

# Applications of Numerical Methods for ODEs

Edward Alexander Small (10786391)

November 2020

Full code, including unit tests, vector classes, function classes, and operator overloads, is available at <https://github.com/Teddyzander/ODESolver>. Contact [edward.small-2@postgrad.manchester.ac.uk](mailto:edward.small-2@postgrad.manchester.ac.uk) for access.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Formal Definitions . . . . .	2
1.1.1	Differential Equation . . . . .	2
1.1.2	Order of Differential Equation . . . . .	2
1.1.3	Ordinary Differential Equation . . . . .	2
<b>2</b>	<b>Basic ODE solvers for Boundary Conditions</b>	<b>2</b>
2.1	Euler Method . . . . .	3
2.1.1	Results for Eq. 3 . . . . .	4
2.2	4th Order Runge Kutta Method . . . . .	5
2.2.1	Results for Eq. 3 . . . . .	5
<b>3</b>	<b>Extending to Initial Value Problems</b>	<b>6</b>
3.1	The Newton Shooting Method . . . . .	7
3.2	Results . . . . .	9
<b>A</b>	<b>Function of <math>\frac{d\mathbf{Y}}{dx}</math></b>	<b>12</b>
<b>B</b>	<b>ODE Solvers</b>	<b>13</b>
<b>C</b>	<b>Falkner Skan Equation</b>	<b>17</b>
<b>D</b>	<b>Newton Shooting Method for FS</b>	<b>18</b>
<b>E</b>	<b>Code to Collect Data for Euler Method, RK4, and FS</b>	<b>19</b>

# 1 Introduction

Not all ODEs can be expressed in a *closed form*. When this is the case, it can prove useful to numerically integrate the system to find approximations for solutions to an ODE. These solutions are often sufficient for applications in many different fields, and there are many ways one could go about numerically integrating an ODE.

## 1.1 Formal Definitions

### 1.1.1 Differential Equation

A differential equation expresses the rate of change of one variable with respect to another. If a variable  $y$  can be expressed by some function  $f$  containing another variable  $x$

$$y = f(x) \tag{1}$$

$$\frac{dy}{dx} = f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \tag{2}$$

### 1.1.2 Order of Differential Equation

The order of a differential equation is the order of the highest derivative involved in the equation. eg

$$f''(x) + f'(x) - \beta f(x) = 1 \tag{3}$$

is second order, as  $f''(x) = \frac{d^2y}{dx^2}$  is the highest derivative[2].

### 1.1.3 Ordinary Differential Equation

An ordinary differential equation (ODE) contains on or more functions (and the derivatives of those functions) of one independent variable. Equation (3) is an example of this.

# 2 Basic ODE solvers for Boundary Conditions

Take the function

$$\frac{d\mathbf{Y}}{dx} = \mathbf{F}(x, \mathbf{Y}) = \begin{pmatrix} x \\ Y_2 \end{pmatrix} \tag{4}$$

With boundary conditions

$$\mathbf{Y}(x=0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mathbf{Y}(x=1) = \begin{pmatrix} 0.5 \\ e \end{pmatrix} \tag{5}$$

Since we know all of the initial conditions, and understand how the function behaves at all points (with the derivative), we can use numerical solutions to estimate the value of any point in this function. Code for this function can found in Appendix A as *functionF2*.

## 2.1 Euler Method

The *Euler Method* is directly taken from the formal definition of a derivative (equation 2). However, instead of taking the limit of  $h \rightarrow 0$ , we instead take  $h$  to be a very very small number, and use this to approximate the value of a nearby point.

$$\mathbf{F}(x, \mathbf{Y}) \approx \frac{\mathbf{Y}(x+h, y+h) - \mathbf{Y}(x, y)}{h} \quad (6)$$

Rearranging this equation can approximate the value of  $\mathbf{Y}(x+h, y+h)$ , which represents a nearby point.

