# Matrix Decomposition

## Algorithm Design

Xinpeng Wei

School of Software, Shanghai Jiao Tong University

# Introduction

# Introduction

Matrix Decomposition is a very large topic and there are lots of different decomposition for a given matrix. Such as

- LU decomposition
- QR decomposition
- Cholesky decomposition

# Introduction

Matrix Decomposition is a very large topic and there are lots of different decomposition for a given matrix. Such as

- LU decomposition
- QR decomposition
- Cholesky decomposition

And LU decomposition is useful and powerful since it can be used to solve matrix equations $Ax = b$ fast and calculate the determinant of a square matrix.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

# LU decomposition

# Definition

# Definition

**Definition**

row operation The row operation performed on matrix is one of the following:

- swap tow rows
- add one row to the other

**Definition**

row operation The row operation performed on matrix is one of the following:

- swap tow rows
- add one row to the other

**Definition**

Unit Lower triangular matrix Given a matrix L whose order is $n \times n$, L is called a Unit Lower triangular matrix if and only if for any i satisfying $1 \leq i \leq n$, $L[i, i] = 1$, and for any $i, j$ satisfying $1 \leq i < j \leq n$, $L[i, j] = 0$.

**Definition**

row operation The row operation performed on matrix is one of the following:

- swap tow rows
- add one row to the other

**Definition**

Unit Lower triangular matrix Given a matrix L whose order is $n \times n$, L is called a Unit Lower triangular matrix if and only if for any i satisfying $1 \leq i \leq n$, $L[i, i] = 1$, and for any $i, j$ satisfying $1 \leq i < j \leq n$, $L[i, j] = 0$.

**Definition**

Upper triangular matrix Given a matrix L whose order is $n \times n$, L is called a Upper triangular matrix if and only if for any $i, j$ satisfying $1 \leq i < j \leq n$, $L[j, i] = 0$.

# Further Definition

**Definition**

LU decomposition Given a matrix A whose order is $n \times n$, $A = LU$ is called a LU decomposition of matrix A if and only L is a Unit Lower triangular matrix and U is an Upper triangular matrix.

**Definition**

LU decomposition Given a matrix A whose order is $n \times n$, $A = LU$ is called a LU decomposition of matrix A if and only L is a Unit Lower triangular matrix and U is an Upper triangular matrix.

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}}_{U}$$

# Overview

There are lots of algorithms designed to determine the LU decomposition of a matrix.
We will explore some of them step by step.

# Overview

There are lots of algorithms designed to determine the LU decomposition of a matrix.
We will explore some of them step by step.

**Assumption**

In our discussion below, we assume the matrix A is fully-ranked because if not we will not get the decomposition. Also, we should note that the major task of LU decomposition is to solve $\mathrm{Ax} = \mathrm{b}$ fast and sound, when A is not fully-marked there is no exact solution to the equation. So our assumption is appropriate.

# Gaussian elimination

# Gaussian elimination

First we describe the algorithm

# Gaussian elimination

First we describe the algorithm

step 1 Start from the first row and consider the first element of the row: if it is zero, swap the rows and get a nonzero first element. Note that this process only fails if and only if all the elements of the first column are zero thus A is not fully-ranked, which is not possible in our context.

First we describe the algorithm

step 1 Start from the first row and consider the first element of the row: if it is zero, swap the rows and get a nonzero first element. Note that this process only fails if and only if all the elements of the first column are zero thus A is not fully-ranked, which is not possible in our context.

step 2 Then repeatedly do the following: for row i $(1 < i \leq n)$ , if the first element of row i is zero, just skip the row and proceed to the next row. Otherwise, we consider row i with a nonzero first element $X_i$, then add row 1 multiply $-X_i/P_1$ to row i, $P_1$ is the first element of row 1,we eliminate the first element of row i to zero.

First we describe the algorithm

step 1 Start from the first row and consider the first element of the row: if it is zero, swap the rows and get a nonzero first element. Note that this process only fails if and only if all the elements of the first column are zero thus A is not fully-ranked, which is not possible in our context.

step 2 Then repeatedly do the following: for row i $(1 < i \leq n)$ , if the first element of row i is zero, just skip the row and proceed to the next row. Otherwise, we consider row i with a nonzero first element $X_i$, then add row 1 multiply $-X_i/P_1$ to row i, $P_1$ is the first element of row 1,we eliminate the first element of row i to zero.

step 3 just continue the loop until we finally get an Upper triangular matrix.

