Scientific Computing Project 1: ODE solver

ID Number: 7635417

Contents

Introduction

It is an unfortunate truth in mathematics that almost all problems which can be posed cannot be given closed form solutions. However, many of these problems are of paramount importance both within mathematics and for other scientific disciplines such as physics or engineering. Thus, there is a need to be able to deal with approximate solutions to the problems we can be presented with.

Ordinary Differential Equations are one of the most important classes of such problems, and find endless uses in all of the sciences [4]. Finding and developing techniques to solve such problems is a key activity for Mathematicians.

In this report, we will focus on solving second order ODEs. We will look at two different classes of problems, Initial Value Problems (IVPs) and Boundary Value Problems (BVPs) [3] and develop C++ code to numerically solve these on an interval.

We will also examine how accurate these codes are, how they could potentially be improved and give some indications of their limitations.

1 | Methods for Solving Ordinary Differential Equations

An Ordinary Differential Equation is an equation for a function of one variable $f(x)$ relating combinations various derivatives together. That is, in general an ODE will have the form [4]:

$$G\left(x, f(x), f'(x), \cdots, f^{(n)}(x)\right) = 0 \tag{1.1}$$

where G is some function, a prime denotes differentiation with respect to x and $f^{(n)}(x)$ is the n-th derivative of f with respect to x. In this project we will concern ourselves primarily with second order equations. For the sake of clarity we will write these in Leibniz's notation and call the function f as $y(x)$

$$\frac{d^2y}{dx^2} + p_1(x,y)\frac{dy}{dx} + p_2(x,y)y = 0 \tag{1.2}$$

where p_1 and p_2 are arbitrary functions.

In general an n-th order equation will require n constants to fix the solution [4]. This is related to how information is lost when we differentiate functions, namely any constant terms are always mapped to zero.

Another important point to note is that an n-th order equation can always be rewritten as a system of n coupled first order equations. To see this, set $z_1 = y$ and $z_2 = y'$ and then note that $z_1' = z_2$ and $z_2' = y''$. Rearrange 1.2 to isolate the second derivative:

$$\frac{d^2y}{dx^2} = -p_1(x,y)\frac{dy}{dx} - p_2(x,y)y \tag{1.3}$$

and now substituting for z_1 and z_2

$$\begin{aligned} \frac{dz_1}{dx} &= z_2 \\ \frac{dz_2}{dx} &= -p_1(x,z_1)z_2 - p_2(x,z_1)z_1\,. \end{aligned} \tag{1.4}$$

This method works for any order ODE, however we will only need the result for second order equations.

Finally, we show how it is possible to write higher order ODEs in vector form for later ease of computation. If we consider a vector of the variables $z_1, z_2, \ldots, z_n$ to be $\vec{z} = (z_1, z_2, \ldots, z_n)^T$ then we can write the functions which the system of ODEs is equal to as

$$\vec{F} = (f_1(x, \vec{z}), f_2(x, \vec{z}), \ldots, f_n(x, \vec{z}))^T$$

and the ODE can be rewritten once again to:

$$\frac{d\vec{z}}{dx} = \vec{F}(x, \vec{z}). \tag{1.5}$$

This is the form we will be using in the code.

1.1 Initial Value Problems

Initial value problems are ODEs where the problem is subject to initial conditions at the start of the range of values we are interested in solving for. For example, a second order ODE on an interval $a \leq x \leq b$. We give the initial data as $y(x = a) = \alpha$ and $y'(x = a) = \beta$. Since the problem is only constrained at one end, we simply have to step though the interval to find values for $y(x)$ and $y'(x)$. We will use three methods to do this, Euler's method, the Midpoint method and a 4^{th} order Runge-Kutta method.

1.1.1 Euler's Method

The simplest method of solving an IVP is Euler's method. We start by breaking the interval into n equal steps to iterate over. That is x ranges from a to b as

$$x_i = a + \sum_{i=0}^{n} ih \tag{1.6}$$

where $h = (b - a)/n$. h here is called the step size, and will be common to all methods we use. If we now consider the Taylor expansion of $y(x)$ in the interval in question (assuming that $y(x)$ has a sufficient number of derivatives) then for each x_i in the interval we can write the Taylor series of the next x step, x_{i+1} as [2]

$$y(x_{i+1}) = y(x_i) + (x_{i+1} - x_i)y'(x_i) + O((x_{i+1} - x_i)^2). \tag{1.7}$$

For sufficiently small step size, we can ignore all terms of order 2 or above. If we now note that $x_{i+1} - x_i = h$ then we can write Euler's method as

$$\begin{aligned} y_0 &= \alpha \\ y_{i+1} &= y_i + hf(x_i, y_i), \quad \text{for } i = 0, 1, \ldots, n-1 \end{aligned} \tag{1.8}$$

We can now write this as an algorithm for later use with programming:

Algorithm 1 Euler's Method

```
1: procedure EULER(a, b, n, α)   ▷ Solve IVP in [a, b] with n steps and initial condition α
2:     h ← (b − a)/n
3:     x ← a                                                    ▷ x starts at a
4:     y ← α                                                    ▷ y(x = a) = α
5:     for i = 1, 2, ..., n do
6:         y ← y + h f(x, y)                                    ▷ Compute the next approximation to y
7:         x ← a + ih                                           ▷ Step x along on the mesh
8:     end for
9:     return y(x = b)                                          ▷ Return the end value
10: end procedure
```

1.1.2 Midpoint Method

The midpoint method is a slight improvement on Euler's method. While we will not derive the method here, the reasoning is similar to that used to get the Euler method, only with a more careful analysis of the Taylor series is used.

Once again, we use the mesh as defined in 1.6. We give the method as [2]

$$\begin{aligned} y_0 &= \alpha \\ y_{i+1} &= y_i + hf\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} f(x_i, y_i)\right), \quad \text{for } i = 0, 1, \ldots, n-1 \end{aligned} \tag{1.9}$$

or as an algorithm:

Algorithm 2 Midpoint Method

```
1: procedure MIDPOINT(a, b, n, α)▷ Solve IVP in [a, b] with n steps and initial condition α
2:     h ← (b − a)/n
3:     x ← a                                                    ▷ x starts at a
4:     y ← α                                                    ▷ y(x = a) = α
5:     for i = 1, 2, ..., n do
6:         y ← y + h f(x + h/2, y + (h/2) f(x, y))              ▷ Compute the next approximation to y
7:         x ← a + ih                                           ▷ Step x along on the mesh
8:     end for
9:     return y(x = b)                                          ▷ Return the end value
10: end procedure
```

As a general rule, the midpoint method will be more accurate than the Euler method, at the expense of requiring a greater amount of computational resources to compute.

