LU factorization

Problem:

\[ Ax = b \]

where \( A \in \mathbb{R}^{n \times n} \), \( x, b \in \mathbb{R}^{n} \), and \( A^{-1} \) exists.

LU factorization or Gaussian elimination reduces the problem to one we know how to solve – triangular systems.

It computes the factorization

\[ A = LU \]

where \( L \) is a unit lower triangular matrix and \( U \) is an upper triangular matrix.

We will assume for now that pivoting for stability or existence is not needed.
The algorithm is:

1. calculate $L$ and $U$, $\Omega \approx \frac{2}{3} n^3$, $\delta = n^2$

2. solve $Ly = b$, $\Omega \approx n^2$, $\delta \approx n^2/2$

3. solve $Ux = y$, $\Omega \approx n^2$, $\delta \approx n^2/2$
Gauss transforms

Let $C_i = I - l_i e_i^T$, where $l_i^T e_j = 0$ for $j \leq i$, be an elementary lower triangular matrix (column form).

$C_i$ is a Gauss transform when $l_i$ is chosen so that a given vector $x \in \mathbb{R}^n$ is transformed into a vector $y \in \mathbb{R}^n$ such that $\xi_j = \eta_j$ for $1 \leq j \leq i$ and $\eta_j = 0$ for $i + 1 \leq j \leq n$.

That is $C_i$ introduces 0 into the positions below the i-th element of $x$. 
Consider the application of a Gauss transform to a matrix $A$.

$$M_1^{-1}A = \begin{pmatrix}
1 & & \\
-\lambda_2^{(1)} & \cdots & \\
\vdots & \ddots & \\
-\lambda_n^{(1)} & & 1
\end{pmatrix} \begin{pmatrix}
\alpha_{11} & \cdots & \alpha_{1n} \\
\vdots & & \vdots \\
\alpha_{n1} & \cdots & \alpha_{nn}
\end{pmatrix}$$

If $\lambda_j^{(1)} = \alpha_{j1}/\alpha_{11}$ then

$$M_1^{-1}A = A^{(1)} = \begin{pmatrix}
\alpha_{11} & \cdots & \alpha_{1n} \\
0 & & \\
\vdots & & A_1 \\
0 & & 
\end{pmatrix}$$

where $A_1 \in \mathbb{R}^{n-1 \times n-1}$.

The first row remains the same and becomes the first row of $U$. Elements in rows 2 to $n$ in the first column are 0 and the rest of the elements are updated.
The application of $M_1^{-1}$ requires a scale of part of the first column of $A$ followed by

$$(I - l_1 e_1^T)A = A - l_1 a_1^T$$

where $a_1^T$ is the first row of $A$, which is a rank-1 update.

Note that due to the structure of $M_1^{-1}$ not all of $A$ needs to be updated – only a submatrix of order $n - 1$.

The process can be repeated to eliminate the nonzeros in the second column of $A^{(1)}$ (or the first column of $A_1$) via $M_2^{-1}$ without destroying the zeros in the first column. This second step requires a rank-1 update of order $n - 2$. After $n - 1$ such steps we have

$$M_{n-1}^{-1} \cdots M_2^{-1}M_1^{-1}A = U = \begin{pmatrix} \mu_{11} & \cdots & \mu_{1n} \\ 0 & \ddots & \vdots \\ 0 & 0 & \mu_{nn} \end{pmatrix}$$
Where is $L$?

Recall the properties of elementary lower triangular matrices discussed earlier. We have

$$A = M_1 M_2 \cdots M_{n-1} U$$

where $M_i = I + l_i e_i^T$ is a elementary lower triangular matrix.

Therefore $M_1 \cdots M_{n-1}$ is a unit lower triangular matrix. And due to the way in which these matrices combine we know that the nonzeros below the diagonal in the $i$-th column are given by the nonzero elements of $l_i$ for $i = 1, \ldots, n - 1$, i.e.,

$$L = \begin{pmatrix}
1 \\
\lambda_2^{(1)} & 1 \\
\vdots & \ddots & \ddots \\
\lambda_n^{(1)} & \cdots & \lambda_n^{(n-1)} & 1
\end{pmatrix}$$
The algorithm is then a series of \( n - 1 \) stages with \( A = A_0 \) the i-th stage of which is a BLAS1 primitive and a BLAS2 primitive:

- Scale the nonzeros below the diagonal of the first column of \( A_{i-1} \) to produce \( l_i \).

