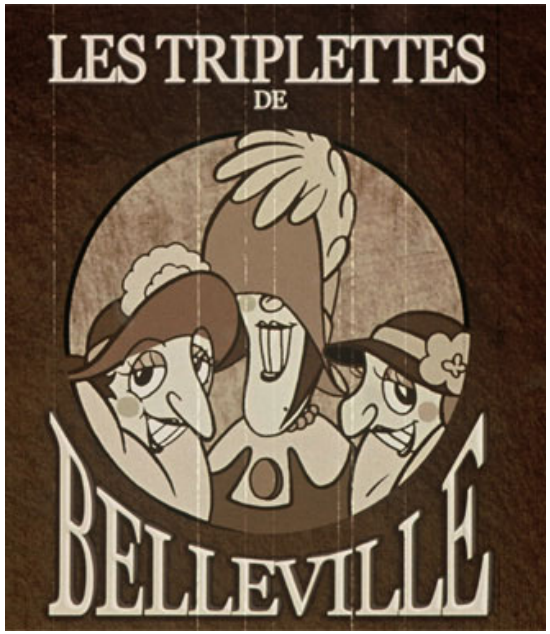
CS319: Scientific Computing

Week 11: Sparse Matrices and templates

Dr Niall Madden

26 + 28 March, 2025



Slides and examples:
https://www.niallmadden.ie/2425-CS319

0. Outline

1. News and Updates
2. Sparse Matrices
   - Triplet
3. Coding `triplet`
4. Compress Sparse Row
5. Templates
   - Function Templates
6. The Standard Template Library
   - Containers
   - Iterators
7. `sets` and `multisets`
8. `vector`
   - Other `vector` methods
   - Range-based for loops
9. Algorithm

1. News and Updates

- Grades for Lab 4, Class Test, and Lab 6 will be posted soon (honest!).
- Presentations have been scheduled (switch to https://www.niallmadden.ie/2425-CS319/2425-CS319-Projects.pdf)

2. 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.

2. 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/networks 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:

2. 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 Sparse Row** storage (CSR) (after triplet)
- **Compressed Sparse Column** storage (CSC)

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

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

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.

For **triplet** form we store a **sparse** matrix with `NNZ` non-zeros in three arraysL

- an `unsigned integer` arrays `I[NNZ]`, with row indices
- an `unsigned integer` arrays `J[NNZ]`,, with column indices.
- 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.

This is also known as **Coordinate list (COO)**

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 square $N \times N$ matrix, with $N = 1000$, three non-zero entries per row, and all data stored as `doubles`.

(a) How many values need to be stored if the A is stored in the usual "dense" format? How much memory is required?

(b) How many values need to be stored if the A is stored in **triplet** format? How much memory is required?

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...

3. Coding triplet

Triplet.h

```
1  // Triplet.h:  For 2425-CS319 Week 11
   // Author: Niall Madden
3
   #ifndef _TRIPLET_H_INCLUDED
5  #define _TRIPLET_H_INCLUDED

7  #include "Vector10.h"
   #include "Matrix11.h"
```

3. Coding triplet

Triplet.h

```
10 class Triplet {
     friend Triplet full2Triplet(Matrix &F, unsigned int NNZ_MAX);
12 private:
     unsigned int *I, *J;
14   double *X;
     unsigned int N;
16   unsigned int NNZ;
     unsigned int NNZ_MAX;

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

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

3. Coding triplet

Triplet.h

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

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

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

38    Vector operator*(Vector u);
      void print(void);
40  };
    #endif
```

3. Coding triplet

Triplet.cpp

```
1  // Triplet.cpp  for 2425-CS319 Week 11
   //   What: Methods for the Triplet class
3  // Author: Niall Madden
   #include <iostream>
5  #include <iomanip>
   #include "Vector10.h"
7  #include "Matrix11.h"
   #include "Triplet.h"
```

