CHAPTER

8 LU Factorization and Banded Matrices

OUTLINE

Lu Lu Lu, I've got some apples. Lu Lu Lu you've got some too.

–"Butters Stotch" in the television series **South Park**

CHAPTER POINTS

- In the case where one wants to solve several systems with changing righthand sides, LU factorization is an efficient solution method.
- Banded matrices have non-zero entries only on or near the diagonal, and the banded structure is preserved by LU factorization.
- With our methods we are able to solve realistic neutron diffusion problems.

In practice it is often the case that we want to solve the same linear system multiple times, just with different right-hand sides. One example in nuclear engineering involves a subcritical nuclear system with a source. We can write the neutron diffusion equation for this system as

$$-\nabla \cdot D(\mathbf{r})\nabla\phi(\mathbf{r}) + \Sigma_a\phi(\mathbf{r}) = \nu\Sigma_f\phi(\mathbf{r}) + Q(\mathbf{r}),$$

where the energy-integrated scalar flux is $\phi(\mathbf{r})$, the diffusion coefficient is $D(\mathbf{r})$, the absorption macroscopic cross-section is Σ_a, and the source rate density is $Q(\mathbf{r})$. As we will see later, this

DOI: 10.1016/B978-0-12-812253-2.00010-8

equation can, after numerical discretization, be written as

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{q},$$

where the entries in $\boldsymbol{\phi}$ and $\mathbf{q}$ represent a discrete value of the scalar flux and source rate density, respectively, and the matrix **A** is a discrete version of the diffusion operator.

From the previous chapter, we know that solving this system with Gaussian elimination is possible. Nevertheless, if you we going to analyze the system with several different source configurations (e.g., strength and position of the source) you would have to perform Gaussian elimination on each source configuration. If you have the intuition that the solution of one of these systems should make it possible to solve all of them, you are on the right track. To put it another way, besides the manipulations to the right-hand side of the equation, Gaussian elimination manipulates the matrix to get it in a form where the solution is reduced to back-substitution. We would like to do this work only once.

It is possible to do the work of Gaussian elimination to put the matrix in a form to make the solution with any right-hand side simple. By simple we mean that it involves one back substitution and one forward substitution, that is back substitution in the opposite direction (from top to bottom).

8.1 LU FACTORIZATION

LU factorization writes a matrix **A** as the product of a lower triangular matrix (that is a matrix with nonzero elements on the diagonal and below) times an upper triangular matrix (that is a matrix with nonzero elements on or above the diagonal). In other words,

$$\mathbf{A} = \mathbf{LU},$$

where

$$\mathbf{L} = \begin{pmatrix} * & 0 & \dots & & \\ * & * & 0 & \dots & \\ * & * & * & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots & \\ * & * & * & \dots & * \end{pmatrix},$$

and

$$\mathbf{U} = \begin{pmatrix} * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ 0 & 0 & \ddots & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & \dots & & 0 & * \end{pmatrix},$$

where the $*$ denote a potentially nonzero matrix element.

The question remains how we find the entries in the matrices in the LU factorization. To see this, consider a matrix we used in the last chapter:

$$\begin{pmatrix} 3 & 2 & 1 \\ -1 & 4 & 5 \\ 2 & -8 & 10 \end{pmatrix}.$$

For this matrix, after Gaussian elimination, the matrix became

$$\begin{pmatrix} 3 & 2 & 1 \\ 0 & 4\frac{2}{3} & 5\frac{1}{3} \\ 0 & 0 & 20 \end{pmatrix}.$$

Notice that this matrix is upper triangular.

