CS 3430: S22: Scientific Computing
Assignment 02
LU Decomposition

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 22, 2022

Learning Objectives

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

Introduction

In this assignment, we'll implement LU decomposition and use it to solve linear systems with the algorithms we discussed in Lectures 03 and 04. This assignment will also give you more exposure to `numpy`. You'll save your coding solutions in `cs3430_s22_hw02.py` included in the zip where I wrote some starter code for you and submit it in Canvas.

Problem 0: (0 points)

Review the slides of Lectures 03 and 04 in Canvas/Announcements (or your class notes if you're attending F2F lectures, and I'm really greatful if you're, because I want to keep my classes F2F!).

When you review Lecture 03, pay close attention to lower- and upper-triangular matrices, the LU decomposition theorem, and how LU decomposition can be done by reducing **A** to **U** while simultaneously turning the identity matrix **I** into **L**. Remember to keep that minus when you record row multipliers in **I** (i.e., **I[k,i] = -r**)! The matrix **L** won't be correct otherwise.

When reviewing Lecture 04, become comfortable with the algorithms we discussed that use LU decomposition to solve linear systems and the formulas for back and forward substitution. You'll code them up in this assignment.

Problem 1: LU Decomposition (1 point)

Implement the function `lu_decomp(A, n)` that does LU decomposition of an $n \times n$ matrix `A` and returns two $n \times n$ matrices U (upper-triangular matrix) and L (lower-triangular matrix) such that $LU = A$. For space efficiency, you can implement this function in such a way that A is destructively modified as it is being converted into U. But, I wouldn't worry about it if I were you. I've always thought that early optimization is the root of all (or most) evil in software engineering. Get the algorithm right first. You can always optimize it later if it works correctly. You should work with float matrices to prevent `numpy` from agressive truncation in integer division.

Here are a couple of test runs of my implementation of `lu_decomp()`.

```
>>> import numpy as np
from cs3430_s22_hw02 import lu_decomp
>>> a = np.array([[2, 3, -1],
                  [0, 1, -3],
                  [4, 5, -2]],
                  dtype=float)
>>> u, l = lu_decomp(a, 3)
```

```
>>> u
array([[ 2.,  3., -1.],
       [ 0.,  1., -3.],
       [ 0.,  0., -3.]])
>>> l
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 2., -1.,  1.]])
>>> np.dot(l, u)
array([[ 2.,  3., -1.],
       [ 0.,  1., -3.],
       [ 4.,  5., -2.]])
```

Let's implement two functions to streamline unit testing `lu_decomp(a, n)`. We define the function `comp_2d_mats(a, b, er` that compares two matrices `a` and `b` and returns `True` if `a` and `b` are of the same shape and, for every valid position `(i, j)` in each matrix, `abs(a[i][j] - b[i][j]) <= err`, where `err` is a given level of error. The source code for this function is in `cs3430_s22_hw02_uts.py`. Comparing matrices this way gives us some protection against float point arithmetic inconsistencies on different platforms and different operating systems in the sense that using $\leq$ is safer than the equality.

```
def comp_2d_mats(a, b, err=0.0001):
    ra, ca = a.shape
    rb, cb = b.shape
    if ra != rb:
        return False
    if ca != cb:
        return False
    for r in range(ra):
        for c in range(ca):
            if abs(a[r][c] - b[r][c]) > err:
                return False
    return True
```

Let's define the second function `lud_test(self, a, err=0.0001, prnt_flag=True)` to streamline our unit tests. The source code is also in `cs3430_s22_hw02_uts.py`. The first parameter `self` indicates that this function is a method of the class `cs3430_s22_hw02_uts`. The function takes a matrix `a`, checks that it is a square matrix, does 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 `comp_2d_mats()` 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 on the console. This unit test verifies the LU-Decomposition Theorem on Slide 6 in Lecuture 03. You don't need to understand how this theorem is proved. Just understand the statement and know that our algorithms work because of this theorem.

```
    def lud_test(self, a, err=0.0001, prnt_flag=True):
        r, c = a.shape
        assert r == c
        u, l = lu_decomp(a.copy(), r)
        m2 = np.matmul(l, u)
        assert cs3430_s22_hw02_uts.comp_2d_mats(a, m2)
        if prnt_flag:
            print('U:')
            print(u)
            print('L:')
            print(l)
            print('Original Matrix:')
            print(a)
            print('L*U:')
            print(m2)
```

Let's do a couple of test runs.

```
>>> from cs3430_s22_hw02_uts import cs3430_s22_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s22_hw02_uts()
>>> a = np.array([[2, 3, -1],
                  [0, 1, -3],
                  [4, 5, -2]],
                  dtype=float)
>>> uts.lud_test(a, err=0.0001)
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, err=0.0001)
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 like displaying full float point extensions, because I think that irrational numbers are beautiful and can tell us volumes about computation. If you don't agree with me or you want to display only a couple of digits after the decimal point, you can use the Python function `round()`. Below're 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 2: Solving Linear Systems with LU Decomposition (2 points)

Implement the function `bsubst(a, n, b, m)` that uses back 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 $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$. The general formula for back substitution is on Slide 9 of Lecture 04.

Here's a test run.

```
>>> from cs3430_s22_hw02 import bsubst
>>> import numpy as np
>>> 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 $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$. The general formula for forward substitution is on Slide 10 of Lecture 04.

