Computational Physics III: Report 2
Computational linear algebra : Systems of linear equations and eigenvalue problems

Due on May 2, 2019

Nicolas Lesimple

Contents

Problem 1

Implementation of the LU Decomposition

LU decomposition of a matrix is the factorization of a given square matrix into two triangular matrices, one upper triangular matrix (U) and one lower triangular matrix (L), such that the product of these two matrices gives the original matrix. It was introduced by Alan Turing in 1948, who also created the Turing machine.

This method of factorizing a matrix as a product of two triangular matrices has **various applications such as solution of a system of equations, which itself is an integral part of many applications such as finding current in a circuit and solution of discrete dynamical system problems, finding the inverse of a matrix and finding the determinant of the matrix.** Basically, the LU decomposition method comes handy whenever it is possible to model the problem to be solved into a matrix form. Conversion to the matrix form and solving with triangular matrices makes it easy to do calculations in the process of finding the solution.

A square matrix A can be decomposed into two square matrices L and U such that A = L*U where U is an upper triangular matrix formed as a result of applying Gauss Elimination Method on A; and L is a lower triangular matrix with diagonal elements being equal to 1. **In this exercise, the LU decomposition algorithm will be implemented with and without pivoting. This algorithm will be integrated into a linear system solver that used also forward and backward substitution.**

(1)

The linear system shown in Figure 1 has to be solved by performing the LU decomposition and backward substitution. A module named *solve_ls* has been developed to allow this kind of task. The code of this function is shown in the Listing 1 just below :

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 13 \\ 2x_1 + 3x_2 - x_4 &= -1 \\ -3x_1 + 4x_2 + x_3 + 2x_4 &= 10 \\ x_1 + 2x_2 - x_3 + x_4 &= 1 \end{aligned}$$

Figure 1: Linear system needed to be solved

Listing 1: A script which solves linear system performing the LU decomposition and backward substitution :

```
function [x] = solve_ls(A,b)
    % solve_ls solves the system of linear equations A*x=b, for a square
    % matrix
    % A and a column vector b, using the LU decomposition and backward
    % substitution
    % ARGS :
    %   - A : Square matrix representing the system.
    %   - b : Vector of size (n,1) representing the solution of the ls
    %   system.
    % RETURN :
    %   - x : Vector representing the solution of the ls system Ax = b.

    [L, U] = LU_square_matrix(A);
    y = Forward(L,b);
    x = Backward(U,y);

end

```

```
function x = Backward(U,y)
    % Solves the nonsingular upper triangular system  Ux = y.
    % ARGS :
    %   - U : Square matrix representing the system. U is (n,n)
    %   - y : Vector of size (n,1) representing the solution of the ls
    %   system Ly = b. y is (n,1).
    % RETURN :
    %   - x : Vector representing the solution of the ls system Ux = y and
    %   also Ax = b.
    %   x is (n,1).

    n = length(y);
    x = zeros(n,1);
    for j=n:-1:2
        x(j) = y(j)/U(j,j);
        y(1:j-1) = y(1:j-1) - x(j)*U(1:j-1,j);
    end
    x(1) = y(1)/U(1,1);
end

function y = Forward(L,b)
    % Solves the nonsingular lower triangular system  Ly = b
    % ARGS :
    %   - L : Square matrix representing the system. L is (n,n)
    %   - b : Vector of size (n,1) representing the linear system.
    % RETURN :
    %   - y : Vector representing the solution of the ls system Ly = b.
    %   y is (n,1).

    n = length(b);
    y = zeros(n,1);
    for j=1:n-1
       y(j) = b(j)/L(j,j);
       b(j+1:n) = b(j+1:n) - L(j+1:n,j)*y(j);
    end
    y(n) = b(n)/L(n,n);
end
```

The good behavior of the module was tested with the given script *test_solve.m*. In this test, the solutions obtained using our module are compared to the one found with Matlab function for identity, real and imaginary random matrices. In all the tested case, the difference between Matlab functions and our own implementation was below 1^{-12}.

Thus, this module was applied to our linear system defined in Figure 1 and the results found was that x = [2,0,6,5]. **The mean difference between the Matlab function and the implemented module was $-9.7961e^{-18}$ allowing us to conclude that the two methods are equal and have the same output.** We also check the output by doing manually the decomposition and the backward substitution. The solution x found by hand was exactly the same thus our code is reliable.

(2)

The goal of this second question is to **implement the LU decomposition of a square matrix without pivoting.** The code allowing this process is described in the following Listing 2. We apply the 'greedy' algorithm explained in the course using two nested for loops.

Listing 2: A script which performs the LU decomposition for a square matrix without pivoting :

```
function [L, U] = LU_square_matrix(A)
    % LU_square_matrix computes the LU decomposition of a square matrix
    % without pivoting.
    % ARGS :
    %   - Matrix on which we want to apply the decomposition.
    % RETURN :
    %   - lower matrix L, and upper matrix U such that A = L*U
    n = size(A, 1);
    L = eye(n); % We first fill up the lower triangular half
    for k = 1 : n
        % For each row k, access columns from k+1 to the end and divide by
        % the diagonal coefficient at A(k ,k)
        L(k+1:n,k) = A(k + 1 : n, k) / A(k, k);
        for l=k+1:n % For each row k+1 to the end, perform Gaussian elimination
            A(l, :) = A(l, :)-L(l, k)*A(k, :); % A is U at the end
        end
    end
    U = A;
end
```

The implementation was tested on random matrices of size up to 100 and compared with Matlabs lu function. For this comparison, two plots were created which are illustrating the difference between the solutions provided by the two methods (one plot for L and one plot for U). These comparisons are visible in Figure 2. Matlab algorithm has one particularity: it typically performs row permutations. Indeed, [L, U, P] = lu(A) gives L, U and the permutation matrix P. **In order to compare properly Matlab code with our own implementation, we run our LU function on $A' = P * A$, where P is taken from Matlabs output.** The code that allows this comparison and that create the plot is shown in Listing 3.

Listing 3: A script which performs the comparison between LU decomposition of Matlab and our own :

```
% Question 2 : Implement the LU decomposition of a square matrix.
% Test your implementation on random matrices of size up to 100.
diff_L = zeros(20,1);
diff_U = zeros(20,1);
diff_LU = zeros(20,1);
for i = 1:20
    m = randi(100,5*i,5*i);
    [L_matlab, U_matlab, P] = lu(m);
    LU_matlab = L_matlab*U_matlab;
    % In order to compare properly, run your LU function on A0 = P A,
    % where P is taken from Matlab's output.
    m_matlab = P*m;
    [L_me, U_me] = LU_square_matrix(m_matlab);
    LU_me = L_me*U_me;
    size_tmp = size(L_me);
    tmp_L = 0;
    tmp_U = 0;
    tmp_LU= 0;
    for k = 1: size_tmp(1)
        for j = 1: size_tmp(2)
            tmp_L = tmp_L + (L_me(k,j)^2-L_matlab(k,j)^2);
            tmp_U = tmp_U + (U_me(k,j)^2-U_matlab(k,j)^2);
            tmp_LU = tmp_LU + (LU_me(k,j)^2-LU_matlab(k,j)^2);
        end
    end
    diff_L(i) = abs(tmp_L);
    diff_U(i) = abs(tmp_U);
    diff_LU(i) = abs(tmp_LU);
end

% Question 2 : Compare your implementation with Matlab?s lu function
% plotting the difference between the solutions provided by the two methods.
figure(1)
x = linspace(5,100,20);
plot(x, diff_L, '-r', 'LineWidth', 0.1)
leg1 = legend('L matrix Difference','Location','NorthEast');
set(leg1,'FontName','Arial','FontSize',10)
title1 = title({'Plot of the difference between L matrices obtained ',...
    'with our own implementation and with Matlab build in funciton :'});
set(title1,'FontName','Arial','FontSize',12)
xlabel('Size of the square random matrix (n*n)','FontName','Arial','FontSize',10);
ylabel('Difference [-]','FontName','Arial','FontSize',10);
grid on;
filename='./plot/LU_decomposition_square_matrix_comparison_L.eps';
print(gcf,'-depsc',filename)
```

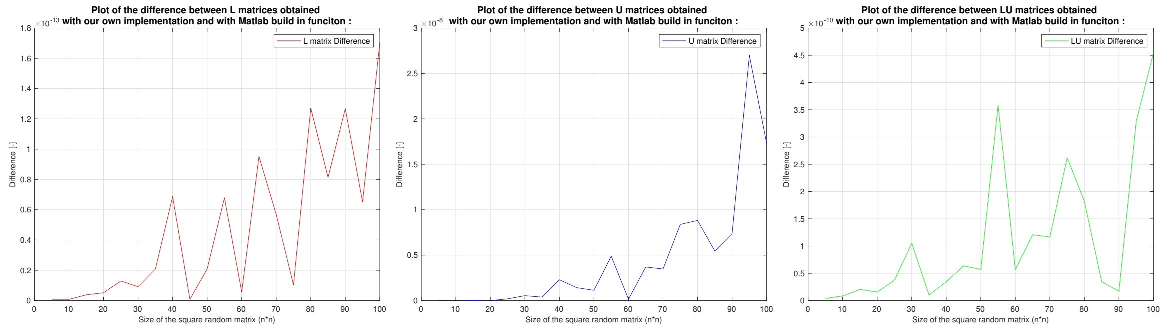


Figure 2: LU matrices difference between Matlab and our own implementation.
Left : L difference.
Middle : U difference.
Right : LU difference.

Thanks to the plot of the difference between the element of the different matrix, we are able to conclude that the overall difference is in the order of $10e^{-13}$ for L, $10e^{-8}$ for U, and $10e^{-10}$ for LU. To calculate this difference, the absolute error between the two matrices returned by the LU decomposition was used :

$$\|A - B\| = |\sum_i \sum_j (a_{ij}^2 - b_{ij}^2)|$$

As we see just before the error is extremely low. It is therefore nearly negligible. However, it seems that it increases as the size of the matrices increases. This make sense when we look at the way I we choose to compute the error. In fact, the larger size a matrix have, the more coefficient a matrix have, the more opportunities are given to these coefficient to deviate from their expected values. **Thus, the conclusion is that the Matlab implementation and our own LU decomposition implementation are similar and thus that the module we create is reliable for square matrix.**

Problem 1 continued on next page. . .