1.1.3 Runge-Kutta Method

Runge-Kutta methods are of a different class to the midpoint and Euler methods, with a different derivation. We will not concern ourselves with this here, but such can be found in [2, 3]. We will simply state the Runge-Kutta method of order 4.

$$\begin{aligned} &y_0 = \alpha, && k_1 = hf(x_i, y_i), \\ &k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{1}{2}k_1\right), && k_3 = hf\left(x_i + \frac{h}{2}, y_i + \frac{1}{2}k_2\right), \\ &k_4 = hf(x_i + h, y_i + k_3), && y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{1.10}$$

with k_1, k_2, k_3, k_4 being intermediate quantities used to write the final y. The Runge-Kutta method is a great deal more exact than the Euler method for each step (having a truncation error of order h^4 [2]) however requires a much greater amount of computational resources to compute. Once again, we write it as an algorithm for later use.

Algorithm 3 Runge-Kutta Method

procedure RUNGE(a, b, n, α) ▷ Solve IVP in $[a, b]$ with n steps and initial condition α
  $h \leftarrow (b - a)/n$
  $x \leftarrow a$ ▷ x starts at a
  $y \leftarrow \alpha$ ▷ $y(x = a) = \alpha$
  for $i = 1, 2, \ldots, n$ **do**
    $k_1 \leftarrow hf(x, y)$
    $k_2 \leftarrow hf\left(x + \frac{h}{2}, y + \frac{1}{2}k_1\right)$
    $k_3 \leftarrow hf\left(x + \frac{h}{2}, y + \frac{1}{2}k_2\right)$
    $k_4 \leftarrow hf(x + h, y + k_3)$
    $y \leftarrow y + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ ▷ Compute the next approximation to y
    $x \leftarrow a + ih$ ▷ Step x along on the mesh
  end for
  return $y(x = b)$ ▷ Return the end value
end procedure

1.2 Boundary Value Problems

We now have three methods for solving IVP problems. Thus, we now can turn our attention to another class of ODEs, boundary value problems. In a boundary value problem, we have an ODE for which the initial data is specified at either end of the the range. That is, in an interval $a \leq x \leq b$ we know that $y(x = a) = \alpha$ and $y(x = b) = \beta$. Problems of this type must be of order 2 or above, as otherwise it does not make sense for the initial data to be specified in more than one place.

This leads us to ask how we can possible solve such equations in terms of the IVP solving methods we have previously defined. In order to do this we require the Newton shooting method.

1.2.1 Newton Shooting

The Newton shooting method of solving a BVP change the problem from a BVP to an IVP as we require. The way this works is that we rewrite the second order equation in terms of a couple set of first order equations, as shown in 1.4 and 1.5. We need to assign a value to the second term of the system of equations. Start by writing the initial conditions in terms of a guess to what the correct version should be. If we call the function $\vec{Y}$ in vector form then the initial conditions will be the vector at the start of the interval. That is

$$\vec{Y}(x=a)=\begin{pmatrix}Y_1(a)\\Y_2(a)\end{pmatrix}=\begin{pmatrix}\alpha\\g\end{pmatrix}. \tag{1.11}$$

If we now use one of the IVP solvers we have detailed above, we can solve the IVP defined by the BVP transformed into coupled ODEs and with initial data given as in 1.11. This will give us the values at the end of interval:

$$\vec{Y}(x=b)=\begin{pmatrix}Y_1(b)\\Y_2(b)\end{pmatrix}=\begin{pmatrix}\beta_1\\\beta_2\end{pmatrix} \tag{1.12}$$

We can now compare our value for for β_1 to the other boundary value which we were given in the problem. If we define a function $\phi(g)$ as

$$\phi(g)=\underbrace{Y_1(x=b;g)}_{\text{IVP version of } Y_1 \text{ for guess } g.}-\underbrace{Y_1(x=b)}_{\text{Known value at boundary}}=\beta_1-Y_1(x=b) \tag{1.13}$$

then the original BVP will be solved when $\phi(g)=0$ thus transforming the problem into finding a g such that this is the case.

We will use Newton's iteration to find such a root. Under this scheme we will choose the next guess to be

$$g_{n+1}=g_n-\frac{\phi(g)}{\phi'(g)} \tag{1.14}$$

where we can obtain $\phi'(x)$ from differentiating Y_1 with respect to g and evaluating this at the end of the interval:

$$\frac{d\phi}{dg}=\left.\frac{dY_1}{dg}\right|_{x=b} \tag{1.15}$$

Since Newton's iteration has quadratic convergence we should very quickly find the correct value of g, allowing us to solve the BVP we are working with.

2 | Implementation

In order to keep the implementation of the problems fairly general, we will attempt to design a program which makes use of classes and inheritance. We will implement three of the problems given in the problem booklet.

Initial Value Problem

The first problem given in the source is an initial value problem with:

$$\frac{d^2y}{dx^2} = \frac{1}{8}\left(32 + 2x^3 - y\frac{dy}{dx}\right) \tag{2.1}$$

and with initial data $y(x=1) = 17$ and $y'(x=1) = 1$. We rewrite this as a system of first order equations as follows:

$$\begin{aligned} \frac{dz_1}{dx} &= z_2 \\ \frac{dz_2}{dx} &= \frac{1}{8}\left(32 + 2x^3 - z_1 z_2\right) \end{aligned} \tag{2.2}$$

This is implemented as `class ivp` in the code, which we will describe soon.

Boundary Value Problem 1

The second problem which is implemented is a boundary value problem of the form:

$$\frac{d^2y}{dx^2} + k\frac{dy}{dx} + xy = 0 \tag{2.3}$$

with data on the boundary given by $y(x=0) = 0$ and $y(x=1) = 1$ and k some number, taken to be of type `double`. We split this into a system of equations like so:

$$\begin{aligned} \frac{dz_1}{dx} &= z_2 \\ \frac{dz_2}{dx} &= -kz_2 - xz_1 \end{aligned} \tag{2.4}$$

This problem is implemented as `class bvp1`.

