Computational Physics III: Report 2
Linear systems solving and diagonalization methods

Due on April 30, 2020

April 27, 2020

Raffaele Ancarola

Contents

Introduction

Solving a system of linear equations

A *linear problem* could be defined as a system of which the describing equations are all linear. Furthermore, a linear system is said to be *determined* if the number of equations N is finite and it corresponds to the number of the unknowns. Such a system defined on a field $\mathbb{K}$ takes the advantage to be written in a matrix form:

$$A \cdot \vec{x} = \vec{b} \tag{1}$$

where A is the describing matrix and $\vec{b} \in \mathbb{K}^N$ the affine component of the system, or the components which are independent with respect to the unknows contained in $\vec{x}$. Because the system is *determined*, the condition that A must satisfy is the *inversibility*, then $A \in \mathcal{GL}(N)$ and a solving the system means to find $\vec{x} \in \mathbb{K}^N$ such that eq. (1) is satisfied. There exist various approaches that can reach this aptempt, in this report three cases will be analysed: the *gauss elimination*, the *LU decomposition* and the *diagonalisation*.

Gauss elimination algorithm

The *Gauss elimination* bases to the fact that any square matrix can be decomposed into a finite sequence of elementary operations $\{P_k\}_{1 \leq k \leq M}, M \in \mathbb{N}^*$. There are basically three kinds of them:

- Multipling of a row by a scalar factor $\lambda \in \mathbb{K}$
- Switching a row with another
- Adding a row with a multiple of another

The purpose of this method is to reduce the involved matrix A into the identity applying the same operations to the vector $\vec{b}$, as shown in the equation (2).

$$A = P_1 \cdot ... \cdot P_M \implies \vec{x} = P_M^{-1} \cdot ... \cdot P_1^{-1} \cdot \vec{b}\,, \quad M \in \mathbb{N}^* \tag{2}$$

LU decomposition

The *LU decomposition* is not a direct method which solves a linear system, but it allows to simplify the resolution by decomposing the A matrix into a lower-triangular matrix L and an upper-triangular matrix U. The simplification is due to the major facility to invert the two matrices precedently presented. Once A is decomposed, the process is straigh-forward:

$$A \cdot \vec{x} = L \cdot U \cdot \vec{x} = \vec{b}$$

$$L \cdot \vec{y} = \vec{b} \tag{3}$$

$$U \cdot \vec{x} = \vec{y} \tag{4}$$

Both equations (3) and (4) can be solved sequentially using the *Gauss elimination* method.

Diagonalization: introduction

In case A is a symmetric matrix, the spectral theorem [] states that such a matrix is equivalent (definition of equivalence here: []) to a diagonal matrix D, where the transition matrix P is unitary ($P^{-1} = \bar{P}^T$), then:

$$A = P \cdot D \cdot \bar{P}^T \implies \vec{x} = P \cdot D^{-1} \cdot \bar{P}^T \cdot \vec{b} \tag{5}$$

Generally diagonalization is not used to solve general systems of linear equations, but it's convenient when the problem is related to find the eigen-base related to the eigen-values.

Problem 1

(1) LU decomposition implementation

This algorithm separes the input matrix A into a lower triangular L and an upper triangular U, garanteeing that $A = L \cdot U$. Neverthless, not all the invertible square matrices are purely LU decomposable, then it may happen that the output can result ill formed. The code (1) shows at line 23 that a division by the diagonal values is performed, causing eventually a singularity. A possible work-around is to apply the partial pivoting technique in order to swap the problematic lines. In listing (1) is shown a full implementation with partial pivoting.

Listing 1: *LU decomposition* implementation with partial pivoting

```
1  function [L, U, P] = lu_decomposition(A)
2      [Ni, Nj] = size(A);
3      assert(Ni == Nj, "The input must be diagonal");
4
5      N = Nj;
6      assert(N > 0, "The input must non empty");
7
8      L = eye(N); % if zeros doesn't give the same result
9      U = A; % if zeros doesn't give the same result
10     P = eye(N); % identity matrix
11
12     for k=1:(N-1)
13         % pivoting section
14         [Amax,r] = max(abs(U(k:N, k)));
15         r = r + k - 1;
16         % swap rows if it's not the identity swap operation
17         U([k r],:) = U([r k],:);
18         P([k r],:) = P([r k],:);
19         L([k r], 1:k-1) = L([r k], 1:k-1);
20
21         % computing LU
22         for i=(k+1):N
23             L(i,k) = U(i,k) / U(k,k);
24             U(i,:) = U(i,:) - L(i,k) * U(k,:);
25         end
26     end
27 end
```

