

Lesson 20-22: GE with Partial Pivoting

L20-1

The most common way to solve the nonsingular system, $Ax = b$, on the computer is to use the algorithm called Gaussian Elimination with partial pivoting.

This algorithm produces a factorization

$$\underline{PA = LU}$$

where P is a unitary permutation matrix, L is a unit lower Δ matrix, and U is an upper Δ matrix. We will derive this factorization and show it requires $O(n^3/3)$ floating point operations to compute.

But, first, let's see how to solve $Ax = b$ once we have $PA = LU$.

$$Ax = b \Rightarrow$$
$$\underline{PA}x = Pb \Rightarrow$$

$$LUx = Pb = \hat{b}$$

① $\hat{b} = Pb$ (rearranging rows of b)

② Solve $Ly = \hat{b}$ $O(n^2/2)$ mults

$$\begin{array}{c} \Delta \\ L \end{array} | = | \hat{b}$$

by forward substitution:

$$y_1 = \hat{b}_1$$

for $i = 2, n$ do

$$y_i = \hat{b}_i - \sum_{j=1}^{i-1} l_{ij} y_j$$

③ Solve $Ux = y$ $O(n^2/2)$ mults

$$\nabla | = |$$

by back substitution:

$$x_n = y_n / u_{nn}$$

for $i = n-1$ down to 1 do

$$x_i = (y_i - \sum_{j=i+1}^n u_{ij} y_j) / u_{ii}$$

Summary : $O(n^3/3)$ mults to get P, L, U
 $O(n^2/2)$ " " get y
 $O(n^2/2)$ " " " x

So, if we had n different right hand sides $b^{(i)}$, $i=1, 2, \dots, n$, we find P, L, U only once, and then solve for each

System $Ax^{(i)} = b^{(i)}$ ($LUx^{(i)} = \hat{b}^{(i)}$)

using the forward + backward substitution.

Total cost for n systems is

$$\frac{n^3}{3} + n(n^2/2 + n^2/2) = \underline{O(n^3)} \text{ mults.}$$

Total cost for k systems is

$$\frac{n^3}{3} + k(n^2/2 + n^2/2)$$

Many applications require the same matrix, but different right hand sides. We can avoid the re-factoring for systems $2, 3, \dots, k$ at great savings, and solve n problems "as cheaply as" only 1!

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ .3 & 1 & 0 \\ -1.5 & 0 & 1 \end{pmatrix} P_1 \begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} = \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & -1 & 6 \end{pmatrix}$$

P_2 L_1 A

L_{20-5}
 $\uparrow P_2 L_1 P_1 A$
 need now to
 put a zero
 where -1 is.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & .04 & 1 \end{pmatrix} \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & -1 & 6 \end{pmatrix} = \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.2 \end{pmatrix}$$

$\uparrow L_2$ $P_2 L_1 P_1 A$ U
 elementary unit lower Δ upper Δ ular.

We have

$$L_2 P_2 L_1 P_1 A = U$$

$$P_2 L_1 P_1 A = L_2^{-1} U$$

$$P_2 L_1 \underbrace{P_2^T}_{I} P_2 P_1 A = L_2^{-1} U$$

$$(P_2 L_1 P_2^T) P_2 P_1 A = L_2^{-1} U$$

Luck #1: $P_2 L_1 P_2^T = \begin{pmatrix} 1 & 0 & 0 \\ -.5 & 1 & 0 \\ .3 & 0 & 1 \end{pmatrix}$, unit lower elementary
 (Same #s as L_1 just reordered)

Luck #2: $L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -.04 & 1 \end{pmatrix}$

L20-6

Luck #3: $(P_2 L_1 P_2^T)^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0.3 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & 0 & 1 \end{pmatrix}$

Note: The inverse of a unit lower (or upper) elementary Δ ular matrix is again a unit lower (or upper) elementary Δ ular matrix with the elements below (or above) the diagonal negated!