(3)

The previous implemented LU decomposition was then tested on the following matrix : $A = \begin{vmatrix} 0 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{vmatrix}$

The results of this decomposition were the following matrix :

$$L = \begin{vmatrix} 1 & 0 & 0 \\ -Inf & 1 & 0 \\ Inf & NaN & 1 \end{vmatrix} ; U = \begin{vmatrix} 0 & -7 & 0 \\ NaN & Inf & NaN \\ NaN & NaN & NaN \end{vmatrix} ; L * U = \begin{vmatrix} NaN & NaN & NaN \\ NaN & NaN & NaN \\ NaN & NaN & NaN \end{vmatrix} ;$$

Here, we can observe that some coefficient of the returned matrix L are Inf, -Inf or NaN. Thus, a problem happen. This is mainly due to our code. Indeed, if one looks carefully at the code, one can see that there is a risk that the code performs a division by 0 which gives the result Inf. More specifically, the code do a division by 0 if there is any 0 on the diagonal of the input matrix. This can be fixed by using the LU decomposition with pivoting and will be done in the next exercise. The NaN values comes form calculus made with these Inf values. That's why the quantity of NaN increase between L, U and then LU. **Thus, we can conclude that the result is not meaningful due to an issue in our code. The issue is a potential division by zero creating Inf values.**

(4)

In this part of the homework, we need to implement the LU decomposition **with pivoting. Our function accepts a square matrix A as input and returns 3 square matrices L, U, P.** This algorithm simply excludes the possibility to perform a division by 0 thanks to the usage of P matrix. The code is therefore much more stable and the problem which occurred in the last paragraph should not happen again. The given script *test_lu.m* was used to make sure our implementation is correct. As it passed all the test, our module can be reliable. The code allowing this LU decomposition with pivoting is shown in Listing 4.

Listing 4: A script which performs the LU decomposition of a square matrix with pivoting:

```
function [ L, U, P ] = lu_decomposition(A)
    % lu_decomposition computes the LU decomposition with pivoting
    % for a square matrix A
    % ARGS :
    %   - A : Square matrix on which we want to apply the decomposition.
    % RETURN :
    %   - Return lower matrix L, upper matrix U and permutation matrix P
    %   such that P*A=L*U
    [n,n]=size(A);
    L=eye(n);
    P=L;
    U=A;
    for k=1:n
        [pivot m]=max(abs(U(k:n,k))); % pivoting process
        m=m+k-1;
        if m~=k
            % interchange rows m and k in U
            temp=U(k,:);
            U(k,:)=U(m,:);
            U(m,:)=temp;
            % interchange rows m and k in P
            temp=P(k,:);
            P(k,:)=P(m,:);
            P(m,:)=temp;
            if k >= 2
                temp=L(k,1:k-1);
                L(k,1:k-1)=L(m,1:k-1);
                L(m,1:k-1)=temp;
            end
        end
        for j=k+1:n
            L(j,k)=U(j,k)/U(k,k);
            U(j,:)=U(j,:)-L(j,k)*U(k,:);
        end
    end
end
```

The reliable module was then tested on the previous matrix A of question 3. The results of this decomposition were the following matrix :

$$L = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -0.6 & -0.2 & 1 \end{vmatrix}; U = \begin{vmatrix} 5 & -1 & 5 \\ 0 & -7 & 0 \\ 0 & 0 & 9 \end{vmatrix}; P = \begin{vmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix}; P^{-1} * L * U = \begin{vmatrix} 0 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{vmatrix};$$

The difference between the LU decomposition is $3.5527e^{-15}$ between the U matrices is 0, between the L matrices is $1.1102e^{-16}$ and between the P matrices is 0. **We thus can easily conclude that this decomposition make sense and thus that our implemented module is reliable.**

Problem 2

Barycenter

In astronomy, the barycenter is the center of mass of two or more bodies that orbit one another and is the point which the bodies orbit. In this exercise, we will study an artificial system consisting of 6 astronomical bodies. The mass of each object will be calculated by solving a linear system. **In fact, the masses and position of each object are linked with the position of the overall barycenter and the overall mass by a mathematical formula.** The distances will be calculated in light-minutes, while the masses are in Earths masses.

We assume that the system is flat (like our solar system) and the objects move in a plane. We know from a snapshot of the system the following coordinates of the astronomical objects : Alpha(0,6), Beta(9,0), Gamma(3,5), Delta(3,4), Epsilon(10,12) and Zeta(7,0).

Then, inside the overall system, some smaller subsystem are defined. The table below lists known barycenters in the system.

Coordinate	Objects	Overall mass
(6,2)	*Alpha,Beta*	12
(4,6)	*Gamma,Epsilon*	14
(25/8,15/4)	*Alpha,Delta,Zeta*	32

In addition, in the case of a system of objects P_i (i=1,...,n), the overall mass M, individual object mass m_i, coordinate r_i, and the center of mass R are linked with the following equation

$$R = \frac{1}{M} * \sum_{i=1}^{n} m_i * r_i$$

Thanks to the definition of these subsystem and from center of mass definition, we are able formulate this problem as a system of linear equation :

$$\begin{cases} \frac{9}{12} * m_{beta} = 6 \\ \frac{6}{12} * m_{alpha} = 2 \\ \frac{3}{14} * m_{gamma} + \frac{10}{14} * m_{epsilon} = 4 \\ \frac{5}{14} * m_{gamma} + \frac{12}{14} * m_{epsilon} = 6 \\ \frac{3}{32} * m_{delta} + \frac{7}{32} * m_{zeta} = \frac{25}{8} \\ \frac{6}{32} * m_{alpha} + \frac{4}{32} * m_{delta} = \frac{15}{4} \end{cases}$$

In our case, each one of the subsystems allows us two to create two different equations. One is corresponding to x coordinate and the other to y coordinate. Thanks to the definition of the linear system, we are able to define the matrix A, the vector b and x theoretical vector allowing us to have the following problem to solve $Ax = b$:

$$A = \begin{vmatrix} 0 & \frac{9}{12} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{3}{14} & \frac{10}{14} & 0 & 0 \\ 0 & 0 & \frac{5}{14} & \frac{12}{14} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{3}{32} & \frac{7}{32} \\ \frac{6}{32} & 0 & 0 & 0 & \frac{4}{32} & 0 \end{vmatrix} ; x = \begin{vmatrix} m_{alpha} \\ m_{beta} \\ m_{gamma} \\ m_{epsilon} \\ m_{delta} \\ m_{zeta} \end{vmatrix} ; b = \begin{vmatrix} 6 \\ 2 \\ 4 \\ 6 \\ \frac{25}{8} \\ \frac{15}{4} \end{vmatrix} ;$$

By solving this system, we are able to find the mass of each objects (e.m stands for Earths mass) :

$$x = \begin{vmatrix} m_{alpha} \\ m_{beta} \\ m_{gamma} \\ m_{epsilon} \\ m_{delta} \\ m_{zeta} \end{vmatrix} = \begin{vmatrix} 4 \\ 8 \\ 12 \\ 2 \\ 24 \\ 4 \end{vmatrix} [e.m];$$

We then **apply the formula** defining the center of mass of an object and thus we are able to finally find the correct barycenter of the overall system where each number correspond to the coordinate of the corresponding object r_i for x or y:

$$R_x = (m_{alpha} * 0 + m_{beta} * 9 + m_{gamma} * 3 + m_{epsilon} * 10 + m_{delta} * 3 + x m_{zeta} * 7)/\sum m_i) = 4.2222;$$

$$R_y = (m_{alpha} * 6 + m_{beta} * 0 + m_{gamma} * 5 + m_{epsilon} * 12 + m_{delta} * 4 + m_{zeta} * 0)/\sum m_i) = 3.7778;$$

Thus, the center of mass of the overall system is situated at the point (4.2222, 3.7778). These values are in the units of light-minutes ($1 l.m \approx 1.8 * 10^{10} m$).

Problem 3

Lotka-Voterra equations

The LotkaVolterra equations, also known as the **predatorprey equations, are a pair of first-order nonlinear differential equations, frequently used to describe the dynamics of biological systems in which a number of species interact being either predators or prey with respect to each other.** Populations x_i change through time t according to the following set of equations :

$$\frac{dx_1}{dt} = r_1 * x_1 x_1 * \sum_i A_{1i} * x_i$$

$$\frac{dx_2}{dt} = r_2 * x_2 x_2 * \sum_i A_{2i} * x_i$$

$$\frac{dx_N}{dt} = r_N * x_N x_N * \sum_i A_{Ni} * x_i$$

More precisely, we will look at the case where only two species interact, one as a predator and the other as prey. The populations change through time according to the pair of equations:

$$\frac{dx}{dt} = \alpha x - \beta xy,$$
$$\frac{dy}{dt} = \delta xy - \gamma y,$$

where

- x is the number of prey (for example, rabbits);
- y is the number of some predator (for example, foxes);
- $\frac{dy}{dt}\frac{dy}{dt} and \frac{dx}{dt}\frac{dx}{dt}$ represent the instantaneous growth rates of the two populations;
- t represents time;
- $\alpha, \beta, \gamma, \delta$ are positive real parameters describing the interaction of the two species.

The LotkaVolterra model makes a number of assumptions, not necessarily realizable in nature, about the environment and evolution of the predator and prey populations:

- The prey population finds ample food at all times.
- The food supply of the predator population depends entirely on the size of the prey population.
- The rate of change of population is proportional to its size.
- During the process, the environment does not change in favor of one species, and genetic adaptation is inconsequential.
- Predators have a limitless appetite.

As differential equations are used, the solution is deterministic and continuous. This, in turn, implies that the generations of both the predator and prey are continually overlapping.

Prey: When multiplied out, the prey equation becomes : $\frac{dx}{dt} = \alpha x - \beta xy$. The prey is assumed to have an unlimited food supply and to reproduce exponentially unless subject to predation; this exponential growth is represented in the equation above by the term αx. The rate of predation upon the prey is assumed to be proportional to the rate at which the predators and the prey meet, this is represented above by βxy. If either x or y is zero, then there can be no predation. With these two terms, the equation above can be interpreted as follows: **the rate of change of the prey's population is given by its own growth rate minus the rate at which it is preyed upon.**

Predators : The predator equation becomes $\frac{dy}{dt} = \delta xy - \gamma y$. In this equation, δxy represents the growth of the predator population. (Note the similarity to the predation rate; however, a different constant is used, as the rate at which the predator population grows is not necessarily equal to the rate at which it consumes the prey). γy represents the loss rate of the predators due to either natural death or emigration, it leads to exponential decay in the absence of prey. **Hence the equation expresses that the rate of change of the predator's population depends upon the rate at which it consumes prey, minus its intrinsic death rate.**

(1)

Now we assume that only a single type of species exists and its population is described by $\frac{dx}{dt} = rx - Ax^2$ with both r and A being positive. In this case, the equation is similar to the one of the prey describe just above. If $\alpha = r$, $\beta = A$ and y = x, we find exactly the prey equation. Thus the first term rx is representing the exponential growth of this population. In fact, preys are assumed to have an unlimited food supply and to reproduce exponentially, unless subject to predation. In our case, by definition with only one species, predation can't exist. That's why we only have one term $A * x^2$: this term represents the loss rate of the population due to either natural death or emigration. **We can thus conclude from this one species equation that the rate of change of the species' population is given by its own growth rate minus its intrinsic death rate.**

In fact, in our case we have just one species. As we explained just before, the $A * x^2$ represents the competition of the specie with the other species and with itself. That why we have x^2 because it is proportional to the rate at which the species meet. Here, we only have one competition within the same specie. This competition can be physically reprensent by the decrease of food or water resources as the population increase.

