03_03_gaussian_elimination

Unknown Author

March 11, 2014

Part I

Gaussian Elimination - I

We need the following functions which we wrote in the last class.

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]]
```

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
```

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
```

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 [6]:

```
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 [7]:

```
print_matrix(A, "%2.0f")
print "--------------------------"
print_matrix(elem2(A, 0, 4), "%2.0f")
print "-------------------------"
print_matrix(A, "%2.0f")
```

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

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 [8]:

```
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 [9]:
```
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
 2  1  2
 3  2  1
--------------------
 0  6  8
 2  1  2
 3  2  1
---------------------
 0  6  8
 2  1  2
 3  2  1
```

1 Sweeping a column

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 [10]:
```
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
```

2 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 [11]:
```
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 [12]:

```
# 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)
```

3 Digression : recursion

In python a function can call itself. It has some uses.

In math some things are defined recursively.

Example

Fibonacci sequences is defined to be sequence $a_1, a_2, \ldots$ where

- $a_1 = 1$,
- $a_2 = 1$,
- for an other $n \geq 3$, $a_n = a_{n-1} + a_{n-2}$.

As demonstration let us define a function `fib(n)` which *returns* the n-th entry of the Fibonacci sequence, first without recursion, second with recursion.

In [13]:

```
def fib_nonrec(n) :
    if n != int(n) :
        print "Please input an positive integer."
        retval = None
    elif n <= 0 :
        print "Please input a natural number."
        retval = None
    elif n == 1 or n == 2 :
        retval = 1
    else :
        i = 2
```

```
        fi = 1
        fi_1 = 1

        while i < n :
            i += 1
            fi_2 = fi_1
            fi_1 = fi
            fi = fi_1 + fi_2
        retval = fi
    return retval
```

In [14]:
```
for i in range(11):
    print fib_nonrec(i)
print fib_nonrec(4.5)
```

```
Please input a natural number.
None
1
1
2
3
5
8
13
21
34
55
Please input an positive integer.
None
```

In [15]:
```
def fib_rec(n) :
    if n != int(n) or n <= 0 :
        print "Please input a natural number."
        retval = None
    elif n == 1 or n == 2 :
        retval = 1
    else :
        retval = fib_rec(n-1) + fib_rec(n-2)
    return retval
```

In [16]:
```
for i in range(11) :
    print fib_rec(i)
print fib_rec(4.5)
```

```
Please input a natural number.
None
1
1
2
3
5
8
13
21
34
55
Please input a natural number.
None
```

Another example

One can define the factorial of a number to be fact(0) = 1, fact(n) = n * fact(n-1)

In [17]:
```
def my_fact(n) :
    if n != int(n) or n < 0 :
        print "Expected a non-negative integer."
        retval = None
    elif n == 0 :
        retval = 1
    else :
        retval = n * my_fact(n-1)
    return retval
```

In [18]:
```
print my_fact(100.0), my_fact(-100), my_fact(1.0/100), my_fact(100)
```

```
9.33262154439e+157 Expected a non-negative integer.
None Expected a non-negative integer.
None 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000000000
```

3.1 Now we are ready to do Gaussian elimination

We shall first do it using recursion. For that we need a function to find the first non-zero column, `c`, and the first row which has a non-zero element in column `c`.

In [19]:
```
def left_most_non_zero(M) :
    """Finds the left most non-zero entry in M. Returns (-1, -1) for zero and empty ma

    if M == [[]] or M == []:
        print "The matrix is empty."
        r = -1
        c = -1
    else :
        cell_found = False
        present_col = 0
        no_of_cols = len(M[0])
        no_of_rows = len(M)

        while present_col < no_of_cols and cell_found == False :
            present_row = 0
            while present_row < no_of_rows and cell_found == False :
                if M[present_row][present_col] != 0 :
```

```
                    cell_found = True
                    r = present_row
                    c = present_col
                present_row += 1
            present_col += 1

        if cell_found == False :
            r = -1
            c = -1
    return (r, c)
```

In [20]:
```
my_matrix = [[0, 0, 1], [0, 0, 2], [0, 1, 3]]
print left_most_non_zero(my_matrix)
```

```
(2, 1)
```

Here is the algorithm.

