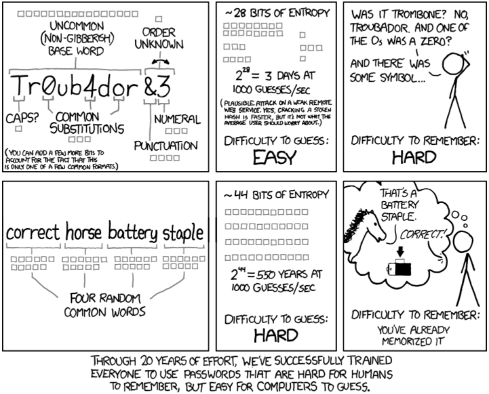
CS319: Scientific Computing (with C++)

# Week 7: The Password Problem; Vectors & Matrices

9am, 23 March, and 4pm, 24 March, 2021




http://xkcd.com/936

[Originally, the Week 6 class was titled "The Password Problem"; but I didn't actually get 'round to it!]

# 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.

# Usual reminders...

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 0: Feedback on Feedback

# Part 0: Feedback on Feedback

- Thank-you to the 8 of you that completed the feedback form circulated by Noelle Cannon.
- On average, it took 1 minutes, 49 seconds to complete.
- Mostly very positive.
- A small number of people are "unsure" or "disagree somewhat" with the statement that "The feedback I have received is helping me to improve my learning". Which is fair! (Will do better!).
- The "live-but-recorded" lectures seem to be popular (which I was unsure of, since the quality is not very high).
- Some good suggestions for improvement, including
  - **"An example of longer code from start to finish, I find it hard to see how the code works as a whole when I only see snippets of code"**. [Response: Fair point. Although the entire code is made available separately, and the snippets have line-numbers, I will do some start-to-finish examples soon.]

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART 0**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 1: The Password Problem (finally!)

# Part 1: The Password Problem

Recall from last week that our aim is to take a very long list of passwords and to determine the most common.

The source of the data is the infamous **RockYou** password file, a list of over 30,000,000 unencrypted passwords stolen from RockYou in 2009, and now widely available online.

The file contains one password per line, in no particular order. The first few are

```
password
mekster11
mekster11
progr4sm
khas8950
emilio1
holiday2
caitlin1
```

**Given a list of 30,000,000 passwords, how shall we work out which 10 (say) occur most frequently?**

Idea:

1. Read the list of passwords from the file.
2. Sort the list alphabetically.
3. Calculate the frequency of each word, while removing duplicates.
4. Make a new list of the unique words, and their frequencies.
5. Sort this list by frequency.

The first step is to open the file, and count the number of lines, and the length of the longest line.

00SortPasswords.cpp

```
std::ifstream InFile;
std::string InFileName="UserAccount-1e4.txt";

InFile.open(InFileName.c_str());
if (InFile.fail() )
{
  std::cerr << "Error: Cannot open " << InFileName <<
    " for reading." << std::endl;
  exit(1);
}

// Need to know the number of lines, and the length of the longest one
unsigned int LineCount, LongestLine;
LineCount =  FileLength(InFile, LongestLine);
```

00SortPasswords.cpp

```
int FileLength(std::ifstream &InFile, unsigned int &LongestLine)
{
  InFile.clear();
  InFile.seekg(std::ios::beg); // Rewind to the start of the file
  char c;
  InFile.get( c );
  unsigned int LineCount=0, ThisLineLength=0;
  LongestLine=0;
  while( ! InFile.eof() ) {
    if (c != '\n')
      ThisLineLength++;
    else  {
      LineCount++;
      if (LongestLine<ThisLineLength)
        LongestLine = ThisLineLength;
      ThisLineLength=0;
    }
    InFile.get( c );
  }
  InFile.clear();
  InFile.seekg(std::ios::beg);     // Rewind
  return(LineCount);
}
```

Now read the file (again) and store the passwords in an array. Again, we write a single stand-alone function to do this.