Boundary Value Problem 2

Finally, we implement another boundary value problem, defined by 2.1 and 2.2 but with conditions $y(x=1)=17$ and $y(x=3)=43/3$. This is implemented as `class bvp2`.

2.1 Background classes

The first important classes in the implementation are MVector and MFunction. These allow us to mimic the Algorithmic definitions of the IVP solvers given above more closely, by overloading various operators in order to make their behavior more like traditional mathematical definitions, instead of C++ versions. We start first with the MVector class.

2.1.1 MVector

MVector is a class to take standard C++ vectors and change their behavior to be more suitable for the tasks we have in mind for them. In MVector.hpp we first declare a vector of `double` type of indeterminate size for us to use. There are then three possible constructors for the MVector class. One to allow us to create an MVector without declaring the size, one to allow us to create an MVector of size n, and one to allow us to create an MVector of size n containing a `double` x in all of the n positions in the vector.

```
1 class MVector{
2   vector<double> v;
3 }
4 public:
5   explicit MVector(){}
6   explicit MVector(int n):v(n){}
7   explicit MVector(int n, double x):v(n,x){}
```

Listing 1: Start of MVector.hpp

The `explicit` keyword stops the compiler from implicitly converting types to facilitate operations. In this case, this is useful as we can be sure that we are not accidentally adding objects which it does not make mathematical sense to add, for example.

We now can over load operators to have them behave with MVectors as we expect them to. First in MVector.hpp we write the function definitions we require MVector to have. Then in MVector.cpp we define these operations.

For example in MVector.cpp we can define the assignment operator for MVectors, allowing us to copy the contents of one MVector into another.

```
  MVector& operator=(const MVector& X);
  double& operator[](int index);
  double operator[](int index) const;
  int size() const;
  vector<double> getVector();
  void push_back(double x);
};
```

Listing 2: Operator overloading in MVector.hpp

```
MVector &MVector::operator=(const MVector& X) {
  if(&X==this) return *this;
    v = X.v;
    return *this;
}
```

Listing 3: Overloading the assignment operator to allow assigning one MVector the contents of another

This works by simply returning the original MVector if it is set equal to itself. Otherwise, the MVector X is copied over the MVector we are assigning to. The `const` keyword here means that while we are passing a reference to the assigning MVector X so as to allow us to access the underlying vector class, we are not allowed to make changes to the MVector.

We also overload operators such as + and - to allow us to add and subtract MVectors and * and / to allow us to divide MVectors component wise by doubles. For example, when we overload the + operator, as can be seen in Listing 4, we walk component wise through the MVector on the left hand side of the sum and add the value at the corresponding index in the MVector on the right hand side. Note that also, this equality will work for MVectors of different sizes, however a warning is produced when addition or subtraction of two vectors of different sizes is occurring, to warn the user that they may receive unexpected results from this operation.

2.1.2 MFunction

The MFunction class is a class which we can only inherit from, and all inheriting classes must implement the () operator for. This is the function type we will use to implement our ODEs with when written into vector form. The most important part of this class definition is

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

The =0 operation at the end of the function definition means that this is a pure virtual function, and all inheriting classes must provide an implementation for it. Additionally, we cannot create an instance of this class, only provided other classes which inherit from it.

```
MVector operator+(const MVector& lhs, const MVector& rhs) {
  // Overload + operator to add MVectors
  if (lhs.size() == rhs.size()){ // Give a warning if the vectors are of difference size,
    MVector temp(lhs); // to state that the operation might not give the expected result.
    for(int i=0; i<temp.size(); i++) {
      temp[i] += rhs[i];
    }
    return temp;
  }
  else {
    cout << "Warning: The vectors are of different sizes! This could be very bad." << endl;
    MVector temp(lhs);
    for(int i=0; i<temp.size(); i++) {
      temp[i] += rhs[i];
    }
    return temp;
  }
}
```

Listing 4: Overloading the + operator for MVectors

2.1.3 IVP solvers

The IVP solvers are implemented as their own class ivp_solvers, with the three solvers being public functions within each class. The constructor for this class takes as an input an `int step` for the step size, and two `double`s for the interval we are solving in. The three IVP solving methods given in the first chapter are then implemented as functions which the class provides, taking input of an MVector y containing at first the initial data and after running the class the solution, an MFunction f implementing the ODE we are aiming to solve and an `int` j to act as control for optionally outputting values to file. As an example, for a midpoint solver we will have the following in solver.hpp:

```
MVector midpoint(MVector &y, MFunction &f, int j);
```

and the implementation, as per Algorithm 2, is given in Listing 5. Here, if the `int j` is set to one then we write values of x, y and y' into a file as we solve the IVP on the mesh.

2.1.4 ODE container

The ODEs are contained in a class which inherits from MFunction. There are, as mentioned before `class ivp`, `class bvp1` and `class bvp2`. For the BVP problems, the first two components of the vector contain the ODE we are aiming to solve. The second two contain the function differentiated with respect to the guess g. As an example, for the second boundary value problem

```
MVector ivp_solvers::midpoint(MVector &y, MFunction &f, int j){
  int i;
  double x, h;
  h = (b-a)/steps;
  x = a;
  y = y;
  if (j == 1){ //printing
    ofstream output("output_midpoint.csv");
    output << x << "," << y[0] << "," << y[1] << endl;
    for(i = 1; i < steps+1; i++){
      x = a + i*h;
      y = y + h*f(x + 0.5*h, y + 0.5*h*f(x,y));
      output << x << "," << y[0] << "," << y[1] << endl;
    }
    output.close();
  }
  else { // not printing
    for(i = 1; i < steps+1; i++){
      x = a + i*h;
      y = y + h*f(x + 0.5*h, y + 0.5*h*f(x,y));
    }
  }
  // cout << y << endl; // optional print
  return y;
}
```

Listing 5: Implementation of the midpoint IVP solver

$$\frac{d^2y}{dx^2} = \frac{1}{8}\left(32 + 2x^3 - y\frac{dy}{dx}\right) = F(x, y, y') \tag{2.5}$$

Now we can differentiate y'' with respect to g as suggested in the problem booklet to get

$$\begin{aligned}\frac{dy''}{dg} &= \frac{\partial F}{\partial x}\frac{dx}{dg} + \frac{\partial F}{\partial y}\frac{dy}{dg} + \frac{\partial F}{\partial y'}\frac{dy'}{dg} \\ &= 0 - \frac{1}{8}\left(y'\frac{dy}{dg} + y\frac{dy'}{dg}\right).\end{aligned} \tag{2.6}$$