3. Coding triplet

Triplet.cpp (Constructor)

```
10  // Standard constructor.
    Triplet::Triplet (unsigned int N, unsigned int nnz_max) {
12    this->N = N;
      this->NNZ_MAX = nnz_max;
14    this->NNZ = 0;

16    X = new double [nnz_max];
      I = new unsigned int [nnz_max];
18    J = new unsigned int [nnz_max];
      for (unsigned int k=0; k<nnz_max; k++)  {
20      I[k]=-1;
        J[k]=-1;
22      X[k]=(double)NULL;
      }
24  }
```

3. Coding triplet

When using a Triplet object to represent a matrix, T, we often need to find where in the array X, the value of $t_{i,j}$ is stored. That is done by the following function.

Triplet.cpp (where)

```
   int Triplet::where(unsigned int i, unsigned int j)
56 {
     unsigned int k=0;
58   do {
       if ( (I[k]==i) && (J[k]==j) )
60       return(k);
       k++;
62   } while (k<NNZ);
     return(-1);
64 }
```

3. Coding triplet

Triplet.cpp (setij)

```
   void Triplet::setij (unsigned int i, unsigned int j, double x)
68 {
     if (i>N-1)
70     std::cerr << "Triplet::setij(): i Index out of bounds." << std
     else if (j>N-1)
72     std::cerr << "Triplet::setij(): j Index out of bounds." << std
     else if (NNZ > NNZ_MAX-1)
74     std::cerr << "Triplet::setij(): Matrix full." << std::endl;
     else
76   {
       int k=where(i,j);
78     if (k == -1)
       {
80       I[NNZ]=i;
         J[NNZ]=j;
82       X[NNZ]=x;
         NNZ++;
84     }
       else
86       X[k]=x;
     }
88 }
```

3. Coding triplet

Triplet.cpp (operator *)

```
Vector Triplet::operator*(Vector u)
{
  Vector v(N); // v = A*u, where A is the implicitly passed Triplet
  v.zero();
  if (N != u.size())
    std::cerr << "Error: Triplet::operator* - dimension mismatch"
              << std::endl;
  else
    for (unsigned int k=0; k<NNZ; k++)
      v.seti(I[k], v.geti(I[k]) + X[k]*u.geti(J[k]));
  return(v);
}
```

3. Coding `triplet`

To demonstrate the use of the `Triplet` class, I've included a program called `00triplet_example.cpp` which shows how to use the Jacobi method to solve a linear system where the matrix is stored in triplet format.

It also provides a (better) way of timing code, and gives the speed-up achieved for given parameters.

You can also run that programme directly on `online-cpp.com`: `https://www.online-cpp.com/eyPg6tLzEw`

4. Compress Sparse Row

If we know that the entries in our matrix are stored in order, then it is possible to store the matrix more efficiently that in Triplet format. One way of doing this is to use **CSR**: **Compressed Sparse Row**, also known as *Yale Format*.

The matrix is stored in 3 vectors:

- a `double` array, x of length `nnz` ("number of nonzero entries") storing the non-zero entries matrix, in column-wise order.
- an `int` array, c of length `nnz` storing **column** index of the entries. That is, $x[k]$ would be found in column $c[k]$ of the full matrix.
- an `int` array, r of length $N + 1$, where $r[i]$ stores that **starting point** of row i as it appears in the arrays x and c, and $r[N] = nnz$.

Note: this format is used by default in Python by the *`scipy.sparse`* module (also *`networkx`*, *`scikit-learn`*, ...).

4. Compress Sparse Row

Example

Show how the matrix below would be stored in CSR

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 5 & -1 & 0 \\ 0 & -2 & 4 & 0 \\ -3 & 0 & 0 & 4 \end{pmatrix}$$

The process of multiplying a matrix (in CSR) by a vector is rather simple:

```
int index=0;
for (int row=0; row<N; row++)
   for (i=r[row]; i<r[row+1]; i++)
   {
     j=c[index];
     v[j] += x[index]*b[i];
     index++;
   }
```

I don't provide code for implementing a CSR class here: that is an exercise.