Let's now **look at the equilibrium of this differential equation. By definition, the equilibrium of the system is defined where the derivate of the system is null** and thus in the case of a differential system of the first order as our, the system is at equilibrium where x = 0 or where $r - Ax = 0$ which is equivalent to say when Ax=r. **This means that the species are at equilibrium when the growth rate is equal to the death rate, where the death rate is linked to the population of the species.**

(2)

Now, we consider 4 interacting populations with

$$A_{ij} = \begin{vmatrix} 0 & 10 & 50 & 0 \\ -1 & 3 & 10 & 0 \\ -2 & 10 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}; ri = \begin{vmatrix} 5 \\ 0 \\ 0 \\ 1 \end{vmatrix};$$

To find the equilibrium population of species we need to solve the linear system Ax = r, i.e all the time derivatives are zero. It means that we will find x vector corresponding to the size of each population allowing the apparition of the equilibrium. To do that, we use the build in Matlab function. This computation gives us the following result :

$$ri = \begin{vmatrix} 1.25 \\ 0.25 \\ 0.05 \\ 1 \end{vmatrix};$$

For example, to reach the equilibrium, the system should be composed of 125 individual of population 1, 25 individual of population 2, 5 individual of population 3 and 100 individual of population 4.

To be more precise, we can analyze, among these populations, which ones act like predators and which ones act like a prey. To do that, we reffered to the above analysis of the problem. As we explained just before, the prey follow an equation of this type $\frac{dx_i}{dt} = r_i x_i - \beta x_i x_j$. while predators follow an equation of this type $\frac{dx_j}{dt} = \delta x_i x_j - \gamma x_j$., where the population i is a prey population and the population j is a predator population (where $-\gamma = r_j$). In our case, we have the following linear problem :

$$\begin{cases} -10 * x_2 - 50 * x_3 + 5 = 0 \\ 1 * x_1 - 3 * x_2 - 10 * x_3 \quad = 0 \\ 2 * x_1 - 10 * x_2 \quad = 0 \\ -x_4 + 1 = 0 \\ \end{cases}$$

Thus, the population should have a negative r_i and positive $A_{i,j}$ coefficient to be a predator. **However, in the definition of our problem, A has a minus sign and thus, a negative value of A coefficient in our above matrix means an increase of the population. On the contrary, a positive value for A implies a decrease of the number of individuals.**

In our system, the population corresponding to the first one x_1 is thus a prey. The population 1 is modelled by an equation that expresses that the rate of change of the preys x_1 population depends upon its own growth rate (which is modeled by the constant 5) minus the rate at which it is preyed upon. In our case this population has 2 predators : x_2 and x_3 as they are both implies in the decrease of the population. **Thanks to this first equation, we are able to see that x_1 is a prey population and that x_2 and x_3 are predators. This fact can be verified in the equation representing these two following populations.**

First, let's focus at the second equation corresponding to the population 2. In this equation, the prey term is visible with a positive coefficient associate to x_1 factor meaning that the more prey of population 1 are present (equivalent to food) the faster the increase of the population 2 will be. Then, we can see that the growth rate depends on two factors x_2 and x_3. Thus the population decrease more if there is a lot of individuals of this population 3 meaning that species 2 and 3 are predator one for the other. They both have negative number describing their interaction. They can be direct predators (they can eat each other) or they can depend on the same food or water resources. Specie 2 also has a negative influence on itself which means that its individuals must fight between each other for resources. For population 3, we thus see the artifact of the predation with population 2 (if population 2 increase, population 3 will decrease) and the prey' effect due to population 1 (if population 1 increase, population 3 will increase). For both population 2 and 3, r coefficient is null because the birth and death rate are already modeled by the relations between the species.

Then, to conclude this analysis of the system, we will focus on the population 4 corresponding to the fourth equation. **This population does not depend on others.** In fact, we find exactly the same equation format as for the one explained in question 1. Indeed, the only non-zero coefficient of A is on the diagonal which means that specie 4 is only in competition with itself. **This type of equation describes a single species model where the rate of change of the species' population is given by its own growth rate minus its intrinsic death rate.** In this case, -1 is the death rate and x_4 represent the population proportion that makes the growth rate changing.

Problem 4

Conditioning

In this exercise, **we will analyze the importance of the conditioning of a linear system. To illustrate the crucial relevance of this, we will solve the following two systems of linear equations that have only a minor difference in the free coefficient of the first equation.** We found the solution of the system using the build in Matlab function to solve the system Ax = b and find x:

$$A_1 = \begin{vmatrix} 1 & 0 & 1 \\ 1.001 & 1 & 0 \\ 0 & -1 & 1 \end{vmatrix} ; b_1 = \begin{vmatrix} 2 \\ 1 \\ 1 \end{vmatrix} ; x_1 = \begin{vmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \end{vmatrix} ;$$

$$A_2 = \begin{vmatrix} 1 & 0 & 1 \\ 1.001 & 1 & 0 \\ 0 & -1 & 1 \end{vmatrix} ; b_2 = \begin{vmatrix} \mathbf{2.001} \\ 1 \\ 1 \end{vmatrix} ; x_2 = \begin{vmatrix} \mathbf{-1} \\ \mathbf{2.001} \\ \mathbf{3.001} \end{vmatrix} ;$$

As you can see, **the solution to the two similar systems (x_1 and x_2) is extremely different. This difference is huge and mainly due to conditioning. We can thus conclude, with this illustration, that the sensitivity of solutions to small deviations in the right-hand side is extreme.** In fact, in our case, for the addition of 0.001 on the first element of the b vector, the solution of the system is transformed from the vector [0,1,2] to the vector [-1, 2.001, 3.001].

In the field of numerical analysis, the condition number of a function with respect to an argument measures how much the output value of the function can change for a small change in the input argument. This is used to measure how sensitive a function is to changes, and how much error in the output results from an error in the input.

The condition number is an application of the derivative and is formally defined as the value of the asymptotic worst-case relative change in output for a relative change in input : the condition number is called $\kappa(A)$: if A is a non-singular matrix (i.e if $det(A) \neq 0$), the condition number is given by $\kappa(A) = \|A^{-1}\| * \|A\|$ where $\|A\| = max_{\|x\|=1}\|A * x\|$. The condition number represents an upper bound for the change δx :

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta b\|}{\|b\|}$$

The condition number is frequently applied to questions in linear algebra. More generally, condition numbers can be defined for non-linear functions in several variables.

A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned. For non-singular matrices, we have the following properties and implications :

- $\kappa(A) \geq 1$
- if $\kappa(A) \sim 1$ the system is well conditioned and thus error in x are not much larger than the errors in b.
- if $\kappa(A) \gg 1$ the system is ill-conditioned and thus errors in x are amplified with respect to errors in b.

As a rule of thumb, if the condition number $\kappa(A) = 10^k$, then you may lose up to k digits of accuracy on top of what would be lost to the numerical method due to loss of precision from arithmetic methods. **However, the condition number does not give the exact value of the maximum inaccuracy that may occur in the algorithm. It generally just bounds it with an estimate (whose computed value depends on the choice of the norm to measure the inaccuracy).**

In our case, the condition number of the matrix is $5.1998e^{03} \approx 5200 \gg 1$. **We are thus in the case of an ill-conditioned system which explained the huge sensibility of the system solution. In fact, in our case, the sensitivity of solutions to small deviations in the right-hand side is extreme due to this conditioning phenomena. Indeed, a small change in b ($\approx 10^3$) induces a change in the solutions which is 3 order of magnitude higher.**

Problem 5

Power Methods

In mathematics, **power method is an eigenvalue algorithm:** given a diagonalizable matrix A, the algorithm will produce a number λ , which is the greatest (in absolute value) eigenvalue of A, and a nonzero vector v, the corresponding eigenvector of λ, such that $Av = \lambda v$. Thus, power iteration is a very simple algorithm, but it may converge slowly. It does not compute a matrix decomposition, and hence it can be used when A is a very large sparse matrix. It's straightforward to implement: **in order to obtain the eigenvector corresponding to a certain (smallest, largest, closest) eigenvalue of a matrix, one has to multiply an initial vector by a specific matrix multiple times.**

The power iteration algorithm starts with a vector b_0, which may be an approximation to the dominant eigenvector or a random vector. The method is described by the recurrence relation $b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$. So, **at every iteration, the vector b_k is multiplied by the matrix A and normalized.** If we assume A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues and the starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue, then a subsequence (b_k) converges to an eigenvector associated with the dominant eigenvalue. Without the two assumptions above, the sequence (b_k) does not necessarily converge.

The implementation of the Power Method was done in the Matlab file named *eig_power*. **This implementation was tested thanks to the test file named *test_power.m*. As the function passed all the test, we know that this module is reliable and can be used for further investigation.** This function is display on the following Listing 5:

Listing 5: Power Method

```matlab
function [vec, val] = eig_power(A)
    % Power method for computing eigenvalues
    % This function computes the eigenvalue lambda of largest module of
    % the matrix A and the corresponding eigenvector.
    % A tolerance is set up to specifies the tolerance of the method.
    % A number max of itereations is also specify.
    % z_0 is used as the initial guess.
    % ARGS :
    %   - A : matrix on which we want to find the eigenvalue lambda of
    %   largest module
    % RETURN :
    %   - vec : eigenvector corresponding to the largest module with unit
    %   norm.
    %   - val : eigenvector corresponding to the largest module

    % Define main parameters
    [na,ma] = size(A);
    nmax = 10000;
    tol = 10^(-10);
    z0 = ones(na,1);

    % Check the size of the input matrix A :
    if na ~= ma
        disp('ERROR:Matrix A should be a square matrix')
        return
    end

    % Apply power method :
    q=z0/norm(z0);
    q2=q;
    relres=tol+1;
    iter=0;
    z=A*q;
    while ((relres(end)>=tol) && (iter<=nmax))
        q=z/norm(z);
        z=A*q;
        val=q'*z;
        vec=q;
        z2=q2'*A;
        q2=z2/norm(z2);
        q2=q2';
        y1=q2;
        costheta=abs(y1'*vec);
        iter=iter+1;
        temp=norm(z-val*q)/costheta;
        relres=[relres; temp];
    end
return
```

In numerical analysis, **the inverse power method is an iterative eigenvalue algorithm.** It allows to find an approximate eigenvector when an approximation to a corresponding eigenvalue is already known. **The method is conceptually similar to the power method but it computes the eigenvalue lambda of smallest module of the input matrix and the corresponding eigenvector.**

The inverse power iteration algorithm starts with an approximation μ for the eigenvalue corresponding to the desired eigenvector and a vector b_0, either a randomly selected vector or an approximation to the eigenvector. **The method is described by the iteration $b_{k+1} = \frac{(A-\mu I)^{-1}b_k}{C_k}$, where C_k are some constants usually chosen as $C_k = \|(A-\mu I)^{-1}b_k\|$.** Since eigenvectors are defined up to multiplication by constant, the choice of C_k can be arbitrary in theory.

