular values N = 4, 20, 100, are displayed.

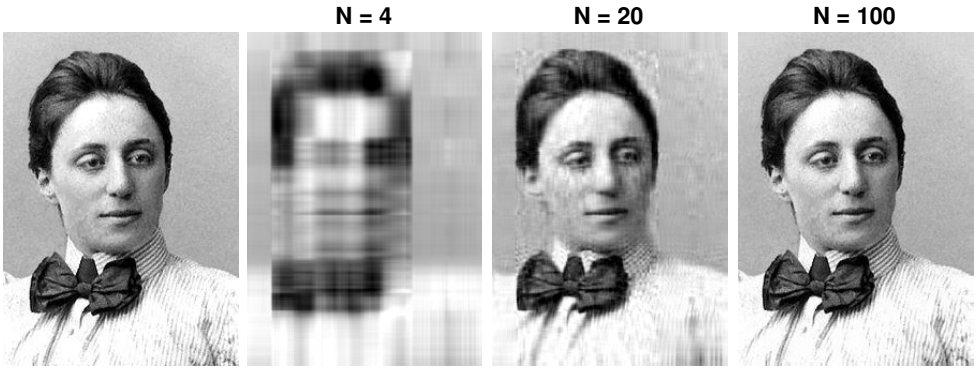

Figure 13: Original image on the left as well as the approximated images for the number of singular values N = 4, 20, 100

These pictures are a perfect illustration of the power of the Singular Value Decomposition for image compression. In fact, here we can see that with 4 singular values, the shape of the original picture is clearly visible and we can easily understand that this picture is a portrait. However, the picture is really blurred. The consequence is that we are not able to distinguish if the person is a man or a woman so we have lost some precious information.
Then, for the picture displaying 20 singular values, we are able to observe the person as in the original image. We have all the information present as in the original image. However, the quality of the image is low. In fact, it's kind of little bit blurred everywhere and we can distinguish pixels. The effect is not wanted and thus the reconstruction of the image is not complete. **However, this image only contains less than a $\frac{1}{10}$ of the initial number of singular values and for some usages can be considered as acceptable quality.**
For N = 100, when you look at both pictures with your own eyes, it seems that the original image and the one with 100 singular values are identical. Here the reconstruction seems perfect. We can thus conclude that the entire information of this image is stored in 100 singular values meaning that we just have to store these 100 vectors instead of the 220 one composing the picture. **This means that it is possible to approximate the initial image by reducing the amount of needed memory while keeping a sufficiently good quality**.

That's why the SVD algorithm can be used as image compression. **The main information is encrypted in a smaller amount of data. The initial image can be relatively well approximated with a reduced amount of vectors. Concretely it is possible to remove some of the vectors to set some singular values to 0 and still have a sufficiently good quality of the picture. Thus, the memory used for this image becomes more than twice time smaller after the usage of this SVD algorithm compared to the original one.** When you look at Figure 13 found in the previous question, all these facts makes sense. Indeed we can see that choosing the 4 first singular values we selected the biggest one but a part of important information is forgotten. Then, when selecting 20 of them, we take into account a large part of the upper part of the graph, meaning the part that contains the main part of information. The same applies to the selection of the 100 singular values. **Thus, the previous graph allows us to observe these different levels of image information selection.**

To really see this process, we plot **the error in term of loss of image information in function of the number of non zero singular values** in Figure 14. The conclusion stays the same: **the SVD**

algorithm is able to encode the main information of the picture with less memory. It's image compression. However, there exists a trade-off between image quality and quantity of space obtain with the compression. By visual inspection, it seems that 100 singular values is a good choice but the user need to choose is own criteria to choose how to compress the picture. As expected in Figure 14, the error is decreasing while the number of singular values keeps increase.

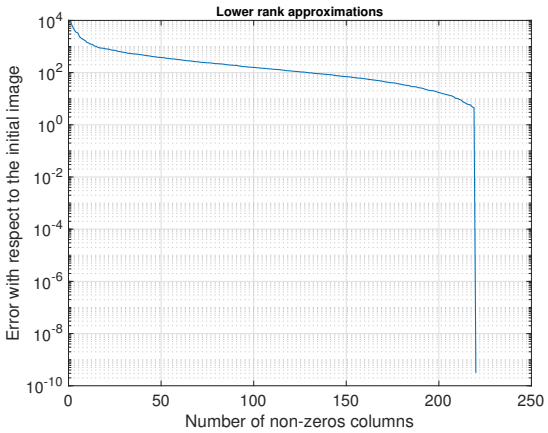

Figure 14: Error in term of lost of image information in function of the number of non zero singular values

