MATH49111/69111 Scientific Computing
Semester 1, 2017

Projects 1

Chris Johnson
chris.johnson@manchester.ac.uk
http://www.maths.manchester.ac.uk/~cjohnson/teaching/sci_comp

List of Projects

Guidelines

Choose only **one** project from those in this booklet to complete and hand in. Students on the MSc in Quantitative Finance are encouraged to choose project 2; mathematics students should only choose project 2 if they have previously studied financial mathematics. The projects are written to be self contained, but you may (indeed, are encouraged to) reference relevant books, papers and other resources. You can of course ask me for any clarifications.

Project 1 is worth 40% of the marks for the course, and is be assessed by a written report, marked out of 40. The report will be marked in accordance with the criteria as described below under "Grading criteria".

This is **not** a group work assignment and your report and all code must be written by you individually. Do not read or copy the work of other students, and do not discuss the projects/-coursework or share your own work with others. Do not copy (or closely paraphrase) material from other sources. Please see the university guidelines on plagiarism (`http://documents.manchester.ac.uk/display.aspx?DocID=2870`). The reports and associated code should be submitted online on Blackboard by the deadline below.

- Write your report as though for a fellow student on your course: you will need to explain carefully the topic of your project, but there is no need to describe more basic mathematical concepts.
- Aim for precision and clarity of writing.
- The report should be well structured, containing:
  - an introduction and description of the problem
  - discussion of the problem formulation and numerical techniques used
  - your results, supported using figures and tables as required, and analysis of these results
  - a conclusion
- Ideally, reports should be typeset in LaTeX and submitted in PDF format[1]. A LaTeX coursework and project template is provided on the course website, which you may use if you wish.

[1] though you may use Microsoft Word or other word-processing software

- The report should **not exceed** 15 pages in length (excluding long tables, code etc. in appendices). This should be regarded as a maximum: a concisely written project may score very high marks using somewhat fewer pages than this.
- Any books or articles referred to must be included in a references section.
- Figures and tables should be clearly labelled, and all figures and tables in the report must be referenced in the text.
- Any code used to generate results used in the report must be included in an appendix. I may ask you to provide the source code files in addition to this. Small portions of code may be included in the text, or references to the code in the appendix can be used.
- Any numerical results presented should be accompanied by a reference to the code which generated them (e.g. the name of a source code file included in the appendix). It is very important that any numerical results presented in the report are generated from the numbers produced by your own code.
- You should attempt every task set in the project, but you do **not** need to write up every task for the report. The tasks that should be written up for the report are indicated clearly in the 'Report' section at the end of the project. However, you are welcome to to write up some of the other tasks in the report as well, and you may find this useful to explain what you have done.

Presentation

The report should be well structured with an introduction of the problem being solved, a problem formulation, results, and a conclusions section. It should be clearly written and free from spelling and grammatical errors. Mathematics should be typeset carefully. The figures and tables should be clear, references in the text, and labelled with suitable captions.

Mathematical content

The report should be factually accurate and the mathematical language precise and clear. There should be clear descriptions of the project and the outcome to be achieved, with evidence of understanding the wider context of the problem. Any results and conclusions made need to be well supported and coherent. Numerical results should be interpreted in the context of the mathematical problem, and the accuracy of the computations should be discussed in some depth. For example, numerical results should supported by validation/test cases (to demonstrate that the code has been implemented correctly), by grid or timestep convergence tests (to check that the result is accurate). Check that the results are a credible solution to the original problem! The report should provide evidence of an understanding, and the competent application, of the range of techniques and methods used in the project, and evidence of technical skills.

C++ code

Most importantly, the code must be correct, producing the results that are claimed. The code should be robust, incorporating tests and error checking (e.g. errors from coding mistakes or invalid user input, as well as numerical error due to discretisation and roundoff) where appropriate. The code should be well structured (e.g. using object oriented programming, where suitable or where specified in the project), and suitably commented. Credit will be given for additional effort put into the code beyond the minimum requirements of the project (for example, where code has been optimised for speed, or has been structured in a novel and suitable way, or where extensive testing has been built in).

Resources

- See the project template file on the course website for an example of LaTeX syntax and for links to web resources for more information about LaTeX.
- For a guide to scientific writing in general, see
  - *Scientists Must Write* (R. Barrass, Routledge, 2002)
  - *The Elements of Style* (W. Strunk & E. B. White, 1999)
  - *What's wrong with these equations?* (N. D. Mermin, in *Physics Today*, October 1989), available at `http://www.pamitc.org/documents/mermin.pdf`
- For technical guidance on plotting figures in MATLAB/Octave and Gnuplot, see the online notes on the course website. For guidance on figure design (much of it beyond the scope of what is required for this course) see, for example, *The Visual Display of Quantitative Information* (E. R. Tufte, Graphics Press, 1983) or *Semiology of Graphics* (J. Bertin, ESRI Press, 2010).

Grading criteria

- 20 marks (50% of the marks for the project) are for obtaining the correct numerical answers, and writing the code that does so. These marks are gained from questions in the report that ask for numerical results to be tabulated or plotted, and from the questions that ask for code to be written. These numerical answers should be incorporated into the written report (not presented as standalone answers to each question).
- The remaining 50% of the marks for each project are for the understanding and analysis demonstrated in the report. These marks assess how you have chosen appropriate algorithms and program architectures in your code, how you have validated your numerical results, and presented these in the context of the mathematical problem. The levels of performance required to gain these marks are described in table 1 below.

Grade	0–24%	25–49%	50–74%	75–100%
Introduction	Cursory or no explanation of mathematical problem, no references	Some description of mathematical problem and/or what is done in this report	Descriptions of problem and the work in this report placed within broader context, with appropriate references	Interesting, cogent account of the work in this report, in the context of the mathematical problem considered and scientific computing more widely. Very well referenced.
Validation	No validation, or flawed validation, of numerical results	Simple validation and/or code error checking performed, with some explanation	Careful validation of numerical results and code tests, with explanation of how these tests relate to one another	Convincing and detailed reasoning of the validity of all the numerical results, drawing together error checks in code, convergence testing supported by theory, appreciation of likely sources of error, suitable and novel choice of test cases
Analysis and context	Results presented with no context, and without references.	Some linking of the results to the problem in question, little or no referencing.	Results drawn together and compared in the context of the mathematical problem described in the project, with references were appropriate to methods etc.	Careful analysis of the results and their meaning, in the context of the mathematical problem and more broadly. Methods well referenced, including broader discussion/references than that presented in the project booklet.
Conclusion	No (or mostly irrelevant) conclusion	Conclusion describes result(s) in simple terms	Conclusion describes work done throughout the report and how it links together	Conclusion draws together strands of work throughout the report, demonstrating how the results obtained relate to each other and the broader context of the mathematical problem
Presentation	Writing and equations unclear and/or marred by typographical errors. Figures poorly chosen and unclear and/or ambiguous (e.g. missing or unreadable scales or axis labels)	Writing is broadly clear though, with some areas of ambiguity and/or language errors. Arguments are not very clearly structured. Limited use of figures or figures are not designed well to serve the arguments in the text.	Accurate writing with few spelling or grammar errors. The argument structure is usually clear. Figures are clear and generally well chosen, though may not be ideally designed to illustrate the arguments in the text.	Precise, analytical writing free from spelling and grammatical errors. The structure of the argument is clear and appropriate for the intended audience (a fellow student on the course). Equations carefully chosen and explained, and well typeset. Figures are clear and attractive, with results presented so as to best illustrate the points being made in the text.

Table 1: Grading criteria for the understanding/analysis, worth 50% of the total marks for the project. The other 50% of marks are for the correctness of the numerical answers and C++ programs, as indicated in the specific questions in the project.

Deadlines

- The written report for the project is due in by:
  3pm, Friday 17th November

Project 1

Ordinary Differential Equations

Original author: Dr. Paul Johnson

In this project we will solve a an ordinary differential equation (ODE) with boundary conditions specified at two values of the independent variable, *i.e.* a boundary value problem (BVP). The technique we will use is the so-called *shooting* method. This project builds on the techniques covered in lectures for *initial* value ODE problems.

1.1 Solving ODE Boundary Value Problems

1.1.1 The initial value problems for ODEs

Recall that a scalar first-order initial value problem is an ordinary differential equation of the form

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y), \quad a \le x \le b, \tag{1.1}$$

subject to an initial condition

$$y(a) = \alpha. \tag{1.2}$$

Any higher order ODE (of order n) may reduced to a vector first order ODE (with a vector of dimension n) by writing the successive derivatives $\mathrm{d}y/\mathrm{d}x, \mathrm{d}^2y/\mathrm{d}x^2, \ldots$ as additional components of the vector. Such a vector ODE system may be written

$$\frac{\mathrm{d}\mathbf{Y}}{\mathrm{d}x} = \mathbf{F}(x, \mathbf{Y}), \quad a \le x \le b, \tag{1.3}$$

where

$$\mathbf{Y} = (Y_1(x), Y_2(x), \ldots, Y_n(x))^T \quad \text{and}$$
$$\mathbf{F} = (F_1(x, \mathbf{Y}), F_2(x, \mathbf{Y}), \ldots, F_n(x, \mathbf{Y}))^T,$$

with initial data

$$\mathbf{Y}(a) = \boldsymbol{\alpha}, \tag{1.4}$$
$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T.$$

1.1.2 Boundary Value Problems

In initial value problems, the boundary conditions of an ODE are specified at a single value of the independent variable x, and we have seen that algorithms for numerically solving ODEs, such as the Euler method, allow integration away from these initial conditions. In boundary value problems, such as

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + \kappa \frac{\mathrm{d}y}{\mathrm{d}x} + xy = 0, \tag{1.5}$$

with the boundary conditions

$$y(0) = 0, \qquad y(1) = 1, \tag{1.6}$$

the conditions are specified at more than one value of the independent variable x. First let's rewrite (1.5) as a system of first order ODEs,

$$Y_1 = y(x); \tag{1.7}$$

$$Y_2 = \frac{\mathrm{d}y}{\mathrm{d}x}; \tag{1.8}$$

$$\frac{\mathrm{d}Y_1}{\mathrm{d}x} = Y_2; \tag{1.9}$$

$$\frac{\mathrm{d}Y_2}{\mathrm{d}x} = -\kappa Y_2 - xY_1. \tag{1.10}$$

so that the boundary conditions (1.6) are now written

$$Y_1(x = 0) = 0, \quad Y_1(x = 1) = 1. \tag{1.11}$$

Importantly, we do not *a priori* know the value of all the unknowns at either boundary (we do not know $Y_2(x = 0)$, nor $Y_2(x = 1)$). We cannot, therefore, simply apply a numerical method for initial value problems, since these require all the unknowns to be specified at some initial value of the independent variable.

1.1.3 The shooting method

The essence of the shooting method is to guess the unknown boundary conditions at the first boundary. Together with the known boundary conditions specified here, these guesses allow the equations to be integrated numerically away from the first boundary. By integrating the equations to the second boundary, we may compare the boundary conditions that are specified at the second boundary with the values predicted by the numerical integration. Because we have guessed some values at the first boundary, it is likely that the values obtained from the numerical integration will not match the boundary conditions specified at the second boundary. However, by iteratively refining the guessed unknowns at the first boundary, we can find guesses which satisfy the boundary conditions at the second boundary, and thus solve the BVP.

Applying this strategy to our problem (1.9), (1.10), (1.11), we start by making a guess, g, for $Y_2(x = 0)$ so that the initial conditions now become

$$\mathbf{Y}(0) = \begin{pmatrix} Y_1(0) \\ Y_2(0) \end{pmatrix} = \begin{pmatrix} 0 \\ g \end{pmatrix}. \tag{1.12}$$

We then solve (1.9) and (1.10) with these initial conditions using any numerical method for ODE initial value problems, to get a solution at $x = 1$,

$$\mathbf{Y}(1) = \begin{pmatrix} Y_1(1) \\ Y_2(1) \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}. \tag{1.13}$$

It is now possible to see how good our guess at the initial condition was, by comparing the value β_1 to our boundary condition $Y_1(x = 1) = 1$. We define the *residual* φ as the difference between the value obtained from numerical integration and the value specified by the boundary condition Y^{BC}, *i.e.*

$$\varphi(g) = Y_1(x = 1; g) - Y_1^{BC}(x = 1) = \beta_1 - 1, \tag{1.14}$$

where β_1 is our solution from numerical integration (from 1.13) and 1 is the required value from the boundary condition (1.11). Since φ is just a function of g (remember $g = Y_2(x = 0)$) and the boundary condition is satisfied when $\varphi = 0$, the problem reduces to the problem of finding a root of a scalar function of one variable,

$$\varphi(g) = 0. \tag{1.15}$$

In exercise sheet 2 (Q2.6) we described an algorithm to solve this problem – Newton's method.

