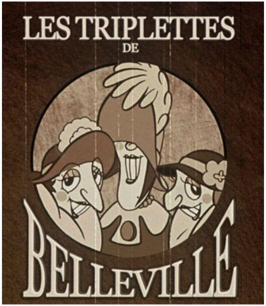
CS319: Scientific Computing

Week 11: Sparse Matrices and templates

Dr Niall Madden

9am and **4pm**, 20 March, 2024



Slides and examples:
https://www.niallmadden.ie/2324-CS319

Outline

1. News and Updates
2. Overview
3. Sparse Matrices
   - Triplet
4. Coding `triplet`
5. CCS

4pm

6. Templates
   - Function Templates
7. The Standard Template Library
   - Containers
   - Iterators

4pm

8. `sets` and `multisets`
9. ~~`vector`~~
   - ~~Other `vector` methods~~
   - ~~Range-based for loops~~
10. ~~Algorithm~~

News and Updates

- Lab 6: grades will be posted soon (honest!).
- Project proposals. Have now all been graded. Talk to me if you need further feedback.
- Presentations have been scheduled (switch to `https://www.niallmadden.ie/2324-CS319/2324-CS319-Projects.pdf`)

Overview

...

Sparse Matrices

Last week we designed a class for representing a matrix. Although we didn't discuss it at the time, the matrices represented are called "**dense**" or "**full**".

Today we want to see how to store **SPARSE MATRICES**: these are matrices that have so many zeros that it is worth our while exploiting the fact.

Sparse Matrices

There are numerous examples of sparse matrices. For example, they occur frequently when solving differential equations numerically.

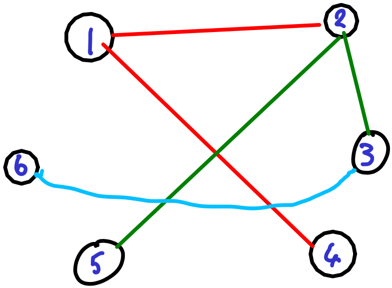
But perhaps the most obvious example is when we use matrices to represents graphs and **networks**.

Most real world networks have far more nodes/vertices than they do connections/edges between those nodes.

In a computational setting, most graphs/~~matrices~~ are represented as a matrix, such as the **adjacency matrix**.

- If the graph has N vertices, the matrix, A, has N rows and columns: each corresponds to vertex.
- If (i, j) is an edge in the graph, then $a_{ij} = 1$. Otherwise, $a_{ij} = 0$.

Example:



$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

Note: NNZ is 10

So Triplet format would need $3 \times 10 = 30$ values to be stored. "Full" format would need 36

Sparse Matrices

Compared to the over-all number of entries in the matrix, the **number of non-zeros** (**NNZs**) is relatively small. So it does not make sense to store them all. Instead, one uses one of the following formats:

- **Triplet** (which we'll look at presently),
- **Compressed Column Storage** (CCS)

And the following formats for very specialised matrices, which we won't study in CS319:

- **Block Compressed Row/Column Storage**
- **Compressed Diagonal Storage**
- **Skyline**

Important: NNZ = "Number of non-zeros"

Although the representation and manipulation of sparse matrices is an major topic in Scientific Computing, there isn't a universally agreed definition of an (abstract) *sparse matrix*.

This is because, when coding, we should ask the question: **"When is it worth the effort to store a matrix in a sparse format, rather than in standard (dense) format?"**

The answer is often context-dependent. But roughly, use a sparse format when

- The memory required by the sparse format is less then the "dense" (or "full") one;
- The expense of updating the sparse format is not excessive;
- Computing a `MatVec` is faster for a sparse matrix.

The basic idea for triplet form is: to store a **sparse** matrix with `NNZ` non-zeros we ...

unsigned

- define `integer` arrays `I[NNZ]` and `J[NNZ]`,
- a `double` array `X[NNZ]`.
- Then entry a_{ij} is stored as `I[k]=i`, `J[k]=j`, `X[k]=`a_{ij}, for some k.

Example: write down the triplet form of the following matrix:

$$\begin{array}{cc} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 1 & 0 & 11 & 0 \\ 1 & 0 & 2 & 0 \\ 9 & 19 & 0 & 29 \\ 0 & 0 & 0 & 5 \end{pmatrix} \\ & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \end{array}$$

$I = [0, 0, 1, 1, 2, 2, 2, 3]$

$J = [0, 2, 0, 2, 0, 1, 3, 3]$

$X = [1, 11, 1, 2, 9, 19, 29, 5]$

$NNZ = 8$

The basic idea for triplet form is: to store a **sparse** matrix with `NNZ` non-zeros we ...

- define `integer` arrays `I[NNZ]` and `J[NNZ]`,
- a `double` array `X[NNZ]`.
- Then entry a_{ij} is stored as `I[k]=i`, `J[k]=j`, `X[k]=`a_{ij}, for some k.

~~**Example:** write down the triplet form of the following matrix:~~

$$\begin{pmatrix} 1 & 0 & 11 & 0 \\ 1 & 0 & 2 & 0 \\ 9 & 19 & 0 & 29 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

Example: Suppose A is a 1000 × 1000 matrix, and every row has 10 non-zeros.

Then "Full" format would store 1,000,000 values, and Triplet would store 30,000.

Our next goal is implement a triplet matrix as a `class`. The main tasks are:

- Decide what private data elements are needed.
- Decide what public methods are needed.
- Implement a matrix-vector multiplication algorithm.

Discussion... What does our class need?

Data (Private)	Methods (Public).
size (=N). NNZ (current) NNZ_MAX } unsigned int	Constructors + Destructors getij() setij() etc.
X, I, J } Arrays.	Geti, Getj, Getx } values of k where a_{ij} is stored. Also: *, =

Triplet.h

```
// Triplet.h:  For 2324-CS319 Week 11
// Author: Niall Madden

#ifndef _TRIPLET_H_INCLUDED
#define _TRIPLET_H_INCLUDED

#include "Vector10.h"  from last week.
#include "Matrix11.h"  // includes zero() method.
```

Coding triplet

Triplet.h

```
class Triplet {
  friend Triplet full2Triplet(Matrix &F, unsigned NNZ_MAX);
private:
  unsigned *I, *J;   -> row & col index
  double *X;   ->  values
  unsigned N;  -> number of rows & cols
  unsigned NNZ;
  unsigned NNZ_MAX;

public:
  Triplet (unsigned N, unsigned nnz_max); // Constructor
  Triplet (const Triplet &t); // Copy constructor
  ~Triplet(void);

  Triplet &operator=(const Triplet &B); // overload assignment
```

Triplet.h

```
  unsigned size(void) {return (N);};
  int where(unsigned i, unsigned j); // negative return on error
  unsigned nnz(void) {return (NNZ);};
  unsigned nnz_max(void) {return (NNZ_MAX);};

  double getij (unsigned i, unsigned j);
  void setij (unsigned i, unsigned j, double x);

  unsigned getI (unsigned k) { return I[k];};
  unsigned getJ (unsigned k) { return J[k];};
  double getX (unsigned k) { return X[k];};

  Vector operator*(Vector u);
  void print(void);
};
#endif
```

If k = A.where(i,j) then
A.I[k] = i, A.J[k] = j and A.X[k] = a_{ij}