united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

SMR/1310 - 7

SPRING COLLEGE ON NUMERICAL METHODS IN ELECTRONIC STRUCTURE THEORY

(7 - 25 May 2001)

"Numerical Linear Algebra - I" (NLA - I)

presented by:

D. ALFÈ

University College London
Department of Geological Sciences
and
Department of Physics and Astronomy
London
United Kingdom

Numerical Linear Algebra I

Dario Alfè

May 10, 2001

Reference: *Numerical recipes, the Art of Scientific Computing*, by Press, Teukolsky, Vetterlong and Flannery, Cambridge University Press, 1992.

1 Linear Algebraic Equations

A set of linear algebraic equations is the following:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N &= b_2 \\ \ldots & \\ a_{M1}x_1 + a_{M2}x_2 + \ldots + a_{MN}x_N &= b_M \end{aligned} \tag{1}$$

where $x_j, j = 1, \ldots N$ are the unknowns, which are related by the M equations. Here the coefficients $a_{ij}, i = 1, \ldots M$ and $j = 1, \ldots N$ are known numbers as well as the right hand side quantities $b_i, i = 1, \ldots M$.

If the number of equations is equal to the number of unknowns ($N = M$) then one can find a unique solution to the linear system, provided that the equations are all linearly independent. If one or more equations are linear combinations of others then there is a *row degeneracy*. It is clear that in this case one can eliminate the equation which is linearly dependent on the others, with the result of having $M < N$, and the linear system does not have a unique solution. In this case the system is called *singular*.

Suppose that some of the equations are 'almost' linear combinations of the others. The analytic solution is still perfectly defined, but if one tries to solve the system on a computer then *roundoff* error may render the equation linearly dependent at some stage in the solution problem.

An other possible problem is that the accumulation of *roundoff* errors may spoil the solution procedure and give wrong results, as it can be verified by substituting the solutions into the unknowns.

1.1 Matrices

Equation (1) can be rewritten in the following way:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2}$$

Where $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ is the vector of the unknowns, $\mathbf{A}$ is the matrix formed with the coefficients of the equations:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ \ldots & \ldots & \ldots & \ldots \\ a_{M1} & a_{M2} & \ldots & a_{MN} \end{bmatrix} \tag{3}$$

and $\mathbf{b} = (b_1, b_2, \ldots, b_M)$ is the vector formed with the right hand sides of equation (1).

Conventionally, the first index of an element a_{ij} denotes the row position in the matrix and the second the column position. Although a matrix is a two dimensional array, a computer will store it as a sequence of numbers in its memory. Depending on the programming language the matrix can be stored by columns (i.e. $a_{11}, a_{21}, \ldots a_{N1}, a_{12}, a_{22}$ etc.) like in FORTRAN, or by rows like in C and PASCAL for example. A matrix element can usually be referenced by the two values of the indices, but it is important to understand the difference between *logical* dimensions and *physical* dimensions. When you pass a matrix to a routine you also need to pass the correct dimensions of the matrix.

1.2 Computational Linear Algebra

There are a number of things that fall in the category of "Computational Linear Algebra". We will consider here the following:

- Solution of the linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ is a square non singular matrix.
- Calculation of the inverse $\mathbf{A^{-1}}$, so that $\mathbf{A} \cdot \mathbf{A^{-1}} = \mathbf{A^{-1}} \cdot \mathbf{A} = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix (all zeros except ones on the diagonal).
- Calculation of the determinant of a square matrix $\mathbf{A}$.

All the tasks that we will describe can be solved using standard linear algebra software, like LAPACK. You should always use these routines whenever available, which are robust, tested and optimized, you cannot hope do to better. However, it is worthwhile to go through some of the algorithms and try to understand what they do.