1.1.4 The shooting method using Newton iteration

We now combine the Newton iteration for root finding with an initial value ODE solver to calculate the solution to boundary value problems. Recall from exercise sheet 2 that, when solving (1.15) by the Newton iteration, we start from some initial guess g_0 and iteratively refine this with successive guesses $g_1, g_2, \ldots$. Given some guess g_n, the next guess g_{n+1} is given by

$$g_{n+1} = g_n - \frac{\varphi(g_n)}{\varphi'(g_n)}. \tag{1.16}$$

If $\varphi(g)$ is nonlinear this iteration converges quadratically (roughly speaking, the number of correct digits doubles with each iteration) *for a sufficiently good initial guess*. If $\varphi(g)$ is linear then the situation is even better: $\varphi'(g)$ is independent of g, and a single iteration of (1.16) solves (1.15) exactly, for any initial guess[1].

We have described in §1.1.3 how the function $\varphi(g)$ is evaluated, by integrating the equations from $x = 0$ to $x = 1$. But to apply the iteration (1.16) we still need to know how to evaluate $\varphi'(g)$, where, from differentiating (1.14)

$$\varphi'(g) = \left.\frac{\mathrm{d}Y_1}{\mathrm{d}g}\right|_{x=1}. \tag{1.17}$$

One method of evaluating $\varphi'(g)$ is to numerically evaluate this derivative using a finite-difference method, as described in lecture 4. Another method is to differentiate the original ODE algebraically with respect to g to get a new initial value problem for φ'. The ODE (1.5) may be written as:

$$y_{xx} = -\kappa y_x - xy =: F(x, y, y_x), \tag{1.18}$$

where subscript x here indicates a derivative with respect to x. Defining $Z_1 = \mathrm{d}y/\mathrm{d}g$, and $Z_2 = \mathrm{d}y_x/\mathrm{d}g$, and differentiating (1.18) with respect to the guess g using the chain rule,

$$\frac{\mathrm{d}y_{xx}}{\mathrm{d}g} = \frac{\partial F}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}g} + \frac{\partial F}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}g} + \frac{\partial F}{\partial y_x}\frac{\mathrm{d}y_x}{\mathrm{d}g}, \tag{1.19}$$

we obtain the set of first order ODEs

$$\frac{dZ_1}{dx} = Z_2, \tag{1.20}$$

$$\frac{dZ_2}{dx} = -\kappa Z_2 - xZ_1, \tag{1.21}$$

with initial conditions

$$Z_1(x = 0) = \frac{d}{dg}\left(Y_1(x = 0)\right) = 0, \tag{1.22}$$

$$Z_2(x = 0) = \frac{d}{dg}\left(Y_2(x = 0)\right) = 1. \tag{1.23}$$

This is an initial value problem for (Z_1, Z_2) which we may integrate numerically to find

$$Z_1(x = 1) = \left.\frac{dY_1}{dg}\right|_{x=1} = \varphi'(g). \tag{1.24}$$

Because our example problem (1.5) is linear, the equations (1.20) are independent of the values of Y_1 and Y_2, and so could be integrated independently of (1.10)[2]. For nonlinear problems, the derivatives of $Z_1, Z_2, \ldots$ at a particular value of x depend on the values of $Y_1, Y_2, \ldots$ at that value of x, and so the system of equations for Y_i and Z_i must be integrated simultaneously, by augmenting our solution vector $\mathbf{Y}$ with the additional components Z_1, Z_2 etc.

[1] Roundoff error means that the solution obtained is not *exact*. A second iteration can often help to reduce this roundoff error, though it does not eliminate it entirely.

[2] For this homogeneous linear problem they are the exactly same equations as (1.10).

1.2 Coding, Examples and Exercises

1.2.1 Creating a Vector class

We will define a new class `MVector` that represents a mathematical vector of real numbers. By overloading operators for this class (for example, operators for adding vectors or multiplying vectors by scalars), we can write the code for the our (vector) ODE integration routines much as we would write it in mathematical notation[3].

Put following class definition for the new class `MVector` into a header file or at the top of your main code. This header file is available on the course website (in the Example Code section) as `mvector.h`.

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>

// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}

    // access element (lvalue)
    double &operator[](int index) { return v[index]; }

    // access element (rvalue)
    double operator[](int index) const { return v[index]; }

    int size() const { return v.size(); } // number of elements

private:
    std::vector<double> v;
};

#endif
```

The class `MVector` has a single member variable, a `std::vector<double>` which stores the elements of the vector. The constructors, access operators `[]` and `size()` member function mimic these parts of the interface of `std::vector<double>`.

We now overload the arithmetic operators `+`, `-`, `/`, `*` for our new `MVector` class. These operators are typically overloaded in non-member functions (see C++ Primer, §§14.1 and 14.3), and so the operator overload is placed outside the class definition, but inside the header file.

A typical implementation will look like

```
// Operator overload for "scalar * vector"
inline MVector operator*(const double& lhs, const MVector& rhs)
```

[3] If our goal was maximum performance we might instead use a library such as ATLAS or BLAS, which are optimised for speed but have a less intuitive interface.

```
{
    MVector temp(rhs);
    for (int i=0; i<temp.size(); i++) temp[i]*=lhs;
    return temp;
}
```

when placed in the header file (note the use of the `inline` keyword, since the function is in the header file[4].)

Tasks:

1. For the four operators `+`, `-`, `/`, `*` there are five overloads to implement. Remember that we can multiply/divide a vector by a scalar, and add/subtract vectors from vectors, but can't add/subtract a scalar from vector nor divide a scalar by a vector. What are the prototypes of the five overloads required?
2. Implement these overloads in your code.
3. Check that the code is working by evaluating the following using `MVector`s to represent $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{x}$:

$$\mathbf{u} = 4.7\mathbf{v} + 1.3\mathbf{w} - 6.7\mathbf{x}$$

   where $\mathbf{v} = (0.1, 4.8, 3.7)$, $\mathbf{w} = (3.1, 8.5, 3.6)$ and $\mathbf{x} = (5.8, 7.4, 12.4)$.
4. Try other combinations of 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 second constructor? (See C++ Primer, §12.4.4).
5. When adding or subtracting two vectors, write a check to make sure that they have the same number of elements, and report an error if they do not.
6. You could also try overloading the << operator to output a vector in the form `(v[0], v[1], ..., v[n])` to `std::cout` and other output streams.

```
// Overload the << operator to output MVectors to screen or file
ostream& operator<<(ostream& os, const MVector& v)
{
    int n = v.size();
    os << "(";
    // write the code to output the vector components here.
    os << ")";
    return os;
}
```

1.2.2 Abstract base class for ODEs

In this section we develop an interface for a function that takes one `double` and one `MVector` parameter and returns a `MVector`. This interface is suitable for representing the vector of derivatives in a vector ODE, i.e. $\mathbf{F}(x, \mathbf{Y})$ from (1.3).

The abstract base class `MFunction` will define this interface using a pure virtual member function (specifically, a pure virtual overload of the function call operator, `operator()`). The class is defined entirely as follows:

```
struct MFunction
{
virtual MVector operator()(const double& x,
                           const MVector& y) = 0;
```

[4]We could instead place the function definition above in a separate `.cpp` file, and leave just the function declaration (prototype) in the header. If this is done, the `inline` keyword is not used in either the definition or declaration.

```
};
```

Recall that:

- a `struct` is a `class` where all members are public (C++ Primer §2.8),
- the syntax `= 0` after the definition of the virtual `operator()` means that it is declared as a *pure* virtual function, with no function definition (function body) (C++ Primer §15.6),
- we cannot instantiate (make an object of) a class with a pure virtual function (an abstract base class). Instead we define a class which *derives* from the abstract base class and overrides its pure virtual function(s). We can then instantiate objects of this derived class.

For example, we could use inheritance to generate a new class that implements the function

$$\mathbf{F}_1(x,\mathbf{Y}) = \begin{pmatrix} Y_1 + xY_2 \\ xY_1 - Y_2 \end{pmatrix},$$

and use it to evaluate

$$\mathbf{v} = \mathbf{F}_1(2,\mathbf{Y}), \tag{1.25}$$

where

$$\mathbf{Y} = \begin{pmatrix} 1.4 \\ -5.7 \end{pmatrix},$$

To do this we would derive the function class from `MFunction`:

```
class FunctionF1 : public MFunction
{
public:
    MVector operator()(const double& x, const MVector& y)
    {
        MVector temp(2);
        temp[0] = y[0] + x*y[1];
        temp[1] = x*y[0] - y[1];
        return temp;
    }
};
```

To evaluate (1.25) we might then write in the `main` function

```
MVector v, y(2);          // initialise y with 2 elements
FunctionF1 f;             // f has order 2 by definition
y[0] = 1.4; y[1] = -5.7; // assign element values in y
v = f(2.,y);              // evaluate function f as required
std::cout << "v=" << v << "; y=" << y << std::endl;
```

(where we have assumed that `<<` is overloaded for `MVector` (as in task 6 of §1.2.1). The output is

```
v=(-10, 8.5); y=(1.4, -5.7)
```

Tasks:

1. Copy the classes `MFunction`, `FunctionF1` and the code immediately above and get it to compile and run. If you have not overloaded `<<`, simply output $\mathbf{v}$ and $\mathbf{Y}$ element-by-element.

2. Declare another `MVector` $\mathbf{u}$ with 2 elements and set them to 1 and 2 respectively. Now let v be defined by the expression
$$\mathbf{v} = \mathbf{u} + \mathbf{F}(2, \mathbf{Y}). \tag{1.26}$$
Can this be written in C++ directly (i.e as `v = u + f(2.0,y)`)? Calculate the result by hand to check your code.
3. Now declare `double`s $h = 0.1$, and $x = 0.5$, and evaluate
$$\mathbf{v} = \mathbf{u} + h\mathbf{F}(x, \mathbf{u} + h\mathbf{Y}). \tag{1.27}$$
Again, try to write this in one line of C++ code. Calculate the result by hand to check your code.

1.2.3 ODE solver function

Below is the prototype (declaration) of a function that can be used to solve ODEs. In order to solve an initial value ODE problem we need to provide the function with the initial conditions, the initial value of the independent variable x, the number of steps, and the function $\mathbf{F}(x, \mathbf{Y})$ that we are solving. On entry the arguments to this function contain all of those elements, and on return the solution can be stored inside the vector y (since `y` is also an 'output parameter', being passed by reference). In this section you must complete the implementation (definition) of this function.