At every iteration, the vector b_k is multiplied by the matrix $(A-\mu I)^{-1}$ and normalized. **It is exactly the same formula as in the power method, except replacing the matrix A by $(A-\mu I)^{-1}$. The closer the approximation μ to the eigenvalue is chosen, the faster the algorithm converges**. However, incorrect choice of μ can lead to slow convergence or to the convergence to an eigenvector other than the one desired. In practice, the method is used when a good approximation for the eigenvalue is known, and hence one needs only few (quite often just one) iterations.

The implementation of the Inverse Power Method with a shift was done in the Matlab file named *eig_ipower*. **This implementation was tested thanks to the test file named *test_power.m*. As the function passed all the test, we know that this module is reliable and can be used for further investigation.**

Listing 6: Inverse Power Method

```
function [vec,val_k]=eig_ipower(A,mu)
    % Inverse power method with shift.
    % This function computes the eigenvalue lambda of smallest module of
    % the matrix A and the corresponding eigenvector.
    % A tolerance is set up to specifies the tolerance of the method.
    % A number max of itereations is also specify.
    % x_0 is used as the initial guess.
    % ARGS :
    %   - A : matrix on which we want to find the eigenvalue lambda of
    %   largest module
    %   - mu : It correspond to the shift.
    % RETURN :
    %   - vec : eigenvector corresponding to the smallest module with unit
```

Problem 5 continued on next page. . .

```
    %   norm.
    %   - val : eigenvector corresponding to the smallest module.

    % Define the parameters and the initial values
    epsilon=1e-15;
    vec0=rand(length(A),1);

    % First definition of the return eigenvector and eigenvalue
    vec=vec0/(norm(vec0));
    val=vec'*A*vec;
    vec=A*vec;
    vec=vec/(norm(vec));
    val_k=vec'*A*vec;

    % Iterate and change the return values since the difference is higher
    % than epsilon
    while ((abs(val-val_k)) > epsilon)
        vec=solve_ls((A-mu*eye(length(A))),vec);
        vec=vec/(norm(vec));
        val=val_k;
        val_k=vec'*A*vec;
    end

end
```

For a given complex Hermitian matrix M and nonzero vector x, the **Rayleigh quotient R(M, x), is defined as:** $R(M,x) = \dfrac{x^*Mx}{x^*x}$. For real matrices and vectors, the **condition of being Hermitian reduces to that of being symmetric, and the conjugate transpose** x^* **to the usual transpose** x'. It can be shown that, for a given matrix, the **Rayleigh quotient reaches its minimum value** $\lambda_{\min}$ **(the smallest eigenvalue of M) when x is** $v_{\min}$ **(the corresponding eigenvector)**. Similarly, $R(M,x) \leq \lambda_{\max}$ and $R(M,v_{\max}) = \lambda_{\max}$.

The Rayleigh quotient is used in the min-max theorem to get exact values of all eigenvalues. It is also used in eigenvalue algorithms (such as Rayleigh quotient iteration) to obtain an eigenvalue approximation from an eigenvector approximation.

The range of the Rayleigh quotient (for any matrix, not necessarily Hermitian) is called a numerical range (or spectrum in functional analysis). When the matrix is **Hermitian, the numerical range is equal to the spectral norm.** Still in functional analysis, $\lambda_{\max}$ is known as the spectral radius.

The implementation of the Rayleigh quotient method **combining inverse power method with a shift (=eigenvector from eigenvalue estimate) with Rayleigh quotient (=eigenvalue from eigenvector estimate)** was done in the Matlab file named *eig_rq*. **This implementation was tested thanks to the test file named *test_rq.m*. As the function passed all the test, we know that this module is reliable and can be used for further investigation.**

Listing 7: Rayleigh Quotient Method

```
function [vec, val] = eig_rq(input_matrix, target)
    % eig_rq computes the closest eigen vector and eigen value
    % of a given matrix with Rayleigh quotient iteration
    %
    % ARGS :
    %   - input matrix (2D complex Hermitian matrix) : matrix
    %   for the eigen value problem;
    %   - target (real scalar) : an estimation to the eigen value;
    % RETURN :
```

Problem 5 continued on next page. . .

```
    %   - a right eigen vector and the corresponding eigen value
    %   of a matrix.

    % Define main parameters
    [m,n] = size(input_matrix);
    nmax = 10000;
    tol = 10^(-10);
    x0 = ones(n,1);
    sigma = target;

    if m~=n
          disp('matrix input_matrix  is not square')  ;
          return;
    end

    vec = x0;
    for k = 0 : nmax
         val = (vec'*input_matrix*vec)/(vec'*vec);
         xhat = (input_matrix-sigma * eye(n,n))\vec;
         vec = xhat/max(xhat);
         if  norm( (input_matrix-val * eye(n,n))*vec )  < tol
             return;
         end
    end
end
```

Problem 6

Tight-binding model for a loop of atoms

One simple approach to modeling the electronic structure of molecular compounds and crystalline materials is the **tight-binding approximation**. In solid-state physics, the tight-binding model (or TB model) is an approach to the calculation of electronic band structure using an approximate set of wave functions based upon superposition of wave functions for isolated atoms located at each atomic site. **A numerical diagonalization of the information provides easy access to modeling the electronic properties of systems composed of a large number of atoms.**

This method will be used to study a loop of N identical atoms connected with chemical bonds. Finding the eigenvalues and eigenvectors of matrix H, where the matrix H is the Hamiltonian, **is equivalent to solving the Schrodinger equation**. The famous equation describes the behavior of particles at a quantum scale and more precisely it describes the evolution of the wave function which is indirectly linked to some real physical properties of a system. Thus, in this case, **eigenvalues of H correspond to discrete energy levels ϵ_i , while the corresponding eigenvectors ψi represent the eigenstates (one-electron wavefunctions)**. Each vector element corresponds to the wavefunction amplitude on the corresponding atom.

(1)

In this problem, we will consider a molecular system representing a loop of N identical atoms. It means that atom i is connected by a chemical bond to atom i + 1 and atom N is connected to atom 1. In the simplest form of the tight-binding approximation, the N*N matrix of the Hamiltonian of this system is written in the following way: matrix element H(i, j) = γ if atoms with indices i and j are connected by a chemical bond, H(i, j) = 0 otherwise. For simplicity, let's assume the value of the hopping integral $\gamma = -1$ (which quantifies the energetic aspect of the chemical bonds between atoms). Thus, if N = 5, our H matrix has the following shape :

$$H = \begin{vmatrix} 0 & -1 & 0 & 0 & -1 \\ -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 \\ -1 & 0 & 0 & -1 & 0 \end{vmatrix};$$

If N is bigger, you just add lines having -1 in the main diagonal. In our case, N = 20. Note also that this is a zero-dimensional system : there is no k dependence for the energy which is purely discrete.

Then to find the smallest ϵ_{min} eigenvalues corresponding to this H matrix, we use the **inverse power method** coded in the previous exercise using -10 as mu input (indication on eigenvalue we want to find). To find the largest ϵ_{max}, we do the same with mu equal to 10. This function gives us the following results :

$$\epsilon_{min} = -2; \epsilon_{max} = 2$$

We also confirm these results with the build in Matlab diagonalization algorithms *eig*. Using this Matlab function, we were able to draw the first observation: all the eigenvalues are degenerated two times (which means two orthogonal eigenvectors are associated with each of these values) except for ϵ_1 and ϵ_N. **These methods give us also the eigenvector corresponding to the eigenstates of each discrete energy levels.** The values of ϵ_1 and ϵ_N corresponds to the lowest and highest discrete energy levels allowed for the system. Each coordinate of the eigenvectors associated with each eigenvalue represents the wave function amplitude ψ_i on the corresponding atom of the loop. An eigenstate can be compared to wavefunction amplitude on the corresponding atom. Resulting eigenstates can be seen in Figure 3. **On this figure, we are able to visualize the value of the wavefunction in the function of the atom position.**

Problem 6 [(1)] continued on next page. . .

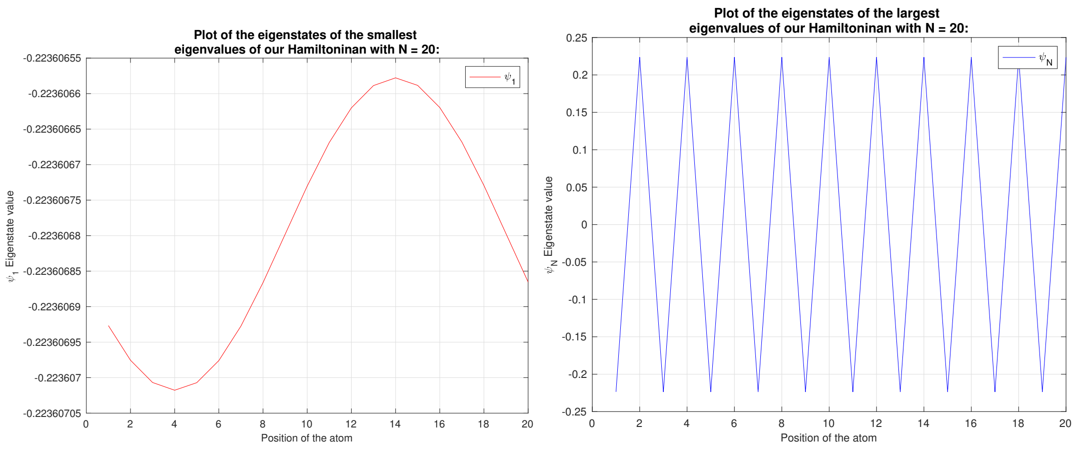


Figure 3: Eigenvector corresponding to the ψ_1 and ψ_N eigenvalue of the Hamiltonian matrix for N=20:

As you can see, **the eigenstates are really different when you consider the smaller or the higher energy level.** In fact, the lower energy level has the shape of a Gaussian curve while the highest energy level seems perfectly periodic, with values being positive and then negative. In addition, the values of the wavefunction of the largest eigenvalues cover a bigger and wider range of values.