**Lemma**

swap two rows is equivalent to left multiplication a identity matrix with row i and j swapped.

**Lemma**

swap two rows is equivalent to left multiplication a identity matrix with row i and j swapped.

Proof. The P satisfies following properties

1. for $1 \leq k \leq n, k \neq i, j$, $P[k, k] = 1$
2. $P[j, i] = P[i, j] = 1$
3. for all other pair $(i, j)$, $P[i, j] = 0$

Now it's easy to confirm that $A_t = PA$ is the same as A with row i and j swapped.

**Lemma**

add one row to the other is equivalent to left multiplication a identity matrix with row i added to row j.

**Lemma**

add one row to the other is equivalent to left multiplication a identity matrix with row i added to row j.

Proof. The P satisfies following properties

1. for $1 \leq k \leq n$, $P[k, k] = 1$.
2. $P[j, i] = 1$.
3. for all other pair $(i, j)$,$P[i, j] = 0$.

The same as last lemma, we can do the matrix multiplication and confirm that $A_t = PA$ is the same as A with row i added to row j.

**Theorem 1**

the row operations performed on matrix are equivalent to pre-multiply a certain matrix P.

**Theorem 1**

the row operations performed on matrix are equivalent to pre-multiply a certain matrix P.

*Proof.* We finish the proof with two kinds of row operations defined above.

**Theorem 2**

the product of two Unit lower triangular matrix is a Unit lower triangular matrix.

**Theorem 2**

the product of two Unit lower triangular matrix is a Unit lower triangular matrix.

*Proof.* let A and B are Unit lower triangular matrix. We compute the product of A and B, let C = AB.

1. we compute all diagonal elements of C. C[i,i]=$\Sigma_{k=1}^{k=n} A[i,k]B[k,i]$. Note that A,B are Unit lower triangular matrix, so for $1 \leq k < i$, $B[k,i] = 0$ and for $i < k \leq n$, $A[i,k] = 0$, $B[i,i] = 1$ and $A[i,i] = 1$. Use the three facts above, we get C[i,i]=1.
2. we now prove for $1 \leq i < j \leq n$, C[i,j]=0. C[i,j]=$\Sigma_{k=1}^{k=n} A[i,k]B[k,j]$, since for $1 \leq k < j$, B[k,j] = 0 and for $i < k \leq n$, A [i,k] = 0, So $1 \leq k < j$, A[i,k]B[k,j] = 0 and for $i < k \leq n$, A [i,k]B[k,j] = 0, we get C[i,j] = 0

Based on the theorems above, we first do the row swap such that during the elimination, the diagonal element of A is nonzero. as we have proved this leads to $PA$, then we do the elimination by adding one row to the other, by the 5, this can be done by a series of pre-multiply: $E_1E_2...E_nPA = U$, and by Lemma 6, we get $PA = E_n^{-1}...E_2^{-1}E_1^{-1}U = LU$, note that the redundant P do not interfere with our goal because in practice we just first do the swap.

**Theorem**

The time complexity of Gaussian Elimination is $O(n^3)$

**Theorem**

The time complexity of Gaussian Elimination is $O(n^3)$

Proof. It's obvious since we eliminate every column use $O(n^2)$ multiplication and addition and there are n rows.

# In practice

- The complexity($O(n^3)$) in real world is not acceptable as n is usually incredibly large, typically $10^5$.

# In practice

- The complexity($O(n^3)$) in real world is not acceptable as n is usually incredibly large, typically $10^5$.
- The conditional number of the procedure above is very large which means the solution is susceptible to inaccuracy of the data caused by float point number operation.

# Left Looking elimination

# Left Looking Elimination

Let's derive a left looking version of Gaussian elimination.

# Left Looking Elimination

Let's derive a left looking version of Gaussian elimination.
We can further extend this approach to our real world practice.

# Left Looking Elimination

Let's derive a left looking version of Gaussian elimination.
We can further extend this approach to our real world practice.