```
// Declaration for an Euler scheme ODE solver
int EulerSolve(int steps, double a, double b, MVector &y, MFunction &f);
```

On entry to the function we need to set:

- `steps` – number of steps in the problem
- `a` – initial value of x
- `b` – final value of x
- `y` – the initial value of $\mathbf{Y}(x = a)$
- `f` – the function defining the problem we are solving

When the function returns, it will output:

- `y` – the solution $\mathbf{Y}(x = b)$
- return value – integer that can give information about any errors that have occurred.

Tasks:

1. Write in your code the prototype for the `EulerSolve` function, as above, and fill in the implementation of the function. This code should carry out the following algorithm for Euler integration:
   (a) Declare and initialise the value of x.
   (b) Declare and calculate the step size h.
   (c) Loop over the number of steps and update x and $\mathbf{Y}$ according to the algorithm
   $$x_i = a + ih. \tag{1.28}$$
   $$\mathbf{Y}_{i+1} = \mathbf{Y}_i + h\mathbf{F}(x_i, \mathbf{Y}_i), \tag{1.29}$$
   for $i = 0, 1, \ldots,$ `steps-1`.
2. Make a new class inheriting from `MFunction` to evaluate the function
$$\mathbf{F}_2(x, \mathbf{Y}) = \begin{pmatrix} x \\ Y_2 \end{pmatrix}.$$

3. Use your function `EulerSolve` to solve the initial value problem
$$\frac{d\mathbf{Y}}{dx} = \mathbf{F_2}(x, \mathbf{Y}), \qquad \text{with} \qquad \mathbf{Y}(x=0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{1.30}$$
on the interval $x \in [0, 1]$. The exact solution at $x = 1$ is
$$\mathbf{Y}(x=1) = \begin{pmatrix} 0.5 \\ e \end{pmatrix}. \tag{1.31}$$
Create a table containing your values of $Y_1(x=1)$ and $Y_2(x=1)$ for different numbers of steps $n = 10, 20, 40, 80, 160\ldots$ up to a limit of your choosing:
4. Think about error checking in your implementation of `EulerSolve`. What should happen if the number of elements in `y` and `f` are different?
5. Write functions `MidpointSolve` and `RungeKuttaSolve` to solve the ODE using the midpoint method and 4th order Runge-Kutta method, respectively. Base your new functions on the prototype and implementation you have already written for `EulerSolve`.
   (a) The midpoint method is given by the recurrence relation
   $$\begin{aligned} x_i &= a + ih, \\ \mathbf{Y}_{i+1} &= \mathbf{Y}_i + h\mathbf{F}\left(x_i + \tfrac{1}{2}h, \mathbf{Y_i} + \tfrac{1}{2}h\mathbf{F}(x_i, \mathbf{Y_i})\right). \end{aligned}$$
   for $i = 0, 1, \ldots,$ `steps` $- 1$.
   (b) The 4th order Runge-Kutta method may be expressed as
   $$\begin{gathered} x_i = a + ih, \\ \mathbf{k}_{i1} = \mathbf{F}(x_i, \mathbf{Y}_i), \quad \mathbf{k}_{i2} = \mathbf{F}\left(x_i + \frac{h}{2}, \mathbf{Y}_i + \frac{h}{2}\mathbf{k}_{i1}\right), \\ \mathbf{k}_{i3} = \mathbf{F}\left(x_i + \frac{h}{2}, \mathbf{Y}_i + \frac{h}{2}\mathbf{k}_{i2}\right), \quad \mathbf{k}_{i4} = \mathbf{F}(x_i + h, \mathbf{Y}_i + h\mathbf{k}_{i3}), \\ \mathbf{Y}_{i+1} = \mathbf{Y}_i + \frac{h}{6}(\mathbf{k}_{i1} + 2\mathbf{k}_{i2} + 2\mathbf{k}_{i3} + \mathbf{k}_{i4}), \end{gathered}$$
   for $i = 0, 1, \ldots,$ `steps` $- 1$.
6. Test the methods against each other on the problem (1.30) Think about how you should measure the error in $\mathbf{Y}$. How does the error of each method depend on the step size h?
7. Now consider the following (nonlinear) ODE:
$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = \frac{1}{8}\left(32 + 2x^3 - y\frac{\mathrm{d}y}{\mathrm{d}x}\right), \tag{1.32}$$
on the interval $x \in [1, 3]$ with the initial conditions
$$y(x=1) = 17, \qquad y'(x=1) = 1. \tag{1.33}$$
   (a) Write the ODE (1.32) as a vector system of first order ODEs (write $Y_1 = y$ and $Y_2 = dy/dx$).
   (b) Calculate the form of the function $\mathbf{F}$ that evaluates to the derivatives of Y_1 and Y_2 for this ODE (as in (1.3)).
   (c) Write a new class that derives from `MFunction` to represent this derivative function.
8. Include an option within your ODE solver to write the values of $\mathbf{Y}_i$, x_i for all i to a file.

1.2.4 Implementing the Newton shooting method

Suppose we wish to solve the BVP defined in (1.5) and (1.6) with $\kappa = 1$. Since this is a BVP, to solve this we use the Newton method and so must integrate (1.20) and (1.21) alongside (1.5) and (1.6). We write a class representing the function of derivatives $\mathbf{F}$ for this equation:

```
class Eqn1p5Derivs : public MFunction
{
public:
    // constructor to initialise kappa
    Eqn1p5Derivs() {kappa=1.0;}

    MVector operator()(const double& x,const MVector& y)
    {
        MVector temp(4);
        temp[0] = y[1];
        temp[1] = -kappa*y[1] - x*y[0];
        temp[2] = y[3];
        temp[3] = -kappa*y[3] - x*y[2];
        return temp;
    }
    void SetKappa(double k) {kappa=k;} // change kappa
private:
    double kappa; // class member variable, accessible within
                  // all Eqn1p5Derivs member functions
};
```

Note that we have included a parameter in the equations as a member variable. In the `main` function we have something like

```
Eqn1p5Derivs f;
int maxNewtonSteps = 100;
double guess = 0;
double tol = 1e-8;
for (int i=0; i<maxNewtonSteps; i++)
{
    MVector y(4);
    // y[0] = y, y[1] = dy/dx, y[2] = Z_1, y[3] = Z_2
    y[0]=0; y[1]=guess; y[2]=0.0; y[3]=1.0;

    RungeKuttaSolve(100, 0.0, 1.0, y, f); // solve IVP

    double phi = y[0] - 1; // calculate residual
    double phidash = y[2]; // 'Jacobian' phidash = z_1(x=1)

    if (std::abs(phi) < tol) break; // exit if converged

    guess -= phi/phidash; // apply newton step
}
```

The `std::abs` function is in the `<cmath>` library.

Tasks:

The aim of these tasks is to solve the ODE (1.32) with the boundary conditions

$$y(x=1) = 17, \qquad y(x=3) = \frac{43}{3}. \tag{1.34}$$

1. Consider that (1.32) may be written as:

$$y'' = \frac{1}{8}\left(32 + 2x^3 - yy'\right), \tag{1.35}$$

$$y'' = F(x, y, y') \tag{1.36}$$

   (a) Differentiate (1.32) with respect to the guess g, using the chain rule.

$$\frac{dy''}{dg} = \frac{\partial F}{\partial x}\frac{dx}{dg} + \frac{\partial F}{\partial y}\frac{dy}{dg} + \frac{\partial F}{\partial y'}\frac{dy'}{dg}. \tag{1.37}$$

   (Hint: $\frac{dx}{dg} = 0$)

   (b) Defining $z = dy/dg$, write down the set of first order ODEs and initial conditions satisfied by z.

   (c) Write a class deriving from `MFunction` that represents the ODEs satisfied by both y and z (this should be a 4-element system, as in the example in §1.2.4.)

   (d) Use this function in a code to solve the boundary value problem defined by (1.32) and (1.34). The Newton iteration for this problem is given by

$$g_{n+1} = g_n - \frac{\varphi(g_n)}{\varphi'(g_n)}, \text{ where } \varphi'(g_n) = z(x=3; g_n). \tag{1.38}$$

2. Check your code against the exact solution, $y(x) = x^2 + 16/x$ and $y'(x=1) = -14$.

3. Implement some error checking. What happens if we reach the maximum number of Newton iterations without converging to the specified tolerance? How should we handle this error?

1.3 The Falkner Skan Equation

The Falkner-Skan equations

$$f''' + ff'' + \beta(1 - f'^2) = 0, \tag{1.39}$$

$$f(0) = f'(0) = 0, \tag{1.40}$$

$$f' \to 1 \text{ as } \eta \to \infty. \tag{1.41}$$

arise from the derivation of the boundary layer flow of a fluid past a wedge. Here β is a parameter and primes on $f(\eta)$ denote derivatives with respect to η.

Tasks:

- Write (1.39) as a first order vector ODE.
- Use a 4th order Runge-Kutta integrator to obtain a numerical solution to the initial value problem specified by the ODE (1.39), the two boundary conditions at $\eta = 0$ (1.40), and a guess $f''(0) = g$.
- Write down the system of first order ODEs satified by df/dg, df'/dg and df''/dg.
- Solve the augmented system (1.39)–(1.41), augmented with your equations for df/dg etc., using a 4th order Runge-Kutta integrator, using appropriate boundary conditions.
- Write code to solve the Falkner-Skan boundary value problem using the shooting method with Newton iteration, to find the value of $f''(0)$ that results in the boundary condition $f' \to 1$ as $\eta \to \infty$ being satisfied.

- To approximate the boundary condition applied in the limit $\eta \to \infty$, you will need to impose the boundary condition at some sufficiently large (but finite) value of $\eta = \eta_\infty$. Check how the value of η_∞ affects the solution. What are the disadvantages of making η_∞ too small, or too large?
- Write a function which encapsulates this Falkner-Skan boundary value problem, taking β (and possibly an initial guess for $f''(0)$) as parameters and returning the calculated value of $f''(0)$ for the given β.

You may find that quite an accurate initial guess for $f''(0)$ is required for the Newton iteration to converge (try a guess of $f''(0) = 0.92$ at $\beta = 1/2$, for example). One way of obtaining a good initial guess is to use a simple form of *parameter continuation*: the solution $f''(0)$ at one value of β is used as the initial guess for the solution at $\beta + \Delta\beta$, for some small increment $\Delta\beta$. Applying this repeatedly, for sufficiently small values of $\Delta\beta$, we can obtain converged numerical solutions over a wide range of values of β. (One avenue for further investigation might be to look at how the size of $\Delta\beta$ influences the number of steps required for the Newton iteration to converge. Could this be used as a way of selecting a good value for $\Delta\beta$ – i.e. one that is neither too small, requiring many unnecessary steps, nor too large, resulting in non-convergence of the Newton method?)

1.4 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 20 marks available for correct numerical results and code. The other 20 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

1. Solve the ODE and initial conditions (1.32) and (1.33) using the three methods (Euler, midpoint and 4th-order Runge-Kutta). In your report state the problem, and comment on the accuracy of each numerical method, and how this varies with the step size h. **[4]**
2. Summarise the Falkner-Skan system (1.39)–(1.41) and the methods you used to solve it. Describe any classes used in your code. **[2]**
3. Include plots of $f(\eta)$, $f'(\eta)$, $f''(\eta)$ against η, for $\beta = 0, 1/2$, and 1. For these values of β, include a table giving the value of $f''(0)$ to as many correct digits as you can (but try not to include any incorrect digits). **[6]**
4. Plot $f''(0)$ against β for $0 \le \beta \le 1$. **[4]**
5. Try to extend the plot for $f''(0)$ for $\beta < 0$. Is there always exactly one solution for $f''(0)$ for a given $\beta < 0$? **[4]**
6. Comment on the accuracy of your solution, and document the step sizes and values of η_∞ used.

Project 2

Historic vs Implied Volatility

Original author: Dr. Paul Johnson,

In the first part of the project we will read data into a C++ program from a file, then process the data with some simple statistical algorithms. In the second part, we will use analytical formulae along with the Newton-Secant method to derive implied volatilities for option prices, and investigate how closely the implied volatility matches the observed historical volatility.

2.1 Background Theory

2.1.1 Statistical Estimates

We first give a brief definition of some of the statistical properties calculated.

The Moving Average

For a vector $\boldsymbol{x} = (x_0, x_1, \dots, x_i, \dots, x_{n-1})$ with n elements, the moving average with m points to the left and p points to the right is defined as:

$$\bar{x}_i = \frac{1}{m+p+1} \sum_{j=-m}^{p} x_{i+j}. \tag{2.1}$$

A weighted moving average, with weights w_j, is defined by

$$\bar{x}_i = \frac{\sum_{j=-m}^{p} w_j x_{i+j}}{\sum_{j=-m}^{p} w_j}, \tag{2.2}$$

which reduces to (2.1) if all the w_j are equal.

The Exponential Moving Average

For a vector $\boldsymbol{x} = (x_0, x_1, \dots, x_i, \dots, x_{n-1})$ with n elements, the exponential moving average is defined as:

$$\begin{aligned} \bar{x}_0 &= x_0 \\ \bar{x}_i &= \alpha x_i + (1-\alpha)\bar{x}_{i-1} \end{aligned}$$

where $\alpha \in [0,1]$ is the smoothing factor.

Historical Volatility

We will use a simple, but naive, algorithm to estimate the volitility of sample data. Assume we have a vector of values $\boldsymbol{S} = (S_0, S_1, \ldots, S_i, \ldots, S_{n-1})$ that represents samples from a process driven by the following stochastic differential equation (SDE) in discrete form:

$$\frac{dS_i}{S_i} = \mu\, dt + \sigma \sqrt{dt}\, \varphi_i \tag{2.3}$$

where $S_{i+1} = S_i + dS_i$, μ is a parameter and φ_i are normally-distributed random variables, $\varphi_i \sim N(0, 1)$. This is a simple model for a stock price **S** following a log normal random walk.

The variance of a random variable X defined as

$$\mathrm{var}(X) = E[X^2] - E[X]^2. \tag{2.4}$$

For data $\boldsymbol{x} = \{x_0, x_1, \ldots, x_i, \ldots, x_{n-1}\}$ sampled from a population, the population variance is

$$\mathrm{var}(X) = \frac{1}{n-1}\left(\sum_{i=0}^{n-1}(x_i)^2\right) - \frac{1}{n^2 - n}\left(\sum_{i=0}^{n-1} x_i\right)^2. \tag{2.5}$$

The variance σ^2 of our sampled SDE $\boldsymbol{S}$ is then calculated follows:

1. Generate the vector $\boldsymbol{d}\,\mathbf{ln}\,(\boldsymbol{S}) = \{\ln(S_1) - \ln(S_0), \ln(S_2) - \ln(S_1), \ldots, \ln(S_{n-1}) - \ln(S_{n-2})\}$ by taking logs of the original data and evaluating the difference between successive data points.
2. Calculate the variance of $\boldsymbol{d}\,\mathbf{ln}\,(\boldsymbol{S})$ using equation (2.5).
3. Then the observed volatility is given by

$$\sigma = \sqrt{\mathrm{var}(\boldsymbol{d}\,\mathbf{ln}\,(\boldsymbol{S}))} \tag{2.6}$$

2.1.2 Generating Option Prices

Consider the well-known Black and Scholes (1973) partial differential equation for an option with an underlying asset following geometric Brownian motion:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0, \tag{2.7}$$

where $V(S, t)$ is the price of the derivative product, S the current value of the underlying asset, t is time, T is the time to maturity, r the risk-free interest rate, σ the volatility of the underlying asset and X is the exercise price of the option.

We can calculate the value of such an option by making a transformations in (2.7)

$$x = \log(S_0/X), \tag{2.8}$$

$$y = \log(S_T/X), \tag{2.9}$$

where S_0 is the asset value at $t = 0$ and S_T is the asset value at maturity $t = T$. The final conditions for a European option expiring at time T with $V(y, T)$ are transformed in straightforward fashion; e.g. the payoff for a call option $V(S, T) = \max(S_T - X, 0)$ becomes

$$V(y, T) = \max(Xe^y - X, 0). \tag{2.10}$$

The value of the option at time $t = 0$ on an underlying asset S_0 has an exact solution given by

$$V(x, 0) = A(x)\int_{-\infty}^{\infty} B(x, y)V(y, T)\, dy, \tag{2.11}$$

where,

$$A(x) = \frac{1}{\sqrt{2\sigma^2\pi T}} e^{-\frac{1}{2}kx - \frac{1}{8}\sigma^2 k^2 T - rT}, \tag{2.12}$$

and

$$B(x, y) = e^{-\frac{(x-y)^2}{2\sigma^2 T} + \frac{1}{2}ky} \tag{2.13}$$

and

$$k = \frac{2r}{\sigma^2} - 1. \tag{2.14}$$

Defining the integrand of (2.11) as

$$f(x,y) = B(x,y)V(y,T), \tag{2.15}$$

we can rewrite (2.11) as

$$V(x,t=0) = A(x)\int_{-\infty}^{\infty} f(x,y)\,dy. \tag{2.16}$$

Thus, to calculate the value of a European call option at the current time $t = 0$, we take the current value of the underlying asset S_0 and exercise price of the option X, and use these to calculate x using (2.8). We then use this value of x in (2.16), and use a numerical integration method, detailed below, to evaluate the value of the option (2.16).

2.1.3 Implied Volatility

Let us define $\hat{V}(S,t;X,T)$ as the observed price for a European call (or put) option on the market, with S the observed stock price at time t, maturity date T and exercise price X, and $V(S,t;X,T,r,\sigma)$ be the price calculated from the Black-Scholes model with extra parameters r the interest rate and σ the volatility. Then if we know what the risk-free interest rate r is, we only need to know what σ is to generate a Black-Scholes price. We find the implied volatility by choosing the correct value of $\sigma = \sigma_{im}$ such that the Black-Scholes price equals the market price, or:

$$V(S,t;X,T,r,\sigma_{im}) = \hat{V}(S,t;X,T). \tag{2.17}$$

To find the implied volatility we define a function

$$g(\sigma) = V(S,t;X,T,r,\sigma) - \hat{V}(S,t;X,T), \tag{2.18}$$

and use the Newton-Secant method to find the value of $\sigma = \sigma_{im}$ such that $g(\sigma_{im}) = 0$.

The Newton-Secant method finds roots of the equation $g(\sigma) = 0$ with the iteration:

$$\sigma_n = \sigma_{n-1} - g(\sigma_{n-1})\frac{\sigma_{n-1} - \sigma_{n-2}}{g(\sigma_{n-1}) - g(\sigma_{n-2})}. \tag{2.19}$$

Two suitable guesses must be supplied to start the Newton-Secant method, and if these are good enough guesses, the sequence σ_n converges to a root of the equation as $n \to \infty$.

2.1.4 Project overview

In this project our aim is to measure observed volatility in historical option prices, and compare this with the implied volatility as predicted by the Black-Scholes equation. This requires

1. Measuring the observed volatility using (2.6).
2. Writing functions to evaluate the value of a call option predicted by Black-Scholes (2.16), using numerical techniques to evaluate the integral.
3. Formulate a function $g(\sigma)$ from (2.18) using this predicted call option value and the current market price, and use the Newton-Secant method to calculate the implied volatility.
4. Compare and contrast this implied volatility with the observed volatility measured in point 1.

2.2 Historical volatility

2.2.1 Creating a mathematical vector class from the standard library

We will define a new class `MVector` that represents a mathematical vector of real numbers. Copy the following class definition for the new class `MVector` into a header file or at the top of your main code.

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>
```

```
// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}

    // access element (lvalue)
    double &operator[](int index) { return v[index]; }

    // access element (rvalue)
    double operator[](int index) const { return v[index]; }

    int size() const { return v.size(); } // number of elements

private:
    std::vector<double> v;
};

#endif
```