5. Templates

We've written several of our own functions and classes. For most of these, they depend on data of a certain **type**. For example, in Week 5, we wrote some functions called `Sort()` for sorting `int`s.

Suppose we wanted to sort arrays of a different type, e.g., `strings`. We could take our old `Sort(int *list, int length)` function from `Week05/04Sort.cpp` and rewrite it as `strings`: `Sort(string *list, int length)`
The code the "new" function would be almost identical: we'd just replace several instances of the datatype `int` with `string`.

To avoid this repetition, and to allow us to write functions or class **generic** datatypes, C++ provides `template`s.

Today we first consider **function templates**.

To perform essentially identical operations for different types of data compactly, use function templates.

- Syntax: `template <typename T>` immediately precedes the function definition. It means that we'll be referring to the generic datatype as `T` in the function definition.
- Write a single function template definition. In it, the generic datatype is named `T`.
- Based on the argument types provided in calls to the function, the compiler automatically creates functions to handle each type of call appropriately.

In the example below, which you can find in detail in `01FunctionTemplate.cpp`, we'll write three functions:

a. `PrintList(MyType *x, int n)`
b. `void Sort(MyType &a, MyType &b)`
c. `void Sort(MyType *x, int n)`

The function prototypes:

01FunctionTemplate.cpp

```
template <typename MyType>
void PrintList(MyType *x, unsigned int n);

template <typename MyType>
void Sort(MyType &a, MyType &b);

template <typename MyType>
void Sort(MyType *list, unsigned int length);
```

The (bubble) Sort functions:

01FunctionTemplate.cpp

```
template <typename MyType>
void Sort(MyType &a, MyType &b) {
  if (a>b)
  {
    MyType tmp=a;
    a=b;
    b=tmp;
  }
}
```

```
template <typename MyType>
void Sort(MyType *x, unsigned int n) {
  for (int i=n-1; i>1; i--)
    for (int k=0; k<i; k++)
      Sort(x[k], x[k+1]);
}
```

01FunctionTemplate.cpp

```
22 int main(void )
   {
24   int Numbers[8];
     char Letters[8];

     for (int i=0; i<8; i++)
28     Numbers[i]=rand()%40;

30   for (int i=0; i<8; i++)
       Letters[i]='A'+rand()%26;
```

01FunctionTemplate.cpp

```
    std::cout << "Before sorting:" << std::endl;
34  std::cout << "Numbers: ";  PrintList(Numbers, 8);
    std::cout << "Letters: ";  PrintList(Letters, 8);

    Sort(Numbers, 8);
38  Sort(Letters, 8);

40  std::cout << "After sorting: " << std::endl;
    std::cout << "Numbers: ";  PrintList(Numbers, 8);
42  std::cout << "Letters: ";  PrintList(Letters, 8);
```

Typical output

```
Before sorting:
Numbers:   23    6   17   35   33   15   26   12
Letters:    B    H    C    D    A    R    Z    O
After sorting:
Numbers:    6   12   15   17   23   26   33   35
Letters:    A    B    C    D    H    O    R    Z
```

6. The Standard Template Library

During the semester, we've focused on designing classes that can be used to solve problems. These included classes: `Stack`, `Vector` and `Matrix`.

However, most of you worked out that, to some extent, these are already supported in C++. The motivations for reinventing them included

- our implementation is simple to use;
- we learned important aspects of C++/OOP;
- we needed to achieve specific tasks efficiently: this is particularly true of our design of sparse matrix classes.

Now we'll look at how to use the built-in implementation that comes with the C++ **Standard Template Library (STL)**.

The **STL** provides

(1) **Containers:** ways of collecting/storing items of some type (template....)

