Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Spring 2018

Instructor: Dr. Sateesh Mane

April 20, 2018

10 Lecture 10

10.1 Applied linear algebra Part 2

We continue the study how to solve a set of coupled linear equations in several variables.

In this lecture we study (some) matrix algorithms.

10.2 Matrix equations

The set of linear equations we wish to solve are

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \quad + a_{1n}x_n = b_1\,; \tag{10.2.1a}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \quad + a_{2n}x_n = b_2\,; \tag{10.2.1b}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \quad + a_{3n}x_n = b_3\,; \tag{10.2.1c}$$
$$\vdots \qquad\qquad\qquad\qquad\quad = \vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \quad + a_{nn}x_n = b_n\,: \tag{10.2.1d}$$

We can express the above in matrix form as

$$Ax = b\,: \tag{10.2.2}$$

Here A is an n n square matrix and x and b are n 1 column vectors.

The entries in A and b are known inputs and we wish to solve for x.

In the above case

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}; \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}: \tag{10.2.3}$$

Let us examine how this is related to Gaussian elimination (with pivoting).

Question: Why not calculate the matrix inverse A^{-1}? Then the solution is

$$x = A^{-1}b\,: \tag{10.2.4}$$

Answer: Computing the inverse of a matrix is computationally expensive and the numerically accuracy of the answer is frequently of poor quality.

It can even be slower to calculate the inverse A^{-1} than to employ other matrix solution algorithms.

10.3 Two unknowns

Before proceeding with techniques for an arbitrary number of unknowns n, it is worthwhile to study the simple case of two unknowns.

The matrix equation $Ax = b$ has the form

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix} \quad \begin{matrix} x_1 \\ x_2 \end{matrix} = \begin{matrix} b_1 \\ b_2 \end{matrix} : \tag{10.3.1}$$

In this simple case it *is* easy to calculate the inverse matrix A^{-1} (and to determine if the inverse exists in the first place).

The determinant of A is

$$= \det(A) = a_{11}a_{22} - a_{12}a_{21} : \tag{10.3.2}$$

If $= 0$ the matrix A is not invertible.

If the matrix A is not invertible, we say that A is **singular.**

If A is invertible (non-singular), then $\neq 0$ and the inverse matrix is

$$A^{-1} = \frac{1}{} \begin{matrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{matrix} : \tag{10.3.3}$$

The solution of eq. (10.3.1) is

$$x = A^{-1}b : \tag{10.3.4}$$

Written out in full, the solution is

$$\begin{matrix} x_1 \\ x_2 \end{matrix} = \frac{1}{} \begin{matrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{matrix} \begin{matrix} b_1 \\ b_2 \end{matrix} = \frac{1}{} \begin{matrix} a_{22}b_1 - a_{12}b_2 \\ -a_{21}b_1 + a_{11}b_2 \end{matrix} : \tag{10.3.5}$$

If the matrix A is not invertible, then the equations in eq. (10.3.1) are either inconsistent else they are not linearly independent.

1. If they are inconsistent, there is no solution.
2. If they are not linearly independent, there are infinitely many solutions (no unique solution).
3. Either way, we exit and report an error.

The solution in eq. (10.3.5) will be unreliable if the equations are ill conditioned.

10.4 Gaussian elimination: review

We review Gaussian elimination with partial pivoting.

To illustrate the main ideas, consider the following set of equations in three variables $(x_1; x_2; x_2)$:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1\,; \tag{10.4.1a}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2\,; \tag{10.4.1b}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3\,: \tag{10.4.1c}$$

The permutations of the rows (due to pivoting and swapping) are saved in an array, say $(s_1; : : : ; s_n)$. In the present example $n = 3$.

After elimination using partial pivoting, the processed equations have the following form.

$$u_{s_1 1}x_1 + u_{s_1 2}x_2 + u_{s_1 3}x_3 = b_{s_1}\,; \tag{10.4.2a}$$
$$a^{0}_{s_2 2}x_2 + a^{0}_{s_2 3}x_3 = b^{0}_{s_2}\,; \tag{10.4.2b}$$
$$a^{00}_{s_3 3}x_3 = b^{00}_{s_3}\,: \tag{10.4.2c}$$

Backsubstitution is employed to solve for x_3, then x_2 and finally x_1.