The class `MVector` has a single member variable, a `std::vector<double>` which stores the elements of the vector. The constructors, access operators and `size()` member function mimic these parts of the interface of `std::vector<double>`.

Now let us add a function to the class so that we can add data to the vector on input from a file. To do this we need to allow access to more of the functionality of the `std::vector`. Add the following function to your class:

```
// add data to your vector
void push_back(double x){v.push_back(x);}
```

Example: Using the MVector

Below is an excerpt from some code which inserts elements into the vector x and then prints them out to the screen.

```
// create an MVector
MVector x;
// add elements to the vector
x.push_back(1.3);
x.push_back(3.5);
// x now contains 1.3 and 3.5
// print x
std::cout << " x:= ( ";
for (int i=0; i<x.size(); i++) std::cout << x[i] << " ";
std::cout << ")" << std::endl;;
```

Tasks:

1. Add a member function to `MVector` that returns the average of all the elements in that vector. Test your results on small sample sets of data against an average calculated by hand.
2. Write a member function that returns a moving average of a `MVector`. The function prototype might look something like:

   ```
   MVector RunningAverage(int m, int p)
   ```

   If this function is called on an `MVector vec`, *i.e.*

   ```
   vec.RunningAverage(m, p);
   ```

   the number of elements returned will be smaller than the number of elements in `vec`, since in (2.1) we can evaluate x_{i+j} only when $i + j$ is in the range $0, \dots, n-1$. Test your function on small data samples.
3. Write a similar function that takes an `MVector` as a parameter and returns an `MVector` containing the exponential average of this parameter. Try different values for α. Describe any qualitative or quantitative differences between the two different types of average.
4. Write a function to return the variance σ^2 of the data stored in a vector **S**, assuming that the data you have follows a log normal process. See the algorithm in §2.1.1; the standard library function for natural logarithms is `std::log` in the `<cmath>` library.

2.2.2 Importing data sets

Example: Read/write numbers to/from a file

See the lecture 2 notes for the use of input streams in reading data from a file. For example,

```
#include <iostream>
#include <fstream>
double x = 1.0, y = 2.0;

int main()
{
// open an output file stream and write some data to it
    std::ofstream output("file.dat");
    if (!output)
    {
        std::cout << "Could not open file for writing" << std::endl;
    }
    else
    {
        output << 2*x << " " << 2*y << std::endl;
        output.close();
    }

// reopen the file with an input file stream to read back data
    std::ifstream input("file.dat");
    if (!input)
    {
        std::cout << "Could not open file for reading" << std::endl;
```

```
    }
    else
    {
        input >> x >> y;
        input.close();
        std::cout << x << " " << y << std::endl;
    }
    return 0;
}
```

Tasks:

- Try compiling and running this example code, and check that you can read and write from files.
- Download historical stock prices from the web[1]. To make things easy open the file in Excel or another spreadsheet package, then copy and paste a column with the stock prices (e.g. the closing price) into a new file (e.g. using notepad on Windows, gedit on Linux). Save the file as a **text** file (very important on Windows). If you wish to, try writing code that can read in the appropriate column of the `.csv` file directly, without needing to go through the spreadsheet.
- Write a program to read the first 5 entries from the text file and output them to screen. Check that the values are correct.
- The input stream has a method `eof()` which returns the value true when the end of the file has been reached. Use this to create a while loop that reads in all values from the file regardless of how big it is.
- Instead of outputting values to screen, create a MVector to store the data and add that data to the vector (you will need to store each element of the data temporarily to do this).
- Think about error checking. What happens if the data is not of the correct format (i.e. column heading left in, or no data entry available)?
- Finally use your functions from the first section to generate the variance of your data set. You have now generated the variance per day. How can this be converted to the variance per year and hence the volatility?

2.3 Numerical prediction of an option price

We now discus how to evaluate the price of a European option using the Black-Scholes solution (2.16). Using inheritance, we can allow for common elements of an option, such as the parameters, to be shared. Please review exercise sheets 3 and 4 for examples of implementing integration in C++.

2.3.1 Integration

The first task is to create a C++ function to calculate an integral of some integrand. The algorithm we choose to implement is Simpsons rule, given by:

$$\int_a^b f(y)dy \approx \frac{h}{3}\left[f(a) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)\right] \tag{2.20}$$

where $x_j = a + jh$ and $h = (b-a)/n$. Note that Simpson's rule is only valid if n is an even number. The integral we wish to evaluate is

$$I = \int_{y=0}^{1} Xe^{-r\sigma T}e^{Ty}dy \tag{2.21}$$

where $X = 10$, $r = 0.05$, $\sigma = 0.1$ and $T = 0.5$ are constants.

[1] At `http://finance.yahoo.com` use the 'Quote lookup' search box to find stock. Go to 'Historical Prices' in the menu on the left, and scroll to the bottom of the page for the 'Download to Spreadsheet' link.

Tasks:

- First, evaluate I algebraically. For the values of X, r, σ and T above, write the numerical value to 6 s.f.
- Create a new program, and enter the following code above your main function:

```
// integrand function
double f(double y)
{
  return 0;// fill in appropriate function here
}
// function to implement integration
double integrate(double a,double b,int n)
{
  double sum=0;
  // fill in Simpsons algorithm here
  return sum;
}
```

  Edit the code so that `f(y)` returns the value of function in (2.21). For the time being declare the constants inside the function. Edit the code so that `integrate(a,b,n)` returns the integrated value of `f(y)` between the limits `a` and `b` with `n` divisions.
- Calculate the solution for the integral I using with different values of n. Create a table comparing your results with the exact solution. Roughly what value of n is needed to get a solution accurate to 6 s.f.?
- Now change your functions so that the constants X, r, σ and T may be passed in as arguments to the `integrate` function. In one program run, calculate the value of the integral with $T = 0$, 0.25, 0.5, 0.75 and 1.

2.3.2 Procedural approach

In this section we shall evaluate the price of a European call option using a procedural approach, before repeating the process with an object orientated approach.

Tasks:

- Add three extra functions into your code, declaring their return type and arguments appropriately:
  - `payoff(y,X)` evaluates the transformed payoff from (2.10);
  - `A(x,r,sigma,T,k)` evaluates function A from (2.12);
  - `B(x,y,sigma,T,k)` evaluates function B from (2.13).

  Using the parameter values from §2.3.1, $y = 0.5$, and $S_0 = 10$, check that each of your functions are working by evaluating them by hand and comparing answers. You will need to calculate the value of x (from the transformation, equation 2.8) and the value of k (from equation 2.14) from other the other parameters before calling the functions.
- Now change your integrand function `f` to evaluate `B` multiplied by `payoff` as in (2.15). You will need to make sure that all the required parameters are passed into the funtion `f` as arguments. The definitions for `f` and `integrate` should now look something like:

```
// integrand function
double f(double x,double y,double X,double r,
  double sigma,double T,double k)
{
  // fill in appropriate function here
}
// function to implement integration
double integrate(double a,double b,int n,double x,
  double X,double r,double sigma,double T,double k)
{
  double sum=0;
  // fill in Simpsons algorithm here
  return sum;
}
```

- Now calculate the value of a Call option with $S_0 = 97$, $X = 100$, $r = 0.06$, $\sigma = 0.2$, $T = 0.75$ using your functions `A` and `integrate` as in (2.16). The limits of integration can be set to $a = -10$ and $b = 10$. Test your code by producing a table of your solution against the exact value (from some other source) against different values of n.

