Mini Project 3

The Conjugate Gradient Method

Author: Dr. Paul Johnson, Paul.Johnson-2@manchester.ac.uk

In this project we shall solve the matrix equation $Ax = b$ where $x$ is the unknown. We shall use an iterative solver, the conjugate gradient method, to solve the equation. Do not worry if you do not follow the theory as the implementation is far more important for the purposes of this course, and by creating vector and matrix classes that can be added/multiplied together, the algorithm should be very easy to implement.

We shall also look at different storage methods for a matrix, and show how the same algorithm can be used to solve for different types of matrix. This is an important component of programming in C++.

3.1 Iterative methods for solving a Matrix equation

3.1.1 The Conjugate Gradient Method

The conjugate gradient method was first put forward by Hestenes and Stiefel (1952) and was reinvented again and popularised in the 1970’s. The method has one important property that it converges to the exact solution of the linear system in a finite number of iterations, in the absence of roundoff error. Used on its own the method has a number of drawbacks, but combined with a good preconditioner it is an effective tool for solving linear equations iteratively. Consider

$$Ax = b;$$

where $A$ is an $n \times n$ symmetric and positive definite matrix. Recall $A$ is symmetric if $A^T = A$ and positive definite if $x^TAx > 0$ for any non zero vector $x$.

Let us define $\langle x, y \rangle$ to be the dot product of two vectors. Now consider the quadratic functional

$$\phi(y) = \langle Ay, y \rangle - \langle b, y \rangle - \langle y, b \rangle.$$
Then for a vector $p$ and $x$ and any scalar $\alpha$

\[
\phi(x + \alpha p) = \langle A(x + \alpha p), x + \alpha p \rangle - \langle b, x + \alpha p \rangle - \langle x + \alpha p, b \rangle
\]

\[
= \phi(x) + \langle Ax - b, \alpha p \rangle + \langle \alpha p, Ax - b \rangle + \alpha^2 \langle Ap, p \rangle,
\]

(3.3)

Differentiating (3.3) with respect to $\alpha$ and equating to zero to find a minimum gives

\[
0 = \langle Ax - b, p \rangle + \langle p, Ax - b \rangle + 2\alpha \langle Ap, p \rangle,
\]

which gives

\[
\alpha = \frac{\langle b - Ax, p \rangle}{\langle Ap, p \rangle},
\]

(3.4)

since $\langle x, y \rangle = \langle y, x \rangle$. Then the functional $\phi(x + \alpha p)$ achieves its minimum at

\[
\phi(x + \alpha p) = \phi(x) - \frac{\langle b - Ax, p \rangle^2}{\langle Ap, p \rangle}.
\]

(3.5)

This proves that $x$ is a solution of $Ax = b$ if and only if $x$ minimizes the functional $\phi(y)$. The proof is as follows. If $x$ is a minimizer of $\phi(y)$, then $Ax = b$, since otherwise there would exist $y = x + \alpha p$ for $\alpha \neq 0$ and $p \neq 0$ such that $\phi(y) \leq \phi(x)$ and $\phi(y)$ would not be a minimum. Conversely, if $Ax = b$, then $\phi(x) \leq \phi(x + \alpha p)$ for any $\alpha$ and $p$, so then $x$ is a minimizer of $\phi(y)$. So solving $Ax = b$ is the same as minimizing $\phi(y)$.

This suggests an iterative algorithm:

\[
x_{k+1} = x_k + \alpha_k p_k,
\]

with

\[
\alpha_k = \frac{\langle b - Ax_k, p_k \rangle}{\langle Ap_k, p_k \rangle}
\]

(3.6)

where $p_k$ is some vector that we must generate.

Now assume that the vectors $p_0, p_1, p_2, \ldots, p_n$ that we generate are conjugate with respect to the matrix $A$, which is defined as

\[
\langle Ap_i, p_j \rangle = 0, \quad \text{for} \quad i \neq j;
\]

(3.7)

then the iterates $x_0, x_1, x_2, \ldots, x_n$, generates by the iterative process (3.6) satisfy that $Ax = b$ for an arbitrary initial point $x_0$.

The important thing to note here is that if this theory holds we have a finite limit $n$ on the number of iterations this scheme will take to converge. The convergence rate will depend on the condition of the matrix. In order to speed up convergence, a preconditioner may be used, but that is not investigated in this task.

### 3.1.2 Formulation

Let us assume that we choose conjugate vectors $p_k$ as

\[
p_{k+1} = r_{k+1} + \beta_k p_k,
\]

(3.8)

where $\beta_k$ is given by the formula

\[
\beta_k = -\frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}
\]

(3.9)
Now we have the conjugate vectors, we can make some simplifications, namely that

$$\langle b - Ax_k, p_k \rangle = \langle r_k, r_k \rangle$$

Then following the theory the conjugate gradient algorithm may be written as follows:

**Algorithm - Conjugate Gradient method:**

- Initialise the method with a guess $x_0$. Then set
  
  $$r_0 = b - Ax_0,$$
  $$p_0 = r_0,$$

- Now construct a loop to iterate through $x_k$, for $k = 0, 1, 2, \ldots$,

  $$\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle Ap_k, p_k \rangle},$$
  $$x_{k+1} = x_k + \alpha_k p_k,$$
  $$r_{k+1} = r_k - \alpha_k Ap_k,$$
  $$\beta_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle},$$
  $$p_{k+1} = r_{k+1} + \beta_k p_k.$$

3.2 Coding, Examples and Exercises

3.2.1 Creating a Math vector from the standard library

Here we shall use the standard vector class to create a new vector class so that we can add them, and multiply them by scalars. Putting extra work into making this class will enable our conjugate gradient method to be written as we would write it on paper.