Partial pivoting

The *LU decomposition* algorithm (presented below in exercise 1.1) can easily run into singularities, especially when A presents zeros as diagonal terms. In order to avoid divergent results, it would better select the rows of which element is not zero in the requested columns and swap them with the current one. More precisely, at the k-th step, select the r-th row such that $A_{rk} = \max_{k \le i \le N} |A_{ik}|$, then swap rows at the position k and r. If the pivoting is applied the resulting *LU decomposition* won't be anymore like it was defined in the previous section, but a correction to equation (3) must be applied:

$$P \cdot A = L \cdot U \implies L \cdot \vec{y} = P \cdot \vec{b} \tag{6}$$

where P is the orthogonal matrix that accumulated all row switching applications. The rest of the solving method remains unchanged.

(1.1) Solving a linear system

A linear system can be solved applying the LU decomposition and then a gauss elimination process, as shown in the equations (6) and (4).

For example, the system in equation (7) is determined and can be solved using the `solve.m` script. Addictionally the `test_solve.m` script compares with the matlab `x = A \ b` verifying that the solution $\vec{x}$ is given correctly by the `solve.m` script.

$$\begin{cases} 2x_1 + x_2 - x_3 + 5x_4 &= 13 \\ x_1 + 2x_2 + 3x_3 - x_4 &= 37 \\ x_1 + x_3 + 6x_4 &= 30 \\ x_1 + 3x_2 - x_3 + 5x_4 &= 19 \end{cases} \implies A = \begin{pmatrix} 2 & 1 & -1 & 5 \\ 1 & 2 & 3 & -1 \\ 1 & 0 & 1 & 6 \\ 1 & 3 & -1 & 5 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 13 \\ 37 \\ 30 \\ 19 \end{pmatrix} \implies \vec{x} = A^{-1} \cdot \vec{b} = \begin{pmatrix} 2 \\ 4 \\ 10 \\ 3 \end{pmatrix} \tag{7}$$

(1.3) Decomposition of a matrix

The example taken in equation (8) is a problematic case where a pure LU decomposition doesn't exist. A necessary and sufficient condition to the existance of a pure LU decomposition is that the matrix must be gauss reductible without any row exchange (ref. [**?**]), that's why if such a decomposition exists, then pivoting matrix P is the identity matrix. So, the form $P \cdot A = L \cdot U$ is obtainable using the pivoting described in the previous section.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 9 \\ 4 & -3 & 1 \end{pmatrix} \implies L = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0.5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 4 & -3 & 1 \\ 0 & 5.5 & 8.5 \\ 0 & 0 & -1.5 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \tag{8}$$

(2) Puzzle board game

The puzzle game problem is described by using the following formalization:

$$b_k = x_k + x_{k-N} + x_{k+N} + x_{k-1} + x_{k+1}, \quad A = T \otimes 1 + 1 \otimes T - 1 \otimes 1 \tag{9}$$

where b_k corresponds to the number of times that the k_{ij} is pressed. Using the `kron` matlab function it's possible to easy construct the matrix A, as shown in the code listing (2).

Listing 2: Construction of the matrix describing the puzzle game problem

```
function A = puzzleA()
    T = diag(ones(N-1,1), -1) + diag(ones(N,1), 0) + diag(ones(N-1,1), 1);
    I = eye(N);
    A = kron(T, I) + kron(I, T) - kron(I, I);
end
```

The index k describes a remapping of a $N \times N$ grid into a N^2 vector, precisely $K_{ij} = (i-1) \cdot N + j$, $1 \leq i, j \leq N$. One the A matrix is composed and given the $\vec{b}$ vector, then the solution is simply given by `x = A \ b`. The code is shown in the attached script `puzzle.m`.

(3) Helicopter power formula: dimensional analysis

The elicopter problem is a dimension problem, because knowing that the involved quantities are P, g, L, ρ_h and ρ_a, their relation will only depend on the units of measure expression. Thus, if the second helicopter has 1/3 of the length with respect the first one, then, taking the formula in the document [**?**], its power is given by $P_2 = 3^{-\beta} P_1$.

