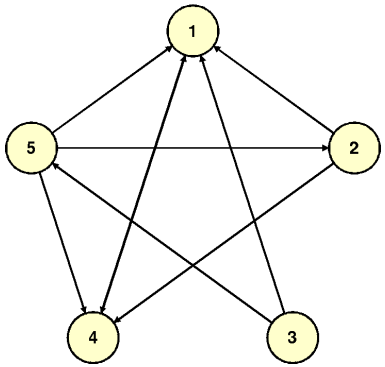
CS319: Scientific Computing (with C++)

Network Analysis

Week 9: **9am** and **4pm**, 08 March 2017



About the labs...

So far you have had to complete **4** assignments, with one more due next week (??).

There will be one further, final assignment ("mini-project") before the end of the semester.

Together the assignments are worth 40% of the final grade for CS319.

If you are unable to complete an assignment before the proposed deadline, it is important that you ask for an extension **in advance**.

The purpose of the assignments is to help you refine your programming skills, and obtain feedback.
They also serve to assess your achievements in programming and scientific computing, in a setting that is more realistic than a written exam.

Our main topic of interest today is to study a tool used in the analysis of networks (e.g., social networks).

First, though, we'll recap a little on operator overloading, and finish some topics we didn't get to last week.

Recall: Operator Overloading

Last week, we saw that we can overload standard operators in C++ as members of a class. The general form of the operator function is:

```
return-type class-name::operator#(arguments)
{
    .
    .
    .    // operations to be performed.
};
```

return-type of a operator is usually the class for which it is defined, but it can be any type. Here `operator` is a key-word. The operator being overloaded is substituted for `#`

Almost all C++ operators can be overloaded:

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	& =
|=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
||	++	--	->*	,	->	[]	()	new	delete

but not . :: .* ?

Recall: Operator Overloading

- **Operator precedence** cannot be changed: `*` is still evaluated before `+`
- The number of arguments that the operator takes cannot be changed, e.g., the `++` operator will still take a single argument, and the `/` operator will still take two.
- The original meaning of an operator is not changed; its functionality is extended.
- It follows from this that operator overloading is always relative to a user-defined type (in our examples, a `class`), and not a built-in type such as `int` or `char`.
- The assignment operator, `=`, is automatically overloaded, but in a way that usually fails except for very simple classes.

For our first example, we extended the definition of + so that we can add two vectors.

Steps:

(1) add the declaration of the operator to the class definition in the header file, Vector08.h:

```
vector operator+(vector b);
```

(2) Add the follow code to Vector08.cpp,

```
1  vector vector::operator+(vector b) {
2     vector c(N); // Make c the same size of a
3     if (N != b.N)
4        std::cerr << "vector::+ : cant add two vectors of different size!"
5             << std::endl;
6     else
7        for (unsigned int i=0; i<N; i++)
8           c.entries[i] = entries[i] + b.entries[i];
9     return(c);
10 }
```

Notice is that, although + is a binary operator, it seems to take only one argument. This is because, when we call the operator, `c = a + b` then `a` is passed **implicitly** to the function and `b` is passed **explicitly**.
Therefore, for example, `a.N` is known to the function simply as `N`.

When writing code for functions, and especially overloaded operators, it can be useful to **explicitly** access the implicitly passed object. That is done using the `this` pointer, which is a pointer to the object itself.

Since `this` is a pointer, its members are accessed using either `(*this).N` or `this->N`.

We used `this` in defining the assignment operator for our class.

```
// Overload the = operator.  From CS319 -- Week 8/9
vector &vector::operator=(const vector &b)
{
   if (this == &b)
      return(*this); // Check for self-assignment

   delete [] entries; // De-allocate any existing memory

   entries = new double[b.N];
   for (unsigned int i=0; i<N; i++)
      entries[i] = b.entries[i];

   return(*this);
}
```

`friend` functions

In all the examples that we have seen so far, the only functions that may access private data belonging to an object has been a member function/method of that object.

However, it is possible to designate non-member as being a `friend` of a class.

For non-operator functions, there is nothing that complicated about `friend`s. However, care must be taken when overloading operators as `friend`s.