In addition, knowing the wave functions, the plot of the probability densities $\mu_i = \psi_i * \psi_i$[m3] for these two states can be seen on Figure 4. **This kind of values is more interesting because it's easier to represent a probability density and it has a stronger physical meaning.**

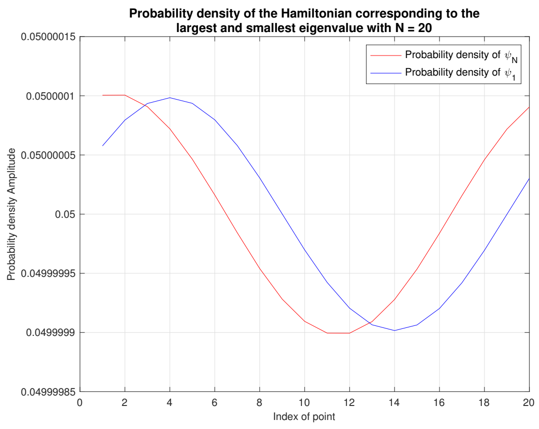


Figure 4: Probability densities corresponding to the ψ_1 and ψ_N eigenvalue of the Hamiltonian matrix for N = 20:

Here, we can conclude that the atom having the highest probability to be in the highest energy state in the molecule is the number 4 while the one having the highest probability to be in the lowest energy state is the atom number 2. These facts can be concluded from the definition of the probability density and Figure 4.

(2)

In this question, we want to analyze the results found for N=21. To do that, we re-use exactly the same procedure as before. The different results are displayed just below in Figure 6:

$$\epsilon_{min} = -2; \epsilon_{max} = 1.9777.$$

Here we can notice that the maximum eigenvalue is different compared to the case where N = 20 while the smallest eigenvalue remains the same. We also note that all the eigenvalues are degenerated two times except for ϵ_1. Unlike in the first case, our highest eigenvalues is two times degenerated.

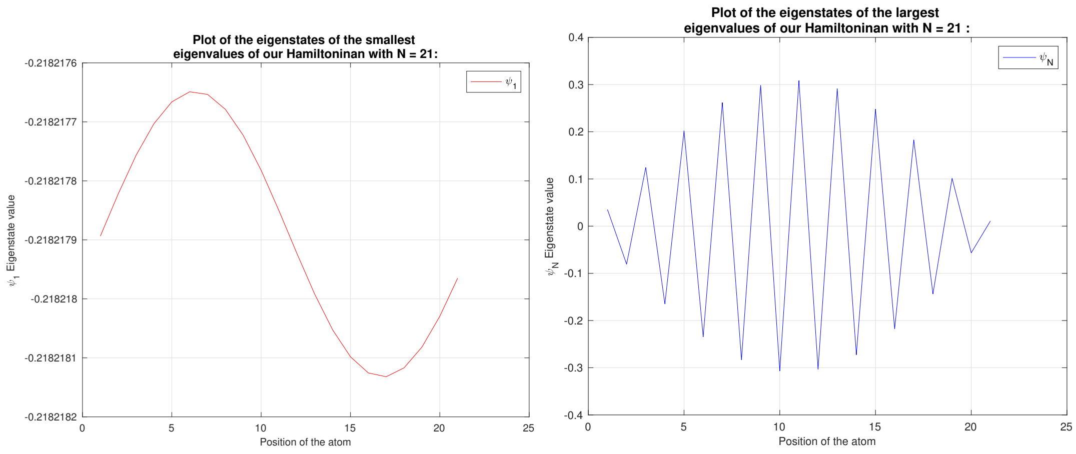


Figure 5: Eigenvector corresponding to the ψ_1 and ψ_N eigenvalue of the Hamiltonian matrix for N=21:

The eigenstates of the smallest eigenvalue have the same shape and the same magnitude as for N = 20. In opposite, the eigenvector of the largest eigenvalue is different. In fact, the symmetric and periodic characteristics present for N = 20 disappeared due to a decrease between position 10 and 20. It's a kind of beat-like behaviour curve. Again, the probability densities have been calculated and plot in Figure 6.

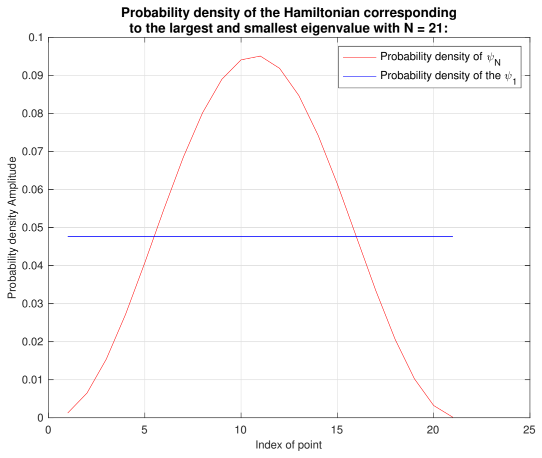


Figure 6: Probability densities corresponding to the ψ_1 and ψ_N eigenvalue of the Hamiltonian matrix for N=21:

The probability density associated with the smallest eigenvalue seems to be constant. However, when we do a strong zoom, some variation can be seen. Thus, from this point of view, the probability density corresponding to the smallest eigenvalue change a lot between N = 20 and N = 21. **In fact, oscillations have a much smaller amplitude than in the case N = 20.** In addition, the probability density linked to the largest eigenvalue has the same shape and did not seems to change.

Thanks to these observed differences between N = 20 and N = 21, more tests have been conducted for odd and even N. **Having an odd number N implies that it is not possible to impose that every eigenvalues (except the smallest and the highest) must be two times degenerated. Thus, the parity of N has a huge influenced on the probability density associated with the largest eigenvalue**: the N*N Hamiltonian with N being an even number results with oscillations in the same order of magnitude for the probability density of the largest and smallest eigenvalues. On the contrary, with N being an odd number, the highest eigenvalue must be two times degenerated and the probability density associated with this eigenvalue is characterized with oscillations of a significantly smaller amplitude than the ones of the density associated with the smallest eigenvalue as we can see in Figure 6.

Problem 7

Band structures of one-dimensional graphene

(1)

In this exercise, **the tight binding approximation presented in the previous exercise will be used to study one-dimensional nanometer-wide stripes cut from the two-dimensional honeycomb lattice of graphene.** Indeed, as previous exercise demonstrates, numerical diagonalization provides easy access to modeling the electronic properties of systems composed of a large number of atoms using the tight-binding approximation.

Graphene nanoribbons (GNRs, also called nano-graphene ribbons or nano-graphite ribbons) are strips of graphene with a width less than 50 nm. The one-dimensional nanometer-wide stripes systems are periodic along the x-direction, **with the periodicity fined by the lattice constant a (approximately 0.4 nm)**. Figure 7 illustrates the physical properties of the system we are studying.

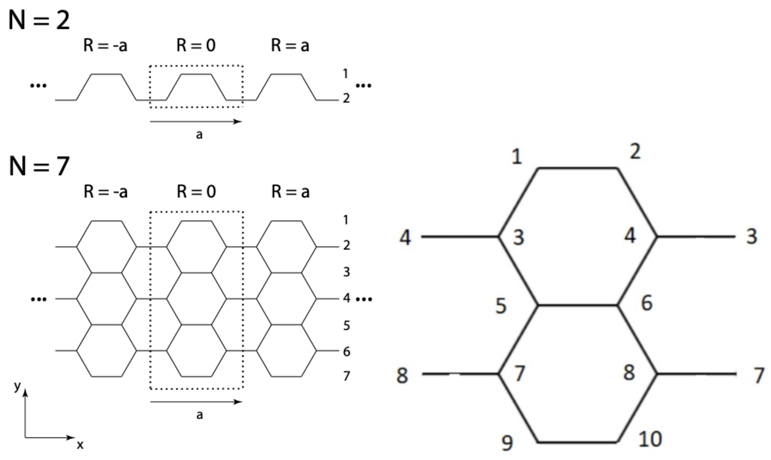


Figure 7: **Left** : Two examples of the atomic structures of graphene nanoribbons described by N = 2 and N = 7. The nanoribbons are periodic along the x axis. The dotted-line rectangles denote the unit cell.
Right : Indexes of Graphene Atoms for the Matrix creation

The rectangles denote unit cells (repeat units). Their width is defined by parameter N. As these **systems are periodic in one dimension, the crystalline momentum k in the first Brillouin zone (k $\in [\pi/a, \pi/a]$) needs to be considered when describing their electronic properties.** In fact, in opposition to the previous exercice, k dependence becomes relevant and it is necessary to consider the discretized energy (discrete in N) as continuous functions of k. This ambivalence (energies being both discrete in N and continuous in k) gives rise to what is called the band structure of the system. The matrix elements of the Hamiltonian matrix H_k for a given k can be written as:

$$H_k(i,j) = \gamma \sum_{j \in n.n.of.i} e^{ikR}$$

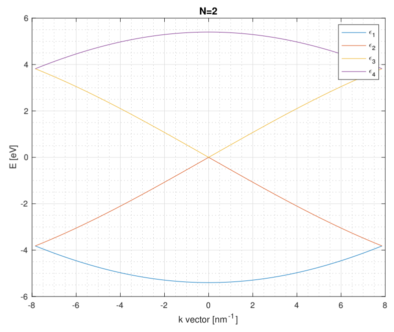
This notation means that only atoms j that are connected to atoms i contribute to the matrix elements. If both atoms are located within the same unit cell, then R = 0. Connections with atoms belonging to the right or the left periodic replicas correspond to $R = a$ and $R = -a$, respectively. Below, assume the nearest-neighbor hopping integral $\gamma = 2.7eV$.

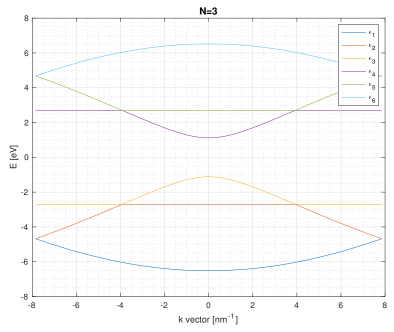
Thus, we use this expression to define the Hamiltonian matrix and solve the system. **The configuration of atoms (numbering)** presented in Figure 7 is used for the index of this Hamiltonian. The part of the code needed to define the Hamiltonian can be shown in Listing 8.