1.3 Gaussian Elimination with Back-substitution

The first algorithm we consider here is probably the most simple and intuitive one, the so called *Gaussian elimination with back-substitution.*

We want to solve the linear system

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{4}$$

We can multiply each equation by a constant or sum or subtract two or more equations without changing the solution of the system. We can also interchange rows or columns of the matrix $\mathbf{A}$ and the corresponding rows of the right hand side $\mathbf{b}$ without affecting the solution. Gaussian elimination takes advantage of these properties to solve the linear system. The algorithm works like that. Take the second equation of the system and multiply it by a_{11}/a_{21} so that the coefficients that multiply x_1 are the same in equation 1 and equation 2. Now subtract equation 1 from equation 2 and write the result as the new equation 2. Do the same thing for the third and the fourth equation and you have

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}, \tag{5}$$

where $a'_{ij} = a_{ij} \times a_{11}/a_{i1} - a_{1j}$, and $b'_i = b_i \times a_{11}/a_{i1} - b_1$. The number of operations performed up to now is 4×3 multiplications (3 multiplications involving the last three coefficients a_{ij},

the multiplication for the first coefficient is not needed of course, and the known term b_i for the 3 equations below equation 1) and 4×3 subtractions. In the case of N equations we have $N(N-1)$ operations (multiplications plus subtractions).

Now iterate the procedure, and eliminate the coefficient of x_2 from the third and the fourth equations,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b''_4 \end{bmatrix}, \tag{6}$$

and we have done additionally $(N-1)(N-2)$ operations. Finally eliminate the coefficient of x_3 from the fourth equation so that you have

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b'''_4 \end{bmatrix}. \tag{7}$$

The total number of operations up to now is $N(N-1) + (N-1)(N-2) + \ldots 2$. In the limit of large N this number is $\approx \frac{1}{3}N^3$.

The procedure just described works fine if the element by which we divide is not zero, otherwise we run into trouble. Is is almost guaranteed that a zero or a near zero element will be found in the procedure for large N, and this makes the algorithm as it stands very unstable. The solution to this problem is the so called *pivoting*. We know that the solution of the system does not change if we exchange two columns of the matrix $\mathbf{A}$, as long as we exchange also the corresponding rows of $\mathbf{x}$. In pivoting we look for the *pivot* element in the row we are dealing with, i.e. the largest element of the row and exchange the current column with the one containing the pivot. This of course scrambles the order of the solution vector $\mathbf{x}$, and we have to keep track of that.

Now we solve the system. We immediately find

$$x_4 = \frac{b'''_4}{a'''_{44}}. \tag{8}$$

Now substitute x_4 in the third equation and solve for x_3:

$$x_3 = \frac{b''_3}{a''_{33}} - x_4 \frac{a''_{34}}{a''_{33}}, \tag{9}$$

and then proceed to the next equation. The ith step of the procedure is

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^{N} x_j a_{ij} \right], \tag{10}$$

where we dropped the primes for simplicity of notation. This procedure is called *back-substitution*. The number of operations in back-substitution is of the order $\frac{1}{2}N^2$, so for large N the total time is dominated by the Gaussian elimination procedure.

Suppose now that you want to solve a second linear system where the coefficients a_{ij} are the same but the right hand side of the equations are different. Naively, one would go through the same procedure again, and solve the system from scratch in $\approx \frac{1}{3}N^3$ number of operations. However, if the right hand sides are known in advance one can solve for all of them at once, so that one only needs to perform additional $\frac{1}{2}N^2$ operations for each new right hand side and the $\approx \frac{1}{2}N^2$ operations of the new back-substitution. If one has N different right hand sides the total number of operations to solve all N systems is $\approx (\frac{1}{3} + \frac{1}{2} + \frac{1}{2})N^3 = \frac{4}{3}N^3$.

1.4 Inversion

Inverting a matrix is one other important task of numerical linear algebra. The knowledge of the inverse of a matrix can also be exploited to solve the linear system (4):

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \longrightarrow \mathbf{x} = \mathbf{A^{-1}} \cdot \mathbf{b}, \tag{11}$$

Gaussian elimination with back-substitution can be used to invert a matrix. It is easy to see that by solving the linear system (4) for a set of right hand side all zeros except for a one in the ith position one gets the ith column of the inverse matrix $\mathbf{A^{-1}}$. Solving again for the one in all possible positions one gets the full inverse matrix. The number of operations needed is less than $\frac{4}{3}N^3$, as one would need to solve N linear systems, because the N right hand sides contain all zeros except for one element. If this is taken into account the number of right hand side manipulation is only $\frac{1}{6}N^3$, which brings the total number of operations to invert a matrix to N^3.

