exercise

January 17, 2024

# 1 Tutorial: LU decomposition, sparsity, implementation issues

As in the previous tutorial, this is a jupyter notebook (a way to interactively run python code with visual output and easy editing). If you have never used it, but are a python programmer or want to learn python, I highly recommend checking it out.

[1]:
```
import numpy as np
# nicely display matricesf
from IPython.display import display, Math

def print_matrix(array, prefix=''):
    matrix = ''
    for row in array:
        try:
            for number in row:
                matrix += (f'{number}&' if number != 0 else '&')
        except TypeError:
            matrix += f'{row}&'
        matrix = matrix[:-1] + r'\\'
    display(Math(prefix + r'\begin{bmatrix}'+matrix+r'\end{bmatrix}'))
```

## 1.1 Code for computing LU-decomposition using minimum degree heuristic

[2]:
```
def row_and_col_degrees(matrix, current_begin, m):
    row_degrees = [0] * m
    col_degrees = [0] * m
    for r in range(current_begin, m):
        for c in range(current_begin, m):
            if matrix[r][c] != 0:
                row_degrees[r] += 1
                col_degrees[c] += 1
    return row_degrees, col_degrees

def swap_rows(matrix, r1, r2):
    matrix[[r1,r2],:] = matrix[[r2,r1],:]

def swap_cols(matrix, c1, c2):
```

```
    matrix[:,[c1,c2]] = matrix[:,[c2,c1]]

def eliminate(matrix, current_begin, m):
    x = matrix[current_begin, current_begin]
    if x == 0:
        raise RuntimeError("Matrix is not regular!")
    for r in range(current_begin + 1, m):
        y = matrix[r, current_begin]
        if y == 0: continue
        coeff = y / x
        for c in range(current_begin + 1, m):
            matrix[r, c] -= coeff * matrix[current_begin, c]

def compute_L(matrix):
    l = np.zeros(matrix.shape)
    m = matrix.shape[0]
    facs = [1.0 / matrix[r,r] for r in range(m)]
    for r in range(m):
        for c in range(r):
            l[r,c] = facs[c] * matrix[r,c]
        l[r,r] = 1
    return l

def compute_U(matrix):
    u = np.zeros(matrix.shape)
    m = matrix.shape[0]
    for r in range(m):
        for c in range(r,m):
            u[r,c] = matrix[r,c]
    return u

def permutation_matrix(perm):
    m = len(perm)
    mat = np.zeros((m,m))
    for r in range(m):
        mat[r,perm[r]] = 1
    return mat

def lu_decomposition_min_degree(matrix):
    matrix = np.copy(matrix)
    m = matrix.shape[0]
    row_perm = list(range(m))
    col_perm = list(range(m))
    for current_begin in range(m):
        # compute degrees and min degree rows/columns
        rd, cd = row_and_col_degrees(matrix, current_begin, m)
        min_row_degree = min(range(current_begin, m), key=rd.__getitem__)
```

```
        filtered_col_degree = lambda c: (m if matrix[min_row_degree][c] == 0␣
↪else cd[c])
        min_col_degree = min(range(current_begin, m), key=filtered_col_degree)
        # perform permutation
        swap_rows(matrix, min_row_degree, current_begin)
        row_perm[current_begin], row_perm[min_row_degree] =␣
↪row_perm[min_row_degree], row_perm[current_begin]
        swap_cols(matrix, min_col_degree, current_begin)
        col_perm[current_begin], col_perm[min_col_degree] =␣
↪col_perm[min_col_degree], col_perm[current_begin]
        # eliminate, leaving eliminated entries as they were
        eliminate(matrix, current_begin, m)
    return compute_L(matrix), compute_U(matrix), row_perm, col_perm
```

[3]:
```
example_matrix = np.array([[2,0,4,0,-2], [3,1,0,1,0], [-1,0,-1,0,-2],␣
↪[0,-1,0,0,-6], [0, 0, 1, 0, 4]])
print_matrix(example_matrix, "B = ")
```

$$B = \begin{bmatrix} 2 & & 4 & & -2 \\ 3 & 1 & & 1 & \\ -1 & & -1 & & -2 \\ & -1 & & & -6 \\ & & 1 & & 4 \end{bmatrix}$$

[4]:
```
L, U, row_perm, col_perm = lu_decomposition_min_degree(example_matrix)
```