Copy the following class definition for the new class `MVector` into a header file or at the top of your main code.

```cpp
// class MVector contains arrays that can work with doubles
class MVector
{
    // storage for the new vector class
    vector<double> v;

    public:
        // constructor
        explicit MVector() {}
        explicit MVector(int n):v(n){}
        explicit MVector(int n,double x):v(n,x){}
        // equate vectors;
        MVector& operator=(const MVector& X)
        {if(&X==this)return *this;v=X.v;return *this;}
        // access data in vector
```
```cpp
double& operator[](int index) { return v[index]; }
// access data in vector (const)
double operator[](int index) const { return v[index]; }
// size of vector
int size() const { return v.size(); }
}; // end class MVector
```

So far so good. The class `MVector` will act in exactly the same way as a `std::vector`, except that we do not have access to all the public functions of the `std::vector`, and we have explicitly chosen the data `double` as the data type stored in the array.

Now for this to be of any use we must overload the operators `+-/*` to work with our new `MVector` class. We shall place the function definition `outside` the class definition but inside the header file. A typical definition will look like

```cpp
// scalar mult vector
MVector operator*(const double& lhs, const MVector& rhs);
```

and the implementation can be placed in a different file

```cpp
MVector operator*(const double& lhs, const MVector& rhs) {
    MVector temp(rhs);
    for (int i = 0; i < temp.size(); i++) temp[i] *= lhs;
    return temp;
}
```

**Tasks:**

1. There are five operators we need. Remember that we can multiply/divide vector by a scalar, add/subtract vectors, but can’t add/subtract a scalar to a vector. What are the five operators that we need?

2. Add the function definitions and implementations into your code.

3. Check that the code is working by evaluating the following using `MVectors` to represent `u`, `v`, `w` and `x`:

   \[ u = 4.7v + 1.3w - 6.7x \]

   where \( v = (0.1, 4.8, 3.7) \), \( w = (3.1, 8.5, 3.6) \) and \( x = (5.8, 7.4, 12.4) \).

4. Try other combinations additions/multiplications and see what happens. What happens when you try to add a double to a vector? What happens if you try again but remove the explicit keyword from the constructors?

5. When adding two vectors check they conform and exit with an error if they do not.
Additional Tasks:

1. Try adding a function to resize the vector using the `resize(int n, double x)` member function of the standard vector class on `v`. Within your function, you should call the standard `clear()` member function before the `resize` member function.

2. You could also try overloading the `<<` operator to output a vector in the form `(v[0], v[1], ..., v[n])`.

3.2.2 More Useful Vector Functions

When working with vectors it will useful to specify a further two member functions to calculate the norm of a vector. We wish to implement the following two norms, the infinity norm

\[ ||x||_\infty = \max(|x_0|, |x_1|, \ldots, |x_n|), \]  

(3.16)

and the \(L^2\) norm

\[ ||x||_2 = \sqrt{x_0^2 + x_1^2 + \cdots + x_n^2}. \]  

(3.17)

Also we would like a function to calculate the dot product of two vectors. It may be defined as

\[ x \cdot y = \sum_{i=0}^{n} x_i y_i. \]  

(3.18)

Example:

Add a member function definition

```cpp
double maxNorm() const;
```

to the `MVector` class that returns the maximum absolute element in the vector. Now write the implementation into your code. You may use the `std::max` function. Remember that the `std::abs` function requires the `<cmath>` library.

Solution:

The function may be written as follows:

```cpp
double maxNorm() const
{
    // calculate the maximum value
    // initialise temp with first element
    double temp = std::abs(v[0]);
    // set temp to be the maximum of temp and the next element
    for(int i=1; i<size(); i++)
    {
        temp = std::max(std::abs(v[i]), temp);
    }
    return temp;
}
```
Tasks:

1. Add the source code above to your program and check that it compiles and runs.

2. Write your own member function to evaluate the \( L^2 \) norm.

3. Add a function outside the class, with the definition:

   ```
   double dot(const MVector& lhs, const MVector& rhs);
   ```

   Fill in the implementation for the function.

4. Test your dot product function by evaluating

   \[ \alpha = \frac{u \cdot u}{v \cdot w} \]

   where \( u = (1.5, 1.3, 2.8) \), \( v = (6.5, 2.7, 2.9) \) and \( w = (0.1, -7.2, 3.4) \). The value of \( \alpha \) should be -1.31915 (to 6 s.f.).

5. Test your \( L^2 \) norm function on the three vectors \( ||u||_2 = 3.4322 \), \( ||v||_2 = 7.61249 \), and \( ||w||_2 = 7.96304 \).

3.2.3 Creating a Matrix class

Now in order to solve the matrix equation we shall need a matrix class, as given below. For the time being do not worry about the way the data is stored, we have used a \texttt{vector} of a \texttt{vector}, storage issues and data access functions are taken care of with the definition.

```c++
// class Matrix is an array that has
// matrix vector multiply function
class Matrix {
  unsigned int N,M;
  // storage for the new matrix class
  vector<vector<double>> A;

public:
  // constructor
  explicit Matrix() : N(0), M(0) {} 
  explicit Matrix(int n, int m) : N(n), M(m), A(n, vector<double>(m)) {}
  explicit Matrix(int n, int m, double x) : N(n), M(m), A(n, vector<double>(m, x)) {}
  // equate matrices;
  Matrix& operator=(const Matrix& rhs) {
    if (&rhs == this) return *this; A = rhs.A; return *this;
  }
  // set all value equal to a double
  Matrix& operator=(double x) {
    for (int i=0; i<rows(); i++) for (int j=0; j<cols(); j++) A[i][j] = x;
    return *this;
  }
};
```
// access data in matrix (const)
double operator()(int i, int j) const { return A[i][j]; }

// access data in matrix
double& operator()(int i, int j){ return A[i][j]; }

// size of matrix
int rows() const { return N; }
int cols() const { return M; }
}; // end class Matrix

Please note that to access elements in the matrix we now use (i, j) rather than [i][j]. The matrix class as listed here is almost complete. All that is left to do is write a function to implement \( \text{Matrix} \times \text{MVector} \). The definition is as follows:

\[
\text{MVector \ operator\ast(\text{const Matrix\& A, const MVector\& x});}
\]

**Tasks:**

1. Declare a new 4 \( \times \) 3 matrix in your code. Now assign values to the matrix (use (i, j) to access elements) and print them out to the screen.