2.3.3 Object Orientated Approach

In this section we develop a protocol for the payoff function of an option using *pure virtual* functions. The class `Payoff` will basically be a definition of the function used to provide an interface. The class is defined entirely as follows:

```
// the option payoff class
struct Payoff
{
  virtual double operator()(double y)=0;
};
```

- a `struct` is a `class` where all members are public (C++ Primer §2.8),
- the syntax ` = 0` after the definition of the virtual `operator()` means that it is declared as a *pure* virtual function, with no function definition (function body) (C++ Primer §15.6).
- we cannot instantiate (make an object of) a class with a pure virtual function (an abstract base class). Instead we define a class which *derives* from the abstract base class and overrides its pure virtual function(s). We can then instantiate objects of this derived class.

Example: Share Value

Create a class that inherits `Payoff` and returns the value of the share as a payoff $F(S) = S$. Under the transformed variables it may be written as $f(y) = S_0 e^y$. The code is as follows:

```
// a call option implementation
class ShareValue: public Payoff
{
private:
    double S0;
public:
    ShareValue(double S0_):S0(S0_){}
    double operator()(double y){
      return S0*exp(y);
    }
};
```

Declare the class in the code as

```
// Declare a ShareValue with scaled value S0=10
ShareValue share(10.);
// output from this line will be 10, since
// 10.*exp(0)=10.
std::cout << share(0.) << std::endl;
// output 3.67879
std::cout << share(-1.) << std::endl;
// output 27.1828
std::cout << share(1.) << std::endl;
```

Tasks:

- Copy the following code for the payoff class at the top of your program and add a definition for a call option underneath (using the example for `ShareValue` as a template) and complete the payoff function for the call option (in transformed variables see equation 2.10). Check that your code still compiles and runs.
- Create a new function `f_object` by copying and pasting the function `f` in your code and renaming it `f_object` . Add `Payoff &payoff` to the argument list in `f_object`. Since the `payoff` function we use inside the copied function is from `Payoff` class, it will just take `y` as an argument rather than `y` and `X`. Test the function `f_object` by calling it from your main function and compare it with the values from `f`. You will need to declare a `CallOption` object to pass in as the argument (since it inherits the payoff function) and set the value of the variable `X` in the object with the declaration statement.
- Create a new function `integrate_object` by copying and pasting the function `integrate` and renaming it. Add `Payoff &payoff` to the argument list in `f_object`. Inside the function `integrate_object` , now use `f_object` instead of `f` as the integrand, you must pass the `payoff` function as an argument on every function call. Test the function `integrate_object` by calling it from your main function. Do you get the same answers as before?
- Try making a `PutOption` class that inherits the `Payoff` class and generate values of the option.
- Write a single function to return the value of an option, your function definition should look something like:

  ```
  double optionValue(double S,double X,double r,double sigma,
                     double T,Payoff &payoff)
  ```

  You can calculate the value of x and k within the function.
- Try making a class that stores all the methods and parameters required to generate option prices.

2.4 Implied Volatilities

We start by implementing the Newton-Secant root-finding algorithm on an simple function $f(x)$, which we choose to have roots in known locations. Here we choose to solve

$$f(x) = x^2 - 2, \tag{2.22}$$

which satisfies $f(x) = 0$ at

$$x = \pm\sqrt{2}. \tag{2.23}$$

The Newton-Secant method for finding a root of $f(x)$ uses the iteration:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}. \tag{2.24}$$

To implement this algorithm in C++, our first task is to implement the function $f(x)$

```
double f(double x) // returns f(x)
{
    return x*x-2.0;
}
```

We then need to implement the Newton-Secant method. Recall that the Newton-Secant method requires two initial guesses for x, which we specify in the parameters of the function.

```
bool NewtonSecant(double &result, double x0, double x1, double tol, int
    maxiter)
{
    // Set up variables here
    int i;
    for (i=0; i<maxiter; i++)
    {
        // Newton-Secant iteration here
        if (std::abs(f(x))<tol) break;
    }
    result = //... result here

    if (i == maxiter)
        return false;
    else
        return true;
}
```

Note that:

- The framework for the algorithm has been provided, but the details have been left for you to fill in.
- We provide a tolerance `tol` such that the iteration stops when $|f(x)| <$ tol. This should be set to some small value, such as 10^{-6}.
- We also set a maximum number of iterations, `maxiter`, after which the iteration stops even if a root has not been found.
- The function returns a `bool` value (true or false) indicating whether the algorithm has succeeded. We say that the algorithm succeeded if the function ended because $|f(x)| <$ tol, and that it failed if the number of iterations reached `maxiter` without this criterion on $|f(x)|$ being satisfied.
- The function must also return the result of the computation, and we do this through the `result` parameter, which is passed by reference.

Tasks:

- Complete the code above for the Newton-Secant method.
- Test your code to solve $f(x) = x^2 - 2 = 0$. Try different initial guesses. Does the method always converge?
- The implied volatility is calculated by setting the the option value $V(S, X, r, t, T, \sigma)$ to be equal to the market value of the option, $\hat{V}(S, X, t, T)$. The former is calculated from the Black-Scholes model and the object oriented approach presented in Section 1.3.3. Regarding the market price $\hat{V}(S, X, t, T)$ of the option, it is calculated for selected call and put options for the **same companies used for the calculation of the historical volatility**. The market value is equal to (bidprice + askprice)/2, where bid and ask prices are obtained from either `http://finance.yahoo.com/` or `http://www.cboe.com/delayedquote/quotetable.aspx` for

each option set of the selected stock. The current value of the underlying asset, S_t, is the last value of the stock time series for the companies selected in Section 1.2.2 from the same sources (select the adjusting one for which dividends are removed). The exercise price of the option, X, is given in the same sources for the selection of the option sets made above. The time to maturity, $T - t$, is equal to the number of days until expiry divided by 365. Finally, the risk-free interest rate, r, can be selected close to 0.2% or 0.002 according to `http://www.forexltd.co.uk/analytic/rates`. Therefore, the implied volatility is the only unknown that you need to calculate.

- Write a function to return $g(\sigma)$, the difference between an option value and the market value, as defined in (2.18). You will need to add in all the relevant parameters for the option in the argument list, including the payoff.
- Plot $g(\sigma)$ for a range of sigma, to verify that the your function $g(\sigma)$ does have a root.
- Edit your secant method algorithm to find the root of the function $g(\sigma)$, again all relevant parameters must be added to the argument list. You may wish to
- Consider a call option for a company as an example with $S = 30.23$, $X = 29$, $\sigma = 0.002$, $\hat{V} = 1.425$ and $T = 0.1205$ (data selected from *cboe* source). Your code should give as a result for the implied volatility of the call option a value equal to 0.1512 or 15.12%. Verify that it does so.
- Consider a European call option with a value today of \$0.05 on an underlying whose price today ($S(t = 0)$) is \$1. Expiry is in 6 months time, the risk-free rate is 5% (this may be assumed fixed), and the strike is \$1. Using the `secantMethod` function evaluate the implied volatility of this option.

2.5 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 20 marks available for correct numerical results and code. The other 20 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

1. Describe the task of finding implied volatility, the methods used to do so, and give a description of any classes used in your code.
2. For a selection of stocks of your choice, plot the time series of the value, the running average and the exponential average. Make comments on the differences between them and how well they can spot trends in the data. **[4]**
3. Using the same selection of stocks as above, find the implied volatility for a selection of different options. You may obtain your option-price data from any (published) source (but this must be clearly stated in your report). One source (quite comprehensive) is:

   `http://finance.yahoo.com/`

   where you need to type the name of the company and select the 'Options' tab.

   Another source could be:

   `http://www.cboe.com/delayedquote/quotetable.aspx`

   which has delayed option price data. The website is a bit busy but looks quite user friendly. If you type in the ticker symbol for a stock (e.g. IBM) and then click on 'submit' on the tab it gives you quotes for option prices. The codes are a bit tricky to decipher but in general, the first three digits identify the company (e.g. IBM), followed by the current date and then the expiry date. Then, the next digit is a letter identifying the expiry month (different for call and put option, e.g. January is A for call option and M for put option), followed by the strike price of the option.

   It is advisable to select options with high volume from either source and ignore any values given already for the implied volatility in the website(s) since they are calculated with different algorithmic procedures. In addition, it is possible that you will see 'no data currently available' or zero values while searching which implies that the market is closed and no contract is traded. You need to try and get your values when the financial markets are open (in UK time, i.e. from 2-3 pm until 10pm on working days). There should be an informative message indicating the remaining period needed until the markets are open.

   Finally, you could use the following source to verify your calculations for the option prices:

   `www.maths.manchester.ac.uk/~pjohnson/pages/blackscholesCalculator.html`

   You may, of course, use other (better?) sources for the selection of the stock and option sets, but these should be stated clearly in your report. **[4]**

4. Demonstrate that your integration routine (using Simpson's rule) is correct by showing that it can integrate (2.21), and that the results converge to the algebraic solution of this integral. **[4]**
5. Validate your code for the implied volitility by doing the tasks in section 2.4. Plot the implied volatility of options against different strike prices for fixed time to maturity and interest rate. **[4]**
6. Include tables of your historical volatilities versus implied volatilities, and describe any differences. **[4]**

Marks will be awarded for clarity and correctness of code as well as answers to the questions and discussion.

Project 3

The Conjugate Gradient Method

Original author: Dr. Paul Johnson

In this project we will solve the matrix equation $A\mathbf{x} = \mathbf{b}$ where $\mathbf{x}$ is the unknown using an iterative solver, the conjugate gradient (CG) method. Such iterative methods are widely used for matrix equations that arise from the discretisation of partial differential equations, where A is large (perhaps $10^6 \times 10^6$), but sparse, with maybe 100 or fewer non-zero elements per row.

3.1 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 1970s. The method is an iterative algorithm that (in the absence of roundoff error) converges to the *exact* solution of the linear system in a number of iterations no larger than the dimension of the system. In practice, sufficient accuracy is usually reached well before this number of iterations. The conjugate gradient method is often used in conjunction with a technique called *preconditioning*, which increases the rate of convergence.

We will solve

$$A\mathbf{x} = \mathbf{b}, \tag{3.1}$$

where A is an $n \times n$ symmetric and positive definite matrix. (Recall that A is symmetric iff $A^T = A$ and positive definite iff $\mathbf{x} \cdot A \cdot \mathbf{x} > 0$ for all non zero vectors $\mathbf{x}$.) Any positive definite matrix is invertable and so (3.1) has a unique exact solution, which we denote by $\mathbf{x}_e$.

The conjugate gradient method iteratively finds the $\mathbf{x}$ that minimises the function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x} \cdot A \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x}. \tag{3.2}$$

This function is minimised when its gradient is zero,

$$\nabla f(\mathbf{x}) = A \cdot \mathbf{x} - \mathbf{b} = 0, \tag{3.3}$$

meaning that minimising (3.2) is equivalent to solving our matrix equation (3.1). Any iterative solution to (3.1) can be written an initial guess of the solution $\mathbf{x}_0$ and an iteration

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \Delta\mathbf{x}_k \\ &= \mathbf{x}_k + \alpha_k \mathbf{p}_k, \end{aligned} \tag{3.4}$$

where (without loss of generality) we have defined the difference between successive steps $\Delta\mathbf{x}_k$ as the product of a direction vector $\mathbf{p}_k$ and a scalar step size α_k. For the iteration (3.4) to be useful we require that

1. $\mathbf{x}_k \to \mathbf{x}_e$ as $k \to \infty$, so that the iteration converges to our desired solution, and
2. α_k and $\mathbf{p}_k$ can be expressed in terms of quantities known at stage k of the iteration.

In addition, it is desirable that:

3. the convergence of the algorithm is rapid, i.e. our choices of $\mathbf{p}_k$ and α_k are such that that $||\mathbf{x}_k - \mathbf{x}_e||$ becomes small for relatively modest values of k.