- Perform a rank-1 update of order \( n - i \) to produce \( A_i \).
Algebraically this can be expressed as:

Let $A_0 = A$ and let

$$A_i = \begin{bmatrix} \alpha_{11}^{(i)} & a_i^T \\ b_i & B_i \end{bmatrix}$$

where $a_i, b_i \in \mathbb{R}^{n-i-1}$ and $B_i \in \mathbb{R}^{n-i-1 \times n-i-1}$.

We then have

\[
\text{do} \quad I = 1, n - 1 \\
\quad l_i = (1/\alpha_{11}^{(i-1)}) b_{i-1} \\
\quad A_i = B_{i-1} - l_i a_{i-1}^T \\
\text{enddo}
\]

The performance when implementing in terms of total primitives depends upon how well-suited the machine is to rank-1 updates. If total primitives are not used then it depends on how well the steps can be overlapped via whatever synchronization method is available.

Note that stage $i$ produces column $i$ of $L$ and row $i$ of $U$ as well as updating the remaining active part of $A$. 
Standard form of LU factorization

\[ i-1 \text{ columns of } L \]

\[ i-1 \text{ rows of } U \]

\[ i-\text{th column of } L \]

\[ i-\text{th row of } U \]

\[ \text{active portion of the matrix for the step } i+1 \rightarrow A_i \]

\[ i-1 \text{ columns of } L \text{ and } i-1 \text{ rows of } U \text{ are not touched.} \]

\[ i-\text{th column of } L \text{ is computed and used along with the } i-\text{th row of } U \]

\[ A_i \text{ computed and the active portion of the matrix updated.} \]
Alternate versions

There are many versions of the LU factorization possible. In the literature they are often referred to by their corresponding loop orderings. A more descriptive and illuminating manner of description is to use the primitives involved and the portions of $L$ and $U$ computed on each step – the standard version is based on a rank-1 update and a vector scale and produces the $i$-th row of $U$ and the $i$-th column of $L$ on each step. The use of the rank-1 update also implies that the version is based on the notion of an immediate update of all data.

Suppose, however, we are on a machine where rank-1 updates are not that efficient, e.g., a register-based vector processor with a single port to memory. We would prefer to have a version in terms of reduction operations, say, matrix-vector products and triangular system solving. (Recall, that on a single vector processor a triangular solve is of BLAS2 level complexity.) The use of reduction operations implies the use of the notion of delayed updates, i.e., updating only the information needed to proceed with the calculation of the next Gauss Transform and no more. By considering the mathematical dependences implicit in the algebra of LU factorization we can derive such a version (and many others as well).
Define

\[ A = [a_1, \ldots, a_n] \]

where \( a_i \in \mathbb{R}^n \) and let

\[ a_i^{(j)} = M_j^{-1} \cdots M_2^{-1} M_1^{-1} a_i \]

Note that \( a_i^{(j)} = a_i^{(i)} \) for \( j > i \) due to the structure of \( M_j^{-1} \). Furthermore, we only need \( a_j^{(j-1)} \) to compute \( M_j^{-1} \) via \( l_j \). So we do not need to produce all of \( A_{j-1} \), as done in the standard immediate update version, to proceed with the \( j \)-th step of elimination. Also note that producing \( a_i^{(i)} \) requires no computation given \( a_i^{(i-1)} \) since the difference is only the creation of 0 in certain positions.

These considerations produce a delayed update version of the algorithm.
Given that $a_i^{(i)}$ is trivially producable at the i-th stage we need only concentrate on computing $a_i^{(i-1)}$. This is given by

$$a_i^{(i-1)} = M_{i-1}^{-1} \cdots M_1^{-1} a_i$$

which is equivalent to solving the unit lower triangular system

$$M_1 \cdots M_{i-1} a_i^{(i-1)} = L^{(i-1)} a_i^{(i-1)} = a_i.$$ 

Note that $L^{(i-1)}$ has the following structure

\[
\begin{bmatrix}
1 & \cdots & 0 \\
& \ddots & \vdots \\
& & 1 \\
1 & \cdots & i-1
\end{bmatrix}
\]

i.e., only nonzero in columns 1 to $i - 1$. 
This structure can be exploited to simplify the solution of the triangular system at each stage.

Partition the system
\[
\begin{pmatrix}
\tilde{L} & 0 \\
B & I
\end{pmatrix}
\begin{pmatrix}
\tilde{a} \\
\tilde{a}
\end{pmatrix}
=
\begin{pmatrix}
\tilde{d} \\
\tilde{d}
\end{pmatrix}
\]
where \( \tilde{L} \) is a unit lower triangular matrix of order \( i - 1 \), \( \tilde{a} \in \mathbb{R}^{i-1} \) and the rest of the system is partitioned conformally.