$$\mathbf{Y}(x+h, y+h) \approx \mathbf{Y}(x, y) + h\mathbf{F}(x, \mathbf{Y}) \quad (7)$$

So, provided an initial condition is available and the derivative is known, a nearby point can be estimated. These steps can then be performed iteratively to find an approximation for the solution of any point. To do this we

- Clarify a lower bound  $x_a$  (the initial condition)
- Clarify the number of iterations  $n$  we are willing to complete
- Clarify an upper bound  $x_b$  (the point we wish to approximate)
- Calculate the step size  $h$  for the algorithm such that  $h = \frac{|a-b|}{n}$
- $x = x_a + ih$  where  $i = 0, 1, 2, \dots, n-1$  is the iteration currently being performed

Physically, what this algorithm does is take a tangent of the line of  $\mathbf{Y}$  at a point and then move along this tangent by an amount  $h$ . Because of this, it is easy to over/underestimate (depending on which way the function predominately curves) the value of  $\mathbf{Y}(x+h, y+h)$ .

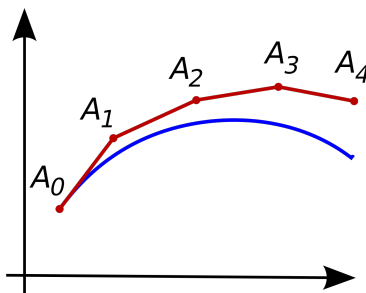


Figure 1: Example of how the Euler method "over shoots" when estimating a value

Accuracy is therefore determined by the size of  $h$ , with small sizes being more desirable, since the accuracy of each  $\mathbf{Y}(x_{i+1}, y_{i+1}) = \mathbf{Y}(x_i + h, y_i + h)$  is

at the mercy of how accurately  $\mathbf{Y}(x_i, y_i)$  has been calculated. The further apart the upper and lower bounds are, the more steps are needed to accurately find an answer. Code for the above algorithm is included in Appendix B as *EulerSolve*.

### 2.1.1 Results for Eq. 3

Since the result of  $\mathbf{Y}$  is available at  $x = 1$ , we can use these results to find how accurate the euler method is for  $x_a = 0$ ,  $x_b = 1$ .

$n$	estimate for $\mathbf{Y}_1$	estimate for $\mathbf{Y}_2$	errors for $\mathbf{Y}_1$	errors for $\mathbf{Y}_2$
16	0.46875	2.63793	0.03125	0.0803533
32	0.484375	2.67699	0.01562	0.0412917
64	0.492188	2.69734	0.0078125	0.0209369
128	0.496094	2.70774	0.0039062	0.0105428
256	0.498047	2.71299	0.00195313	0.0052902
512	0.499023	2.71563	0.000976563	0.00264983
1024	0.499512	2.71696	0.000488281	0.0013261

Table 1: Estimates and errors for  $Y$  using the Forward Euler Method for different number of steps  $n$

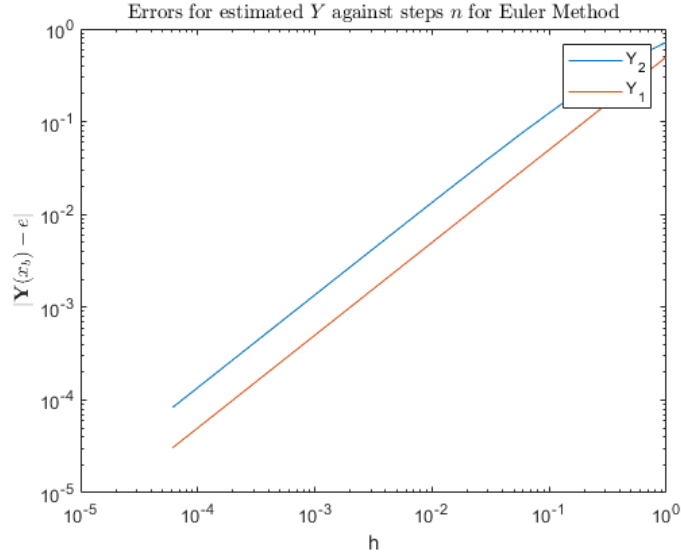


Figure 2: Graph to show how error size decreases as step size decreases

As shown from table 1 and figure 2, as  $h$  decreases (or  $n$  increases), error reduces. In fact, doubling the step size reduces the error by a half. The gradient of the line in figure 2 shows that the Euler Method is order 1. This can be shown

