Mathematical Tripos Part IB: Lent 2020

Numerical Analysis – Lecture 12[1]

If the ODE is stiff, we might prefer a *Newton–Raphson* method. Let $\boldsymbol{\psi}(\boldsymbol{y}) = \boldsymbol{y} - \sigma_s h \boldsymbol{f}(t_{n+s}, \boldsymbol{y}) - \boldsymbol{v}$ so the equation we want to solve is $\boldsymbol{\psi}(\boldsymbol{y}_{n+s}) = 0$. The Newton-Raphson method corresponds to the following iteration rule:

$$\boldsymbol{y}_{n+s}^{[j+1]} = \boldsymbol{y}_{n+s}^{[j]} - \left(\frac{\partial \boldsymbol{\psi}}{\partial \boldsymbol{y}}(\boldsymbol{y}_{n+s}^{[j]})\right)^{-1} \boldsymbol{\psi}(\boldsymbol{y}_{n+s}^{[j]}). \tag{4.16}$$

The justification of the above is as follows: suppose that $\boldsymbol{y}_{n+s}^{[j]}$ is an approximation to the solution. We linearise $\boldsymbol{\psi}$ locally around $\boldsymbol{y}_{n+s}^{[j]}$ to get

$$\boldsymbol{\psi}(\boldsymbol{y}_{n+s}) \approx \boldsymbol{\psi}(\boldsymbol{y}_{n+s}^{[j]}) + \frac{\partial \boldsymbol{\psi}}{\partial \boldsymbol{y}}(\boldsymbol{y}_{n+s}^{[j]})(\boldsymbol{y}_{n+s} - \boldsymbol{y}_{n+s}^{[j]}).$$

Setting the right-hand side to zero we get (4.16).

The snag is that repeatedly evaluating and inverting (i.e. LU-factorizing) the Jacobian matrix $\frac{\partial \boldsymbol{\psi}}{\partial \boldsymbol{y}}$ in every iteration is *very* expensive. The remedy is to implement the *modified Newton–Raphson method*, namely

$$\boldsymbol{y}_{n+s}^{[j+1]} = \boldsymbol{y}_{n+s}^{[j]} - \left(\frac{\partial \boldsymbol{\psi}}{\partial \boldsymbol{y}}(\boldsymbol{y}_{n+s}^{[0]})\right)^{-1} \boldsymbol{\psi}(\boldsymbol{y}_{n+s}^{[j]}). \tag{4.17}$$

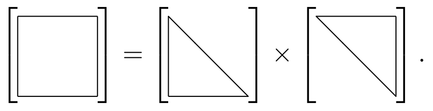
Thus, the Jacobian need be evaluated only *once* a step.

Important observation for future use: Implementation of (4.17) requires repeated solution of linear algebraic systems *with the same matrix.* We will soon study LU factorization of matrices, and there this remark will be appreciated as important and lead to substantial savings. For stiff equations it is much cheaper to solve nonlinear algebraic equations with (4.17) than using a minute step size with a 'bad' (e.g., explicit multistep or explicit RK) method.

5 Numerical linear algebra

5.1 LU factorization and its generalizations

Let A be a real $n \times n$ matrix. We say that the $n \times n$ matrices L and U are an *LU factorization* of A if ***(1)*** L is *unit* lower triangular, i.e., $L_{i,j} = 0$ for $i < j$ and $L_{ii} = 1$ for all i, ***(2)*** U is upper triangular, $U_{i,j} = 0$, $i > j$; and ***(3)*** $A = LU$. Therefore the factorization takes the form



Application 1 Calculation of a determinant: $\det A = (\det L)(\det U) = (\prod_{k=1}^{n} L_{k,k}) \cdot (\prod_{k=1}^{n} U_{k,k})$. This is much faster than the using the formula

$$\det A = \sum_{\sigma} \operatorname{sign}(\sigma) A_{1,\sigma(1)} \dots A_{n,\sigma(n)} \tag{5.1}$$

where the summation is over all permutations σ of $\{1, \dots, n\}$. The number of terms in the sum is $n!$. For a matrix of size $n = 30$, evaluating (5.1) would take more than 10^{10} years, assuming a 10^9 flop/sec. computer (flop = floating point operation)!

[1]Corrections and suggestions to these notes should be emailed to `h.fawzi@damtp.cam.ac.uk`.

Application 2 Testing for nonsingularity: $A = LU$ is nonsingular iff all the diagonal elements of L and U are nonzero.

Application 3 Solution of linear systems: Let $A = LU$ and suppose we wish to solve $A\boldsymbol{x} = \boldsymbol{b}$. This is the same as $L(U\boldsymbol{x}) = \boldsymbol{b}$, which we decompose into $L\boldsymbol{y} = \boldsymbol{b}$, $U\boldsymbol{x} = \boldsymbol{y}$. Both latter systems are triangular and can be calculated easily. Thus, $L_{1,1}y_1 = b_1$ gives y_1, next $L_{2,1}y_1 + L_{2,2}y_2 = b_2$ yields y_2 etc. Having found $\boldsymbol{y}$, we solve for $\boldsymbol{x}$ in reverse order: $U_{n,n}x_n = y_n$ gives x_n, $U_{n-1,n-1}x_{n-1}+U_{n-1,n}x_n = y_{n-1}$ produces x_{n-1} and so on. This requires $\mathcal{O}(n^2)$ computational operations (usually we only bother to count multiplications/divisions).

