factorisation of special matrices LU , LL^T and LDL^T

أ.اسماء ابوبكر علي عون

Soma126@googlemail.com

Abstract

In this research the main points of the factorisation of special matrices LU, matrices, and how to computed, matrices and how it works by MATLAB, will be discussed. And how the special matrices works on a number of examples with useful results will illustrated. And finally a summary will be confirmed

Introduction

In numerical analysis and linear algebra, the lower - upper decomposition (LU) factors or matrix factorization as the product of the lower triangular matrix and the upper triangular matrix. The product sometimes includes a switching matrix as well. The LU decomposition can be viewed as a matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using LU analysis, which is also an essential step when inverting a matrix or calculating a matrix determinant. LU analysis was introduced by Polish mathematician Tadeusz Banachiewicz in 1938.

Factorisation of special matrices LU , LL^T and LDL^T

Why factorise?

Matrix factors provide an easy way to solve Ax = b for x. For example if A can be factored

into lower and upper triangular matrices LU = A, then x is found from

1. Find L,U such that LU = A.

2. Solve $L\underline{y} = \underline{b}$ for $\underline{y}$.

3. Solve $U\underline{x} = y$ for x.

Step 1 is expensive — roughly $2n^3/3$ flops for full matrices. Steps 2 and 3 are cheap (n^2 flops for full matrices) and can be repeated without recalculating L,U. Factorisation is then particularly useful when the same matrix is to be used with a collection of different right hand sides. For example find $\underline{x}^{(1)}$, $\underline{x}^{(2)}$,……from.

$$A_X{}^{(j+1)} = x^{(j)} \qquad for\, j = 0,1,\ldots$$

Given $\underline{x}^{(0)}$, or from

$$A_X{}^{(j)} = b^{(j)} \qquad for\, j = 1,2,\ldots$$

Given the $\underline{b}^{(j)}$s.

General, full matrices

As we know the LU algorithm that factors an n × n matrix A lower triangular matrix L and upper triangular matrix U such that A = LU. The flops count for this factorisation is roughly $2n^3/3$.

Mat-lab does not produce LU factors exactly as above, but it does produce an upper

Triangular matrix U and a permuted lower triangular matrix L as follows:

>> [L U] = lu(A);

Assuming that matrix A has already been input. Nevertheless, the result is that LU = A

(Apart from rounding errors). Given the factors, the solution of Ax = $\underline{b}$ is:

>> y = L\b;

>> x = U\y;

And the flops count for these two solves is identical to that for standard triangular matrices --i.e. n^2 flops for each solve. Note that y is just a convenient intermediate result that plays no further part. There is much more detail about the routine lu in the Mat- lab documentation.

Symmetric, full matrices — LL^T factors

The LU factorisation above can be simplified considerably if the matrix A to be factored

is symmetric. There are two ways to do this, we will consider the Cholesky factorisation

LL^T first. Here L is lower triangular rather than unit lower triangular. This factorisation

is suitable for symmetric, positive definite matrices.

First consider multiplying lower triangular (not unit lower triangular) matrix L and its

Transpose L^T .together. Take the 3 × 3 case as an illustration:

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} = \begin{pmatrix} l^2{}_{11} & l_{11}l_{21} & l_{11}l_{31} \\ l_{21}\ l_{11} & l_{21}^2 + l^2{}_{22} & l_{21}l_{31} + l_{22}l_{32} \\ l_{31}l_{11} & l_{21}l_{31} + l_{22}l_{32} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

The result LL^T is symmetric. Now set LLT = A for symmetric A .

$$\begin{pmatrix} l^2{}_{11} & l_{11}l_{21} & l_{11}l_{31} \\ l_{21}\ l_{11} & l_{21}^2+l^2{}_{22} & l_{21}l_{31}+l_{22}l_{32} \\ l_{31}l_{11} & l_{21}l_{31}+l_{22}l_{32} & l_{31}^2+l_{32}^2+l_{33}^2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

One can find all the element of L in turn starting from the top the top row and working down.

One only need consider about half of the equations since the others are just copies of them.

Two orders will work:

$$\begin{pmatrix} 1 & * & * \\ 2 & 4 & * \\ 3 & 5 & 6 \end{pmatrix} and \begin{pmatrix} 1 & * & * \\ 2 & 3 & * \\ 4 & 5 & 6 \end{pmatrix}$$

i.e. working down columns or along rows.

