02_17_sweeping

Unknown Author

February 17, 2014

Part I

Matrices, Inverses, Equations, Solution

Today I'll be using lists as I am still not sure if numpy and scipy are installed in the lab.

1 Matrices, reading from a file.

It will be painful to input a 16x16 matrix whenever we need to. So we shall load matrices from files. We shall use the split function (*method*) we learnt before. We shall write a function which takes the name of file as input and returns a list of lists of numbers.

In [1]:
```
def read_matrix(filename) :
    fl = open(filename, 'r')
    matrix = []
    for line in fl :
        row = []
        words = line.split()
        for word in words :
            row.append(float(word))
        matrix.append(row)
    return matrix
```

In [2]:
```
A = read_matrix("files/A.txt")
print A
```

```
[[1.0, 2.0, 3.0], [2.0, 1.0, 2.0], [3.0, 2.0, 1.0]]
```

The output isn't really very pretty. Let us make it pretty by

In [3]:
```
def print_matrix(matrix, myformat) :
    for row in matrix :
        for no in row :
            print myformat % no,
        print
```

In [4]:
```
print_matrix(A, "%10.0f")
```

```
         1          2          3
         2          1          2
         3          2          1
```

2 Coding elementary row operations

Row operation 1 : Interchanging two rows

As one can guess, this will be a function which takes a matrix as an input, and row numbers of two rows; like `elem1(A, 0, 2)`. However, before we do that, it might be useful to define a function called `copy_row(source)`.

In [5]:
```
def copy_row(src) :
    dest = []
    for entry in src :
        dest.append(entry)
    return dest

def elem1(matrix, row1, row2) :
    if row1 < 0 or row2 < 0 or row1 >= len(matrix) or row2 >= len(matrix) :
        print "Row out of range."
    else :
        temp_row = copy_row(matrix[row1])
        matrix[row1] = copy_row(matrix[row2])
        matrix[row2] = copy_row(temp_row)
    return matrix
```

Note lists are also passed by reference. So modifying matrix actually changes the actual matrix. You can see it below.

In [6]:
```
print_matrix(elem1(A, 1, 2), "%2.0f")
print "--------------------------"
print_matrix(A, "%2.0f")
```

```
 1  2  3
 3  2  1
 2  1  2
--------------------------
 1  2  3
 3  2  1
 2  1  2
```

Row operation 2 : multiplying a row by a non-zero scalar

The functionw will take a matrix, and row number and a scalar as input. We'll also check if the scalar is non-zero.

In [7]:
```
def elem2(matrix, row, scalar) :
    if row < 0 or row >= len(matrix) :
        print "row %d out of range." % row
    elif scalar == 0 :
        print "scalar has to be non-zero."
    else :
        for i in range(len(matrix[row])) :
            matrix[row][i] *= scalar
    return matrix
```

In [8]:
```
print_matrix(A, "%2.0f")
print "---------------------------"
print_matrix(elem2(A, 0, 4), "%2.0f")
print "--------------------------"
print_matrix(A, "%2.0f")
```

```
 1  2  3
 3  2  1
 2  1  2
---------------------------
 4  8 12
 3  2  1
 2  1  2
--------------------------
 4  8 12
 3  2  1
 2  1  2
```

row operation 3 : replacing a row by the sum of that row and a multiple of another row.

Here we need 4 inputs to the function: + The matrix + The row to be changed + The row whose multiple will be added to the first row, + a scalar : the multiplication factor.

Again for this we use some helper functions which are readily available for vectors.

In [9]:
```
def add_vects(lst1, lst2) :
    return [a + b for (a, b) in zip(lst1, lst2)]

def scalar_mult(a, lst) :
    return [a * no for no in lst]


def elem3(matrix, row_2b_changed, row_used_4_change, scalar) :
    temp_row = copy_row(scalar_mult(scalar, matrix[row_used_4_change]))
    matrix[row_2b_changed] = add_vects(matrix[row_2b_changed], temp_row)
    return matrix
```

In [10]:
```
print_matrix(A, "%2.0f")
print "-"*20
print_matrix(elem3(A, 0, 1, -2), "%2.0f")
print "-"*21
print_matrix(A, "%2.0f")
```

```
 4  8 12
 3  2  1
 2  1  2
--------------------
-2  4 10
 3  2  1
 2  1  2
---------------------
-2  4 10
 3  2  1
 2  1  2
```

3 Sweeping a column

Let us read another matrix B. This is the same one as in the class last week.