(3.1) Approaching the problem

The same formula cited above can be expressed in a logarithmic form:

$$\ln(P) = \alpha \cdot \ln(g) + \beta \cdot \ln(L) + \gamma \cdot \ln(\rho_h) + \delta \cdot \ln(\rho_a) \tag{10}$$

Assigning for each quantity its corresponding SI unit of measure [1], or rather, $[P] = \mathrm{kg\,m^2/s^3}$, $[g] = \mathrm{m/s^2}$, $[L] = \mathrm{m}$, $[\rho] = \mathrm{kg/m^3}$, then the logarithm of m, s and kg can be treated as a vector basis. At this point the equation (10) can be rewritten as:

$$\ln(\mathrm{kg}) \cdot (1 - \gamma - \delta) + \ln(\mathrm{m}) \cdot (2 - \alpha - \beta + 3\gamma + 3\delta) + \ln(\mathrm{s}) \cdot (-3 + 2\alpha) = 0 \tag{11}$$

$$\implies \begin{cases} \gamma + \delta = & 1 \\ \alpha + \beta - 3\gamma - 3\delta = & 2 \\ 2\alpha = & 3 \end{cases} \tag{12}$$

This system is indetermined, thus it cannot be computationally solved using the `solve.m` script, because it's matrix representation is not a square matrix.

(3.2) + (3.3), Adding a constraint

In the case where $\alpha = \gamma$, the equation found in the previous point reduces to a determined system of linear equations, which has a square matrix form A.

$$\begin{cases} \alpha + \delta & = 1 \\ 2\alpha - \beta + 3\delta & = -2 \\ 2\alpha & = 3 \end{cases} \implies A = \begin{pmatrix} 1 & 0 & 1 \\ 2 & -1 & 3 \\ 2 & 0 & 0 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} \tag{13}$$

Now the system is solveable and the solution is straight forward:

$$\alpha = \gamma = \frac{3}{2}, \qquad \beta = \frac{7}{2}, \qquad \delta = -\frac{1}{2} \tag{14}$$

So, the output power of the second helicopter $P_2 = 3^{-\frac{7}{2}} \cdot P_1 \approx 0.021 P_1$.

The eigenvalue problem and diagonalization

Let $\hat{A}$ be an operator defined over an hilbert space $\mathcal{H}$. By solving an eigenvalue problem is meant to find all vectors (or functions) $x \in \mathcal{H}$ such that there exists a real (or complex) value λ that satisfies the following condition:

$$\hat{A} \cdot x = \lambda \cdot x, \lambda \in \mathbb{K} \tag{15}$$

In the case of this report, the interest is to computationally solve the eigenvalue problem for finite rank operators, which can be expressed as square matrices. So, let N be rank of a square matrix A and $\vec{v} \in \mathbb{K}^N$, then the equation (15) is equivalent to:

$$A \cdot \vec{v} = \lambda \cdot \vec{v}, \lambda \in \mathbb{K} \tag{16}$$

Power method

The power method bases its functioning on the iterative application of a specific operation T. The principle is that every iteration step tends to minimise of the distance between the old evaluated eigen value λ_{k-1}

and the current λ_k. Given the unitary vector $\vec{v_k} \in \mathbb{K}^N | \|\vec{v_k}\| = 1$ at the iteration step k, the corresponding diagonal value, relative to a square matrix A, is given by the hermitian scalar product (see [3] for the notation):

$$\lambda_k = \langle \vec{v_k}, A\vec{v_k} \rangle, \quad \lambda_k \in \mathbb{K} \tag{17}$$

The operation T mentioned above variates depending on the specific method, which of there are three:

- *Power* method: $T = A$.
- *Inverse power* method: $T = (A - 1 \cdot \tau)^{-1}$, $\tau \in \mathbb{K}$ is a fixed eigenvalue target.
- *Rayleigh quotient* method: $T = (A - 1 \cdot \lambda_{k-1})^{-1}$, $\lambda_{k-1} \in \mathbb{K}$ is the old evaluated eigenvalue, as defined in equation (17).

Notice that for the *Tayleigh quotient* method the application is adapting during each iteration, garanteeing a faster convergence.