Use the "row" version. Row 1 has 1 equation:

$$l^2{}_{11} = a_{11} \implies l_{11} = \sqrt{a_{11}}$$

Row 2 has 2 equations:

$$l_{21}l_{11} = a_{21} \Longrightarrow\ l_{21} = \frac{a_{21}}{l_{11}}$$

$$l_{21}^2 + l_{22}^2 = a_{22} \Longrightarrow l_{22} = \sqrt{a_{22} - l_{21}^2}$$

And row 3 has 3 equations:

$$l_{31}l_{11} = a_{31} \Longrightarrow\ l_{31} = \frac{a_{31}}{l_{11}}$$

$$l_{31}l_{21} + l_{32}l_{22} = a_{32} \implies l_{32} = \frac{a_{32} - l_{31}l_{21}}{l_{22}}$$

$$l_{31}^2 + l_{32}^2 + l_{33}^2 = a_{33} \implies l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2}.$$

At each stage the calculation is rearranged to find a new element of L in terms of elements
of A and previously calculated elements of L. Clearly this process will fail if any of the Divisors L_{11},L_{22},... are zero or if the numbers under the square roots are negative. The General n × n case is given in the following algorithm.

Algorithm 2.1 Full Symmetric Matrix LL^T Factorisation
Input: n and the matrix A.

Output: The matrix L.

1: $l_{1,1} = \sqrt{a_{1,1.}}$

2: for $i = 1; i - 1\ do$

3: $for\ j = 1 : i - 1\ do$

4: $l_{i.j} = \frac{1}{l_{j.j}}(a_{i.j} - \sum_{k=1}^{j=1} l_{i,k} l_{j,k})$

5: end for

6: $l_{i,j} = \sqrt{a_{i,j} - \sum_{k=1}^{i=1} l_{i,k}^2}$

7: end for

Note that the summation convention is that $\sum_{k=p}^{q} w_k = 0$ when $q < p$ so that the sum so that the sum $\sum_{k=1}^{j=1}$ is zero when $j < 2$. This is also used in the MATLAB code below . It simply makes the the coding more compact .

The flops count for this algorithm is $n^3/3$ which is one half of the standard LU factorisation. Once the factor L is calculated, then the equation $A\underline{x}=\underline{b}$ (with $A = LL^T$)

Can be solved as usual form :

- Solve $L\underline{y}=\underline{b}$ for $\underline{y}$.
- Solve $L^T\underline{x}=\underline{y}$ for $\underline{x}$.

Each of these solves is triangular and so takes n^2flops.

The algorithm above appears to make sense for any real, symmetric matrix. However,

If we have a real, symmetric matrix A and do arithmetic exactly, then the algorithm will

Do the following:

• It will work (i.e. not divide by zero or take the square root of a negative number)

when A is positive definite.

• It will probably fail when A is positive semi-definite, but in very special cases it will

work leaving $l_{nn} = 0$ and all the other $d_{jj} > 0$, j = 1 : n − 1.

• It will fail (i.e. divide by zero or take the square root of a negative number) otherwise.

A mathematical statement covering the above is that the real-valued matrix A is symmetric.
Positive definite if and only if it can be factored into the form $LL^T = A$, where L is a real,
lower triangular matrix with nonzero diagonal entries.
When we use standard floating point computer arithmetic (i.e. not exact), the results
get a bit fuzzier, but we still need the matrix to be positive definite to get useful results.
As an example that works, the following Matlab code generates a symmetric 4 × 4
matrix A, a EDU>> n = 4;

```
A = pascal (n);     % generate a test matrix
L = zeros (n,n);    % set aside storage for result
L(1,1) = sqrt(A(1,1)); % step 1
For i=2:n % step 2
for j=1:i-1
L(i,j) = (A(i,j) - sum(L(i,1:j-1).*L(j,1:j-1)))/L(j,j); % step 3
End
L(i,i) = sqrt(A(i,i) - sum(L(i,1:i-1).^2)); % step 4
End
A, L, norm(L*L'-A)    % display results
```

A =

1	1	1	1
1	2	3	4
1	3	6	10
1	4	10	20

L =

1	0	0	0
1	1	0	0
1	2	1	0
1	3	3	1

ans =

0

So it does the correct thing. The built-in command chol does the LL^T Factorisation rather more compactly, but note that chol actually outputs L^T Rather than L.

2.4 Symmetric, full matrices — LDL^T factors

This factorisation process is used for symmetric, full matrices which are not necessarily

Positive definite. The Cholesky solution method fails when the matrix is not positive

Definite, e.g. by coming up with the square root of a negative number in Step 1 or 6, or

Dividing by zero in Step 4. The LDL^T factorisation below is much more flexible, but still

not infallible!