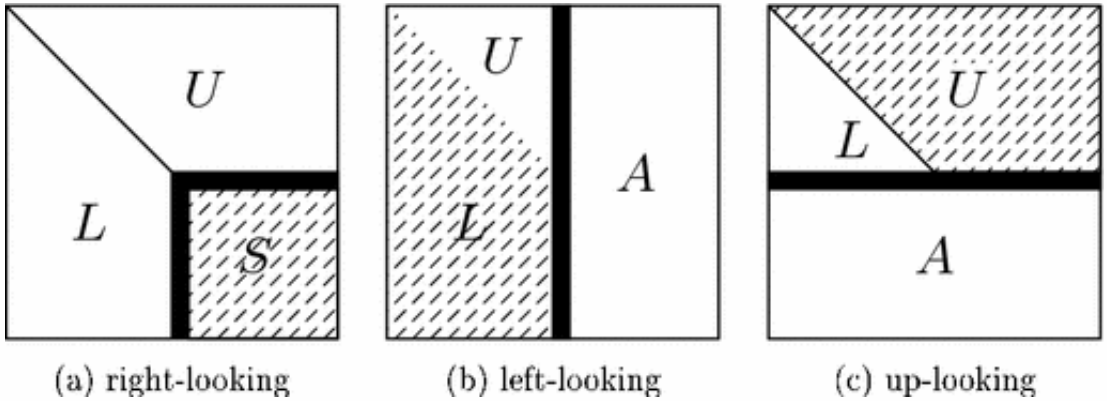


Fig: Three main *-looking LU

# Declaration

Given an input matrix A with order of $n \times n$ can be represented as a product of two triangular matrices L and U. We write A as follow:

$$\begin{pmatrix} A_{11} & \boldsymbol{\alpha_{12}} & A_{13} \\ \boldsymbol{\alpha_{21}} & a_{22} & \boldsymbol{\alpha_{23}} \\ A_{31} & \boldsymbol{\alpha_{32}} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & \mathbf{l_{32}} & L_{33} \end{pmatrix} \times \begin{pmatrix} U_{11} & \mathbf{u_{12}} & U_{13} \\ 0 & u_{22} & \mathbf{u_{23}} \\ 0 & 0 & U_{33} \end{pmatrix}$$

where $A_{ij}$ is a block, $\boldsymbol{\alpha_{ij}}$ is a vector and $a_{ij}$ is a scalar. The dimensions of different elements in the matrices are as follows:

# Declaration

$$\begin{pmatrix} A_{11} & \boldsymbol{\alpha_{12}} & A_{13} \\ \boldsymbol{\alpha_{21}} & a_{22} & \boldsymbol{\alpha_{23}} \\ A_{31} & \boldsymbol{\alpha_{32}} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & \mathbf{l_{32}} & L_{33} \end{pmatrix} \times \begin{pmatrix} U_{11} & \mathbf{u_{12}} & U_{13} \\ 0 & u_{22} & \mathbf{u_{23}} \\ 0 & 0 & U_{33} \end{pmatrix}$$

- $A_{11}, L_{11}, U_{11}$ are $k \times k$ blocks
- $\boldsymbol{\alpha_{12}}, \mathbf{u_{12}}$ are $k \times 1$ vectors
- $A_{13}, U_{13}$ are $k \times n - (k+1)$ blocks
- $\boldsymbol{\alpha_{21}}, \mathbf{l_{21}}$ are $1 \times k$ row vectors
- $a_{22}, u_{22}$ are scalars
- $\boldsymbol{\alpha_{23}}, \mathbf{u_{23}}$ are $1 \times n - (k+1)$ row vectors
- $A_{31} L_{31}$ are $n - (k+1) \times k$ blocks
- $\boldsymbol{\alpha_{32}}, \mathbf{l_{32}}$ are $n - (k+1) \times 1$vectors
- $A_{33}, L_{33}, U_{33}$ are $n - (k+1) \times n - (k+1)$ blocks

# Solve

$$\begin{pmatrix} A_{11} & \boldsymbol{\alpha_{12}} & A_{13} \\ \boldsymbol{\alpha_{21}} & a_{22} & \boldsymbol{\alpha_{23}} \\ A_{31} & \boldsymbol{\alpha_{32}} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & \mathbf{l_{32}} & L_{33} \end{pmatrix} \times \begin{pmatrix} U_{11} & \mathbf{u_{12}} & U_{13} \\ 0 & u_{22} & \mathbf{u_{23}} \\ 0 & 0 & U_{33} \end{pmatrix}$$

