1 21st of November 2018 — F. Poloni

In this lecture we address the problem of solving linear systems exactly.

Someone could observe that this subject has already been studied in the numerical linear algebra course, but we are interested in computing the solution to this problem quickly when the dimensions are large and the matrix A is sparse.

Since the complexity of Gauss method is cubic, this algorithm is unfeasible for large inputs.

Let us see some real life examples, where the matrices are large and sparse.

Local function on graphs: A **local function on graphs** is a function that depend on few nearby vertices. This kind of functions lead to a sparse adjacency matrix A, as can be observed in Figure 1.1;

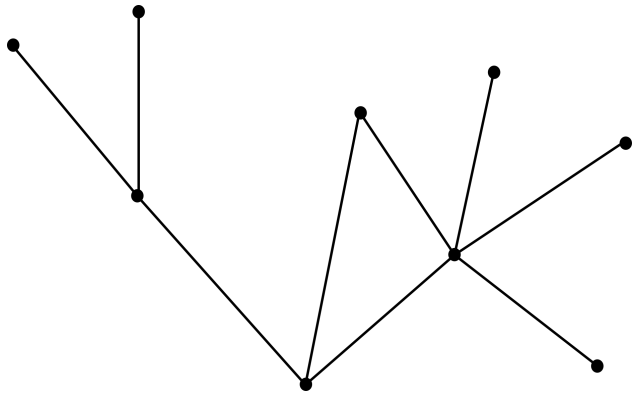

Figure 1.1: A local function on graph.

Images: Take an $m \times m$ image and blur it (each pixel is obtained as the average of its neighbours). $T : \mathcal{M}(m, \mathbb{R}) \to \mathcal{M}(m, \mathbb{R})$ such that $T(A)_{ij} = \frac{1}{9}(A_{i-1j} + A_{i-1j-1} + A_{i-1j+1} + A_{ij} + A_{ij-1} + A_{ij+1} + A_{i+1j} + A_{i+1j-1} + A_{i+1j+1})$. T may be written as a matrix that maps all the m images to a set of m blurred images and has the following shape $T \in \mathcal{M}(m^2, \mathbb{R})$ such that the (i, j)-th row of T has exactly 9 entries with value $\frac{1}{9}$ and all the others are 0. The non zero entries correspond to $A_{i-1j}, \ldots, A_{i+1j+1}$;

KKT systems: constrained optimization;

Engineering problem: To check stability of a bridge, it gets split into small cells. It can be proven that the stress on each of these cells corresponds to the force applied by the neighbours. In the end, this local phenomenon may be represented by a sparse matrix.

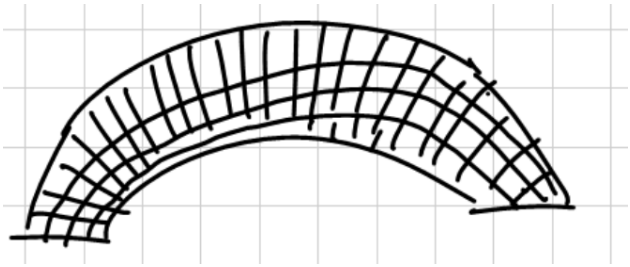

FIGURE 1.2: Graphic idea of a bridge partitioned into small blocks

1.1 Gaussian elimination and LU factorization

Gaussian elimination can be seen as a factorization: $A = LU$. The intuition is to proceed iteratively, multiplying each time for a new matrix, just like QR factorization.

Since the idea of Gauss elimination is to add multiples of row 1 to all the rows from 2 to n to kill off $A_{2:end,1}$ we have that:

STEP 1:

$$\begin{pmatrix} 1 & & & & \\ * & 1 & & & \\ * & & 1 & & \\ * & & & 1 & \\ * & & & & 1 \end{pmatrix} \begin{pmatrix} \circledast & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} \circledast & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{pmatrix},$$

Where the $\circledast$ is called **pivot**.

$$L_1 A = A_1$$
$$(L_1)_{k1} = -\frac{A_{k1}}{A_{11}}, \quad k = 2, 3, \ldots, m$$

STEP 2: we multiply for a matrix that has an "identity frame" and inside does the same L_1 was doing before.