In the LDL^T factorisation, L is unit lower triangular and D is diagonal. Take the 3× 3

case as an illustration:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}, \quad D = \begin{pmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{pmatrix}$$

And so:

$$LDL^T = \begin{pmatrix} d_1 & d_1 l_{21} & d_1 l_{31} \\ d_1 l_{21} & d_1 l^2{}_{21} + d_2 & d_1 l_{21} l_{31} + d_2 l_{32} \\ d_1 l_{31} & d_1 l_{21} l_{31} + d_2 l_{32} & d_1 l_{31}^2 + d_2 l_{32}^2 + d_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = A$$

As in the LL^T case we can find all the elements of L and D in turn starting from the top

row and working down. We only need consider about half of the equations since the others are just copies of them. As before, two different orders for solving the equations will work:

$$\begin{pmatrix} 1 & * & * \\ 2 & 4 & * \\ 3 & 5 & 6 \end{pmatrix} \quad and \quad \begin{pmatrix} 1 & * & * \\ 2 & 3 & * \\ 4 & 5 & 6 \end{pmatrix}$$

i.e. working down columns or along rows. The general n × n case is given in the following

algorithm below.

Algorithm 2.2 Full Symmetric Matrix LDL^T Factorisation
input: n and the matrix A.

Output: The matrices L and D.
1: $d_1 = a_{1,1}, l_{11} = 1.$

2:$for\ i = 2: n\ do$

3: $for\ j = 1: i - 1\ do$

4: $l_{i,j} = \frac{1}{d_j}(a_{i,j} - \sum_{k=1}^{j=1} d_k l_{i,k} l_{j,k})$

5: $end\ for$

6: $d_i = a_{i,j} - \sum_{k=1}^{i=1} d_k l^2{}_{i,k}\ , \ell_{i,i} = 1.$

7: $end\ for$

Watch out for the summation convention again in Step 4, giving sum zero when j < 2. We should check if $d_j = 0$ in step 6 and stop if it is, otherwise we'll end up dividing by zero.

The flops count for LDL^T factorisation is again roughly $n^3/3$, almost exactly the

same as for the LL^T factorisation. Once the factors L,D are calculated, then the equation

A$\underline{x}$ = $\underline{b}$ (with A = LDL^T) can be solved as usual from:

• Solve L$\underline{y}$ = $\underline{b}$ for $\underline{y}$,

• solve D$\underline{z}$ = $\underline{y}$ for $\underline{z}$, (i.e. $z_i = y_i/d_i$ for i = 1 : n)

• solve L^T $\underline{x}$ = $\underline{z}$ for $\underline{x}$.

This involves two triangular solves (each about n^2 flops) and one diagonal solve (n flops)

So takes about $2n^2$ flops in total for large n.

Example 2.3 Try the LL^T and LDL^T factorisations on the matrix

$$\begin{pmatrix} 1 & 2 & 0 \\ 2 & 1 & 3 \\ 0 & 3 & 4 \end{pmatrix}$$

The LL^T algorithm fails when calculating ℓ_{22}

$\ell_{11} = \sqrt{a_{11}} \Rightarrow \ell_{11} = 1$

$\ell_{21} = a_{21}/\ell_{11} \Rightarrow \ell_{21} = 2/1 = 2$

$\ell_{22} = \sqrt{a_{22} - l_{21}^2} \Rightarrow l_{22} = \sqrt{1-4} = i\sqrt{3}$

Which is no use when dealing with real equations. On the other hand, LDL^T produces

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \quad , \quad D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

With no problem.

The LU factorisation for general (i.e. not symmetric) tridiagonal matrices can be cut down to be much simpler and cheaper than the full method. In fact we need only calculate three vectors rather than two large triangular matrices. Sometimes this is called the Crout or Thomas algorithm.

Consider a 3 × 3 example:

$$LD = \begin{pmatrix} 1 & 0 & 0 \\ l_2 & 1 & 0 \\ 0 & l_3 & 1 \end{pmatrix} \begin{pmatrix} u_1 & w_1 & 0 \\ 0 & u_2 & w_2 \\ 0 & 0 & u_3 \end{pmatrix}$$

$$= \begin{pmatrix} u_1 & w_1 & 0 \\ l_2 u_1 & l_2 w_1 + u_2 & w_2 \\ 0 & l_3 u_2 & l_3 w_2 + u_3 \end{pmatrix} \quad , \quad (1)$$

The matrix L is specified by vector $\underline{\ell}$ and the matrix U by vectors $\underline{u}$,$\underline{w}$. The general n × n