We want to find the matrices **L** and **U** such that

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = \begin{pmatrix} 3 & 2 & 1 \\ -1 & 4 & 5 \\ 2 & -8 & 10 \end{pmatrix}.$$

We have to make a choice at this point. The choice we have is which diagonal, either **L** or **U**, we want to be all ones. We will choose to have all the $l_{ii} = 1$, that is the **L** matrix has a diagonal of all 1's (this choice makes our algorithm equivalent to an algorithm called Doolittle's method). Performing the matrix multiplication we get 9 equations and 9 unknowns for the factorization. The product of the first column of **U** by the first column of **L** gives

$$\begin{aligned} u_{11} &= 3 \\ l_{21}u_{11} &= -1 \\ l_{31}u_{11} &= 2. \end{aligned}$$

Which simplifies to,

$$\begin{aligned} u_{11} &= 3, \\ l_{21} &= -\frac{1}{3}, \\ l_{31} &= \frac{2}{3}. \end{aligned}$$

Notice that the solutions below the diagonal, the l_{21} and l_{31}, are the factors we used in our Gaussian elimination algorithm to remove the first column in the second and third equations (see Section 7.1). Also, the u_{11} value is that found in the same position in our Gaussian-eliminated matrix.

Continuing on, the equations from the second column are

$$\begin{aligned} u_{12} &= 2, \\ u_{12}l_{21} + u_{22} &= 4, \end{aligned}$$

$$u_{12}l_{31} + u_{22}l_{32} = -8,$$

the solutions to these equations are

$$\begin{aligned} u_{12} &= 2, \\ u_{22} &= 4\frac{2}{3}, \\ l_{32} &= -2. \end{aligned}$$

Looking back to Section 7.1 of the previous chapter, we see that the value of l_{32} is the value of the factor used to eliminate the second column below the diagonal in the Gaussian elimination example from before. Finally, the last three equations are

$$\begin{aligned} u_{13} &= 1, \\ u_{13}l_{21} + u_{23} &= 5, \\ u_{13}l_{31} + u_{23}l_{32} + u_{33} &= 10. \end{aligned}$$

The solutions to these equations are

$$\begin{aligned} u_{13} &= 1, \\ u_{23} &= 5\frac{1}{3}, \\ u_{33} &= 20. \end{aligned}$$

These are precisely the values of the third column in our Gaussian elimination example.

In summary, our factorization looks like:

$$\begin{pmatrix} 1 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 \\ \frac{2}{3} & -2 & 1 \end{pmatrix} \begin{pmatrix} 3 & 2 & 1 \\ 0 & 4\frac{2}{3} & 5\frac{1}{3} \\ 0 & 0 & 20 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 1 \\ -1 & 4 & 5 \\ 2 & -8 & 10 \end{pmatrix}.$$

Some observations are in order:

1. The upper triangular matrix is the same as the matrix we received after doing Gaussian elimination.
2. The non-zero and non-diagonal elements of the lower triangular matrix are the factors we used to arrive at our Gaussian matrix.

This suggests that we can reformulate our Gaussian elimination example to give us an LU factorization. What we will have to do is

1. Store the factors used to eliminate matrix elements in the appropriate place to get the **L** matrix, and
2. Perform Gaussian elimination as usual.

The code below modifies our Gauss elimination function to do just this. Also, we apply the function to our example matrix.

```
In [1]: import numpy as np
        def LU_factor(A):
            """Factor in place A in L*U=A. The lower triangular parts of A
            are the L matrix.  The L has implied ones on the diagonal.
            Args:
                A: N by N array
            Side Effects:
                A is factored in place.
            """
            [Nrow, Ncol] = A.shape
            assert Nrow == Ncol
            N = Nrow
            for column in range(0,N):
                for row in range(column+1,N):
                    mod_row = A[row]
                    factor = mod_row[column]/A[column,column]
                    mod_row = mod_row - factor*A[column,:]
                    #put the factor in the correct place in the modified row
                    mod_row[column] = factor
                    #only take the part of the modified row we need
                    mod_row = mod_row[column:N]
                    A[row,column:N] = mod_row
            return

        #let's try it on a 3 x 3 to start
        A = np.array([(3.0,2,1),(-1,4,5),(2,-8,10)])
        print("The original matrix is\n",A)
        LU_factor(A)
        print("The LU factored A is\n",A)
```

```
The original matrix is
 [[  3.   2.   1.]
 [ -1.   4.   5.]
 [  2.  -8.  10.]]
The LU factored A is
 [[  3.           2.           1.        ]
 [ -0.33333333   4.66666667   5.33333333]
 [  0.66666667  -2.          20.        ]]
```

This algorithm gives us the same thing our onerous, by-hand procedure did. Notice that we did not return an **L** and a **U** matrix, rather we factored in place the original matrix **A** by overwriting it. This is a reasonable thing to do because **A** could be a very large system and we do not want to duplicate that memory unnecessarily.

