CS 3430: S24: Scientific Computing
Assignment 02
LU Decomposition

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 20, 2024

Learning Objectives

1. LU Decomposition
2. Solving Linear Systems with LU Decomposition

Introduction

In this assignment, we will implement LU decomposition and use it to solve linear systems with the algorithm we discussed in Lecture 03 and will continue to discuss on Lecture 04 on 01/22/2024. This assignment will also give you more exposure to `numpy`. You will save your coding solutions in `cs3430_s24_hw02.py` included in the zip where I wrote some starter code for you and submit it in Canvas. My unit tests for this assignment are in `cs3430_s24_hw02_uts.py`. More on them later.

Optional Problem 0

Review the slides of Lecture 03 in Canvas and/or your class notes if you come to my F2F lectures (and I am really greatful if you are!).

When reviewing Lecture 03, pay close attention to lower-and upper-triangular matrices, forward and backward substitution, the LU decomposition theorem, and how LU decomposition allows us to solve some linear systems by reducing **A** to **U** while simultaneously turning the identity matrix **I** into **L**. Remember that minus when you record row multipliers in **I** (i.e., **I[k,i] = -r**)! The matrix **L** may be incorrect otherwise. I plan to go over this theme some more in Lecture 04 on 01/22/2024.

You may also want to review the numerical instability theorem that we went through at the end of Lecture 03. If you could not attend Lecture 03 in person, I placed a few slides on this theorem at the end of the Lecture 03 PDF in Canvas. Do not think that we are unfairly picking on Python 3 in this theorem. Rest assured that one can construct proofs of this theorem, i.e., find very small or very large real numbers whose behavior do not conform to the regular laws of arithmetic, in many other programming languages. The existence of such constructions is a major reason why we need to use `np.allclose()` or our own customized comparisons when working with real numbers instead of `==`.

Problem 1: LU Decomposition (2 points)

Implement the function `lu_decomp(A, n)` that does the LU decomposition of an $n \times n$ matrix `A` and returns 2 $n \times n$ matrices U (upper-triangular matrix) and L (lower-triangular matrix) such that $LU = A$. For space efficiency, you may want to implement this function in such a way that A is destructively modified as it is being converted into U. While this does not pay off on smaller matrices, it does make a difference on larger ones. But, get the algorithm right first and optimize it later if it works correctly. A couple of test runs of my implementation of `lu_decomp()`.

```
>>> import numpy as np
>>> A = np.array([[2, 3, -1],
                  [0, 1, -3],
                 [4, 5, -2]],
                dtype=float)
>>> AC = A.copy()
>>> np.allclose(A, AC)
True
>>> u, l = lu_decomp(A, A.shape[0])
>>> np.dot(l, u)
array([[ 2.,  3., -1.],
       [ 0.,  1., -3.],
       [ 4.,  5., -2.]])
>>> np.allclose(np.dot(l, u), AC)
True
>>> np.allclose(A, AC)
False
```

As you may have noticed in the above segment, my implementation of `lu_decomp()` is destructive. Hence, `np.allclose()` returns `False` when `A` is compared to its own copy `AC`.

Let us define the function `lud_test(self, a, prnt_flag=True)` to streamline our tests. My source code is in `cs3430_s24_hw02_uts.py`. The first parameter `self` indicates that this function is a method of the class `cs3430_s24_hw02_uts`. The function takes a matrix `a`, checks that it is square, does the LU decomposition of `a` with `lu_decomp()`, computes the LU product (i.e., the product of the two matrices returned by `lu_decomp()`), and uses `np.allclose()` to compare the original matrix `a` with the LU product. If the print flag `prnt_flag` is set to `True`, then matrices U, L, LU, and the original matrix `a` are printed out.

```
    def lud_test(self, a, prnt_flag=True):
        r, c = a.shape
        assert r == c
        u, l = lu_decomp(a.copy(), r)
        ## np.matmul() does the same as np.dot()
        m2 = np.matmul(l, u)
        assert np.allclose(a, m2)
        if prnt_flag:
            print('U:')
            print(u)
            print('L:')
            print(l)
            print('Original Matrix:')
            print(a)
            print('L*U:')
            print(m2)
```