00SortPasswords.cpp

```
void ReadPasswords(std::ifstream &InFile, std::string *Passwords,
        unsigned int &LineCount, unsigned int LongestLine)
{
  int WordsRead=0;
  char *c_string_word = new char [LongestLine+1];
  for (unsigned int Line=0; Line < LineCount; Line++)
  {
    InFile.getline(c_string_word, LongestLine+1);
    Passwords[Line] = c_string_word;
    if (Passwords[Line].length() == 0) // that was a blank line
      Line--;
    else
      WordsRead++;
  }
  LineCount = WordsRead;
  delete [] c_string_word;
}
```

The next step (main, Line 55) is to call the `MergeSort()` function. We then have the task of finding which word occurs most frequently. The approach is to create to new arrays:

(a) a new list of `string`s, called `UniqueWords`, where each password appears, but only once.

(b) a corresponding `int` array `WordFreq`. When we are done, if `WordFreq[k]=x`, then `UniqueWords[k]` appeared $x$ times in the original list.

**00SortPasswords.cpp**

```
   std::string *UniqueWords = new std::string [LineCount+1];
66   unsigned int *WordFreq  = new unsigned int [LineCount+1];
   unsigned int UniqueWordsFound;

// The first one can't already be on the list
70   UniqueWords[0] = Passwords[0];
   WordFreq[0] = 1;
72   UniqueWordsFound=1;
```

*continued...*

00SortPasswords.cpp

```
for (unsigned int i=0; i<LineCount; i++)
{
  if (Passwords[i] !=  UniqueWords[UniqueWordsFound-1])
  { // We have found a new word
    UniqueWords[UniqueWordsFound] = Passwords[i];
    WordFreq[UniqueWordsFound] = 1;
    UniqueWordsFound++;
  }
  else
    WordFreq[UniqueWordsFound-1]++;
}
```

**Explanation:**

Our next step is to create a list to the 10 most frequently used. This information will be stored in two arrays:

`string Top10[10];`

`int Top10Freq[10];`

We will keep this list ordered. Then iterate through the UniqueWords list. If we find a word that occurs more often than the (current) 10th most common, we insert it into the list:

00SortPasswords.cpp

```
  // Insert the 1st into list and set rest to 0
  Top10[0]=UniqueWords[0];
  Top10Freq[0]=WordFreq[0];
  for (int i=1; i<10; i++)
  {
    Top10[i]="";
    Top10Freq[i]=0;
  }
// See if this is at least as freq as the 10th most
  for (unsigned int i=1; i<UniqueWordsFound; i++)
    if (WordFreq[i] > Top10Freq[9])
      Insert(Top10, Top10Freq, UniqueWords[i], WordFreq[i]);
```

To finish, we'll see how the Insert function works:

00SortPasswords.cpp

```
// Insert NewString into the list Top10, ordered by
// NewCount in Top10Freq, bumping anything if needed
void Insert(std::string *Top10, unsigned int *Top10Freq,
            std::string NewString, unsigned int NewCount)
{
  if (NewCount <= Top10Freq[9])
    std::cerr << "Error: new entry would not make top 10" << std::endl;
  else
  {
    Top10[9]=NewString;
    Top10Freq[9]=NewCount;
    for (int i=8; i>=0; i--)
    {
      if (Top10Freq[i]<NewCount)
      {
        Top10[i+1] = Top10[i];
        Top10Freq[i+1] = Top10Freq[i];
        Top10[i]=NewString;
        Top10Freq[i]=NewCount;
      }
    }
  }
}
```

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART 1**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 2: Vectors and Matrices

Motivation

# Part 2: Vectors and Matrices

This is a course in Scientific Computing. "**Sci-Comp**" problems that we've looked at so far include

- optimisation;
- searching and list processing.

Many of the more advanced and more general problems in Scientific Computing are based around **vectors** and **matrices**. So one of our goals is to implement C++ classes for such structures, along with standard operations such as matrix-vector multiplication.

Along the way, we'll learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

Our first step will be to study some problems and applications so that, before we design any classes or algorithms, we'll know what we will use them for. These problems include:

1. Basic analysis of matrices, for example with applications to image processing, graphs and networks.
2. Solution of linear systems of equations, for example with applications to data fitting;
3. Estimation of (certain) eigenvalues, for example with applications to search engine analysis.

Of these problems, probably the most ubiquitous is the solution of (large) systems of simultaneous equations.