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & * & 1 & & \\ & * & & 1 & \\ & * & & & 1 \end{pmatrix} \begin{pmatrix} * & \circledast & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & \circledast & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{pmatrix}$$

$$L_2 A_1 = A_2$$
$$(L_2)_{k2} = \frac{(A_1)_{k2}}{(A_1)_{22}}, \quad k = 3, \ldots, m$$

STEP 3: we go on and

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & * & 1 & \\ & & * & & 1 \end{pmatrix} \begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & \circledast & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & \circledast & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & * & * \end{pmatrix}$$

$$L_3 A_2 = A_3$$

$$(L_3)_{k3} = \frac{(A_2)_{k3}}{(A_2)_{33}}, \quad k = 4, \dots, m$$

STEP 4: one more operation

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & * & 1 \end{pmatrix} \begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & \circledast & * \\ 0 & 0 & 0 & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & \circledast & * \\ 0 & 0 & 0 & 0 & * \end{pmatrix}$$

$$L_4 A_3 = A_4$$

$$(L_4)_{k4} = \frac{(A_3)_{k4}}{(A_3)_{44}}, \quad k = 5, \dots, m$$

In the generic case we have $L_{m-1} L_{m-2} \dots L_1 A = U$, where U is upper triangular, or $A = \underbrace{L_1^{-1} L_2^{-1} \dots L_{m-1}^{-1}}_{=L} U$, with U upper triangular and L lower triangular.

Theorem 1.1. *Let $A \in \mathcal{M}(m, \mathbb{R})$ such that we do not encounter zero pivots in the algorithm. A admits a factorization $A = LU$, where L is lower triangular with ones on its diagonal, and U is upper triangular.*

Observation 1.1 (Stroke of luck)**.** *The product of the ${L_i}^{-1}$'s (denoted L) can be computed for free, since the following holds:*

$$\begin{bmatrix} 1 & & & & \\ -a_2 & 1 & & & \\ -a_3 & & 1 & & \\ -a_4 & & & 1 & \\ -a_5 & & & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & -b_3 & 1 & & \\ & -b_4 & & 1 & \\ & -b_5 & & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & -c_4 & 1 & \\ & & -c_5 & & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & -d_5 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & & & \\ a_2 & 1 & & & \\ a_3 & b_3 & 1 & & \\ a_4 & b_4 & c_4 & 1 & \\ a_5 & b_5 & c_5 & d_5 & 1 \end{bmatrix}$$

ALGORITHM 1.1 LU factorization, Matlab implementation.

```
1  function [L, U] = lu_factorization(A)
2    m = size(A, 1);
3    L = eye(m);
4    U = A;
5    for k = 1 : m - 1
6      % compute "multipliers"
7      L(k+1:end, k) = U(k+1:end, k) / U(k, k);
8      % update U
9      U(k+1:end, k) = 0;
10     U(k+1:end, k+1:end) = U(k+1:end, k+1:end) ...
11         - L(k+1:end, k) * U(k, k+1:end);
12   end
```

The idea behind the implementation of Algorithm 1.1 is shown in Figure 1.3.

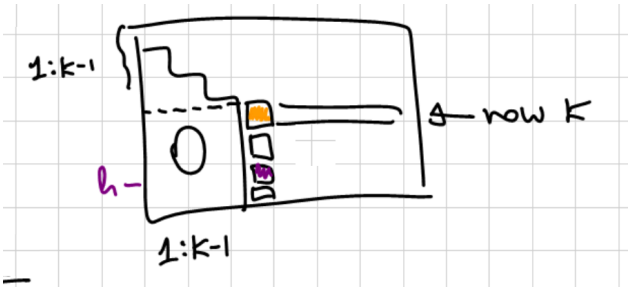

FIGURE 1.3: Assuming that we are at step k, we have that the h-th multiplier is expressed as $\frac{A_{hk}}{A_{kk}}$ and this muliplier goes to the right position in L.

Observation 1.2. *The computational complexity of this algorithm is concentrated at lines* 10, 11 *and the cost of this operation is* $O((m-1)^2 + (m-2)^2 + \cdots + 2^2 + 1)$. *So we have that for a dense matrix the computational complexity is* $\frac{2}{3}m^3 + O(m^2)$, *in other words half as much as QR factorization.*

Something on Matlab ...

Implementation of \ in Matlab: We consider important to remark how the operator \ is implemented in Matlab. It works using the LU factorization, making some checks and changing the algorithm as follows:

- upper/lower triangular systems: back-substitution ($O(n^2)$);
- non-triangular linear systems: LU with partial pivoting (then throw away the factors);
- symmetric and/or sparse systems: uses appropriate LU variants (will see in the following).
- non-square matrices: solves the system "in the least squares sense $\min_x \|Ax-b\|_2$".

Why isn't there any check on the orthogonality of the matrix? Because in that case $A^{-1} = A^T$, and hence it is easy to solve the system. Well, in order to find out that a matrix is orthogonal we need to compute $A^T A$, which is too costly.
Obs: LU without pivoting isn't stable, as shown in Section 1.1.1, so Matlab uses pivoting.

missing

1.1.1 Stability of LU

A downside of this approach is that it's not numerically stable. The intuition is that the condition is bad whenever the matrix A has a very small pivot.

