LU decomposition

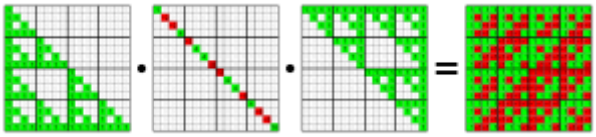
In numerical analysis and linear algebra, **LU decomposition** (where 'LU' stands for 'lower upper', and also called **LU factorization**) factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. The LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using the LU decomposition, and it is also a key step when inverting a matrix, or computing the determinant of a matrix. The LU decomposition was introduced by mathematician Tadeusz Banachiewicz in 1938.[1]

Contents

Definitions

Let A be a square matrix. An **LU factorization** refers to the factorization of A, with proper row and/or column orderings or permutations, into two factors, a **unit** lower triangular matrix L and an upper triangular matrix U,

$$A = LU,$$



LDU decomposition of aWalsh matrix

In the lower triangular matrix all elements above the diagonal are zero, in the upper triangular matrix, all the elements below the diagonal are zero. For example, for a 3-by-3 matrix A, its LU decomposition looks like this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Without a proper ordering or permutations in the matrix, the factorization may fail to materialize. For example, it is easy to verify (by expanding the matrix multiplication) that $a_{11} = l_{11}u_{11}$. If $a_{11} = 0$, then at least one of l_{11} and u_{11} has to be zero, which implies either L or U is singular. This is impossible if A is nonsingular (invertible). This is a procedural problem. It can be removed by simply reordering the rows of A so that the first element of the permuted matrix is nonzero. The same problem in subsequent factorization steps can be removed the same way; see the basic procedure below

LU factorization with Partial Pivoting

It turns out that a proper permutation in rows (or columns) is sufficient for the LU factorization. The **LU factorization with Partial Pivoting** (LUP) refers often to the LU factorization with row permutations only

$$PA = LU,$$

where L and U are again lower and upper triangular matrices, and P is a permutation matrix which, when left-multiplied to A, reorders the rows of A. It turns out that all square matrices can be factorized in this form,[2] and the factorization is numerically stable in practice.[3] This makes LUP decomposition a useful technique in practice.

LU factorization with full pivoting

An **LU factorization with full pivoting** involves both row and column permutations,

$$PAQ = LU,$$

where L, U and P are defined as before, and Q is a permutation matrix that reorders the columns of A.[4]

LDU decomposition

An **LDU decomposition** is a decomposition of the form

$$A = LDU,$$

where D is a diagonal matrix and L and U are *unit* triangular matrices, meaning that all the entries on the diagonals of L and U are one.

Above we required that A be a square matrix, but these decompositions can all be generalized to rectangular matrices as well. In that case, L and D are square matrices both of which have the same number of rows as A, and U has exactly the same dimensions as A. *Upper triangular* should be interpreted as having only zero entries below the main diagonal, which starts at the upper left corner.

Example

We factorize the following 2-by-2 matrix:

$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}.$$

One way to find the LU decomposition of this simple matrix would be to simply solve the linear equations by inspection. Expanding the matrix multiplication gives

$$\begin{aligned}
l_{11} \cdot u_{11} + 0 \cdot 0 &= 4 \\
l_{11} \cdot u_{12} + 0 \cdot u_{22} &= 3 \\
l_{21} \cdot u_{11} + l_{22} \cdot 0 &= 6 \\
l_{21} \cdot u_{12} + l_{22} \cdot u_{22} &= 3.
\end{aligned}$$

This system of equations is underdetermined. In this case any two non-zero elements of L and U matrices are parameters of the solution and can be set arbitrarily to any non-zero value. Therefore, to find the unique LU decomposition, it is necessary to put some restriction on L and U matrices. For example, we can conveniently require the lower triangular matrix L to be a unit triangular matrix (i.e. set all the entries of its main diagonal to ones). Then the system of equations has the following solution:

$$\begin{aligned}
l_{21} &= 1.5 \\
u_{11} &= 4 \\
u_{12} &= 3 \\
u_{22} &= -1.5
\end{aligned}$$

