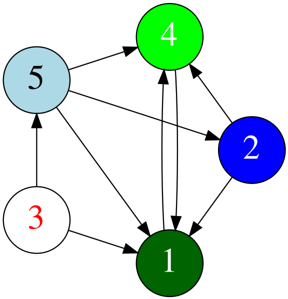
CS319: Scientific Computing (with C++)

Week 9: More Operator Overloading; Network Analysis

9am, 13 April, and 4pm, 14 April, 2021



Usual reminders...

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			LECTURE		

1. Two recorded classes this week: Tuesday at 09.00, and Wednesday at 16.00.
2. **Lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 1: Recapping on Operator Overloading

Here is a short summary of what we covered on Operator Loading before the Easter break

Part 1: Recapping on Operator Overloading

In Week 8 we began study of a major aspect of Object Oriented Programming: **"Operator overloading"** .

We saw how to overload the assignment (`=`) operator, and the addition (`+`) operator for vectors. Now we''ll overload (`*`) for matrix-vector multiplication.

First we'll summarise some of the major points from Week 8.

. . . .

See also **"extras"** section of Week 8 lectures for more examples of classes and overloading (points, dates, complex numbers); Code for these is in the `Week08/extras/` folder on the repository/website.

These slides do not include all issues concerning operator overloading. Among the topics omitted are:

- overloading the unary `++` and `--` operators. There are complications because they work in both prefix and postfix form.
- Overloading the ternary operator: `? :`
- **Important:** overloading the `[]` operator.

Part 1: Recapping on Operator Overloading

- To overload an operator we create an **operator function** – usually as a member of the class.
- The general form of the operator function is:

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

- *return-type* of a operator is usually the class for which it is defined, but it can be any type.
- `operator` is a new keyword. The operator being overloaded is substituted for `#`
- Almost all C++ operators can be overloaded (see notes from Week 8 for full list) but not `.` `::` `.*` `?`
- 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.

Part 1: Recapping on Operator Overloading

- The original meaning of an operator is not changed; its functionality is extended.
- Operator overloading is always relative to a user-defined type (in our examples, a `class`).
- The assignment operator, `=`, is automatically overloaded, but in a way that usually fails except for very simple classes (see notes from Week 8)
- For binary operators that belong to a class, the left argument is passed **implicitly**; an example of this is overloading the binary `+` operator for `Vector` class.
- If `w` is a pointer, then `w->N` is equivalent to `(*w).N`.
- To explicitly reference the implicitly passed object, use the `this` pointer, which is a pointer to the object itself.

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 1

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 2: Unary Operators

A **binary** operator is one that takes two arguments, for example the multiplication operator.
But some operators are **unary**, meaning they take a single argument.

Part 2: Unary Operators

So far we have discussed just the **binary** operator, `+`. By "**binary**", we mean it takes **two** arguments.

But many C++ operators are **unary**: they take only one argument.

The most common examples of unary operators are `++` and `--`, but for our `Vector` class, we'll first over load the `-` (minus) operator. Note that this can be used in two ways:

- $c = -a$ (unary).
- $c = a - b$ (binary)

In the first case here, "minus" is an example of a **prefix** operator. (See Week 8 "Extras" for example of overloading **postfix** operators, like `a++`, which are a little more complicated).

After that we will then define the binary minus operator, by using addition and unary minus.

Part 2: Unary Operators

See `Vector09.cpp` for more details

```
// Overload the unary minus (-) operator.  As in b=-a;
Vector Vector::operator-(void)
{
  Vector b(N); // Make b the size of a
  for (unsigned int i=0; i<N; i++)
    b.entries[i] = -entries[i];
  return(b);
}

// Overload the binary minus (-) operator.  As in c=a-b
// This implementation reuses the unary minus (-) operator
Vector Vector::operator-(Vector b)
{
  Vector c(N); // Make b the size of a
  if (N != b.N)
    std::cerr << "Vector:: operator- : dimension mismatch!"
              << std::endl;
  else
    c = *this + (-b);
  return(c);
}
```

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 2

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 3: Preprocessor Directives

As with the *Vector* class, we would like to overload some functions and arithmetic operators for *Matrix*. In the files `Matrix09.h` and `Matrix09.cpp` examples are given for

- defining the copy constructor;
- overloading the assignment operator.

They follow the same ideas as the corresponding components of the *vector* class.

With those done, we can think about overloading the multiplication operator for *Matrix*-*Vector* multiplication.

This introduces a few small new complications:

- the return type is different from the class type;
- if we use multiple source files, how do we know where exactly to place the `#include` directives?

So, before we can proceed, we need to take a short detour to consider **preprocessor** directives.

The preprocessor in C++ is a hang-over over from early versions of C. Originally, that language did not have a construct for defining constants and including header files. To get around this, an early version of C introduced the **preprocessor**. This is a program that

- reads and modifies your source code by checking for any lines that being with a hash symbol (`#`);
- carries out any operations required by these lines;
- forms a new source code that is then compiled.

We usually don't get to see this new file, though you can view it by compiling with certain options (with `g++`, this is `-E`).

The preprocessor is *separate* from the compiler, and has its own syntax.

The simplest preprocessor directive is #define. This is used for defining global constants, and doing a simple search-and-replace. For example,

```
#define SIZE 10
```

will find every instance of the word (well, token, really) *SIZE* and replaces it with 10.