Formula for the factors LU of:

$$A \equiv \begin{pmatrix} q_1 & & r_1 & & \\ & \ddots & \ddots & \ddots & \\ & p_j & q_j & r_j & \\ & \ddots & \ddots & \ddots & \\ & & p_n & & q_n \end{pmatrix} \qquad (2)$$

is reasonably simple as well:

Row 1:

$$u_1 = q_1 \quad , \quad w_1 = r_1$$

Row $j = 2{:}n-1$

$$l_j u_{j-1} = p_j \quad , \quad l_j w_{j-1} + u_j = q_j \quad , \quad w_j = r_j$$

Row n

$$l_n u_{n-1} = p_n \quad , \quad l_n w_{n-1} + u_n = q_n \quad .$$

So the unknown coefficients ℓ_j ,u_j ,w_j can be found from the algorithm below:
Algorithm 2.4 Factorise Tridiagonal A
Input: n and the three vectors p,q,r that form the sub-, main- and super-diagonals of

Matrix A as in (2).

Output: The elements of L and U in vectors ℓ,u,w as in (1).
1: $u_1 = q_1$, $w_1 = r_1$.

2: for j = 2:n−1 do

3: $\ell_j = p_j/u_{j-1}$, $u_j = q_j - \ell_j w_{j-1}$, $w_j = r_j$.

4: end for

5: $\ell_n = p_n/u_{n-1}$, $u_n = q_n - \ell_n w_{n-1}$.

Solution of Ax = b is easy once the tridiagonal factors have been calculated.
Algorithm 2.5 Solve Tridiagonal Ax = b

Input: n, tridiagonal matrix A, right hand side b.

Output: The solution x.

1: Find the vectors ℓ,u,w using Algorithm 2.4.

2: $y_1 = b_1$

3: for i = 2:n do

4: $y_i = b_i - \ell_i y_{i-1}$.

5: end for

6: $x_n = y_n/u_n$

7: for i = n − 1 : −1 : 1 do

8: $x_i = (y_i - w_i x_{i+1})/u_i$.

9: end for

We should of course check for division by zero in these algorithms since they fail in that case. However when speed is the objective it may be enough to let the code fail

and investigate why after the event. (That probably doesn't match any acceptable coding standard, so be careful!)

Fortunately the algorithm works without excessive growth of rounding errors in many cases including when A is: positive definite; strictly diagonally dominant (by

rows or columns); and symmetric positive definite.

The flops count for the factorisation process is reasonably straightforward. Steps 2–

4 of Algorithm 2.4 have one each of ÷, ×,− repeated n − 2 times for a total of 3n − 6

Flops. Adding in Step 5, the final total for Algorithm 2.4 is 3n − 3 flops. This is of course

Significantly less than the $2n^3/3$ flops for LU decomposition of full matrices. The $L\underline{y} = \underline{b}$

and $U\underline{x} = \underline{y}$ solves are also relatively cheap: Steps 3–5 of Algorithm 2.5 require 2n−2 flops

And Steps 6–9 together take 3n − 2 flops. The total for a complete tridiagonal factor and

Solve is thus 8n − 7 flops.

Because tridiagonal systems come up so often, many efforts have been made to design

Fast code for them. Sometimes they are even hand coded in low level machine language to speed them up. The problem is that the algorithm cannot be vectorised or parallelised

Easily and efficiently. Some attempts at tackling vectorisation and parallelisation of this

Problem can be found in [4, 6].

Conclusion

In this research, we tried to discussion a lot of ideas about factorisation of special matrices LU, and we tried to the answer to the main question,. On the other hand, we did a lot of examples to show it.

By proofing factorisation of special matrices LU by a using MATLAB as we have done as a part of examples and theorems.

Finally, these examples and theorems useful to understand concourse using the MATLAB,. In the end, the factorisation of special matrices LU very useful , thought out, comeback ranking for page ,but it is very important to keep in mind that it can be developed further.

References

[1] R. L. Burden and J. D. Faires. Numerical Analysis. Brooks Cole, 7 edition, 2001.
[2] G. Golub and C. F. van Loan. Matrix Computations. John Hopkins University Press, 3 edition, 1995.
[3] D. J. Higham and N. J. Higham. MATLAB Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.
[4] R. W. Hockney and C. R. Jesshope. Parallel Computers 2. Adam Hilger, 1988.
[5] C. Moler J. R. Gilbert and R. Schreiber. Sparse matrices in matlab: design and implementation. SIAM J Matrix Anal. Appl., 13:333–356, 1992.
[6] C. H. Walshaw. Diagonal dominance in the parallel partition method for tridiagonal systems. SIAM J Matrix Anal. Appl., 16:1086–1099, 1995.