Substituting these values into the LU decomposition above yields

$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix}.$$

Existence and uniqueness

Square matrices

Any square matrix A admits an *LUP* factorization.[2] If A is invertible, then it admits an *LU* (or *LDU*) factorization if and only if all its leading principal minors are nonzero.[5] If A is a singular matrix of rank k, then it admits an *LU* factorization if the first k leading principal minors are nonzero, although the converse is not true.[6]

If a square, invertible matrix has an *LDU* factorization with all diagonal entries of L and U equal to 1, then the factorization is unique.[5] In that case, the *LU* factorization is also unique if we require that the diagonal of L (or U) consists of ones.

Symmetric positive definite matrices

If A is a symmetric (or Hermitian, if A is complex) positive definite matrix, we can arrange matters so that U is the conjugate transpose of L. That is, we can write A as

$$A = LL^*.$$

This decomposition is called the Cholesky decomposition. The Cholesky decomposition always exists and is unique — provided the matrix is positive definite. Furthermore, computing the Cholesky decomposition is more efficient and numerically more stable than computing some other LU decompositions.

General matrices

For a (not necessarily invertible) matrix over any field, the exact necessary and sufficient conditions under which it has an LU factorization are known. The conditions are expressed in terms of the ranks of certain submatrices. The Gaussian elimination algorithm for obtaining LU decomposition has also been extended to this most general case.[7]

Algorithms

The LU decomposition is basically a modified form of Gaussian elimination. We transform the matrix A into an upper triangular matrix U by eliminating the entries below the main diagonal. The Doolittle algorithm does the elimination column by column starting from the left, by multiplying A to the left with atomic lower triangular matrices. It results in a *unit lower triangular* matrix and an upper triangular matrix. The Crout algorithm is slightly different and constructs a lower triangular matrix and a *unit upper triangular* matrix.

Computing the LU decomposition using either of these algorithms requires $2n^3 / 3$ floating point operations, ignoring lower order terms. Partial pivoting adds only a quadratic term; this is not the case for full pivoting.[8]

Closed formula

When an LDU factorization exists and is unique there is a closed (explicit) formula for the elements of L, D, and U in terms of ratios of determinants of certain submatrices of the original matrix A.[9] In particular, $D_1 = A_{1,1}$ and for $i = 2, \ldots, n$, D_i is the ratio of the i^{th} principal submatrix to the $(i-1)^{th}$ principal submatrix. Computation of the determinants is computationally expensive and so this explicit formula is not used in practice.

Doolittle algorithm

Given an $N \times N$ matrix

$$A = (a_{i,j})_{1 \le i,j \le N}$$

we define

$$A^{(0)} := A$$

We eliminate the matrix elements below the main diagonal in the n-th column of $A^{(n-1)}$ by adding to the i-th row of this matrix the n-th row multiplied by

$$-l_{i,n} := -\frac{a_{i,n}^{(n-1)}}{a_{n,n}^{(n-1)}}$$

for $i = n+1, \ldots, N$. This can be done by multiplying $A^{(n-1)}$ to the left with the lower triangular matrix

$$L_n = \begin{pmatrix} 1 & 0 & & \cdots & 0 \\ 0 & \ddots & \ddots & & \\ & & 1 & & \\ \vdots & & -l_{n+1,n} & & \vdots \\ & & \vdots & \ddots & 0 \\ 0 & & -l_{N,n} & & 1 \end{pmatrix}.$$

We set

$$A^{(n)} := L_n A^{(n-1)}.$$

After $N-1$ steps, we eliminated all the matrix elements below the main diagonal, so we obtain an upper triangular matrix $A^{(N-1)}$. We find the decomposition

$$A = L_1^{-1} L_1 A^{(0)} = L_1^{-1} A^{(1)} = L_1^{-1} L_2^{-1} L_2 A^{(1)} = L_1^{-1} L_2^{-1} A^{(2)} = \cdots = L_1^{-1} \dots L_{N-1}^{-1} A^{(N-1)}.$$

