Elements of Mathematics

Exercise Sheet 9

Submission due date: **11.01.2022, 10:15h**

THEORY

1 Frobenius Matrices

Let $\ell_j := (0, \ldots, 0, \ell_{j+1,j}, \ldots, \ell_{m,j})^\top \in \mathbb{R}^m$, $e_j \in \mathbb{R}^m$ be the j-th unit vector and $I \in \mathbb{R}^{m \times m}$ be the identity matrix. Then show that the matrix

$$L_j := I + \ell_j e_j^\top \in \mathbb{R}^{m \times m}$$

satisfies:

1. The matrix L_j is an invertible lower triangular matrix.
2. The inverse of L_j is given by $L_j^{-1} := I - \ell_j e_j^\top \in \mathbb{R}^{m \times m}$.
3. For $i \leq j$ it holds that $L_i L_j = I + \ell_j e_j^\top + \ell_i e_i^\top$ and $L_i^{-1} L_j^{-1} = I - \ell_j e_j^\top - \ell_i e_i^\top$.

(9 Points)

Solution:

1. First note that $\ell_j e_j^\top$ is a lower triangular matrix with zeroes on its diagonal because $\ell_{i,j} = 0$ for $i \leq j$. Therefore L_j is a lower triangular matrix with ones on its diagonal and thus invertible (note, e.g., that $\det(L_j) = 1 \neq 0$).
2. Since the inverse matrix is unique it is sufficient to show that $L_j(I - \ell_j e_j^\top) = I$. By inserting the definition we find that

$$\begin{aligned} L_j(I - \ell_j e_j^\top) &= (I + \ell_j e_j^\top)(I - \ell_j e_j^\top) \\ &= I + \ell_j e_j^\top - \ell_j e_j^\top - \ell_j e_j^\top \ell_j e_j^\top \\ &= I - \ell_j (e_j^\top \ell_j) e_j^\top \\ &= I, \end{aligned}$$

   where we have exploited $e_j^\top \ell_j = 0$ which follows from $\ell_{j,j} = 0$.
3. We insert definitions and compute the products. First,

$$\begin{aligned} L_i L_j &= (I + \ell_i e_i^\top)(I + \ell_j e_j^\top) \\ &= I + \ell_i e_i^\top + \ell_j e_j^\top + \ell_i e_i^\top \ell_j e_j^\top \\ &= I + \ell_i e_i^\top + \ell_j e_j^\top + \ell_i (e_i^\top \ell_j) e_j^\top \\ &= I + \ell_i e_i^\top + \ell_j e_j^\top, \end{aligned}$$

   where we have exploited $e_i^\top \ell_j = 0$, which follows from $\ell_{i,j} = 0$ for all $i \leq j$. The second statement follows along the same lines.

2 Permutation Matrices and Row Swap

A matrix $P \in \mathbb{R}^{m\times m}$ is called *permutation matrix* if it has exactly one entry of 1 in each row and each column and 0s elsewhere.

1. **Sparse representation:** A permutation matrix $P = (p_{ij})_{ij} \in \mathbb{R}^{m\times m}$ can be represented by an m-dimensional vector $\texttt{piv} \in \{1,2,\ldots,m\}^m$ in the following way:

$$\texttt{piv}_i = j \quad :\Leftrightarrow \quad p_{ij} = 1.$$

   Given a permutation matrix P with its sparse representation piv. Determine the sparse representation, say $\texttt{pivT} \in \{1,2,\ldots,m\}^m$, of the transpose $P^T = (\widetilde{p}_{ij})_{ij}$, so that

$$\texttt{pivT}_i = j \quad :\Leftrightarrow \quad \widetilde{p}_{ij} = 1.$$

   Hint: Have a look at the routine `numpy.argsort()`.

2. **Inverse:** Show that the inverse of a permutation matrix $P \in \mathbb{R}^{m\times m}$ is given by its transpose, i.e., $P^T = P^{-1}$.

3. **Row Swap**: Let $P_{jk} \in \mathbb{R}^{m\times m}$ be the permutation matrix which results from interchanging the j-th and k-th row ($k \geq j$) of the identity matrix in $\mathbb{R}^{m\times m}$. Thus if its applied to a matrix $A \in \mathbb{R}^{m\times n}$ it interchanges the j-th and k-th row of A. Show that

$$P_{jk}^\top = P_{jk}.$$

   In particular we find $P_{jk} = P_{jk}^{-1}$, i.e., P_{jk} is self-inverse.