2. Add the definition for \( \text{Matrix} \times \text{MVector} \) multiplication into your code. Complete the implementation of the function.

3. Let \( A = a_{i,j} \) be a 4 \( \times \) 3 matrix, such that \( a_{i,j} = 3 \ast i + j \) for \( 0 \leq i \leq 3 \) and \( 0 \leq j \leq 2 \) and let \( \mathbf{x} = (0.5, 1.6, 3.2) \). Then test your function by evaluating

\[
b = Ax
\]

Check your solution for \( b \) by using hand or by using matlab.

**Additional Tasks:**

1. Create a resize function for \( \text{MMatrix} \) as you have done with the \( \text{MVector} \) class. The definition should be something like `resize(int n, int m, double x)`, you **must** use a `clear()` function on \( A \) before resizing. See the constructors in the class for the syntax on how to resize.

2. You could try overloading the `<<` operator to output the matrix in the form

\[
(A[0][0], A[0][1], ..., A[0][m]) \\
(A[1][0], A[1][1], ..., A[1][m]) \\
\vdots \\
(A[n][0], A[n][1], ..., A[n][m])
\]
3.2.4 Implementing the Conjugate Gradient method

Let $A$ be an $n \times n$ matrix and $x$, $b$ are $n$ element vectors. Then solve

$$Ax = b.$$ 

The particular problem we shall solve here is derived from a one dimensional finite difference scheme, but we shall just present the matrix itself without derivation. The matrix may be written:

$$A = a_{i,j} = \begin{cases} 
2 & \text{if } i = j \\
-1 & \text{if } |i - j| = 1 \\
0 & \text{otherwise}
\end{cases}, \quad (3.19)$$

and the vector $b = b_i = \frac{2}{(n+1)^2}$. We can use $x_i = 0$ as our initial guess.

Tasks:

1. Declare a matrix $A$ in your code with $n = 5$ rows and $n = 5$ columns.
2. Assign each element in the matrix a value according to (3.19).
3. Print out your matrix to the screen or a file and check that it is properly assigned.
4. Calculate the residual $r = b - Ax$.

Now we must implement the conjugate gradient algorithm as stated in (3.10) and (3.11–3.15). Your code will look something like:

```cpp
// initialise vectors
// fill this in...
for (iter = 0; iter < maxiter; iter++)
{
    // calculate new values for x and r
    // fill this in...
    // check if solution is satisfied
    if (r.l2norm() < tolerance) break;
    // calculate new conjugate vector p
    // fill this in...
}
```

Tasks:

1. Implement the conjugate gradient (CG) method, with $maxiter = 1000$ and $tolerance = 10^{-6}$. With $n = 25$ the method should take 12 iterations to converge.
2. Plot $x_i$ versus $i$ for $n = 10, 25, 100$. Try plotting $x_i$ against $\frac{i+1}{n+1}$ where $i$ runs between 0 and $n - 1$ so that they lie on top of each other. What do you notice?
3. Alter the code to output the number of iterations needed for convergence. Create a table showing the number of elements \( n \) against the iterations for convergence. Also add a column showing the computation time (see website for more information).

4. Change the matrix \( A \) to be

\[
A = a_{i,j} = \begin{cases} 
2(i + 1)^2 + m & \text{if } i = j \\
-(i + 1)^2 & \text{if } |i - j| = 1 \\
0 & \text{otherwise}
\end{cases}
\]  

(3.20)

with the vector \( b = b_i = 2.5 \) and \( m = 10 \). and rerun your results. What happens if you vary the value of \( m \), particularly if \( m \) is small or even 0? Can you find out (from external sources) why this might happen?

Report:
- Include annotated code of the MVector and Matrix classes, and your CG implementation
- Include the figure of \( x_i \) against \( i \) for different values of \( n \).
- include the table for convergence
- Include a discussion of the results, and what happens when \( A \) is changed.

3.2.5 A New Type of Matrix

You may notice when printing out the matrix \( A \) from above that most of the elements in the matrix, especially as \( n \) gets large, are zero. A more efficient way of storing the matrix would be to just store the diagonals, rather than the whole matrix. We call this a **banded** matrix.

Assume \( B \) is a \( n \times n \) banded matrix. Now let \( l \) be the number of non-zero diagonal to the left of center, and \( r \) the number of non-zero diagonals to the right. Then the size of the storage will have to be \( n \times (l + r + 1) \). If for instance \( A \) 5 \( \times \) 5 matrix with 4 bands is given by

\[
A = \begin{bmatrix}
a_{0,0} & a_{0,1} & a_{0,2} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\
a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,2} & a_{3,3} & a_{3,4} \\
a_{4,3} & a_{4,4}
\end{bmatrix} = \begin{bmatrix}
1 & 6 & 10 \\
13 & 2 & 0 & 11 \\
14 & 3 & 8 & 12 \\
0 & 4 & 9 \\
16 & 5
\end{bmatrix}
\]

then can express the matrix \( B \) as a 5 \( \times \) 4 matrix

\[
B = \begin{bmatrix}
{b_{0,0}} & b_{0,1} & b_{0,2} & b_{0,3} \\
{b_{1,0}} & b_{1,1} & b_{1,2} & b_{1,3} \\
{b_{2,0}} & b_{2,1} & b_{2,2} & b_{2,3} \\
{b_{3,0}} & b_{3,1} & b_{3,2} & b_{3,3} \\
{b_{4,0}} & b_{4,1} & b_{4,2} & b_{4,3}
\end{bmatrix} = \begin{bmatrix}
1 & 6 & 10 \\
13 & 2 & 0 & 11 \\
14 & 3 & 8 & 12 \\
0 & 4 & 9 \\
16 & 5
\end{bmatrix} .
\]
If we recall that \( l \) is the number of diagonals left of the center we can express the relation between the matrices as

\[
a_{i,j} = b_{i,j+l-i}.
\]

