Part 2: LU decomposition, Inverse matrix, and Gauss-Seidel method

Introduction to Numerical Problem Solving, Spring 2017

 Sakari Lukkarinen

Helsinki Metropolia University of Applied Sciences

LU decomposition (factorization)

Any square matrix A can be expressed as a product of a lower triangular matrix *L* and an upper triangular matrix *U*:

$$A = LU$$

The process of computing *L* and *U* is called LU decomposition or factorisation. After decomposing A, it is easy to solve the equation $Ax = b$. First the equation is rewritten as $LUx = b$ and then it is solved in parts.

Step 1: Decompose $A = LU$

Step 2: Solve for y in the equation $Ly = b$

Step 3: Solve for x in the equation $Ux = y$

Source: Jaan Kiusalaas (2013). Numerical Methods in Engineering with Python 3.

Doolittle’s decomposition

Doolittle’s decomposition is closely related to Gaussian elimination

$$L = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

It is usual practice to have the multipliers in same matrix

$$[L\backslash U] = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix}$$

Hand calculations – Example 2.5

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 6 & -1 \\ 2 & -1 & 2 \end{bmatrix}$$

$$\text{row 2} \leftarrow \text{row 2} - 1 \times \text{ row 1 (eliminates } A_{21})$$

$$\text{row 3} \leftarrow \text{row 3} - 2 \times \text{ row 1 (eliminates } A_{31})$$

$$\mathbf{A}' = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & -2 \\ 2 & -9 & 0 \end{bmatrix}$$

$$\text{row 3} \leftarrow \text{row 3} - (-4.5) \times \text{ row 2 (eliminates } A_{32})$$

$$\mathbf{A}'' = [\mathbf{L} \backslash \mathbf{U}] = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & -2 \\ 2 & -4.5 & -9 \end{bmatrix}$$

Continued

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -4.5 & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & -2 \\ 0 & 0 & -9 \end{bmatrix} \qquad \begin{bmatrix} \mathbf{U} \mid \mathbf{y} \end{bmatrix} = \left[\begin{array}{ccc|c} 1 & 4 & 1 & 7 \\ 0 & 2 & -2 & 6 \\ 0 & 0 & -9 & 18 \end{array}\right]$$

$$\begin{bmatrix} \mathbf{L} \mid \mathbf{b} \end{bmatrix} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 7 \\ 1 & 1 & 0 & 13 \\ 2 & -4.5 & 1 & 5 \end{array}\right]$$

$$y_1 = 7$$

$$y_2 = 13 - y_1 = 13 - 7 = 6$$

$$y_3 = 5 - 2y_1 + 4.5y_2 = 5 - 2(7) + 4.5(6) = 18$$

$$x_3 = \frac{18}{-9} = -2$$

$$x_2 = \frac{6 + 2x_3}{2} = \frac{6 + 2(-2)}{2} = 1$$

$$x_1 = 7 - 4x_2 - x_3 = 7 - 4(1) - (-2) = 5$$

LU Decomposition in Python and NumPy

By Michael Halls-Moore on January 21st, 2013

```
In [8]: from pprint import pprint
        from scipy.linalg import lu, solve

        A = array([ [7, 3, -1, 2],
                    [3, 8, 1, -4],
                    [-1, 1, 4, -1],
                    [2, -4, -1, 6] ])

        b = array([1, 2, 3, 4])
        P, L, U = lu(A)

        print("A:")
        pprint(A)

        print("P:")
        pprint(P)

        print("L:")
        pprint(L)

        print("U:")
        pprint(U)
```

```
A:
array([[ 7,  3, -1,  2],
       [ 3,  8,  1, -4],
       [-1,  1,  4, -1],
       [ 2, -4, -1,  6]])
P:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
L:
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.42857143,  1.        ,  0.        ,  0.        ],
       [-0.14285714,  0.21276596,  1.        ,  0.        ],
       [ 0.28571429, -0.72340426,  0.08982036,  1.        ]])
U:
array([[ 7.        ,  3.        , -1.        ,  2.        ],
       [ 0.        ,  6.71428571,  1.42857143, -4.85714286],
       [ 0.        ,  0.        ,  3.55319149,  0.31914894],
       [ 0.        ,  0.        ,  0.        ,  1.88622754]])
```