Jacobi method

Let A be a real symmetric matrix, by the spectral theorem [4] such a matrix is diagonalizable in a form $A = PDP^T$, where P is an orthogonal matrix and D a real diagonal matrix. The idea at the base is that any orthogonal matrix P can be decomposed into a series of *axis aligned* rotation matrices $\{J^{(pq)}(\theta)\}, p, q \in \mathbb{N}*, p < q$: an *axis aligned* rotation matrix $J^{(pq)}$ is defined as:

$$\begin{cases} J^{(pq)}_{pp} = J^{(pq)}_{qq} & = \cos(\theta) \\ J^{(pq)}_{qp} = -J^{(pq)}_{pq} & = \sin(\theta) \\ \forall i, j \neq p, q : & J^{(pq)}_{ij} = \delta_{ij} \end{cases} \tag{18}$$

A property of such a matrix is that the determinant is 1 for any value of θ and $J^{-1} = J^T$. The *Jacobi* method essentially applies a series of axis aligned rotations to the matrix A until the diagonal form is reached, adapting each step the value of θ. So, let M be the total number of steps necessary to reach the diagonalization, then for $0 \leq k \leq M$ and $A_0 = A$ the relation (19) shows by induction that the next step is realized by an axis aligned rotation of the matrix A_k and it's also possible at the end to obtain the transition matrix $P = P_M$.

$$D = J^T_{M-1} A_{M-1} J_{M-1} = P^T A P \qquad \Longrightarrow \qquad A_{k+1} = J^T_k A_k J_k, \qquad P_{k+1} = P_k J_k \tag{19}$$

Problem 2

(1) Power iteration methods implementation

As mentioned in the section (), the all methods involve a matrix application, which means that the temporal complexity is at least N^2, where N is the size of the involved matrix. Addictionally, the number of iterations is not simple to estimate and the worst case is potentially infinite. The same could be claimed for the tolerance that the minimized value $|\lambda_k - \lambda_{k-1}|$ should have. In fact the matlab constant `eps` is often too low as tolerance and for larger values of N the loop doesn't exit. The implementation codes can be found in the scripts `eig_power.m`, `eig_ipower.m` and `eig_rq.m`.

(2) Eigenmodes of a vibrating string

(2.1) Formalization of the problem

The description of a vibrating string follows the wave equation (20). Applying a then separation of variables, the problem reveals to be depend on an addictional variable $\lambda \in \mathbb{R}$.

$$\frac{\delta^2 u}{\delta x^2} = \frac{\kappa}{\rho}\frac{\delta^2 u}{\delta t^2} \quad u(0,t) = 0, \quad u(L,t) = 0 \quad u(x,y) = \omega(x)\cdot v(t) \quad \Longrightarrow \begin{cases} \omega''(x) + \lambda\omega(x) & = 0 \\ v''(t) + \frac{\lambda\rho}{\kappa} v(t) & = 0 \end{cases} \tag{20}$$

The sign of λ will now determine the form of the solution. Analysing first the case of $\lambda = 0$, the solution would be a linear equation $\omega(x) = Cx + B$. Unfortunately, applying the boundary conditions $\omega(0) = B = 0$ and $\omega(L) = CL + B = 0$ the implication is $B = 0$ and $C = 0$, thus $\omega(x) = 0$. The same proof is applicable to the case $\lambda < 0$: the solution would be $\omega(x) = B\exp(\gamma x) + C\exp(-\gamma x), \gamma = \sqrt{-\lambda}$, but $\omega(0) = B + C = 0$ and $\omega(L) = B\exp(\gamma L) + C\exp(-\gamma L) = 0$. These conditions imply that $C = -B$, $2B\sinh(\gamma L) = 0$, which means that $\gamma = 0$ or $B = 0$, in both cases the solution turns to be trivial.

It remains the case where $\lambda > 0$, here the solution takes a sinusoidal form:

$$\begin{cases} \omega(x) & = B_x \sin(\gamma x) + C_x \cos(\gamma x), \quad \gamma = \sqrt{\lambda} \\ v(t) & = B_t \sin(2\pi\nu t) + C_t \cos(2\pi\nu t), \quad \nu = \frac{\sqrt{\frac{\lambda\rho}{\kappa}}}{2\pi} \end{cases} \tag{21}$$

This solution is not ill formed because $\omega(0) = C_x = 0$, then by the second boundary condition $\omega(L) = B_x \sin(\gamma L) = 0$. Imposing $B_x \neq 0$, the resulting condition is $\sin(\gamma L) = 0 \iff \gamma_n L = \pi n, \quad n \in \mathbb{N}*$. In other words, this means that for each natural number n, there exists a λ_n given by:

$$\lambda_n = \left(\frac{\pi n}{L}\right)^2 \tag{22}$$

Thus, the implicit condition to obtain a non-trivial solution is given by equation (22). Notice, by its expression, that dimensionally λ is $1/\mathrm{m}^2$ in the SI unit system [1].