Now, we calculate the equation and get following equations:

- $L_{11} \times U_{11} = A_{11}$
- $L_{11} \times \mathbf{u_{12}} = \boldsymbol{\alpha_{12}}$
- $L_{11} \times U_{13} = A_{13}$
- $\mathbf{l_{21}} \times U_{11} = \boldsymbol{\alpha_{21}}$
- $\mathbf{l_{21}} \times \mathbf{u_{12}} + u_{22} = a_{22}$
- $\mathbf{l_{21}} \times U_{13} + \mathbf{u_{23}} = \boldsymbol{\alpha_{23}}$
- $L_{31} \times U_{11} = A_{31}$
- $L_{31} \times \mathbf{u_{12}} + \mathbf{l_{32}} \times u_{22} = \boldsymbol{\alpha_{32}}$

# Clarification

The method is called Left-looking elimination because if computes the $k^{th}$ column of L and U using their $1, 2 \ldots k-1$ columns.

# Clarification

The method is called Left-looking elimination because if computes the $k^{th}$ column of L and U using their $1, 2 \ldots k-1$ columns.

Suppose we already know the first $k-1$ columns. We can extract the parts interesing us and get the following equation:

$$\begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \mathbf{u_{12}} \\ u_{22} \\ \mathbf{l_{32}} \times u_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\alpha_{12}} \\ a_{22} \\ \boldsymbol{\alpha_{32}} \end{pmatrix}$$

# Compute

$$\begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \mathbf{u_{12}} \\ u_{22} \\ \mathbf{l_{32}} \times u_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\alpha_{12}} \\ a_{22} \\ \boldsymbol{\alpha_{32}} \end{pmatrix}$$

$\mathbf{u_{12}}$, $u_{22}$, $\mathbf{l_{32}}$ is all what we need to get the $k^{th}$ column of L and U, and it is clear how to get them:

- first compute $L_{11} \times \mathbf{u_{12}} = \boldsymbol{\alpha_{12}}$, we get $\mathbf{u_{12}}$
- then compute $u_{22} = a_{22} - \mathbf{l_{21}} \times \mathbf{u_{12}}$, we get $u_{22}$
- finally compute $\mathbf{l_{32}} = \frac{1}{u_{22}}(\boldsymbol{\alpha_{32}} - L_{31} \times \mathbf{u_{12}})$,we get $\mathbf{l_{32}}$

# Compute

$$\begin{pmatrix} L_{11} & 0 & 0 \\ \mathbf{l_{21}} & 1 & 0 \\ L_{31} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \mathbf{u_{12}} \\ u_{22} \\ \mathbf{l_{32}} \times u_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\alpha_{12}} \\ a_{22} \\ \boldsymbol{\alpha_{32}} \end{pmatrix}$$

$\mathbf{u_{12}}, u_{22}, \mathbf{l_{32}}$ is all what we need to get the $k^{th}$ column of L and U, and it is clear how to get them:

- first compute $L_{11} \times \mathbf{u_{12}} = \boldsymbol{\alpha_{12}}$, we get $\mathbf{u_{12}}$
- then compute $u_{22} = a_{22} - \mathbf{l_{21}} \times \mathbf{u_{12}}$, we get $u_{22}$
- finally compute $\mathbf{l_{32}} = \frac{1}{u_{22}}(\boldsymbol{\alpha_{32}} - L_{31} \times \mathbf{u_{12}})$,we get $\mathbf{l_{32}}$

step 2 and 3 are easy since they just involve multiplication and addition. How to compute $u_{12}$?

# Compute $u_{12}$

Note $L_{11}$ is the upper $k \times k$ blocks of Unit lower triangular L, so $L_{11}$ is also Unit lower triangular. Hence we can derive a simple algorithm.

```
x ← b;
for i = 1 to n do
    if x(i) ≠ 0 then
        for j = i + 1 to n do
            x(j) = x(j) − L(j, i) ∗ x(i)
        end
    end
end
```

# Complexity

**Theorem**

The time complexity of Left-Looking Elimination is $O(n^2)$

**Theorem**

The time complexity of Left-Looking Elimination is $O(n^2)$

Proof. Every time we compute $\mathbf{u_{12}}$, we use $O(n + \text{number of multiplications})$ And we have n iterations. So it seems likely to be $O(n^3)$, but the vector addition can be improved to nearly $O(1)$, thus we can think this approach can achieve at most $O(n^2)$.