In [11]:
```
B = read_matrix("files/B.txt")
print_matrix(B, "%2.0f")
```

```
 0  2  3  2
 1  4  5  5
 3  0  2  6
```

If we recall, sweeping a column requires a matrix and an entry (position in the matrix) which we call pivot. We shall define the pivot as the 2-tuple (row_no, col_no).

In [12]:
```
def sweep(matrix, pivot) :
    (r, c) = pivot
    if r < 0 or r >= len(matrix) :
        print "row out of range"
    elif c < 0 or c >= len(matrix[r]) :
        print "column out of range"
    elif matrix[r][c] == 0:
        # pivot cannot be zero.
        print("pivot cannot be zero.")
    else :
        # print_matrix(matrix, "%5.2f")
        pivot = matrix[r][c]

        # Step 1 : multiply the row containing the pivot by 1/pivot to make the pivot
        matrix = elem2(matrix, r, 1.0/pivot)
        # print_matrix(matrix, "%5.2f")

        # Step 2 : for row != that of pivot, subtract matrix[row][c] times the row con
        for i in range(len(matrix)) :
            if i != r :
                matrix = elem3(matrix, i, r, -matrix[i][c])
                # print_matrix(matrix, "%5.2f")
    return matrix
```

In [13]:
```
print_matrix(sweep(B, (1, 2)), "%6.2f")
```

```
 -0.60  -0.40   0.00  -1.00
  0.20   0.80   1.00   1.00
  2.60  -1.60   0.00   4.00
```

4 Gaussian elimination

Gaussian elemination is an algorithm to reduce a matrix to its reduced echelon form. Here *reduction* means performing row operations till the final matrix satisfies the definition of a reduced echelon form. For your convenience let us recall the definition.

An $m \times n$ matrix is said to be in echelon form, if it has r, $0 \leq r \leq m$ non-zero rows and 1. All the non-zero rows are on the top. 1. For $1 \leq i \leq r$, p_i denotes the column containing the *first non-zero entry* of the i-th row, then $p_1 < p_2 < \cdots < p_r$. 1. $a_{ip_i} = 1$. The matrix is said to be in reduced echelon form if in addition to being in the echelon form, the p_i'th columns have all but one zeroes. The non-zero entry is a_{ip_i}.

Example :

The matrix

0 1 4 5

1 4 3 3

0 0 0 5

is not even in echelon form as $p_1 = 2$ and $p_2 = 1$ and hence $p_1 < p_2$ is not satisfied. Note, for the sake of giving an example, here $p_3 = 4$.

2 1 4 5

1 4 3 3

0 0 0 5

is also not in echelon form because of the same reason.

1 4 5 3

0 3 8 2

0 0 5 6

does satisy the conditions on p_is but $a_{2p_2} \neq 1$ and hence is not in echelon form.

1 4 0 3

0 1 0 2

0 0 1 6

is in echelon form but not in reduced echelon form as p_2-th column has extra non-zero entries (namely 4).

1 0 0 3

0 1 0 2

0 0 1 6

is in reduced echelon form.

The algorithm to reduce to echelon form.

1. First find the *first* non-zero column. Suppose the the column number is p_1 and the first non-zero entry is in the i-th row.
2. Switch the 1st row with the i-th row.
3. Sweep the p_1-th column
4. Repeat this for the smaller matrix spanning row 2 - last row and column $p_1 + 1$ to the last column.

First let us write a function to find the first column with a non-zero entry.

It will return a tuple (row, column) for the non-zero entry. If the matrix is zero. It will return (-1, -1).

In [14]:
```
def first_non_zero(matrix) :
    found_entry = False
    column_scanning = 0
    if matrix == [] or matrix == [[]] :
        print "The matrix is empty."
        row = -1
        col = -1
    else :
        while found_entry == False and column_scanning < len(matrix[0]) :
            # len(matrix[0]) = number of columns of the matrix.
            row_scanning = 0
            while found_entry == False and row_scanning < len(matrix) :
                #print "r, c : ", row_scanning, column_scanning
                if matrix[row_scanning][column_scanning] != 0 :
                    found_entry = True
                    row = row_scanning
                    col = column_scanning
                else :
                    row_scanning += 1
            column_scanning += 1
        if found_entry == False :
            row = -1
            col = -1
            print "The matrix is the zero matrix."
    return (row, col)
```

```
In [15]: # To test it out :
         print first_non_zero([[0, 0, 0, 0], [0, 2, 1, 0], [0, 0, 5, 1], [0, 5, 5, 6]])
         print first_non_zero([[0, 0, 0, 0], [0, 0, 0, 0]])
         print first_non_zero([[], []])
```

```
(1, 1)
The matrix is the zero matrix.
(-1, -1)
The matrix is the zero matrix.
(-1, -1)
```