So, $P_2 P_1 A = (P_2 L_1 P_2^T)^{-1} L_2^{-1} U$

$P_2 P_1 A = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & 0 & 1 \end{pmatrix}}_P \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -0.04 & 1 \end{pmatrix} U$

Another permutation matrix

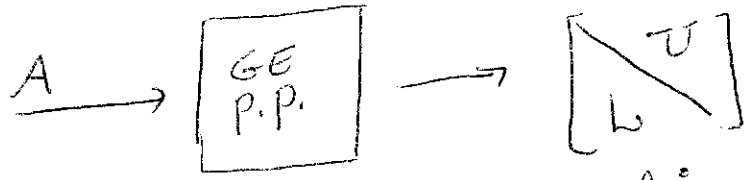
Luck #4: This is a unit lower (but not elementary) Δ ular matrix with the same elements below diagonal "joined" in same locations

$PA = \begin{pmatrix} 1 & 0 & 0 \\ +0.5 & 1 & 0 \\ -0.3 & -0.04 & 1 \end{pmatrix} * \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.2 \end{pmatrix}$

$L \qquad U$

#s in L are the elements to be zeroed divided by the diagonal (pivot) element.

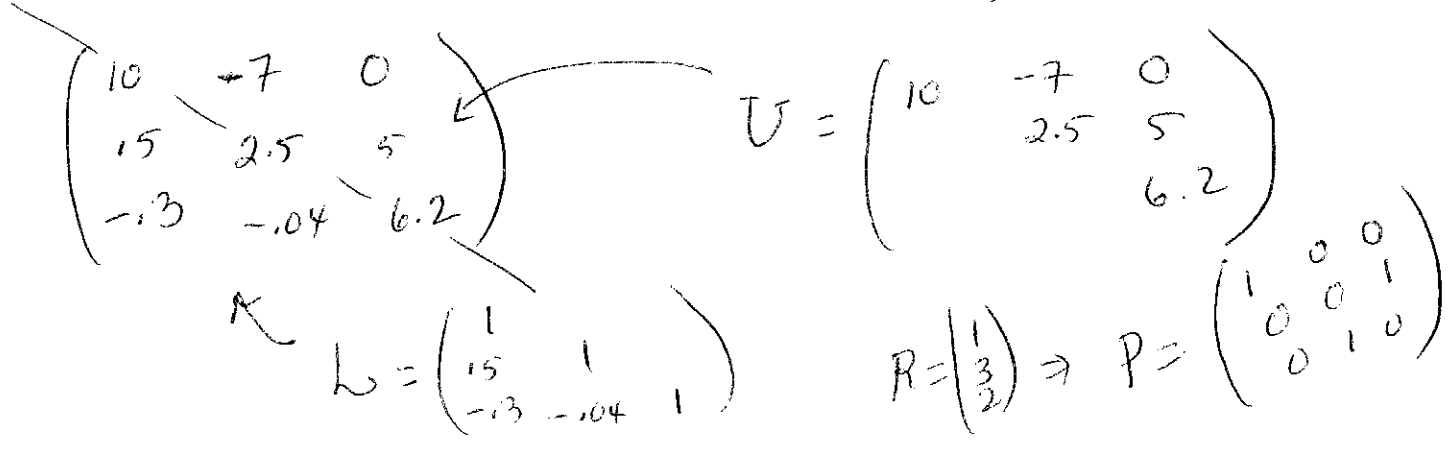
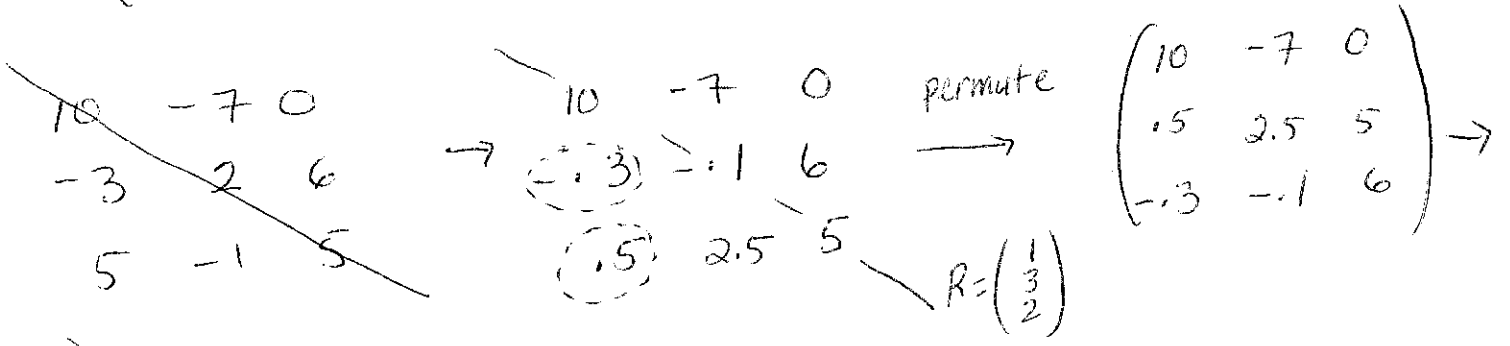
On the computer, we do not have a data structure called L or U. We overwrite the matrix A.



