Question 2

Sparce matrix and vector products

December 2016

Objective : Accept matrix indices and values from an input file ,store them using linked lists and pointers and use it to compute both Ax and x^TA.

Storing A

Two linked lists have been used here : sa1 and sa2.
Note that only non-zero elements have been stored even if input file consists of zero elements.

```
/*****************************START*******************************/
typedef struct S1   //for storing the non-zero diagonal
   elements
{
    float value;
    int i;
    struct S1 *p;

}SA1;

typedef struct S2   //for storing the non-zero non-diagonal
   elements
{
    float value;
    int i,j;
    struct S2 *p;

}SA2;


struct S1 *sa1=NULL;         //linked list of non-zero
   diagonal elements in order
struct S2 *sa2=NULL;       //linked list of non-zero
   non-diagonal elements in order, from starting row and
```

```
    starting column and traverse row-wise. ie, for a
    particular i, j from 0 to N-1

/*********************************END*******************************/
```

sa1 holds the non-zero diagonal elements. Each time a non-zero diagonal element is read from the input file, it is inserted into the linked list such that the linked list is sorted in ascending order with respect to row number i. This is achieved by means of the function :

```
void createlist1(int r,float val)    //create the linked
    list for non-zero diagonal elements
{
  .....
}
```

The new element details are stored in temp and sa1 points to the first element of the list. The pointer top is used for navigating through the list and inserting the new element in the correct position in the list.

```
/**********************************START*****************************/
        if(sa1==NULL)       //if list is empty
        {
            sa1=temp;   //temp is the first element in the
                list and hence sa1 points to temp
        }
        else if(sa1 != NULL )       //if list is not empty
        {
            top=sa1;        //to navigate through the list

            if((temp->i)<(top->i))      //if the new item if
                to be added in front of sa1(first element in
                the linked list with lowest row index up till
                now)
            {
             temp->p=top;    //temp points to top
```

```
            sa1 = temp;     //sa1 points to temp as the
                element with the lowest row index
        }

        else if((temp->i)>(top->i))     //if the new
            item's index is greater than the row-index of
            top
        {
            while(flag==1)  //till new element is
                inserted
            {
                if(top->p==NULL)    //if temp is to be
                    appended to the end of the list
                {
                    top->p=temp;    //last element of
                        sa1 points to temp
                    flag=0;         //denotes that new
                        element has been inserted
                }
                else if((temp->i)>(top->i) &&
                    (temp->i)<(top->p->i))    //if the
                    element is to be inerted in between
                    the list
                {
                    temp->p=top->p;     //temp is linked
                        to the element to which top is
                        linked to
                    top->p=temp;        //temp is linked
                        to top
                    flag=0;         //denotes that new
                        element has been inserted
                }
                top=top->p;     //incrementing top to
                    next element in the linked list
            }
        }

    }
/************************************END*********************************/
```

sa2 holds the non-zero non-diagonal elements. Each time a non-zero non-diagonal element is read from the input file, it is inserted into the linked list such that the linked list is sorted in ascending order with respect the quantity : i*N+j , where i is row number , j is column number of the element and N is the dimension of matrix A. This is achieved by means of the function :

```
void createlist2(int r, int c, float val)    //create the
    linked list for non-zero non-diagonal elements
{
  .....
}
```

The new element details are stored in Temp and sa2 points to the first element of the list. The pointer Top is used for navigating through the list and inserting the new element in the correct position in the list.

```
/***********************************START**********************************/

        if(sa2==NULL)        //if list is empty
         {
             sa2=Temp;   //Temp is the first element in the
                 list and hence sa2 points to Temp
         }
         else if(sa2 != NULL)        //if list is not empty
         {
             Top=sa2;        //to navigate through the list

             index1=(Temp->i)*N + (Temp->j);      //index
                 value for Temp
             index2=(Top->i)*N + (Top->j);        //index
                 value for Top

             if(index1<index2)         //if the new item if to
                 be added in front of sa2(first element in the
                 linked list with lowest index up till now)
             {
                 Temp->p=Top;         //Temp points to Top
                 sa2 = Temp;          //sa2 points to Temp as
                     the element with the lowest index
             }

             else if(index1>index2)       //if the new item's
                 index is greater than the index of Top
             {
                 while(flag==1)       //till new element is
```

```
                    inserted
                {
                    if(Top->p==NULL)    //if Temp is to be
                        appended to the end of the list
                    {
                        Top->p=Temp;    //last element of
                            sa2 points to Temp
                        flag=0;         //denotes that new
                            element has been inserted
                    }
                    else if(index1>index2 &&
                        index1<((Top->p->i)*N + (Top->p->j)))
                        //if the element is to be inerted in
                        between the list
                    {
                        Temp->p=Top->p;     //Temp is linked
                            to the element to which Top is
                            linked to
                        Top->p=Temp;        //Temp is linked
                            to Top
                        flag=0;             //denotes that
                            new element has been inserted
                    }
                    Top=Top->p;             //incrementing
                        top to next element in the linked list
                    index2=(Top->i)*N + (Top->j);
                        //re-initialisisng the value of index2
                }
            }
        }
/********************************END*******************************/
```

Compute Ax and x^TA

It has been carried out in the function :

```
int Ax_xTA(int x[])     //for calculating the required
    matrix products
{
  ......
}
```

The algorithm makes use of the face that :

1. b=Ax :
   For diagonal element : b[i]=A[i][i]*x[i]
   For non-diagonal element :b[i]=A[i][j]*x[j]
   top and Top are pointers used to traverse through the lists sa1 and sa2 respectively.