Now we are ready to do Gaussian elimination

```
In [16]: def gauss_elim_prelim(matrix) :
             row = 0
             row_offset = 0
             col_offset = 0

             (r, c) = first_non_zero(matrix)
             if r == -1 :
                 print "Nothing to do: The matrix is zero."
             else :
                 matrix = elem1(matrix, 0, r)
                 matrix = sweep(matrix, (0, c))
                 print_matrix(matrix, "%6.2f")
```

```
In [17]: gauss_elim_prelim([[0,0,2,2],\
                      [1,4,4,4],\
                      [0,1,3,3]])
```

```
  1.00   4.00   4.00   4.00
  0.00   0.00   2.00   2.00
  0.00   1.00   3.00   3.00
```

```
In [18]: def gauss_elim(matrix) :
             row = 0
             row_offset = 0
             col_offset = 0

             while row < len(matrix) :
                 submatrix = []
                 for i in range(len(matrix) - row_offset) :
                     row_submatrix = []
                     for j in range(len(matrix[0]) - col_offset) :
                         row_submatrix.append(matrix[i + row_offset][j + col_offset])
                     submatrix.append(row_submatrix)
                 #print_matrix(submatrix, "%6.2f")
                 print row, row_offset, col_offset, submatrix, matrix
                 print "-"*20

                 (r, c) = first_non_zero(submatrix)
                 if r == -1 :
                     print "Nothing to do: The (sub)matrix is zero."
                 else :
                     r += row_offset
                     c += col_offset
                     matrix = elem1(matrix, row_offset, r)
                     matrix = sweep(matrix, (row_offset, c))
                     row_offset += 1
                     col_offset = c+1
                 row += 1
             return matrix
```

```
In [19]: read_matrix("files/B.txt")
         print_matrix(gauss_elim(B), "%6.2f")
         print_matrix(gauss_elim([[0,0,0,0], [0,1,1,0], [0,0,1,0]]), "%6.2f")
```

```
0 0 0 [[-0.6000000000000001, -0.40000000000000036, 0.0, -1.0], [0.2,
0.8, 1.0, 1.0], [2.6, -1.6, 0.0, 4.0]] [[-0.6000000000000001,
-0.40000000000000036, 0.0, -1.0], [0.2, 0.8, 1.0, 1.0], [2.6, -1.6,
0.0, 4.0]]
--------------------
1 1 1 [[0.6666666666666666, 1.0, 0.6666666666666667],
[-3.333333333333335, 0.0, -0.33333333333333304]] [[1.0,
0.6666666666666672, -0.0, 1.6666666666666665], [0.0,
0.6666666666666666, 1.0, 0.6666666666666667], [0.0,
-3.333333333333335, 0.0, -0.33333333333333304]]
--------------------
2 2 2 [[5.000000000000002, 3.0000000000000018]] [[1.0, 0.0,
-1.0000000000000009, 0.9999999999999993], [0.0, 1.0, 1.5, 1.0], [0.0,
0.0, 5.000000000000002, 3.0000000000000018]]
--------------------
  1.00   0.00   0.00   1.60
  0.00   1.00   0.00   0.10
  0.00   0.00   1.00   0.60
0 0 0 [[0, 0, 0, 0], [0, 1, 1, 0], [0, 0, 1, 0]] [[0, 0, 0, 0], [0, 1,
1, 0], [0, 0, 1, 0]]
--------------------
1 1 2 [[0.0, 0.0], [1.0, 0.0]] [[0.0, 1.0, 1.0, 0.0], [0.0, 0.0, 0.0,
0.0], [0.0, 0.0, 1.0, 0.0]]
--------------------
2 2 3 [[0.0]] [[0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0, 0.0,
0.0, 0.0]]
--------------------
The matrix is the zero matrix.
Nothing to do: The (sub)matrix is zero.
  0.00   1.00   0.00   0.00
  0.00   0.00   1.00   0.00
  0.00   0.00   0.00   0.00
```