Computational Physics (PHYS6350)

Lecture 4: Linear algebra and matrices

- Linear systems of equations
  - Gaussian elimination
  - LU decomposition

January 28, 2025

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics/tree/spring2025

Linear system of equations

⚠️ **Warning**: The following Python examples are intended for educating concepts rather than for practical use. For practical implementations, consider using standard libraries such as `numpy`, `scipy` or `linalg`.

Linear system of equations

Generic problem: solve a system of linear equations

$$\sum_{j=1}^{N} a_{1j} x_j = v_1,$$

$$\dots$$

$$\sum_{j=1}^{N} a_{Nj} x_j = v_N.$$

Matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{v}.$$

$$\mathbf{A} = \begin{pmatrix} a_{11} & \dots & a_{1N} \\ \dots & \dots & \dots \\ a_{N1} & \dots & a_{NN} \end{pmatrix}$$

A unique solution exists if all equations are *linearly independent and consistent*

Equivalent to condition $\det A \neq 0$

Cramer's rule

Cramer's rule:

Solution to $A\mathbf{x} = \mathbf{b}$ reads

$$x_i = \frac{\det(A_i)}{\det(A)} \qquad i = 1, \ldots, n$$

where A_i is the matrix formed by replacing the i-th column of A by the column vector $\mathbf{b}$.

For example

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}.$$

solved as

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \quad y = \frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, \text{ and } z = \frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}.$$

Cramer's rule has theoretical importance but little practical use

Gaussian elimination

Gaussian elimination is the base procedure for solving systems of linear equations

It is based on iterative transformation of the system of linear equations which preserve the solution (the solution stays the same)

The goal is to eliminate all entries in matrix ***A*** below the main diagonal

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & \tilde{a}_{12} & \tilde{a}_{13} & \tilde{a}_{14} \\ 0 & 1 & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & 0 & 1 & \tilde{a}_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{pmatrix}$$

The following two operations preserve the solution:

1. One can multiply any row by a non-zero number
2. One can subtract from a given row any other row (with any non-zero factor)

Gaussian elimination

1. Starting from the first row, divide the row by its diagonal element a_{11}

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & a_{14}/a_{11} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v_1/a_{11} \\ v_2 \\ v_3 \\ v_4 \end{pmatrix}$$

2. Make all entries in column 1 below the main diagonal equal to zero by subtracting the first equation from row *j* with a factor a_{j1}. We get

$$\begin{pmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} & a_{14}/a_{11} \\ 0 & a_{22}-a_{21}a_{12}/a_{11} & a_{23}-a_{21}a_{13}/a_{11} & a_{24}-a_{21}a_{14}/a_{11} \\ 0 & a_{32}-a_{31}a_{12}/a_{11} & a_{33}-a_{31}a_{13}/a_{11} & a_{34}-a_{31}a_{14}/a_{11} \\ 0 & a_{42}-a_{41}a_{12}/a_{11} & a_{43}-a_{41}a_{13}/a_{11} & a_{44}-a_{41}a_{14}/a_{11} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v_1/a_{11} \\ v_2-a_{21}v_1/a_{11} \\ v_3-a_{31}v_1/a_{11} \\ v_4-a_{41}v_1/a_{11} \end{pmatrix}$$

or

$$\begin{pmatrix} 1 & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v'_1 \\ v'_2 \\ v'_3 \\ v'_4 \end{pmatrix}$$

3. Repeat steps 1-2 for the 2nd row and column, and then for all others

$$\begin{pmatrix} 1 & a''_{12} & a''_{13} & a''_{14} \\ 0 & 1 & a''_{23} & a''_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} v''_1 \\ v''_2 \\ v''_3 \\ v''_4 \end{pmatrix}$$

Final result:

$$\begin{pmatrix} 1 & \tilde{a}_{12} & \tilde{a}_{13} & \tilde{a}_{14} \\ 0 & 1 & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & 0 & 1 & \tilde{a}_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{pmatrix}$$

Gaussian elimination

Example: Work out Gaussian elimination for system of equations

$$\begin{pmatrix} 2 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix}$$

Gaussian elimination

Example: Work out Gaussian elimination for system of equations

$$\begin{pmatrix} 2 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix}$$

The result should be

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -2 \\ 3.6 \\ -2 \\ 1 \end{pmatrix}$$

