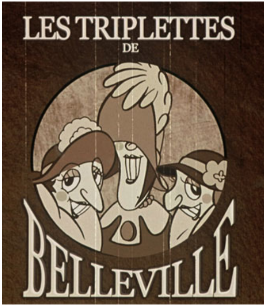
Annoated slides from 9am

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

4pm

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`)

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.

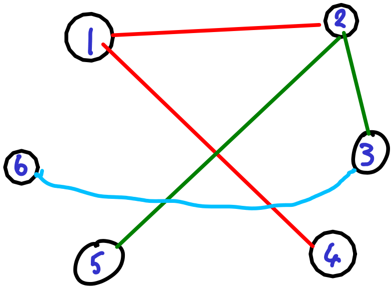
But perhaps the most obvious example is when we use matrices to represents graphs and **networks**.

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:

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

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

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

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

$NNZ = 8$

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
X I J } Arrays.	geti j() seti j() etc.
	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
```

Coding triplet

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

Coding triplet

Triplet.cpp

```
// Triplet.cpp  for 2324-CS319 Week 11
//   What: Methods for the Triplet class
// Author: Niall Madden
#include <iostream>
#include <iomanip>
#include "Vector10.h"
#include "Matrix11.h"
#include "Triplet.h"
```

Coding triplet

Triplet.cpp (Constructor)

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

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

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 i, unsigned j)
{
  unsigned int k=0;
  do {
    if ( (I[k]==i) && (J[k]==j) )
      return(k);
    k++;
  } while (k<NNZ);
  return(-1);
}
```

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)
    {
      I[NNZ]=i;
      J[NNZ]=j;
      X[NNZ]=x;
      NNZ++;
    }
    else
      X[k]=x;
  }
}
```

← Find if a_{ij} is already stored in the Matrix.

a_{ij} not previously stored.

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

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.

CCS

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 **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 $c[N] = nnz$.

CCS

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

. . . .

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 `int`s.

Suppose we wanted to sort arrays of a different type, e.g., `string`s. We could take our old `Sort(int *list, int length)` function from `Week05/04Sort.cpp` and rewrite it as `string`s: `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 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

```
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

```
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;
```

// Random ints between 0 and 34

Random Upper-case letter.

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
```

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 <u>list</u> 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 `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;

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

types of data to store.

```
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).

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)

sets and multisets

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

02set_and_multiset.cpp

```
int main(void )
{
  std::set <int> set_int;
  std::set <int>::iterator set_int_i;
  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)
  {
    set_int.insert(i);
    multi_int.insert(i);
  }
  for (int i=20; i>0; i-=2) // (20,18,16,...,4,2)
  {
    set_int.insert(i);
    multi_int.insert(i);
  }
```

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

02set_and_multiset.cpp

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

The output is

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

sets and multisets

Next we will iterate over the `set`: Finished here 5pm

02set_and_multiset.cpp

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

The output is

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

vector

To use vector, we must

```
#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)

03STL_vector.cpp

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

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

Output (so far):

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

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);
vec_int.insert(vec_int_i,10);

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

Output (continued):

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

vector

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

03STL_vector.h

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

Output (all):

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

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

could be replaced with

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

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.