

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
- 6 Templates
 - Function Templates

4pm

- 7 The Standard Template Library
 - Containers
 - Iterators
- 8 `sets` and `multisets`
- 9 `vector`
 - Other vector methods
 - Range-based for loops
- 10 `Algorithm`

4pm

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>)

Triplet.h

```
1 // Triplet.h: For 2324-CS319 Week 11
  // Author: Niall Madden
3
4 #ifndef _TRIPLET_H_INCLUDED
5 #define _TRIPLET_H_INCLUDED
6
7 #include "Vector10.h" ← from last week.
  #include "Matrix11.h" // includes zero() method.
```

Triplet.h

```
10 class Triplet {  
    friend Triplet full2Triplet(Matrix &F, unsigned NNZ_MAX);  
12 private:  
    unsigned *I, *J; → row & col index  
14    double *X; → values  
    unsigned N; → number of rows & cols  
16    unsigned NNZ;  
    unsigned NNZ_MAX;  
  
    public:  
20    Triplet (unsigned N, unsigned nnz_max); // Constructor  
    Triplet (const Triplet &t); // Copy constructor  
22    ~Triplet(void);  
  
24    Triplet &operator=(const Triplet &B); // overload assignment
```

Triplet.h

```
26 unsigned size(void) {return (N);};  
   int where(unsigned i, unsigned j); // negative return on error  
28 unsigned nnz(void) {return (NNZ);};  
   unsigned nnz_max(void) {return (NNZ_MAX);};  
  
   double getij (unsigned i, unsigned j);  
32 void setij (unsigned i, unsigned j, double x);  
  
34 unsigned getI (unsigned k) { return I[k];};  
   unsigned getJ (unsigned k) { return J[k];};  
36 double getX (unsigned k) { return X[k];};  
  
38 Vector operator*(Vector u);  
   void print(void);  
40 };  
#endif
```

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

Triplet.cpp

```
1 // Triplet.cpp for 2324-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"
```

Triplet.cpp (Constructor)

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

16     X = new double [nnz_max];
17     I = new unsigned [nnz_max];
18     J = new unsigned [nnz_max];
19     for (unsigned k=0; k<nnz_max; k++) {
20         I[k]=-1;
21         J[k]=-1;
22         X[k]=(double) NULL;
23     }
24 }
```


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)

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

Coding triplet

Triplet.cpp (setij)

```
68 void Triplet::setij (unsigned i, unsigned j, double x)
69 {
70     if (i>N-1)
71         std::cerr << "Triplet::setij(): i Index out of bounds." << std::endl;
72     else if (j>N-1)
73         std::cerr << "Triplet::setij(): j Index out of bounds." << std::endl;
74     else if (NNZ > NNZ_MAX-1)
75         std::cerr << "Triplet::setij(): Matrix full." << std::endl;
76     else
77     {
78         int k=where(i,j); ← Find if aij is already
79         if (k == -1)      stored in the
80         {                Matrix.
81             I[NNZ]=i;    | aij not
82             J[NNZ]=j;    | previously stored -
83             X[NNZ]=x;
84             NNZ++;
85         }
86         else
87             X[k]=x;
88     }
```

Triplet.cpp (operator *)

```
180 Vector Triplet::operator*(Vector u)
181 {
182     Vector v(N); // v = A*u, where A is the implicitly passed Triplet
183     v.zero();
184     if (N != u.size())
185         std::cerr << "Error: Triplet::operator* - dimension mismatch"
186                     << std::endl;
187     else
188         for (unsigned k=0; k<NNZ; k++)
189             v.seti(I[k], v.geti(I[k]) + X[k]*u.geti(J[k]));
190     return(v);
191 }
```

Check, eg, for Matrix on Slide 10
that this works.

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.

If we know that the entries in our matrix are stored in order, then it is possible to store the matrix more efficiently than in Triplet format. One way of doing this is to use **CCS: Compressed Column Storage**, also known as *Harwell-Boeing*

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, r of length `nnz` storing **row** index of the entries. That is, $x[k]$ would be found in row $r[k]$ of the full matrix.
- ▶ an `int` array, c of length $N + 1$, where $c[k]$ stores that starting point of column k as it appears in the arrays x and r , and $c[N] = nnz$.

Example

Show how the matrix below would be stored in CCS

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

$$X = [2, -1, -2, -3, 5, -1, -2, 4, -3, 4]$$

$$r = [0, 3, 6, 8, 10]$$

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

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

```
int index=0;
2 for (int col=0; col<N; col++)
    for (j=c[col]; j<c[col+1]; j++)
4     {
        i=r[index];
6        v[i] += x[index]*b[j];
        index++;
8    }
```

.....

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

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 `ints`.

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 **templates**.

Today we first consider **function templates**. We'll return to the related idea of **class templates** later.

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

`04FunctionTemplate.cpp`, we'll write three functions:

01

- 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

```

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

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

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

The (bubble) `Sort` functions:

01FunctionTemplate.cpp

```

52 template <typename MyType>
void Sort(MyType &a, MyType &b) {
    if (a>b)
54     {
        MyType tmp=a;
56         a=b;
        b=tmp;
58     }
}
```

```

68 template <typename MyType>
void Sort(MyType *x, unsigned int n) {
    for (int i=n-1; i>1; i--)
70         for (int k=0; k<i; k++)
            Sort(x[k], x[k+1]);
72 }
```

01FunctionTemplate.cpp

```
22 int main(void )  
23 {  
24     int Numbers[8];  
25     char Letters[8];  
  
26     for (int i=0; i<8; i++)  
27         Numbers[i]=rand()%40; // Random ints between  
28                                0 and 39  
29     {  
30         for (int i=0; i<8; i++)  
31             Letters[i]='A'+rand()%26;
```

Random Upper-case letter .

01FunctionTemplate.cpp

```

34  std::cout << "Before sorting:" << std::endl;
    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 0

After sorting:

Numbers: 6 12 15 17 23 26 33 35

Letters: A B C D H 0 R Z

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.

Rather like a list in Python except
all the entries are of the
same type

`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 `sets`, `multisets` and `vectors`.

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;

sets and multisets

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

```
#include <set>
```

Suppose we want to create a `multiset` to store `strings` (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;
```

types of data to store.

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

```
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).

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 occurrences)
- ▶ `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)

sets and multisets

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

02set_and_multiset.cpp

```
10 int main(void )
11 {
12     std::set <int> set_int;
13     std::set <int>::iterator set_int_i;
14     std::multiset <int> multi_int;
15     std::multiset <int>::iterator multi_int_i;

16     for (int i=0; i<=20; i+=3) // (0,3,6,9,12,15,18)
17     {
18         set_int.insert(i);
19         multi_int.insert(i);
20     }
21     for (int i=20; i>0; i-=2) // (20,18,16,...,4,2)
22     {
23         set_int.insert(i);
24         multi_int.insert(i);
25     }
26 }
```

sets and multisets

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.
```

sets and multisets

Next we will iterate over the `set`:

Finished here 5pm

02set_and_multiset.cpp

```
40  std::cout << "The set has " << set_int.size() <<
    " items." << std::endl;
42  std::cout << "\t They are: ";
    for (set_int_i = set_int.begin();
44         set_int_i != set_int.end();
        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.
```