**Theorem**

The time complexity of Left-Looking Elimination is $O(n^2)$

Proof. Every time we compute $\mathbf{u_{12}}$, we use $O(n + \text{number of multiplications})$ And we have n iterations. So it seems likely to be $O(n^3)$, but the vector addition can be improved to nearly $O(1)$, thus we can think this approach can achieve at most $O(n^2)$.

**In practice**

Although far more better than the Gauss-Elimination, it is not practical to put into use in industry as well due to quadratic complexity.

# A Practival method: G-P Algotithm

# Brief

Gauss Elimination and Left-Looking Elimination are not efficient in large scale input.

But in real practice, a certain kind of problems share some common characteristics.

# Brief

Gauss Elimination and Left-Looking Elimination are not efficient in large scale input.
But in real practice, a certain kind of problems share some common characteristics.

Gilbert Peierls Algorithm is one of the algorithms that exploit these characteristics. Almost all further improvement algorithms are based on the idea proposed by Gilbert Peierls Algorithm.

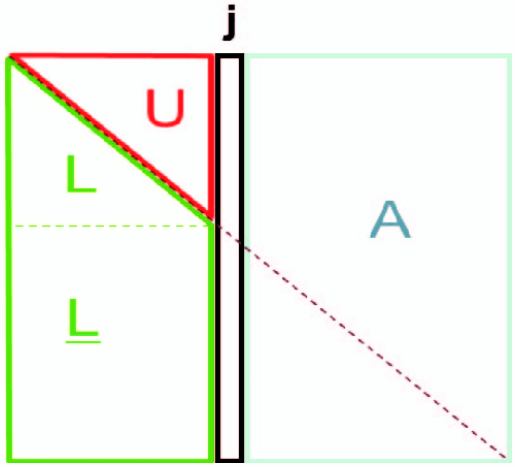


Fig: Gilbert Peierls LU

**Definition**

sparse matrix A matrix is called sparse if the nonzero elements is very few. In practice, we usually call a matrix sparse if it has O(n)

**Definition**

sparse matrix A matrix is called sparse if the nonzero elements is very few. In practice, we usually call a matrix sparse if it has $O(n)$

For sparse matrixs, we usually them using the column compressed form and it only takes $O(n)$ space.

# Column compressed form

**Definition**

column compressed form A column compressed form of a matrix consists of three vectors: $A_p$, $A_i$, $A_x$

| column | definition |
|---|---|
| $A_x$ | contains all the nonzero elements of A from column 1 to column n |
| $A_i$ | contains all the nonzero elements column index from column 1 to column n |
| $A_p$ | contains all the ending index of a given column in $A_i$ except the first element is always 0 |

# Quiz

Given matrix

$$\begin{pmatrix} 5 & 0 & 0 \\ 4 & 2 & 0 \\ 3 & 1 & 8 \end{pmatrix}$$

What's the corresponding column compressed format ?

# Quiz

Given matrix

$$\begin{pmatrix} 5 & 0 & 0 \\ 4 & 2 & 0 \\ 3 & 1 & 8 \end{pmatrix}$$

What's the corresponding column compressed format ?

| column | value |
|---|---|
| $A_x$ | $5, 4, 3, 2, 1, 8$ |
| $A_i$ | $0, 1, 2, 1, 2, 2$ |
| $A_p$ | $0, 3, 5, 6$ |

# Gilbert Peierls Algorithm

**Aim**

Decomposition arbitary non singular sparse matrix $A$ as $PA = LU$ in time proportion to the flop count of the L and U.

**Aim**

Decomposition arbitary non singular sparse matrix $A$ as $PA = LU$ in time proportion to the flop count of the L and U.

**Definition**

flops(LU) The symbol flops(LU) is the number of nonzero multiplications performed when multiplying two matrices L and U.

**Aim**

Decomposition arbitary non singular sparse matrix $A$ as $PA = LU$ in time proportion to the flop count of the L and U.

**Definition**

flops(LU) The symbol flops(LU) is the number of nonzero multiplications performed when multiplying two matrices L and U.

We analysis the complexity the algorithm based on the flops(LU).