If we now let $z_1 = \frac{dy}{dg}$ and $z_2 = \frac{dy'}{dg}$ then we can write

$$\begin{aligned} \frac{dz_1}{dx} &= z_2 \\ \frac{dz_2}{dx} &= -\frac{1}{8}\left(y' z_1 + y z_2\right) \end{aligned} \tag{2.7}$$

with initial conditions

$$\begin{aligned} z_1(x=1) &= 17 \\ z_2(x=1) &= \frac{43}{3}. \end{aligned} \tag{2.8}$$

This now allows us to write the system given in Listing 6, which we use as the definition of the function for this problem.

```
25 MVector bvp2::operator()(const double& x, const MVector& y){
26   MVector temp(4);
27   temp[0] = y[1];
28   temp[1] = 4 + (1./4.)*x*x*x - (1./8.)*y[0]*y[1];
29   temp[2] = y[3];
30   temp[3] = (-1./8.)*(y[1]*y[2] + y[0]*y[3]); // This is given by dy''/dg
31   return temp;
32 }
```

Listing 6: The definition of the () for `class bvp2`

2.1.5 Newton Shooting

The shooting function is implemented as a function in main.cpp, defined as given in 1.14. This gives us our new guess for g before running one of the IVP solvers on the problem again.

2.1.6 BVP solving

With all of this in place, we can solve a BVP given the initial conditions. An example for `class bvp1` see Listing 7. Here we run the shooting method for up to 500 times, either until we reach a tolerance for the value of ϕ or we run out of steps. j is the control int for deciding if we write the output of the final set of IVP iteration to file.

2.1.7 Main

The main file presents a command line argument structure for selecting the problem you wish to solve, which method you wish to use, how many iteration steps you would like to use, and whether you would like to print the results of solving the problem to a file.

```
for (n = 0; n < 500; n++){
  y[0] = 0.; y[1] = guess; y[2] = 0.; y[3] = 1.;
  solv.euler(y, f, j);
  temp = shooting(guess, y, d);
  guess = temp[0];
  if (abs(temp[1])<tol){ // finish the loop when sufficiently close
    break;
}
```

Listing 7: Code for solving bvp1

This is achieved by reading the number of command line arguments into `argc` and the characters passed on the command line into an array of characters given by `argv[]`. This gives the start of main as

```
int main(int argc, char* argv[]){
// Initialising variables here
if (argc < 4){
  std::cout << "Usage: ode [-b1/-b2/-i] [method] [steps] -f" << std::endl;
  return 0; // the minimum number of arguments is 3
}
```

Listing 8: First part of `int main`

If the number of command line arguments which are passed to the program are lower than the number of required arguments, at least enough to define the problem, the solving method and the number of steps, then the program outputs a message giving details of its usage.

We then use the arguments that have been passed on the command line to select which problem we wish to solve. For example, if we wish to solve the first boundary value problem, using the Euler method, with 1000 steps in the IVP solving stage, we would call the command ode with the following arguments:

```
ode -b1 euler 1000
```

which will trigger the following conditions in the source.

```
if (argc > 3){
```

and

```
if (flag1.compare("-b1") == 0){
```

and

```
if (method.compare("euler") == 0){
```

Finally, the number of steps is read into an int using

```
steps = std::stoi(argv[3]);
```

This then initializes a new instance of the bvp we wish to solve as f, and a new instance of the ivp solver class to use to solve with.

```
bvp1 f;
ivp_solvers solv (steps, 0, 1);
```

After this, the code shown in Listing 7 is ran to solve the BVP. Also, as an added safety consideration, if after 500 iterations of the newton shooting method we have still not converged onto a root, the following displays a warning for the user that the answers are almost certainly not correct.

```
if(abs(temp[1])>1.){ // deal with being unable to find the root
  cout << "A root was not found during the shooting method. Something is probably wrong.";
}
```

Listing 9: A condition to deal with the shooting method being unable to find a root

This now allows us to solve the IVPs and BVPs given in the problem booklet.

2.2 Building

Building of the program is controlled using a GNU Make file. We build a series of object files, MVector.o, odes.o, solvers.o and main.o and then link them together to give the final binary ode. The makefile is given by (with some options for cleaning left out) the code in Listing 10. The -O2 flag sets tells the compiler to optimize the code in an attempt to get some increase in speed of execution, and -std=c++11 enables some C++11 features.

```
CXX=g++
CXXFLAGS+=-O2 -std=c++11

all: ode

MVector.o:
  $(CXX) $^ $(CXXFLAGS) MVector.cpp -c

solvers.o:
  $(CXX) $^ $(CXXFLAGS) solvers.cpp -c

odes.o:
  $(CXX) $^ $(CXXFLAGS) odes.cpp -c

main.o:
  $(CXX) $^ $(CXXFLAGS) main.cpp -c

ode: MVector.o solvers.o odes.o main.o
  $(CXX) $^ $(CXXFLAGS) -o $@
```

Listing 10: Makefile for the project

3 Analysis and Conclusions

Now that we have the code given above, we need to get some handle on if it produces correct solutions for the problems we have attempted to solve. Since solving the BVPs requires solving IVPs in the course of the solution, I will focus primarily on showing that the code can correctly solve BVPs and assume that the IVP solving works as required.

3.1 BVP 1

The first initial value problem has an analytic solution for $k = 1$, which can be found using computer algebra systems. While I will not show the solution here (it is rather ugly, involving Airy functions) I will show a plot of the function.