The bad feature of eq. (10.4.2) is of course that the entries on the right hand side are not the original values b_i. Instead, they are the processed values b_{s_1}, $b^{0}_{s_2}$ and $b^{00}_{s_3}$. The processing depends on the details of the Gaussian elimination for the coefficients a_{ij}.

We wish to find an algorithm where we can separate the processing of the coefficients a_{ij} from the right hand side values b_i.

It is possible to do this, in an elegant manner.

1. It was mentioned previously that Gaussian elimination does not require extra storage.
2. We can overwrite the original matrix.
3. The matrix coefficients a_{ij} are processed into an upper triangular matrix U, but that occupies only about half of the space of the original matrix A.
4. The trick is to store the additional data required in a *lower triangular matrix* L.
5. There is enough space in the lower triangle of A to store the required information.

The **LU decomposition** algorithm performs the required tasks.

The algorithm is also called **LU factorization**.

10.5 LU decomposition Part 1

We set aside Gaussian elimination for the moment and analyze matrix equations in general.

Suppose we have a very special set of linear equations to solve, which have the following form.

$$LUx = \quad : \tag{10.5.1}$$

Here "LU" is a product of two n n square matrices L and U, where L is a lower triangular matrix and U is an upper triangular matrix.

The solution of eq. (10.5.1) proceeds using two backsubstitution steps.

1. First de ne a temporary column vector y and solve the equation

$$Ly = \quad : \tag{10.5.2}$$

2. This is accomplished via backsubstitution where we solve for the entries in the forward order $y_1; : : : ; y_n$.
3. Next we solve the equation

$$Ux = y : \tag{10.5.3}$$

4. This is accomplished via backsubstitution where we solve for the entries in the reverse order $x_n; : : : ; x_1$.
5. The solution for x is the solution of eq. (10.5.1).

Returning to Gaussian elimination (with partial pivoting and swapping of rows), we shall see how the algorithm can be reformulated using LU decomposition. This is a good thing to do.

10.6 Backsubstitution

The word \backsubstitution" has been used many times, but no actual formula has been written down how to do it.

Suppose L and U are lower and upper triangular matrices, respectively. Let us write explicit formulas how to perform backsubstitution to solve the equations

$$Ux = \quad ; \qquad Ly = \quad : \tag{10.6.1}$$

Suppose we have an n n upper triangular matrix U as follows

$$U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{pmatrix} : \tag{10.6.2}$$

1. The solution of the equation $Ux =$ is obtained in **reverse order** by solving for x_n rst and working backwards to x_1.
2. The solution for x_n is given by
$$x_n = \frac{\ }{u_{nn}} : \tag{10.6.3}$$
3. The solution for x_i, where $i = n \quad 1;\ldots;1$, is given by
$$x_i = \frac{1}{u_{ii}} \quad \Big(\quad \sum_{j=i+1} u_{ij} x_j \Big) \qquad (i = n \quad 1;\ldots;1) : \tag{10.6.4}$$

Suppose we have an n n lower triangular matrix L as follows

$$L = \begin{pmatrix} \ell_{11} & 0 & \cdots & \cdots & 0 \\ \ell_{21} & \ell_{22} & 0 & \cdots & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn} \end{pmatrix} : \tag{10.6.5}$$

1. The solution of the equation $Ly =$ is obtained in **forward order** by solving for y_1 rst and working forwards to y_n.
2. The solution for y_1 is given by
$$y_1 = \frac{1}{\ell_{11}} : \tag{10.6.6}$$
3. The solution for y_i, where $i = 2;\ldots;n$, is given by
$$y_i = \frac{1}{\ell_{ii}} \quad \Big(\quad \sum_{j=1}^{i-1} \ell_{ij} y_j \Big) \qquad (i = 2;\ldots;n) : \tag{10.6.7}$$

10.7 LU decomposition Part 2

It is simplest to explain the algorithm by working through a simple example.