Solving using LU decomposition

```
In [11]: from scipy.linalg import lu, solve

         A = array([ [7, 3, -1, 2],
                     [3, 8, 1, -4],
                     [-1, 1, 4, -1],
                     [2, -4, -1, 6] ])

         b = array([1, 2, 3, 4])

         P, L, U = lu(A)
         y = solve(dot(P, L), b)
         x = solve(U, y)

         print("x:")
         pprint(x)

x:
array([-1.27619048,  1.87619048,  0.57142857,  2.43809524])
```

Step 1: Decomp.
A =PLU

Step 2: Solve y
PLy = b

Step 3: Solve x
Ux = y

Gaussian elimination code (review)

```
def gaussElimin(a,b):
    n = len(b)
    # Elimination Phase
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
    # Back substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

```
A = array([ [7., 3, -1, 2],
            [3, 8, 1, -4],
            [-1, 1, 4, -1],
            [2, -4, -1, 6] ])

b = array([1., 2, 3, 4])
gaussElimin(A, b)
```

```
array([-1.27619048,  1.87619048,  0.57142857,  2.43809524])
```

LU decomposition and solver code

```python
def LUdecomp(a):
    n = len(a)
    # Elimination and [L\U] matrix composition
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                a[i,k] = lam
    return a

def LUsolve(a,b):
    n = len(a)
    # Forward substitution (solve Ly = b)
    for k in range(1,n):
        b[k] = b[k] - dot(a[k,0:k],b[0:k])
    # Back substitution (solve Ux = y)
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

```python
A = array([ [7., 3, -1, 2],
           [3, 8, 1, -4],
           [-1, 1, 4, -1],
           [2, -4, -1, 6] ])

b = array([1., 2, 3, 4])
LUdecomp(A)
LUsolve(A, b)
```

```
array([-1.27619048,  1.87619048,  0.57142857,  2.43809524])
```

Comparison of codes

```python
def gaussElimin(a,b):
    n = len(b)
    # Elimination Phase
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
    # Back substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

```python
def LUdecomp(a):
    n = len(a)
    # Elimination and [L\U] matrix composition
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                a[i,k] = lam
    return a

def LUsolve(a,b):
    n = len(a)
    # Forward substitution (solve Ly = b)
    for k in range(1,n):
        b[k] = b[k] - dot(a[k,0:k],b[0:k])
    # Back substitution (solve Ux = y)
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Exercises 05

- Solve the problems 1-3.

Matrix inversion

”Computing matrix inversion and solving linear equations are related tasks. The most economical way to invert a matrix is to solve

$$AX = I$$

where *I* is the identity matrix.

Inversion of large matrices should be avoided whenever possible because of its high cost. The cost of inversion is considerably more expensive than solving $Ax = b$, for example, with LU decomposition.”

Source: Jaan Kiusalaas (2013). Numerical Methods in Engineering with Python 3.

Matrix inversion in Python

```
from scipy.linalg import inv

A = array([ [7., 3, -1, 2],
           [3, 8, 1, -4],
           [-1, 1, 4, -1],
           [2, -4, -1, 6] ])
inv(A)
```

```
array([[ 0.38730159, -0.32063492,  0.0952381 , -0.32698413],
       [-0.32063492,  0.45396825, -0.0952381 ,  0.39365079],
       [ 0.0952381 , -0.0952381 ,  0.28571429, -0.04761905],
       [-0.32698413,  0.39365079, -0.04761905,  0.53015873]])
```

```
# What is A^(-1)*A?
dot(inv(A), A)
```

```
array([[  1.00000000e+00,   0.00000000e+00,   1.11022302e-16,
          2.22044605e-16],
       [  1.11022302e-16,   1.00000000e+00,  -1.11022302e-16,
         -4.44089210e-16],
       [ -4.16333634e-17,  -2.77555756e-17,   1.00000000e+00,
          0.00000000e+00],
       [  0.00000000e+00,  -4.44089210e-16,  -1.11022302e-16,
          1.00000000e+00]])
```

```
around(dot(inv(A), A), 8)
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1., -0., -0.],
       [-0., -0.,  1.,  0.],
       [ 0., -0., -0.,  1.]])
```