In the following class definition we define such a banded matrix:

```cpp
// class Banded stores only elements in the diagonals
class Banded
{
    unsigned int N,M;
    // storage for the banded matrix
    vector<vector<double>> A;
    int l,r; // number of left/right diagonals
public:
    // constructor
    explicit Banded():N(0),M(0){}
    explicit Banded(int n,int m,int lband,int rband)
    :N(n),M(m),A(n,vector<double>(lband+rband+1)),l(lband),r(rband){}
    explicit Banded(int n,int m,int lband,int rband,double x)
    :N(n),M(m),A(n,vector<double>(lband+rband+1,x)),l(lband),r(rband){}
    // equate matrices;
    Banded& operator=(const Banded& rhs)
    {
        N=rhs.N;M=rhs.M;A=rhs.A;l=rhs.lband();r=rhs.rband();return *this;
    }
    // access data in Banded matrix (const)
    double operator()(int i,int j) const {/* fill this in */}
    // access data in Banded matrix
    double& operator()(int i,int j){/* fill this in */}
    // size of vector
    int rows() const {return N;};
    int cols() const {return M;};
    // total number of bands
    int bands() const {return r+l+1;}
    int lband() const {return l;} // number of left bands
    int rband() const {return r;} // number of right bands
}; // end class Banded
```

Example:

- Overload the `<<` operator to output the matrix.

Solution:

First define the operator as follows:

```cpp
std::ostream& operator<<(std::ostream& output,const Banded& banded)
```

Then the function implementation will be

```cpp
// overload output streams
std::ostream& operator<<(std::ostream& output,const Banded& banded)
{
```
for (int i = 0; i < banded.rows(); i++)
{
    output << "(";
    // calculate position of lower and upper band
    int jmin = std::max(std::min(i - banded.lband(), banded.cols()), 0);
    int jmax = std::min(i + banded.rband() + 1, banded.cols());
    for (int j = 0; j < jmin; j++) output << 0 << "\t";
    for (int j = jmin; j < jmax; j++)
        output << banded(i, j) << "\t";
    for (int j = jmax; j < banded.cols(); j++)
        output << 0 << "\t";
    output << ")\n";
}
return output;

Tasks:

1. Copy the code for the banded matrix along with the function to output the matrix to the screen. Now fill in the access functions to elements in the matrix. Think carefully about how to do this. If you have added any range-checking state why you have done this. Assign some values to a matrix and check they have assigned correctly by printing to the screen.

2. Now make the matrix $A$ from (3.19) a banded matrix, we will have that $l = 1$ and $r = 1$. Setup the matrix, and print to screen to check it has worked.

3. Create a function with the definition

   ```cpp
   MVector operator*(const Banded A, const MVector& x) const;
   ```

   to implement matrix vector multiplication for $Banded$ matrices. Complete the definition and make sure that it is efficient (i.e. doesn’t multiply by zero entries).

4. Using the values for $A$, $x$ and $b$ from §3.2.4, calculate the residual $r$. Check your answer against previous values.

5. Now run the (CG) algorithm on the banded matrix. Note here that because we have overloaded the operators you should not need to change anything in the code.

6. Create a table to show the number of iterations and computation time for different values of $n$.

Report:

- Include annotated code of the Banded class.
- Include the table showing number of iterations and computation time for the two matrix classes.
Compare and contrast the different methods for storing a matrix.

### 3.3 Huge Systems and Sparse Matrices

**Case 1:**

Let \( u \) be the solution to the equation

\[
\nabla^2 u(s, t) = 2.
\]

Now define the \( n^2 \) vector \( x \) as an approximation of \( u \) such that

\[
x = x_{k+nl} \approx u(s_k, t_l)
\]

Then \( x \) is an approximation to (3.21) if it solves the matrix equation

\[
Ax = b,
\]

where \( A \) is an \( n^2 \times n^2 \) matrix, \( x \) and \( b \) are \( n^2 \) element vectors, and the matrix \( A \) may be written:

\[
A = a_{i,j} = \begin{cases} 
4 & \text{if } i = j \\
-1 & \text{if } |i - j| = n \\
-1 & \text{if } \{|i - j| = 1 \text{ and } (i + j) \mod (2n) \neq 2n - 1\} \\
0 & \text{otherwise}
\end{cases}
\]

The vector \( b = b_i = \frac{2}{(n+1)^2} \) and \( x = x_i = 0 \) for our initial guess.

**Case 2:**

Let the matrix \( A \) be written as:

\[
a_{i,j} = \begin{cases} 
\{2(\lfloor i/n \rfloor + 1)^2 + 2(\lfloor i/n \rfloor + 1)^2 + m\} & \text{if } i = j \\
-(\lfloor i \mod n \rfloor + 1)^2 & \text{if } |i - j| = n \\
-(\lfloor i/n \rfloor + 1)^2 & \text{if } \{|i - j| = 1 \text{ and } (i + j) \mod (2n) \neq 2n - 1\} \\
0 & \text{otherwise}
\end{cases}
\]

where \( \lfloor i/n \rfloor \) represents integer division and is a whole number. We also have \( b = b_i = 2.5, m = 10 \) and \( x = x_i = 0 \) as our initial guess. Try varying the value of \( m \) again and see what happens.