1.5 LU decomposition

Gaussian elimination with back-substitution can be used for solving linear systems and finding the inverse of a matrix. This algorithm is very simple and stable, however, it is not the best algorithm, and the 'LU decomposition' we will describe in this section is superior.

Suppose that one is able to perform the following decomposition for the matrix $\mathbf{A}$:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U}, \tag{12}$$

where $\mathbf{L}$ and $\mathbf{U}$ are a lower triangular and upper triangular matrices respectively,

$$\mathbf{L} = \begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix}, \qquad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}, \tag{13}$$

the one can rewrite the linear system (4) as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{14}$$

Now one can use *forward-substitution* and solve the system

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}: \tag{15}$$

$$y_1 = \frac{b_1}{l_{11}} \tag{16}$$

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=1}^{i-1} y_j l_{ij} \right], \qquad i = 2, 3, \ldots, N \tag{17}$$

and *back-substitution* for the system

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}: \tag{18}$$

$$x_N = \frac{y_N}{u_{NN}} \tag{19}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{N} x_j u_{ij} \right], \qquad i = N-1, N-2, \ldots, 1. \tag{20}$$

The procedure to perform an LU decomposition is not particularly instructive and we will not address it here.

If we count the number of operations needed to invert a matrix or to solve a set of linear systems we find N^3 and $\frac{4}{3}N^3$ respectively, i.e. exactly the same as in the Gaussian elimination procedure. However, once a matrix **A** has been LU decomposed one can solve a linear system for as many right hand sides as wanted, which don't have to be known in advance. This is a distinct advantage of LU decomposition over the previous algorithm.

It can be shown that the diagonal of the **L** matrix can be set to one without loss of generality, so that a call to a routine that performs the LU decomposition of a matrix **A** may look like this:

```
call ludcmp(a,n,np,indx,d)
```

where `a` is the matrix to decompose. The routine would then return **L** and **U** in the lower triangular and upper triangular part of `a` respectively. The diagonal of **L** is not returned (all ones). As in the Gaussian elimination also in this algorithm case we face the problem of possible division by zero, it is necessary then to do 'pivoting', which implies that the returned matrix is the LU decomposition not of the original **A** but of a row permutation of it. We need to keep track of these permutations, and this is done in the output vector `index`. The output `d` is 1 or -1 depending on whether the number of row interchanges was even or odd, respectively.

Once we have the LU decomposition of the matrix **A** we can solve any linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ using back-substitution. This would be done with a call to the back-substitution routine of the type:

```
call lubksb(a,n,np,indx,b)
```

where the right hand side **b** is passed to the routine in the vector `b`, which contains the solution **x** on exit. One can then repeat the call to the back-substitution routine with as many different right hand sides as wanted.

The calls just described are the calls to the actual routines implemented in the *Numerical recipes*. It is clear how to invert a matrix using these two routines: one just needs to solve N linear systems with the right hand side being $(1, 0, \ldots, 0), (0, 1, \ldots, 0) \ldots (0, 0, \ldots, 1)$, and the solutions vectors are the columns of $\mathbf{A}^{-1}$.

1.6 Determinant of a Matrix

The determinant of a triangular matrix is just product of the elements on the diagonal. The determinant of a product of matrices is just the product of the determinants, so that if we know the LU decomposition of a matrix we can calculate its determinant by multiplying the diagonal elements:

$$\det(\mathbf{A}) = \det(\mathbf{L} \cdot \mathbf{U}) = \det(\mathbf{L}) \det(\mathbf{U}) = \prod_{j=1}^{N} l_{jj} \prod_{j=1}^{N} u_{jj}. \tag{21}$$

We have seen that we can set $l_{ii} = 1$ without loss of generality, so that the determinant of the **L** matrix is equal to 1, and we have

$$\det(\mathbf{A}) = \det(\mathbf{U}) = \prod_{j=1}^{N} u_{jj}. \tag{22}$$

If we use the routine `ludcmp` we can calculate the determinant as

```
call ludcmp(a,n,np,indx,d)
do j = 1, n
```

```
d = d*a(j,j)
enddo
```