[5]:
```
print_matrix(L, "L = ")
row_perm
```

$$L = \begin{bmatrix} 1.0 & & & & \\ & 1.0 & & & \\ & -1.0 & 1.0 & & \\ & 4.0 & -2.0 & 1.0 & \\ -1.0 & & -3.0 & & 1.0 \end{bmatrix}$$

[5]:
```
[3, 4, 2, 0, 1]
```

[6]:
```
print_matrix(U, "U = ")
col_perm
```

$$U = \begin{bmatrix} -1.0 & & & -6.0 & \\ & 1.0 & & 4.0 & \\ & & -1.0 & 2.0 & \\ & & & -14.0 & \\ & & & & 1.0 \end{bmatrix}$$

[6]:
```
[1, 2, 0, 4, 3]
```

### 1.2 How are the matrices permuted?

[7]:
```
print_matrix(L @ U, "LU = ")
print_matrix(example_matrix, "B = ")
```

$$LU = \begin{bmatrix} -1.0 & & & -6.0 \\ & 1.0 & & 4.0 \\ & -1.0 & -1.0 & -2.0 \\ & 4.0 & 2.0 & -2.0 \\ 1.0 & & 3.0 & & 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & & 4 & & -2 \\ 3 & 1 & & 1 & \\ -1 & & -1 & & -2 \\ & -1 & & & -6 \\ & & 1 & & 4 \end{bmatrix}$$

[8]:
```
P_col = permutation_matrix(col_perm)
print_matrix(P_col)
P_row = permutation_matrix(row_perm)
print_matrix(P_row)
```

$$\begin{bmatrix} & 1.0 & & & \\ & & 1.0 & & \\ 1.0 & & & & \\ & & & & 1.0 \\ & & & 1.0 & \end{bmatrix}$$

$$\begin{bmatrix} & & & 1.0 & \\ & & & & 1.0 \\ & & 1.0 & & \\ 1.0 & & & & \\ & 1.0 & & & \end{bmatrix}$$

[9]:
```
print_matrix(L @ U @ P_col, "LUP_c = ")
print_matrix(P_row @ example_matrix, "P_rB = ")
```

$$LUP_c = \begin{bmatrix} & -1.0 & & & -6.0 \\ & & & 1.0 & 4.0 \\ -1.0 & & & -1.0 & -2.0 \\ 2.0 & & & 4.0 & -2.0 \\ 3.0 & 1.0 & & & 1.0 \end{bmatrix}$$

$$P_rB = \begin{bmatrix} & -1.0 & & & -6.0 \\ & & & 1.0 & 4.0 \\ -1.0 & & & -1.0 & -2.0 \\ 2.0 & & & 4.0 & -2.0 \\ 3.0 & 1.0 & & & 1.0 \end{bmatrix}$$

[10]:
```
print_matrix(L @ U, "LU = ")
print_matrix(P_row @ example_matrix @ np.transpose(P_col), "P_rBP_c^T = ")
```

$$LU = \begin{bmatrix} -1.0 & & & -6.0 & \\ & 1.0 & & 4.0 & \\ & -1.0 & -1.0 & -2.0 & \\ & 4.0 & 2.0 & -2.0 & \\ 1.0 & & 3.0 & & 1.0 \end{bmatrix}$$

$$P_r B P_c^T = \begin{bmatrix} -1.0 & & & -6.0 & \\ & 1.0 & & 4.0 & \\ & -1.0 & -1.0 & -2.0 & \\ & 4.0 & 2.0 & -2.0 & \\ 1.0 & & 3.0 & & 1.0 \end{bmatrix}$$

Is $P_r^T L U P_c = B$?

[11]:
```
np.all(np.transpose(P_row) @ L @ U @ P_col == example_matrix)
```

[11]:
```
True
```

### 1.3 Solve $Bx = a$ using LU-decomposition (with permutations)

We solve $P_r^T L U P_c \Delta_x = a_j$.

Intermediate systems:

$P_r^T z = a_j \Leftrightarrow z = P_r a_j$, i.e., first just permute $a_j$ according to our row permutation.

$Ly = z$, solve by forward substitution.

$Uw = y$, solve by backward substitution.

$P_c \Delta_x = w$, i.e., undo the column permutation.