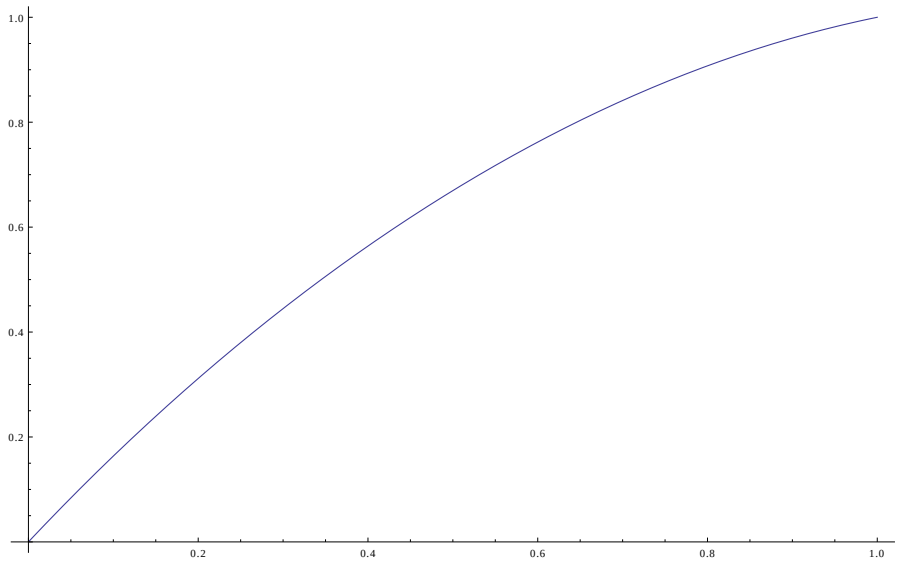

Figure 3.1: The analytic solution of bvp1

If we now solve this BVP using the code given, with options

```
ode -b1 runge-kutta 300 -f
```

we can produce another plot, with the points from the file plotted against the analytic solution of the BVP.

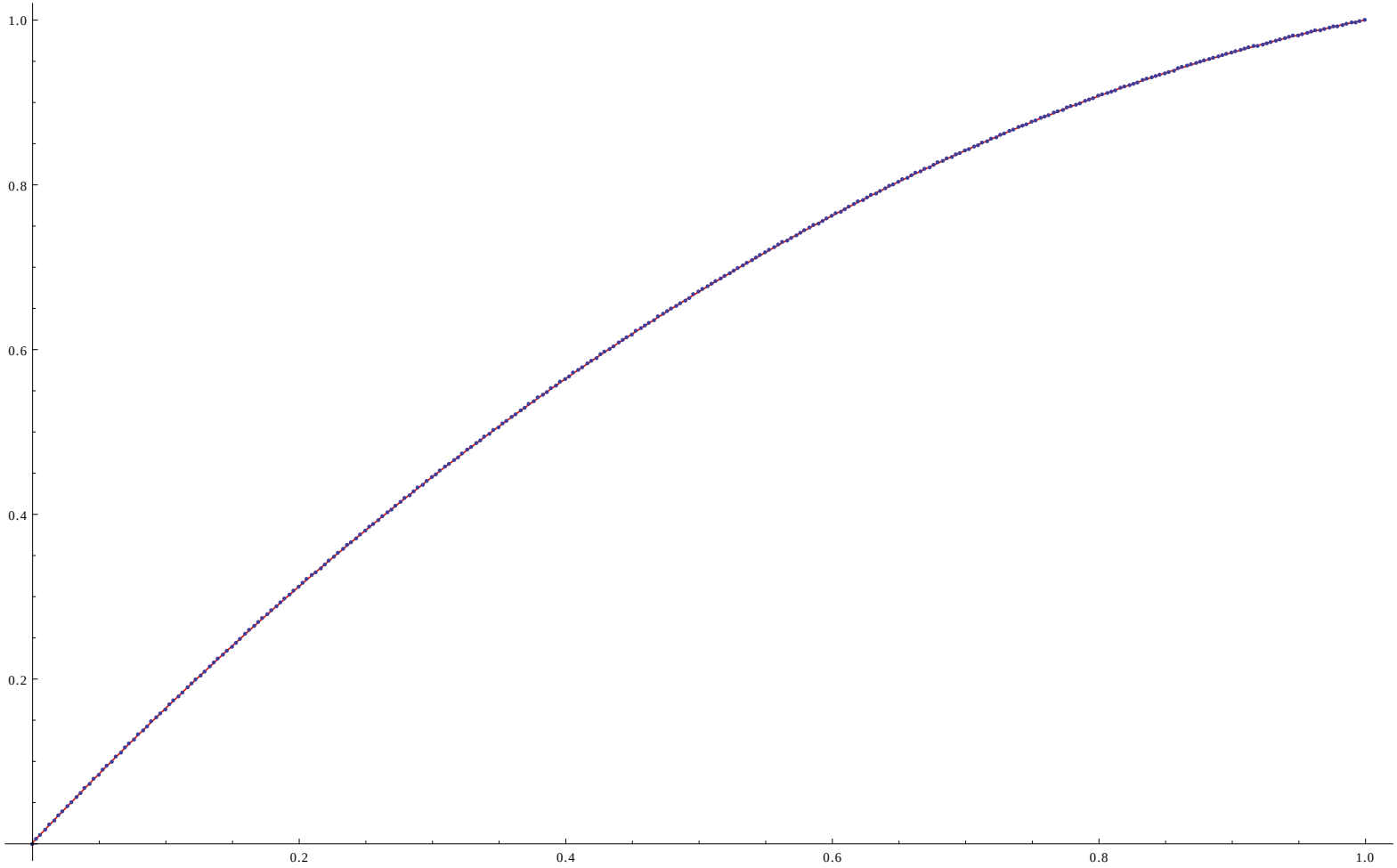

Figure 3.2: The analytic solution (in red) plotted against the points from the BVP solver in the code

As we can see, this is incredibly close to the BVP we are solving, giving good evidence for the numerical methods being accurate.

3.2 BVP 2

The second BVP has a nicer closed form solution, given in the problem book: $y(x) = x^2 + \frac{16}{x}$ with $y'(x = 1) = -14$. If we pick a different method this time, say the midpoint method, we can see how accurate the program is here. Running

```
ode -b2 midpoint 1000 -f
```

and then examining the output file shows that the method produces $y'(x = 1) = -14.0077$, not a perfect result but certainly close. The value given for $y(x = 3)$ is 14.3333, in perfect agreement with the closed form solution. Increasing the number of steps decreases the error on the guess term for the derivative, and at 100000 steps we find an error of only 0.0001 for the value of the first derivative, in very good agreement with the solution we expect.

3.3 Improvements

There are a number of potential improvements which could be made to the code to improve the clarity of the code and the re-usability.

- Clarity: While the code has been designed in an object oriented fashion, this could be taken further in order to decrease the amount of code which is duplicated. For example, the two bvp solvers share very similar code to initialize the problem and solve it in main.cpp, which is an unnecessary duplication of similar code. This could be changed by having a single bvp class with different overloading of the () operator for each individual bvp, however this would involve large changes to the structure of the code.
- Re-usability: Currently, in order to solve a different problem we need to change the code to include that problem and recompile before we can solve it. There are potentially ways to allow us to not have to do this to change the problem, for example introducing bindings to a interpreted language such as python, allowing us to change function definitions at run time.