(2.2) Implementation

Using a finite difference discretization, or rather given a step Δx and setting $x_i = (i-1) * \Delta x$, the problem is expressed as follow:

$$\frac{-\omega_{i-1} + 2\omega_i - \omega_{i+1}}{\Delta x^2} = \lambda\omega_i \qquad \iff \qquad A\vec{\omega} = \lambda\vec{\omega}, \qquad \vec{\omega} = (\omega_1, ..., \omega_N) \tag{23}$$

A in equation (23) is the tridiagonal matrix containing -1 in the lower and the upper diagonal and 2 on the diagonal, all the terms divided by the squared step Δx^2. This matrix is symmetric, then by the spectral theorem [4], it's eigenvalues are real and for each pair of eigenvalues, the associated eigenspaces are orthogonal each other. Furthermore, the matrix is positive definite, thus all the eigenvalues are strictly positive. Notice that this setup automatically meets the boundary conditions, because the first and the last line of A already discards the boundary terms (setting them implicitly to zero).

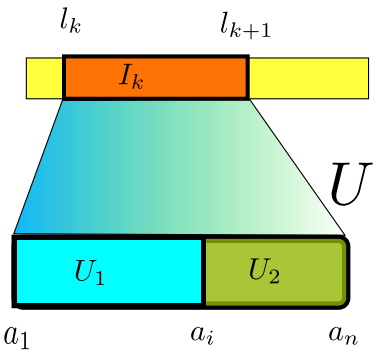
(2.3) + (2.4), Find the first eigenvalues

The first thing to notice is that the matrix A is not degenerate (by its positivity), then each eigenspace's dimension is 1. This clearly means that the first four eigenvalues are different each other. Theoretically, by the expression in equation (22), the eigenvalues for $L = 1$ are $\lambda_1 = \pi^2 \approx 9.8696$, $\lambda_2 = 4\pi^2 \approx 39.4784$, $\lambda_3 = 9\pi^2 \approx 88.8264$, $\lambda_4 = 16\pi^2 \approx 157.9137$ and $\lambda_5 = 25\pi^2 \approx 264.7401$. In order to find the first one, it's enough to use the *inverse power* method setting the target to 0.

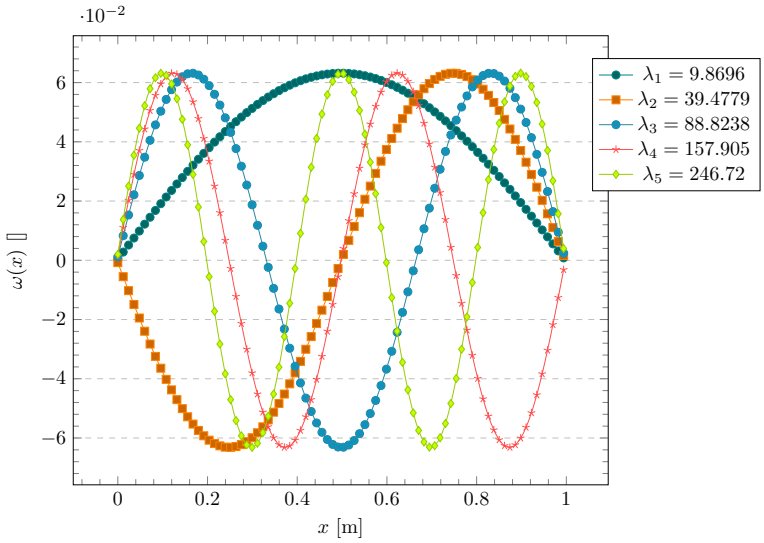
The other four can be found using an *interval bisection* strategy: let $l_k \in \mathbb{R}_+, k \in \mathbb{N}, l_0 = \lambda_1$, let l_{k+1} be the eigenvalue closest to $2l_k$ and consider the interval $I_k = [l_k, l_{k+1}]$. The strategy consists in finding

all eigenvalues contained in the interval I_k using a bisection algorithm and iterating with a new interval until the number of expected eigenvalues is reached. It may happen that $l_{k+1} = l_k$, then l_k is doubled until the evaluation of l_{k+1} mentioned above converges to a different and bigger value than l_k. Notice that l_k is always an eigenvalue, this characteristic is foundamental to speed up the eigenvalue research inside the interval I_k.