A:
 (stored over top of A)
 (u_{11}, u_{22}, \dots stored on A's diagonal)

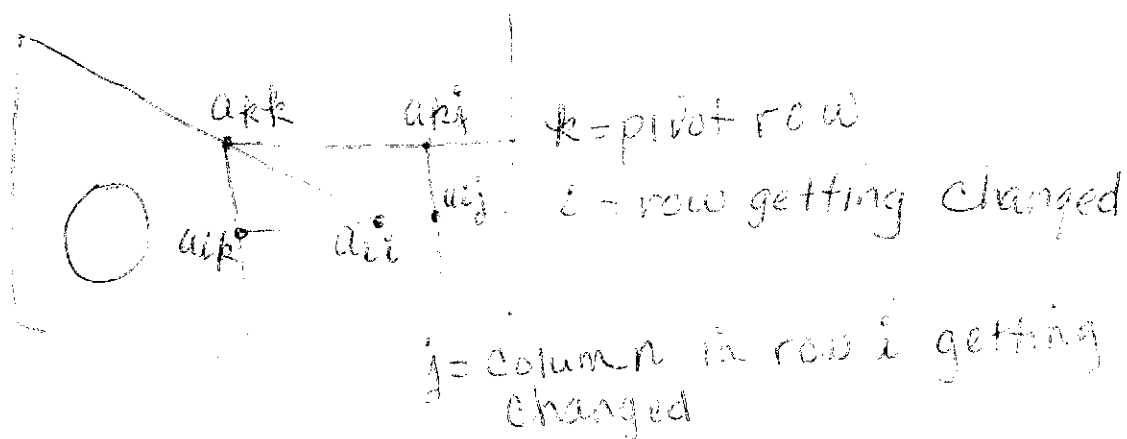
The permutation "matrix" can be stored as an integer vector R. $R(i)=j$ means original row j is now in row i.

For the 3x3 example, we start with $R = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, no permuting done yet.



GE WITH PARTIAL PIVOTING

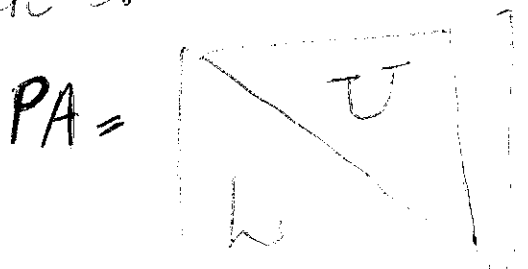
L20-8



$a_{ik} \rightarrow$ will become zero this step, so instead of storing a zero there, we store the multiplier, which is the respective component of L .

ON INPUT: $A(n \times n)$ dense matrix

ON OUTPUT: A contains L and U , the original entries of A have been overwritten.



Note: diagonal of L has ones, so diagonal of A contains the diagonal of U on output.

```

P matrix [ For k=1, n do
           end   R(k)=k

```

```

For k = 1, 2, n-1 do (pivot row)

```

Find the row to be pivot row

```

imax = k
For irow = k+1, n do
  if |airow, k| > |aimax, k| then
    imax = irow
  end
end

```

Interchange row k with the row found above

```

For j = 1 to n do
  t = akj
  akj = aimax, j
  aimax, j = t
end