Application 4 The inverse of A: It is straightforward to devise a direct way of calculating the inverse of triangular matrices, subsequently forming $A^{-1} = U^{-1}L^{-1}$.

The calculation of LU factorization We denote the *columns* of L by $\boldsymbol{l}_1, \boldsymbol{l}_2, \ldots, \boldsymbol{l}_n$ and the *rows* of U by $\boldsymbol{u}_1^\top, \boldsymbol{u}_2^\top, \ldots, \boldsymbol{u}_n^\top$. Hence

$$A = LU = [\ \boldsymbol{l}_1 \quad \boldsymbol{l}_2 \quad \cdots \quad \boldsymbol{l}_n\ ] \begin{bmatrix} \boldsymbol{u}_1^\top \\ \boldsymbol{u}_2^\top \\ \vdots \\ \boldsymbol{u}_n^\top \end{bmatrix} = \sum_{k=1}^{n} \boldsymbol{l}_k \boldsymbol{u}_k^\top. \tag{5.2}$$

Since the first $k-1$ components of $\boldsymbol{l}_k$ and $\boldsymbol{u}_k$ are all zero, each rank-one matrix $\boldsymbol{l}_k\boldsymbol{u}_k^\top$ has zeros in its first $k-1$ rows and columns. We begin our calculation by extracting $\boldsymbol{l}_1$ and $\boldsymbol{u}_1^\top$ from A, and then proceed similarly to extract $\boldsymbol{l}_2$ and $\boldsymbol{u}_2^\top$, etc.

First we note that since the leading $k-1$ elements of $\boldsymbol{l}_k$ and $\boldsymbol{u}_k$ are zero for $k \geq 2$, it follows from (5.2) that $\boldsymbol{u}_1^\top$ is the first row of A and $\boldsymbol{l}_1$ is the first column of A, divided by $A_{1,1}$ (so that $L_{1,1} = 1$).

Next, having found $\boldsymbol{l}_1$ and $\boldsymbol{u}_1$, we form the matrix $A_1 = A - \boldsymbol{l}_1\boldsymbol{u}_1^\top = \sum_{k=2}^{n} \boldsymbol{l}_k\boldsymbol{u}_k^\top$. The first row & column of A_1 are zero and it follows that $\boldsymbol{u}_2^\top$ is the second row of A_1, while $\boldsymbol{l}_2$ is its second column, scaled so that $L_{2,2} = 1$.

We can thus summarize the LU decomposition algorithm as follows: Set $A_0 := A$. For all $k = 1, 2, \ldots, n$ set $\boldsymbol{u}_k^\top$ to the kth row of A_{k-1} and $\boldsymbol{l}_k$ to the kth column of A_{k-1}, scaled so that $L_{k,k} = 1$. Set $A_k := A_{k-1} - \boldsymbol{l}_k\boldsymbol{u}_k^\top$ and increment k.

At each step k, the dominant cost is to form $\boldsymbol{l}_k\boldsymbol{u}_k^\top$. Since the first $k-1$ components of $\boldsymbol{l}_k$ and $\boldsymbol{u}_k$ are zero the cost of forming this rank-one matrix is $(n-k+1)^2$. Thus the total cost of the algorithm is $\sum_{k=1}^{n}(n-k+1)^2 = \sum_{j=1}^{n} j^2 = \mathcal{O}(n^3)$.

Relation to Gaussian elimination In Gaussian elimination, we perform a series of elementary row operations on A to transform it into an upper triangular matrix. Each elementary row operation consists in adding a multiple of the k'th row to the j'th row ($j > k$). One can easily show that such operations can be represented using unit lower triangular matrices. Thus Gaussian elimination can be written concisely as:

$$L_n L_{n-1} \ldots L_1 A = U$$

where each L_k is unit lower triangular and U is upper triangular. This gives $A = LU$ where $L = L_1^{-1} \ldots L_n^{-1}$ is unit lower triangular. In the algorithm described above, the matrix A_k is the matrix obtained after k steps of Gaussian elimination, except for the first $k-1$ rows and columns which are zero in A_k. The only difference between Gaussian elimination and LU is that Gaussian elimination is usually applied to a linear system $A\boldsymbol{x} = \boldsymbol{b}$ and the lower triangular matrices are not stored. One advantage of using LU decomposition is that it can be reused for different right-hand sides: in Gaussian elimination the solution for each new $\boldsymbol{b}$ would require $\mathcal{O}(n^3)$ computational operations, whereas with LU factorization $\mathcal{O}(n^3)$ operations are required for the initial factorization, but then the solution for each new $\boldsymbol{b}$ only requires just $\mathcal{O}(n^2)$ (forward/backward substitution).