Again, note that in this test I apply `lu_decomp()` to a copy of the matrix `a`, because my implementation of `lu_decomp()` is destructive. If your implementation is not, this test will still pass. Below are a couple of test runs. The mantissas of the real numbers below may be different on your computational devices. It all depends on how many memory unit cells can be allocated to individual floats on your architecture.

```
>>> from cs3430_s24_hw02_uts import cs3430_s24_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s24_hw02_uts()
>>> a = np.array([[2, 3, -1],
                  [0, 1, -3],
                  [4, 5, -2]],
                 dtype=float)
>>> uts.lud_test(a)
U:
[[ 2.  3. -1.]
 [ 0.  1. -3.]
 [ 0.  0. -3.]]
L:
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 2. -1.  1.]]
Original Matrix:
[[ 2.  3. -1.]
 [ 0.  1. -3.]
 [ 4.  5. -2.]]
L*U:
[[ 2.  3. -1.]
 [ 0.  1. -3.]
 [ 4.  5. -2.]]

>>> a = np.array([[73, 136, 173, 112],
                  [61, 165, 146, 14],
                  [137, 43, 183, 73],
                  [196, 40, 144, 31]],
                  dtype=float)
>>> uts.lud_test(a)
U:
[[  73.          136.          173.          112.        ]
 [   0.           51.35616438    1.43835616  -79.5890411 ]
 [   0.            0.         -135.72712723 -466.09895972]
 [   0.            0.            0.          295.71529613]]
L:
[[ 1.          0.          0.          0.        ]
 [ 0.83561644  1.          0.          0.        ]
 [ 1.87671233 -4.13256868  1.          0.        ]
 [ 2.68493151 -6.33128834  2.29420978  1.        ]]
Original Matrix:
[[ 73. 136. 173. 112.]
 [ 61. 165. 146.  14.]
 [137.  43. 183.  73.]
 [196.  40. 144.  31.]]
L*U:
[[ 73. 136. 173. 112.]
 [ 61. 165. 146.  14.]
 [137.  43. 183.  73.]
 [196.  40. 144.  31.]]
```

By the way, as you can see in the above test runs, I display the full float point extensions of the mantissas, because I think that irrational numbers are beautiful and can tell us volumes about computation. But, if you do not agree with me, which is fine, or you want to display only a couple of digits after the decimal point, you can use the Python function `round()`. Below are a couple of examples.

```
>>> import math
>>> math.pi
3.141592653589793
>>> round(math.pi, 2)
3.14
>>> round(math.pi, 3)
3.142
>>> round(math.pi, 4)
3.1416
```

Problem 02: Backward and Forward Substitution (1 point)

Implement the function `bsubst(a, n, b, m)` that uses backward substitution to solve $ax = b_1, b_2, ..., b_m$, where a is an $n \times n$ upper-triangular matrix, n is its dimension, and b is an $n \times m$ matrix of m $n \times 1$ vectors b_1, b_2, ..., b_m. This function returns the $n \times m$ matrix x of m $n \times 1$ vectors x_1, x_2, ..., x_m such that $ax_1 = b_1$,

$ax_2 = b_2$, ..., $ax_m = b_m$.

Here's a test run of my implementation.

```
>>> a = np.array([[1, 3, -1],
                  [0, 2,  6],
                  [0, 0, -15]],
                  dtype=float)
>>> b = np.array([[-4,   5],
                  [10,  11],
                  [-30, 28]],
                  dtype=float)
>>> x = bsubst(a, 3, b, 2)
>>> x
array([[  1.        , -30.16666667],
       [ -1.        ,  11.1       ],
       [  2.        ,  -1.86666667]])
>>> np.dot(a, x[:,0])
array([ -4.,  10., -30.])
>>> np.dot(a, x[:,1])
array([ 5., 11., 28.])
```

Implement the function `fsubst(a, n, b, m)` that uses forward substitution to solve $ax = b_1, b_2, ..., b_m$, where a is an $n \times n$ lower-triangular matrix, n is its dimension, and b is an $n \times m$ matrix of m $n \times 1$ vectors b_1, b_2, ..., b_m. This function returns the $n \times m$ matrix x of m $n \times 1$ vectors x_1, x_2, ..., x_m such that $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$.