analytically by remembering that the local error on each step is

$$Kh^2 = O(h^2) \quad (8)$$

so for all steps

$$\frac{|x_a - x_b|}{h} kh^2 = O(h) \quad (9)$$

## 2.2 4th Order Runge Kutta Method

The 4th Order Runge Kutta Method (RK4) takes the Euler Method one step further. Fundamentally, it estimates the point  $\mathbf{Y}(x+h, y+h)$  by taking the slope at four points between  $\mathbf{Y}(x, y)$  and  $\mathbf{Y}(x+h, y+h)$ , weighting these values, and then taking an average of them, shown by

$$k_1 = f(x, \mathbf{Y}) \quad (10)$$

$$k_2 = f\left(x + \frac{h}{2}, \mathbf{Y} + k_1 \frac{h}{2}\right) \quad (11)$$

$$k_3 = f\left(x + \frac{h}{2}, \mathbf{Y} + k_2 \frac{h}{2}\right) \quad (12)$$

$$k_4 = f(x+h, \mathbf{Y} + k_3 h) \quad (13)$$

Which then gives

$$\mathbf{Y}(x+h, y+h) = \mathbf{Y}(x, y) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (14)$$

These steps can then be performed iteratively in exactly the same way as the Euler Method (clarifying bounds, picking step size, etc). Code for RK4 can be found in *Appendix B ODE Solvers* as *RungeKutta* function.

### 2.2.1 Results for Eq. 3

Again, since the result of  $\mathbf{Y}$  is available at  $x = 1$ , we can use these results to find how accurate RK4 is for  $x \in [0, 1]$ .

$n$	estimate for $\mathbf{Y}_1$	estimate for $\mathbf{Y}_2$	errors for $\mathbf{Y}_1$	errors for $\mathbf{Y}_2$
4	0.5	2.71821	0	7.18893e-05
8	0.5	2.71828	0	4.98404e-06
16	0.5	2.71828	0	3.28118e-07
32	0.5	2.71828	0	2.10479e-08
64	0.5	2.71828	0	1.33272e-09
128	0.5	2.71828	0	8.38409e-11

Table 2: Estimates and errors for  $Y$  using RK4 for different number of steps  $n$

RK4 converges so quickly, that it's a little harder to see the relationship between the error and the step size (though the clue is in the name!). However, the slope of the line on the loglog graph in figure 3 shows that it is order 4. This can also be proven analytically[1]. For each step RK4 takes the Taylor series up to a truncation error of