Consider the following set of equations in three unknowns $(x_1; x_2; x_3)$.

$$x_1 \quad 2x_2 + x_3 = b_1; \tag{10.7.1a}$$
$$2x_1 \quad x_2 \quad 4x_3 = b_2; \tag{10.7.1b}$$
$$4x_1 \quad x_2 \quad 2x_3 = b_3: \tag{10.7.1c}$$

We express this is matrix form $Ax = b$ as

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 1 & 4 \\ 4 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}: \tag{10.7.2}$$

We focus on the matrix A itself. That is where the processing occurs.

The array of swap indices is initialized to $(1; 2; 3)$.

The first step is to compute the scaled values to determine the initial pivot

$$\frac{|a_{11}|}{\hat{a}_1} = \frac{1}{2}; \qquad \frac{|a_{21}|}{\hat{a}_2} = \frac{2}{4} = \frac{1}{2}; \qquad \frac{|a_{31}|}{\hat{a}_3} = \frac{4}{4} = 1: \tag{10.7.3}$$

Hence we set $r_1 = 3$ and swap rows 1 and 3. The array of swap indices is updated to $(3; 2; 1)$.

The swapped matrix is (call it A_1)

$$A_1 = \begin{pmatrix} 4 & 1 & 2 \\ 2 & 1 & 4 \\ 1 & 2 & 1 \end{pmatrix}: \tag{10.7.4}$$

We now subtract multiples of row 1 from rows $2; : : : ; n$, to eliminate the entries in column 1 in rows $2; : : : ; n$. In this case it is rows 2 and 3.

1. We subtract $\frac{1}{2}$ times row 1 from row 2. We subtract $\frac{1}{4}$ times row 1 from row 3.
2. We obtain (the numbers are displayed as fractions to avoid issues of roundoff error)

$$A_2 = \begin{pmatrix} 4 & 1 & 2 \\ 0 & \frac{1}{2} & 3 \\ 0 & \frac{7}{4} & \frac{3}{2} \end{pmatrix}: \tag{10.7.5}$$

We now fill the empty slots with data to construct the lower triangular matrix L.

1. **The multiplier to subtract row j from row i is placed in the slot A_{ij}.**
2. Note that $i > j$ in all cases, because we swapped the rows during the partial pivoting. The swapping of the rows is essential to the algorithm.

Hence the filled matrix is

$$A_3 = \begin{pmatrix} 4 & 1 & 2 \\ \mathbf{\frac{1}{2}} & \frac{1}{2} & 3 \\ \mathbf{\frac{1}{4}} & \frac{7}{4} & \frac{3}{2} \end{pmatrix}. \tag{10.7.6}$$

1. Note that the entries in red are for bookkeeping purposes.
2. They are not to be used to determine the largest pivot.
3. **Their values must not be changed when subtracting one row from another.**

Next we determine the pivot r_2. To accomplish this we compute the scaled values

$$\frac{|a'_{22}|}{s'_2} = \frac{1/2}{3} = \frac{1}{6}; \qquad \frac{|a'_{32}|}{s'_3} = \frac{7/4}{7/4} = 1: \tag{10.7.7}$$

Hence we set $r_2 = 3$ and swap rows 2 and 3. The array of swap indices is updated to $(3; 1; 2)$.

The bookkeeping entries must also be swapped. This is very important.

Hence the matrix after swapping rows 2 and 3 is

$$A_4 = \begin{pmatrix} 4 & 1 & 2 \\ \mathbf{\frac{1}{4}} & \frac{7}{4} & \frac{3}{2} \\ \mathbf{\frac{1}{2}} & \frac{1}{2} & 3 \end{pmatrix}. \tag{10.7.8}$$

We now subtract multiples of row 2 from rows $3; \ldots; n$, to eliminate the entries in column 2 in rows $3; \ldots; n$. In this case it is only row 3.