Consider now a real interval U such that the extrema are two eigenvalues a_1 and a non necessarely consecutive a_n, so the idea is to divide the interval into two disjoint sub-intervals $U_1 = [a_1, a_i]$ and $U_2 = [a_i, a_n]$ where a_i is the eigenvalue closest to $\frac{a_1+a_n}{2}$, evaluated with the *inverse power* method. If $a_i = a_1$ or $a_i = a_n$, then all the eigenvalues have been found inside the interval U, otherwise the algorithm recurses inside U_1 and U_2 until all eigenvalues are found.



(a) Scheme of the *interval bisection* strategy



(b) Eigenvectors corresponding to the first five eigenvalues

Now taking $U = I_k$ for each k iteration step, the eigenvalues are garanteed to be found in exponentially growing intervals. This approach increases the computation efficiency in most cases, but it can perform a lot of useless operations if the last requested eigenvalues are situated in an extremely large interval, inducing an averange case time complexity of $\mathcal{O}(2^{\log(n)}) = \mathcal{O}(n)$, where n is the number of requested eigenvalues. The figure (1a) graphically shows how the strategy is applied and the code listing (11) contains a matlab implementation.

The graph in figure (1b) shows the resulting eigenvectors with their correspective eigenvalues. Notice that they don't differ from the expected eigenvalues, which demonstrate that the *inverse power* method is enough reliable.

(3) Jacobi method implementation

As presented in previous section (Jacobi method), this algorithm diagonalises a symmetric matrix A applying a series of rotations. The most important step of the algorithm is the determination of the rotation angle cosine and sine $\cos(\theta)$ and $\sin(\theta)$ respectively starting from given p and q. The better those values are determined, then the faster is the convergence of A_k to a diagonal matrix. The code in the listing (3) shows exactly how those values are determined, maintaining the idea that each rotation should reduce the target A_{pq} to zero. Furthermore the algorithm exits when all the off-diagonal terms are reduces to zero or rather this is done by evaluating the euclidean squared norm of the off-diagonal values.

Listing 3: Determination of $\cos(\theta)$ and $\sin(\theta)$

```
% cos and sin deduction from p, q and a square matrix A
function [c, s] = from_pq(p, q, A)
    if abs(A(p,q)) > eps
        tau = (A(q,q) - A(p,p)) / (2 * A(p,q));
        if tau >= 0
            tau = -tau + sqrt(1 + tau^2);
        else
            tau = -tau - sqrt(1 + tau^2);
        end
        c = 1.0 / sqrt(1 + tau^2);
        s = tau * c;
    else
        c = 1.0;
        s = 0.0;
    end
end
```

(3.1) + (3.2) Comparing classic and cyclic jacobi methods

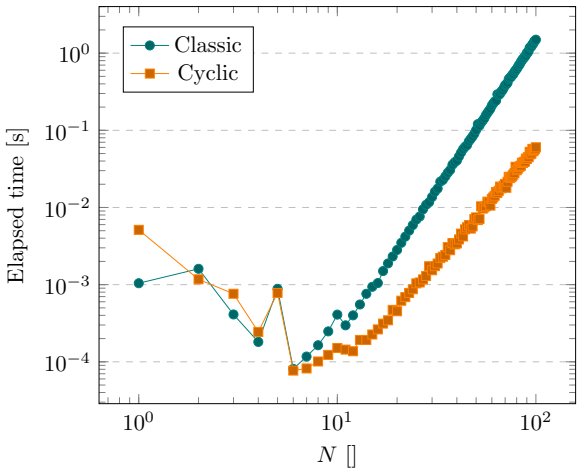
There are two possible approaches in order to determine the p and q coefficients: the *classic Jacobi* method shown in listing (4) takes p and q as the indexes corresponding to the maximum off-diagonal term, while *cyclic Jacobi* method loops directly over all the indexes of the lower triangular side of A as shown in script (5). Although both algorithm share the same time complexity $\mathcal{O}(N^3)$, where N is the size of A, the *cyclic Jacobi* method performs a constant number of `while` iterations for any value of N. The graphs in figures (2a) and (2b) show exactly this result, demonstrating that the *cyclic* approach is better than the *classic* one.