Denote the upper triangular matrix $A^{(N-1)}$ by U, and $L = L_1^{-1} \dots L_{N-1}^{-1}$. Because the inverse of a lower triangular matrix L_n is again a lower triangular matrix, and the multiplication of two lower triangular matrices is again a lower triangular matrix, it follows that L is a lower triangular matrix. Moreover, it can be seen that

$$L = \begin{pmatrix} 1 & 0 & \dots & & 0 \\ l_{2,1} & \ddots & \ddots & & \\ & & 1 & & \\ \vdots & & l_{n+1,n} & \ddots & \vdots \\ & & \vdots & 1 & 0 \\ l_{N,1} & & l_{N,n} & l_{N,N-1} & 1 \end{pmatrix}.$$

We obtain $A = LU$.

It is clear that in order for this algorithm to work, one needs to have $a_{n,n}^{(n-1)} \neq 0$ at each step (see the definition of $l_{i,n}$). If this assumption fails at some point, one needs to interchange n-th row with another row below it before continuing. This is why the LU decomposition in general looks like $P^{-1} A = LU$.

Crout and LUP algorithms

The LUP decomposition algorithm by Cormen et al. generalizes Crout matrix decomposition. It can be described as follows.

1. If A has a nonzero entry in its first row, then take a permutation matrix P_1 such that AP_1 has a nonzero entry in its upper left corner. Otherwise, take for P_1 the identity matrix. Let $A_1 = AP_1$.
2. Let A_2 be the matrix that one gets from A_1 by deleting both the first row and the first column. Decompose $A_2 = L_2 U_2 P_2$ recursively. Make L from L_2 by first adding a zero row above and then adding the first column of A_1 at the left.
3. Make U_3 from U_2 by first adding a zero row above and a zero column at the left and then replacing the upper left entry (which is 0 at this point) by 1. Make P_3 from P_2 in a similar manner and define $A_3 = A_1 / P_3 = AP_1 / P_3$. Let P be the inverse of P_1 / P_3.
4. At this point, A_3 is the same as LU_3, except (possibly) at the first row. If the first row of A is zero, then $A_3 = LU_3$, since both have first row zero, and $A = LU_3 P$ follows, as desired. Otherwise, A_3 and LU_3 have the same nonzero entry in the upper left corner, and $A_3 = LU_3 U_1$ for some upper triangular square matrix U_1 with ones on the diagonal (U_1 clears entries of LU_3 and adds entries of A_3 by way of the upper left corner). Now $A = LU_3 U_1 P$ is a decomposition of the desired form.

Randomized Algorithm

It is possible to find a low rank approximation to the LU decomposition using a randomized algorithm. Given an input matrix A and a desired low rank k, the randomized LU returns permutation matrices P, Q and lower/upper trapezoidal matrices L, U of size $m \times k$ and $k \times n$ respectively, such that with high probability $\|PAQ - LU\|_2 \leq C\sigma_{k+1}$, where C is a constant that depends on the parameters of the algorithm and σ_{k+1} is the $(k+1)$th singular value of the input matrix A.[10]

Theoretical complexity

If two matrices of order n can be multiplied in time $M(n)$, where $M(n) \geq n^a$ for some $a>2$, then the LU decomposition can be computed in time $O(M(n))$.[11] This means, for example, that an $O(n^{2.376})$ algorithm exists based on the Coppersmith–Winograd algorithm.

Sparse matrix decomposition

Special algorithms have been developed for factorizing large sparse matrices. These algorithms attempt to find sparse factors L and U. Ideally, the cost of computation is determined by the number of nonzero entries, rather than by the size of the matrix.

These algorithms use the freedom to exchange rows and columns to minimize fill-in (entries which change from an initial zero to a non-zero value during the execution of an algorithm).