In general, this use of the #define directive to define identifiers to be used like "global variables" is not very good practice. However, it can be very useful as a way of checking if a piece of code has already been compiled.

The most familiar preprocessor is #include, e.g.,

```
#include <iostream>
#include "Vector09.h"
```

This tells the preprocessor to take the named file(s) and insert them into the current file.

If the name is contained in angle brackets, as in iostream, this means the preprocessor will look in "the usual place" – where the compiler is installed on your system.

If the named file is in quotes, it looks in the current directory, or in the specified location.

Finally, we have **conditional compilation**.

Suppose we want to write a member function for the *Matrix* class that involves the Vector class.

So we need to include Vector09.h in Matrix09.h. But then if our main source file includes both Matrix09.h and Vector09.h we could end up defining it twice.

To get around this we use *conditional compilation*.

In the files we can have such lines as the following in Vector09.h

```
#ifndef _VECTOR_H_INCLUDED
#define _VECTOR_H_INCLUDED
// stuff goes here
#endif
```

In another use, we might want the compiler to behave in particular ways for particular operating systems. E.g.,

```
#ifndef linux
system("PAUSE");
#endif
```

. . . .

Other applications of preprocessor directives include defining parameterized macro (which is like a function), and `#pragma` directives for certain compilers. The latter is used a lot in parallel computing.

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 3

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 4: Overloading ∗ for MatVec

Finally, we get around to being able to multiply matrices and vectors using ∗

Part 4: Overloading $*$ for MatVec

Finally, we are ready to overload `operator*` for multiplication of a vector by a matrix: $c = A * b$, where A is an $N \times N$ matrix, and c and b are vectors with N entries.

Since the left operand is a matrix, we'll make this operator a member of the `Matrix` class, and add this line to the definition in `Matrix09.h` :

```
Vector operator*(Vector b);
```

Part 4: Overloading * for MatVec

The code from `Matrix09.cpp` is given below. Compare with `MatVec`

```
// Overload the operator multiplication (*) for a Matrix-Vector
// product.  Matrix is passed implicitly as "this", the Vector is
// passed explicitly.  Will return v=(this)*u
Vector Matrix::operator*(Vector u)
{
  Vector v(N); // v = A*u, where A is the implicitly passed Matrix
  if (N != u.size())
    std::cerr << "Error: Matrix::operator* - dimension mismatch"
              << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
    {
      double x=0;
      for (unsigned int j=0; j<N; j++)
        x += entries[i*N+j]*u.geti(j);
      v.seti(i,x);
    }
  return(v);
}
```

Equipped with this, we can now write a neater version of the `Jacobi` function:

Old Version

```
Vector d(N), r(N);
do  {
  count++;
  MatVec(T,x,d);
  VecAdd(d, b, d);
  MatVec(Dinv, d, x);

  MatVec(A, x, r);
  VecAdd(r, b, r, 1.0, -1.0);
}   while ( r.norm() > tol);
```

New Version

```
Vector r(N);
do {
  x = Dinv*(b+T*x);
  count++;
  r = b-A*x;
}   while ( r.norm() > tol);
```

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 4

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 5: `friend` functions

A methods of a class have access to other methods. But we can grant the same access to external functions by making them `friend`s.

Part 5: `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 **insertion/put-to** `<<` and **extraction/get-from** `>>` operators can only be overloaded as friends.

In last week's version of the Vector class, we could output its elements using the print() method. E.g.:

```
Vector v;
v.zero()
std::cout << "v  has values ";
v.print();
```

But it would be much more convinient just to do

```
std::cout << "v  has values " << v;
```

But the **insertion** operator was not defined for our class.

We can fix that, by overloading it. However, the << operator belongs to std::cout, not to Vector. So it cannot access its entries member.

Here is how we resolve this...

We add the following line to the class definition in Vector09.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.

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 5

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

PART 6: Linear Algebra

Part 6: 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.

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

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

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

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: see Week 10)

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 $\mathbf{v}$ *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 $\mathbf{v}$ 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 $(1, 1)^T$.

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...

CS319 – Week 9: More Operator Overloading; Network Analysis

END OF PART 6

CS319 – Week 9: More Operator Overloading; Network Analysis

Start of ...

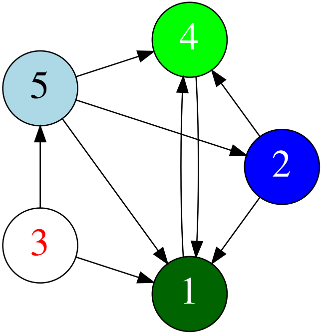
PART 7: PageRank

And now for a multi-billion dollar algorithm

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:



Part 7: PageRank

To summarise:

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

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

(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 $\big(\sigma S + (1-\sigma)/N\big)^T$.

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

Part 7: 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	$\begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$	$\begin{pmatrix} 0.4267 \\ 0.0867 \\ 0.0300 \\ 0.3417 \\ 0.1150 \end{pmatrix}$	$\begin{pmatrix} 0.4026 \\ 0.0626 \\ 0.0300 \\ 0.4621 \\ 0.0428 \end{pmatrix}$	$\begin{pmatrix} 0.4742 \\ 0.0421 \\ 0.0300 \\ 0.4109 \\ 0.0428 \end{pmatrix}$