Listing 4: Implementation of the classic Jacobi method

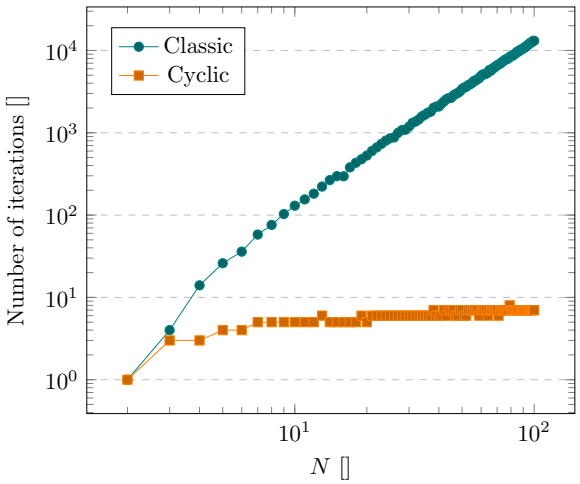
```
% P = transition matrix such that A * P = P * D
% A = diagonal matrix of eigenvalues
function [P, D] = eig_j(A)
    [Ni, Nj] = size(A);
    assert(Ni == Nj, "The input must be diagonal");
    N = Ni;
    P = eye(N);
    D = A; % copy matrix A

    while off(D, N) > (1e-9 * N^2)
        [p, q] = offmax(D,N);
        [c, s] = from_pq(p, q, D);
        [D, P] = transition(D, P, p, q, c, s, N);
    end
end
```

Listing 5: Implementation of the cyclic Jacobi method

```
while off(D, N) > (1e-9 * N^2)
    for p = 1:(N-1)
        for q = (p+1):N
            [c, s] = from_pq(p, q, D);
            [D, P] = transition(D, P, p, q, c, s, N);
        end
    end
end
```



(a) Elapsed time of both algorithm as function of the matrix size N



(b) Iteration count of both algorithm as function of the matrix size

(3.3) Applying the axis aligned rotation matrix J

The `transition` function is supposed to preform the matrix multiplications $A_{k+1} = J^T A_k J$ and $P_{k+1} = P_k J$. However, the standard matrix multiplication is unnecessarily expensive and given that the J application differs from the identity just for few symmetric terms, then the computation time can be drammaticly reduced.

Let $B = J^T A$ and noting a_{ij} and b_{ij} the i, j-th components of A and B respectively, then:

$$\forall j: \quad b_{ij} = \begin{cases} i = p: & \cos(\theta)\, a_{pj} + \sin(\theta)\, a_{qj} \\ i = q: & -\sin(\theta)\, a_{pj} + \cos(\theta)\, a_{qj} \\ \text{otherwise}: & a_{ij} \end{cases} \tag{24}$$

The same reasonning can be done for the multiplication BJ noticing that $BJ = (J^T B^T)^T = (J^T (b_{ji}))^T$, which means that if $a'_{ij} = (BJ)_{ij}$, the entire transformation becomes:

$$\forall i: \quad a'_{ij} = \begin{cases} j = p: & \cos(\theta)\, b_{ip} + \sin(\theta)\, b_{iq} \\ j = q: & -\sin(\theta)\, b_{ip} + \cos(\theta)\, b_{iq} \\ \text{otherwise}: & a_{ij} \end{cases} \tag{25}$$

The same can be applied at the same time for the evolution of the matrix P (see equation (19)). The code listing (6) contains a matlab implementation of the transformation of A and P; notice that this approach reduces exactly the time complexity to $\mathcal{O}(N)$, avoiding the $\mathcal{O}(N^3)$ standard matrix product.

Listing 6: Transition optimized code

```
% apply a rotation to A: J(p,q)' A J(p,q)
function [A, P] = transition(A, P, p, q, c, s, N)
    % left multiplication by J'
    for j = 1:N
        t = A(p,j);
        A(p,j) = c * t - s * A(q,j);
        A(q,j) = s * t + c * A(q,j);
    end

    % right multiplication by J
    for i = 1:N
        t = A(i,p);
```

```
13          A(i,p) = c * t - s * A(i,q);
14          A(i,q) = s * t + c * A(i,q);
15
16          tp = P(i,p);
17          P(i,p) = c * tp - s * P(i,q);
18          P(i,q) = s * tp + c * P(i,q);
19      end
20  end
```

(4) Landau levels in a square-lattice model

Conclusion

Appendix: matlab codes

Listing 7: Algorithm which solves a system

```
function x = solve(A, b)
    % decompose LU
    [L, U, P] = lu_decomposition(A);

    y = solve_lower(L, P * b);
    x = solve_upper(U, y);
end

function y = solve_lower(L, b)
    [~,N] = size(L);
    y = b;
    % start by the top
    for k=1:N-1 % iteration on columns
        y(k) = y(k) /  L(k,k);
        for i=k+1:N % iteration on rows
            y(i) = y(i) - y(k) * L(i,k);
        end
    end
    y(N) = y(N) / L(N,N);
end

function x = solve_upper(U, y)
    [~,N] = size(U);
    x = y;
    % start by the top
    for k=N:-1:2 % iteration on columns
        x(k) = x(k) /  U(k,k);
        for i=k-1:-1:1 % iteration on rows
            x(i) = x(i) - x(k) * U(i,k);
        end
    end
    x(1) = x(1) / U(1,1);
end
```