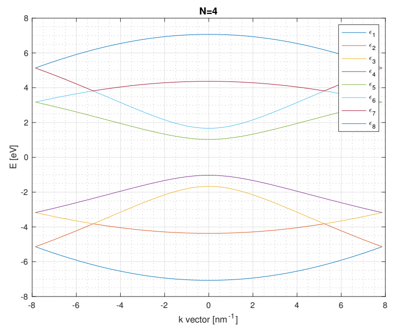
Listing 8: Hamiltonian Matrix creation

```
% Creation of the Hamiltonian Matrix called H
for k=-pi/a:pi/a/nb:pi/a
    % We first initialized the matrix position corresponding to non
    % null value for the hamiltonian
    for i=1:4:2*N
        i1=i;
        H(i1,i1+1)=1;                   % left bound
        H(i1,mod(i1-2-1,2*N)+1)=1;     % lower bound
        H(i1,mod(i1+2-1,2*N)+1)=1;     % upper bound
        i2=i+1;
        H(i2,i2-1)=1;                   % right bound
        H(i2,mod(i2+2-1,2*N)+1)=1;     % upper bound
        H(i2,mod(i2-2-1,2*N)+1)=1;     % lower bound
        i3=i+2;
        H(i3,i3+1)=exp(-1i*k*a);       % left bound
        H(i3,mod(i3+2-1,2*N)+1)=1;     % upper bound
        H(i3,mod(i3-2-1,2*N)+1)=1;     % lower bound
        i4=i+3;
        H(i4,i4-1)=exp(1i*k*a);      % right bound
        H(i4,mod(i4+2-1,2*N)+1)=1;     % upper bound
        H(i4,mod(i4-2-1,2*N)+1)=1;     % lower bound
    end
    % Increment the counter
    l=l+1;
    % Put some specific values to 0 in the Hamiltonian
    if N~=2
        H(2*N-1,1)=0;
        H(2*N,2)=0;
        H(1,2*N-1)=0;
        H(2,2*N)=0;
    end
    % Multiply the values corresponding to the definition with gamma
    Hfin=gamma*H(1:2*N,1:2*N);
```

Then, when the Hamiltonian is constructed, we follow the same procedure as the one done in the previous exercise. In fact, we **plot the band structures**, which is the dependences of eigenvalues $\epsilon_i(k)$ on momentum k for $k \in [\pi/a, \pi/a]$ for several values of N (N = 2...7). For each value of N, a separate plot was made. In each of these plots, all 2N eigenvalues $\epsilon_i(k)$ (bands) are plotted as lines. To succeed this task, **smart usage of for loop and the Matlabs diagonalization routine *eig* were used**.







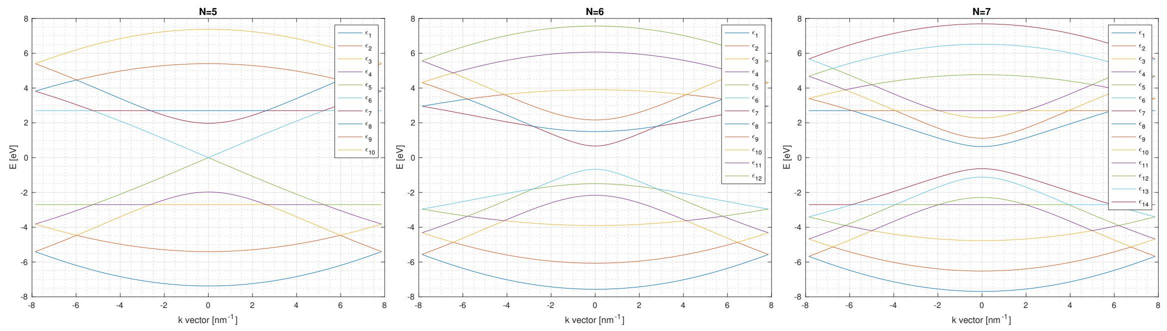


Figure 8: Band structure for different N, which means for different graphene nanoribbon configuration

Results displaying the band structure can be seen in Figure 8. The first thing that can be noticed is **the symmetry with the axis x = 0** because the eigenvalues of the transpose of a matrix are the same one of the original matrices. In addition, another axe of **symmetry can be noticed : y = 0**. These band structures are essential will help us to understand and point out the properties of materials. In fact, thanks to them we are able to learn a lot about the electronic structure. **This information will allow us to predict if a given material will or will not conduct electricity which means being a metal or an insulator.** This reasoning will be completed more precisely in the following part.

(2)

In our particular system, N lowest bands are populated by electrons, while the remaining N bands remain unpopulated. The energy difference between the highest occupied and lowest unoccupied states is called the band gap E_g, that is

$$E_g = min_{k,i=N+1,..,2N}\epsilon_i(k)max_{k,i=1,..,N}\epsilon_i(k)$$

This expression assumes the eigenvalues are sorted in ascending order. That's why during the process, a sorting process needs to occur. **We plot the magnitude of the band gap $E_g(N)$ for $N = 1...200$ in order to succeed to classify our system as metals or insulators.**

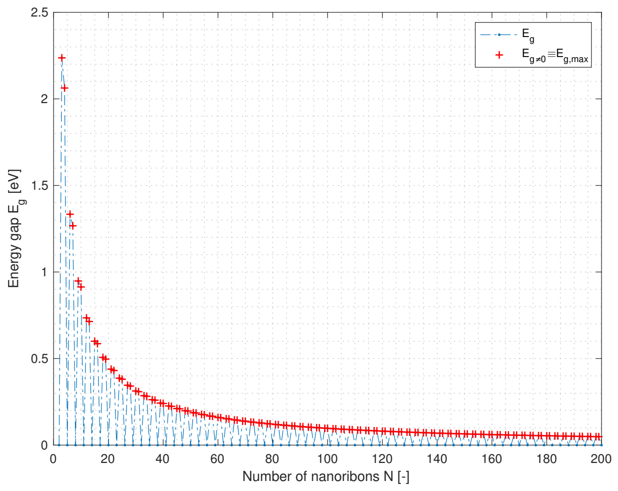


Figure 9: Magnitude of the band gap as a function of N

By definition, zero values of E_g corresponds to metals, while positive E_g corresponds to insulators (semiconductors). Indeed, this band gap can be linked to the property of conductivity by studying the number of valence electron per atom and the band structure. **As a matter of fact, if a band is filled with electrons, where filled means that the band does not allow any movement of the electrons, thus the material will be an insulator.** However, if the **bands overlap each other, it is possible to allocate the electrons differently** and therefore allow the possibility to have current flowing through the material.

Overlaps are visible in the majority of the band structure plots. But, remember that in our case, N lowest bands are populated by electrons, while the remaining N bands remain unpopulated. It means that even if some bands may overlap some others,**to be a metal, the configurations need to show a general overlap of all the bands. Indeed, if the overlap is not total, the low energy states are filled with no possibilities for an electron belonging to these bands to jump to higher energetic ones. This energy needed to jump is the previously defined energy gap.** That's why an insulator is defined by a huge band gap while a semiconductor is defined by a band gap in the order of some electrons-volts. Thus, we can use the several plots of Figure 8 to see in which N case an overlap is visible. **There is no overlap when the band structure is not linked in our plot, meaning that a hole is situated near the origin of the plot**: by knowing that we can conclude that for N = 3, 4, 6 and 7 there is no overlap while overlaps are observable in graph N=2, 5. Thus N = 3,4,6,7 can be considered as semiconductors while N = 2,5 can be considered as metal.

This is indeed verified by our experimental results where we plot the band gap for several values of N. Several N correspond to gap band of 0 meaning that these configurations can be considered as metal as they have good conducting properties while other configurations (the one with gap band higher than 0) are semiconductors. **In the graph above, all semiconductors are shown with a red cross while metals are situated near the** $E_g = 0$.

Problem 8

Jacobi method

(1)

One of the major drawbacks of the symmetric QR algorithm is that it is not parallelizable. Each orthogonal similarity transformation that is needed to reduce the original matrix A to diagonal form is dependent upon the previous one. In view of the evolution of parallel architectures, it is, therefore, worthwhile to consider whether there are alternative approaches to reducing an N*N symmetric matrix A to diagonal form that can exploit these architectures. Jacobi method can be perfectly parallelized and have thus a huge advantage.

Jacobi s method is an easily understood algorithm for finding all eigenpairs for a symmetric matrix. In fact, in numerical linear algebra, the Jacobi eigenvalue algorithm is an iterative method for the calculation of the eigenvalues and eigenvectors of a real symmetric matrix (a process known as diagonalization). It is a reliable method that produces uniformly accurate answers for the results. For matrices of order up to 10*10, the algorithm is competitive with more sophisticated ones. A solution is guaranteed for all real symmetric matrices when Jacobis method is used. This limitation is not severe since many practical problems of applied mathematics and engineering involve symmetric matrices. From a theoretical viewpoint, the method embodies techniques that are found in more sophisticated algorithms.

The classical Jacobi algorithm proceeds as follows: find indices p and q, $p \neq q$, such that $|a_{pq}|$ is maximized. Then, we use a single Jacobi rotation to zero a_{pq}, and then repeat this process until off(A) is sufficiently small. The main drawback of this method is the complexity. In fact, the classical Jacobi algorithm converges quadratically as a function of sweeps, where 1 sweep = n (n*n matrix size) Jacobi rotations. Finding p and q is actually a very expensive step O(n^2) operations. **To win some efficiency for our Jacobi method, we avoid matrix multiplications in favor of vector operations**. This classical method was implemented in the file name *eig_j.m*. This module was also tested with the script *test_j.m* to prove its reliability. Indeed all the tests were passed meaning that we can trust in our code implementation for this Jacobi classic method.

Listing 9: Jacobi Method

```
function [val, count] = eig_j(input_matrix)
    % eig_j computes the eigenvalues of a matrix.
    % The original Jacobi's method of iteration is used
    % to compute the eigenpairs of a symmetric matrix.
    % ARGS :
    %   - input matrix (2D real symmetric matrix) : matrix
    %   for the eigen value problem.
    % RETURNS :
    %   -val : an array with eigenvalues.

    % Initialization of all the needed variables
    epsilon = 10^(-10);
    D = input_matrix;
    [n,n] = size(input_matrix);
    V = eye(n);

    % Select element element of largest magnitude.
    % Use vector representation to be effective.
    [m1,p] = max(abs(D-diag(diag(D))));
    [m2,q] = max(m1);
    p = p(q);
    count = 0;

    if m2 == 0
        val = diag(D);
        return
    end

    % Main part of the module using epsilon in the while condition
    while (off(D) > epsilon)
      count = count + 1;
      if D(p,q) ~= 0
          t = D(p,q)/(D(q,q) - D(p,p));
          c = 1/sqrt(t*t+1);
          s = c*t;
      else
          c=1;
          s=0;
      end
      R = [c s; -s c];
      D([p q],:) = R'*D([p q],:);
      D(:,[p q]) = D(:,[p q])*R;
      V(:,[p q]) = V(:,[p q])*R; % V is the matrix of eigenvectors
      [m1,p] = max(abs(D-diag(diag(D))));
      [m2,q] = max(m1);
      p = p(q);
    end

    D = diag(diag(D)); % D is the solution: diagonal matrix of eigenvalues.
    val = diag(D);
end
```

(2)

The classical Jacobi algorithm is impractical because it requires O(n^2) comparisons to find the largest off-diagonal element. A variation, called the cyclic-by-row algorithm, avoids this expense by simply cycling through the rows of A, in order. This cyclic method was implemented in the file name *eig_j_cyclic.m*. This module was also tested with a similar script like the one for the classic method named *test_j_cyclic.m* to prove its reliability. Indeed all the tests were passed meaning that we can trust in our code implementation for this Jacobi cyclic method.