Backsubstitution

After the Gaussian elimination we have the following system of equations

$$\begin{pmatrix} 1 & \tilde{a}_{12} & \tilde{a}_{13} & \tilde{a}_{14} \\ 0 & 1 & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & 0 & 1 & \tilde{a}_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{pmatrix}$$

or

$$\begin{aligned} x_1 + \tilde{a}_{12}x_2 + \tilde{a}_{13}x_3 + \tilde{a}_{14}x_4 &= \tilde{v}_1, \\ x_2 + \tilde{a}_{23}x_3 + \tilde{a}_{24}x_4 &= \tilde{v}_2, \\ x_3 + \tilde{a}_{34}x_4 &= \tilde{v}_3, \\ x_4 &= \tilde{v}_4. \end{aligned}$$

The solution now proceeds through *backsubstitution*, i.e. we go from x_4 to x_1

$$x_4 = \tilde{v}_4$$

$$x_3 = \tilde{v}_3 - \tilde{a}_{34}x_4$$

$$x_2 = \tilde{v}_4 - \tilde{a}_{23}x_3 - \tilde{a}_{24}x_4$$

$$x_1 = \tilde{v}_4 - \tilde{a}_{12}x_2 - \tilde{a}_{13}x_3 - \tilde{a}_{14}x_4.$$

Gaussian elimination: Implementation

```
import numpy as np

def linsolve_gaussian(A0, v0):
    # Initialization
    A = A0.copy()
    v = v0.copy()
    N = len(v)

    # Gaussian elimination
    for r in range(N):
        # Divide row r by diagonal element
        div = A[r,r]
        if (div == 0.):
            print("Diagonal element is zero! Cannot solve the system with simple Gaussian elimination")
            return None
        A[r,:] /= div
        v[r]   /= div

        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = A[r2,r]
            A[r2,:] -= mult * A[r,:]
            v[r2] -= mult * v[r]

    # Backsubstitution
    x = np.empty(N,float)
    for r in range(N-1,-1,-1):
        x[r] = v[r]
        for c in range(r+1,N):
            x[r] -= A[r][c] * x[c]

    return x
```

Gaussian elimination: Implementation

```python
import numpy as np

def linsolve_gaussian(A0, v0):
    # Initialization
    A = A0.copy()
    v = v0.copy()
    N = len(v)

    # Gaussian elimination
    for r in range(N):
        # Divide row r by diagonal element
        div = A[r,r]
        if (div == 0.):
            print("Diagonal element is zero! Cannot solve the system with simple Gaussian elimination")
            return None
        A[r,:] /= div
        v[r]   /= div

        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = A[r2,r]
            A[r2,:] -= mult * A[r,:]
            v[r2] -= mult * v[r]

    # Backsubstitution
    x = np.empty(N,float)
    for r in range(N-1,-1,-1):
        x[r] = v[r]
        for c in range(r+1,N):
            x[r] -= A[r][c] * x[c]

    return x
```

```python
A = np.array([[ 2,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)
v = np.array([ -4, 3, 9, 7 ],float)

x = linsolve_gaussian(A,v)
print('x  =',x)
print('Ax =', A.dot(x))
print('v  =', v)
```

```
x  = [ 2. -1. -2.  1.]
Ax = [-4.  3.  9.  7.]
v  = [-4.  3.  9.  7.]
```

Gaussian elimination: Zero diagonal elements

The trivial implementation of Gaussian elimination will fail if any of the diagonal elements becomes equal to zero in the process of solving

For example:

$$\begin{pmatrix} 0 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix}$$

```
A = np.array([[ 0,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)
v = np.array([ -4, 3, 9, 7 ],float)
x = linsolve_gaussian(A,v)
print('x  =',x)
print('Ax =', A.dot(x))
print('v  =', v)
```

```
Diagonal element is zero! Cannot solve the system with simple Gaussian elimination
```

Gaussian elimination: Pivoting

We can simply exchange rows 1 & 2 and avoid the vanishing diagonal element

$$\begin{pmatrix} 0 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 3 & 4 & -1 & -1 \\ 0 & 1 & 4 & 1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 3 \\ -4 \\ 9 \\ 7 \end{pmatrix}$$

This is called ***pivoting***, which does not change the solution

The optimal choice of a pivot is the **largest element in magnitude** (minimizes the round-off error by avoiding division by small numbers). For numerical stability, pivoting should be performed even when there are no vanishing diagonal elements

Partial pivoting: Exchange rows only

Full pivoting: Exchange both rows and columns (changes the ordering of elements in x)