Here's a test run of my implementation.

```
>>> a = np.array([[1, 0, 0],
                  [2, 1, 0],
                  [-1, 3, 1]],
                  dtype=float)
>>> b = np.array([[-4, 10],
                  [ 2, 12],
                  [ 4, 21]],
                 dtype=float)
>>> x = fsubst(a, 3, b, 2)
[[ -4.  10.]
 [ 10.  -8.]
 [-30.  55.]]
>>> np.dot(a, x[:,0])
array([-4.,  2.,  4.])
>>> np.dot(a, x[:,1])
array([10., 12., 21.])
```

Problem 03: Solving Linear Systems with LU Decomposition (2 points)

Now that we have `lu_decomp()`, `bsubst()`, and `fsubst()`, we can use the LU decomposition to solve linear systems. Toward that end, implement the function `lud_solve(a, n, b, m)` that applies the LU decomposition method discussed in Lecture 03 to solve the linear system $ax = b_1, b_2, ..., b_m$, where a is an $n \times n$ matrix, b is an $n \times m$ matrix of m $n \times 1$ vectors $b_1, b_2, ..., b_m$.

Specifically, this function uses the LU decomposition to factor the matrix `a` into U and L. Then it uses forward substitution to solve $Ly = b$ for y, uses back substitution to solve $Ux = y$ for x, and returns x, which is an $n \times m$ matrix of m $n \times 1$ vectors x_i such that $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$.

To test this function, we can define the function `check_lin_sys_sol()` that verifies that a specific solution solves the linear system at a given error level. The source code is in `cs3430_s24_hw02_uts.py`. The parameters `a`, `n`, `b`, and `m` are the same as in `lud_solve`, x is an $n \times m$ matrix of m $n \times 1$ vectors x_i such that $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$, and `err` is a given error level.

```
def check_lin_sys_sol(self, a, n, b, m, x, err=0.0e-11):
        ra, ca = a.shape
        assert ra == n
        assert ca == n
        assert b.shape[0] == n
        assert b.shape[1] == m
        assert b.shape == x.shape
        for c in range(m):
            bb = np.array([np.matmul(a, x[:,c])]).T
            for r in range(n):
                assert np.allclose(abs(b[r][c] - bb[r][0]), err)
```

We can use `check_lin_sys_sol()` to define the function `lud_solve_test` to test `lud_solve()`. The source code is in `cs3430_s24_hw02_uts.py`. The parameters `a`, `n`, `b`, and `m` are the same as in `check_lin_sys_sol`, the last two keyword arguments specify an accepted error level and a print flag that, if true, displays the matrices and vectors.

```
    def lud_solve_test(self, a, n, b, m, err=0.0e-11, prnt_flag=True):
        x = lud_solve(a, n, b, m)
        self.check_lin_sys_sol(a, n, b, m, x, err=err)
        if prnt_flag:
            print('A:')
            print(a)
            print('b:')
            print(b)
            print('x:')
            print(x)
            print('A*x:')
            print(np.dot(a, x))
```

Here are a couple of test runs.

```
>>> from cs3430_s24_hw02_uts import cs3430_s24_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s24_hw02_uts()
>>> a = np.array([[1, 3, -1],
                  [2, 8, 4],
                  [-1, 3, 4]],
                 dtype=float)
>>> b = np.array([[-4],
                  [2],
                  [4]],
                 dtype=float)
>>> uts.lud_solve_test(a, 3, b, 1)
A:
[[ 1.  3. -1.]
 [ 2.  8.  4.]
 [-1.  3.  4.]]
b:
[[-4.]
 [ 2.]
 [ 4.]]
x:
[[ 1.]
 [-1.]
 [ 2.]]
A*x:
[[-4.]
 [ 2.]
 [ 4.]]
```

```
>>> a = np.array([[ 73., 136., 173., 112.],
                  [ 61., 165., 146.,  14.],
                  [137.,  43., 183.,  73.],
                  [196.,  40., 144.,  31.]])
>>> b = np.array([[4.0,  1.0],
                  [-1.0, 2.0],
                  [3.0,  3.0],
                  [5.0,  4.0]])
>>> uts.lud_solve_test(a, 4, b, 2)
A:
[[ 73. 136. 173. 112.]
 [ 61. 165. 146.  14.]
 [137.  43. 183.  73.]
 [196.  40. 144.  31.]]
b:
[[ 4.  1.]
 [-1.  2.]
 [ 3.  3.]
 [ 5.  4.]]
x:
[[ 0.04757985  0.01383696]
 [ 0.01254129 -0.00353939]
 [-0.04682867  0.01351588]
 [ 0.06180728 -0.01666954]]
A*x:
[[ 4.  1.]
 [-1.  2.]
 [ 3.  3.]
 [ 5.  4.]]
```

