EE2703 - Week 8

Santhosh S P ee21b119

April 16, 2023

1 Importing required libraries

[ ]:
```
import numpy as np
%load_ext cython
```

- Cython module should be installed before using `pip install cython` in the environment.
- We import numpy and load the cython module using the magic command `%load_ext`.

2 Input matrices for testing

`A` and `B` are `10 x 10` and `10 x 1` matrices made up of random floats.

[ ]:
```
A=100*np.random.rand(10,10)
B=100*np.random.rand(10,1)
```

3 Some elementary functions

Three functions are defined:

- `leftmost_nonzero_col_finder(matrix)`: This finds the leftmost column of the augmented matrix, containing atleast one non-zero entry. This function returns the index of the found column.
- `swaprows(matrix,leftmost_nonzero_col_index2)`: This makes sure that the first non-zero column has a non-zero entry in the first row. If not, its made non-zero by swapping rows. This function returns a 2-D numpy array with rows swapped.
- `makezeros(matrix,leftmost_nonzero_col_index3)`: This function makes the elements in the pivot position 1 and makes all elements below the pivot position zero. This function returns a 2-D numpy array, which is the Row-Echelon-Form of the given augmented matrix.

[ ]:
```
def leftmost_nonzero_col_finder(matrix):
    leftmost_nonzero_col_index1=0
    num_rows,num_cols=np.shape(matrix)
    #finding the leftmost column containing a non zero entry
    for i in range(num_cols):
```

```
        #the below statement prints true if a column wuth atleast one non-zero␣
 ↪entry is detected
        if not(np.all(matrix[:,i]==0)):
            leftmost_nonzero_col_index1=i
            break

    #print(leftmost_nonzero_col_index1)
    return leftmost_nonzero_col_index1


def swaprows(matrix,leftmost_nonzero_col_index2):
    #to make sure that first non-zero column has a non-zero entry in the first␣
 ↪row (by row swapping)

    #in the below block, if the element in the first row is non-zero, no␣
 ↪swapping is done
    #if not swapping is done
    col1=matrix[:,leftmost_nonzero_col_index2]
    ind1=np.nonzero(col1)[0][0]
    matrix[[0, ind1]] = matrix[[ind1, 0]]
    return matrix


def makezeros(matrix,leftmost_nonzero_col_index3):

    num_rows,num_cols=np.shape(matrix)

    #zeroes below the pivot position
    for j in range(num_rows):

        #ERT to make topmost position of col 1
        if j==0:
            matrix[j,:]=matrix[j,:]/matrix[j,leftmost_nonzero_col_index3]
        #zeros below the pivot
        else:
            matrix[j,:]=matrix[j,:]-(matrix[j,leftmost_nonzero_col_index3]/
 ↪matrix[0,leftmost_nonzero_col_index3])*matrix[0,:]

    return matrix
```

4 The `GaussJordanSolver()` function

The algorithm used is:

- First the left-most column containing a non-zero entry should be determined.
- The first non-zero column should have a non-zero entry in the first row, which can be made sure by swapping the rows if needed.

- Perform elementary row transformations (ERT) to make the first non-zero entry 1 (found in the previous step) and make the entries below this leading 1 equal to 0.
- Repeat the above three steps again on the submatrix consisting of all except the first row, till the final row. Now the matrix is in Row-Echelon-Form.
- For each row containing a leading 1, use ERT to make elements above the leading 1 in each row, zero. Now the matrix is in RREF form.

[ ]:
```python
def GaussJordanSolve(A,B):

    #augmented matrix
    input=np.c_[A,B]

    num_rows,num_cols=np.shape(input)

    #creating an empty matrix
    ref_matrix=np.zeros_like(input)

    # actual reduction process happens here
    for num in range(num_rows):

        if num!=0:
            input=input[1:,:]

        #finding the left-most column containing a non-zero entry
        leftrow=leftmost_nonzero_col_finder(input)

        # to make sure that the first non-zero column has a non-zero entry in␣
↪the first row
        input=swaprows(input,leftrow)

        #making pivot 1 and the entries below it zero
        input=makezeros(input,leftrow)

        # the above steps are repeated for submatrix consisting of all except␣
↪the first row

        #row echoleon form
        ref_matrix[num:,:]=input[0,:]


    #indices of pivots
    pivots=np.argwhere(ref_matrix==1)

    #clearing the values upward (i.e ERT to make them zero, starting form the␣
↪rightmost pivot)
    for i in range(-1,-len(pivots)-1,-1):
```

```
        #location of the pivot
        row_ind=pivots[i][0]
        col_ind=pivots[i][1]

        #clearing the values upward
        for j in range(row_ind):
            if ref_matrix[j][col_ind]!=0:
                ref_matrix[j,:]=ref_matrix[j,:
↪]-ref_matrix[j][col_ind]*ref_matrix[row_ind,:]

    #ref_matrix contains the RREF of the augmented matrix

    # Examining the RREF to comment on the nature of the solution

    #if the system contains a row full of zeroes except the final element
    #then the system is inconsistent
    for row in ref_matrix:
        if len(np.nonzero(row[0:-1])[0])==0 and row[-1]!=0:
            #print("Inconsistent")
            return None


    #position of the pivots in RREF
    new_pivots=np.argwhere(ref_matrix==1)
    sum=0


    for k in new_pivots:
        if k[0]==k[1]:
            sum+=1

    # if every row has a pivot (1) and the final column is non-zero, it has an␣
↪unique solution
    if sum==num_rows and not(np.all(ref_matrix[:,i]==0)):
        #print("Unique solution")
        return np.array([ref_matrix[:,-1]]).T

    #otherwise it has infinite solutions
    else:
        #print("Infinite solutions")
        return None
```