We leave the problem of finding suitable directions $\mathbf{p}_k$ for now, and look instead at finding suitable step sizes α_k assuming that $\mathbf{p}_k$ is known. From (3.2) and (3.4) the value of f at step $k+1$ is

$$\begin{aligned} f(\mathbf{x}_{k+1}) &= f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) && (3.5)\\ &= \frac{1}{2}(\mathbf{x}_k + \alpha_k \mathbf{p}_k)\cdot A \cdot (\mathbf{x}_k + \alpha_k \mathbf{p}_k) - \mathbf{b}\cdot(\mathbf{x}_k + \alpha_k \mathbf{p}_k) && (3.6)\\ &= f(\mathbf{x}_k) + \frac{\alpha_k^2}{2}\mathbf{p}_k \cdot A \cdot \mathbf{p}_k + \alpha_k (A\cdot \mathbf{x}_k - \mathbf{b})\cdot \mathbf{p}_k. && (3.7)\end{aligned}$$

To minimise $f(\mathbf{x}_{k+1})$ we differentiate this with respect to α_k and equate to zero,

$$\frac{\mathrm{d}}{\mathrm{d}\alpha_k} f(\mathbf{x}_{k+1}) = \alpha_k \mathbf{p}_k \cdot A \cdot \mathbf{p}_k + (A \cdot \mathbf{x}_k - \mathbf{b})\cdot \mathbf{p}_k = 0, \tag{3.8}$$

which implies that

$$\alpha_k = \frac{(\mathbf{b} - A\cdot \mathbf{x}_k)\cdot \mathbf{p}_k}{\mathbf{p}_k \cdot A \cdot \mathbf{p}_k} \tag{3.9}$$

minimises $f(\mathbf{x}_{k+1})$ for our chosen $\mathbf{p}_k$. Substituting this value of α into (3.7) we find

$$\begin{aligned} f(\mathbf{x}_{k+1}) &= f(\mathbf{x}_k) - \frac{[(\mathbf{b} - A\cdot\mathbf{x}_k)\cdot A \cdot \mathbf{p}_k]^2}{\mathbf{p}_k \cdot A \cdot \mathbf{p}_k} && (3.10)\\ &\le f(\mathbf{x}_k). && (3.11)\end{aligned}$$

So, given a direction for our step $\mathbf{p}_k$, we now know from (3.9) the size of step to take α_k, in order to minimise f.

We now seek a procedure for finding a set of the direction vectors $\mathbf{p}_k$ that results in rapid convergence of the algorithm. We first write our desired exact solution $\mathbf{x}_e$ in terms of a basis $\{\mathbf{e}_i\}$,

$$\mathbf{x}_e = \sum_i \varphi_i \mathbf{e}_i. \tag{3.12}$$

We then have

$$\mathbf{b} = A \cdot \mathbf{x}_e = \sum_i \varphi_i A \cdot \mathbf{e}_i, \tag{3.13}$$

and so for any basis vector $\mathbf{e}_j$,

$$\mathbf{e}_j \cdot \mathbf{b} = \sum_i \varphi_i (\mathbf{e}_j \cdot A \cdot \mathbf{e}_i). \tag{3.14}$$

Now, suppose we choose our basis vectors to be *mutually conjugate*, *i.e.* that

$$\mathbf{e}_j \cdot A \cdot \mathbf{e}_i = 0, \quad \text{when } i \neq j. \tag{3.15}$$

Then (3.14) simplifies to

$$\mathbf{e}_j \cdot \mathbf{b} = \varphi_j (\mathbf{e}_j \cdot A \cdot \mathbf{e}_j), \tag{3.16}$$

and hence we can easily evaluate

$$\varphi_j = \frac{\mathbf{e}_j \cdot \mathbf{b}}{\mathbf{e}_j \cdot A \cdot \mathbf{e}_j}. \tag{3.17}$$

By evaluating each of the φ_j we could in theory compute $\mathbf{x}_e$ through (3.12). This is exactly the this computation performed by the iterative procedure (3.4), if we choose our direction vectors $\mathbf{p}_i$ to be equal to our conjugate basis vectors $\mathbf{e_i}$ and choose α_i through (3.9). At each stage of the iteration, by incrementing our iterative solution by $\alpha_k \mathbf{e}_k$ we exactly compute one component φ_i of our solution vector (note the similarity between (3.9) and our the expression for φ, (3.17)). After n iterations (where n is the size of our system) we will have found all the φ_i, and thus have solved our problem exactly.

Although this algorithm does converge, and is realisable as an iteration (conditions 1 and 2 above), for most sets of conjugate basis vectors $\{\mathbf{e}_i\}$ the convergence is very slow (it does not satisfy condition 3). That is to say, we might need to take a significant fraction of n steps before $\mathbf{x}_k$ is a reasonably close approximation to our solution $\mathbf{x}_e$. Performing $O(n)$ iterations is impractical for large problems (recall that $n \approx 10^6$ is typical), and so we would like to chose a conjugate basis $\{\mathbf{e}_i\}$ such that $\mathbf{x}_k$ closely approximates $\mathbf{x}_e$ within the first few iterations.

Thinking in terms of the minimisation of the function f, a sensible choice of direction for the first step is the direction at which f decreases most steeply, namely

$$\mathbf{p}_0 = -\nabla f(\mathbf{x_0}) = \mathbf{r}_0, \tag{3.18}$$

where we define the residual

$$\mathbf{r}_i = \mathbf{b} - A \cdot \mathbf{x_i}. \tag{3.19}$$

It is tempting to think that

$$\mathbf{p}_i = \mathbf{r}_i \tag{3.20}$$

is good choice for the subsequent directions $\mathbf{p}_i$ as well, but this choice does not lead to the $\mathbf{p}_i$ being mutually conjugate[1]. Instead, we construct the direction vectors for $i \geq 1$ by

$$\mathbf{p}_i = \mathbf{r}_i - \sum_{j<i} \frac{\mathbf{p}_j \cdot A \cdot \mathbf{r}_i}{\mathbf{p}_j \cdot A \cdot \mathbf{p}_j} \mathbf{p}_j. \tag{3.21}$$

This construction of mutually conjugate vectors is a Gram-Schmidt process using the inner product $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u} \cdot A \cdot \mathbf{v}$ — we are setting $\mathbf{p}_i$ to $\mathbf{r}_i$, then subtracting off the components of $\mathbf{r}_i$ that are not conjugate to the previously-found direction vectors.

3.1.1 Formulation

Equipped with the expression for $\mathbf{p}_i$ through (3.21), we are in a position to write down an algorithm for the Conjugate Gradient method:

- Initialise the method with a guess $\mathbf{x}_0$. Set

$$\begin{aligned}\mathbf{r}_0 &= \mathbf{b} - A\mathbf{x}_0, \\ \mathbf{p}_0 &= \mathbf{r}_0, \end{aligned} \tag{3.22}$$

- Now iterate for $k = 0, 1, 2, \ldots$, setting

$$\alpha_k = \frac{\mathbf{r}_k \cdot \mathbf{r}_k}{\mathbf{p}_k \cdot A \cdot \mathbf{p}_k}, \tag{3.23}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{3.24}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A \mathbf{p}_k, \tag{3.25}$$

$$\beta_k = \frac{\mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k \cdot \mathbf{r}_k}, \tag{3.26}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k. \tag{3.27}$$

Several of the equations here are written in a different form to those in the section above; for example, (3.21) is rewritten as (3.27) to avoid the calculating the sum. It is relatively straightforward to show by induction that the definitions are equivalent (try it if you are not convinced!).

3.2 Coding, Examples and Exercises

3.2.1 Creating a Vector class

We will define a new class `MVector` that represents a mathematical vector of real numbers. By overloading operators for this class (for example, operators for adding vectors or multiplying vectors by scalars), we can write the code for the our (vector) ODE integration routines much as we would write it in mathematical notation[2].

[1]The choice (3.20) leads to the *steepest descent* algorithm, which for most problems is inferior to conjugate gradients.

[2]If our goal was maximum performance we might instead use a library such as ATLAS or BLAS, which are optimised for speed.

Copy the following definition of the class `MVector` into a header file or at the top of your main code. The code is available on the course website (in the Example Code section) as `mvector.h`.

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>

// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}

    // access element (lvalue)
    double &operator[](int index) { return v[index]; }

    // access element (rvalue)
    double operator[](int index) const { return v[index]; }

    int size() const { return v.size(); } // number of elements

private:
    std::vector<double> v;
};

#endif
```

The class `MVector` has a single member variable, a `std::vector<double>` which stores the elements of the vector. The constructors, access operators and `size()` member function mimic these parts of the interface of `std::vector<double>`.

We now overload the arithmetic operators `+`, `-`, `/`, `*` for our new `MVector` class. These operators are typically overloaded in non-member functions (see C++ Primer, §§14.1 and 14.3), and so the operator overload is placed outside the class definition, but inside the header file.

A typical implementation will look like

```
// Operator overload for "scalar * vector"
inline MVector operator*(const double& lhs,const MVector& rhs)
{
    MVector temp(rhs);
    for (int i=0;i<temp.size();i++) temp[i]*=lhs;
    return temp;
}
```

when placed in the header file (note the use of the `inline` keyword, since the function is in the header file[3].)

[3]We could instead place the function definition above in a separate `.cpp` file, and leave just the function declaration (prototype) in the header. If this is done, the `inline` keyword is not used in either the definition or declaration.

Tasks:

1. For the four operators `+`, `-`, `/`, `*` there are five overloads to implement. Remember that we can multiply/divide a vector by a scalar, and add/subtract vectors from vectors, but can't add/subtract a scalar from vector nor divide a scalar by a vector. What are the prototypes of the five overloads required?
2. Implement these overloads in your code.
3. Check that the code is working by evaluating the following using `MVector`s to represent $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{x}$:

$$\mathbf{u} = 4.7\mathbf{v} + 1.3\mathbf{w} - 6.7\mathbf{x}$$

   where $\mathbf{v} = (0.1, 4.8, 3.7)$, $\mathbf{w} = (3.1, 8.5, 3.6)$ and $\mathbf{x} = (5.8, 7.4, 12.4)$.
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 second constructor? (See C++ Primer, §12.4.4).
5. When adding two vectors, check they have the same number of elements and exit with an error if they do not.
6. You could also try overloading the << operator to output a vector in the form `(v[0], v[1], ..., v[n])`:

```
// Overload the << operator to output MVectors to screen or file
ostream& operator<<(ostream& os, const MVector& v)
{
    int n = v.size();
    os << "(";
    // write the code to output the vector components here.
    os << ")";
    return os;
}
```

3.2.2 More Useful Vector Functions

It will be useful to define two further member functions to calculate the norm of a vector. We will implement the infinity norm

$$||\mathbf{x}||_\infty = \max(|x_0|, |x_1|, \ldots, |x_n|), \tag{3.28}$$

and the L^2 norm

$$||\mathbf{x}||_2 = \sqrt{x_0^2 + x_1^2 + \cdots + x_n^2}. \tag{3.29}$$

We will also need a function to calculate the dot product of two vectors,

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n} x_i y_i. \tag{3.30}$$

Example:

Add a member function definition

```
double LInfNorm() const;
```

to the `MVector` class that returns the L^∞ norm of that vector, and write the implementation. You may find it useful to use the `std::max` function (in the `<algorithm>` header) and/or the `std::abs` function (in the `<cmath>` header).

Solution:

The function may be written as follows:

```
double LInfNorm() const
{
    double maxAbs = 0;
    std::size_t s = size();
    for (int i=0; i<s; i++)
    {
        maxAbs = std::max(std::abs(v[i]), maxAbs);
    }
    return maxAbs;
}
```

Tasks:

1. Add the code for the `LInfNorm` function above to your program and test it.
2. Write your own member function to evaluate the L^2 norm.
3. Write a non-member function to calculate the dot product, with the prototype

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

4. Test your dot product function by evaluating

   $$\alpha = \frac{\mathbf{u}\cdot\mathbf{u}}{\mathbf{v}\cdot\mathbf{w}} \tag{3.31}$$

   where $\mathbf{u} = (1.5, 1.3, 2.8)$, $\mathbf{v} = (6.5, 2.7, 2.9)$ and $\mathbf{w} = (0.1, -7.2, 3.4)$. The value of alpha should be -1.31915 (to 6 s.f.).
5. Test your L^2 norm function on the three vectors $||\mathbf{u}||_2 = 3.4322$, $||\mathbf{v}||_2 = 7.61249$, and $||\mathbf{w}||_2 = 7.96304$.

3.2.3 Creating a Matrix class

We will also need a class for matrices, available on the course website as `mmatrix.h`:

```
#ifndef MMATRIX_H // the 'include guard'
#define MMATRIX_H

#include <vector>

// Class that represents a mathematical matrix
class MMatrix
{
public:
    // constructors
    MMatrix() : nRows(0), nCols(0) {}
    MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n * m, x)
     {}

    // set all matrix entries equal to a double
    MMatrix &operator=(double x)
```

```
    {
        for (int i = 0; i < nRows * nCols; i++) A[i] = x;
        return *this;
    }