8.1.1 Forward and Backward Substitution

Now that we have discussed how to do LU factorization, the question we have not answered is how we can take an LU factored matrix and get the solution to the system $\mathbf{Ax} = \mathbf{b}$. To do this, we note that we can easily solve systems that only involve a lower (or upper) triangular matrix using back (or forward) substitution. For example, both

$$\mathbf{Ly} = \mathbf{b},$$

and

$$\mathbf{Ux} = \mathbf{y},$$

are easy to solve. In fact that is exactly what we want to do, notice that if we solve

$$\mathbf{Ly} = \mathbf{b},$$

that implies we know

$$\mathbf{y} = \mathbf{L}^{-1}\mathbf{b}.$$

Therefore, if we take the original system

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{b},$$

and operate by $\mathbf{L}^{-1}$, we get

$$\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{b}.$$

Therefore, given an LU factorization of a matrix **A** we want to solve

$$\mathbf{Ly} = \mathbf{b},$$

then

$$\mathbf{Ux} = \mathbf{y}.$$

The following code does just this.

```
In [2]: def LU_solve(A,b):
            """Take a LU factorized matrix and solve it for RHS b
            Args:
                A: N by N array that has been LU factored with
                assumed 1's on the diagonal of the L matrix
                b: N by 1 array of righthand side
            Returns:
                x: N by 1 array of solutions
            """
            [Nrow, Ncol] = A.shape
            assert Nrow == Ncol
            N = Nrow
            x = np.zeros(N)
            #temporary vector for L^-1 b
            y = np.zeros(N)
            #forward solve
            for row in range(N):
                RHS = b[row]
                for column in range(0,row):
                    RHS -= y[column]*A[row,column]
                y[row] = RHS
            #back solve
            for row in range(N-1,-1,-1):
                RHS = y[row]
            for column in range(row+1,N):
                RHS -= x[column]*A[row,column]
            x[row] = RHS/A[row,row]
        return x

In [3]: #let's try it on a 3 x 3 to start
        A = np.array([(3.0,2,1),(-1,4,5),(2,-8,10)])
        LU_factor(A)
        b = np.array([6,8,4])
        x = LU_solve(A,b)
        print("The solution to the system is",x)

The solution to the system is [ 1.  1.  1.]
```

This code gives the correct answer, a fact that is easily checked as you see that the sum of each row of this matrix equals the corresponding row on the right-hand side.

8.2 LU WITH PIVOTING, AND OTHER SUPPOSEDLY FUN THINGS

One thing we have not discussed is pivoting. Pivoting with LU is needed in the same cases as it is in Gaussian elimination. Nevertheless, when we switch equations around, we need to make sure that we keep track of where we have switched rows, so we can do the same to the right-hand side when we do our forward and back solves. This complicates things considerably, but only because we have to add that bookkeeping to our algorithm.

First, we will need to be able to swap rows in the matrix:

```
In [4]: def swap_rows(A, a, b):
            """Rows two rows in a matrix, switch row a with row b
            args:
            A: matrix to perform row swaps on
            a: row index of matrix
            b: row index of matrix

            returns:
            nothing

            side effects:
            changes A to have rows a and b swapped
            """
            assert (a>=0) and (b>=0)
            N = A.shape[0] #number of rows
            assert (a<N) and (b<N) #less than because 0-based indexing
            temp = A[a,:].copy()
            A[a,:] = A[b,:].copy()
            A[b,:] = temp.copy()
```