You should note that these matrices are incredibly sparse.
Tasks:

1. Input the matrix $A$ for case 1 into a $Matrix$ class, with $n = 5$. Print out to screen to check that it has been entered correctly. You may ask to see a solution to check against your values. Solve the problem using $n = 5$ with your CG solver.

2. Input the matrix $A$ for case 1 into a $Banded$ class, with $n = 5$. Print out to screen to check that it has been entered correctly. Note here that you are still storing a lot of zero entries. You can check against your print out from the $Matrix$ class. Solve the problem using $n = 5$ with your CG solver.

3. Now create a new $SparseMatrix$ class, that stores only the non-zero diagonals (in section 3 you will need just 5 columns for the non-zero diagonals). Use the storage classes that you have already been given as a guide. Define and implement a matrix vector multiplication function for the new class.

4. Input the matrix $A$ for case 1 into a $SparseMatrix$ class, with $n = 5$. Print out to screen to check that it has been entered correctly.

5. Now solve case 1 with all your available storage classes for values of $n$ up to 100 or more. Create a table or figure showing iterations to converge and computation times for each class.

   WARNING Before running calculations on large matrices make sure you have optimised your programs before running them. If you cannot find a value state this in your report and try to explain why you can’t.

6. Try plotting the results in 3D by outputting your data
   - in columns (for gnuplot)
     \[
     i \quad j \quad x_{in+j}
     \]
     for $0 < i < n$ and $0 < j < n$.
   - in a matrix format (for matlab)
     \[
     \begin{array}{cccc}
     x_0 & x_1 & \ldots & x_{n-1} \\
     x_n & x_{n+1} & \ldots & x_{2n-1} \\
     \vdots & \vdots & \ldots & \vdots \\
     x_{n(n-1)} & x_{n(n-1)+1} & \ldots & x_{n^2-1}
     \end{array}
     \]

7. Repeat the analysis for case 2.

3.4 Report

Write a project report as a connected piece of prose. You may use any suitable reference sources, but these must be clearly identified in the report at the points of use.
1. State a brief theory behind the conjugate gradient method.

2. Give a detailed description of any classes used, alongside code listings, describing what they are used for and what benefits they bring.

3. Include tables or figures of convergence and computation times for the problems stated in the report.

4. Try to explain why certain problems are more difficult (or impossible) to solve.

5. Include some 3D plots of the solutions in section 3.

6. Compare and contrast your three matrix storage classes for the problems stated in section 3.

Marks will be awarded for clarity and correctness of code as well as answers to the questions and discussion.
Sorting and/or searching through data is something that is extremely common when programming. It is an essential part of many algorithms, but the basic sorting problem is interesting in its own right. In this project, we shall implement some standard sorting algorithms with increasing levels of abstraction. The sorting methods will be used to implement the game of life using very little memory. It will be assumed that students are familiar with objects, inheritance and can output data to files.

4.1 Sorting and searching

4.1.1 The sorting problem

The sorting problem is to find a permutation $\pi$ of a sequence of $n$ elements $a_0, a_1, \ldots, a_{n-1}$ such that $a_{\pi(0)} \leq a_{\pi(1)} \leq \cdots \leq a_{\pi(n-1)}$, where $\leq$ is a suitably defined partial order. Note that the indexing is from 0 to be consistent with standard C++ convention. For our purposes, a partial order is defined as a relation $\leq$ on a set $S$, such that for $a, b, c \in S$:

- $\leq$ is reflexive — $a \leq a$ is true.
- $\leq$ is transitive — $a \leq b$ and $b \leq c$ $\Rightarrow$ $a \leq c$.
- $\leq$ is antisymmetric — $a \leq b$ and $b \leq a$ $\Rightarrow$ $a = b$.

An important point is that we must be able to define such a relation if it is to be possible to sort our set.

4.1.2 Bubblesort

Bubblesort is very simple sorting algorithm. The idea is to move the greatest element to the $n$-th position in the sequence and then to move the greatest element in the subsequence $a_1, \cdots, a_{n-1}$ to the $n-1$-th position and so on. In each (sub)sequence, the method used to move elements is
to compare two neighbouring elements \( a_i \) and \( a_{i+1} \) and interchange them if \( a_{i+1} < a_i \). Assuming that the sequence is stored in an MVector, see §4.2.1, the algorithm used to move the greatest element to the final position of a sequence of length \( n \) would be

\[
\text{for}(i=0; i<n-1; ++i) \\
\{ \\
\quad \text{if}(a[i+1] < a[i]) \{ a\text{.swap}(i, i+1); \} \\
\}
\]

where \( \text{swap}(i, i+1) \) is a member function of the MVector class that exchanges the entries \( a[i] \) and \( a[i+1] \).

### 4.1.3 Quicksort

Quicksort is an alternative sorting algorithm that runs much more quickly than bubble sort in most cases and can be defined recursively. The idea is to choose a random element from the sequence, suppose it has the value \( x \). The sequence is then divided into three subsequences

- \( S_1 \): all elements with values less than \( x \).
- \( S_2 \): all elements with values equal to \( x \).
- \( S_3 \): all elements with values greater than \( x \).

The quicksort algorithm is then applied to the subsequences \( S_1 \) and \( S_3 \) and the sorted output consists of (sorted) \( S_1 \) followed by \( S_2 \) followed by (sorted) \( S_3 \). When writing recursive algorithms it is extremely important to have a termination criterion (otherwise the algorithm will continue forever). The termination criterion in this case is that if the sequence contains one or no elements there is no sorting to be done, so it just returns the sequence.