Stage \( i \) then becomes

- Solve \( \tilde{L}\tilde{a} = \tilde{d} \) (a triangular system of order \( i - 1 \))
- \( \tilde{a} \leftarrow \tilde{d} - B\tilde{a} \) (an \( n-i+1 \times i-1 \) matrix vector product)

which uses the two preferred BLAS2 primitives for register-based vector processors, as desired. The i-th stage computes the i-th column of \( L \) and the i-th column of \( U \) with no further update of the matrix.

Several other versions are possible depending on the preferred primitives.
Column-oriented delayed update version of LU factorization

- $i-1$ columns of $U$
- $n-i$ columns of $A$
- $i$-th column of $A, L,$ and $U$
- $i-1$ columns of $U$ completed and not touched
- $i-1$ columns of $L$ completed and used on the $i$-th step
- $n-i$ columns of $A$ not touched
- $i$-th columns of $L$ and $U$ computed from $i$-th column of $A$
BLAS3-based versions

There are two basic ways to get a BLAS3-based version of LU factorization

1. generalize the factorization from scalar operations to submatrix operation to get a block LU factorization, i.e., a different factorization.

2. rearrange the standard LU factorization to generate submatrix computations
A block generalized factorization

Consider approach 1 first. Let $A \in \mathbb{R}^{n \times n}$ be a diagonally dominant or positive definite matrix and partition the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where $A_{11} \in \mathbb{R}^{\omega \times \omega}$, $A_{22} \in \mathbb{R}^{n-\omega \times n-\omega}$ and the rest of the matrix is partitioned conformally.

Define $L$ and $U$,

$$L = \begin{pmatrix} I & 0 \\ L_{21} & L_{22} \end{pmatrix} \quad U = \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

where $I$ is the identity of order $\omega$, $L_{22} \in \mathbb{R}^{n-\omega \times n-\omega}$ is a lower triangular matrix, $U_{11} \in \mathbb{R}^{\omega \times \omega}$ is a dense matrix, $U_{22} \in \mathbb{R}^{n-\omega \times n-\omega}$ and the rest of the blocks are dimensioned conformally.

Note that $U$ is block upper triangular and that $L$ is such that the diagonal blocks of order $\omega$ are the identity.
If we multiply $L$ and $U$ and set equal corresponding blocks we have

- $A_{11} = U_{11}$
- $U_{12} = A_{12}$
- $L_{21}U_{11} = A_{21}$
- $L_{22}U_{22} = A_{22} - L_{21}U_{12}$

These identities yield the following algorithm

(i) Compute $U_{11}^{-1} = A_{11}^{-1}$ (or a factorization of $A_{11}$)

(ii) Compute $L_{21} = A_{21}U_{11}^{-1}$ (matrix product or two triangular solves depending on choice in (i))

(iii) Compute $\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12}$ (a rank-$\omega$ update)

(iv) Proceed recursively on $\tilde{A}_{22}$ to compute $L_{22}$ and $U_{22}$
Suppose $U_{11}^{-1}$ is computed explicitly via the Gauss-Jordan reduction, then at each stage we perform

(i) G-J on an $\omega \times \omega$ matrix

(ii) matrix-multiplication with an $h_i \times \omega$ matrix times an $\omega \times \omega$ matrix ($h_i$ varies with the stage)

(iii) rank-$\omega$ update to an $h_i \times h_i$ matrix

The operation count is

$$\Omega \approx (1 + \frac{2}{k^2})\frac{2n^3}{3}$$

where $k = n/\omega$. The redundancy can be significant when $k$ is small.
Operation distribution

- G-J: \(2n\omega^2\) (\(k\) calls of \(2\omega^3\) each); quadratically increasing in \(\omega\)

- MXM:
  \[
  n^2\omega + \frac{n\omega}{2} - n\omega^2 - \frac{n^2}{2};
  \]
  increasing until \(\omega = n/2\) then decreasing

- Rank-\(\omega\):
  \[
  \frac{2}{3}n^3 - n^2\omega + \frac{n}{3}\omega^2;
  \]
  decreasing but dominant unless \(k\) is small

The fraction of the total operations performed via the G-J primitive (the slowest) is

\[
\frac{3}{2 + k^2}
\]

Therefore, we want \(\omega\) large enough so all primitives perform well but not so large as to cause degradation by placing too many operations in the slower primitives.
Data transfer analysis

Suppose the i-th call to the

- mxm primitive requires \( l_i^m \) loads and \( \sigma_i^m \) operations
- rank-\( \omega \) primitive requires \( l_i^R \) loads and \( \sigma_i^R \) operations
- G-J primitive requires \( l_i^{GJ} \) loads and \( \sigma_i^{GJ} \) operations.