General treatment of orderings that minimize fill-in can be addressed using graph theory. O(n^3)

Applications

Solving linear equations

Given a system of linear equations in matrix form

$$\boldsymbol{Ax} = \boldsymbol{b},$$

we want to solve the equation for x given A and b. Suppose we have already obtained the LUP decomposition of A such that $\boldsymbol{PA} = \boldsymbol{LU}$, so $\boldsymbol{LUx} = \boldsymbol{Pb}$.

In this case the solution is done in two logical steps:

1. First, we solve the equation $\boldsymbol{Ly} = \boldsymbol{Pb}$ for y;
2. Second, we solve the equation $\boldsymbol{Ux} = \boldsymbol{y}$ for x.

Note that in both cases we are dealing with triangular matrices (L and U) which can be solved directly by forward and backward substitution without using the Gaussian elimination process (however we do need this process or equivalent to compute the LU decomposition itself).

The above procedure can be repeatedly applied to solve the equation multiple times for different b. In this case it is faster (and more convenient) to do an LU decomposition of the matrix A once and then solve the triangular matrices for the different b, rather than using Gaussian elimination each time. The matrices L and U could be thought to have "encoded" the Gaussian elimination process.

The cost of solving a system of linear equations is approximately $\frac{2}{3}\boldsymbol{n}^3$ floating point operations if the matrix $\boldsymbol{A}$ has size $\boldsymbol{n}$. This makes it twice as fast as algorithms based on the QR decomposition, which costs about $\frac{4}{3}\boldsymbol{n}^3$ floating point operations when Householder reflections are used. For this reason, the LU decomposition is usually preferred.[12]

Inverting a matrix

When solving systems of equations, b is usually treated as a vector with a length equal to the height of matrix A. Instead of vector b, we have matrix B, where B is an n-by-p matrix, so that we are trying to find a matrix X (also a n-by-p matrix):

$$\boldsymbol{AX} = \boldsymbol{LUX} = \boldsymbol{B}.$$

We can use the same algorithm presented earlier to solve for each column of matrix X. Now suppose that B is the identity matrix of size n. It would follow that the result X must be the inverse of A.[13] An implementation of this methodology in the C programming language can be found here.

Computing the determinant

Given the LUP decomposition $\boldsymbol{A} = \boldsymbol{P}^{-1}\boldsymbol{LU}$ of a square matrix A, the determinant of A can be computed straightforwardly as

$$\det(A) = \det(P^{-1})\det(L)\det(U) = (-1)^S \left(\prod_{i=1}^{n} l_{ii}\right)\left(\prod_{i=1}^{n} u_{ii}\right).$$

The second equation follows from the fact that the determinant of a triangular matrix is simply the product of its diagonal entries, and that the determinant of a permutation matrix is equal to $(-1)^S$ where S is the number of row exchanges in the decomposition.

In the case of LU decomposition with full pivoting, $\det(A)$ also equals the right-hand side of the above equation, if we let S be the total number of row and column exchanges.

The same method readily applies to LU decomposition by setting P equal to the identity matrix.