[12]:
```
a_j = np.transpose(np.array([7, -2, 0, 3, 0]))
print_matrix(a_j, "a_j = ")
```

$$a_j = \begin{bmatrix} 7 \\ -2 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

[30]:
```
def permute(vec, permutation):
    m = vec.shape[0]
    vec = np.copy(vec)
    vec[list(range(m))] = vec[[permutation[i] for i in range(m)]]
    return vec

print(row_perm)
```

```
z = permute(a_j, row_perm)
print_matrix(z, "z = ")
```

```
[3, 4, 2, 0, 1]
```

$$z = \begin{bmatrix} 3 \\ 0 \\ 0 \\ 7 \\ -2 \end{bmatrix}$$

[14]:
```
def forward_substitution(matrix, b):
    m = matrix.shape[0]
    values = [0] * m
    for r in range(m):
        coeff = matrix[r,r]
        value = b[r]
        for c in range(r):
            value -= matrix[r,c] * values[c]
        values[r] = value / coeff
    return np.transpose(np.array(values))

y = forward_substitution(L, z)
print_matrix(y, "y = ")
```

$$y = \begin{bmatrix} 3.0 \\ 0.0 \\ 0.0 \\ 7.0 \\ 1.0 \end{bmatrix}$$

[15]:
```
def backward_substitution(matrix, b):
    m = matrix.shape[0]
    values = [0] * m
    for r in range(m-1, -1, -1):
        coeff = matrix[r,r]
        value = b[r]
        for c in range(r,m):
            value -= matrix[r,c] * values[c]
        values[r] = value / coeff
    return np.transpose(np.array(values))

w = backward_substitution(U, y)
print_matrix(w, "w = ")
```

$$w = \begin{bmatrix} -0.0 \\ 2.0 \\ -1.0 \\ -0.5 \\ 1.0 \end{bmatrix}$$

[16]:
```
def unpermute(vec, permutation):
    m = vec.shape[0]
    result = np.zeros(vec.shape)
    for i, p in enumerate(permutation):
        result[p] = vec[i]
    return result

delta_x = unpermute(w, col_perm)
print_matrix(delta_x, "\Delta_x = ")
```

$$\Delta_x = \begin{bmatrix} -1.0 \\ -0.0 \\ 2.0 \\ 1.0 \\ -0.5 \end{bmatrix}$$

### 1.4 Putting everything together

[17]:
```
def solve_system(L, U, row_perm, col_perm, a_j):
    z = permute(a_j, row_perm)
    y = forward_substitution(L, z)
    w = backward_substitution(U, y)
    return unpermute(w, col_perm)
```

[18]:
```
print_matrix(solve_system(L, U, row_perm, col_perm, a_j), "\Delta_x = ")
```

$$\Delta_x = \begin{bmatrix} -1.0 \\ -0.0 \\ 2.0 \\ 1.0 \\ -0.5 \end{bmatrix}$$

[31]:
```
print_matrix(np.linalg.solve(example_matrix, a_j))
```

$$\begin{bmatrix} -1.0000000000000002 \\ -0.0 \\ 2.0 \\ 1.0000000000000004 \\ -0.5 \end{bmatrix}$$

### 1.5 Re-using a factorization ($\eta$-files)

Old $B$:

[19]:
```
print_matrix(example_matrix, "B = ")
```

$$B = \begin{bmatrix} 2 & & 4 & & -2 \\ 3 & 1 & & 1 & \\ -1 & & -1 & & -2 \\ & -1 & & & -6 \\ & & 1 & & 4 \end{bmatrix}$$

Assume we want to exchange column 4 with $a_j$:

[20]:
```
a_j = np.transpose(np.array([7,-2,0,3,0]))
print_matrix(a_j, "a_j = ")
```

$$a_j = \begin{bmatrix} 7 \\ -2 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

We usually do that if $a_j$ is the column of our entering variable. So we have solved for $\Delta_x$ w.r.t. that column:

[21]:
```
delta_x = solve_system(L, U, row_perm, col_perm, a_j)
print_matrix(delta_x, "\Delta_x = ")
```

$$\Delta_x = \begin{bmatrix} -1.0 \\ -0.0 \\ 2.0 \\ 1.0 \\ -0.5 \end{bmatrix}$$

Let us rewrite replacing column $i = 4$ by matrix multiplication with $E = I + B^{-1}(a_j - a_i)e_i^T$.

This is $E = I + (\Delta_x - e_i)e_i^T$.