We then have

\[
\mu = \frac{l_{GJ}}{\sigma} + \sum_{i=1}^{k-1} \frac{l_i^m}{\sigma} + \sum_{i=1}^{k-1} \frac{l_i^R}{\sigma}
\]

where \( \sigma \) is the total number of operations and \( l_{GJ} = \sum_{i=1}^{k} l_i^{GJ} \)

This can be rewritten in terms of local \( \mu \)'s as

\[
\mu = \gamma_{GJ} \mu_{GJ} + \sum_{i=1}^{k-1} \gamma_i^m \mu_i^m + \sum_{i=1}^{k-1} \gamma_i^R \mu_i^R
\]

where \( \sigma_{GJ} = \sum_{i=1}^{k} \sigma_i^{GJ} \), \( \gamma_{GJ} = \sigma_{GJ}/\sigma \), \( \gamma_i^m = \sigma_i^m/\sigma \), \( \gamma_i^R = \sigma_i^R/\sigma \), \( \mu_i^m = l_i^m/\sigma_i^m \), and \( \mu_i^R = l_i^R/\sigma_i^R \).
The global $\mu$ is therefore a linear combination of the local $\mu$'s where the weights are the fraction of operations performed in the $i$-th instance of the local primitive.

Define,

$$\gamma_R = \sum_{i=1}^{k-1} \gamma_i^R$$

and

$$\mu_R = \frac{1}{\gamma_R} \sum_{i=1}^{k-1} \gamma_i^R \mu_i^R$$

It can be shown that for $1 \leq \omega \leq \bar{\omega} \approx \sqrt{CS}$

$$\mu = \gamma_R \mu_R + \eta$$

where $\eta \approx C/n$ for some constant $C$ and $\gamma_R \leq 1$ (typically $\approx 1$).
We then have

1. if $\omega$ is small
   \[\mu \approx \frac{\gamma R}{2\omega}\]

2. if $\omega$ is moderate
   \[\mu \approx \frac{\gamma R}{\omega}\]

3. $\omega = (CS)/(\sqrt{CS} + p)$
   \[\mu \approx \frac{\gamma R}{\sqrt{CS}}\]

4. if $\omega > (CS)/(\sqrt{CS} + p)$ then $\mu_{GJ}$ begins to dominate.
Note the conflict between the arithmetic time and data transfer optimizations. $T_a$ wants $\omega$ only large enough to get good local performance and a good distribution of operations

$$\Delta_l/T_a \text{ wants } \omega \to CS(\sqrt{CS} + p)^{-1}.$$  

This conflict is seen in the optimal $\omega$ relative to a given $n$. Two-level blocking can be used to mitigate the conflict.
Double Level Blocking

Outer-to-inner: In single level algorithm with blocksize $w < w$
replace BLAS2-based kernels
with BLAS3-based kernels

\[
\begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array}
\]

For the generalized block LU under consideration
this means replacing the GJ kernel with a block
LU factorization of $A_{11}$ using a blocksize $q < w$

This does not alter data loading for this form
of LU but it does improve the efficiency within
cache.
Double Level Blocking

**Inner-to-outer:** Regroup several small BLAS3 primitives in the single level algorithm using blocksize $q$ into one using blocksize $w$.

For the generalized block LU under consideration this means grouping several rank $q$ updates into a single rank $w$ update.

This is done by only performing part of each rank $q$ update — the part which affects the matrices which deving the rank $w$ update.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
</tbody>
</table>

\[ w = 2q \]

\[
\begin{align*}
A &\leftarrow A^{-1} \\
D &\leftarrow DA \\
E &\leftarrow EA \\
C &\leftarrow C - DB \\
F &\leftarrow F - EB \\
H &\leftarrow H - DJ \\
C &\leftarrow C^{-1} \\
F &\leftarrow FC \\
\text{Then a rank } w \text{ update of } G
\end{align*}
\]
Next we present ways to generate BLAS3-based versions of the standard LU factorization.

We will consider the two techniques used in the notes already:

1. a recursive formulation based on equating terms in a partitioned form of the factorization.