. The time complexity is $O(n + \text{number of flops performed})$, we can remove the $O(n)$ term using the follow one.

. The time complexity is $O(n + \text{number of flops performed})$, we can remove the $O(n)$ term using the follow one.

```
x ← b;
for all i satisfying x(i) ≠ 0 do
    for j = i + 1 to n do
        x(j) = x(j) − L(j, i) ∗ x(i)
    end
end
```

. The time complexity is $O(n + \text{number of flops performed})$, we can remove the $O(n)$ term using the follow one.

```
x ← b;
for all i satisfying x(i) ≠ 0 do
    for j = i + 1 to n do
        x(j) = x(j) − L(j, i) ∗ x(i)
    end
end
```

Improvement: This would reduce the $O(n)$ to $O(\eta(x))$

. The time complexity is $O(n + \text{number of flops performed})$, we can remove the $O(n)$ term using the follow one.

```
x ← b;
for all i satisfying x(i) ≠ 0 do
    for j = i + 1 to n do
        x(j) = x(j) − L(j, i) ∗ x(i)
    end
end
```

Improvement: This would reduce the $O(n)$ to $O(\eta(x))$

Q: how to recognize nonzero pattern of x before we compute x itself?

**Theorem**

Let $G = G(L_k)$ be the directed graph of L with $k-1$ vertices representing the already computed $k-1$ columns. $G(L_k)$ has an edge $j \rightarrow i$ if and only if $l_{ij} \neq 0$. Let $\beta = \{i|b_i \neq 0\}$ and $X = \{i|x_i \neq 0\}$, Now the elements of X is given by

$$X = \mathrm{Reach}_{G(L)}(\beta)$$

**Theorem**

Let $G = G(L_k)$ be the directed graph of L with $k-1$ vertices representing the already computed $k-1$ columns. $G(L_k)$ has an edge $j \rightarrow i$ if and only if $l_{ij} \neq 0$. Let $\beta = \{i | b_i \neq 0\}$ and $X = \{i | x_i \neq 0\}$, Now the elements of X is given by

$$X = \mathrm{Reach}_{G(L)}(\beta)$$

Proof. The proof is very tricky so we don't cover it here.

**Theorem**

Let $G = G(L_k)$ be the directed graph of L with $k-1$ vertices representing the already computed $k-1$ columns. $G(L_k)$ has an edge $j \rightarrow i$ if and only if $l_{ij} \neq 0$. Let $\beta = \{i | b_i \neq 0\}$ and $X = \{i | x_i \neq 0\}$, Now the elements of X is given by

$$X = \mathrm{Reach}_{G(L)}(\beta)$$

Observation: The nonzero pattern of $X$ is computed by determining the vertices reachable from the vertices of the set $\beta$.

**Theorem**

Let $G = G(L_k)$ be the directed graph of L with $k-1$ vertices representing the already computed $k-1$ columns. $G(L_k)$ has an edge $j \rightarrow i$ if and only if $l_{ij} \neq 0$. Let $\beta = \{i | b_i \neq 0\}$ and $X = \{i | x_i \neq 0\}$, Now the elements of X is given by

$$X = \mathrm{Reach}_{G(L)}(\beta)$$

Observation: The nonzero pattern of X is computed by determining the vertices reachable from the vertices of the set $\beta$.

Solution: So we can just use a depth first search from the vertives of the set $\beta$ during which we also can get a topological order of X which is useful for eliminating unknowns in the next step.

Numerical Factorization consists of solving for each column of L and U.

Numerical Factorization consists of solving for each column of L and U.

We usually solve the equation in increasing order of the row index.

Numerical Factorization consists of solving for each column of L and U.

We usually solve the equation in increasing order of the row index.
But row indeces computed from DFS aren't in right order and sorting is really time-cosuming.

# Numerical Factorization

Numerical Factorization consists of solving for each column of L and U.

We usually solve the equation in increasing order of the row index.
But row indeces computed from DFS aren't in right order and sorting is really time-cosuming.

Loose: We use topological order of the row indices instead.

Numerical Factorization consists of solving for each column of $L$ and $U$.

We usually solve the equation in increasing order of the row index.
But row indeces computed from DFS aren't in right order and sorting is really time-cosuming.

Loose: We use topological order of the row indices instead.