Additionally, a number of standard libraries could make portions of the code a lot cleaner, for example the ODEInt library in LibBoost [1] implements many of the IVP solving techniques that have been implemented in this project, and a number of other techniques. Using this would be a good idea, as the validity of the source will have been verified to a greater extent than coding the solvers from the start.

3.4 Conclusion

The ode solver developed in this project is fairly accurate, and stand up well when testing against problems which we can be sure of the answer to. However, there are a number of improvements which could be made to the code, which would further increase the usefulness of the project. Overall however, the project works well and produces useful answers to problems which could be found in the sciences, engineering or mathematics.

A | Source code

A.1 cpp files

A.1.1 main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include "MFunction.hpp"
#include "MVector.hpp"
#include "solvers.hpp"
#include "odes.hpp"

MVector shooting(double g, MVector &y, MVector d){
  double phi, phidash, lower, upper;
  MVector out(2);
  lower = d[0]; upper = d[1];
  phi = y[0] - upper; // check the boundary conditions
  phidash = y[2];
  g = g - phi/phidash;
  out[0] = g; out[1] = phi;
  return out;
}

int main(int argc, char* argv[]){
  int j = 0;
  int steps, n;
  double tol, guess;
  string method, flag1, flag4;
  method = argv[2]; // allow us to check this is given correctly later
  tol = 0.00000000001; // set the tolerance
  if (argc < 4){
    std::cout << "Usage: ode [-b1/-b2/-i] [method] [steps] -f" << std::endl;
    return 0; // the minimum number of arguements is 3
  }
```

```
32    if (argc == 5){
33      flag4 = argv[4];
34      if (flag4.compare("-f") == 0){
35        j = 1;
36      }
37      else{
38        std::cout << "Assuming not writing into file" << std::endl;
39        j = 0;
40      }
41    }
42    if (method.compare("euler") != 0 && method.compare("midpoint") != 0 && method.compare("run
43      std::cout << "Please choose either the euler, midpoint or runge-kutta method" << endl;
44      return 0;
45    }
46    if (argc > 3){ // so long as we have more than 2 arguments we are good to go
47      std::string flag1 = argv[1];
48      steps = std::stoi(argv[3]); // c++11 way to change a string to an int
49      if (flag1.compare("-b1") == 0){
50        MVector d(2); // set the boundary conditions
51        MVector y(4); // to contain the problem
52        MVector temp(2); // keep the output from shooting
53        d[0] = 0; d[1] = 1;
54        guess = 5.;
55        bvp1 f;
56        ivp_solvers solv (steps, 0, 1);
57          for (n = 0; n < 500; n++){
58            y[0] = 0.; y[1] = guess; y[2] = 0.; y[3] = 1.;
59            solv.euler(y, f, j);
60            temp = shooting(guess, y, d);
61            guess = temp[0];
62            if (abs(temp[1])<tol){ // finish the loop when sufficently close
63              break;
64            }
65          }
66        if (method.compare("midpoint") == 0){
67          for (n = 0; n < 500; n++){
68            y[0] = 0.; y[1] = guess; y[2] = 0.; y[3] = 1.;
69            solv.midpoint(y, f, j);
70            temp = shooting(guess, y, d);
71            guess = temp[0];
72            if (abs(temp[1])<tol){
73              break;
74            }
```

```
75          }
76        }
77        if (method.compare("runge-kutta") == 0){
78          for (n = 0; n < 500; n++){
79            y[0] = 0.; y[1] = guess; y[2] = 0.; y[3] = 1.;
80            solv.runge(y, f, j);
81            temp = shooting(guess, y, d);
82            guess = temp[0];
83            if (abs(temp[1])<tol){
84              break;
85            }
86          }
87        }
88        if(abs(temp[1])>1.){ // deal with being unable to find the root
89          std::cout << "A root was not found during the shooting method. Something is probably
90        }
91        std::cout << "y(x): " << y[0] << std::endl; // output the important values
92        std::cout << "y'(x): " << y[1] << std::endl;
93        }
94      if (flag1.compare("-b2") == 0){
95        MVector d(2); // set the boundary conditions
96        MVector y(4); // to contain the problem
97        MVector temp(2); // keep the output from shooting
98        d[0] = 17.; d[1] = (43./3.);
99        guess = 2.;
100       bvp2 f;
101       ivp_solvers solv (steps, 1., 3.);
102       if (method.compare("euler") == 0){
103         for (n = 0; n < 500; n++){
104           y[0] = 17.; y[1] = guess; y[2] = 0.; y[3] = 1.;
105           solv.euler(y, f, j);
106           temp = shooting(guess, y, d);
107           guess = temp[0];
108           if (abs(temp[1])<tol){ // finish the loop when sufficently close
109             break;
110           }
111         }
112       }
113       if (method.compare("midpoint") == 0){
114         for (n = 0; n < 500; n++){
115           y[0] = 17.; y[1] = guess; y[2] = 0.; y[3] = 1.;
116           solv.midpoint(y, f, j);
117           temp = shooting(guess, y, d);
```

```
118           guess = temp[0];
119           if (abs(temp[1])<tol){
120             break;
121           }
122         }
123       }
124       if (method.compare("runge-kutta") == 0){
125         for (n = 0; n < 500; n++){
126           y[0] = 17.; y[1] = guess; y[2] = 0.; y[3] = 1.;
127           solv.runge(y, f, j);
128           temp = shooting(guess, y, d);
129           guess = temp[0];
130           if (abs(temp[1])<tol){
131             break;
132           }
133         }
134       }
135       if(abs(temp[1])>1.){ // deal with being unable to find the root
136         std::cout << "A root was not found during the shooting method. Something is probably
137       }
138       std::cout << "y(x): " << y[0] << std::endl; // output the important values
139       std::cout << "y'(x): " << y[1] << std::endl;
140       return 0;
141       }
142     if (flag1.compare("-i") == 0){
143       MVector d(2);
144       d[0] = 17; d[1] = 1;
145       MVector y(2);
146       y = d;
147       ivp f;
148       ivp_solvers solv (steps, 1., 3.);
149       if (method.compare("euler") == 0){
150         solv.euler(y, f, j);
151       }
152       if (method.compare("midpoint") == 0){
153         solv.midpoint(y, f, j);
154       }
155       if (method.compare("runge-kutta") == 0){
156         solv.runge(y, f, j);
157       }
158       std::cout << "y(x): " << y[0] << std::endl;
159       std::cout << "y'(x): " << y[1] << std::endl;
160       return 0;
```