2. manipulation of the operator definition of the factorization,

The first is used to derive a version based on an immediate application of a rank-\(\omega\) update. The second is used to derive a block version of the delayed update version based on triangular solves.
We have seen a block generalization of the rank-1 based LU factorization. Next a similar version is derived that produces the standard LU factorization.

Partition $A = LU$ as follows:

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
= 
\begin{pmatrix}
L_{11} & 0 \\
L_{21} & L_{22}
\end{pmatrix}
\begin{pmatrix}
U_{11} & U_{12} \\
0 & U_{22}
\end{pmatrix}
$$

where $A_{11}$, $L_{11}$, and $U_{11}$ are all $\omega \times \omega$ and the rest are partitioned conformally.

Equating terms yields

$$
\begin{align*}
A_{11} &= L_{11}U_{11} & A_{12} &= L_{11}U_{12} \\
A_{21} &= L_{21}U_{11} & A_{22} &= L_{22}U_{22} + L_{21}U_{12}
\end{align*}
$$

This yields the following basic step for the algorithm

- compute $L_{11}U_{11} = A_{11}$ ($\omega \times \omega$ LU factorization)
- solve $U_{11}^T L_{21}^T = A_{21}^T$ (triangular solve with multiple rhs)
- solve $L_{11}U_{12} = A_{12}$ (triangular solve with multiple rhs)
- update $A_{22} \leftarrow L_{22}U_{22} = A_{22} - L_{21}U_{12}$ (rank-$\omega$ update)
The factorization is completed by applying the procedure recursively to $A_{22}$ to compute $L_{22}$ and $U_{22}$.

The algorithm has a similar update pattern to the rank-1 based approach. The next $\omega$ rows of $U$ and columns of $L$ are computed on each block step.

Also note that if this approach is compared to the generalized block LU factorization that uses $L_{11}U_{11} = A_{11}$ instead of Gauss-Jordan to compute $A_{11}^{-1}$ it is seen that the two are only separated by a simple application of associativity.
Block form of LU factorization

Active portion of the matrix for the step \( i+1 \) — A \(_i\)

(i–1) \( W \) columns of L

next \( W \) columns of L

(i–1) \( W \) rows of U completed and not touched

next \( W \) rows of U computed

remaining portion of A updated using the computed columns of L and rows of U

got s rank— \( W \) update
Recall that the version based on BLAS2 reduction operations manipulated the definition of the factorization in terms of the application of the elementary triangular matrices $M_i^{-1}$. A BLAS3-based version can be derived in a similar fashion.

Define $A = [a_1, \ldots, a_n]$ where $a_i$ is the i-th column of $A$ and

$$a_i^j = M_{j}^{-1} \cdots M_2^{-1} M_1^{-1} a_i$$

The BLAS2 version discussed earlier assumed that

$$M_1 \cdots M_{i-1}$$

were all known as well as the first $i - 1$ columns of $U$ at the start of the i-th step. The i-th step consisted of two main parts

1. produce $a_i^i$ from $a_i$

2. produce $l_i$ and $U e_i$ from $a_i^i$

These can be generalized to use BLAS3 primitives.
Suppose $n = k\omega$. Then we can define a k-step procedure which works with $\omega$ columns at a time to produce $LU$.

Assume that at step $j$ we know $M_1 \cdots M_{i-1}$ and the first $i-1$ columns of $U$.

The first part of step $j$ produces $a_{i-1}^i, a_{i+1}^i, \cdots, a_{i+\omega-1}^i$ from $a_i, \cdots, a_{i+\omega-1}$ by solving

$$(M_1M_2\cdots M_{i-1})[a_{i-1}^i, a_{i+1}^i, \cdots, a_{i+\omega-1}^i] = [a_i, \cdots, a_{i+\omega-1}]$$

This is a structured unit lower triangular system as in the BLAS2-based version but it is solved with $\omega$ righthand-side vectors.
After the first part of step k the matrix $A$ has been transformed into the form (omitting the storage of $L$ from the diagram)
\( M_i, \ldots M_{i+\omega-1} \) and the next \( \omega \) columns of \( U \) can now be produced from the updated columns of \( A \) via a BLAS2-based LU factorization (or BLAS3-based with smaller blocksize if double level blocking is used). The LU factorization of the updated \( \omega \) columns transforms \( A \) into the form

[Diagram of LU factorization]

This approach is typically used on out-of-core solvers with column major ordering assumed where it is more efficient than the rank-\( \omega \) based approach.