Next, we will change our algorithm to handle pivoting and keep track of the pivots:

```
In [5]: def LU_factor(A,LOUD=True):
            """Factor in place A in L*U=A. The lower triangular parts of A
```

```
        are the L matrix.  The L has implied ones on the diagonal.

        Args:
            A: N by N array
        Returns:
            a vector holding the order of the rows,
            relative to the original order
        Side Effects:
            A is factored in place.
        """
        [Nrow, Ncol] = A.shape
        assert Nrow == Ncol
        N = Nrow
        #create scale factors
        s = np.zeros(N)
        count = 0
        row_order = np.arange(N)
        for row in A:
            s[count] = np.max(np.fabs(row))
            count += 1
        if LOUD:
            print("s =",s)
        if LOUD:
            print("Original Matrix is\n",A)
        for column in range(0,N):
            #swap rows if needed
            largest_pos = np.argmax(np.fabs(A[column:N,column]/s[column]))
                          + column
            if (largest_pos != column):
                if (LOUD):
                    print("Swapping row",column,"with row",largest_pos)
                    print("Pre swap\n",A)
                swap_rows(A,column,largest_pos)
                #keep track of changes to RHS
                tmp = row_order[column]
                row_order[column] = row_order[largest_pos]
                row_order[largest_pos] = tmp
                #re-order s
                tmp = s[column]
                s[column] = s[largest_pos]
                s[largest_pos] = tmp
                if (LOUD):
                    print("A =\n",A)
            for row in range(column+1,N):
                mod_row = A[row]
                factor = mod_row[column]/A[column,column]
                mod_row = mod_row - factor*A[column,:]
                #put the factor in the correct place in the modified row
                mod_row[column] = factor
                #only take the part of the modified row we need
                mod_row = mod_row[column:N]
                A[row,column:N] = mod_row
        return row_order
    #let's try it on a 4 x 4 to start
```

```
        A = np.array([(3.0,2,1,-2),(-1,4,5,4),(2,-8,10,3),(-2,-8,10,0.1)])
        answer = np.ones(4)
        b = np.dot(A,answer)
        print(b)
        row_order = LU_factor(A)
        print(A)
        print("The new row order is",row_order)

[  4.   12.    7.    0.1]
s = [  3.   5.  10.  10.]
Original Matrix is
 [[  3.    2.    1.   -2. ]
 [ -1.    4.    5.    4. ]
 [  2.   -8.   10.    3. ]
 [ -2.   -8.   10.    0.1]]
Swapping row 1 with row 2
Pre swap
 [[  3.           2.           1.          -2.        ]
 [ -0.33333333   4.66666667   5.33333333   3.33333333]
 [  0.66666667  -9.33333333   9.33333333   4.33333333]
 [ -0.66666667  -6.66666667  10.66666667  -1.23333333]]
A =
 [[  3.           2.           1.          -2.        ]
 [  0.66666667  -9.33333333   9.33333333   4.33333333]
 [ -0.33333333   4.66666667   5.33333333   3.33333333]
 [ -0.66666667  -6.66666667  10.66666667  -1.23333333]]
[[  3.           2.           1.          -2.        ]
 [  0.66666667  -9.33333333   9.33333333   4.33333333]
 [ -0.33333333  -0.5         10.           5.5       ]
 [ -0.66666667   0.71428571   0.4         -6.52857143]]
The new row order is [0 2 1 3]
```

We need to change the `LU_solve` function to take advantage of the swapped rows. In function below, we make the necessary modification, and test the solution

```
In [6]: def LU_solve(A,b,row_order):
            """Take a LU factorized matrix and solve it for RHS b

            Args:
                A: N by N array that has been LU factored with
                assumed 1's on the diagonal of the L matrix
                b: N by 1 array of righthand side
                row_order:  list giving the re-ordered equations
                from the LU factorization with pivoting
            Returns:
                x: N by 1 array of solutions
            """
            [Nrow, Ncol] = A.shape
            assert Nrow == Ncol
            assert b.size == Ncol
            assert row_order.max() == Ncol-1
            N = Nrow
```

```
        #reorder the equations
        tmp = b.copy()
        for row in range(N):
            b[row_order[row]] = tmp[row]

        x = np.zeros(N)
        #temporary vector for L^-1 b
        y = np.zeros(N)
        #forward solve
        for row in range(N):
            RHS = b[row]
            for column in range(0,row):
                RHS -= y[column]*A[row,column]
            y[row] = RHS
        #back solve
        for row in range(N-1,-1,-1):
            RHS = y[row]
            for column in range(row+1,N):
                RHS -= x[column]*A[row,column]
            x[row] = RHS/A[row,row]
        return x


    print(A,row_order)
    x = LU_solve(A,b,row_order)
    print("The solution to the system is",x)
```

```
[[  3.          2.          1.          -2.        ]
 [  0.66666667  -9.33333333   9.33333333   4.33333333]
 [ -0.33333333  -0.5        10.           5.5       ]
 [ -0.66666667   0.71428571   0.4         -6.52857143]] [0 2 1 3]
The solution to the system is [ 1.  1.  1.  1.]
```