1.7 Iterative improvement of a solution to linear equations

We mentioned earlier that the accumulation of *roundoff* errors may spoil the solution of a linear system. This can be easily verified by substituting the numerical solution into the equations of the linear system. If this happens one can *iteratively improve* the solution in the following way. Suppose that a vector $\mathbf{x}$ is the exact solution of the linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{23}$$

Because of roundoff errors when you solve the system with one of the methods described you don't find $\mathbf{x}$ but only a wrong solution $\mathbf{x} + \delta\mathbf{x}$, where $\delta\mathbf{x}$ is an unknown error. When you multiply $\mathbf{A}$ by the wrong solution $\mathbf{x} + \delta\mathbf{x}$ you don't obtain $\mathbf{b}$ but $\mathbf{b} + \delta\mathbf{b}$

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{24}$$

Now subtract (23) from (24)

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \tag{25}$$

$\delta\mathbf{b}$ is known, so you can solve (25) for $\delta\mathbf{x}$ and improve your solution. In principle the corrector vector $\delta\mathbf{x}$ could be wrong, for the same reason the solution $\mathbf{x}$ was wrong in first place, so you may need to iterate the procedure until it converges to the right solution. If you have chosen LU decomposition to solve the system than you can simply use back-substitution to solve (25), which is an order N^2 task. Since you have already performed an order N^3 task by LU decomposing the matrix $\mathbf{A}$ the additional time to improve the solution is usually negligible, and it is well worth doing it.

2 Eigensystems

An $N \times N$ matrix $\mathbf{A}$ has *eigenvector* $\mathbf{x}$ and *eigenvalue* λ if

$$\mathbf{A} \cdot \mathbf{x} = \lambda\mathbf{x}. \tag{26}$$

It is clear that any multiple of an eigenvector $\mathbf{x}$ is still and eigenvector of the matrix with the same eigenvalue, so we will not consider this as a distinct eigenvector. Eq. (26) is obviously always satisfied for the zero vector, which we will not consider as an eigenvector. A sufficient and necessary condition for Eq. (26) to hold is

$$\det|\mathbf{A} - \lambda\mathbf{1}| = 0 \tag{27}$$

which is an Nth degree equation in λ whose solutions are the eigenvalues of $\mathbf{A}$. The solution of this equation would yield the eigenvalues of the matrix, but this is not usually a good method to solve the problem. If two or more eigenvalues are equal they are called *degenerate*. If $\lambda = 0$ is an eigenvalue than the matrix is *singular*, and cannot be inverted. By adding $\tau\mathbf{x}$ on both sides of Eq. (26) one gets

$$\mathbf{A} \cdot \mathbf{x} + \tau\mathbf{x} = (\mathbf{A} + \tau\mathbf{1}) \cdot \mathbf{x} = (\lambda + \tau)\mathbf{x}, \tag{28}$$

which shows that the eigenvalues of a matrix can be *shifted* by adding to the matrix a constant times the identity matrix. Note that the eigenvectors are not modified by this procedure. This property can be useful sometime to remove singularity from a matrix which can then be inverted.

2.1 Definitions

A matrix is called *symmetric* if it is equal to its transpose,

$$\mathbf{A} = \mathbf{A}^T \qquad \text{or} \qquad a_{ij} = a_{ji} \tag{29}$$

It is called *hermitian* if it is equal to the complex conjugate of its transpose

$$\mathbf{A} = \mathbf{A}^\dagger \qquad \text{or} \qquad a_{ij} = a^*_{ji} \tag{30}$$

It is called *orthogonal* if the transpose is equal to its inverse

$$\mathbf{A}^{-1} = \mathbf{A}^T \tag{31}$$

and *unitary* if its hermitian conjugate is equal to the inverse. For real matrices *hermitian* is the same as *symmetric* and *unitary* is the same as *orthogonal.* Hermitian and symmetric matrices are particularly important in quantum mechanics problems, because the operators associated to physical observable can be represented as hermitian matrices, or symmetric matrices in the special case where they are real. Therefore, in what follows we will only consider these type of matrices. One important feature of hermitian matrices is that the eigenvalues are real. If the matrix is real and symmetric then also the eigenvectors are real. If the eigenvalues of a symmetric matrix are non degenerate (i.e. all distinct) then the eigenvectors are orthogonal. This can be seen with the following argument. Construct the matrix $\mathbf{X}$ whose columns are the eigenvectors of $\mathbf{A}$, then the eigenvalue equation (26) can be written