    // access element, indexed by (row, column) [rvalue]
    double operator()(int i, int j) const
    {
        return A[j + i * nCols];
    }

    // access element, indexed by (row, column) [lvalue]
    double &operator()(int i, int j)
    {
        return A[j + i * nCols];
    }

    // size of matrix
    int Rows() const { return nRows; }
    int Cols() const { return nCols; }

private:
    unsigned int nRows, nCols;
    std::vector<double> A;
};

#endif
```

Internally the matrix entries are stored in a `std::vector<double>`, in *row-major* order – look at the `operator()` functions to see how the matrix indices i and j correspond to an index into the internal `std::vector`. Note that the class uses zero-based indexing, following the C++ convention, rather than the one-based indexing used in mathematics (the top-left element of an `MMatrix m` is accessed by `m(0,0)`, rather than `m(1,1)`).

Tasks:

1. Declare a new 4×3 matrix in your code. Assign some values to the matrix (use `m(i,j)` to access elements), display them on the screen.
2. We will need to compute matrix-vector products using the `MMatrix` and `MVector` classes. We can overload the operator `MMatrix::operator*` to do this, using the prototype

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

   Add this definition to the `MMatrix` class and implement the function.
3. Let $A = a_{i,j}$ be a 4×3 matrix, such that $a_{i,j} = 3 * i + j$ for $0 \le i \le 3$ and $0 \le j \le 2$ and let $\mathbf{x} = (0.5, 1.6, 3.2)$. Then test your function by evaluating

   $$\mathbf{b} = A \cdot \mathbf{x} \tag{3.32}$$

   Check your solution for **b** by hand or with another piece of software, such as MATLAB/Octave.

Additional Tasks:

1. Try overloading the << operator of the `MMatrix` class so that the matrix can be displayed using `std::cout` or written to a file. You may find the `width(n)` and `precision(n)` methods of output stream objects (such as `std::cout`) useful in getting the columns to line up correctly – see the `stream_formatting.cpp` example file on the course website.

3.2.4 Implementing the Conjugate Gradient method

We will now use the `MMatrix` and `MVector` classes to solve

$$A \cdot \mathbf{x} = \mathbf{b} \tag{3.33}$$

using the Conjugate Gradient method, where A is an $n \times n$ matrix and $\mathbf{x}$, $\mathbf{b}$ are n-element vectors. As an example we will use a (symmetric, positive-definite) matrix A that arises from a finite-difference discretisation of Poisson's equation in one dimension,

$$\frac{\partial^2 \varphi}{\partial x^2} = g(x). \tag{3.34}$$

The matrix entries[4] are

$$A = a_{i,j} = \begin{cases} 2 & \text{if} \quad i = j, \\ -1 & \text{if} \quad |i-j| = 1, \\ 0 & \text{otherwise,} \end{cases} \tag{3.35}$$

and the vector $\mathbf{b}$ is given by

$$\mathbf{b} = b_i = \frac{1}{(n+1)^2}, \tag{3.36}$$

corresponding to $g(x)$ being a constant.

Tasks:

1. Write a function to set the elements of an $n \times n$ `MMatrix` according to (3.35).
2. Test your function on a 5×5 matrix, displaying the matrix to screen to show that it has the correct entries.
3. Using this matrix, calculate the initial residual $\mathbf{r}_0 = \mathbf{b} - A \cdot \mathbf{x}_0$ for an initial guess $\mathbf{x}_0$ of your choosing. Check that $\mathbf{r}_0 = \mathbf{b}$ if you use use $\mathbf{x}_0 = 0$.
4. Write a function to implement the conjugate gradient method, as outlined in (3.22) and (3.23–3.27). Your code will probably look similar to the following (with comments replaced by appropriate implementation):

```
int maxIterations = 1000;
double tolerance = 1e-6;

// ...initialise vectors here...

for (int iter=0; iter<maxIterations; iter++)
{
   // ...calculate new values for x and r here...

   // check if solution is accurate enough
   if (r.L2Norm() < tolerance) break;

   // ...calculate new conjugate vector p here...
```

[4]This example matrix is actually *tridiagonal*, a special case for which there is a very efficient method of solving $A \cdot \mathbf{x} = \mathbf{b}$, the Thomas algorithm.

```
}
```

5. Use this function to solve $A \cdot \mathbf{x} = \mathbf{b}$, with A and $\mathbf{b}$ defined by (3.35) and (3.36) respectively, an initial guess $\mathbf{x_0} = 0$ and with `tolerance` $= 10^{-6}$. With $n = 25$ the method should take 12 iterations to converge.
6. Plot x_i versus i for $n = 10, 25, 100$. Try plotting x_i against $(i+1)/(n+1)$ (where i runs between 0 and $n-1$). What do you notice?
7. Alter your function 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).
8. Change the matrix A to be

$$A = a_{i,j} = \begin{cases} 2(i+1)^2 + m & \text{if} \quad i = j \\ -(i+1)^2 & \text{if} \quad |i-j| = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{3.37}$$

with the vector $\mathbf{b} = b_i = 2.5$ and $m = 10.$ and rerun your results. What happens to the rate of convergence as you vary the value of m, particularly if m is small or even 0? Can you find out why this might happen? See, for example Shewchuk (1994) and Saad (2003).

Report:

- Include annotated code of the `MVector` and `MMatrix` classes, and your CG implementation
- Include the figure of x_i against i for different values of n.
- Include the table showing 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 are zero, especially as n gets large. A more efficient way of storing the matrix would be to just store the three diagonals with non-zero entries, rather than the whole matrix.

A *banded* matrix is one in which the nonzero entries lie on only a small number of diagonals. If B is a $n \times n$ banded matrix, with l non-zero diagonals below / to the left of the main diagonal and r non-zero diagonals above / to the right, e.g.

$$B = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & & \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & \\ & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ & & b_{3,2} & b_{3,3} & b_{3,4} \\ & & & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} 1 & 6 & 10 & & \\ 13 & 2 & 0 & 11 & \\ & 14 & 3 & 8 & 12 \\ & & 0 & 4 & 9 \\ & & & 16 & 5 \end{bmatrix} \tag{3.38}$$

(here $l = 1$ and $r = 2$), then we can write all of the nonzero terms in a $n \times (l + r + 1)$ matrix $B^\star$,

$$B^\star = \begin{bmatrix} b^\star_{0,0} & b^\star_{0,1} & b^\star_{0,2} & b^\star_{0,3} \\ b^\star_{1,0} & b^\star_{1,1} & b^\star_{1,2} & b^\star_{1,3} \\ b^\star_{2,0} & b^\star_{2,1} & b^\star_{2,2} & b^\star_{2,3} \\ b^\star_{3,0} & b^\star_{3,1} & b^\star_{3,2} & b^\star_{3,3} \\ b^\star_{4,0} & b^\star_{4,1} & b^\star_{4,2} & b^\star_{4,3} \end{bmatrix} = \begin{bmatrix} & 1 & 6 & 10 \\ 13 & 2 & 0 & 11 \\ 14 & 3 & 8 & 12 \\ 0 & 4 & 9 & \\ 16 & 5 & & \end{bmatrix}. \tag{3.39}$$

The terms $b^\star_{i,j}$ of $B^\star$ are given by

$$b^\star_{i,j+l-i} = b_{i,j}. \tag{3.40}$$

If $l + r + 1 \ll n$ the matrix $B^\star$ can be stored in significantly less space than B.

In the following class definition (available on the course website as `mbandedmatrix.h`) we define such a banded matrix:

```
#ifndef MBANDEDMATRIX_H // the 'include guard'
#define MBANDEDMATRIX_H

#include <vector>
class MBandedMatrix
{
public:
    // constructors
    MBandedMatrix() : nRows(0), nCols(0) {}
    MBandedMatrix(int n, int m, int lband, int rband, double x = 0) :
        nRows(n), nCols(m), A(n * (lband + rband + 1), x), l(lband), r(
    rband) {}


    // access element [rvalue]
    double operator()(int i, int j) const
    {
        // fill this in...
    }

    // access element [lvalue]
    double &operator()(int i, int j)
    {
        // fill this in...
    }
    // size of matrix
    int Rows() const { return nRows; }
    int Cols() const { return nCols; }

    int Bands() const { return r + l + 1; } // total number of bands
    int LBands() const { return l; } // number of left bands
    int RBands() const { return r; } // number of right bands
private:
    unsigned int nRows, nCols;
    std::vector<double> A;
    int l, r; // number of left/right diagonals
};

#endif
```

Example:

- Overload the << operator to output the matrix.

Solution:

First define the operator as follows:

```
std::ostream& operator<<(std::ostream& output,const MBandedMatrix&
    banded);
```

Then the function implementation will be

```
std::ostream& operator<<(std::ostream& output, const MBandedMatrix&
    banded)
{
    int r = banded.Rows(), c = banded.Cols();
    for (int i = 0; i < banded.Rows(); i++)
    {
        // calculate position of lower and upper band
        int jmin = std::max(std::min(i-banded.LBands(), banded.Cols()),0)
    ;
        int jmax = std::min(i+banded.RBands()+1, banded.Cols());

        output << "( ";
        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<c; j++)
            output << 0 << "\t ";
        output << ")\n";
    }
    return output;
}
```

Tasks:

1. Copy the code for the banded matrix and the overload of `operator<<`. Fill out the access functions to elements in the matrix (the overloads of `operator()`). Note that the parameters `i` and `j` of these access functions refer to the rows and columns of the matrix B in (3.38), not the rows and columns of the compressed form B^*. Test these by making a new `MBandedMatrix`, assigning some values to it, and checking that these entries are correct when you display the matrix.
2. Now make the matrix A from (3.35) a banded matrix (this matrix has $l = 1$ and $r = 1$). Setup the matrix, and display it to the screen to check that the elements are as expected.
3. Create a function with the prototype

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

   to implement matrix vector multiplication for `MBandedMatrix` matrices. Complete the implementation of the function, making sure that it is reasonably efficient (don't multiply by zero entries in the banded matrix that are outside the defined bands).
4. Using the values for A, $\mathbf{x}$ and $\mathbf{b}$ from §3.2.4, calculate the residual $\mathbf{r}$. Check your answer against the values obtained previously.

5. Now run the (CG) algorithm on the banded matrix. Other than changing some types to `MBandedMatrix` this should not involve any changes to the routine, since we have overloaded the matrix-vector multiplication for banded matrices.
6. Create a table to show the number of iterations and computation time for different values of n.

Report:

- Include commented code for the `MBandedMatrix` class.
- Include the table showing number of iterations and computation time for the two matrix classes, and interpret the results.
- Compare and contrast the different methods for storing a matrix.

3.3 Larger systems

Let u be the solution to the equation

$$\nabla^2 u(s,t) = 1, \tag{3.41}$$

in two dimensions. Now define the vector $\mathbf{x}$ with n^2 components as an approximation of u at discretised values of $s = s_k$, $t = t_l$, such that

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

Then $\mathbf{x}$ is an approximation to (3.41) if it solves the matrix equation

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

where A is an $n^2 \times n^2$ matrix with elements

$$A = a_{i,j} = \begin{cases} 4 & \text{if} \quad i = j \\ -1 & \text{if} \quad |i-j| = n \\ -1 & \text{if} \quad |i-j| = 1 \text{ and} \\ & \qquad (i+j) \mod (2n) \neq 2n-1 \\ 0 & \text{otherwise,} \end{cases} \tag{3.42}$$

and $\mathbf{b} = b_i = 1/(n+1)^2$ is a vector with n^2 components. Note that in the definition of A, (3.42) we use the zero-based indexing convention of C++, i.e. where $i = j = 0$ corresponds to the top left entry of the matrix, denoted $a_{0,0}$.

Tasks:

1. Create a `MMatrix` representing the matrix A with $n = 5$, and display it to screen to check that you it is correct[5].
2. Solve the problem for $n = 5$ with your CG solver.
3. Repeat steps 1 and 2, but implementing A in an `MBandedMatrix`.
4. Once you are sure that your systems for $n = 5$ are correct, solve the systems for both types of matrix with n up to 100 or more. Create a table or figure showing iterations to converge and computation times for each class. If your CG method does not find a value, state this in your report and try to explain why this is the case. Note: Before running calculations on large matrices you may wish to look at the notes on the website about turning on compiler optimisations.
5. Try plotting the results using a contour plot, or even as a 3D graph, by outputting your data as a text file with data in a matrix format,

$$\begin{matrix} x_0 & x_1 & \dots & x_{n-1} \\ x_n & x_{n+1} & \dots & x_{2n-1} \\ \vdots & \vdots & \dots & \vdots \\ x_{n(n-1)} & x_{n(n-1)+1} & \cdots & x_{n^2-1} \end{matrix}$$

[5] Ask for a solution if you want to check that you have understood (3.42) properly.

In gnuplot, plot this with