Matrix inversion using LU decomposition

```python
def matInv(a):
    n = len(a[0])
    aInv = identity(n)
    a = LUdecomp(a)
    for i in range(n):
        aInv[:,i] = LUsolve(a,aInv[:,i])
    return aInv
```

```python
A = array([ [7., 3, -1, 2],
           [3, 8, 1, -4],
           [-1, 1, 4, -1],
           [2, -4, -1, 6] ])

matInv(A)
```

```
array([[ 0.38730159, -0.32063492,  0.0952381 , -0.32698413],
       [-0.32063492,  0.45396825, -0.0952381 ,  0.39365079],
       [ 0.0952381 , -0.0952381 ,  0.28571429, -0.04761905],
       [-0.32698413,  0.39365079, -0.04761905,  0.53015873]])
```

```python
## The inverse is found by solving AX = I column by column,
## where b is a column from identity matrix
A = array([ [7., 3, -1, 2],
           [3, 8, 1, -4],
           [-1, 1, 4, -1],
           [2, -4, -1, 6] ])

b = array([1, 0, 0, 0])
solve(A, b)
```

```
array([ 0.38730159, -0.32063492,  0.0952381 , -0.32698413])
```

$$A = \begin{bmatrix} 7 & 3 & -1 & 2 \\ 3 & 8 & 1 & -4 \\ 2 & -4 & -1 & -1 \\ 2 & -4 & -1 & 6 \end{bmatrix} \qquad \mathrm{I} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ \mathbf{0} & 1 & 0 & 0 \\ \mathbf{0} & 0 & 1 & 0 \\ \mathbf{0} & 0 & 0 & 1 \end{bmatrix}$$

$$Ax = b \qquad \mathrm{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \ \ldots$$

Exercises

Solve problems 4-6.

Iterative methods

”Iterative, or indirect methods, start with an initial guess of the solution x and then repeatedly improve the solution until the change in x becomes neglible. Because the required number of iterations can be large, the indirect methods are, in general, slower than direct methods.

Advantages of iterative methods:

1. Iterative procedures are *self-correcting*, meaning the round-off errors (or even mistakes) in one iterative cycle are corrected in subsequent cycles.
2. You need to *store only nonzero elements*. This makes it possible to deal very large and sparse matrices efficiently.”

Source: Jaan Kiusalaas (2013). Numerical Methods in Engineering with Python 3.

Gauss-Seidel method

The equations **Ax=b** are in scalar notation

$$\sum_{j=1}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, n$$

Extracting the term containing x_i from the summation sign yields

$$A_{ii}x_i + \sum_{\substack{j=1\\ j\neq i}}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, n$$

Solving for x_i, we get

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} A_{ij}x_j\right), \quad i = 1, 2, \ldots, n$$

The last equation suggests the following iterative scheme:

$$x_i \leftarrow \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} A_{ij}x_j\right), \quad i = 1, 2, \ldots, n$$

Source: Jaan Kiusalaas (2013). Numerical Methods in Engineering with Python 3.

Gauss-Seidel with relaxation

Convergence of the Gauss-Seidel method can be improved by a technique known as *relaxation*. The idea is to take the new value of x_i as a weighted average of its previous value and the value predicted by Eq. (2.34). The corresponding iterative formula is

$$x_i \leftarrow \frac{\omega}{A_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij} x_j \right) + (1 - \omega) x_i, \quad i = 1, 2, \ldots, n \tag{2.35}$$

where the weight ω is called the *relaxation factor*. It can be seen that if $\omega = 1$, no relaxation takes place, because Eqs. (2.34) and (2.35) produce the same result. If $\omega < 1$, Eq. (2.35) represents interpolation between the old x_i and the value given by Eq. (2.34). This is called *under-relaxation*. In cases where $\omega > 1$, we have extrapolation, or *over-relaxation*.

Source: Jaan Kiusalaas (2013). Numerical Methods in Engineering with Python 3.

Exercises

Solve problems 8-10.