That is, we want to solve a linear system of 3 equations in 4 unknowns: *find* $x_1, x_2, x_3$, *such that*

$$\begin{aligned} 3x_1 + 2x_2 + 4x_3 &= 19 \\ x_1 + 2x_2 + 3x_3 &= 14 \\ 5x_1 + 1x_2 + 6x_3 &= 25 \end{aligned}$$

This can be expressed as a **matrix-vector equation:**

More generally, the linear system of $N$ equations in $N$ unknowns: *find* $x_1, x_2, \dots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

This, as a **matrix-vector equation** is:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

So, to proceed, we need to be able to represent **vectors** and **matrices** in our codes.

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART 2**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 3: A vector `class`

Our first focus will be on defining a class of vectors. Intuitively, we know it needs the following components:

Due to the level of detail in the matrix and vector classes, the following example is divided into three source files:

1. `Vector.h`, the header file which contains the class definition. Include this header file in another source file with:
   `#include "Vector.h"`
   Note that this is **not** `<Vector.h>`
2. `Vector.cpp`, which includes the code for the methods in the *`Vector`* class;
3. `01TestVector.cpp`, a test stub.

The test stub can be compiled from the command line with

```
g++ -Wall Vector.cpp 01TestVector.cpp
```

Using `Code::blocks` you need to create a new "project" and include all three source files.

See Vector.h for more details

```
// File:  Vector.h (Version W07.1)
// Author:  Niall Madden (NUI Galway) Niall.Madden@NUIGalway.ie
// Date:  Week 7 of 2021-CS319
// What:  Header file for vector class
// See also:  Vector.cpp and 01TestVector.cpp
class Vector {
private:
  double *entries;
  unsigned int N;
public:
  Vector(unsigned int Size=2);
  ~Vector(void);

  unsigned int size(void) {return N;};
  double geti(unsigned int i);
  void seti(unsigned int i, double x);

  void print(void);
  double norm(void); // Compute the 2-norm of a vector
  void zero(void); // Set entries of vector to zero.
};
```

# Part 3: A vector class



**Vector.cpp**

```cpp
Vector::Vector(unsigned int Size)
{
  N = Size;
  entries = new double[Size];
}

Vector::~Vector()
{
  delete [] entries;
}

void Vector::seti(unsigned int i, double x)
{
  if (i<N)
    entries[i]=x;
  else
    std::cerr << "Vector::seti(): Index out of bounds."
              << std::endl;
}
```

## Vector.cpp continued

```
double Vector::geti(unsigned int i)
{
  if (i<N)
    return(entries[i]);
  else {
    std::cerr << "Vector::geti():␣Index␣out␣of␣bounds."
              << std::endl;
    return(0);
  }
}

void Vector::print(void)
{
  for (unsigned int i=0; i<N; i++)
    std::cout << "[" << entries[i] << "]" << std::endl;
}
```

Vector.cpp continued

```
double Vector::norm(void)
{
  double x=0;
  for (unsigned int i=0; i<N; i++)
    x+=entries[i]*entries[i];
  return (sqrt(x));
}

void Vector::zero(void)
{
  for (unsigned int i=0; i<N; i++)
    entries[i]=0;
}
```

Here is a simple implementation of a function that computes $\mathbf{c} = \alpha \mathbf{a} + \beta \mathbf{b}$

See `01TestVector.cpp` for more details

```
// c = alpha*a + beta*b where a,b are vectors; alpha, beta are scalars
void VecAdd (vector &c, vector &a, vector &b,
       double alpha, double beta)
{
  unsigned int N;
  N = a.size();

  if ( (N != b.size()) )
    std::cerr << "dimension mismatch in VecAdd " << std::endl;
  else
  {
    for (unsigned int i=0; i<N; i++)
      c.seti(i, alpha*a.geti(i)+beta*b.geti(i) );
  }
}
```

## Exercise (7.1)

*The method `Vector::norm()` computes the Euclidian norm of a vector:*

$$\|v\|_2 = \Big(\sum_{i=1}^{n} (v_i)^2\Big)^{1/2}.$$

*This is a special case of the so-called p-norm:*

$$\|v\|_p = \Big(\sum_{i=1}^{n} |v_i|^p\Big)^{1/p}.$$

*where $p \geq 1$. Rewrite the `Vector::norm()` function so that it takes a double `p` as an optional second argument, and computes the p-norm of the vector. If p is not provided, it should default to $p = 2$. In addition, if $p = 0$ is given, it should compute the max-norm:*

$$\|v\|_\infty = \max_{i=1}^{n} |v_i|.$$

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART Part 3**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 4: Solving Linear Systems