We subtract $(1/2)/(7/4) = 2/7$ times row 2 from row 3. This yields

$$A_5 = \begin{pmatrix} 4 & 1 & 2 \\ \mathbf{\frac{1}{4}} & \frac{7}{4} & \frac{3}{2} \\ \mathbf{\frac{1}{2}} & 0 & \frac{24}{7} \end{pmatrix}. \tag{10.7.9}$$

We place the multiplier $2/7$ in the slot A_{32}.

Since there are no more rows to process, this yields the final overwritten matrix.

$$A_{LU} = \begin{pmatrix} 4 & 1 & 2 \\ \mathbf{\frac{1}{4}} & \frac{7}{4} & \frac{3}{2} \\ \mathbf{\frac{1}{2}} & \mathbf{\frac{2}{7}} & \frac{24}{7} \end{pmatrix}. \tag{10.7.10}$$

10.8 LU decomposition Part 3

Technically, what we derived above is not the full lower triangular matrix L.

The matrix L also has entries on its main diagonal, but they all equal 1.
Hence they need not be stored explicitly.

For an n n square matrix A, the L and U lower and upper triangular matrices of the LU decomposition of A have the following form.

$$L = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ \alpha_{21} & 1 & 0 & \cdots & 0 \\ \alpha_{31} & \alpha_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & 1 \end{pmatrix}; \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{pmatrix}: \tag{10.8.1}$$

The original matrix A equals the product of LU (up to a permutation of the rows).

Let us verify this for the equations in eq. (10.7.1) using the overwritten matrix in eq. (10.7.10).

1. From eq. (10.7.10), the matrices L and U are

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ \frac{1}{2} & \frac{2}{7} & 1 \end{pmatrix}; \quad U = \begin{pmatrix} 4 & 1 & 2 \\ 0 & \frac{7}{4} & \frac{3}{2} \\ 0 & 0 & \frac{24}{7} \end{pmatrix}: \tag{10.8.2}$$

2. The rst row of LU is just 1 times the rst row of U:

$$(LU)_{\text{row }1} = \begin{array}{ccc} 4 & 1 & 2 \end{array}: \tag{10.8.3}$$

3. The second row of LU is $\frac{1}{4}$ times the rst row of U plus 1 times the second row of U:

$$\begin{aligned} (LU)_{\text{row }2} &= \frac{1}{4} \begin{array}{ccc} 4 & 1 & 2 \end{array} + \begin{array}{ccc} 0 & \frac{7}{4} & \frac{3}{2} \end{array} \\ &= \begin{array}{ccc} 1 & 2 & 1 \end{array}: \end{aligned} \tag{10.8.4}$$

4. The third row of LU is $\frac{1}{2}$ times the rst row of U plus $\frac{2}{7}$ times the second row of U plus 1 times the third row of U:

$$\begin{aligned} (LU)_{\text{row }3} &= \frac{1}{2} \begin{array}{ccc} 4 & 1 & 2 \end{array} + \frac{2}{7} \begin{array}{ccc} 0 & \frac{7}{4} & \frac{3}{2} \end{array} + \begin{array}{ccc} 0 & 0 & \frac{24}{7} \end{array} \\ &= \begin{array}{ccc} 2 & 1 & 4 \end{array}: \end{aligned} \tag{10.8.5}$$

5. Using the array of swap permutations (3;1;2), this agrees with eq. (10.8.6) up to a permutation of the equations.

$$\begin{pmatrix} 4 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 4 \end{pmatrix} \begin{pmatrix} x_3 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_3 \\ b_1 \\ b_2 \end{pmatrix}: \tag{10.8.6}$$

10.9 Permutation matrix

Notice from the example in Sec. 10.8 that the algorithm must also return the array of swap permutations to the calling application. The calling application must know the swap order of the returned rows.

The array of swap indices can also be expressed as a **permutation matrix**.

A **permutation matrix** is a square matrix whose elements are all 0 or 1, and it has exactly one nonzero entry (equal to 1) in each row and each column.