### 4.1.4 Heapsort

A heap is a special type of binary tree in which the value at each vertex is always greater than or equal to the values at its sons, see Figure 4.1(b). Arranging a sequence into a heap allows a particularly fast sort to be performed.

#### Binary Trees

A binary tree is a particular type of directed, acyclic, graph that is a very convenient data structure, see [http://en.wikipedia.org/wiki/Binary_tree](http://en.wikipedia.org/wiki/Binary_tree) or many textbooks for full details. The easiest way to understand a tree is to look at a picture and Figure 4.1(a) shows a simple binary tree.
Figure 4.1: (a) a binary tree with three levels: the root is the node labelled 0 with two sons 1 and 2. Each son has two sons of its own 3 and 4 and 5 and 6 and these are the leaves. (b) the tree rearranged into a heap in which the value at a vertex is always greater than or equal to to values of its sons.

Making a heap from a tree

A recursive algorithm for creating a heap from a tree is relatively easy to construct. Consider a tree consisting of a root with only two sons: if either of the values at the sons is greater than the value at the root, then exchanging that value with the value at the root will make a local heap. For a general tree, we apply this algorithm to every vertex of the tree starting from the bottom and working upwards. (We cannot apply the algorithm if a vertex has no sons, so we actually start from the penultimate row.) If an exchange takes place then we must check that the lower portion of the tree remains a heap. Thus, we apply the algorithm recursively to the (sub)tree with the recently-exchanged son as the root.

The algorithm applied to the tree in Figure 4.1(a) yields the heap in Figure 4.1(b). We shall illustrate the algorithm by representing the tree in a linear array reading across each row in turn; the initial configuration is 0123456 and the final configuration (heap) is 6453102.

- **Vertex** 2: greatest-son-value is 6, exchange 2 and 6 to give 0163452.
- **Vertex** 1: greatest-son-value is 4, exchange 1 and 4 to give 0463152.
- **Vertex** 0: greatest-son-value is 6, exchange 0 and 6 to give 6403152.
- **Apply algorithm recursively to exchanged son**
  - **Vertex** 0: greatest-son-value is 5, exchange 0 and 5 to give 6453102

To understand this algorithm it really helps to draw your own tree and draw out the moves made to generate the heap.
Sorting the heap

Once we have a heap then we can easily find the element with greatest value ... it's at the top of the heap! The idea of heapsort is to remove this top value and replace it by the value of one of the leaves and then apply the recursive heap-generation algorithm to the root, which move the element with next greatest value to the top of the heap and so on and so on. Thus we build up the sorted data by removing the top of the heap each time.

4.2 Coding, Examples and Exercises

Initially, we shall write our algorithms to sort the data in a vector class

4.2.1 Creating our own vector class from the standard library

You should use the following class definition for the new class $MVector$ in a header file or at the top of your main code.

```cpp
// class MVector contains arrays that can work with doubles
class MVector
{
    // storage for the new vector class
    vector<double> v;

    public:
        // constructor
        explicit MVector();
        explicit MVector(int n); v(n);
        explicit MVector(int n, double x); v(n, x);
        // equate vectors;
        MVector& operator=(const MVector& X);
        {if(&X==this)return *this; v=X.v; return *this;}
        // access data in vector
        double& operator[](int index){return v[index];}
        // access data in vector (const)
        double operator[](int index) const {return v[index];}
        // size of vector
        int size() const {return v.size();}
}; // end class MVector
```

The class $MVector$ should act in exactly the same way as a $std :: vector$, except that we do not have access to all the public functions of the $std :: vector$, and we have explicitly chosen the data $double$ as the data type stored in the array. N.B. You will have to include the $<vector>$ header in order to use the standard vector class.

Tasks:

1. Write a very simple main program that creates an $MVector$ of size 10, stores whatever data you like and outputs that data to a file.
2. Add range-checking errors to your square-bracket [ ] access functions so that if the index is out of range then the program exits with an error. Test that the range-checking works! N.B. This can save your life when developing code, but will make your code run more slowly, so you should remove it in final “production runs”.

3. Add an additional member function to the MVector class

```cpp
void MVector::swap(int i, int j);
```

that interchanges the values of the i-th and j-th entries of the MVector.

4. If you like overload the << operator to produce pretty output of the vector in the form (v[0], v[1],..., v[n-1]).

### 4.2.2 Creating initial data

**Example:**

Add a member function

```cpp
void MVector::initialise_random(double xmin, double xmax);
```

to the MVector class that assign a random (double) value, between the limits xmin and xmax, to each entry in the vector. You may use the C++ `rand()` function defined in the `<cstdlib>` library.

**Solution:**

The random number generator requires an initial seed, set using the function `std::srand(...)`. If this is not set then the same pseudo-random sequence will be generated for every vector, so the initial data won’t be very random at all! We shall modify the constructor of the MVector to set the seed using the system time, which requires the inclusion of the `<time.h>` header. Thus, we need to include two headers at the top of our code.

```cpp
#include <time.h>
#include <cstdlib>
```

The constructor is modified as follows

```cpp
explicit MVector(int n) : v(n) {
    srand(static_cast<unsigned>(time(NULL)));}
```

We then fill in the random entries using the `rand()` function. The standard random number generator returns an integer between 0 and `RAND_MAX`, so we scale this value to lie within our desired range.

```cpp
void MVector::initialise_random(double xmin, double xmax) {
    const unsigned n = this->size();
    for(unsigned i=0; i<n; i++)
```
{   //Scale the random number to lie between xmin and xmax
   v[i] = xmin + rand()*(xmax - xmin) / static_cast<double>(RAND_MAX);
}

Tasks:

1. Add the source code above to your program and make sure that it compiles and runs.

2. Make sure that you modify all constructors to set the seed of the random number generator.

3. Check the functionality by generating a number of vectors of varying lengths and fill them with random initial values. How can you test the randomness of the entries? (This is a hard question that you may like to think about further).