```
gnuplot> set hidden3d
gnuplot> splot 'my_file.txt' matrix with lines
```

In MATLAB/Octave, use

```
>> dat = load('my_file.txt');
>> surf(dat);
```

3.4 Report

Write your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 20 marks available for correct numerical results and code. The other 20 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

1. State briefly the theory behind the conjugate gradient method.
2. Give a description of any classes used, alongside code listings, describing what they are used for. **[5]**
3. Include tables or figures of convergence and computation times for the problems stated in the report. **[5]**
4. Try to explain why certain problems take many iterations to converge using the Conjugate Gradient method.
5. Include some contour plots or 3D plots of the solutions in section 3. **[5]**
6. Compare and contrast your two matrix storage classes for the problems stated in §3.3. **[5]**

Bibliography

Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003. URL http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. 1994. URL http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

Project 4

Sorting algorithms

Original author: Dr. Andrew Hazel

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 will implement some standard sorting algorithms with increasing levels of abstraction. The sorting methods can then be used to implement the 'Game of Life' using very little memory.

4.1 Sorting algorithms

4.1.1 The sorting problem

The *sorting problem* is to find a permutation π of a sequence of n elements $a_0, a_1, \cdots, a_{n-1}$ 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[1]. 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

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

[1] and a somewhat baffling dance: https://www.youtube.com/watch?v=lyZQPjUT5B4.

where `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, denote its value by x, and divide the sequence into three subsequences

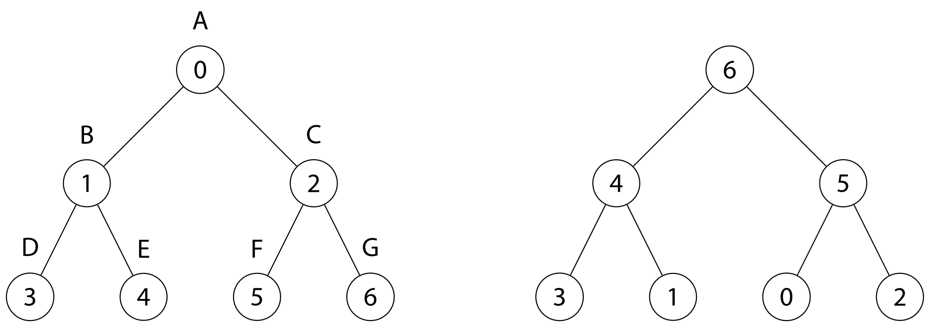
- S_1: all elements with values less than x.
- S_2: all elements with values equal to x.
- S_3: all elements with values greater than x.

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 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` or any textbook on data structures for full details). The easiest way to understand a tree is to look at a picture: figure 4.1(a) shows a simple binary tree. A binary tree consists of a set of nodes, each of which has at most two 'child' nodes. The 'top' node of a tree (as drawn here) is called its root. Note that any node within a binary tree forms the root of a (smaller) binary tree – e.g. the node labelled 2 is at the root of a tree containing three nodes, labelled 2, 5 and 6.

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 children, see Figure 4.1(b). An immediate consequence of this is that the value at the root node must be maximal among all the values stored in the heap. Arranging a sequence into a heap allows a particularly fast sort to be performed.



(a) a binary tree (b) a heap

Figure 4.1: (a) a binary tree with three levels: the root is the top node labelled A (storing value 0) with two children B and C (storing values 1 and 2 respectively). Each child has two children of its own (D and E, and F and G). Nodes with no children are called leaves. (b) the tree rearranged into a heap in which the value at a node is always greater than or equal to the values of its children.

Making a heap from a tree

A recursive algorithm for creating a heap from a tree is relatively easy to construct. We start by constructing a routine that makes a node N the root of a heap, by rearranging the values at N and its descendent nodes so that they satisfy the heap property. To do this, let's assume that both children of N are already at the root of a heap. Then if the value at node N is larger than (or equal to) the values at both its children, then N itself is the root of a heap and we are done. If not, we swap the value at N with the value at its largest child L. This will ensure that the value at N is at least as large as the value at any of its descendents, but this swap may mean that node L is no longer the root of a heap. So we apply our routine to the swapped child L (we can do this since both children of L are themselves at the root of a heap). This may result in another swap, and we would then apply the routine recursively to the swapped child of L, and so on down.

Our routine can be used to turn a whole binary tree into a heap by applying it to each node in turn, starting at the the leaves of the tree and ending at the root. We'll illustrate the algorithm by representing the values stored in the tree in figure 4.1(a) in a linear array, reading across each row of the tree in turn. The initial configuration is 0123456 and the final configuration (heap) is 6453102. We work backwards through the tree, ignoring nodes G, F, E and D since they have no children (and so are already the root of a heap of size 1).

- **Node C**: greatest child value is 6, exchange 2 and 6 to give 0163452.
- **Node B**: greatest child value is 4, exchange 1 and 4 to give 0463152.
- **Node A**: greatest child value is 6, exchange 0 and 6 to give 6403152.
- **Apply algorithm recursively to exchanged child**
    - **Node C:** greatest-child-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. Note that by working backwards through the tree from node G to node A, we guarantee that each time we apply our routine to a node, both children of that node will already be at the root of a heap.

Sorting the heap

Once the whole binary tree is arranged as a heap, we can easily find the element with greatest value, since it's at the root of the heap. The idea of heapsort is to remove this top value and replace it with the value from one of the leaf nodes. We can then apply our recursive heap-generation algorithm to the root node, which moves the element with next greatest value to the top of the heap. Repeating this, we build a sorted list by removing the top of the heap each time.

4.2 Coding, Examples and Exercises

Initially, we'll write our algorithms to sort the data in a `MVector` class.

4.2.1 Creating our own vector class from the standard library

Use the following header file to define a new class `MVector` (the file is available as `mvector.h` on the course website).

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>

// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}
    // access element (lvalue)
    double &operator[](int index) { return v[index]; }

    // access element (rvalue)
    double operator[](int index) const { return v[index]; }

    int size() const { return v.size(); } // number of elements

private:
    std::vector<double> v;
};

#endif
```

The class `MVector` represents a vector of `double` variables, similarly to a `std::vector<double>`.

Tasks:

1. Write a very simple main program that creates an `MVector` of size 10, store 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 be extremely useful when developing code, but will make your code run more slowly, so you should remove it in final "production runs" where speed is important. The `assert` macro in the `<cassert>` library might be useful here (see C++ Primer §6.14).
3. Add an additional member function to the `MVector` class

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

   that exchanges 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

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

to the `MVector` class that assign a pseudo-random (`double`) value, between the limits xmin and xmax, to each entry in the vector.

Solution:

Pseudo-random numbers (potentially of rather low quality) are generated by the `std::rand()` function in the `<cstdlib>` standard library header. `std::rand()` returns a random integer between 0 and the value of `RAND_MAX` (also defined in the `<cstdlib>` header. We can therefore implement the `initialise_random` method by rescaling this range of pseudo-random numbers to the desired range [`xmin`, `xmax`]:

```
void MVector::initialise_random(double xmin, double xmax)
{
    size_t s = v.size();
    for (size_t i=0; i<s; i++)
        v[i] = xmin + (xmax-xmin)*rand()/static_cast<double>(RAND_MAX);
}
```

Note that both `rand()` and `RAND_MAX` are integers, and so the expression `rand()/RAND_MAX` would be an integer division, resulting in an integer only. We therefore usually need to cast either `rand()` or `RAND_MAX` to a double, `rand()/static_cast<double>(RAND_MAX)` in order to specify floating-point division and a fractional answer (between 0 and 1). *An aside:* In fact, in our implementation of `initialise_random` the multiplication `(xmax-xmin)*rand()` occurs before the division (the associativity of both operators is left-to-right), and so the left operand of \ is promoted to a `double` (the type of `xmin-xmax`) anyway. The `static_cast <double>` is therefore not strictly required in our implementation of `initialise_random`, but using it has no run-time cost, and it emphasises that floating point division is required in this function. *End of aside.*

The `rand()` pseudo-random number generator always produces the same sequence of values each time a program is run. To avoid this we can seed the generator with the time that the program is run by putting the statement

```
std::srand(std::time(NULL));
```

at the start of `main`. (The `std::time` function requires the `<ctime>` header.)

Although pseudo-random numbers produced by the `rand()` function are not sufficiently random for many scientific tasks, they will be sufficient for our purposes. Methods of generating better pseudo-random numbers are available in the `<random>` header of the C++**11** standard library.

Tasks:

1. Add the source code for `initialise_random` to your program and make sure that it compiles and runs.
2. 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,

   ```
   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

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

   should produce the output

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

(assuming that you have overloaded the << operator).

2. Write a program that computes the average sort time for a randomly-initialised vector of length n. See the course website for timing code.

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 to implementing the quicksort algorithm: 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. Create two new functions,

   ```
   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 at `h[0]` and the children of the vertex `h[i]` are located at `h[2*i+1]` and `h[2*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 children).
2. Write two new functions:

   ```
   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 almost always 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 object-oriented 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 an abstract base class `SortableContainer` that contains all interfaces required for sorting data stored within the class.
7. Modify your sorting algorithms so that they work with `SortableContainer`s rather than `MVector`s, 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 `SortableContainer`s. One object that will be useful later is an array of coordinates. Firstly, we define a two-dimensional coordinate structure

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

(recall that a `struct` is a class with members that are `public` by default).

Tasks:

1. Create a class `CoordinateArray` that inherits from `SortableContainer` and stores a `std::vector` of `IntegerCoordinate` objects,

   ```
       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

   ```
       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

   $$(X,Y) < (A,B) \quad \text{if} \quad X < A \quad \text{or} \quad [X = A \text{ and } Y < B].$$
3. Test your new class by sorting some `CoordinateArray`s 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 `IntegerCoordinate`s (for various n), 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 `SortableContainer`s 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 resources (see for example the routines in the `<algorithm>` header, in C++ Primer Appendix A.2, or at `http://en.cppreference.com/w/cpp/algorithm`)
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 alive 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 alive, otherwise it remains dead.
- If a live cell has 2 or 3 live neighbours it remains alive, otherwise it dies.

These rules can be summarized by the notation B3/S23, indicating that a dead cell becomes alive 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[2] storage method it 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 will 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

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

    // Constructor
    Life();

    // Advance time one unit
    void tick();
};
```

   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. Feel free to investigate other methods to evolve the game of life[3]. You may use the standard template library 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) $\{(5,5),(5,4),(5,6)\}$.

[2]The trade-off is worth considering. Many interesting patterns in the game of life occur on a very large grid of cells, but only a small fraction of those cells are live at any one time. A grid of dimension $10^6 \times 10^6$ requires 10^{12} bits ($\approx$ 116GB) of storage – a typical amount of RAM for a large server. However, it may be that only 10^8 of these cells are alive at any one time. By storing the coordinates of each live cell as two 32-bit integers, we can store these live cells in ≈ 0.75GB of RAM – a typical amount for a mobile phone. Our coordinate representation has the advantage that each coordinate (as a 32-bit integer) has a range of $\approx 4 \times 10^9$, much larger than the fixed $10^6 \times 10^6$ grid. However, once the density of live cells becomes greater than $1/64$, the fixed grid uses less memory than storing coordinates.

[3]More complex algorithms have remarkable properties: the Hashlife algorithm of Bill Gosper (`https://dx.doi.org/10.1016%2F0167-2789%2884%2990251-3`) can be used to calculate the state of a game of life after a very large number of time steps (10^{30} or more), *without needing to calculate the state at each previous timestep*.

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 your report as a connected piece of prose. Read the Guidelines section at the start of this document for other instructions on the format of the report. Some questions have marks in square brackets, indicating the 20 marks available for correct numerical results and code. The other 20 marks are awarded for the overall quality of validation, analysis and the write-up, as detailed in the grading criteria provided in the guidelines.

Your report must include:

- code listings for the three sorting algorithms: bubblesort, quicksort and heapsort; **[9]**
- graphs of the average sort time as a function of the number of objects being sorted for the three algorithms; **[4]**
- 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 5206 generations (time steps) of the acorn pattern described in Task 3 above. **[2]**
- answers to the following questions:
  - what happens when you start with the initial configurations in Task 2 above? **[1]**
  - what is the total number of live cells after 5206 generations of the acorn pattern?**[1]**