In particular:

- All arguments are passed explicitly to `friend` functions/operators.
- Certain operators, particularly the **Put to** `<<` and **Get from** `>>` operators can only be overloaded as friends.

friend functions

Suppose we add the following line to the class definition in `Vector08.h`

```
friend std::ostream &operator<<(std::ostream &, vector &v);
```

And the we define:

```
std::ostream &operator<<(std::ostream &output, vector &v)
{
   output << "[";
   for (unsigned int i=0; i<v.size()-1; i++)
      output << v.entries[i] << ",";
   output << v.entries[v.size()-1] << "]";

   return(output);
}
```

Now we can display a vector using `std::cout` directly.

Linear Algebra

This is a course about programming and scientific computing. The idea is the study some standard algorithms in computational science, and along the way master the necessary components of C++ to allow us to implement them.

Much of this involves **numerical linear algebra**: algorithms for working with **matrices** and **vectors**. Last week, we looked a the Jacobi and Gauss-Seidel methods for solving linear systems. Today we'll look at the other major topic associated with numerical linear algebra: **computing eigenvalues and eigenvectors**.

After that, you'll learn about matrix storage methods, and some related ideas.

Recall... Matrices

As we know, a **matrix** is a rectangular array of numbers.

In C++, an $N \times N$ matrix of doubles can be declared as:

```
double A[5][5];
```

Then its members are

$$\begin{pmatrix} A[0][0] & A[0][1] & A[0][2] & A[0][3] & A[0][4] \\ A[1][0] & A[1][1] & A[1][2] & A[1][3] & A[1][4] \\ A[2][0] & A[2][1] & A[2][2] & A[2][3] & A[2][4] \\ A[3][0] & A[3][1] & A[3][2] & A[3][3] & A[3][4] \\ A[4][0] & A[4][1] & A[4][2] & A[4][3] & A[4][4] \end{pmatrix}$$

To use dynamic memory allocation to reserve memory for an $N \times N$ matrix:

```
double **A;
A = new double* [N];
for (k=0; k<N; k++)
      A[k] = new double [N];
```

However, it is simpler if we store the matrix as a one dimensional array...

Recall... Matrices

Instead of the 2D array we had before, we'll store our matrices as one-dimensional arrays. If this is done row-wise we get:

$$\begin{pmatrix} A[0] & A[1] & A[2] & A[3] & A[4] \\ A[5] & A[6] & A[7] & A[8] & A[9] \\ A[10] & A[11] & A[12] & A[13] & A[14] \\ A[15] & A[16] & A[17] & A[18] & A[19] \\ A[20] & A[21] & A[22] & A[23] & A[24] \end{pmatrix}$$

(However, there are many other ways of storing a matrix...)

There are many extremely important algorithms involving matrices, and most of these involve multiplying a matrix by a vector.

MatrixVectorMult: $\mathbf{u} = A\mathbf{b}$

for $i = 0$ to $N - 1$ **do**
  $u_i \leftarrow 0$
  for $j = 0$ to $N - 1$ **do**
    $u_i \leftarrow u_i + A_{ij} b_j$
  end for
end for

It is important to understand this seemingly simple computation, so that we can generalise to other storage methods.

Definition (Eigenvalues and Eigenvectors)

Let A be an $N \times N$ matrix.

A (real or complex-valued) number λ is an **eigenvalue** *of A if there is a* **nonzero** *vector $v \in \mathbb{R}^N$ such that $A\mathbf{v} = \lambda\mathbf{v}$.*

The vector **v** *is then called an* **eigenvector** *of A corresponding to the eigenvalue λ.*

The name comes from the German: "**eigen**" can be translated as "**characteristic**", meaning that the eigenvalues of a matrix represent some of its intrinsic properties.

Note: If **v** is an eigenvector corresponding to λ, so too is the vector $\alpha\mathbf{v}$, for *any* number $\alpha \neq 0$.

Example

$$\begin{pmatrix} 2 & 2 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix} = 4 \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

So $\lambda = 4$ is an eigenvalue with a corresponding eigenvector $(1, 1)^T$.

Computing Eigenvalues and Eigenvectors

The standard way of finding the eigenvalues and vectors of a matrix is:

1. subtract λ from each diagonal entry,
2. Compute the determinant – this will be a polynomial of degree n.
3. Find its roots: these are the *eigenvalues* of A.

However,

(a) this is very tedious to do for all, but very small matrices

(b) it only works for small matrices,

(c) there is an easier way if you only want the largest eigenvalue and corresponding eigenvector.