5 Comparing the results

5.1 Testing the `GaussJordanSolve()` function

[ ]:
```
print(GaussJordanSolve(A=A,B=B))
%timeit GaussJordanSolve(A=A,B=B)
```

```
[[ 0.92894496]
 [-2.07699835]
 [ 1.498962  ]
 [ 1.2266986 ]
 [-0.23390642]
 [-1.77001101]
 [-0.95917603]
 [ 2.25385485]
 [-0.05683222]
 [ 0.27177082]]
767 µs ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

5.2 Solving using `np.linalg.solve()` function

[ ]:
```
print(np.linalg.solve(A,B))
```

```
[[ 0.92894496]
 [-2.07699835]
 [ 1.498962  ]
 [ 1.2266986 ]
 [-0.23390642]
 [-1.77001101]
 [-0.95917603]
 [ 2.25385485]
 [-0.05683222]
 [ 0.27177082]]
```

6 Optimized version using Cython

6.1 Importing the required modules

- `%%cython --annotate` makes cython available for the particular cell to run and shows the python-cython interaction for each line of the code.
- The `cimport` statement imports the necessary C types and functions from the NumPy library.

6.2 Optimizing the `leftmost_nonzero_col_finder()` function

- The input matrix is explicitly declared as a 2-D NumPy array of type `np.float64_t`. This enables faster access to the array elements.
- The `cpdef` keyword is used here to make sure that it can be called from both Python and C. The `int` before the function name is added as the function returns an integer, similar to the

sytax of defining functions in C.

- The variables `leftmost_nonzero_col_index1`, `num_rows`, `num_cols` are defined as C variables of type `int`.
- Similarly the index of the for loop, `i` is also defined as a C variable of type `int`.

6.3 Optimizing the `swaprows()` function

- The input matrix is explicitly declared as a 2-D NumPy array of type `np.float64_t` and the input index is defined as `int`.
- The `cpdef` keyword is used here to make sure that it can be called from both Python and C. The `ndarray[np.float64_t, ndim=2]` before the function name is added as the function returns a 2-D NumPy array of type `np.float64_t`, similar to the sytax of defining functions in C.
- The variable `ind1` is defined as C variable of type `int`.

6.4 Optimizing the `makezeros()` function

- The input matrix is explicitly declared as a 2-D NumPy array of type `np.float64_t` and the input index is defined as `int`.
- The `cpdef` keyword is used here to make sure that it can be called from both Python and C. The `ndarray[np.float64_t, ndim=2]` before the function name is added as the function returns a 2-D NumPy array of type `np.float64_t`, similar to the sytax of defining functions in C.
- The variables `num_rows` and `num_cols` are defined as C variables of type `int`.
- `new_matrix` is declared as a 2-D NumPy array of type `np.float64_t`.

6.5 Optimizing the `GaussJordanSolve()` function

- The input matrices A and B are declared as a 2-D NumPy arrays of type `np.float64_t`.
- The `cpdef` keyword is used here to make sure that it can be called from both Python and C. The `ndarray[np.float64_t, ndim=2]` before the function name is added as the function returns a 2-D NumPy array of type `np.float64_t`, similar to the sytax of defining functions in C.
- The variables `num_rows` and `num_cols` are defined as C variables of type `int`. Also the variables `num`,`leftrow`,`row_ind`,`col_ind`,`i`,`j` are also defined as `int`.
- `pivots`,`new_pivots`,`ref_matrix` and `input` are declared as a 2-D NumPy arrays of type `np.float64_t`.
- `k` is defined as a 1-D NumPy array of type `np.float64_t`.
- `sum` is defined as type `np.float64_t`.

[ ]:
```
%%cython --annotate
import numpy as np
cimport numpy as np
```

```
from numpy cimport ndarray


cpdef int leftmost_nonzero_col_finder(ndarray[np.float64_t, ndim=2] matrix):
    cdef int leftmost_nonzero_col_index1=0
    cdef int num_rows = matrix.shape[0]
    cdef int num_cols = matrix.shape[1]
    #finding the leftmost column containing a non zero entry

    cdef int i
    for i in range(num_cols):