We now move towards learning about **matrices**. When implementing the class, we will learn about

- operator overloading;
- `friend` functions and the `this` pointer;
- static variables.
- and much more

One of the most ubiquitous problems in scientific computing is the solution of (large) systems of simultaneous equations. That is, we want to solve a linear system of $N$ equations in $N$ unknowns: *find* $x_1, x_2, \dots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

There are several classic approaches:

1. Gaussian Elimination;
2. Related: $LU$- and Cholesky factorisation;
3. Stationary Iterative schemes such as **Jacobi's method**, **Gauss-Seidel** and Successive Over Relaxation (SOR);
4. Krylov subspace methods, of which Conjugate Gradients is the best known;
5. Enhancements of the Methods 3 and 4, using preconditioning with, for example, MultiGrid and Incomplete $LU$-factorisation.

Of the approaches listed above, Jacobi's is by far the simplest to implement, and so is the one we will study first.

**See annotated slides**.

**See video or annotated slides**

**See video or annotated slides**

**See video or annotated slides**

**See video or annotated slides**

**See video or annotated slides**

Now that we know the method, let us summarise the steps, so as to work out what standard operations on vectors and matrices we need.

We expressed the problem as a matrix-vector equation: *Find* $\mathbf{x}$ *such that*

$$A\mathbf{x} = \mathbf{b},$$

*where* $A$ *is a* $N \times N$ *matrix, and* $\mathbf{b}$ *and* $\mathbf{x}$ *are (column) vector with* $N$ *entries.*

We then derived **Jacobi's method**: choose $\mathbf{x}^{(0)}$ and set

$$x^{(k+1)} = D^{-1}(b + Tx^{(k)}).$$

where $D = \mathrm{diag}(A)$ and $T = D - A$.

Looking at this we see that the fundamental operations are: **vector addition** and **matrix-vector multiplication**.

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART 4**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 5: A matrix `class`

## Part 5: A matrix class

Since we already have Vector class, our next step is to write a class implementation for a matrix, along with the associated functions.

Then we need to define a function to multiply a matrix by vector.

First though, we consider the matrix representation. The most natural approach might seem to be to construct a two dimensional array. This can be done as follows (see Lab 4):

```
double **entries = new double *[N];
for (int i=0; i<N; i++)
   entries[i] = new double N;
```

A simpler, faster approach is to store the $N^2$ entries of the matrix in a single, one-dimensional, array of length $N^2$, and then take care how the access is done:

# Part 5: A matrix class

**Matrix.h**

```
// File:  Matrix.h (W07.1)
// Author:  Niall Madden (NUI Galway) Niall.Madden@NUIGalway.ie
// Date:  Week of 2021-CS319)
// What:  Implementation of "Matrix":  a class of square matrices
// See also:  Matrix.cpp and 02TestMatrix.cpp

class Matrix {
private:
  double *entries;
  unsigned int N;
public:
  Matrix (unsigned int Size=2);
  ~Matrix(void) { delete [] entries; };

  unsigned int size(void) {return (N);};
  double getij (unsigned int i, unsigned int j);
  void setij (unsigned int i, unsigned int j, double x);

  void print(void);
};
```

# Part 5: A matrix class

from Matrix.cpp

```
Matrix::Matrix (unsigned int Size)
{
  N = Size;
  entries = new double [N*N];
}

void Matrix::setij (unsigned int i, unsigned int j, double x)
{
  if (i<N && j<N)
    entries[i*N+j]=x;
  else
    std::cerr << "Matrix::setij(): Index out of bounds."
              << std::endl;
}
```

from Matrix.cpp

```
double Matrix::getij (unsigned int i, unsigned int j)
{
  if (i<N && j<N)
    return(entries[i*N+j]);
  else
  {
    std::cerr << "Matrix::getij(): Index out of bounds."
              << std::endl;
    return(0);
  }
}

void Matrix::print (void)
{
// std::cout << "Matrix is of size " << M << "-by-"
// << N << std::std::endl;
  for (unsigned int i=0; i<N; i++)
  {
    for (unsigned int j=0; j<N; j++)
      std::cout << "[" << entries[i*N+j] << "]";
    std::cout << std::endl;
  }
```