In the following table in Figure 15, **the memory required to store the image on our computer after saving it in *.eps* format directly from our Matlab code is given**. This memory is dependent on the function of the number of non-zero singular values. Here we are able to see that the needed memory is not linearly correlated with the number if singular values kept. In fact, in this case, N = 20 should be 5 times the one corresponding to N = 4. However, the number of memory units required to store the image should be a linear function of the number of non-zero singular values. By doing the assumption that each non-zero value of a matrix corresponds to 1 memory unit, and by looking at the way the matrix multiplication USV' is computed, we can find a theoretical amount of storage needed for each picture : in that case the one with N = 20 should take 5 times more memory than the one corresponding to 4 and so on. In our case, the algorithm used to save it or the *.eps* format could have impacted the overall storage required. **Even though this relation is not linear (or even exponential), these values illustrate the power and the high efficiency of the singular value decomposition to try to minimize the required memory to save the image.**

N	Memory
4	186 Ko
20	283 Ko
100	385 Ko
220	485 Ko

Figure 15: Amount of memory needed to store image containing the N singular vector of the picture.

5. Double occupancy in the mean-field Hubbard model

On one hand, **the Hubbard Hamiltonian (HH) offers one of the most simple ways to get insight into how the interactions between electrons give rise to insulating, magnetic, and even novel superconducting effects in a solid.** It was written down in the early 1960s and initially applied to the behavior of the transition-metal monoxides (FeO, NiO, CoO), compounds which are antiferromagnetic insulators, yet had been predicted to be metallic by methods which treat strong interactions less carefully. In fact, we saw in the previous report that, a method using the Hamiltonian called the tight-binding approximation was used in order to get a more precise understanding of the interaction taking place at the atomic level of various compounds. However, in the last report, we never took into account the impact of electron-electron interactions, a well known physical process with a huge impact.

On the other hand, **double occupancy plays an important role in the study of correlated systems.** It is indicative of the Mott metal-insulator transition and of local moment formation. In optical lattice experiments, the double occupancy gives information about the phase. On the theory side, the double occupancy has been used to benchmark approximations.

In this exercise, we will mix these three principles: Hamiltonian, electron-electron interactions and correlated systems to **study the mean-field approximation of the Hubbard model.** The Hubbard model is an approximate model used, especially in solid-state physics, to describe the transition between conducting and insulating systems. **The Hubbard model, is the simplest model of interacting particles in a lattice, with only two terms in the Hamiltonian**: a kinetic term allowing for tunneling ("hopping") of particles between sites of the lattice and a potential term consisting of an on-site interaction. The model was originally proposed to describe electrons in solids and has since been the focus of particular interest as a model for high-temperature superconductivity. For electrons in a solid, **the Hubbard model can be considered as an improvement on the tight-binding model, which includes only the hopping term.** For strong interactions, it can give qualitatively different behavior from the tight-binding model and correctly predicts the existence of so-called Mott insulators, which are prevented from becoming conducted by the strong repulsion between the particles.

In this model, an electron placed on a given site interacts by means of the Coulomb interaction with a mean population of electrons with an opposite spin on the same site. The overall Hamiltonian composed of the tight-binding term and the Coulomb repulsion term expressed in the language of second quantization given by :

$$\hat{\mathcal{H}} = -t \sum_{<i,j>,\sigma} a^{\dagger}_{i,\sigma} a_{j,\sigma} + U \sum_{i,\sigma} n_{i,\sigma} \langle n_{-\sigma} \rangle,$$

The main idea behind this approximation is that a doubly occupied lattice site adds a potential energy U to the system, as a consequence of the electron-electron repulsion. **In practice, the mean-field approximation of the Hubbard model Hamiltonian can be solved numerically similar to the standard tight-binding model. The goal of this exercise, however, is not to perform such calculations, but to translate the second quantization language of quantum physics into the linear algebra language of numerical calculations and analyze a property of interest.** 128-sites fragment of graphene for different values of U/t and the latter are given. The goal is to **examine the evolution of the ground-state double occupancy as a function of the U/t ratio.**

The singular value decomposition provides us with a strategy to address the double occupancy of the system, without actually computing the ground-state. In fact, the double occupancy operator for the k-th site is given by $D_k = n_{k,+1} * n_{k,1}$. The quantity of our interest is

$$\frac{1}{n} \sum_{k=1}^{n} < \psi | D_k | \psi > .$$

Concretely, **for each value of U/t, we have to compute the previous quantity which is the mean value over all eigenstates of the double occupancy operator, where** $n_{i,\sigma} = a^{T}_{i,\sigma} a_{i,\sigma}$**, the**

operator that count the number of occupancy of a given state. Let us define O_k as the n*2 matrix built by taking the two columns of Q referring to the k-th site (i.e., columns k and k + n). Q is the matrix whose rows contain the first n eigenstates of the Hamiltonian. Such O_k matrix can be decomposed as $O_k = S\Sigma V$. In particular, S is a unitary n*n matrix. Thus, we can write the singular decomposition of O_k as : $O_k = S^T$. Given that, we can rewrite $< \psi|D_k|\psi >$ as :

$$< \psi|D_k|\psi >= \left[(S^\dagger Q)_{1,k}(S^\dagger Q)_{2,k+n} - (S^\dagger Q)_{1,k+n}(S^\dagger Q)_{2,k}\right]^2.$$

To model and compute this entire process we process as follow :

- We import the wavefunctions with the *.csv* files (matrices Q) given on moodle into Matlab.
- For each file, we compute the equation described above meaning that for each k, we first build O_k, and then we find S and compute the contribution.
- We plot the expectation value of double occupancy as a function of U/t (the values of U/t are listed in values u.csv given on moodle)

The two first steps were achieved with the following piece of code presented in Listing 7.