We now have a fully functional LU factorization algorithm and solving capability. We have not presented an exhaustive testing of the algorithm here, but this should be done before you use it in your own coding.

8.3 BANDED AND SYMMETRIC MATRICES

In many engineering calculations, symmetric and banded matrices arise from the equations. These matrices are often the result of a discretization of a diffusion equation. This is an important topic because, in truth, until about 20 years ago, large matrices that were not banded and symmetric were too difficult for all but the most specialized codes to handle.

For a matrix to be symmetric is means that if I transpose the matrix, the matrix does not change, i.e. if,

$$A_{ij} = A_{ji}$$

then the matrix is symmetric. This is equivalent to saying $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$. One feature of a symmetric matrix is that it is possible to store only the lower or upper triangular part of the matrix because of the symmetry.

Banded matrices take their name from the structure of the matrix. When the non-zeros of the matrix are shown they form bands around the diagonal. A diagonal matrix is a banded matrix of bandwith 1:

$$\begin{pmatrix} * & 0 & \dots & & \\ 0 & * & 0 & \dots & \\ 0 & 0 & * & 0 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ 0 & 0 & \dots & \dots & * \end{pmatrix}$$

The bandwidth is 1 because there is only 1 diagonal or off diagonal that has non-zero elements.

A common banded matrix is the tri-diagonal matrix. This matrix has a bandwidth of 3 and looks like

$$\begin{pmatrix} * & * & 0 & \dots & & & \\ * & * & * & 0 & \dots & & \\ 0 & * & * & * & 0 & \dots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & & \\ \vdots & & 0 & * & * & * & 0 \\ 0 & \dots & & 0 & * & * & * \\ 0 & 0 & \dots & & 0 & * & * \end{pmatrix}$$

BOX 8.1 NUMERICAL PRINCIPLE

A matrix that does not change when the transpose is taken, i.e., $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$, is called a symmetric matrix.

The main diagonal of a matrix is defined as the all the entries in a matrix where the row number equals the column number.

A banded matrix is a matrix whose non-zero entries are confined to the a subset of diagonals around the main diagonal of a matrix. The width of the band around the main diagonal that contains the non-zero elements is called the bandwidth. A diagonal matrix is a banded matrix with bandwidth of 1. A tridiagonal matrix has nonzero elements directly above and below the main diagonal.

To solve a tridiagonal system we can make some shortcuts. The primary shortcut is that we do not need to create a matrix at all. We only need to store three vectors, one for the diagonal and two for the off diagonals. We could redefine our algorithms to respect this sparsity of the matrix, i.e., take advantage of all the 0's. We will not do this here because the algorithm gets much more complicated. If we were writing a code to solve very large tridiagonal systems, we would want to take advantage of this efficiency because the amount of memory needed to store three vectors grows linearly with the number of equations, but grows quadratically if we store the whole matrix and all those wasteful zeros.

Pentadiagonal systems also arise in practice, namely 2-D diffusion equations will have 5 non-zero terms in each row. Furthermore, a 3-D diffusion equation will lead to a heptadiagonal matrix or a matrix that has 7 non-zero elements in each row. These matrices typically have a bandwidth much larger than 5 or 7 because the ordering of unknowns often makes it so that there are zeros between the non-zero elements in a row.