1. First find the first non-zero column, say c, and find a row which has a non-zero entry on that column. Let that row by r.
2. Interchange rows 0 and r. (elementary operation 1).
3. Sweep respect to pivot $(0, c)$.
4. Repeat the procedure on the submatrix whose first row is the elements of the 1st row from $c + 1$ till the end till the last row from $c + 1$ to the last column.

Thus *to use recursion* we shall write a function which given a matrix M and a coordinate (r, c) *return*s a submatrix as above. We also need a function which will help us to copy the submatrix at the correct position.

In [21]:
```
def extract_sub_matrix(M, t) :
    """M : matrix; t = (r, c) is the tuple such that every entry of the submatrix has
       coordinates (i, j) with i > r and j > c."""

    r = t[0]
    c = t[1]

    submat = []
    no_rows_M = len(M)
    if no_rows_M == 0 :
        print "The matrix is empty."
    else :
        no_cols_M = len(M[0])
        for i in range(r+1, no_rows_M) :
            current_row = []
            for j in range(c+1, no_cols_M) :
                current_row.append(M[i][j])
            submat.append(current_row)
    return submat
```

In [22]:
```
new_matrix=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
print_matrix(new_matrix, "%3.0f")
print
print_matrix(extract_sub_matrix(new_matrix, (-1,-1)), "%3d")
```

```
  1   2   3   4
  5   6   7   8
  9  10  11  12

  1   2   3   4
  5   6   7   8
  9  10  11  12
```

In [23]:
```
def copy_back_submatrix(M, S) :
    """Copies submatrix S into M to the lower right of t."""
    no_rows_M = len(M)
    no_rows_S = len(S)

    if no_rows_M == 0 :
        print "M is anyway empty."
        retM = []
    elif no_rows_S == 0 :
        print "Nothing to copy: S is empty"
        retM = M
    else :
        no_cols_M = len(M[0])
        no_cols_S = len(S[0])

        r = no_rows_M - no_rows_S
        c = no_cols_M - no_cols_S
        if r < 0 or c < 0 :
            print "Matrix S is too big."
            retM = S
        else :
            retM = []
            for i in range(no_rows_M) :
                present_row = []
                for j in range(no_cols_M) :
                    if i < r or j < c :
                        present_row.append(M[i][j])
                    else :
                        present_row.append(S[i-r][j-c])
                retM.append(present_row)
    return retM
```

In [24]:
```
print_matrix(new_matrix, "%3d")
print "-"*15
print_matrix(copy_back_submatrix(new_matrix, [[-2], [-3]]), "%3d")
```

```
  1   2   3   4
  5   6   7   8
  9  10  11  12
---------------
  1   2   3   4
  5   6   7  -2
  9  10  11  -3
```

In [25]:
```
def red_to_ech_form(M) :
    """Reduce matrix M to echelon form."""

    # First find the non-zero column.
    t = left_most_non_zero(M)
    r = t[0]
    c = t[1]

    if r == -1 or c == -1 :
        retM = M
    else :
        # Swap row r and row 0 and sweep
        retM = M
        retM = elem1(retM, r, 0)
        retM = sweep(retM, (0, c))

        Mprime = extract_sub_matrix(retM, (0, c))
        Mprimeech = red_to_ech_form(Mprime)
        retM = copy_back_submatrix(retM, Mprimeech)

    return retM
```

In [26]:
```
another_matrix = [[0,.5,1,0],[2,11,0,0],[1,0,0,9]]
print_matrix(another_matrix, "%10.5f")
print "-"*50
print_matrix(red_to_ech_form(another_matrix), "%10.5f")
```

```
   0.00000    0.50000    1.00000    0.00000
   2.00000   11.00000    0.00000    0.00000
   1.00000    0.00000    0.00000    9.00000
--------------------------------------------------
The matrix is empty.
Nothing to copy: S is empty
   1.00000    5.50000    0.00000    0.00000
   0.00000    1.00000    2.00000    0.00000
   0.00000    0.00000    1.00000    0.81818
```