Listing 7: Code allowing the loading of all the files and the calculus of O_k :

```
% Initialization of the variable to store the results
result = zeros(36,1);
% We iterate on each file given on moodle
for i=1:36
    % Loading of each file
    Q = csvread(char(['./wavefunctions/wavefunctions_' num2str(i) '.csv']));
    [n,~]=size(Q);

    % We compute the equation given in the exercise paper
    mean_k = zeros(n,1);
    for k=1:n
        O = [Q(:,k) Q(:,k+n)];
        [S,T,V] = svd(O);
        matrix = conj(S')*Q;
        mean_k(k) = (matrix(1,k)*matrix(2,k+n)-matrix(1,k+n)*matrix(2,k))^2;
    end
    result(i) = sum(mean_k)/n;
end
```

In Figure 16 the evolution of the expectation value of the double occupancy operator as a function of U/t is shown.

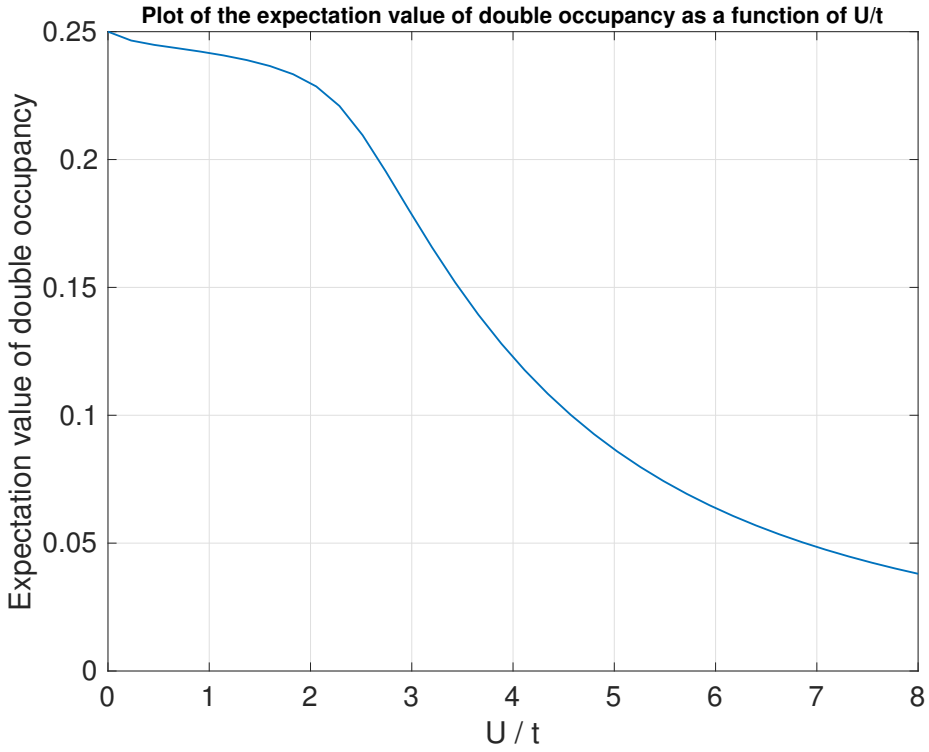

Figure 16: Expectation value of double occupancy as a function of U/t

Several facts can be noticed from the previous figure. But first we need to recall some information: **there are 4 statistically equivalent configurations** for the spin of 2 electrons in our case which are 'up and up', 'up and down', 'down and up' and finally 'down and down'. Moreover, by definition, **two electrons with opposite spins have more potential energy than when their spins are pointing in the same direction.**

First, we can see that **the overall trend is a decreasing trend.** The decrease between U/t in the range of 0 to 2 is small and then the decrease is higher from 2 to 5 and finally, the trend is less steep. We also note that **the first expectation value of the double occupancy is 0.25. This makes a lot of sense as there are 4 different configurations.** In fact, when U/t is near to 0, the 4 configurations appears with an equal probability which makes sense.

Then, as we said, **the expectation value of double occupancy decrease while U/t increases meaning that the Coulomb interaction promotes the configuration that minimizes the energy between two spins.** When U/t increase, it means that we are more in the situation where two electrons have opposite spins (meaning that they have more potential energy). **The described decreasing trend implies that this configuration is, therefore, less likely to happen** as we can see in Figure 16.

To conclude, due to the predominance of potential energy over the kinetic energy, we observe that the double occupancy expectation decrease with the potential.