ECE 4960 Spring 2017
Homework 4: Applying the Wilkinson principle to sparse matrix computation
Due end of day of 2/24

We will use the sparse matrix computation for illustration of the Wilkinson principle. Three common matrix operations will be implemented in the full matrix format AND in the row-compressed format in Part I, and two checksum procedures will be employed in Part II. The `rowPermute( )` function will switch the position of the *i*-th and *j*-th rows. The `rowScale( )` function will add the *i*-th row multiplied by a constant *a* to the *j*-th row. The `productAx( )` function will implement the product of matrix *A* and vector *x*. The return integer is reserved for the indicator of successful operations, e.x., 0 means no error or exception. If you want to improve the code readability, use local `define` statement.

(1) Row permute:
```
// Switch row[i] and row[j] for matrix A and vector x
int rowPermute(matrix* A, vec* x, int i, int j);
```

(2) Row scaling:
```
// Add a*row[i] to row[j] for matrix A and vector x
int rowScale(matrix* A, vec* x, int i, int j, double a);
```

(3) Vector product:
```
// Return the product of Ax = b
int productAx(matrix* A, vec* x, vec* b);
```

Implement these methods as a function of sparse matrix structure. For the full-matrix representation, feel free to use built-in utility functions, but you have to implement the sparse matrix functions in your own code with basic data structure. Notice that the row scaling operation can increase or decrease your non-zero elements. It is your choice how to handle the operation, but before exiting the row scaling function, your matrix needs to be in the correct format. These functions should be usable for any matrix and vector with an arbitrary rank n (< 50,000) and number of non-zero elements nNZ (< 10^6).

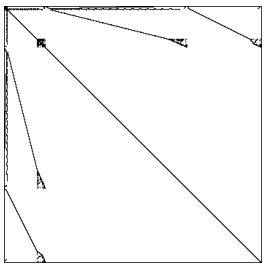
Part I: (Ground truth can be known) Use the matrix and vector below for validation.

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}; \; x = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

Write a test function to compare the resulting matrix and vector from the full matrix and from the sparse matrix using the second norm: $\sum_{i,j} \left(a_{ij,\,full} - a_{ij,sparse}\right)^2$ and $\sum_{i} \left(x_{i,\,full} - x_{i,sparse}\right)^2$

(1) Permutation of (1, 3) and then (1, 5).
(2) Test 3.0*row[1] + row [4] and -4.4*row[5] + row[2].
(3) Test $Ax = b$.

Part II: (Unknown ground truth) Load a large sparse matrix from the class blackboard site or from the matrix market (http://math.nist.gov/MatrixMarket/extreme.html) called memplus (rank: 17758; 126,150 nonzero entries) generated from the SPICE simulation of memory circuits. The structure plot of the sparse matrix is:



The file format has the first line as comments. The second line contains rowRank, columnRank and nNZ. The following lines are the matrix elements in the coordinate storage format using 1 as the starting index, unlike the 0 indexing in C/C++.

As this matrix cannot be handled by the full matrix in many computers, we cannot use the two data structures of full and sparse formats to perform the Wilkinson tests. Although theoretically you can compare the row and column compressed format, but we will use another type of checks to illustrate the Wilkinson principle. Here, we will use two calculation methods on the same data structure similar to the binary checksum. You have to use the SAME sparse matrix functions you have constructed in Part I.

(1) Perform permutation of (1, 3), (1, 5), (10, 3000), (5000, 10000), and (6, 15000).
(2) Perform 3.0*row[2] + row[4], permute (2, 5), and then perform -3.0*row[5] + row[4].
(3) Perform $Ax = b$, where x is a column vector with all its elements equal to 1.0. You can readily see that the signed sum of the elements in b is the signed sum of all elements of A.
(4) Compute $\left| \sum_{i=1}^{nNz} a_i - \sum_{j=1}^{n} b_j \right|$, which should be smaller than a set tolerance (say 10^{-7}).

Call a proper cpuTime( ) and memoryUsage functions on your platform to record the CPU time and memory usage of your operation in Part II.

Use Git to store a version of your final program. You will need to use these functions (maybe with some evolution if necessary), but you should maintain the regression testing. Notice that you can use a debugger to reasonably check Part I by walking through the computation and compare with estimated calculation, but this would not be possible for this large matrix if the checksum failed in Part II. However, CPU time and memory usage can offer additional clues in regression testing.

You can work individually or in a two-person group. Submit a zipped file by email with your netID in the filename such as: netIDnetID4960Hw4.zip. The zip file should contain a short one-page report (brief testing results of Part I and II) and all source files. Do not include the matrix file or binaries executables, so your zip file should have a file size less than 100kB.