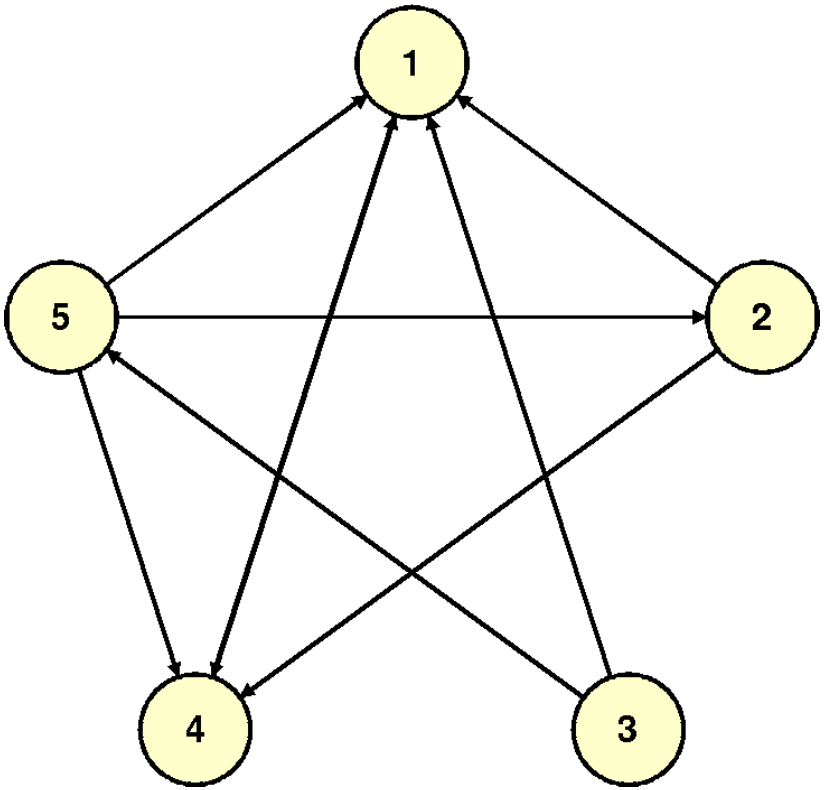
This easier way is called the "**Power Method**". It relies just on matrix-vector multiplication, scalar-vector multiplication, and computation of vector norms. However, in the case we are interested in, it is even easier...

PageRank

Google initial break-through in search engine design was derived from their **PageRank** algorithm which gives an objective way of computing the relative importance of web-pages.

The basic idea is this: **the importance of a web-page is the probability that you are looking at it at any given time**.

To see how this works, consider the following example:



PageRank

To summarise:

(1) Form the **adjacency matrix**, $A = (a_{ij})_{i=1}^{N}$, for the network:

$$a_{i,j} = \begin{cases} 1 & \text{if the graoh has an edge going from } i \text{ to Node } j; \\ 0 & \text{otherwise.} \end{cases}$$

(2) Make the associated **Markov matrix**, $S = (s_{ij})_{i=1}^{N}$, where $S_{i,j}$ is the *proportion* of vertices in A which start at i and go to j. (That is, divide the entries in row i by the sum of the entries in that row). If there are no entries in a given row of A, set the corresponding entries of S to $1/N$.

(3) Choose a "damping" value σ, e.g., $\sigma = 0.85$.

(4) Set the matrix G to be $\left(\sigma S + (1-\sigma)/N\right)^T$.

(5) Now find the eigenvector associated with the eigenvalue that is 1. This can be done with the **Power Method**.

PageRank

We should mention that there are many other network analysis tools. However, most of them depend on both

- Formation of the adjacency matrix;
- Multiplication of matrices.

An example of this is finding the number of routes of a given length between two vertices in a graph.

All we have to do now is get the eigenvector of G associated with the eigenvalue 1.

The Power Method Algorithm

INPUTS: G (the Google Matrix), TOL
$u \leftarrow (1/N, 1/N, \ldots, 1/N)$
$v \leftarrow (0, 0, \ldots, 0)$
$d \leftarrow u - v$
while $\|d\| \leq TOL$ **do**
  $v \leftarrow u$
  $u \leftarrow Gv$
  $d \leftarrow u - v$
end while
RETURN(u)

(This is a simplified version of the Power Method because we know $\lambda = 1$.)