   ```
   /*********************************START****************************/

       while(top!=NULL)    //condition for reaching end of
           the linked list
       {
           b[top->i]+=(top->value)*x[top->i];  //diagonal
               elements multiplied by the corresponding
               element in vector x and stored in the
               corresponding location in b
           top=top->p;             //incrementing top to
               next element in the linked list
       }

       while(Top!=NULL)    //condition for reaching end of
           the linked list
       {
           b[Top->i]+=(Top->value)*x[Top->j]; // the
               element pointed to by Top is multiplied by
               the corresponding element in x and added to
               corresponding element in b
             Top=Top->p; //incrementing top to next
                 element in the linked list
       }
   /***********************************END****************************/
   ```

2. c=x^TA :
   $(A^T x)^T = x^T A$
   Since c[] is a 1-D array, the transpose doesn't matter.
   For diagonal element : c[i]=A[i][i]*x[i]

For non-diagonal element :c[j]=A[i][j]*x[i]
top and Top are pointers used to traverse through the lists sa1 and sa2 respectively.

```
/********************************START***************************/

    while(top!=NULL)    //condition for reaching end of
        the linked list
    {
        c[top->i]=b[top->i];              //diagonal
            elements multiplied by the corresponding
            element in vector x and stored in the
            corresponding location in c
        top=top->p;              //incrementing top to
            next element in the linked list
    }

    while(Top!=NULL)    //condition for reaching end of
        the linked list
    {
        c[Top->j]+=(Top->value)*x[Top->i]; // the
            element pointed to by Top is multiplied by
            the corresponding element in x and added to
            corresponding element in c
        Top=Top->p; //incrementing top to next element
            in the linked list
    }

/**********************************END***************************/
```

The above two codes are edited versions of the ones used in the original program. They have been edited to increase readability and understanding of the code.

Note :
The above stated multiplications could have been achieved without two different linked lists sa1 and sa2 and without any sorting. That is, we could have achieved the above vectors b[] and c[] by inserting the read elements directly into a linked list of structure type SA2 and using the algorithm for

non-diagonal element multiplication. That would have made the code even more simpler. But the advantage of having 2 different linked lists comes into play when we want to calculate trace of the matrix A or if we want to access just the diagonal elements say, to find the determinant in case of a triangular matrix A. It also comes in handy while determining diagonisability of the matrix. The sorting comes in handy as having an order is always a good thing. We will know what index to expect for next. It makes row-wise access easier too(Suppose while calculating the average no. of elements in a row).

Comparison of Algorithms

This algorithm would be more efficient than the one given in Numerical Recipes in C.

- The book algorithm converts a matrix A into sparse matrix and hence a lot of storage space is wasted while storing matrix A and then converting it to a sparse matrix. The above given algorithm directly accepts disordered matrix elements and puts them into proper place , just that only non-zero elements are stored, and that too as in the form of a matrix expanded into a 1-D array.

- The algorithm given in the book stores all the diagonal elements, regardless of whether they are non-zero or not, whereas the above specified algorithm stores only non-zero relevant elements.

- The row-indexed sparse storage mode given in the book requires storage of about two times the number of matrix elements, ie, number of 2* non-diagonal non-zero elements + 2*N(dimension of A) +2, whereas the above specified method requires 3* number of non-zero non-diagonal elements + 2* number of non-zero diagonal elements + number of non-zero elements(links or pointers). Depending on the type of matrix, one of them would be more storage efficient.

- In Numerical Recipes, they dont store the indices for the diagonal elements nor the column indices for more than 2 elements in the same row, though all column indices of non-diagonal non-zero element are stored. In the above approach, both row and column indices of non-zero non-diagonal elements and row-indices of non-zero diagonal elements have to be stored.

- Access of any diagonal element in Numerical Recipes is O(1) whereas in the above approach, we have to traverse the list inorder to locate the diagonal element.

- The time complexity for storing the sparse matrix
  In Numerical Recipes : O(n^2) where n= dimension of A
  In the above algorithm : O(n) almost along with the overhead of function calls and if -else conditions.

- Time complexity for calculating matrix product
  In Numerical Recipes : O(nm) where n = matrix dimension and m= no. of non-zero non-diagonal elements
  In the above algorithm : O(p) where p = no. of non-zero elements in matrix A