4.2.3 Implementing the sorting algorithms

Bubblesort

Tasks:

1. Add a function to your program in a (global) namespace called Sort

```cpp
namespace Sort
{
    void bubble(MVector &v);
}
```

that implements the bubblesort algorithm described in §4.1.2. The function should return the sorted data in the `MVector` passed into the function, i.e. the code

```cpp
MVector v(3);
v[0] = 5.5; v[1] = 2.0; v[2] = 1.0;
Sort::bubble(v);
std::cout << v;
```

should produce the output

```
(1.0, 2.0, 5.5)
```

assuming that you have overloaded the `<<` operator; if not, why not do it now?

2. Write a program that computes the average sort time for a randomly-initialised vector of length n. You might want to use the following function to make your timings:
//Return number of seconds since the program started running
double timer()
{
    time_t t = clock();
    return static_cast<double>(t) /
           static_cast<double>(CLOCKS_PER_SEC);
}

Report:
Your report for this section should contain
- A hard-copy of your function bubble(...).
- A graph of the average sort time as a function of $n$. Can you deduce the functional form of the curve?
- An analysis of the expected form of the curve from theoretical considerations, i.e. what is the complexity of the algorithm?

Quicksort
There are two subtleties here: 1) How does one choose the “random” value in the sequence? 2) How does one divide the sequence into the three subsequences? For the first of these, you could always use the random number generator used to generate the random initial data. The second requires a bit more thought. If you think about it carefully you will find that you can divide the sequence “in place” (i.e. without using an significant extra memory) by using a series of carefully chosen exchanges. You will need to keep track of the starts and ends of each subsequence. That said, it is always better to first write a version that works and then think about how to make it more efficient. You may find that you need to add additional functions to the $MVector$ class.

Tasks:
1. Add two functions to the $Sort$ namespace

```cpp
namespace Sort
{
    void quick_recursive(MVector &v, int start, int end);
    void quick(MVector &v) {quick_recursive(v, 0, v.size() - 1);}
}
```

to implement the quicksort algorithm described in §4.1.3. The `quick_recursive(...)` function should perform a quicksort on the section of the $MVector$ between the indices `start` and `end`. The function `quick(...)` is a simple “wrapper” to the recursive function that does the actual work. Be very careful when implementing the recursive algorithm to ensure that it has an appropriate termination criterion.
2. Write a program that calculates the average sort time for a randomly-initialised vector of length \( n \).

**Report:**

Your report for this section should contain

- A hard-copy of your function `quick_recursive(...)`. 
- A graph of the average sort time as a function of \( n \). Can you deduce the functional form of the curve?
- An analysis of the expected form of the curve from theoretical considerations, \( i.e. \) what is the complexity of the algorithm?

**Heapsort**

Heapsort requires some thought about how to represent the heap structure. One way, not the only way, is to store the heap in a vector-like structure with the indexing as shown in Figure 4.1(a). In other words, the root is \( h[0] \) and the sons of the vertex \( h[i] \) are located at \( h[2 \times i + 1] \) and \( h[2 \times i + 2] \). Draw lots of pictures and convince yourself that this scheme really does work!

**Tasks:**

1. Using the heap-storage scheme suggested above, find the criterion that determines whether a vertex is a leaf (\( i.e. \) has no sons)?

2. Add two functions to the `Sort` namespace

```cpp
namespace Sort {
    void heap_from_root(MVector &v, int i, int n);
    void heap(MVector &v);
}
```

The first of these should make a heap with the \( i \)-th vertex as the root, assuming that only the first \( n \) values stored in the vector make up the heap data. The second should implement the heapsort algorithm described in §4.1.4. It should use the function `heap_from_root(...)` to build the heap and then to perform the sorting. If you are careful, you should be able to perform the sorting efficiently without significant additional memory. Once again, the most effective way to write code is to get the thing working first and then worry about efficiency.

3. Write a program that calculates the average sort time for a randomly-initialised vector of length \( n \).
Report:

Your report for this section should contain

- A hard-copy of your functions `heap_from_root(...)` and `heap(...)`. 
- A graph of the average sort time as a function of $n$. Can you deduce the functional form of the curve?
- An analysis of the expected form of the curve from theoretical considerations, i.e. what is the complexity of the algorithm?

4.2.4 Abstracting the sorting algorithms

We have now implemented sorting algorithms for `MVector` objects, but we can use OO techniques to write much more general algorithms to sort any objects with certain properties. Have a careful look at your algorithms and think about the operations used. You should find that you can write all the algorithms using only three operations on the data stored in the `MVector`: `size()`, `swap(i,j)` and `cmp(i,j)`, where `cmp(i,j)` is a comparison operator, e.g. `<`. You do not need an equality operator.

Tasks:

1. Work out how to write a test for the equality of two elements in the vector using only the comparison operator.

2. Copy your code into a new file, keeping a backup of the previous version.

3. In the new file, add a member function

   ```
   bool MVector::cmp(int i, int j);
   ```

   that returns true if the value $v[i]$ is less than the value $v[j]$ and false otherwise.

4. In the new file, rewrite your sorting algorithms to use the `cmp` member function, rather than `<`, where appropriate. Test that your algorithms still work.

5. If you’ve done everything correctly you should be able to change the program to sort the values in descending order by changing a single `<` to a `>`. Can you see where to do this? [Hint: it’s in the `MVector` class].