Here's an example.

```
>>> from cs3430_s22_hw02 import fsubst
>>> import numpy as np
>>> 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.])
```

Implement the function `lud_solve(a, n, b, m)` that applies Algorithm 1 (See slides 3–8 in Lecture 04) 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 $b_1, b_2, ..., b_m$.

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, let's 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_s22_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=0.0001):
        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 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_s22_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.

```
    def lud_solve_test(self, a, n, b, m, err=0.0001, 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))
```

A couple of test runs.

```
>>> from cs3430_s22_hw02_uts import cs3430_s22_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s22_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.]]
```

Implement the function `lud_solve2(u, l, n, b, m)` that applies Algorithm 2 (See slides 11–18 in Lecture 04) 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 $b_1, b_2, ..., b_m$. This function takes the U and L matrices obtained from LU Decomposition of a (these matrices are given in the first two parameters). The function uses L to convert b to c as would occur in Gauss-Jordan Elimination to row reduce $[A|b]$ to $[U|c]$, uses back substitution to solve $Ux = c$ for x, and returns x, which is an $n \times m$ matrix of m $n \times 1$ vectors x_i such that $ax_1 = b_1$, $ax_2 = b_2$, ..., $ax_m = b_m$.

We can define the function `lud_solve2_test` to test our implementation of `lud_solve2`. The source code is in `cs3430_s22_hw02_uts.py`. The procedure applies `lu_decomp` to factor a given matrix `a` into `u` and `l`, verifies the correctness of the solution at a given error level with `check_lin_sys_sol`, and, if the print flag is `True`, prints everything out.

```
    def lud_solve2_test(self, a, n, b, m, err=0.0001, prnt_flag=True):
        aa = a.copy()
        u, l = lu_decomp(aa, n)
        bb = b.copy()
```

```
        x = lud_solve2(u, l, n, bb, 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're a couple of test runs.

```
>>> from cs3430_s22_hw02_uts import cs3430_s22_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s22_hw02_uts()
>>> 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],
              [3.0],
              [5.0]])
>>> uts.lud_solve2_test(a, 4, b, 1, err=0.0001)
A:
[[ 73. 136. 173. 112.]
 [ 61. 165. 146.  14.]
 [137.  43. 183.  73.]
 [196.  40. 144.  31.]]
b:
[[ 4.]
 [-1.]
 [ 3.]
 [ 5.]]
x:
[[ 0.04757985]
 [ 0.01254129]
 [-0.04682867]
 [ 0.06180728]]
A*x:
[[ 4.]
 [-1.]
 [ 3.]
 [ 5.]]

>>> 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_solve2_test(a, 4, b, 2, err=0.0001)
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're now in the position to test `lud_solve` on many linear systems. The pickled files `ab_5x5.pck`, `ab_50x50.pck`, and `ab_100x100.pck` in the zip archive for this assignment each contain 100 randomly generated square linear systems $[\mathbf{A} \mid \mathbf{b}]$. The name of the file specifies the size of the system. For example, in `ab_5x5.pck`, all $\mathbf{A}$ matrices are 5×5 and all column vectors $\mathbf{b}$ is 5×1. In case you've never dealt with the `pickle` package, *pickling* is the Python jargon for object persistence. It allows us to persist objects on storage devices and then restore those objects into the running Python.

Let's define a function to load the pickled systems from a pck file. The source code is in `cs3430_s22_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_s22_hw02_uts import cs3430_s22_hw02_uts
>>> import numpy as np
>>> uts = cs3430_s22_hw02_uts()
>>> linsys = uts.load_lin_systems('hw/hw02/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's define another function we can use to test `lud_solve` on these pickled systems. The source code is in `cs3430_s22_hw02_uts.py`.

```
    def lud_solve_test_lin_systems(self, file_name, err=0.0001):
        print('Testing LUD on {} ...'.format(file_name))
        lu_failures = []
        lin_systems = self.load_lin_systems(file_name)
        for A, b in lin_systems:
            try:
                self.lud_solve_test(A, A.shape[0], b, 1, err=err, prnt_flag=False)
            except Exception as e:
                print(e)
                lu_failures.append((A, b))
        print('Testing LUD had {} failures...'.format(len(lu_failures)))
        assert len(lu_failures) == 0
```

Below's my output of running the unit tests on my implementation.

```
***** CS3430: S22: HW02: Problem 02: Unit Test 15 ************
Testing LUD on ab_5x5.pck ...
Testing LUD had 0 failures...
CS 3430: S22: HW02: Problem 02: Unit Test 15: pass
.
***** CS3430: S22: HW02: Problem 02: Unit Test 16 ************
Testing LUD on ab_50x50.pck ...
Testing LUD had 0 failures...
CS 3430: S22: HW02: Problem 02: Unit Test 16: pass
.
***** CS3430: S22: HW02: Problem 02: Unit Test 17 ************
Testing LUD on ab_100x100.pck ...
Testing LUD had 0 failures...
CS 3430: S22: HW02: Problem 02: Unit Test 17: pass
.
```

What to Submit

Save all your code in `cs3430_s22_hw02.py` and submit it in Canvas. I wrote 22 unit tests in `cs3430_s22_hw02_uts.py` for you to test your code with for each problem. As usual, I recommend that you take it one unit test at a time: comment out all unit tests and then uncomment and run them one by one as you work on each problem.

Happy Hacking!