Listing 10: Cyclic Jacobi Method

```
function [val,count] = eig_cj(input_matrix)
    % eig_j computes the eigenvalues of a matrix.
    % The cyclic Jacobi's method of iteration is used
    % to compute the eigenpairs of a symmetric matrix.
    % ARGS :
    %   - input matrix (2D real symmetric matrix) : matrix
    %   for the eigen value problem.
    % RETURNS :
    %   -val : an array with eigenvalues.
    %   -count  Number of Jacobi rotation.

    % Initialization
    epsilon = 10^(-10);
    count = 0;
    D = input_matrix;
    [n,n] = size(input_matrix);
    V = eye(n);

    % Select element element of largest magnitude.
    % Use vector representation to be effective.
    [m1,p] = max(abs(D-diag(diag(D))));
    [m2,q] = max(m1);
    p = p(q);

    % Take care of a specific case
    if m2 == 0
        val = diag(D);
        return
    end

    % Main part of the module using epsilone in the while condition
    while(off(D) > epsilon)
        for p = 1:(n-1)
          for q = (p+1):n
              count = count+1;
              if D(p,q) ~= 0
                 t = D(p,q)/(D(q,q) - D(p,p));
                 c = 1/sqrt(t*t+1);
                 s = c*t;
              else
                 c=1;
                 s=0;
              end
              R = [c s; -s c];
              D([p q],:) = R'*D([p q],:);
              D(:,[p q]) = D(:,[p q])*R;
              V(:,[p q]) = V(:,[p q])*R;
          end
        end
    end
    D = diag(diag(D));
    val = diag(D);
end
```

A comparison of performance was done between the two methods. To do that we studied 2 particular quantities : computation time and the number of Jacobi rotations required for performing diagonalization for a broad range of matrix sizes. This comparison is shown in Figure 10.

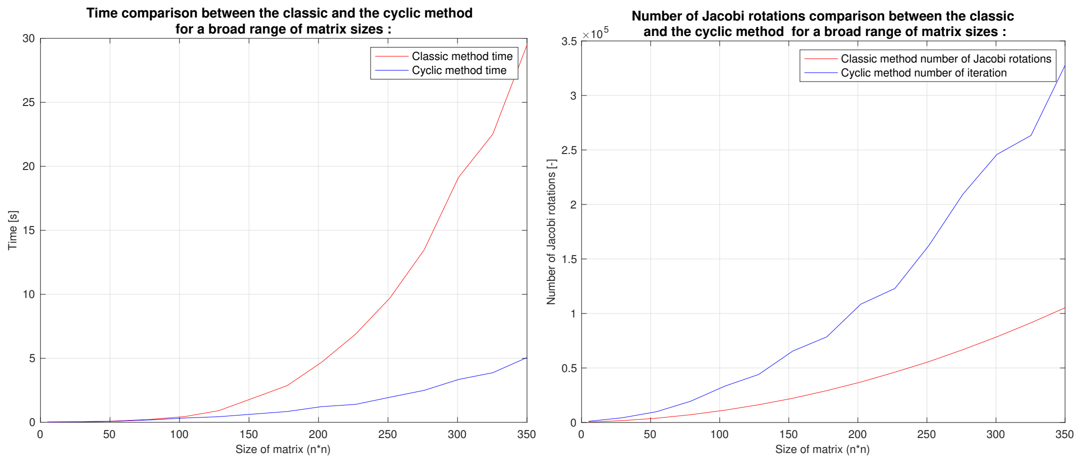


Figure 10: **Comparison between Classic and Cyclic Jacobi method**:
Left : Time comparison.
Right : Number of Jacobi rotations comparison.

Considering the time (curve on the left part of Figure 10), **we conclude as expected that the cyclic method takes much less time than the classic method.** As we explained above, this is mainly due to the search for p and q. Following this reasoning, we also conclude that **the number of Jacobi rotation is higher for the cyclic method compared to the classical method.** It makes sense as the cyclic-by-row algorithm, avoids the O(n^2) comparisons used to find the largest off-diagonal element, by simply cycling through the rows of A, in order. **As it cycles over the rows, the number of Jacobi rotation increase but the impact about the time of these decrease. In fact, due to the huge complexity of finding the largest element of a big matrix, this operation takes a lot of time and thus at the end, the cyclic method takes less time than the classical one.** We prove all this fact on a matrix having sizes in the range of 30*30 to 350*350.

As a conclusion, we can say that the cyclic method induces much more iterations because of the two added for loops but the time efficiency is at the same time strongly improved. In the end, every scientist is interested in the time taken to finish a computation and not the number of iteration.

Problem 9

Two-dimensional quantum well

In solid-state systems, **potential profiles can be tailored by means of varying chemical composition in order to spatially localize electrons.** Consider a 2D quantum well. Electrons in such a system are subjected to the following potential profile:

$$V(x,y) = \begin{cases} V_0 < 0, & x^2 + \frac{y^2}{c^2} < r^2 \\ 0, & \text{otherwise} \end{cases}$$

where $V_0 = 1.5eV$ is the width of the potential and r = 1 nm. **The parameter c = 1 corresponds to the circular shape quantum well, while $c > 1$ makes it elliptic.** We will see the effect on c in few lines.

One of the difficult parts of this problem is the creation of the Hamiltonian taking into account the boundary conditions. In fact, the Hamiltonian operator by discretizing space in 2D is defined as:

$$H = -\frac{\hbar^2}{2m_{\text{el}}}\Delta + V(x)$$

where m_{el} is the free electron mass and h is the reduced Plancks constant. As we are in 2D, we use a grid of N N points in a square region defined by the range parameter a. In addition we assume N = 100 and a = 5r. The matrix creation process was done thanks to the code shown in the Listing 11 below :

Listing 11: Code allowing the creation of the Hamiltonian of the problem taking into account the boundary condition

```
% Define the parameters that characterized the problem
c=1;
N=100;
R=1e-9;
a=5*R;
dx=a/(N+1);
e=1.60217656535e-19;
mel=9.1093829e-31;
hbar= 1.05457172647e-34;
V0=-1.5*e;

H= gallery('tridiag',N^2,1,-4,1);
psi=zeros(size(H));
for i=1+N:N^2
    H(i,i-N)=1;
    H(i-N,i)=1;
end

for i=2:N
    H(1+N*(i-1), N*(i-1)) = 0;
    H( N*(i-1), 1+N*(i-1)) = 0;
end

H=-hbar^2/(2*mel*dx^2)*H;

for i=1:N
    for j=1:N
```

```
        if (i*dx-a/2)^2+((j*dx-a/2)/c)^2<R^2
            H(N*(i-1)+j,N*(i-1)+j)=H(N*(i-1)+j,N*(i-1)+j)+V0;
        end
    end
end
```

(1)

Consider the case of c = 1, which correspond to the potential in a perfect circle. We want to find the number of bound solutions to this problem, which means the number of negative eigenvalues. To do that, we will use the Matlabs diagonalization routine *eigs* specially designed to deal with sparse matrices. **We find out that 10 bound solutions are present in our problem.** More specifically, we look at the three lowest energy states in more details. They are characterized by the following eigenvalues ϵ_i (in eV):

$$\epsilon_1 = -1.3371eV; \epsilon_2 = -1.0900eV; \epsilon_3 = -1.0900eV$$

Note that the last two states seem to be degenerated but the $\epsilon_2 \neq \epsilon_3$ if we look closely enough at the decimals. Then, the wavefunctions ψ_i as well as the probability densities $\psi_i * \psi_i$ were visualized. To do that, Matlabs function pcolor was used. Results can be found in Figure 11 and 12. **The wave function itself do not have a physical meaning but the modulus of the wave function describes the probability to find the electron at a given position if a measurement is performed.**

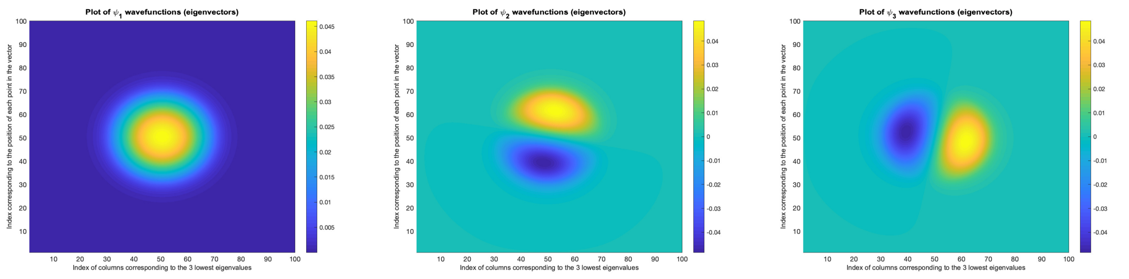


Figure 11: Wavefunction ψ_i of the 3 smallest eigenvalues of the Hamiltonian.

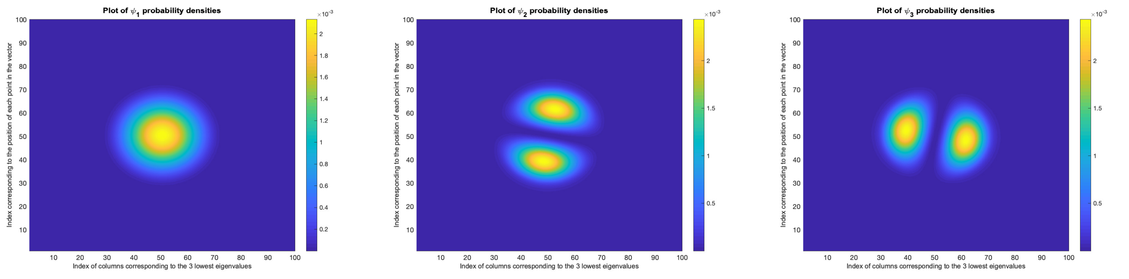


Figure 12: Probability densities ψ_i of the 3 smallest eigenvalues of the Hamiltonian.

(2)

Then, **we investigate a specific scattering solution. We choose the wavefunction corresponding to positive energy of approximately 1 eV.** Again, the wavefunction and the probability density are presented in figure 13.