        #the below statement prints true if a column wuth atleast one non-zero entry is detected
        if not(np.all(matrix[:,i]==0)):
            leftmost_nonzero_col_index1=i
            break

    #print(leftmost_nonzero_col_index1)
    return leftmost_nonzero_col_index1



cpdef  ndarray[np.float64_t, ndim=2] swaprows(ndarray[np.float64_t, ndim=2] matrix,int leftmost_nonzero_col_index2):
    #to make sure that first non-zero column has a non-zero entry in the first row (by row swapping)

    #in the below block, if the element in the first row is non-zero, no swapping is done
    #if not swapping is done
    col1=matrix[:,leftmost_nonzero_col_index2]
    cdef int ind1 = np.where(matrix[:,leftmost_nonzero_col_index2]!=0)[0][0]
    matrix[[0, ind1]] = matrix[[ind1, 0]]
    return matrix


cpdef ndarray[np.float64_t, ndim=2] makezeros(ndarray[np.float64_t, ndim=2] matrix,int leftmost_nonzero_col_index3):
    cdef int num_rows = matrix.shape[0]
    cdef int num_cols = matrix.shape[1]

    # Create a new zero-filled array of the same shape and type as the input array
    cdef ndarray[np.float64_t, ndim=2] new_matrix = np.zeros_like(matrix)
```

```
    # Copy the input array into the new array
    new_matrix[:] = matrix[:]

    #zeros below the pivot position
    cdef int j
    for j in range(num_rows):

        #ERT to make topmost position of col 1
        if j==0:
            new_matrix[j,:]=new_matrix[j,:]/
↪new_matrix[j,leftmost_nonzero_col_index3]
        #zeros below the pivot
        else:
            new_matrix[j,:]=new_matrix[j,:
↪]-(new_matrix[j,leftmost_nonzero_col_index3]/
↪new_matrix[0,leftmost_nonzero_col_index3])*new_matrix[0,:]

    return new_matrix


cpdef np.ndarray[np.float64_t, ndim=2] GaussJordanSolve(np.ndarray[np.
↪float64_t, ndim=2] A,
                                                       np.ndarray[np.float64_t,␣
↪ndim=2] B):
    cdef np.ndarray[np.float64_t, ndim=2] input = np.c_[A,B]
    cdef int num_rows = input.shape[0]
    cdef int num_cols = input.shape[1]

    cdef np.ndarray[np.float64_t, ndim=2] ref_matrix = np.zeros_like(input)
    cdef int num,leftrow,row_ind,col_ind,i,j
    cdef np.ndarray[np.int64_t, ndim=2] pivots, new_pivots
    cdef np.ndarray[np.int64_t, ndim=1] k
    cdef np.float64_t sum

    for num in range(num_rows):
        if num!=0:
            input=input[1:,:]
        leftrow=leftmost_nonzero_col_finder(input)
        input=swaprows(input,leftrow)
        input=makezeros(input,leftrow)
        #row echoleon form
        ref_matrix[num:,:]=input[0,:]

    #indices of pivots
    pivots=np.argwhere(ref_matrix==1)
```

```
    #clearing the values upward (i.e ERT to make them zero, starting form the␣
↪rightmost pivot)
    for i in range(-1,-len(pivots)-1,-1):
        row_ind=pivots[i][0]
        col_ind=pivots[i][1]

        for j in range(row_ind):
            if ref_matrix[j][col_ind]!=0:
                ref_matrix[j,:]=ref_matrix[j,:
↪]-ref_matrix[j][col_ind]*ref_matrix[row_ind,:]

    #ref_matrix is the RREF now
    for row in ref_matrix:
        if len(np.nonzero(row[0:-1])[0])==0 and row[-1]!=0:
            #print("Inconsistent")
            return None

    new_pivots=np.argwhere(ref_matrix==1)
    sum=0
    for k in new_pivots:
        if k[0]==k[1]:
            sum+=1
    if sum==num_rows and not(np.all(ref_matrix[:,i]==0)):
        #print("Unique solution")
        return np.array([ref_matrix[:,-1]]).T
    else:
        #print("Infinite solutions")
        return None
```

[ ]: <IPython.core.display.HTML object>

7 Result of the optimized version

The unoptimized version and optimized version of `GaussJordanSolve()` take almost the same time to run.

[ ]:
```
print(GaussJordanSolve(A=A,B=B))
%timeit GaussJordanSolve(A=A,B=B)
```

```
[[ 0.92894496]
 [-2.07699835]
 [ 1.498962  ]
 [ 1.2266986 ]
 [-0.23390642]
 [-1.77001101]
 [-0.95917603]
 [ 2.25385485]
```

```
 [-0.05683222]
 [ 0.27177082]]
776 µs ± 15.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```