C code examples

```c
/* INPUT: A - array of pointers to rows of a square matrix having dimension N
 *        Tol - small tolerance number to detect failure when the matrix is near degenerate
 * OUTPUT: Matrix A is changed, it contains both matrices L-E and U as A=(L-E)+U such that P*A=L*U.
 *        The permutation matrix is not stored as a matrix, but in an integer vector P of size N+1
 *        containing column indexes where the permutation matrix has "1". The last element P[N]=S+N,
 *        where S is the number of row exchanges needed for determinant computation, det(P)=(-1)^S
 */
int LUPDecompose(double **A, int N, double Tol, int *P) {

    int i, j, k, imax;
    double maxA, *ptr, absA;

    for (i = 0; i <= N; i++)
        P[i] = i; //Unit permutation matrix, P[N] initialized with N

    for (i = 0; i < N; i++) {
        maxA = 0.0;
        imax = i;

        for (k = i; k < N; k++)
            if ((absA = fabs(A[k][i])) > maxA) {
                maxA = absA;
                imax = k;
            }

        if (maxA < Tol) return 0; //failure, matrix is degenerate

        if (imax != i) {
            //pivoting P
            j = P[i];
            P[i] = P[imax];
            P[imax] = j;

            //pivoting rows of A
            ptr = A[i];
            A[i] = A[imax];
            A[imax] = ptr;

            //counting pivots starting from N (for determinant)
            P[N]++;
        }

        for (j = i + 1; j < N; j++) {
            A[j][i] /= A[i][i];

            for (k = i + 1; k < N; k++)
                A[j][k] -= A[j][i] * A[i][k];
        }
    }

    return 1;  //decomposition done
}


/* INPUT: A,P filled in LUPDecompose; b - rhs vector; N - dimension
 * OUTPUT: x - solution vector of A*x=b
 */
void LUPSolve(double **A, int *P, double *b, int N, double *x) {

    for (int i = 0; i < N; i++) {
        x[i] = b[P[i]];

        for (int k = 0; k < i; k++)
```

```
            x[i] -= A[i][k] * x[k];
    }

    for (int i = N - 1; i >= 0; i--) {
        for (int k = i + 1; k < N; k++)
            x[i] -= A[i][k] * x[k];

        x[i] = x[i] / A[i][i];
    }
}

/* INPUT: A,P filled in LUPDecompose; N - dimension
 * OUTPUT: IA is the inverse of the initial matrix
 */
void LUPInvert(double **A, int *P, int N, double **IA) {

    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            if (P[i] == j)
                IA[i][j] = 1.0;
            else
                IA[i][j] = 0.0;

            for (int k = 0; k < i; k++)
                IA[i][j] -= A[i][k] * IA[k][j];
        }

        for (int i = N - 1; i >= 0; i--) {
            for (int k = i + 1; k < N; k++)
                IA[i][j] -= A[i][k] * IA[k][j];

            IA[i][j] = IA[i][j] / A[i][i];
        }
    }
}

/* INPUT: A,P filled in LUPDecompose; N - dimension.
 * OUTPUT: Function returns the determinant of the initial matrix
 */
double LUPDeterminant(double **A, int *P, int N) {

    double det = A[0][0];

    for (int i = 1; i < N; i++)
        det *= A[i][i];

    if ((P[N] - N) % 2 == 0)
        return det;
    else
        return -det;
}
```

C# code examples

```
public class SystemOfLinearEquations
    {
        public double[] SolveUsingLU(double[,] matrix, double[] rightPart, int n)
        {
            // decomposition of matrix
            double[,] lu = new double[n, n];
            double sum = 0;
            for (int i = 0; i < n; i++)
            {
                for (int j = i; j < n; j++)
                {
                    sum = 0;
                    for (int k = 0; k < i; k++)
                        sum += lu[i, k] * lu[k, j];
                    lu[i, j] = matrix[i, j] - sum;
                }
                for (int j = i + 1; j < n; j++)
                {
                    sum = 0;
                    for (int k = 0; k < i; k++)
                        sum += lu[j, k] * lu[k, i];
                    lu[j, i] = (1 / lu[i, i]) * (matrix[j, i] - sum);
                }
            }

            // find solution of Ly = b
            double[] y = new double[n];
            for (int i = 0; i < n; i++)
```

```
            {
                sum = 0;
                for (int k = 0; k < i; k++)
                    sum += lu[i, k] * y[k];
                y[i] = rightPart[i] - sum;
            }
            // find solution of Ux = y
            double[] x = new double[n];
            for (int i = n - 1; i >= 0; i--)
            {
                sum = 0;
                for (int k = i + 1; k < n; k++)
                    sum += lu[i, k] * x[k];
                x[i] = (1 / lu[i, i]) * (y[i] - sum);
            }
            return x;
        }
}
```

See also

- Block LU decomposition
- Bruhat decomposition
- Cholesky decomposition
- LU Reduction
- Matrix decomposition
- QR decomposition