Partial pivoting implementation

```
def linsolve_gaussian_partialpivot(A0, v0):
    # Initialization
    A = A0.copy()
    v = v0.copy()
    N = len(v)

    # Gaussian elimination
    for r in range(N):
        # Find the pivot element (largest in magnitude)
        r_pivot = r
        for i in range(r + 1, N):
            if (abs(A[i][r]) > abs(A[r_pivot][r])):
                r_pivot = i

        # Swap the rows
        A[[r,r_pivot]] = A[[r_pivot,r]]
        v[[r,r_pivot]] = v[[r_pivot,r]]

        # Divide row r by the pivot element
        div = A[r,r]
        if (div == 0.):
            print("Diagonal element is zero! The system appears to be singular")
            return None
        A[r,:] /= div
        v[r]   /= div


        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = A[r2,r]
            A[r2,:] -= mult * A[r,:]
            v[r2] -= mult * v[r]


    # Backsubstitution
    x = np.empty(N,float)
    for r in range(N-1,-1,-1):
        x[r] = v[r]
        for c in range(r+1,N):
            x[r] -= A[r][c] * x[c]

    return x
```

Partial pivoting implementation

```python
def linsolve_gaussian_partialpivot(A0, v0):
    # Initialization
    A = A0.copy()
    v = v0.copy()
    N = len(v)

    # Gaussian elimination
    for r in range(N):
        # Find the pivot element (largest in magnitude)
        r_pivot = r
        for i in range(r + 1, N):
            if (abs(A[i][r]) > abs(A[r_pivot][r])):
                r_pivot = i

        # Swap the rows
        A[[r,r_pivot]] = A[[r_pivot,r]]
        v[[r,r_pivot]] = v[[r_pivot,r]]

        # Divide row r by the pivot element
        div = A[r,r]
        if (div == 0.):
            print("Diagonal element is zero! The system appears to be singular")
            return None
        A[r,:] /= div
        v[r]   /= div


        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = A[r2,r]
            A[r2,:] -= mult * A[r,:]
            v[r2] -= mult * v[r]


    # Backsubstitution
    x = np.empty(N,float)
    for r in range(N-1,-1,-1):
        x[r] = v[r]
        for c in range(r+1,N):
            x[r] -= A[r][c] * x[c]

    return x
```

```python
A = np.array([[ 0,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)
v = np.array([ -4, 3, 9, 7 ],float)
x = linsolve_gaussian_partialpivot(A,v)
print('x  =',x)
print('Ax =', A.dot(x))
print('v  =', v)
```

```
x  = [ 1.61904762 -0.42857143 -1.23809524  1.38095238]
Ax = [-4.  3.  9.  7.]
v  = [-4.  3.  9.  7.]
```

LU decomposition

At the end of Gaussian elimination we have

$$\begin{pmatrix} 1 & \tilde{a}_{12} & \tilde{a}_{13} & \tilde{a}_{14} \\ 0 & 1 & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & 0 & 1 & \tilde{a}_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{pmatrix}$$

i.e. our matrix became **upper triangular** $\mathbf{Ax} = \mathbf{v}, \quad \Rightarrow \quad \mathbf{Ux} = \tilde{\mathbf{v}},$

Discarding the pivoting for a moment, all steps of the Gaussian elimination can be represented by matrix multiplication, i.e.

$$\mathbf{U} = \mathbf{L_{N-1}} \dots \mathbf{L_0 A}$$

where e.g.

$$\mathbf{L}_0 = \frac{1}{a_{11}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & a_{11} & 0 & 0 \\ -a_{31} & 0 & a_{11} & 0 \\ -a_{41} & 0 & 0 & a_{11} \end{pmatrix}, \qquad \mathbf{L}_1 = \frac{1}{a'_{22}} \begin{pmatrix} a'_{22} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -a'_{32} & a'_{22} & 0 \\ 0 & -a'_{42} & 0 & a'_{22} \end{pmatrix},$$

LU decomposition

The matrices $\mathbf{L}_i$ are lower triangular

$$\mathbf{L}_0 = \frac{1}{a_{11}}\begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & a_{11} & 0 & 0 \\ -a_{31} & 0 & a_{11} & 0 \\ -a_{41} & 0 & 0 & a_{11} \end{pmatrix},$$

$$\mathbf{L}_1 = \frac{1}{a'_{22}}\begin{pmatrix} a'_{22} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -a'_{32} & a'_{22} & 0 \\ 0 & -a'_{42} & 0 & a'_{22} \end{pmatrix},$$