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + O(h^5) \quad (15)$$

So globally

$$\frac{|x_a - x_b|}{h}kh^5 = O(h^4) \quad (16)$$

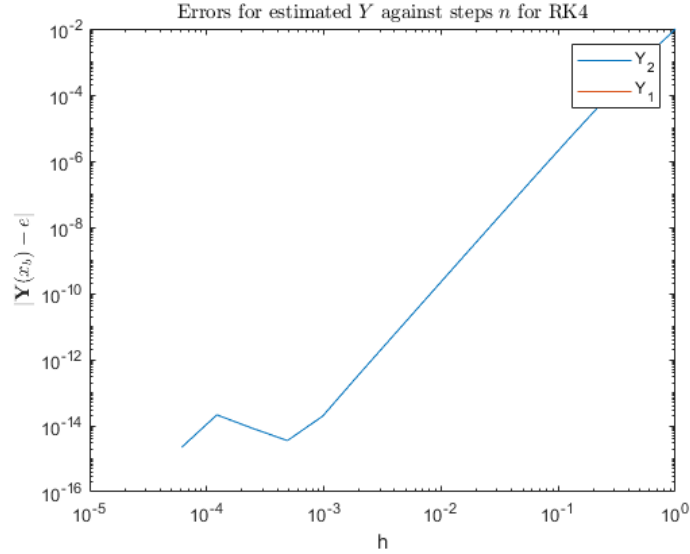


Figure 3: Graph to show how error size decreases as step size decreases ( $Y_1$  is along the 0 line)

The strange behaviour beyond  $10^{-14}$  it's caused by round off errors for doubles (since the maximum size of a double is  $10^{-16}$ ).

### 3 Extending to Initial Value Problems

Take the below 3rd order ODE, known as the *Falker-Skan* (FS) equation

$$f''' + ff'' + \beta(1 - f') = 0 \quad (17)$$

where

$$f(0) = f'(0) = 0 \text{ and } f' \rightarrow 1 \text{ as } \eta \rightarrow \infty \quad (18)$$

The equation describes a two-dimensional laminar flow past a wedge, where  $\beta$  is related to the angle of the wedge.

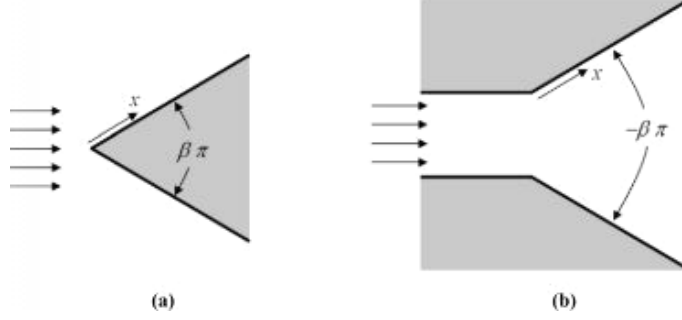


Figure 4: Physical representation of Eq (13)[3]

The ODE can be reduced to a first order vector ODE by the following. If

$$\mathbf{Y} = \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix}, \quad (19)$$

$$Y_1 = f, Y_2 = f', \text{ and } Y_3 = f'' \quad (20)$$

then the first order vector ODE  $\mathbf{Y}$  is as follows

$$\frac{d\mathbf{Y}}{d\eta} = \begin{pmatrix} Y_1' \\ Y_2' \\ Y_3' \end{pmatrix} = \begin{pmatrix} Y_2 \\ Y_3 \\ -\beta(1 - Y_2^2) - Y_1 Y_3 \end{pmatrix} \quad (21)$$

### 3.1 The Newton Shooting Method

(17) and (18) represent not just a boundary problem (finding  $f$  and  $f''$  as  $\eta \rightarrow \infty$ ), but also an initial value problem since we do not know the value of  $f''(0)$ . To solve this issue, we need to use the *Newton Shooting Method*[4].

Start by taking some guess  $g$  for  $f''(0)$ , giving

$$\hat{\mathbf{Y}}(0) = \begin{pmatrix} \hat{Y}_1(\eta=0) \\ \hat{Y}_2(\eta=0) \\ \hat{Y}_3(\eta=0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} \quad (22)$$

Using RK4 for some large value for  $\eta$ , we can then get a vector  $\hat{\mathbf{Y}}$  at  $\eta \approx \infty$

$$\hat{\mathbf{Y}}(\infty) = \begin{pmatrix} \hat{Y}_1(\eta \rightarrow \infty) \\ \hat{Y}_2(\eta \rightarrow \infty) \\ \hat{Y}_3(\eta \rightarrow \infty) \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} \quad (23)$$