From our example earlier, the first few results are:

Iteration	0	1	2	3
u	$\begin{pmatrix}0.2\\0.2\\0.2\\0.2\\0.2\end{pmatrix}$	$\begin{pmatrix}0.4267\\0.0867\\0.0300\\0.3417\\0.1150\end{pmatrix}$	$\begin{pmatrix}0.4026\\0.0626\\0.0300\\0.4621\\0.0428\end{pmatrix}$	$\begin{pmatrix}0.4742\\0.0421\\0.0300\\0.4109\\0.0428\end{pmatrix}$

Sparse Matrices

Suppose we wanted to use the algorithm above to analyse a social network. For the problem to be interesting, our network should have thousands, perhaps millions, of nodes.

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 Row Storage** (CRS) (after triplet)
- **Compressed Column Storage** (CCS)
  And the following formats for very specialised matrices:
- **Block Compressed Row/Column Storage**
- **Compressed Diagonal Storage**
- **Skyline**

Triplet

The basic idea for triplet form is:

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

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

Triplet

Our next goal is to work out how we should implement a triplet matrix in our programmes? The main tasks are:

- Decide if we should create a whole new class for triplets?
- Implement a matrix-vector multiplication algorithm.

Discussion...

Triplet

Triplet.h ⟵ link!

```
1  // Triplet.h:  For CS319 Week 9
   // Author: Niall Madden,
3  //   Date: 08 March 2017

5  #ifndef _TRIPLET_H_INCLUDED
   #define _TRIPLET_H_INCLUDED
7
   #include "Vector.h"
9  #include "Matrix.h"

11 class triplet {
     friend triplet full2triplet(matrix &F, unsigned int NNZ_MAX);
13 private:
     unsigned int *I, *J;
15   double *X;
     unsigned int N;
17   unsigned int NNZ;
     unsigned int NNZ_MAX;
```

Triplet

Triplet.h ⟵ link!

```
20  public:
      triplet (unsigned int N, unsigned int nnz_max); // Constructor
22    triplet (const triplet &t); // Copy constructor
      ~triplet(void);
24
      triplet &operator=(const triplet &B); // overload assignment operator
26
      unsigned int size(void) {return (N);};
28    int where(unsigned int i, unsigned int j); // allow negative return
      // on errors
30    unsigned int nnz(void) {return (NNZ);};
      unsigned int nnz_max(void) {return (NNZ_MAX);};
32
      double getij (unsigned int i, unsigned int j);
34    void setij (unsigned int i, unsigned int j, double x);

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

40    vector operator*(vector u);
      void print(void);
42  };
    #endif
```

Triplet

Triplet.cpp ← link!

```
1  // Triplet.cpp  for CS319 Week 9
   //   What: Methods for the matrix class, including operator overloading
3  // Author: Niall Madden
   //   Date: 08/03/2017
5  #include <iostream>
   #include <iomanip>
7  #include "Vector.h"
   #include "Matrix.h"
9  #include "Triplet.h"

11 // Standard  constructor. Copy constructor.
   triplet::triplet (unsigned int N, unsigned int nnz_max) {
13   this->N = N;
     this->NNZ_MAX = nnz_max;
15   this->NNZ = 0;

17   X = new double [nnz_max];
     I = new unsigned int [nnz_max];
19   J = new unsigned int [nnz_max];
     for (unsigned int k=0; k<nnz_max; k++)  {
21     I[k]=-1;
       J[k]=-1;
23     X[k]=(double)NULL;
     }
25 }
```

Triplet

Triplet.cpp ⟵ link!

```
void triplet::setij (unsigned int i, unsigned int j, double x)
{
  if (i>N-1)
    std::cerr << "triplet::setij(): i Index out of bounds." << std::endl;
  else if (j>N-1)
    std::cerr << "triplet::setij(): j Index out of bounds." << std::endl;
  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;
  }
}
```

Triplet

Triplet.cpp ⟵ link!

```
// Overload the operator* (multiplication)operator for a
// triplet-vector product.
// Triplet is passed implicitly as "this", the vector
// is  passed explicitly. Will return v=(this)*u
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);
}
```

To demonstrate the use of the Triplet class, I've included a program called 01triplet_example which shows how to use the Jacobi method to solve a linear system where the matrix is stored in triplet format.

Compressed Column Storage

If we know that the entries in our matrix are stored in order than it is possible to store the matrix more efficiently that in Triplet format. One way of doing this is to use **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 `integer` array, r of length `nnz` storing column index of the entries. That is, $x[k]$ would be found in row $r[k]$ of the full matrix.
- a `integer` 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:

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

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

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

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

The last format to consider is **Compressed Row Storage**. However it is almost identical to CCS except that the array `r` stores the starting point of a row, and the array `c` stores the column indices of all entries.