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{X} \cdot \mathrm{diag}(\lambda_1, \ldots, \lambda_N) \tag{32}$$

now take the transpose of the equation above, since the matrix $\mathbf{A}$ is symmetric we have:

$$\mathbf{X^T} \cdot \mathbf{A} = \mathrm{diag}(\lambda_1, \ldots, \lambda_N) \cdot \mathbf{X^T} \tag{33}$$

Now multiply (32) on the left by $\mathbf{X^T}$ and (33) on the right by $\mathbf{X}$ and you have

$$\mathrm{diag}(\lambda_1, \ldots, \lambda_N) \cdot (\mathbf{X^T} \cdot \mathbf{X}) = (\mathbf{X^T} \cdot \mathbf{X}) \cdot \mathrm{diag}(\lambda_1, \ldots, \lambda_N) \tag{34}$$

and since the $\lambda_i, i = 1, \ldots, N$ are all distinct then the matrix $\mathbf{X^T} \cdot \mathbf{X}$ has to be diagonal, which means than the eigenvectors are all orthogonal. Since the eigenvectors are defined with the freedom of a multiplicative constant they can be normalised to one, and the matrix $\mathbf{X}$ is orthogonal, $\mathbf{X^T} \cdot \mathbf{X} = \mathbf{1}$. If some of the eigenvalues are degenerate then the corresponding eigenvectors don't need to be orthogonal, although they are still orthogonal to the eigenvectors corresponding to different eigenvalues. A linear combination of the eigenvectors in the degenerate subspace is still an eigenvector corresponding to the same eigenvalue. Using this property one can always form linear combinations of eigenvectors so that they are all orthogonal (Gram-Schmidt orthogonalisation for example).

2.2 Matrices in Quantum mechanics

The basic equation to be solved in a quantum mechanics problem is

$$H\psi = \epsilon\psi \tag{35}$$

where H is the Hamiltonian of the system, ψ the *eigenfunction* and ϵ the corresponding *eigenvalue.* The eigenfunctions are vectors of the Hilbert space, and we indicate with $H\psi$ the vector of the Hilbert space which is the result of the application of the Hamiltonian operator H on ψ. The scalar product of two vectors in the Hilbert space will be indicated with $\psi \cdot \phi$.

The way to solve (35) is to expand ψ as a linear combination of a complete orthonormal basis set $\{\phi_i, i = 1, \ldots N, \phi_i \cdot \phi_j = 1 \text{ if } i = j,\ 0 \text{ otherwise}\}$, and rewrite Eq. (35)

$$H \sum_{i=1}^{N} c_i \phi_i = \epsilon \sum_{i=1}^{N} c_i \phi_i, \tag{36}$$

so the problem of finding ψ has now become the problem of finding the coefficients c_i.

We make now a brief regression on basis sets. In general a complete basis set contains an infinite number of basis functions, so that the sums here above would have an infinite number of terms. Of course in a practical solution of the problem only a finite number of basis functions can be included in the calculations. This brings us to the discussion on the choice of the best basis set. There are two main approaches to the problem. The first is to look for the best basis functions appropriate to the problem, so that one can hope to include only a small number of different functions and approximate the wave-function accurately enough. For example, if one is dealing with atoms or groups of atoms, or even periodic solids, one may think that an appropriate basis set would be a series of functions localised on the atoms, like Gaussian for example. In fact, some quantum mechanics codes like CRYSTAL of GAUSSIAN use indeed Gaussian as basis set to expand the orbitals. The drawback of this approach is that the basis functions are somewhat biased by the choices of the user. The second philosophy is the plane wave approach, which comes naturally when one is dealing with periodic solids. This is the approach used in the PWSCF code. The advantage of this approach is the simplicity and the generality, one can improve the quality of the basis set simply including more and more plane waves with larger and larger frequencies, and there is a very precise way of doing this systematically. The drawback of plane waves is that one needs a much larger number of them to get an accurate description of the wave-functions.

