Solve with LU decomposition [Direct method]

1. Implement a routine `lu,piv = lu_factor(A)` which computes the LU decomposition $PA = LU$ (by applying Gaussian elimination with row pivoting; see Algorithm 1) for a given matrix $A \in \mathbb{R}^{n\times n}$ and another routine `lu_solve((lu, piv), b)` which takes the output of `lu_factor(A)` and returns the solution x of $Ax = b$ for some $b \in \mathbb{R}^n$ (in case the system admits an *unique* solution).
    - Store L and U in one array `lu` and the permutation P as sparse representation in an array `piv`.
    - If the system $Ax = b$ admits an unique solution then compute it by using your routine `solve_tri` from previous exercises or an appropriate SciPy routine. If the system is not uniquely solvable, check whether the system has infinitely many or no solution and give the user a respective note.
    - *Hint:* With the numpy routines `numpy.triu` and `numpy.tril` you can extract the factors L and U from the array `lu`. Also observe that we expect A to be of square format (for simplicity).
2. Test your routine at least on the systems which you were asked to solve by hand previously. Verify that $PA = LU$ and potentially $Ax = b$. For this purpose you can use `numpy.allclose()`.
3. Find SciPy routines to perform the factorization and solution steps and compare to your routine.

INPUT: $A \in \mathbb{R}^{n\times n}$
OUTPUT: LU decomposition $PA = LU$

```
# FACTORIZATION
initialize piv = [1,2,...,n]
for j = 1,...,n-1 do
    # Find the j-th pivot pivot
    k_j := arg max_{k>=j} |a_kj|
    if a_{k_j j} != 0 then
        # Swap rows
        A[k_j,:] <-> A[j,:]
        piv[k_j] <-> piv[j]
        # Elimination
        for k = j+1,...,n do
            l_kj := a_kj / a_jj
            a_kj = l_kj
            for i = j+1,...,n do
                a_ki = a_ki - l_kj a_ji
            end
        end
    end
end
```

Algorithm 1: In-place Gaussian Elimination with Row Pivoting for implementing Factorization (same as above just without the b vector.)