(2) **Iterators:** for accessing items in the containers

(3) **Algorithms:** for operating on the contents of containers, such as finding a particular item, or sorting (a subset) of them.

(4) **functors:** essentially, a class which defines the operator(). We won't say more than this right now.

It has to be noted, though: the STL is not that easy to use. In particular the error messages generated are rather verbose and unhelpful.

A **container** stores objects/elements. These elements can have basic data-type (e.g., `char`, `int`, `double`, ...) or can be objects (e.g., `string`, or user-defined objects).

The most important types of containers are:

`vector`: an indexed sequence (often called "*random access*", though this would be better called "*arbitrary access*". All the items are of the same type. It can be resized, and have new items added to the end. One can also add items to positions not the end, but this is slow.

`set`: a collection of unique items (of the same type), stored in order. When defined relative to a user-defined class, an overloaded `operator<` (less than) must be provided for correct operation.

`multiset`: an ordered collection, like a set, but can have repeated values.

`list`: a doubly linked list.

`stack`: a stack.

`...` etc...

We'll focus on `set`s, `multiset`s and `vector`s.

An `iterator` is an object used to select (or move between) elements in a container.

We can think of them as pointers, that allow us to reference particular elements.

They come in particular flavours:

- forward, reverse, and bidirectional iterators;
- random-access/indexed-access iterators;
- input and output iterators;

7. sets and multisets

To use a `set` or `multiset`, we must

```
#include <set>
```

Suppose we want to create a `multiset` to store `string`s (which just happen to be passwords...), and an iterator for it, we could define

```
1  std::multiset <std::string> multi_pwd;
   std::multiset <std::string>::iterator multi_pwd_i;
```

To add an item to the (multi)set, we could used

```
multi_pwd.insert(MyString);
```

This will add the new string to the `multiset`, automatically choosing its position so that it remains ordered. (If we use a set, it gets inserted into the correct position, providing this does not result in duplication).

7. sets and multisets

Other important methods include

- `begin()` (returns an iterator that points to the first element)
- `end()` (returns an iterator that points to *one past* the end of the set).
- `clear()` (remove contents)
- `count()` (count number of occuences)
- `empty()` (is the set empty?)
- `erase()` (remove an element, or range of elements)
- `find()` (locate an element; return an iterator)
- `size()` (number of elements)
- `swap()` (swap contents of two sets of same type)
- `for_each()` (apply a particular function to each item in a container)

7. sets and multisets

An example of using `begin` and `end` with a set and `multiset`:

02set_and_multiset.cpp

```
10 int main(void )
   {
12   std::set <int> set_int;
     std::set <int>::iterator set_int_i;
14   std::multiset <int> multi_int;
     std::multiset <int>::iterator multi_int_i;

     for (int i=0; i<=20; i+=3) // (0,3,6,9,12,15,18)
18   {
       set_int.insert(i);
20     multi_int.insert(i);
     }
22   for (int i=20; i>0; i-=2) // (20,18,16,...,4,2)
     {
24     set_int.insert(i);
       multi_int.insert(i);
26   }
```

First, we will see how to iterate over the `multiset`:

02set_and_multiset.cpp

```
28   std::cout << "The multiset has " << multi_int.size() <<
       " items." << std::endl;
30   std::cout << "\t They are: ";
     for (multi_int_i = multi_int.begin();
32        multi_int_i != multi_int.end();
          multi_int_i++)
34     std::cout << std::setw(3) << *multi_int_i;
     std::cout << std::endl;
36   std::cout << "\t 6 occurs " << multi_int.count(6) <<
       " time(s)." << std::endl;
```

The output is

```
1  The multiset has 17 items.
     They are:   0  2  3  4  6  6  8  9 10 12 12 14 15 16 18 18 20
3    6 occurs 2 times.
```

Next we will iterate over the `set`:

02set_and_multiset.cpp

```
  std::cout << "The set has " << set_int.size() <<
40    " items." << std::endl;
  std::cout << "\t They are: ";
42  for (set_int_i = set_int.begin();
       set_int_i != set_int.end();
44     set_int_i++)
    std::cout << std::setw(3) << *set_int_i;
46  std::cout << std::endl << "\t 6 occurs " << set_int.count(6)
          << " time(s)." << std::endl;
```

The output is

```
1 The set has 14 items.
        They are:   0  2  3  4  6  8  9 10 12 14 15 16 18 20
3       6 occurs 1 time.
```

8. vector

To use vector, we must

```
1    #include <vector>
```

Unlike a set, we can access a vector by index. Moreover, by default it is not sorted, though there are algorithms to sort its contents.

Since it is unordered, a new item usually gets added to the end, using push_back

This can be removed, using pop_back

Other important methods include

- at
- operator[]
- back (not the same as end)

8. vector

03STL_vector.cpp

```
10 #include <vector>          // vector
   #include <algorithm>       // sort
12 void print_int (int i) { std::cout << std::setw(3) << i; }
   int main(void )
14 {
     std::vector <int> vec_int;
16   std::vector <int>::iterator vec_int_i;
     std::cout << "Vector has " << vec_int.size() <<
18     " elements." << std::endl ;

20   for (int i=3; i>=0; i--)
       vec_int.push_back(i*3); // (9,6,3,0)

     std::cout << "Vector has " << vec_int.size() << " elements: ";
24   for (unsigned int i=0; i<vec_int.size(); i++)
       std::cout << std::setw(3) << vec_int[i];
```

Output (so far):

```
1 Vector has 0 elements.
  Vector has 4 elements:   9   6   3   0
```

8. vector

This snippet demonstrates the use of

- the `find` and `insert` methods;
- the `for_each` iterate through an entire container.

03STL_vector.cpp

```
   vec_int_i = find (vec_int.begin(),vec_int.end(),3);
28 vec_int.insert(vec_int_i,10);

30
   std::cout << std::endl;
   std::cout << "Vector has " << vec_int.size() << " elements: ";
32 for_each (vec_int.begin(), vec_int.end(), print_int);
```

Output (continued):

```
1 Vector has 0 elements.
  Vector has 4 elements:   9  6  3  0
3 Vector has 5 elements:   9  6 10  3  0
```

8. vector

Finally, we show how to sort the items in the list:

03STL_vector.h

```
    std::cout << "Sorting the vector..." << std::endl;
36  sort(vec_int.begin(), vec_int.end());
    std::cout << "Now vector is: ";
38  for_each (vec_int.begin(), vec_int.end(), print_int);
```

Output (all):

```
1 Vector has 0 elements.
  Vector has 4 elements:   9  6  3  0
3 Vector has 5 elements:   9  6 10  3  0
  Sorting the vector...
5 Now vector is:   0  3  6  9 10
```

Other important methods include

- `begin()` (returns an iterator that points to the first element)
- `end()` (returns an iterator that points to *one past* the end of the set).
- `clear()` (remove contents)
- `count()` (count number of occuences)
- `empty()` (is the set empty?)
- `erase()` (remove an element, or range of elements)
- `find()` (locate an element; return an iterator)
- `size()` (number of elements)
- `swap()` (swap contents of two sets of same type)
- `for_each()` (apply a particular function to each item in a container)

The **ranged-based** *for* loop is a recent addition to C++, so it might not work with old compilers. With g++, you may need to enable the `c++11` option.

In the code above, the line

```
1    for_each (vec_int.begin(), vec_int.end(), print_int);
```

could be replaced with

```
1    for (int i : vec_int)
        print_int(i);
```

9. Algorithm

To use `algorithm`, we must

```
#include <algorithm>
```

Useful functions that this provides include

- `for_each`
- `sort` and `partial_sort`
- `search`
- `copy` and `fill`
- `merge`
- `set_union`, `set_difference`
- etc.