Let us see an example:

Example 1.1.

$$A = \begin{bmatrix} 10^{-30} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10^{30} & 1 \end{bmatrix} \begin{bmatrix} 10^{-30} & 1 \\ 0 & 1 - 10^{30} \end{bmatrix}$$

In this case the LU factorization produces L, U with norm much larger than $\|A\|$.

Luckily, it's easy to circumvent this issue multiplying L_is by some permutation matrices (which swap rows in order to keep "large" pivots), as follows

$$L_{m-1}P_{m-1}\dots L_2P_2L_1P_1A = U$$

Observation 1.3. *Thanks to another "stroke of luck" we can reorder those factors:*

$$L_{m-1}P_{m-1}\dots L_2P_2L_1P_1 = \widehat{L}_{m-1}\widehat{L}_{m-2}\dots\widehat{L}_1P_{m-1}P_{m-2}\dots P_1$$

where $\widehat{L}_i$ have the same structure as the L_i.

We can now introduce the following:

Theorem 1.2. *Let $A \in \mathcal{M}(m, \mathbb{R})$. A admits a factorization $A = PLU$, where P is a permutation matrix, L is lower triangular with ones on its diagonal, and U is upper triangular.*

What about the stability of this improvement to LU factorization, called **LU with partial pivoting**?

It's not stable at all, in the worst case it may happen that $\frac{\|U\|}{\|A\|}$ may grow as $\approx 2^m$, although matrices for which this happens are very rare.

1.1.2 Gaussian elimination on sparse matrices

Given a sparse matrix :

$$A = \begin{pmatrix} * & * & & & * & * & & * & \\ & * & & * & & * & * & & \\ * & & * & * & * & & * & & \\ * & & & * & & & * & & \\ & * & & & & * & & * & * \\ * & & & * & & * & * & & \\ * & * & & & & & * & & * \\ * & * & * & * & & & & & \\ * & & * & & & * & & & * \end{pmatrix}$$

Gaussian elimination causes some fill-in, due to the sum of a multiple of the first row, which has non zero entries in different positions:

$$\begin{pmatrix} * & * & & & * & * & & * & \\ & * & & * & & * & * & & \\ * & & * & * & * & & * & & \\ * & & & * & & & * & & \\ & * & & & & * & & * & * \\ * & & & * & & * & * & & \\ * & * & & & & & * & & * \\ * & * & * & * & & & & & \\ * & & * & & & * & & & * \end{pmatrix} \rightarrow \begin{pmatrix} * & * & & & * & * & & * & \\ & * & & * & & * & * & & \\ 0 & * & * & * & * & * & * & * & \\ 0 & * & & * & * & * & * & * & \\ & * & & & & * & & * & * \\ 0 & * & & * & * & * & * & * & \\ 0 & * & & & * & * & * & * & * \\ 0 & * & * & * & * & * & & * & \\ 0 & * & * & & * & * & & * & * \end{pmatrix}$$

We may observe that the computational complexity of sparse LU is linear in the number of non zero entries of the final matrix, obtained by the algorithm, which is possibly much larger than the number of non zeros in A.

How to circumvent this problem? At each step we may use as pivot row the most sparse one. This computation may be done in a more sofisticate way, considering the "relative" position of non zeros between couples of rows.

Because of this trade-off the choice is made in relation to the needs of the implementation. We won't study any algorithm that deals with sparse matrices, since they are very complicated and make use of heuristics.

There are some lucky cases in which the fill-in is almost none, for example a matrix that only has 5 diagonals which entries are different from 0 (called **tridiagonal**). In this particular case L is tridiagonal and lower triangular and U is tridiagonal and upper triangular, as shown below:

$$A = \begin{pmatrix} * & * & * & 0 & 0 & 0 & \cdots & 0 \\ * & * & * & * & 0 & 0 & \cdots & 0 \\ * & * & * & * & * & 0 & \cdots & 0 \\ 0 & * & * & * & * & * & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & * & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * & * \end{pmatrix}$$

$$L = \begin{pmatrix} * & 0 & 0 & 0 & 0 & \cdots & 0 \\ * & * & 0 & 0 & 0 & \cdots & 0 \\ * & * & * & 0 & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & * & * & * & 0 & 0 \\ & & & * & * & * & 0 \\ & & & & * & * & * \end{pmatrix} U = \begin{pmatrix} * & * & * & & & & & \\ 0 & * & * & * & & & & \\ 0 & 0 & * & * & * & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & 0 & * & * & * & \\ 0 & \cdots & 0 & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & 0 & * & * \end{pmatrix}$$

Observation 1.4. *We should remark that if we are interested in high-performance computing we need to pay attention to the blocking, because we go from vector-vector operation to matrix-matrix operation and some of these operations may be performed more efficiently. Parallel/multithreaded implementations are available by means of parallel libraries for Matlab.*