Let's come back now to Eq. (36). If we make the scalar product of both sides with ϕ_j we get

$$\sum_i c_i \phi_j \cdot H\phi_i = \epsilon \sum_i c_i \phi_j \cdot \phi_i = \epsilon c_j \tag{37}$$

which can be rewritten as

$$\mathbf{H} \cdot \mathbf{c} = \epsilon \mathbf{c} \tag{38}$$

with $H_{ij} = \phi_j \cdot H\phi_i$ and $\mathbf{c} = (c_1, c_2, \ldots, c_N)$. We see then that the problem of finding ϵ and the expansion coefficients c_i is the problem of finding the eigenvalues and the eigenvectors of the matrix $\mathbf{H}$. Since the Hamiltonian is an hermitian operator, the matrix $\mathbf{H}$ is hermitian.

2.3 Diagonalisation of a matrix

We have seen that

$$\mathbf{X^T} \cdot \mathbf{A} \cdot \mathbf{X} = \mathrm{diag}(\lambda_1, \ldots, \lambda_N) \tag{39}$$

which means that the matrix of the eigenvectors can be used to *diagonalise* the matrix $\mathbf{A}$. Eq. (39) is a particular case of a *similarity transform* of the matrix $\mathbf{A}$,

$$\mathbf{A} \rightarrow \mathbf{Z^{-1}} \cdot \mathbf{A} \cdot \mathbf{Z} \tag{40}$$

where $\mathbf{Z}$ is some transformation matrix (doesn't need to be orthogonal). Similarity transformations don't change the eigenvalues of the matrix,

$$\det|\mathbf{Z^{-1}} \cdot \mathbf{A} \cdot \mathbf{Z} - \lambda\mathbf{1}| = \det|\mathbf{Z^{-1}} \cdot (\mathbf{A} - \lambda\mathbf{1}) \cdot \mathbf{Z}| = \det|\mathbf{A} - \lambda\mathbf{1}| \tag{41}$$

Most diagonalisation routines use this property, i.e. they build up the matrix of the eigenvectors with a series of similarity transformations:

$$\mathbf{A} \rightarrow \mathbf{Z_1^{-1}} \cdot \mathbf{A} \cdot \mathbf{Z_1} \rightarrow \mathbf{Z_2^{-1}} \cdot \mathbf{Z_1^{-1}} \cdot \mathbf{A} \cdot \mathbf{Z_1} \cdot \mathbf{Z_2} \rightarrow \text{etc.} \tag{42}$$

and

$$\mathbf{X} = \mathbf{Z_1} \cdot \mathbf{Z_2} \cdot \ldots \tag{43}$$

We will discuss now in some details one of these diagonalisation procedures, the Jacobi method.

2.4 Jacobi matrix diagonalisation method

The Jacobi method is a simple algorithm that can be used to diagonalise symmetric matrices. It is not the most efficient, so in general one would not use it, but it is very stable and relatively easy to parallelise so that it can be solved on an array of processors. The Jacobi idea is to build up the matrix of the eigenvectors with a series of simple similarity transformations of the form

$$\mathbf{J}_{pq} = \begin{array}{c} \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{array}{l} \\ \\ \leftarrow p \\ \\ \leftarrow q \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & \uparrow & & \uparrow & & \\ & & p & & q & & \end{array} \end{array} \tag{44}$$

which are simple planar rotations, called *Jacobi rotation.* A Jacobi rotation differs from the identity matrix by having c on the diagonal positions qq and pp and s and $-s$ at positions pq and qp respectively; c and s are chosen so that $c^2 + s^2 = 1$. The matrix $\mathbf{A}' = \mathbf{J^T}_{pq}\mathbf{A}\mathbf{J}_{pq}$ is identical to $\mathbf{A}$ except for the rows and columns p and q, and we have:

$$\begin{aligned} a'_{rp} &= ca_{rp} - sa_{rq} \\ a'_{rq} &= ca_{rq} + sa_{rp} \end{aligned} \qquad r \neq p, r \neq q \tag{45}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \tag{46}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \tag{47}$$