6. Copy your code into yet another new file and add a protocol class `SortableContainer` that contains all interfaces required for sorting data stored within the class.

7. Modify your sorting algorithms so that they work with `SortableContainers` rather than `MVectors`, and modify the `MVector` class so that it inherits from `SortableContainer`. Test your algorithm.
Sorting a two-dimensional array of coordinates

Now that we have abstracted the sorting algorithms we can create any number of new SortableContainers. One object that will be useful later is an array of coordinates. Firstly, we define a two-dimensional coordinate structure

```c
struct IntegerCoordinate {
  unsigned X;
  unsigned Y;
};
```

Tasks:

1. Create a class CoordinateArray that inherits from SortableContainer and stores a vector IntegerCoordinate objects.

   ```c
   std::vector<IntegerCoordinate> v;
   ```

   Make sure that you include the necessary access functions so that you can access any coordinates stored in the array. You should also include a `resize(int n)` function.

2. Write a comparison operator

   ```c
   bool CoordinateArray::cmp(int i, int j);
   ```

   that returns true if the coordinate `v[i]` is lexicographically less than coordinate `v[j]` and false otherwise. A lexicographical ordering is defined by

   \[(X,Y) < (A,B) \text{ if } X < A \text{ or } [X = A \text{ and } Y < B].\]

3. Test your new class by sorting some CoordinateArrays of your choice.

Report:

Your report for this section should contain

- A hard-copy of your abstracted sorting functions and your CoordinateArray class.
- A comparison of average times taken to sort \(n\) doubles and \(n\) IntegerCoordinates and an interpretation of the results.

4.2.5 Sorting in the STL (optional)

The C++ standard template library (STL) has a number of sorting and searching functions. These are implemented at a yet further level of abstraction because we have so far assumed that all our SortableContainers can be indexed by an integer and that we only want one type of comparison operator for each container. This is too much of a restriction, as far as the STL
is concerned. The two ideas are that: 1) the entries in a container can all be accessed using an iterator: an object that iterates through the data; and 2) the comparison is performed by a special comparison functor: an object that behaves like a function.

Tasks

1. Read-up on sorting and searching in the STL using available web-resources

2. Convert your program so that the sorting is performed by the STL sort() algorithm.

4.3 The game of life using a CoordinateArray

The game of life is a cellular automaton which obeys a very simple set of rules. The classic game is played on a two-dimensional square grid of cells and each cell is either live or dead. As time advances the state of the cells changes according to various rules that depend on the states of the eight neighbouring cells. The classic rules are:

- If a dead cell has exactly 3 live neighbours it becomes live, otherwise it remains dead.
- If a live cell has 2 or 3 live neighbours it remains live, otherwise it dies.

These rules can be summarized by the notation B3/S23, indicating that a dead cell becomes live with 3 neighbours and a live cell stays alive with 2 or 3 neighbours.

One simple way to represent the game grid is to construct a two-dimensional array of booleans, but we must then store one boolean for every point in the grid. A more memory-efficient storage method is to store only the locations of the live cells in some sort of data structure. The problem is that the state of the neighbours of any live points can only be found by searching through the data structure.

We shall create a Life class that stores the coordinates of the live cells as a CoordinateArray.

Tasks

1. Write a class Life with the following prototype

```cpp
class Life
{
public:
  // Storage for the coordinates of the live cells
  CoordinateArray LiveCells;

  // Constructor
  Life();
};
```

1 The trade-off is worth considering. Imagine that we are playing on a grid of dimension $2^{32} \times 2^{32}$ (the standard maximum of an unsigned integer). Then we will required $2^{64}$ bits to store the entire grid as booleans, to convert this into bytes we divide by $2^8$, so we have $2^{61}$ bytes = $2^{51}$ kB = $2^{41}$ MB = $2^{31}$ Gb, which is an awful lot of storage! Storing a cell’s coordinates requires two integers (usually 8 bytes) per live point, which means that we can hold $2^{27}$ live points in 1 Gb of RAM.
You can use the lexicographical sorting on the CoordinateArray to allow relatively fast determination of the live neighbours of the cell and to determine the dead cells that have live neighbours. You may also feel free to investigate other methods to evolve the game of life and may use the STL if you wish. The only restriction is that the data structure that represents the state of the game must consist only of a container that stores the coordinates of the live cells.

2. Evolve the following two initial configurations of live cells
   (a) \{ (1,1), (1,2), (2,1), (2,2) \},
   (b) \{ (5,5), (5,4), (5,6) \}.

3. Evolve the following initial configuration (the acorn) of live cells:
   \{ (10001, 10001), (10002, 10001), (10002, 10003),
       (10004, 10002), (10005, 10001), (10006, 10001), (10007, 10001) \}.

4.4 Report

Write a project report as a connected piece of prose. You may use any suitable reference sources, but these must be clearly identified in the report at the points of use.

Your report must include:

- code listings for the three sorting algorithms: bubblesort, quicksort and heapsort;
- graphs of the average sort time as a function of the number of objects being sorted for the three algorithms;
- a comparison of the data in your graphs with the theoretical complexity of the algorithms;
- a discussion of which sorting algorithm is “best” in your opinion with reasons for your choice;
- a code listing for your game of life object;
- a representation (figure) of the state of the game after 5026 generations (time steps) of the acorn pattern described in Task 3 above.
- answers to the following questions:
  - what happens when you start with the initial configurations in Task 2 above?
what is the total number of live cells after 5026 generations of the acorn pattern?

Marks will be awarded for clarity and correctness of code as well as answers to the questions and discussion. The majority of the marks for this project (80%) will be awarded for the work described in section 2, with only 20% of the marks available for the investigation of the game of life, section 3.