We'll test this by implementing matrix-vector multiplication function:

02TestMatrix.cpp

```
// File:     02TestMatrix.h (Set v=A*u)
// Author:   Niall Madden (Niall.Madden@NUIGalway.ie)
// Date:     Week 7 of 2021-CS319)
// What:     Test the implementation Matrix class
```

```
void MatVec(Matrix &A, Vector &u, Vector &v)
{
  unsigned int N;
  N = A.size();
  if ( (N != u.size()) || ( N != v.size() ) )
    std::cerr << "dimension mismatch in MatVec " << std::endl;
  else
    for (unsigned int i=0; i<N; i++)
    {
      double x=0;
      for (unsigned int j=0; j<N; j++)
        x += A.getij(i,j)*u.geti(j);
      v.seti(i,x);
    }
}
```

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**END OF PART 5**

**CS319 – Week 7**
**Week 7: The Password Problem; Vectors & Matrices**

**Start of ...**

# PART 6: Coding Jacobi's Method

# Part 6: Coding Jacobi's method

Now we can implement Jacobi's method. The specific example coded, we will solve $N = 3$ equations whose matrix representation is

$$9x_1 + 3x_2 + 3x_3 = 15 \quad (1)$$

$$3x_1 + 9x_2 + 3x_3 = 15 \quad (2)$$

$$3x_1 + 3x_2 + 9x_3 = 15 \quad (3)$$

This problem is constructed so that the solution is $x_1 = x_2 = x_3 = 1$.

Have a look at the `main()` function in `03Jacobi.cpp` to see how the problem is set up, and how the Jacobi solver is called. Here we will focus on that solver.

# Part 6: Coding Jacobi's method

See 03Jacobi.cpp for more details

```
// Use Jacobi's method to solve Ax=b,
// On entry :  x is the initial guess
// On exit :  x is the estimate for the solution
void Jacobi(Matrix &A, Vector &b, Vector &x,
            unsigned int &count, double tol)
{
  unsigned int N=A.size();
  count=0;
  if ( (N != b.size()) || (N != x.size() ) )
    std::cout << "Jacobi: error - A must be the same size as b,x"
              << std::endl;
```

# Part 6: Coding Jacobi's method

See 03Jacobi.cpp for more details

```
  Matrix Dinv(N), T(N);    // The diagonal and off-diagonal matrices
  for (unsigned int i=0; i<N; i++)
    for (unsigned int j=0; j<N; j++)
      if (j != i)
      {
        T.setij(i,j, -A.getij(i,j));
        Dinv.setij(i,j,0.0);
      }
      else
      {
        T.setij(i,j, 0.0);
        Dinv.setij(i,j, 1.0/A.getij(i,j));
      }
```

# Part 6: Coding Jacobi's method

See 03Jacobi.cpp for more details

```
// Now implement the algorithm:
Vector d(N), r(N);
do
{
  count++;
  MatVec(T,x,d);         // Set d=T*x
  VecAdd(d, b, d);       // set d=b+d (so d=b+T*x)
  MatVec(Dinv, d, x);  // set x = inverse(D)*(b+T*x)

  MatVec(A, x, r);       // set r=A*x
  VecAdd(r, b, r, 1.0, -1.0); // set r=b-A*r

}   while ( r.norm() > tol);
```

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Of course, the above code would be a lot neater, and much more readable, if we were able to write, for example, `r=A*x` instead of `MatVec(A,x,r)` ....

## Exercise (7.2)

*Write a method `Matrix::norm()` that returns the "Entry-wise" 2-norm of a matrix (also called the* **Frobenius** *or Hilbert–Schmidt norm) :*

$$\|A\|_p = \Big(\sum_{i=1}^{n}\sum_{j=1}^{n} |A_{i,j}|^p\Big)^{1/p}.$$

*and the max-norm:*

$$\|A\|_0 = \max_{i,j} |A_{i,j}|.$$