Proof. Once all the unknowns $x_j$ of which it's dependent are computed, an unknown $x_i$ can be computed.

# Complete picture

We finally got the algorithm:

```
L ← I;
for i = 1 to n do
    X ← L \ k_th column of A;
    for j = 1 to i do
        U(j, k) = x(j)
    end
    for j = k to n do
        L(j, k) = x(j)/U(k, k)
    end
end
```

$x = L \backslash b$ in the above algorithm denotes the solution of: $Lx = b$. In this case, b is the $k^{th}$ column of A.

# Complete picture

We finally got the algorithm:

```
L ← I;
for i = 1 to n do
    X ← L \ k_th column of A;
    for j = 1 to i do
        U(j, k) = x(j)
    end
    for j = k to n do
        L(j, k) = x(j)/U(k, k)
    end
end
```

$x = L \backslash b$ in the above algorithm denotes the solution of: $Lx = b$. In this case, b is the $k^{th}$ column of A.

**Theorem**

The time complexity of the algorithm is $O(\eta(A) + \text{flops}(LU))$.

$\eta(A)$ is the number of nonzeros in the matrix A and $\text{flops}(LU)$ is the flop count of the product of the matrices L and U. We also do a depth first search, it costs nearly $O(n)$.

**Theorem**

The time complexity of the algorithm is $O(\eta(A) + \text{flops}(LU))$.

$\eta(A)$ is the number of nonzeros in the matrix A and $\text{flops}(LU)$ is the flop count of the product of the matrices L and U. We also do a depth first search, it costs nearly $O(n)$.

Claim: $\text{flops}(LU)$ is less then the nonzero element in L or U.

**Theorem**

The time complexity of the algorithm is $O(\eta(A) + \text{flops}(LU))$.

$\eta(A)$ is the number of nonzeros in the matrix A and $\text{flops}(LU)$ is the flop count of the product of the matrices L and U. We also do a depth first search, it costs nearly $O(n)$.

Claim: $\text{flops}(LU)$ is less then the nonzero element in L or U.

Q: Won't it reach $O(n^2)$?

**Theorem**

The time complexity of the algorithm is $O(\eta(A) + \text{flops}(LU))$.

$\eta(A)$ is the number of nonzeros in the matrix A and $\text{flops}(LU)$ is the flop count of the product of the matrices L and U. We also do a depth first search, it costs nearly $O(n)$.

Claim: $\text{flops}(LU)$ is less then the nonzero element in L or U.

Q: Won't it reach $O(n^2)$?

A: In practice, it works really fast, slightly bigger than $O(n)$.

**Theorem**

The time complexity of the algorithm is $O(\eta(A) + \text{flops}(LU))$.

$\eta(A)$ is the number of nonzeros in the matrix A and $\text{flops}(LU)$ is the flop count of the product of the matrices L and U. We also do a depth first search, it costs nearly $O(n)$.

Claim: $\text{flops}(LU)$ is less then the nonzero element in L or U.

Q: Won't it reach $O(n^2)$?

A: In practice, it works really fast, slightly bigger than $O(n)$.

So the $\text{flops}(LU)$ dominants and we can assume it's $O(\text{flops}(LU))$.

# Conclusion

The Gilbert Peierls Algorithm is the base decomposition algorithm for scientific computing package such as Matlab, numpy, OpenBLAS and SuiteSparse.

# Numpy

We can see KLU's usage in numpy's main repository on GitHub.

| | |
|---|---|
| klu.c | v3.7.1 |
| klu_analyze.c | v4.3.0 |
| klu_analyze_given.c | v5.2.0 |
| klu_defaults.c | v4.3.0 |
| klu_diagnostics.c | v4.3.0 |
| klu_dump.c | v4.4.6 |
| klu_extract.c | v3.3.0 |
| klu_factor.c | v5.2.0 |
| klu_free_numeric.c | v3.3.0 |
| klu_free_symbolic.c | v3.3.0 |

```
klu_defaults (&Common) ;
Symbolic = klu_analyze (n, Ap, Ai, &Common) ;
Numeric = klu_factor (Ap, Ai, Ax, Symbolic, &Common) ;
klu_solve (Symbolic, Numeric, 5, 1, b, &Common) ;
klu_free_symbolic (&Symbolic, &Common) ;
klu_free_numeric (&Numeric, &Common) ;
```