For the example in Sec. 10.8, the array of swap indices was $(3; 1; 2)$.

1. For this case the permutation, say P, is a $3 \quad 3$ matrix given by

$$P = \begin{matrix} 0 & & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix} : \tag{10.9.1}$$

2. Applying the above permutation matrix to the right hand side column vector b yields

$$Pb = \begin{matrix} 0 & & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix} \begin{matrix} 1 \\ b_1 \\ b_2 \\ b_3 \end{matrix} = \begin{matrix} 0 \\ b_3 \\ b_1 \\ b_2 \end{matrix} 1 : \tag{10.9.2}$$

3. This is the swap order of the rows in the LU factorization, as we saw in eq. (10.8.6).

In general, if the array of swap indices is $(s_1; : : : ; s_n)$ for n variables, then the nonzero entries of the permutation matrix P are given by the rule

$$P_{i\, s_i} = 1 \qquad (i = 1; : : : ; n) : \tag{10.9.3}$$

Using the Kronecker delta, the formula is

$$P_{ij} = {}_{s_i;j} \qquad (i; j = 1; : : : ; n) : \tag{10.9.4}$$

Hence for the example in Sec. 10.8, the nonzero entries are $P_{13} = 1$, $P_{21} = 1$ and $P_{32} = 1$.

The inverse of a permutation matrix P is its transpose, i.e. $P^{\ 1} = P^T$.

$$PP^T = P^T P = I_{n \ n} : \tag{10.9.5}$$

Hence a permutation matrix is an **orthogonal matrix.**

10.10 PA = LU algorithm

In terms of a permutation matrix P, the relation between the original matrix A and the LU factorization is

$$PA = LU: \tag{10.10.1}$$

Many people call this the **PA = LU algorithm.**

All square matrices can be factorized in this way.

The factorization procedure is computationally stable.

Hence PA = LU is a popular choice to solve systems of linear equations.

The original matrix equation is Ax = b. Multiplying both sides by P yields

$$PAx = Pb: \tag{10.10.2}$$

Since PA = LU, the equations to solve using the LU decomposition and backsubstitution are

$$LUx = Pb: \tag{10.10.3}$$

We first solve the following equation for a temporary column vector y using backsubstitution

$$Ly = Pb: \tag{10.10.4}$$

We then solve the following equation for x also using backsubstitution

$$Ux = y: \tag{10.10.5}$$

10.11 Full pivoting and PAQ

Partial pivoting swaps only rows, not columns.

The columns of the original matrix A are not permuted.

Full pivoting swaps both the rows and columns of A.

The relation between the original matrix A and the LU factorization is

$$PAQ = LU : \tag{10.11.1}$$

Both P and Q are permutation matrices: P describes the permutation of the rows and Q describes the permutation of the columns.

Theoretical analyses say that partial pivoting is almost as good as full pivoting, and it is simpler to implement in practice.

We shall not discuss full pivoting.

10.12 Computational complexity

For a square matrix A of size n n, the computational complexity of the LU decomposition is $O(n^3)$.

Some experts give the total number of computations as $\frac{1}{3}n^3$.

The multiplication of two matrices of size n k and k m requires nkm computations. If all the matrices are square, so that n = k = m, the computational cost is n^3.

Hence LU decomposition has about the same computational complexity as the multiplication of two square matrices of size n n.

Of course, there are also pivots to be calculated and swaps to be performed in the LU factorization algorithm. But the overall computational complexity is $O(n^3)$.

10.13 Matrix inversion

LU factorization can be used to solve for multiple right hand sides simultaneously.

The procedure is as follows.