```
161     }
162   }
163   else{
164     return 1; // return 1 after unexpected input, as per unix rules
165   } // at least, I think it's a unix thing
166   return 1;
167 }
```

A.1.2 odes.cpp

```
1  #include "MFunction.hpp"
2  #include "MVector.hpp"
3  #include "odes.hpp"
4
5
6  MVector ivp::operator()(const double& x, const MVector& y){
7    MVector temp(2);
8     temp[0] = y[1];
9     temp[1] = 4 + (1./4.)*x*x*x - (1./8.)*y[0]*y[1];
10   return temp;
11 }
12
13 MVector bvp1::operator()(const double& x, const MVector& y){
14   MVector temp(4);
15   temp[0] = y[1];
16   temp[1] = -k*y[1] - x*y[0];
17   temp[2] = y[3];
18   temp[3] = -k*y[3] - x*y[2];
19   return temp;
20 }
21 void bvp1::setk(double j){
22   k = j;
23 }
24
25 MVector bvp2::operator()(const double& x, const MVector& y){
26   MVector temp(4);
27   temp[0] = y[1];
28   temp[1] = 4 + (1./4.)*x*x*x - (1./8.)*y[0]*y[1];
29   temp[2] = y[3];
30   temp[3] = (-1./8.)*(y[1]*y[2] + y[0]*y[3]);
31   return temp;
32 }
```

A.1.3 solvers.cpp

```
#include "MFunction.hpp"
#include <fstream>
#include <iostream>
#include "MVector.hpp"
#include "solvers.hpp"


MVector ivp_solvers::euler(MVector &y, MFunction &f, int j){
  int i;
  double x, h;
  h = (b-a)/steps;
  x = a; // For printing, these are set to their initial values
  y = y; // and the loop started from 1 instead of 0 to get the file right
  if (j == 1){ // when printing
    ofstream output("output_euler.csv");
    output << x << "," << y[0] << "," << y[1] << endl;
    for(i = 1; i < steps+1; i++){
      x = a + i*h;
      y = y + h*f(x, y);
      output << x << "," << y[0] << "," << y[1] << endl;
    }
    output.close();
  }
  else { // not printing
    for(i = 1; i < steps+1; i++){
      x = a + i*h;
      y = y + h*f(x, y);
    }
  }
  // cout << y << endl; // optional print
  return y;
}
MVector ivp_solvers::midpoint(MVector &y, MFunction &f, int j){
  int i;
  double x, h;
  h = (b-a)/steps;
  x = a;
  y = y;
  if (j == 1){ //printing
    ofstream output("output_midpoint.csv");
    output << x << "," << y[0] << "," << y[1] << endl;
    for(i = 1; i < steps+1; i++){
```

```
43        x = a + i*h;
44        y = y + h*f(x + 0.5*h, y + 0.5*h*f(x,y));
45        output << x << "," << y[0] << "," << y[1] << endl;
46      }
47      output.close();
48    }
49    else { // not printing
50      for(i = 1; i < steps+1; i++){
51        x = a + i*h;
52        y = y + h*f(x + 0.5*h, y + 0.5*h*f(x,y));
53      }
54    }
55    // cout << y << endl; // optional print
56    return y;
57  }
58  MVector ivp_solvers::runge(MVector &y, MFunction &f, int j){
59    int i;
60    double x, h;
61    MVector k1, k2, k3, k4;
62    h = (b-a)/steps;
63    x = a;
64    y = y;
65    k1 = h*f(x,y);
66    k2 = h*f(x + 0.5*h, y + 0.5*k1);
67    k3 = h*f(x + 0.5*h, y + 0.5*k2);
68    k4 = h*f(x + h, y + k3);
69    if (j == 1){
70      ofstream output("output_runge.csv");
71      output << x << "," << y[0] << "," << endl; //<< y[1] << endl;
72      for(i = 1; i < steps+1; i++){
73        x = a + i*h;
74        k1 = h*f(x,y);
75        k2 = h*f(x + 0.5*h, y + 0.5*k1);
76        k3 = h*f(x + 0.5*h, y + 0.5*k2);
77        k4 = h*f(x + h, y + k3);
78        y = y + (1./6.)*(k1 + 2*k2 + 2*k3 + k4);
79        output << x << "," << y[0] << endl;
80      }
81      output.close();
82    }
83    else {
84      for(i = 1; i < steps+1; i++){
85        x = a + i*h;
```

```
      k1 = h*f(x,y);
      k2 = h*f(x + 0.5*h, y + 0.5*k1);
      k3 = h*f(x + 0.5*h, y + 0.5*k2);
      k4 = h*f(x + h, y + k3);
      y = y + (1./6.)*(k1 + 2*k2 + 2*k3 + k4);
    }
  }
  // cout << y << endl; // option print statement
  return y;
}
```

A.1.4 MVector.cpp

```
#include<vector>
#include<iostream>
#include"MVector.hpp"

using namespace std;

MVector &MVector::operator=(const MVector& X) {
  if(&X==this) return *this;
    v = X.v;
    return *this;
  }

// The following two operators allow to use commands such as a=v[i] or v[j]=b for
// MVector objects v
//double& operator[](int index){return v[index];}
// access data in vector (const)
//double operator[](int index) const {return v[index];}

double& MVector::operator[](int index) {
// Overload [] operator for writing
  return v[index];
}

double MVector::operator[](int index) const {
// Overload [] operator for reading
  return v[index];
}

// Methods (functions for this class
int MVector::size() const {
// Return size of vector
```