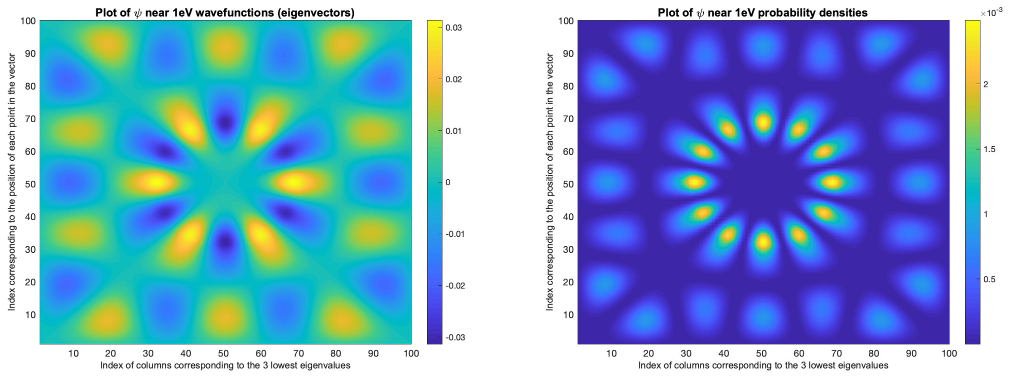


Figure 13: Wavefunction and probability density ψ_{54} of the eigenvalues near to 1 of the Hamiltonian.
Left : Wavefunction
Right : Probability density

Then, **the probability of finding particle inside the quantum well was calculated for this state and the wavefunction obtained for three bound solutions** :

$$\|\psi_1\| = 0.987075; \|\psi_2\| = 0.995487; \|\psi_3\| = 0.995487; \|\psi_{54}\| = 69.0083;$$

All these values make sense with the graph presented previously. Moreover, it makes sense to have **an extremely high probability to be in the smallest energy level and to have a lowest probability to be in a more excited state.** This is the case for us, as the $\|\psi_1\|$, $\|\psi_2\|$, and $\|\psi_3\|$ corresponding to the 3 lowest level of energy have a value near 100 percent while the probability to be in a more exciting state which correspond to $\|\psi_{54}\|$ is lower (approximately 69 percent).

(3)

Now, we consider the case of $c > 1$ which means that the shape of the quantum well becomes elliptic. We choose to make this c value varying and thus we are able to plot the number of bound states as a function of c in Figure 14 :

Problem 9 [(3)] continued on next page. . .

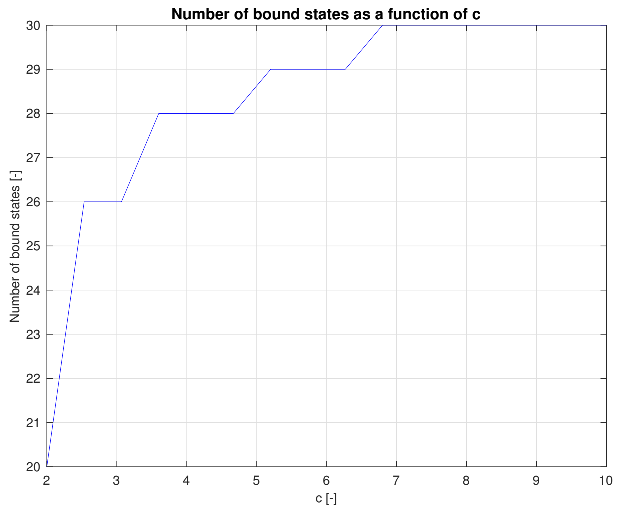


Figure 14: Number of bound states as a function of c.

As you can see, the number of bound states of the Hamiltonian increase as c increase. In addition, around c = 7, the number of bound states reach a plateau. **Remember that the constant c measures the ellipticity of the region inside which the potential is non-zero. Thus all this process makes sense as increasing c is equivalent to increase the width of the quantum well.** This process increases the size of the area covered by the potential and when the quantum well has the same size or is bigger than our NxN grid, the max number of states in the system is reached and thus the number of bound states cannot increase more. **In fact, if it touches the border of the system, no more bound state can be formed because of the finite size of this system.**

(4)

Our goal is now to understand the effect of an increase of c on the wavefunctions and the probability densities. To be able to compare the wavefunctions (ψ_1 denotes the eigenvector associated to the lowest energy,...) and the eigenvalues corresponding to the three lowest energy states for c = 1 (Figure 11 and 12) and some for $c > 1$ (Figure 15 and 16), plot of them are displayed just below.

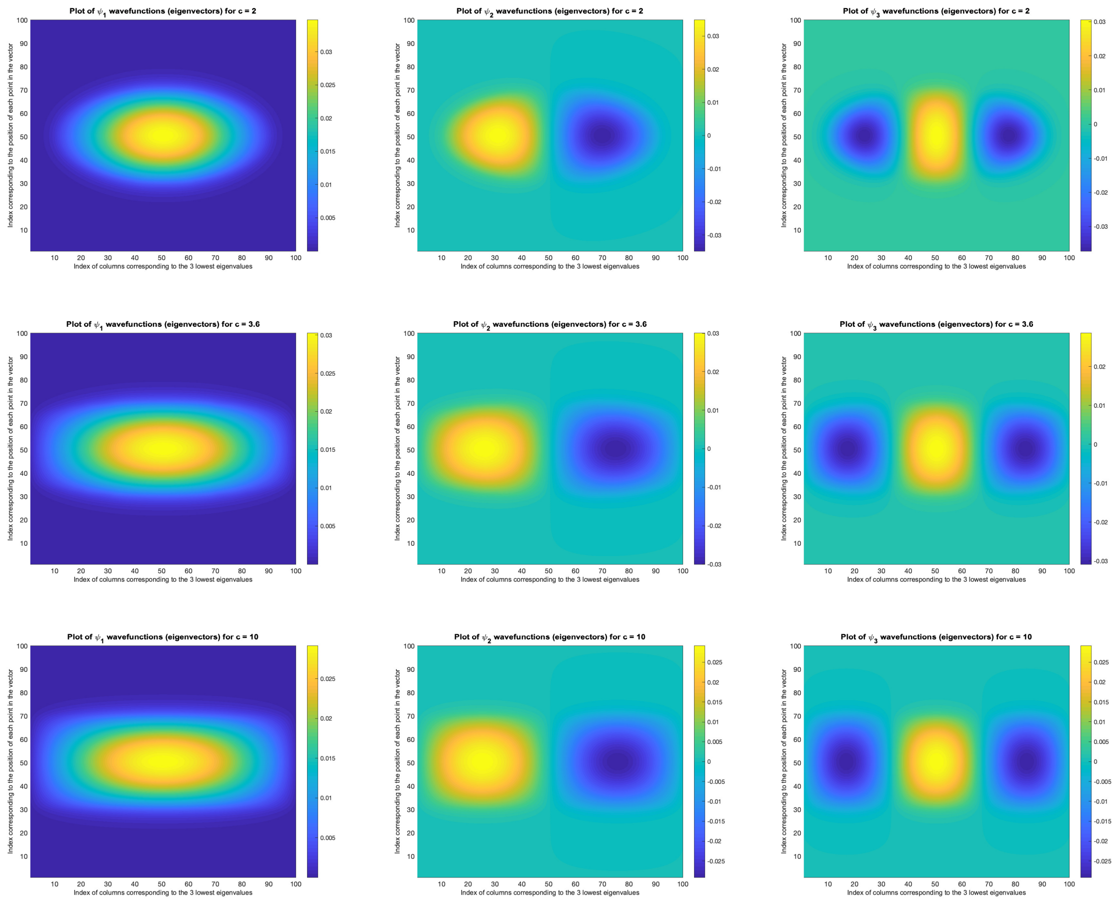


Figure 15: Wavefuntions ψ_i of the 3 smallest eigenvalues of the Hamiltonian for c = 2, 3.6 or 10

Problem 9 continued on next page. . .

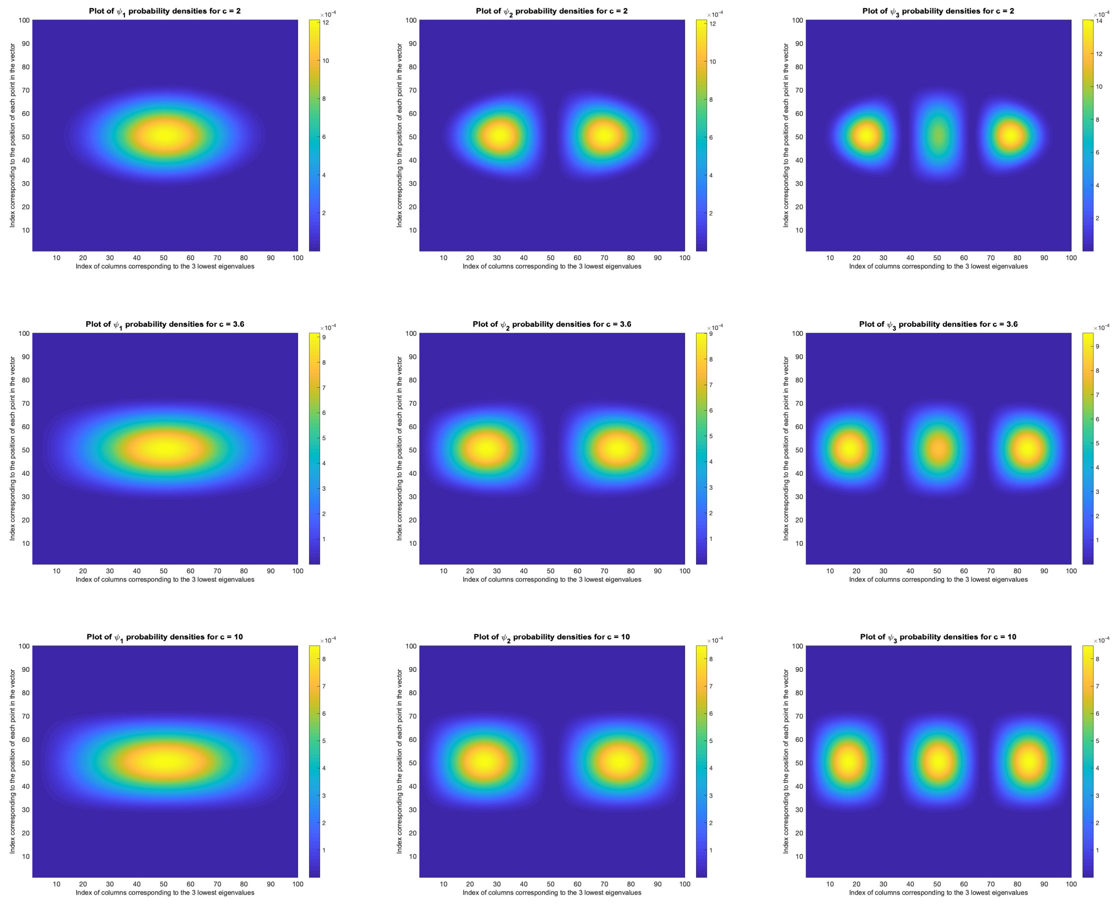


Figure 16: Probability density ψ_i of the 3 smallest eigenvalues of the Hamiltonian for c = 2, 3.6 or 10

As expected thanks to the analysis of the previous question, **an increase of c is equivalent to increase the width of the quantum well.** In fact, as you can see on the Figures, increasing c allows an increase of the size of the area covered by the potential and thus also an increase of the area corresponding to a higher probability to be in this well. In fact, the ellipticity is clearly observed especially for ψ_1.

This effect is the change of the preferential regions of space. This change is easily visible for the third lowest energy state. Indeed, this was not observed in the case c = 1 where the most regions observed were just 2. **This process means that a new preferred area is created with the increasing of c.**