Inverses are also lower triangular

$$\mathbf{L}_0^{-1} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & 1 & 0 & 0 \\ a_{31} & 0 & 1 & 0 \\ a_{41} & 0 & 0 & 1 \end{pmatrix},$$

$$\mathbf{L}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a'_{22} & 0 & 0 \\ 0 & a'_{32} & 1 & 0 \\ 0 & a'_{42} & 0 & 1 \end{pmatrix},$$

Therefore,

$$\begin{aligned} \mathbf{A} &= \mathbf{L}_0^{-1} \ldots \mathbf{L}_{N-1}^{-1}\mathbf{U} \\ &= \mathbf{L}\mathbf{U}, \end{aligned}$$

where

$$\mathbf{L} = \mathbf{L}_0^{-1} \ldots \mathbf{L}_{N-1}^{-1} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a'_{22} & 0 & 0 \\ a_{31} & a'_{32} & a''_{33} & 0 \\ a_{41} & a'_{42} & a''_{43} & a'''_{44} \end{pmatrix}.$$

LU decomposition

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

```python
def lu_decomp(A):
    # Initialization
    U = A.copy()
    N = len(v)
    L = np.zeros((N,N), float)

    # Gaussian elimination
    for r in range(N):
        # Record the elements of L
        for r2 in range(r,N):
            L[r2][r] = U[r2][r]

        # Divide row r by diagonal element
        div = U[r,r]
        if (div == 0.):
            print("Diagonal element is zero! LU decomposition without pivoting is not possible!")
            return None
        U[r,:] /= div

        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = U[r2,r]
            U[r2,:] -= mult * U[r,:]

    return L, U
```

LU decomposition

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

```python
def lu_decomp(A):
    # Initialization
    U = A.copy()
    N = len(v)
    L = np.zeros((N,N), float)

    # Gaussian elimination
    for r in range(N):
        # Record the elements of L
        for r2 in range(r,N):
            L[r2][r] = U[r2][r]

        # Divide row r by diagonal element
        div = U[r,r]
        if (div == 0.):
            print("Diagonal element is zero! LU decomposition without pivoting is not possible!")
            return None
        U[r,:] /= div

        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = U[r2,r]
            U[r2,:] -= mult * U[r,:]

    return L, U
```

```python
A = np.array([[ 2,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)

L, U = lu_decomp(A)
print('L   =', L)
print('U   =', U)
print('LU  =', np.dot(L,U))
print('A   =', A)
```

```
L   = [[  2.    0.    0.    0. ]
 [  3.    2.5   0.    0. ]
 [  1.   -4.5 -13.6   0. ]
 [  2.   -3.  -11.4  -1. ]]
U   = [[ 1.   0.5  2.   0.5]
 [ 0.   1.  -2.8 -1. ]
 [-0.  -0.   1.  -0. ]
 [-0.  -0.  -0.   1. ]]
LU  = [[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
A   = [[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
```

LU decomposition and systems of linear equations

LU decomposition is particularly useful for repeated solution of systems of linear equations

$$\mathbf{Ax} = \mathbf{v}$$

the matrix **A** stays the same but where the vector **v** can change.

Indeed the system of equations becomes

$$\mathbf{LUx} = \mathbf{v}.$$

Let us define

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

then

$$\mathbf{Ly} = \mathbf{v}.$$

We can solve the system for $\boldsymbol{x}$ in two steps:

1. First we solve the equation $\boldsymbol{Ly} = \boldsymbol{v}$ using forward substitution, in analogy to backsubstitution we used before.
2. Once we have y, we can solve $\boldsymbol{Ux} = \boldsymbol{y}$ for $\boldsymbol{x}$ using backsubstitution,

LU decomposition and systems of linear equations

```
def solve_using_lu(L,U,v):
    # L*U*x = v
    # First solve L*y = v with forward substitution
    # Then solve U*x = y with backsubstitution
    # Initialization

    N = len(v)
    # Backsubstitution for L*y = v
    y = np.empty(N,float)
    for r in range(N):
        y[r] = v[r]
        for c in range(r):
            y[r] -= L[r][r - 1 - c] * y[r - 1 - c]
        y[r] /= L[r][r]

    # Backsubstitution for U*x = y
    x = np.empty(N,float)
    for r in range(N-1,-1,-1):
        x[r] = y[r]
        for c in range(r+1,N):
            x[r] -= U[r][c] * x[c]
    return x
```

```
A = np.array([[ 2,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)

L, U = lu_decomp(A)

v = np.array([ -4, 3, 9, 7 ],float)
x = solve_using_lu(L,U,v)
print('x  =', x)
print('Ax =', A.dot(x))
print('v  =', v)
```

LU decomposition with pivoting

Not every non-singular matrix allows for LU decomposition because its diagonal elements may end up being zero.