Unit Testing

We can now test `lud_solve` on many linear systems. The pickled files `ab_5x5.pck` and `ab_10x10.pck` in the zip contain 100 randomly generated square linear systems $[\mathbf{A} \mid \mathbf{b}]$ each. The name of the file specifies the size of the system, i.e., in `ab_5x5.pck`, all $\mathbf{A}$ matrices are 5×5 and all column vectors $\mathbf{b}$ is 5×1.

Let us define a function to load the pickled systems from a pck file. The source code is in `cs3430_s24_hw02_uts.py`.

```
    import pickle
    def load_lin_systems(self, file_name):
        with open(file_name, 'rb') as fp:
            return pickle.load(fp)
```

Here's how we can load all 100 linear systems from a given file.

```
>>> from cs3430_s24_hw02_uts import cs3430_s24_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s24_hw02_uts()
>>> linsys = uts.load_lin_systems('ab_5x5.pck')
>>> len(linsys)
100
>>> A, b = linsys[0]
>>> A
array([[110., 176., 124.,  89., 193.],
       [162., 102.,  50., 125., 102.],
       [ 93., 117.,  66., 110., 164.],
       [  3.,  83., 156.,  73., 183.],
       [ 32., 137.,  51., 158.,  38.]])
>>> b
array([[ 36.],
       [165.],
       [116.],
       [156.],
       [125.]])
>>> A1, b1 = linsys[1]
>>> A1
array([[ 18.,  47., 119.,  12.,  64.],
       [ 93., 134.,  71.,  10., 113.],
       [187.,  80., 152.,  92.,  75.],
       [ 11., 194.,  74., 120., 175.],
       [156., 147., 151., 122., 105.]])
>>> b1
array([[ 82.],
       [ 48.],
       [174.],
       [168.],
       [173.]])
```

Let us define another function we can use to test `lud_solve()` on these pickled systems. The source code is in `cs3430_s24_hw02_uts.py`.

```
    def lud_solve_test_lin_systems(self, file_name, err=0.0e-11):
        print('Testing LUD on {} ...'.format(file_name))
        lu_failures = []
        lin_systems = self.load_lin_systems(file_name)
        for A, b in lin_systems:
            try:
                AC = A.copy()
                self.lud_solve_test(A, A.shape[0], b, 1, err=err, prnt_flag=False)
                # make sure that A is intact.
                assert np.allclose(A, AC)
            except Exception as e:
                lu_failures.append((A, b))
        print('Testing LUD had {} failures...'.format(len(lu_failures)))
        assert len(lu_failures) == 0
```

Below are a few test runs.

```
>>> uts = cs3430_s24_hw02_uts()
>>> uts.lud_solve_test_lin_systems('ab_5x5.pck')
Testing LUD on hw/hw02/ab_5x5.pck ...
Testing LUD had 0 failures...
>>> uts.lud_solve_test_lin_systems('ab_10x10.pck')
Testing LUD on hw/hw02/ab_10x10.pck ...
Testing LUD had 0 failures...
```

What to Submit

Save all your code in `cs3430_s24_hw02.py` and submit it in Canvas. I wrote 19 unit tests in `cs3430_s24_hw02_uts.py` for you to test your code with. I recommend that you comment them all out at first and uncomment and run them one by one as you work on each problem.

Happy Hacking!