1. Suppose we have k column vectors b_j, $j = 1;\ldots;k$.
2. We set up k solution column vectors x_j, $j = 1;\ldots;k$.
3. We form them into n k matrices, call then B and X.
4. Then we solve the matrix equation

$$AX = B: \tag{10.13.1}$$

5. Multiply both sides of eq. (10.13.1) by the permutation matrix P to obtain

$$PAX = PB: \tag{10.13.2}$$

6. Then note that PA = LU so eq. (10.13.2) becomes

$$LUX = PB: \tag{10.13.3}$$

7. We solve eq. (10.13.3) for X. We perform the LU factorization only once.

Now suppose that $B = I_{n\ n}$, the n n unit matrix. Then eq. (10.13.1) yields

$$AX = I: \tag{10.13.4}$$

Substituting B = I in eq. (10.13.3) yields

$$LUX = P: \tag{10.13.5}$$

The solution for X is the inverse A^{-1}.

In many practical applications, it is better to employ the matrices L and U directly. We do not need the inverse matrix A^{-1}.

10.14 Determinant

The official formula/definition for the determinant of an $n \times n$ square matrix requires a sum of products of matrix elements. Each term in the sum is a product of elements with exactly one element from each row and each column.

Basically, we sum over all permutations of $(1,\ldots,n)$. There are totally $n!$ permutations. Hence to calculate the determinant by summing over all permutations requires $n!$ computations. For even moderate values of n, such as $n = 10$, this is a large number of computations.

We can calculate the determinant of of a square matrix A using LU factorization.

The calculation requires $O(n^3)$ computations.

First we note that $PA = LU$. Then

$$\det(PA) = \det(P)\det(A); \tag{10.14.1a}$$

$$\det(LU) = \det(L)\det(U): \tag{10.14.1b}$$

1. Next we note that the determinant of a lower or upper triangular matrix equals the product of the elements along its main diagonal.
2. This is because any other set of products of elements contains entries from both above and below the main diagonal, hence it will multiply by zero.

In our situation, all the diagonal elements of the matrix L equal 1. Therefore $\det(L) = 1$. Also the determinant of U is given by

$$\det(U) = \prod_{i=1}^{n} u_{ii}: \tag{10.14.2}$$

The determinant of a permutation always has the value ± 1. Then $\det(P) = 1$ if P contains an even permutation of $(1,\ldots,n)$ and $\det(P) = -1$ if P contains an odd permutation of $(1,\ldots,n)$. The formula is

$$\det(P) = \begin{cases} +1 & \textbf{the number of swaps is even} \\ -1 & \textbf{the number of swaps is odd}: \end{cases} \tag{10.14.3}$$

Hence we must keep a count of the number of swaps in the LU decomposition.

The value of the determinant of A is therefore

$$\det(A) = \pm \det(U) \quad (\textbf{for even/odd number of swaps}): \tag{10.14.4}$$

Note that A is singular if and only of U is also singular. If we detect $\det(U) = 0$ (basically, if we we find any diagonal element of U equals zero), then we know the matrix A is singular.

For the example of the matrix in eq. (10.7.10), there were two swaps so $\det(P) = 1$. Then

$$\det(A) = \det(U) = u_{11}u_{22}u_{33} = 4 \quad \frac{7}{4} \quad \frac{24}{7} = 24: \tag{10.14.5}$$

10.15 C++ code

Examples of working C++ functions to implement the PA = LU algorithm are given below.

Note that the indexing of all the arrays follows the C/C++ convention.

Hence the indices run from 0 through n 1 for an array of length n.