$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{48}$$

If we choose c and s so that

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} = \begin{pmatrix} a'_{pp} & 0 \\ 0 & a'_{qq} \end{pmatrix} \tag{49}$$

we have set to zero one (and its symmetric) off-diagonal element of $\mathbf{A}$. The next step is to generate a second transformation for a different couple of indices (p', q'). However, this second transformation will zero the elements $a_{p'q'}, a_{q'p'}$, but will also affect all the elements in the rows and columns (p', q'). If $p' \neq p$ and $q' \neq q$ than the previously zeroed element in position (p, q) will not be affected, but it is clear that as we go along with all the elements in the matrix it is inevitable to modify rows and columns in which some of the elements were already zeroed by previous transformations. So the matrix cannot be diagonalised 'exactly'. However, it is easy to see that each rotation reduces the 'norm' of the *off-diagonal* elements of the transformed matrix,

$$off(\mathbf{A}') = \sum_{j \neq i} a'^2_{ij} = off(\mathbf{A}) - 2a^2_{pq} \tag{50}$$

This is because

$$\sum_{r\neq p,r\neq q} a'^2_{rp} + a'^2_{rq} = \sum_{r\neq p,r\neq q} a^2_{rp} + a^2_{rq} \tag{51}$$

and $a'_{pq} = a'_{qp} = 0$. So each Jacobi rotation reduces the norm of the off-diagonal part of the matrix, provided that one doesn't choose the transformation corresponding to an element which is already zero. It is possible then to diagonalise the matrix 'iteratively', applying Jacobi rotations until the matrix is diagonal within machine accuracy. Accumulating the products of the Jacobi rotations one obtains the matrix of the eigenvectors $\mathbf{X} = \prod \mathbf{J}_{pq}$.

Since the choice of the element to zero in a particular step of the procedure is completely arbitrary one may ask what is the best thing to do. Naively, one would say that the best choice would be the largest element among the survivors. This is true, of course, and is the choice in Jacobi's original algorithm in 1846, when the calculations were done by hand. On large matrices to be diagonalised on a computer however this is very inefficient because the searching procedure among all the element of the matrix involves order N^2 operations, while one Jacobi transformation only involves order N operations. A better algorithm is the *cyclic Jacobi method*, in which one simply annihilate the elements in a sequential order, as $\mathbf{J}_{12}, \mathbf{J}_{13}, \ldots, \mathbf{J}_{1n}, \mathbf{J}_{23}, \mathbf{J}_{24}$ etc.

A cycle going through the all the elements of the matrix is called a *sweep*. There are $N(N-1)/2$ different rotations in a sweep, each involving order $4N$ operations, so each sweep has a cost of order $2N^3$ operations. Typical matrices need between 6 and 10 sweeps to be diagonalised, or $12N^3$ to $20N^3$ operations.

2.4.1 A parallel implementation of the Jacobi algorithm

The idea to solve the problem on a ring of processors (PE) is the following. Each *sweep* if formed by $N(N-1)/2$ Jacobi rotations. Let us also assume N an even number. Among all the possible N rotations it is always possible to chose $N/2$ of them which are independent, i.e. applied to $N/2$ not overlapping couples of indices. We call a *sub-sweep* each of the possible $(N-1)$ choices of $N/2$ independent rotations. Now, each *sub-sweep* can be divided among a set of (at most) $N/2$ different PE's. Moreover, each PE only needs to know only the columns of the matrix $\mathbf{A}$ corresponding to the partial part of the *sub-sweep* which must be solved on that PE. Example: suppose $N = 4$ and 2 PE's. There are 3 possible *sub-sweeps*,
(1,2)(3,4); (1,4)(2,3); (1,3)(2,4);
for the first *sub-sweep* PE 1 houses the columns $(1,2)$ and PE 2 the columns $(3,4)$. PE 1 calculates $\mathbf{J}_{12}$ and makes the product $\mathbf{A}\cdot\mathbf{J}_{12}$, while PE 2 calculates $\mathbf{J}_{34}$ and makes the product $\mathbf{A}\cdot\mathbf{J}_{34}$. Note that PE 1 only needs columns 1 and 2 to do the product, while PE 2 only columns 3 and 4. Then PE 1 sends $\mathbf{J}_{12}$ to PE 2 and receives $\mathbf{J}_{34}$ from PE 2. Now both PE's can calculate $\mathbf{J}^{\mathbf{T}}\cdot\mathbf{A}\cdot\mathbf{J}$ and the first *sub-sweep* is complete. Before starting the second *sub-sweep*, column 2 is sent from PE 1 to PE 2 and column 4 from PE 2 to PE 1 and the algorithm is repeated. Finally, column 4 is sent from PE 1 to PE 2 and column 3 from PE 2 to PE 1 and the last *sub-sweep* is performed. In the general case one has N columns and p PE's, with $N/2 \geq p$ and the algorithm is implemented in the following way:

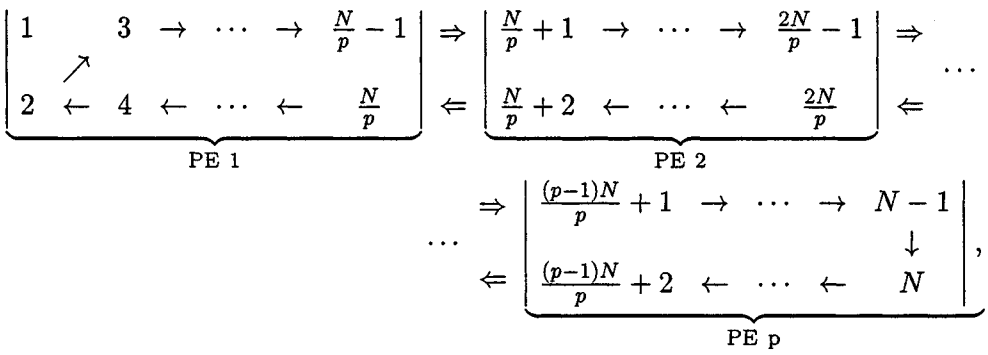
Starting configuration:

$$\underbrace{\left|\begin{matrix}1 & 3 & \cdots & \frac{N}{p}-1\\ 2 & 4 & \cdots & \frac{N}{p}\end{matrix}\right|}_{\text{PE 1}}\underbrace{\left|\begin{matrix}\frac{N}{p}+1 & \cdots & \frac{2N}{p}-1\\ \frac{N}{p}+2 & \cdots & \frac{2N}{p}\end{matrix}\right|}_{\text{PE 2}}\cdots\underbrace{\left|\begin{matrix}\frac{(p-1)N}{p}+1 & \cdots & N-1\\ \frac{(p-1)N}{p}+2 & \cdots & N\end{matrix}\right|}_{\text{PE p}},$$

the couples that define the Jacobi rotations are the columns of the matrices above, for example the first PE houses the first N/p columns and applies the Jacobi rotations for the couples

$(1,2),(3,4),\cdots(\frac{N}{p}-1,\frac{N}{p})$. Each PE calculates its own part of the matrix $\mathbf{J}$ and sends it to all the other PE's. Once it has in its own memory all the matrix $\mathbf{J}$, it can compute $\mathbf{J^T}\cdot\mathbf{A}\cdot\mathbf{J}$, concluding the *sub-sweep*.

Following steps:



with the symbols $(\rightarrow,\leftarrow)$ we indicate the logical movements of the columns within each PE, while with $(\Rightarrow,\Leftarrow)$ we refer to physical columns exchange. After $(N-1)$ steps the *sweep* is completed and one can check the quantity $off(\mathbf{A})$ defined above by summing the squares of the off-diagonal terms of the matrix $\mathbf{A}$ on each PE and then summing the partial results.

2.5 Other algorithms

The Jacobi algorithm is simple and stable, but it is not the most efficient one. There are other algorithms that perform the task much quicker. Probably the most widely used is the so called QR algorithm, which can be effectively used on *tridiagonal* matrices, i.e. a matrices with non-zero elements only on the diagonal and right above and below the diagonal. So one first tridiagonalises the matrix, using the Givens method (very similar to the Jacobi method) or the Householder method (more efficient) and then applies the QR algorithm to the tridiagonal matrix. We will not discuss these algorithms here, but they can be found in the *Numerical Recipes*. The total number of operations in the QM algorithm is $\approx 4N^3$, i.e. markedly better then the Jacobi algorithm.