```

update P matrix { d=R(k)
R(k)=R(imax)
R(imax)=d }

rows to update {

```

For i = k+1, n do
  aik = aik / akk

```

columns in row i to change {

```

For j = k+1, n do
  aij = aij - aik * akj
end

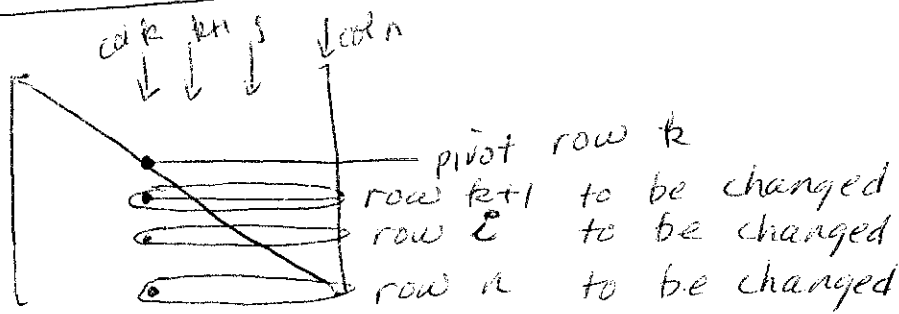
```

- end

end

WORK: O(N³/3)

MATLAB VERSION OF GE. W. P.P



```

For k=1, n do
    R(k)=k
end
} or R=1:n => R = [ 1
                    2
                    3
                    :
                    n ]
    
```

```

For k=1, 2, ..., n-1
    (pivot rows)
    
```

```

[maxi, imax] = max(abs(A(k:n, k)))
    
```

← find max element in col k from the diag down. store the row of this element in imax.

```

C = A(k, :);
    
```

← store row k of A in C.

```

A(k, :) = A(imax, :);
    
```

← row k of A is what row imax used to be

```

A(imax, :) = C;
    
```

← row imax of A is now what used to be row k.

```

d = R(k);
R(k) = R(imax);
R(imax) = d;
    
```

} record in the permutation

```

For i=k+1, n do
    
```

← update row i

```

A(i, k) = A(i, k) / A(k, k);
    
```

← multiplier

```

A(i, k+1:n) = A(i, k+1:n) - A(i, k) * A(k, k+1:n);
    
```

new row i previous row i multiplier pivot row

end

Let's now discuss how accurately we can expect our computed solution from G.E.P.P. to be.

Example :

$$\begin{aligned} .780x + .563y &= .217 \\ .457x + .330y &= .127 \end{aligned}$$

With a 3 digit (base 10) machine, G.E.P.P.

gives

$$\begin{aligned} \tilde{x} &= 1.71 \\ \tilde{y} &= -1.98 \end{aligned}$$

As the values for x and y . Using these values, we compute the residuals

$$r_1 = .217 - (.780(1.71) + .563(-1.98))$$

$$r_1 = \underline{\underline{-.00206}}$$

and

$$r_2 = -[.457(1.71) + .33(-1.98)] + .127$$

$$r_2 = \underline{\underline{-.00107}}$$

r_1 and r_2 are small for our 3-digit machine. We can't expect much better!

Does that mean \tilde{x} and \tilde{y} are close to the true x and y ?

The answer is no since

$$x^* = 1.000$$

$$y = -1.000$$

So $\tilde{x} + \tilde{y}$ are really inaccurate!

Note: Small residuals do not imply correct (accurate to ϵ_{mach} or even a few digits accurate!) solutions.

In fact $\frac{\|x - \tilde{x}\|}{\|x\|} = .71$ here!

The problem is that the 2 lines in the example are nearly parallel, so the problem is nearly singular \Rightarrow poorly conditioned.
So how much accuracy can we expect?

Thm: Wilkinson; The computed solution \tilde{x} from GE. P.P. satisfies

$$(A + E)\tilde{x} = b \quad \text{where}$$

$$\frac{\|E\|}{\|A\|} = c \epsilon_{mach}$$

Wilkinson's theorem says that \tilde{x} exactly solves a nearby problem to $Ax = b$.

So, we know GE p.p. is backward stable. This means,

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq c \cdot K(A) \cdot \epsilon_{\text{mach}}$$

where $K(A) = \|A\| \cdot \|A^{-1}\|$.

So, for our example $\epsilon_{\text{mach}} = 10^{-3}$ and $K(A)$ was around 10^3 , so no digits of accuracy can be expected.