```
32    return v.size();
33  }
34
35  vector<double> MVector::getVector() {
36  // Return the underlying vector object
37    return v;
38  }
39
40  void MVector::push_back(double x) {
41  // Push an element into vector at the back
42  int s = this->size()+1;
43  vector<double> temp(s);
44    for(int i=0; i<s-1; i++) {
45      temp[i] = v[i];
46    }
47    temp[s-1] = x;
48    v = temp;
49  }
50
51  // Overload a series of operators so that they can be used on MVector objects
52
53  MVector operator*(const double& lhs, const MVector& rhs) {
54    // Overload * operator to multiply MVectors with scalars
55    MVector temp(rhs);
56    for(int i=0; i<temp.size(); i++) {
57      temp[i] *= lhs;
58    }
59    return temp;
60  }
61
62  MVector operator/(const MVector& lhs, const double& rhs) {
63    // Overload / operator to divide MVectors by scalars
64    MVector temp(lhs);
65    for(int i=0; i<temp.size(); i++) {
66      temp[i] = temp[i]/rhs;
67    }
68    return temp;
69  }
70
71  MVector operator+(const MVector& lhs, const MVector& rhs) {
72    // Overload + operator to add MVectors
73    if (lhs.size() == rhs.size()){ // Give a warning if the vectors are of difference size,
74      MVector temp(lhs); // to state that the operation might not give the expected result.
```

```
75      for(int i=0; i<temp.size(); i++) {
76        temp[i] += rhs[i];
77      }
78      return temp;
79    }
80    else {
81      cout << "Warning: The vectors are of different sizes! This could be very bad." << endl;
82      MVector temp(lhs);
83      for(int i=0; i<temp.size(); i++) {
84        temp[i] += rhs[i];
85      }
86      return temp;
87    }
88  }
89
90  MVector operator-(const MVector& lhs, const MVector& rhs) {
91    // Overload - operator to subtract MVectors
92    if (lhs.size() == rhs.size()){
93      MVector temp(lhs);
94      for(int i=0; i<temp.size(); i++) {
95        temp[i] -= rhs[i];
96      }
97      return temp;
98    }
99    else{
100     cout << "Warning: The vectors are of different sizes! This could be very bad." << endl;
101     MVector temp(lhs);
102     for(int i=0; i<temp.size(); i++) {
103       temp[i] -= rhs[i];
104     }
105     return temp;
106   }
107 }
108
109 double operator*(const MVector& lhs, const MVector& rhs) {
110   // Overload the * operator to compute innter product of MVectors
111   double ip = 0.;
112   for(int i=0; i<rhs.size(); i++) {
113     ip += (lhs[i]*rhs[i]);
114   }
115   return ip;
116 }
117
```

```
ostream& operator<<(ostream& os, const MVector& v) {
  // Overload the << operator to output MVectors to screen or file
  int n = v.size();
  cout << "(";
  for(int i=0; i<n; i++) {
    os << v[i];
    if(i<n-1) cout << ",";
  }
  cout << ")";
  return os;
}
```

A.2 Header files

A.2.1 odes.hpp

```
#ifndef ODES_HPP_
#define ODES_HPP_

class ivp : public MFunction { // an initial value problem class
  MVector operator()(const double& x, const MVector& y);
};

class bvp1 : public MFunction { // a boundary value problem class
  double k;
public:
  bvp1(){ // by default set k = 1 on initialising class
    k = 1.;
  }
  MVector operator()(const double& x, const MVector& y);
  void setk(double j);
};

class bvp2 : public MFunction { // a boundary value problem class
  MVector operator()(const double& x, const MVector& y);
};

#endif /* ODES_HPP_ */
```

A.2.2 solvers.hpp

```
#ifndef SOLVERS_HPP_
#define SOLVERS_HPP_

```

```
#include "MFunction.hpp"
#include <fstream>
#include <iostream>

class ivp_solvers {
  int steps,j; // variables all functions in the class need
  double a,b;
  MVector out;
  ofstream output;
public:
  ivp_solvers (int step, double m, double n){
    steps = step; // number of steps
    a = m; // lower end of interval
    b = n; // upper end of interval
  }
  MVector euler(MVector &y, MFunction &f, int j);
  MVector midpoint(MVector &y, MFunction &f, int j);
  MVector runge(MVector &y, MFunction &f, int j);
};

#endif /* SOLVERS_HPP_ */
```

A.2.3 MVector.hpp

```
#ifndef MVECTOR_HPP_
#define MVECTOR_HPP_
#include<vector>
#include<iostream>
using namespace std;

class MVector {
  vector<double> v;
public:
  // Constructors: allow various constructors with different arguments
  explicit MVector(){}
  explicit MVector(int n):v(n){}
  explicit MVector(int n, double x):v(n,x){}
  // Destructor
  ~MVector() {}  // Destructor does nothing here

  MVector& operator=(const MVector& X);
  double& operator[](int index);
  double operator[](int index) const;
  int size() const;
```

```
21    vector<double> getVector();
22    void push_back(double x);
23  };
24
25  // Function types for operator overloading
26  MVector operator*(const double& lhs, const MVector& rhs);
27  MVector operator/(const MVector& lhs, const double& rhs);
28  MVector operator+(const MVector& lhs, const MVector& rhs);
29  MVector operator-(const MVector& lhs, const MVector& rhs);
30  double operator*(const MVector& lhs, const MVector& rhs);
31  ostream& operator<<(ostream& os, const MVector& v);
32
33  #endif /* MVECTOR_HPP_ */
```

A.2.4 MFunction.hpp

```
1   /*
2    * MFunction.hpp
3    *
4    *  Created on: 13 Oct 2013
5    *      Author: martin
6    */
7
8   #ifndef MFUNCTION_HPP_
9   #define MFUNCTION_HPP_
10  #include"MVector.hpp"
11
12  class MFunction {
13  public:
14    virtual MVector operator()(const double& x, const MVector& y) = 0;
15    MFunction() {}
16    ~MFunction() {}
17  };
18
19  #endif /* MFUNCTION_HPP_ */
```

Bibliography

[1] K. Ahnert and M. Mulansky. "Odeint - Solving Ordinary Differential Equations in C++". In: *AIP Conference Proceedings* 1389.1 (2011), pp. 1586–1589. DOI: http://dx.doi.org/10.1063/1.3637934. URL: http://scitation.aip.org/content/aip/proceeding/aipcp/10.1063/1.3637934.

[2] Richard Burden and Douglas Faires. *Numerical analysis*. Cengage Learning, 1993.

[3] David F Griffiths and Desmond J Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer, 2010. ISBN: 978-0-85729-147-9. DOI: 10.1007/978-0-85729-148-6.

[4] Morris Tenenbaum and Harry Pollard. *Ordinary Differential Equations*. Dover Publications, 1985. ISBN: 978-0-486-64940-5.