In the general case, we need to allow the possibility to perform partial pivoting by exchanging the rows of our matrix.

If we do that, what we get the LU-decomposition with pivoting, which can be written as

$$\mathbf{PA} = \mathbf{LU}.$$

Here **P** is a row permutation operator.

Solving the system of equations

$$\mathbf{Ax} = \mathbf{v},$$

is also straightforward using forward and backsubstitution passes, except that we have to exchange the rows in the vector **v** to account for the row swaps that we did.

LU decomposition with pivoting

```python
def lu_decomp_partialpivot(A):
    # Initialization
    U = A.copy()
    N = len(v)
    L = np.zeros((N,N), float)

    # Keep track of all row swaps
    row_map = [i for i in range(N)]

    # Gaussian elimination
    for r in range(N):
        # Find the pivot element (largest in magnitude)
        r_pivot = r
        for i in range(r + 1, N):
            if (abs(U[i][r]) > abs(U[r_pivot][r])):
                r_pivot = i

        row_map[r], row_map[r_pivot] = row_map[r_pivot], row_map[r]
        U[[r,r_pivot]] = U[[r_pivot,r]]
        L[[r,r_pivot]] = L[[r_pivot,r]]

        # Record the elements of L
        for r2 in range(r,N):
            L[r2][r] = U[r2][r]

        # Divide row r by the pivot element
        div = U[r,r]
        if (div == 0.):
            print("Diagonal element is zero! The system appears to be singular")
            return None
        U[r,:] /= div


        # Now subtract this row from the lower rows
        for r2 in range(r+1,N):
            mult = U[r2,r]
            U[r2,:] -= mult * U[r,:]

    return L, U, row_map
```

```python
A = np.array([[ 2,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)

L, U, row_map = lu_decomp_partialpivot(A)
print('L   =', L)
print('U   =', U)
print('LU  =', np.dot(L,U))
print('A   =', A)
```

```
L   = [[ 3.          0.          0.          0.        ]
 [ 1.         -5.33333333  0.          0.        ]
 [ 2.         -1.66666667  4.25        0.        ]
 [ 2.         -4.66666667  0.5        -1.        ]]
U   = [[ 1.          1.33333333 -0.33333333 -0.33333333]
 [-0.          1.         -0.25       -1.        ]
 [ 0.          0.          1.          0.        ]
 [-0.         -0.         -0.          1.        ]]
LU  = [[ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2.  1.  4.  1.]
 [ 2. -2.  1.  3.]]
A   = [[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
```

The matrices A and L*U coincide up to a permutation of rows, as they should.

LU decomposition with pivoting

```python
def solve_using_lu_partialpivot(L,U,row_map,v):
    # L*U*x = v
    # First solve L*y = v with forward substitution
    # Then solve U*x = y with backsubstitution
    # Initialization

    N = len(v)
    # Backsubstitution for L*y = v
    y = np.empty(N,float)
    for rr in range(N):
        r = row_map[rr]
        y[rr] = v[r]
        for c in range(rr):
            y[rr] -= L[rr][rr - 1 - c] * y[rr - 1 - c]
        y[rr] /= L[rr][rr]

    # Backsubstitution for U*x = y
    x = np.empty(N,float)
    for rr in range(N-1,-1,-1):
        x[rr] = y[rr]
        for c in range(rr+1,N):
            x[rr] -= U[rr][c] * x[c]
    return x
```

```python
A = np.array([[ 0,  1,  4,  1 ],
              [ 3,  4, -1, -1 ],
              [ 1, -4,  1,  5 ],
              [ 2, -2,  1,  3 ]],float)

L, U, row_map = lu_decomp_partialpivot(A)
v = np.array([ -4, 3, 9, 7 ],float)
x = solve_using_lu_partialpivot(L,U,row_map,v)
print('x  =', x)
print('Ax =', A.dot(x))
print('v  =', v)
```

```
x  = [ 1.61904762 -0.42857143 -1.23809524  1.38095238]
Ax = [-4.  3.  9.  7.]
v  = [-4.  3.  9.  7.]
```