Note: If $\frac{\|x - \tilde{x}\|}{\|x\|} \leq 10^{-d}$, then we expect roughly d digits of accuracy since it is the relative error that is related to accuracy.

But the residuals in the example were small. Can that be explained?

$$r = b - A\tilde{x}$$

$$r = b - (b - E\tilde{x}) = E\tilde{x} \quad \text{by Wilkinson}$$

$$\text{So } \|r\| \leq \|E\| \cdot \|\tilde{x}\| \leq c \|A\| \cdot \|\tilde{x}\| \epsilon_{\text{mach}} \quad \text{by Wilkinson}$$

$$\text{So } \frac{\|r\|}{\|A\| \cdot \|\tilde{x}\|} \leq c \cdot \epsilon_{\text{mach}} \quad \left(\begin{array}{l} \text{a "relative residual" can} \\ \text{be expected to be} \\ O(\epsilon_{\text{mach}}) ! \end{array} \right)$$

$$\text{Also, } r = b - A\tilde{x} = Ax - A\tilde{x} = A(x - \tilde{x}) = Ae$$

where $e = x - \tilde{x}$, the absolute error.

$$\boxed{Ae = r} \quad \text{always true for linear nonsingular systems.}$$

$$\text{So } \|r\| \leq \|A\| \cdot \|e\|.$$

This shows if $\|e\|$ small ($\|A\|$ usually $O(1)$), then $\|r\|$ small. But the equation

$$e = A^{-1}r \Rightarrow \|e\| \leq \|A^{-1}\| \|r\| \quad \text{does not let us}$$

conclude that if residuals are small, then errors are small since $\|A^{-1}\|$ usually large for ill-conditioned problems.

(Assuming original problem was such that $\|A\| = O(1)$).

Gaussian Elimination for Special Matrices

Certain problems $Ax=b$, $A_{n \times n}$ nonsingular do not require pivoting. Two famous classes are diagonally dominant matrices and positive definite symmetric matrices. There are special routines that can take advantage of symmetry (Cholesky decomposition) or of sparseness (for tridiagonal) matrices.

The Cholesky algorithm is developed by partitioned matrix multiplication. Before doing this, let's develop our original $A=LU$ row-wise algorithm (no pivoting) by simple matrix multiplication.

Submatrix Derivation

$$\text{Let } A = A^{(1)} = \begin{bmatrix} a_{11} & a_{1^T} \\ \hat{a}_1 & A_{11} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \underline{l}_1 & L_1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{1^T} \\ & U_1 \end{bmatrix}.$$

We know a_{11} , a_{1^T} , \hat{a}_1 , A_{11} and we want an algorithm for computing the L and U , and in the end, we want the $L+U$ stored over-top of A as the algorithm proceeds row by row.

Simple multiplication shows,

$$u_{11} = a_{11}, \quad u_{1^T} = a_{1^T} \quad (\text{already know 1st row of } U!)$$

Note: The pivot row remains unchanged!

More multiplication: $\underline{l}_1, u_{11} = \hat{a}_1 \Rightarrow \underline{l}_1 = \hat{a}_1 / u_{11}$

We can get all the multipliers at once:

$$\underline{l}_1 = \hat{a}_1 / u_{11} \quad \text{OR}$$

$$\underline{l}_1 \leftarrow \hat{a}_1 = \hat{a}_1 / a_{11}$$

store it back in \hat{a}_1 location.

More multiplication:

$$\underline{L}_1 \underline{u}_1^T + L_1 U_1 = A_{11} \Rightarrow L_1 U_1 = A_{11} - \underline{L}_1 \underline{u}_1^T$$

or

$$L_1 U_1 = A_{11} - \begin{matrix} \hat{a}_1 \\ \uparrow \\ \text{stored} \\ \text{there now} \end{matrix} \begin{matrix} \hat{a}_1^T \\ \uparrow \\ \text{stored} \\ \text{there} \end{matrix}$$

$$L_1 U_1 = \begin{bmatrix} a_{22} & a_{23} & \dots & a_{2n} \\ a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} - \begin{bmatrix} a_{21} \\ a_{31} \\ \vdots \\ \hat{a}_{n1} \end{bmatrix} \begin{bmatrix} a_{12} & a_{13} & \dots & a_{1n} \end{bmatrix}$$

\uparrow A_{11} \uparrow \hat{a}_1 as overwritten
 \uparrow \hat{a}_1^T as overwritten

Note: The right hand side shows the new submatrix that will replace A_{11} 's location is simply row operations. (The first new row is the first old row minus the 1st multiplier times the pivot row.)