Notes

1. Schwarzenberg-Czerny, A. (1995). "On matrix factorization and efficient least squares solution". *Astronomy and Astrophysics Supplement* **110**: 405. Bibcode:1995A&AS..110..405S(http://adsabs.harvard.edu/abs/1995A&AS..110..405S).
2. Okunev & Johnson (1997) Corollary 3
3. Trefethen & Bau (1997), p. 166
4. Trefethen & Bau (1997), p. 161
5. Horn & Johnson (1985), Corollary 3.5.5
6. Horn & Johnson (1985), Theorem 3.5.2
7. Okunev & Johnson (1997)
8. Golub & Van Loan (1996), p. 112, 119
9. Householder (1975)
10. Shabat, Gil; Shmueli, Yaniv; Aizenbud, Yariv; Averbuch, Amir (2016). "Randomized LU Decomposition"(https://dx.doi.org/10.1016/j.acha.2016.04.006) *Applied and Computational Harmonic Analysis* doi:10.1016/j.acha.2016.04.006 (https://doi.org/10.1016%2Fj.acha.2016.04.006)
11. Bunch & Hopcroft (1974)
12. Trefethen & Bau (1997), p. 152
13. Golub & Van Loan (1996), p. 121

References

- Bunch, James R.; Hopcroft, John (1974), "Triangular factorization and inversion by fast matrix multiplication", *Mathematics of Computation* **28** (125): 231–236, doi:10.2307/2005828, ISSN 0025-5718, JSTOR 2005828.
- Cormen, Thomas H; Leiserson, Charles E; Rivest, Ronald L; Stein, Clifford (2001), *Introduction to Algorithms*, MIT Press and McGraw-Hill, ISBN 978-0-262-03293-3
- Golub, Gene H; Van Loan, Charles F. (1996), *Matrix Computations* (3rd ed.), Baltimore: Johns Hopkins, ISBN 978-0-8018-5414-9.
- Horn, Roger A.; Johnson, Charles R. (1985), *Matrix Analysis*, Cambridge University Press, ISBN 0-521-38632-2 See Section 3.5. *N* − 1

- Householder, Alston S. (1975), *The Theory of Matrices in Numerical Analysis*, New York: Dover Publications, MR 0378371.
- Okunev, Pavel; Johnson, Charles R. (1997), *Necessary And Sufficient Conditions For Existence of the LU Factorization of an Arbitrary Matrix*, arXiv:math.NA/0506382.
- Poole, David (2006), *Linear Algebra: A Modern Introduction* (2nd ed.), Canada: Thomson Brooks/Cole, ISBN 0-534-99845-3.
- Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP (2007), "Section 2.3", *Numerical Recipes: The Art of Scientific Computing* (3rd ed.), New York: Cambridge University Press, ISBN 978-0-521-88068-8.
- Trefethen, Lloyd N.; Bau, David (1997), *Numerical linear algebra*, Philadelphia: Society for Industrial and Applied Mathematics, ISBN 978-0-89871-361-9.

External links

References

- LU decomposition on *MathWorld*.
- LU decomposition on *Math-Linux*.
- LU decomposition at *Holistic Numerical Methods Institute*
- LU matrix factorization MATLAB reference.

Computer code

- LAPACK is a collection of FORTRAN subroutines for solving dense linear algebra problems
- ALGLIB includes a partial port of the LAPACK to C++, C#, Delphi, etc.
- C++ code, Prof. J. Loomis, University of Dayton
- C code, Mathematics Source Library
- LU in X10
- [1] Lu in C++ Anil Pedgaonkar
- Randomized LU MATLAB Code

Online resources

- WebApp descriptively solving systems of linear equations with LU Decomposition
- Matrix Calculator, bluebit.gr
- LU Decomposition Tool, uni-bonn.de
- LU Decomposition by Ed Pegg, Jr., The Wolfram Demonstrations Project, 2007.