FURTHER READING

LU factorization is a common technique for solving linear systems of equations. It is also the basis for the SuperLU package [11], which has a Python interface through the `scipy` package.

PROBLEMS

Short Exercises

8.1. Compute the LU factorization of the matrix

$$\mathbf{A} = \begin{pmatrix} -1 & 2 & 3 \\ 4 & -5 & 6 \\ 7 & 8 & -9 \end{pmatrix}.$$

8.2. Consider the n by n matrix defined by

$$\mathbf{A} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \dots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \dots & \frac{1}{n+1} \\ \vdots & & & & \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \dots & \frac{1}{2n-1} \end{pmatrix},$$

and the vector

$$\mathbf{b} = (1, -1, 1, -1 \dots)^{\mathrm{T}}.$$

Write a function to generate $\mathbf{A}$ and $\mathbf{b}$ as a function of n. Use it to solve the system $\mathbf{Ax} = \mathbf{b}$ for $n = 2, 4, 8, 16$ via LU factorization. What do you notice about your answers and the resulting factorization?

Programming Projects

1. Matrix Inverse via LU Factorization

In the previous chapter we presented an approach to compute the inverse of a matrix. Here is another way to compute the inverse; this time using LU factorization. Compute the LU factorization of **A** and then use this to solve

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i \qquad i = 1, \dots, n,$$

where $\mathbf{b}_i$ is a vector of zeros except at position i where it has 1, and $\mathbf{x}_i$ is the ith column of the inverse. Test your implementation on the matrix from Short Exercise problem 8.1 above. Check that the inverse of **A** times the original matrix gives the identity matrix.

2. Shielding a Radioactive Source

Consider the problem of a slab geometry radiation source surrounded by a shield. You are interested in computing the leakage rate of radiation outside the shield. If the shield is a pure absorber and the source is collimated into a beam, the net neutron current density J [neutrons/cm^2·s] of neutrons can be described by the equation

$$\frac{dJ}{dx} + \Sigma_\mathrm{a} J(x) = Q(x), \qquad J(0) = 0,$$

where x is the position in the slab which extends from $x = 0$ to $x = 10$ cm, $Q(x)$ [neutrons/cm^3·s] is the spatially dependent source strength, and Σ_a [cm^{-1}] is the macroscopic absorption cross-section. The leakage rate density out of the shield is $J(10)$. A simple discretization of this equation leads to the system of equations

$$\frac{J_i - J_{i-1}}{\Delta x} + \frac{\Sigma_\mathrm{a}}{2}(J_i + J_{i-1}) = Q_i, \qquad i = 0 \dots I$$

In this equation $\Delta x = 10/I$, $J_i \approx J(i\Delta x)$, and $Q_i = Q(i\Delta x + \Delta x/2)$. Solve this system of equations and find the net leakage rate density with $I = 100$ and $\Sigma_\mathrm{a} = 1$ with several different source configurations:

- A thin source at the left of the slab

$$Q(x) = \begin{cases} 1 & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases},$$

- A thin source at the middle of the slab

$$Q(x) = \begin{cases} 1 & 4.5 \le x \le 5.5 \\ 0 & \text{otherwise} \end{cases},$$

- A thick source at the left of the slab

$$Q(x) = \begin{cases} 1 & 0 \le x \le 4 \\ 0 & \text{otherwise} \end{cases},$$

- A thin source at the middle of the slab

$$Q(x) = \begin{cases} 1 & 3 \le x \le 7 \\ 0 & \text{otherwise} \end{cases}.$$

Compare your solution to the analytic solution to problem. The analytic solution is an integral over an exponential function. To compute your numerical solution you will not need Gaussian elimination or LU factorization.

3. LU Factorization of a Tridiagonal System

Before we mentioned that it was possible to LU factorize a tridiagonal matrix. Modify the LU factorization without pivoting function, `LU_factor`, defined above to work with tridiagonal matrix. Your modified function should take as input three vectors, representing the main diagonal and two off diagonals. The function should return the three vectors that yield the LU factorization. Check your algorithm on one of the tridiagonal matrices used in this chapter. Also, use this function to see how large of a tridiagonal system you can solve on your computer.