Note: One could also perform the right hand side as column operations if desired. (The new first col is the old first col minus a_{12} times the multiplier column, etc.)

Note: The left hand side is the LU factorization of the right hand side \Rightarrow we now should

in step 2 to attempt to factor the new right hand side which is now stored in the A_{11} position. The algorithm is recursive, so let's look at step k :

$$A^{(k)} = \begin{bmatrix} 1 & \\ \underline{l}_k & L_k \end{bmatrix} \begin{bmatrix} u_{kk} & u_k^T \\ & U_k \end{bmatrix} = \begin{bmatrix} a_{kk} & \underline{a}_k^T \\ \hat{a}_k & A_{kk} \end{bmatrix}$$

$$\Rightarrow u_{kk} = a_{kk}, \quad \underline{u}_k^T = \underline{a}_k^T \quad \left(\begin{array}{l} \text{The } k\text{th row of } U \text{ is} \\ \text{already known and} \\ \text{is the present} \\ \text{pivot row.} \end{array} \right)$$

$$l_k = \hat{a}_k / u_{kk} \Rightarrow$$

$$1) \quad \underline{\hat{a}}_k \leftarrow \underline{\hat{a}}_k / a_{kk} \quad (\text{store multiplier vector})$$

$$L_{kk} U_{kk} = A_{kk} - \underline{l}_k \underline{u}_k^T$$

$$A^{(k+1)} = L_{kk} U_{kk} = A_{kk} - \underline{\hat{a}}_k \underline{a}_k^T \quad 2)$$

Left
Submatrix
to factor

$$1) \Rightarrow \left[\begin{array}{l} \text{For } i = k+1, n \\ a_{ik} = a_{ik} / a_{kk} \end{array} \right.$$

$$2) \Rightarrow \left[\begin{array}{l} \text{For } i = k+1, n \\ \left[\begin{array}{l} \text{For } j = k+1, n \\ a_{ij} = a_{ij} - a_{ik} * a_{kj} \end{array} \right. \end{array} \right.$$

GE.P.P (NO pivoting)

For $k=1, n-1$ do

[For $i=k+1, n$
 $a_{ik} = a_{ik}/a_{kk}$
For $i=k+1, n$
 [For $j=k+1, n$
 $a_{ij} = a_{ij} - a_{ik} * a_{kj}$

⇓

For $k=1, n-1$ do

[For $i=k+1, n$
 $a_{ik} = a_{ik}/a_{kk}$
 [For $j=k+1, n$
 $a_{ij} = a_{ij} - a_{ik} * a_{kj}$

kij -
form

So, $L_{k-1} \cdot \underline{L}_k = \underline{A}_k$ leads to

```

For i = 1, 2, ..., k-1 do
    
$$a_{ki} \leftarrow (a_{ki} - \sum_{j=1}^{i-1} a_{ij} a_{kj}) / a_{ii}$$

end

```

Multiply:

$$d_{kk} = \underline{L}_k^T \underline{L}_k + R_{kk}^2 \Rightarrow$$

$$R_{kk} = (d_{kk} - \underline{L}_k^T \underline{L}_k)^{1/2}$$

Cholesky (Crout) Algorithm (L stored in lower Δ of A)
 $O(n^3/3)$ mult + additions
 (1/2 the work and storage of $A=LU$)

```

 $a_{ii} \leftarrow \sqrt{a_{ii}}$ 
For k = 2, n do
    For i = 1, k-1 do
        
$$a_{ki} \leftarrow (a_{ki} - \sum_{j=1}^{i-1} a_{ij} a_{kj}) / a_{ii}$$

    end
    
$$a_{kk} \leftarrow (a_{kk} - \sum_{j=1}^{k-1} a_{kj}^2)^{1/2}$$

end

```