AMS 562 Midterm Project
Sparse Matrix Storage Schemes

Due: Monday, 11/05, 11:59 pm

Background:

In scientific computing, matrices are commonly used, e.g. solving linear systems. As we know, matrices have 2D structure with shape n by m, where n is the number of rows, and m is that for the column. In general, matrices require nm storage space, and this is called *dense* matrices. However, most problems, in practice, are very large and the resulting linear systems can be huge. Therefore, using dense storage can be inefficient in both memory usage and computation cost. Most critically, dense storage can be infeasible in practice, e.g. considering a matrix of size $100,000$ by $100,000$, if we use double precision floating number, then we need approximately 80G memory, which is practically impossible. Luckily, most of the matrices in practice are *sparse*, i.e. majority of the entries are zero. Since storing the zero entries is useless, people have come up different ideas to represent sparse matrices efficiently.

Coordinate Format

Intuitively, a *coordinate* (COO, a.k.a. *ijv*) format can be used, i.e. storing the row and column index pair and its corresponding entry.

$$\begin{vmatrix} 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 3 & 0 & 2 & 2 \\ 1 & 0 & 0 & 1 \end{vmatrix}$$

With COO format, the above matrix can be stored as:

$$\begin{aligned} \text{i} &: \begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \text{j} &: \begin{bmatrix} 0 & 2 & 1 & 0 & 2 & 3 & 0 & 3 \end{bmatrix} \\ \text{v} &: \begin{bmatrix} 1 & 2 & 2 & 3 & 2 & 2 & 1 & 1 \end{bmatrix} \end{aligned}$$

There are many limitations with COO format, one of them is that it's not efficient to query entries. However, COO is useful as an intermediate sparse representation.

In C++, we can use the following structure to represent a COO **square** matrix.

```
struct COOMatrix {
        double *v;                ///< value data array
        int *i;                   ///< row indices
        int *j;                   ///< column indices
        int n;                    ///< size of the square matrix
        int nnz;                  ///< number of total non-zeros
};
```

Compressed Sparse Row Format

Compressed sparse row (CSR, a.k.a. *compressed row storage (CRS), Yale*) format is one of the earliest ideas yet still popular. Similar to COO, CSR format also uses three arrays to represent sparse matrices, namely `indptr`, `indices`, and `values`. The key difference between COO and CSR is that the latter compresses the row indices thus the clustering of the non-zeros of each row is guaranteed. For instance, the matrix used in the previous section can be represented with CSR as the following:

$$
\begin{aligned}
\text{indptr}: & \begin{bmatrix}0 & 2 & 3 & 6 & 8\end{bmatrix} \\
\text{indices}: & \begin{bmatrix}0 & 2 & 1 & 0 & 2 & 3 & 0 & 3\end{bmatrix} \\
\text{values}: & \begin{bmatrix}1 & 2 & 2 & 3 & 2 & 2 & 1 & 1\end{bmatrix}
\end{aligned}
$$

Some remarks are:

1. Given a matrix with n rows, `indptr` is of size $n+1$ and its first entry is zero.

2. The last entry of `indptr`, i.e. `indptr[n]`, is the total number of non-zeros.

3. The i-th entry of `indptr` is `indices` and `values`'s index that points to the first non-zero value of i-th row.

4. The difference between i and $i+1$ entries of `indptr` is the number of non-zeros in row i, e.g. $\text{indptr}_3 - \text{indptr}_2 = 3$, and the third row starts at indptr_2, which is 3 (4-th entry), so the third row's values are $\begin{bmatrix}3 & 2 & 2\end{bmatrix}$ and their corresponding column indices are $\begin{bmatrix}0 & 2 & 3\end{bmatrix}$.

