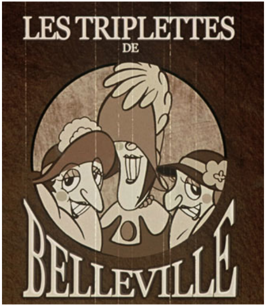
CS319: Scientific Computing

Week 11: Sparse Matrices and templates

Dr Niall Madden

26 + 28 March, 2025



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

0. Outline

3. Coding triplet

Triplet.cpp (setij)

```
void Triplet::setij (unsigned i, unsigned j, double x)
{
  if (i>N-1)
    std::cerr << "Triplet::setij(): i Index out of bounds." << std
  else if (j>N-1)
    std::cerr << "Triplet::setij(): j Index out of bounds." << std
  else if (NNZ > NNZ_MAX-1)
    std::cerr << "Triplet::setij(): Matrix full." << std::endl;
  else
  {
    int k=where(i,j);
    if (k == -1) // nothing stored in [i,j].
    {                so this is a
      I[NNZ]=i;      new entry.
      J[NNZ]=j;
      X[NNZ]=x;
      NNZ++;
    }
    else
      X[k]=x;
  }
}
```

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 k=0; k<NNZ; k++)
      v.seti(I[k], v.geti(I[k]) + X[k]*u.geti(J[k]));
  return(v);
}
```

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/H`~~`yTZcvAN7`~~`t`

See link on website.

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.
  r[i]
- an `int` array, r of length $N+1$, where $r[k]$ stores that **starting point** of row k as it appears in the arrays x and c, and $r[N] = nnz$.
  i

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

Example

Show how the matrix below would be stored in CSR

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

$x = [\,2 \;\; -1 \;\; 5 \;\; -1 \;\; -2 \;\; 4 \;\; -3 \;\; 4$

$c = \{\,0 \;\; 1 \;\; 1 \;\; 2 \;\; 1 \;\; 2 \;\; 0 \;\; 3$

$r = [\,0, 2, 4, 6, 8\,]$

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**. 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 `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;
  }
}
```

Important: this assumes that the operator > is overloaded for this type.

```
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

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

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

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

01FunctionTemplate.cpp

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

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

  std::cout << "After sorting: " << std::endl;
  std::cout << "Numbers: ";  PrintList(Numbers, 8);
  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;

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

note: we are really using templates.

```
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).

Finished here Friday @ 12.