Listing 8: Power iteration method implementation

```
function [ vec, val ] = eig_power(inputmatrix)
    [Ni, Nj] = size(inputmatrix);
    assert(Ni == Nj || Ni == 0, 'Matrix must be square  or non-zero')
    vec = zeros(Ni, 1);
    vec(1) = 1;
    val = ctranspose(vec) * inputmatrix * vec;
    oldval = 0;

    while abs(oldval - val) > eps
        vec = inputmatrix * vec;
        vec = vec / norm(vec);
        oldval = val;
        val = ctranspose(vec) * inputmatrix * vec;
    end
```

```
end
```

Listing 9: Inverse power iteration method implementation

```
function [ vec, val ] = eig_ipower(inputmatrix, target)
    [Ni, Nj] = size(inputmatrix);
    assert(Ni == Nj || Ni == 0, 'Matrix must be square  or non-zero')
    vec = rand(Ni,1);
    vec = vec / norm(vec);
    val = ctranspose(vec) * inputmatrix * vec;
    oldval = target;
    I_target = eye(Ni) * target;
    count = 0;

    % exit if the delta is zero or the delta is constant
    while abs(oldval - val) > eps && count < 100
        vec = (inputmatrix - I_target) \ vec;
        vec = vec / norm(vec);

        oldval = val;
        val = ctranspose(vec) * inputmatrix * vec;

        count = count + 1;
    end

    if norm((inputmatrix * vec)  - (val * vec)) > 1e-10
        % rerun with this target
        [vec, val] = eig_ipower(inputmatrix, val);
    end
end
```

Listing 10: Rayleigh quotient iteration method implementation

```
function [ vec, val ] = eig_rq(inputmatrix, target)
    [Ni, Nj] = size(inputmatrix);
    assert(Ni == Nj || Ni == 0, 'Matrix must be square or non-zero')
    vec = zeros(Ni, 1);
    vec(1) = 1;

    val = ctranspose(vec) * inputmatrix * vec;
    oldval = target;
    I = eye(Ni);

    while abs(val - oldval) > 1e-10
        vec = (inputmatrix - I * oldval) \ vec;
        vec = vec / norm(vec);
        oldval = val;
        val = ctranspose(vec) * inputmatrix * vec;
    end
    %val = newval - target;
end
```

Listing 11: Interval bisection strategy implementation

```
function [eigval, eigvect] = eig_first(A, n)
    [psi_lower, lower] = eig_ipower(A, 0);
    found = {{lower, psi_lower}};
```

```
4
5       while length(found) < n
6           [psi_upper, upper] = eig_ipower(A, lower * 2);
7           result = eig_between(A, lower, upper);
8           found = {found{:} result{:}};
9           if abs(upper - lower) > 1e-10
10              found = {found{:} {upper, psi_upper}};
11              lower = upper;
12          else
13              lower = lower * 2;
14          end
15      end
16
17      % rearrange output
18      eigval = cell(n,1);
19      eigvect = cell(n,1);
20      for i = 1:n
21          eigval{i} = found{i}{1};
22          eigvect{i} = found{i}{2};
23      end
24  end
25
26  % lower and upper are supposed to be eigenvalues
27  function found = eig_between(A, lower, upper)
28      found = {};
29      if abs(lower - upper) > 1e-10
30          mean = (lower + upper) / 2.0;
31          [psi, val] = eig_ipower(A, mean);
32
33          if abs(lower - val) > 1e-10 && abs(upper - val) > 1e-10
34              flow = eig_between(A, lower, val);
35              fhigh = eig_between(A, val, upper);
36              found = {flow{:}, {val, psi}, fhigh{:}};
37          end
38      end
39  end
```

Documentation and sources

[1] https://en.wikipedia.org/wiki/SI_base_unit

[2] https://math.stackexchange.com/questions/1274373/proof-for-existence-of-lu-decompositio

[3] https://en.wikipedia.org/wiki/Inner_product_space

[4] https://en.wikipedia.org/wiki/Spectral_theorem