10.15.1 C++ code: LU decomposition

```
int LU_decomposition(const int n,
                     std::vector<std::vector<double>> & a,
                     std::vector<int> & swap_indices,
                     int & swap_count)
{
  const double tol = 1.0e-14;  // tolerance for pivot

  swap_indices.clear();
  swap_count = 0;

  if (n < 1) return 1; // fail
  if (a.size() < n) return 1; // fail

  int i=0;
  int j=0;

  swap_indices.reserve(n);
  for (i = 0; i < n; ++i) {
    swap_indices.push_back(i); // initialize to (0,...n,-1)
  }

  for (i = 0; i < n; ++i) {
    // step 1: calculate scaled coeffs, compare for max pivot
    int pivot_row = i;
    double max_pivot = 0;
    for (j=i; j < n; ++j) {
      int k = i;
      double a_hat = std::abs(a[j][k]);
      if (a_hat > 0.0) {
        for (k=i+1; k < n; ++k) {
          double tmp = std::abs(a[j][k]);
          if (a_hat < tmp) a_hat = tmp;
        }
        double a_scaled = std::abs(a[j][i])/a_hat;  // note: a_ji not a_ij
        if (max_pivot < a_scaled) {
          max_pivot = a_scaled;
          pivot_row = j;
        }
      }
    }
```

```
    // step 2: if max pivot <= tol return fail (inconsistent or not linearly independent)
    if (std::abs(a[pivot_row][i]) <= tol) {
      swap_indices.clear();
      swap_count = 0;
      return 1; // fail
    }

    // step 3: found the max pivot, swap rows if row != i
    if (pivot_row != i) {
      // swap the whole row, including lower triangular entries
      // keep track of number of swaps
      // update the swap index
      ++swap_count;
      int si = swap_indices[i];
      swap_indices[i] = swap_indices[pivot_row];
      swap_indices[pivot_row] = si;

      for (j=0; j < n; ++j) {
        double tmp = a[i][j];
        a[i][j] = a[pivot_row][j];
        a[pivot_row][j] = tmp;
      }
    }
    double inv_pivot = 1.0/a[i][i];    // this is nonzero

    // step 4: eliminate column i from rows i+1 <= j < n, overwrite make LU matrix
    for (j=i+1; j < n; ++j) {
      double multiplier = a[j][i]*inv_pivot;
      a[j][i] = multiplier;     // store multiplier in lower triangular matrix

      for (int k=i+1; k < n; ++k) {
        a[j][k] -= multiplier*a[i][k];  // calculate upper triangular matrix
      }
    }
  }
  return 0;
}
```

10.15.2 C++ code: LU solver

```
int LU_solve(const int n,
             const std::vector<std::vector<double>> & LU,
             const std::vector<int> & swap_indices,
             const int swap_count,
             const std::vector<double> & rhs,
             std::vector<double> & x)
{
  x.clear();
  if (n < 1) return 1;   // fail

  std::vector<double> y(n, 0.0); // temporary storage

  // forward substitution Ly = rhs
  int i=0;
  for (i = 0; i < n; ++i) {
    int si = swap_indices[i];
    double sum = rhs[si];
    for (int j = 0; j < i; ++j) {
      sum -= LU[i][j] * y[j];
    }
    y[i] = sum;
  }

  // backward substitution Ux = y
  for (i = n-1; i >= 0; --i) {
    double sum = y[i];
    for (int j = i+1; j < n; ++j) {
      sum -= LU[i][j] * x[j];
    }
    x[i] = sum / LU[i][i];
  }

  return 0;
}
```

10.15.3 C++ code: LU inverse

```
int LU_inverse(const int n,
               const std::vector<std::vector<double>> & LU,
               const std::vector<int> & swap_indices,
               const int swap_count,
               std::vector<std::vector<double>> & inv_matrix)
{
  if (n < 1) return 1;   // fail

  for (int j = 0; j < n; ++j) {
    std::vector<double> rhs(n, 0.0);
    std::vector<double> x(n, 0.0);
    rhs[j] = 1.0;

    inv_matrix[j].clear();

    int rc = LU_solve(n, LU, swap_indices, swap_count, rhs, x);
    if (rc) {
      for (j = 0; j < n; ++j) {
        inv_matrix[j].clear(); // fail, clear everything
      }
      return rc; // fail
    }

    for (int i = 0; i < n; ++i) {
      inv_matrix[i][j] = x[i];
    }
  }

  return 0;
}
```

10.15.4 C++ code: LU determinant

```
int LU_determinant(const int n,
                   const std::vector<std::vector<double>> & LU,
                   const std::vector<int> & swap_indices,
                   const int swap_count,
                   double & det)
{
  det = 0;
  if (n < 1) return 1;   // fail

  det = ((swap_count % 2) == 0) ? 1 : -1;
  for (int i = 0; i < n; ++i) {
    det *= LU[i][i];
  }

  return 0;
}
```