Solution:

```python
import numpy as np
import scipy.linalg as linalg
```

```python
def lu_factor(A, printsteps=False):
    """
    Compute (partially row) pivoted LU decomposition of a matrix.
    The decomposition is:
       P A = L U
    where P is a permutation matrix, L lower triangular with unit
    diagonal elements, and U upper triangular.

    Parameters
    ----------
    A : (n, n) array_like
        Matrix to decompose
    printsteps : switch to print intermediate steps

    Returns
    -------
    lu : (n, n) ndarray
        Matrix containing U in its upper triangle, and L in its
        lower triangle.
        The unit diagonal elements of L are not stored.
    piv : (n,) ndarray
        Pivot indices representing the permutation matrix P:
        row i of matrix was interchanged with row piv[i].
    """
    m,n = np.shape(A)
    if m != n:
        raise ValueError("expected square matrix")

    # in-place elimination
    lu = A
    # make sure that the data type is 'float'
    lu = lu.astype('float64')
    piv = np.arange(m)
    if printsteps:
        print("input","\n lu =\n",lu, "\n piv=\n",piv,\
                  "\n--------------------------\n")
    ### ELIMINATION with partial row pivoting (-> get P A = L U )
    for j in range(min(m,n)-1):

        # find pivot
        k_piv = j + np.argmax(np.abs(lu[j:,j]))

        # only if pivot is nonzero we proceed, otherwise we go to next column
        if lu[k_piv, j] != 0:
            ## ROW SWAP
            lu[[k_piv, j]] = lu[[j, k_piv]]
            # store row swap in piv
            piv[[k_piv, j]] = piv[[j, k_piv]]

            ## ELIMINATION
            for k in range(j+1,n): # rows
                lkj = lu[k,j] / lu[j,j]
                lu[k,j] = lkj
                for i in range(j+1,n): # columns
                    lu[k,i] = lu[k,i] - lkj*lu[j,i]
        if printsteps:
            print(j,"\n lu =\n", np.round(lu,3), "\n\n piv=\n",piv,\
                  "\n--------------------------\n")
    return lu, piv

def lu_solve(lupiv, b, info=False):
    """
```

```
    Solve a system A x = b, where A is given as
       P A = L U
    with P being a permutation matrix, L lower triangular with unit
    diagonal elements, and U upper triangular.

    Parameters
    ----------
    lupiv : tuple containing lu and piv computed by lu_factor
    b : (n,) ndarray
        right-hand side
    info : switch whether to print info about existence of solutions

    Returns
    -------
    x : (n,) ndarray or None
        unique solution if exists, otherwise nothing
    """
    lu, piv = lupiv
    lu = np.round(lu, 12)
    m,n = np.shape(lu)
    L = np.eye(m,m) + np.tril(lu[:,:min(m,n)], k=-1)
    U = np.triu(lu[:,:n])

    first0row = -1
    if 0 in list(U.diagonal()):
        # check if any diagonal element is zero (then U is singular)
        # if so, overwrite first0row with the row index of the first zero row
        first0row = list(U.diagonal()).index(0.0)
        if info:
            print("first zero row is (start counting from 0):", first0row)
    if first0row < 0:
        # no zero rows (U is invertible) detetected,
        # since first0row is still negative
        if info:
            print("the system A x = b has an unique solution")
        z = linalg.solve_triangular(L, b[piv], lower = True)
        x = linalg.solve_triangular(U, z, lower = False)
        return x
    else:
        # at least one zero row ==> A is singular
        z = linalg.solve_triangular(L, b[piv], lower = True)
        print("z", z)
        if np.all(lu[first0row:,-1]==0) and np.all(z[first0row:]==0):
            # if all rows including transformed rhs are zero then the system has infinitely
    many sols
            if info:
                print("LinAlg Warning: A is singular, but the system A x = b, has infinitely
    many solutions")
        else:
            # otherwise the system does not admit a solution
            if info:
                print("LinAlg Warning: A is singular and no solution exists")
        return None

if __name__ == "__main__":
    printsteps = 1
    AA = [np.array([[2,  1,  3],
                    [1, -1, -1],
                    [3, -2,  2]]),
         np.array([[1,  2,  2],
                   [2,  0,  1],
                   [3,  2,  3]]),
```

```
        np.array([[1,  1,  2],
                  [1, -1,  0],
                  [2,  0,  2]]),
        np.array([[0.5,  -2,  0],
                  [2,8,-2],
                  [1,0,2]])]

    bb = [np.array([-3, 4, 5]),
          np.array([ 1, 3, 4]),
          np.array([ 2, 0, 1]),
          np.array([ -1,10,4])]

    for i, (A,b) in enumerate(zip(AA, bb)):
        print("\n\n--------------------------------\n\
              Example",i+1,\
              "\n--------------------------------")

        print("##### FACTORIZATION #####")
        lu, piv = lu_factor(A, printsteps=printsteps)

        # sparse representation of P^T
        pivT = np.argsort(piv)

        # extract L and U from lu
        m,n = np.shape(A)
        L = np.eye(m,m)+np.tril(lu[:,:min(m,n)], k=-1)
        U = np.triu(lu[:,:n])

        # print tests and results
        print(" A = \n",np.around(A, 3))
        print("\n L = \n",np.around(L, 3))
        print("\n U = \n",np.around(U, 3))
        print("\n piv =",piv,",  pivT =", np.argsort(piv),"\n")
        print(" P A = L U     is ", np.allclose(A[piv],L@U, atol = 1e-6))
        print(" A   = P^T L U is ", np.allclose(A,(L@U)[pivT], atol = 1e-6),"\n")

        print("##### SOLUTION #####")
        x = lu_solve((lu,piv), b, info = True)
        if not np.all(x == None):
            print(" x =\n", np.around(x, 3))
            print("\n A x = b     is ", np.allclose(A@x,b, atol = 1e-6))

        if i<2: # (not that 3. example is not sovable)
            print("\n\n ===== SciPy Factor =====")
            lu, piv = linalg.lu_factor(A)
            L = np.eye(m,m)+np.tril(lu[:,:min(m,n)], k=-1)
            U = np.triu(lu[:,:n])
            print("\n SciPy L = \n",np.around(L, 3))
            print("\n SciPy U = \n",np.around(U, 3))
#             print("\n SciPy piv =", piv," (note that this is LAPACK's piv)\n\n")

    # Another example with a rather large (random) matrix
    from time import time
    n = 10
    A = np.random.rand(n, n)
    b = np.random.rand(n)
    print("\n\n--------------------------------\n\
      Example: Large Random",\
      "\n--------------------------------")

    print("##### FACTORIZATION #####")
    t = time()
```

```python
    lu, piv = lu_factor(A, printsteps=0)
    print("time our factorization:", time()-t)

    # sparse representation of P^T
    pivT = np.argsort(piv)

    # extract L and U from lu
    m,n = np.shape(A)
    L = np.eye(m,m)+np.tril(lu[:,:min(m,n)], k=-1)
    U = np.triu(lu[:,:n])

    # print tests and results
    print(" P A = L U     is ", np.allclose(A[piv],L@U, atol = 1e-6))
    print(" A   = P^T L U is ", np.allclose(A,(L@U)[pivT], atol = 1e-6),"\n")

    t = time()
    lu, piv = linalg.lu_factor(A)
    print("time SciPy factorization:", time()-t)
```