[22]:
```
e_i = np.transpose(np.array([0,0,0,1,0]))
e_i.reshape(5,1)

E = np.eye(5) + (delta_x - e_i).reshape(5,1) @ np.transpose(e_i).reshape(1,5)
print_matrix(E, "E = ")
```

$$E = \begin{bmatrix} 1.0 & & & -1.0 & \\ & 1.0 & & & \\ & & 1.0 & 2.0 & \\ & & & 1.0 & \\ & & & -0.5 & 1.0 \end{bmatrix}$$

[23]:
```
print_matrix(example_matrix, "B = ")
print_matrix(example_matrix @ E, "BE = ")
```

$$B = \begin{bmatrix} 2 & & 4 & & -2 \\ 3 & 1 & & 1 & \\ -1 & & -1 & & -2 \\ & -1 & & & -6 \\ & & 1 & & 4 \end{bmatrix}$$

$$BE = \begin{bmatrix} 2.0 & & 4.0 & 7.0 & -2.0 \\ 3.0 & 1.0 & & -2.0 & \\ -1.0 & & -1.0 & & -2.0 \\ & -1.0 & & 3.0 & -6.0 \\ & & 1.0 & & 4.0 \end{bmatrix}$$

[24]:
```
def inverted_E(u,v):
    return np.eye(u.shape[0]) - (u.reshape(-1,1) @ np.transpose(v).
 ↪reshape(1,-1)) / (1.0 + np.dot(v,u))

E_inv = inverted_E(delta_x - e_i, e_i)
print_matrix(E_inv, "E^{-1} = ")
print_matrix(E, "E = ")
```

$$E^{-1} = \begin{bmatrix} 1.0 & & & 1.0 & \\ & 1.0 & & & \\ & & 1.0 & -2.0 & \\ & & & 1.0 & \\ & & & 0.5 & 1.0 \end{bmatrix}$$

$$E = \begin{bmatrix} 1.0 & & & -1.0 & \\ & 1.0 & & & \\ & & 1.0 & 2.0 & \\ & & & 1.0 & \\ & & & -0.5 & 1.0 \end{bmatrix}$$

Actually, we don't need to work with a full matrix $E^{-1}$, since its inverse has a very nice structure (see board)! It is an identity matrix, with the $i$th column changed by subtracting:

$\frac{\Delta_x - e_i}{\Delta_{x_i}}$.

[25]:
```
def apply_inverted_E(u, delta_x, i):
    fac = u[i] / delta_x[i]
    e_i = np.zeros(delta_x.shape)
    e_i[i] = 1
    return u - fac * (delta_x - e_i)

# test that apply_inverted_E works
t = np.array([1,2,3,4,5])
print_matrix(t, "t = ")
print_matrix(apply_inverted_E(t, delta_x, 3), "E^{-1}t = ")
print_matrix(E_inv @ t, "E^{-1}t = ")
```

$$t = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

$$E^{-1}t = \begin{bmatrix} 5.0 \\ 2.0 \\ -5.0 \\ 4.0 \\ 7.0 \end{bmatrix}$$

$$E^{-1}t = \begin{bmatrix} 5.0 \\ 2.0 \\ -5.0 \\ 4.0 \\ 7.0 \end{bmatrix}$$

Let's suppose we want to solve a system with matrix $BE$ and right-hand side $a_k$, i.e., $P_r^T L U P_c E \Delta_x' = a_k$.

[26]:
```
a_k = np.transpose(np.array([1,2,0,0,0]))
print_matrix(a_k, "a_k = ")
```

$$a_k = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

[27]:
```
def solve_system_with_etas(L, U, row_perm, col_perm, rhs, etas):
    """
    Each eta is a containing the information we need:
    a tuple of (delta_x, replaced_column_index).
    """
    solution = solve_system(L, U, row_perm, col_perm, rhs)  # first a regular␣
 ↪solve
    for delta_x, replaced_index in etas:
        solution = apply_inverted_E(solution, delta_x, replaced_index)
    return solution
```

[28]:
```
solution = solve_system_with_etas(L, U, row_perm, col_perm, a_k, [(delta_x, 3)])
print_matrix(solution, "\Delta_x' = ")
```

$$\Delta_x' = \begin{bmatrix} 1.8571428571428572 \\ 0.42857142857142855 \\ -3.7142857142857144 \\ 2.0 \\ 0.9285714285714286 \end{bmatrix}$$

[29]:
```
# let's check that by solving BEx = a_k using numpy:
print_matrix(np.linalg.solve(example_matrix @ E, a_k), "check = ")
```

$$check = \begin{bmatrix} 1.8571428571428574 \\ 0.42857142857142816 \\ -3.714285714285714 \\ 1.9999999999999998 \\ 0.9285714285714285 \end{bmatrix}$$