With these pieces of information, we can create the following C++ structure to represent sparse **square** matrices with CSR format.

```
struct CSRMatrix {
        double *value;          ///< value data array
        int *indices;           ///< column indices array
        int *indptr;            ///< row pointer array
        int n;                  ///< size of the square matrix
};
```

Matrix Vector Multiplication

Matrix vector multiplication is a useful operation in practice and builds the basis of many algorithms, e.g. iterative linear solvers. Essentially, given a matrix $\boldsymbol{A}$ and a **dense** vector $\boldsymbol{x}$, we want to find the dot product $\boldsymbol{y}$ between them, i.e. $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$. The pseudo code reads:

```
Algorithm: Matrix-Vector Multiplication (MV)
Inputs: square matrix $\boldsymbol{A}$ with size $n$, vector $\boldsymbol{x}$.
Output: vector $\boldsymbol{y}$, s.t. $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

$\boldsymbol{y} = \mathbf{0}$ // initial output to zeros
for $i \in [0, n)$, do
    for $j \in [0, n)$, do
        $y_i \leftarrow y_i + A_{ij}x_j$
    end for
end for
```

The algorithm is generic, for sparse matrices, we just need to make sure $A_{ij} \neq 0$ since we don't store the zero entries.

```
Algorithm: Sparse Matrix-Vector Multiplication (SMV)
Inputs: sparse square matrix $\boldsymbol{A}$ with size $n$, vector $\boldsymbol{x}$.
Output: vector $\boldsymbol{y}$, s.t. $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

$\boldsymbol{y} = \mathbf{0}$ // initial output to zeros
for $i \in [0, n)$, do
    for $j \in [0, n)$, do
        if $A_{ij} \neq 0$, then
            $y_i \leftarrow y_i + A_{ij}x_j$
        end if
    end for
end for
```

One of the core tasks of this project is to write routines computing the dot product of both COO and CSR storage schemes.

Extracting Diagonal Entries

Unlike dense storage, extracting certain entries becomes a nontrivial task with sparse matrices. In general, a *searching* algorithm is needed. For instance, given a CSR matrix $\boldsymbol{A}$, in order to find its i-th row and j-th column, we need first to locate at its i-th row (how?), and then search for j in the corresponding `indices` sub-list.

A matrix's diagonal entries are those with same row and column indices. In this project, you can assume that the diagonal entries always exist. The pseudo code of such operation is:

```
Algorithm: Extracting diagonal entries
Inputs: square matrix $\boldsymbol{A}$ with size $n$
Output: vector diag, which stores the diagonal entries of $\boldsymbol{A}$.

for $i \in [0, n)$, do
    $\text{diag}_i = A_{ii}$
end for
```

Finding A_{ii}s for COO format is straightforward, i.e. you loop through all non-zero pairs and extract out the ones whose i is equal to j. For CSR format, the complexity goes to the local search given a specific row. Intuitively, we can perform a *linear search* algorithm:

```
Algorithm: Linear search
Inputs: array $\boldsymbol{x}$ and target value $v$.
Output: $i$, which is the index of $\boldsymbol{x}$ where $x_i = v$.

for $i \in [0, n)$, do
    if $x_i$ is equal to $v$, then
        return $i$
    end if
end for
return $n$ // find nothing
```

For CSR format, you can assume the column indices of each row are sorted, so that applying binary search algorithm is feasible. **Notice that implementing binary search is optional.**

Your Tasks

Starting with the skeleton project, you need to write the initialization and finalization routines for both COO and CSR matrices (10 points). Then, finish implementing the construction routines for the two storage schemes, i.e. for COO, given triplets (i, j, v) assign them to the corresponding arrays; for CSR, given a row and its corresponding column indices and entry values, build the three arrays, i.e. `indptr`, `indices`, and `values`, accordingly (20 points). Next step is to implement the matrix-vector multiplication routines for them (40 points). Finally, implement functions to extract diagonal entries for both COO and CSR (30 points). **In addition, if you use binary search instead of linear search for extracting CSR diagonals, you will gain an extra of 15 points.** Total points are 100+15.

Some Assumptions

1. The column indices of each row in CSR are sorted in ascending order.
2. For CSR construction, you can assume the order is sequential, i.e. row i is constructed right after $i - 1$ is done. This eases the construction process of `indptr` (why?).

Hint

Please refer to the `README.md` description in the skeleton project.