(9 Points)

Solution:

1. $\texttt{piv}_i = j \Leftrightarrow 1 = p_{ij} = \widetilde{p}_{ji} \Leftrightarrow: \texttt{pivT}_j = i$
2. By definition, the columns of a permutation matrix are given by the m unit vectors (in potentially permuted order), which are orthonormal.
3. Let $P_{jk} = (q_{i\ell})_{i\ell}$ and $P_{jk}^T = (\widetilde{q}_{i\ell})_{i\ell}$, then by definition we find

$$q_{i\ell} = \begin{cases} 1: & (i = \ell, k \neq i \neq j) \text{ or } (i = j, \ell = k) \text{ or } (i = k, \ell = j) \\ 0: & \text{else} \end{cases}$$

   and therefore

$$\widetilde{q}_{i\ell} = q_{\ell i} = \begin{cases} 1: & (\ell = i, k \neq \ell \neq j) \text{ or } (\ell = j, i = k) \text{ or } (\ell = k, i = j) \\ 0: & \text{else} \end{cases}.$$

   Thus, we obviously find $q_{i\ell} = \widetilde{q}_{i\ell}$.

3 Proof for LU decomposition (with row pivoting)

Let $m \in \mathbb{N}$. As above, let $P_{jk} \in \mathbb{R}^{m\times m}$ be the permutation matrix which results from interchanging the j-th and k-th row ($k \geq j$) of the identity matrix in $\mathbb{R}^{m\times m}$. Further for $\ell_j := (0,\ldots,0,\ell_{j+1,j},\ldots,\ell_{m,j})^\top \in \mathbb{R}^m$ and the j-th unit vector $e_j \in \mathbb{R}^m$, let $L_j := I + \ell_j e_j^\top \in \mathbb{R}^{m\times m}$. Then show that for all $1 \leq j < i \leq k_i \leq m$ we have

$$P_{ik_i} L_j = \widehat{L}_j P_{ik_i}$$

where $\widehat{L}_j := I + (P_{ik_i}\ell_j)e_j^\top$. *(4 Points)*

Solution:

We find

$$
\begin{aligned}
P_{ik_i} L_j &= P_{ik_i}\left(I + \ell_j e_j^\top\right) \\
&= P_{ik_i} + P_{ik_i}\ell_j e_j^\top \\
&= P_{ik_i} + P_{ik_i}\ell_j e_j^\top P_{ik_i}^\top P_{ik_i} \\
&= (I + P_{ik_i}\ell_j e_j^\top P_{ik_i}^\top) P_{ik_i} \\
&= (I + P_{ik_i}\ell_j (P_{ik_i} e_j)^\top) P_{ik_i}. \\
&= (I + P_{ik_i}\ell_j e_j^\top) P_{ik_i}.
\end{aligned}
$$

Since $j < i \leq k_i$ we find that $P_{ik_i} e_j = e_j$, since only zeroes are swapped.

PROGRAMMING

4 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 **??**) 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.

(12 Points)

Solution:

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

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
    ----------
```

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 := argmax_{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: Gaussian Elimination with Row Pivoting

```
    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
```

```python
        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 = 0
    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()
    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)
```

5 The Cholesky Decomposition for SPD Matrices

1. Find SciPy routines to perform the Cholesky factorization and solution steps separately. Non-obligatory: Implement them on your own.

2. Compare the performance of the Cholesky routines to the LU routines from SciPy by testing them on large symmetric and positive definite matrices $A \in \mathbb{R}^{n\times n}$ (e.g., $A = B^T B + \delta I$ for some random $B \in \mathbb{R}^{n\times n}$ and $\delta > 0$) and some (random) right-hand side $b \in \mathbb{R}^n$.

(6 Points)

Solution:

```
import numpy as np
import scipy.linalg as linalg
from time import time

def Aspd(n,delta):
    B = np.random.rand(n,n)
    return B.T@B + delta * np.eye(n)
```

```python
def FacSol(A,b, method = "lu"):
    """
    ...
    """
    # choose correct SciPy routines
    if method == "lu":
        factor = lambda A : linalg.lu_factor(A)
        solve = lambda tup, b : linalg.lu_solve(tup, b, overwrite_b=True)
    else:
        factor = lambda A : linalg.cho_factor(A)
        solve = lambda tup, b : linalg.cho_solve(tup, b, overwrite_b=True)

    # factor
    tfac = time()
    tup = factor(A)
    tfac = time()-tfac

    # solve
    tsolve = time()
    x = solve(tup, b)
    tsolve = time()-tsolve

    return {"x": x, method: tup, "t": (tfac, tsolve)}

if __name__ == "__main__":
    n = 1000
    delta = 0.5
    runs = 10
    for i in range(runs):
        print("run", i)
        A = Aspd(n, delta)
        b = np.random.rand(n)
        print("dim, method, [t_factor t_solve], Ax==b")
        for method in ["lu", "cho"]:
            data = FacSol(A.copy(),b.copy(), method)
            print(n, method, "\t ",np.round(data["t"],4),"\t   ",np.allclose(A.dot(
    data["x"]), b))
        print("---------------------------------")
```

Total Number of Points = 40 (T:22, P:18)