Knowing that we expect  $Y_2 \rightarrow 1$  as  $\eta \rightarrow \infty$ , we can define a function  $\phi(g)$  that compares the actual value against the calculated value from the guess  $g$

$$\phi(g) = \hat{Y}_2(\eta \rightarrow \infty; g) - Y_2(\eta \rightarrow \infty) = \alpha_2 - 1 \quad (24)$$

From this point we can calculate a new guess  $g_{n+1}$  that will be closer to the true value of  $f''(0)$ , such

$$g_{n+1} = g_n - \frac{\phi(g)}{\phi'(g)} \quad (25)$$

For linear equations,  $\phi'(g)$  is independent of  $g$ , so this will converge in a single step. For non-linear equations, such as FS, the newton shooting method converges quadratically.

$\phi'(g)$  will also need to be calculated using finite difference methods, so we introduce 3 new variables, and then differentiate (17) with respect to  $g$ . If

$$Z_1 = \frac{df}{dg}, Z_2 = \frac{df'}{dg}, \text{ and } Z_3 = \frac{df''}{dg} \quad (26)$$

and

$$f''' = -\beta(1 - f') - ff'' := F(\eta, f, f', f'') \quad (27)$$

We can use partial differentiation and the chain-rule to find

$$\begin{aligned} \frac{dF}{dg} &= \frac{\partial F}{\partial \eta} \frac{d\eta}{dg} + \frac{\partial F}{\partial f} \frac{df}{dg} + \frac{\partial F}{\partial f'} \frac{df'}{dg} + \frac{\partial F}{\partial f''} \frac{df''}{dg} \\ &= 0 + Z_1 \frac{\partial F}{\partial f} + Z_2 \frac{\partial F}{\partial f'} + Z_3 \frac{\partial F}{\partial f''} \\ &= -Z_1 f'' + 2\beta Z_2 f' - Z_3 f \end{aligned} \quad (28)$$

So, as a vector ODE  $\mathbf{Z}$ , we get

$$\mathbf{Z} = \begin{pmatrix} Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} \quad (29)$$

$$\frac{d\mathbf{Z}}{d\eta} = \begin{pmatrix} Z_1' \\ Z_2' \\ Z_3' \end{pmatrix} = \begin{pmatrix} Z_2 \\ Z_3 \\ -Y_3 Z_1 + 2\beta Y_2 Z_2 - Y_1 Z_3 \end{pmatrix} \quad (30)$$

Where

- $Z_1(0) = 0$
- $Z_2(0) = 0$
- $Z_3(0) = 1$



- $Z_2(\eta = \infty) = \left. \frac{dY_2}{dg} \right|_{\eta=\infty} = \phi'(g)$

Code that describes the first order vector ODEs  $\mathbf{Y}$  and  $\mathbf{Z}$  are in appendix C *FalkerSkanEq*. Code that applies the newton shooting method to FS is in appendix D *FSBoundary*.

### 3.2 Results

Running the above algorithm for some values of  $\beta$  with a large enough  $\eta$  (in this case, 10) we can find estimates for  $f''(0)$  to a tolerance of  $|f'(\eta = \infty) - \hat{Y}_2(\eta = 10)| \leq 10^{-8}$ .  $\eta = 10$  was used as Schlichting[5] showed that

$$f'(10) \approx 1, \text{ for } -0.199 \leq \beta \leq 1.6 \quad (31)$$

$\beta$	estimated $f''(0)$	$h = 0.08$	$h = 0.04$	$h = 0.02$	$h = 0.01$
0	0.469600	0.46960001	0.46959999	0.46959999	0.46959999
$\frac{1}{3}$	0.802126	0.80212561	0.80212559	0.80212559	0.80212559
$\frac{10}{11}$	1.182817	1.18281694	1.18281722	1.18281723	1.18281724

Table 3: Estimates for  $f''(0)$  for different values of  $\beta$

As shown in section 2.2.1, RK4 is of order  $O(h^4)$ , so we should expect the ratio of errors to scale at a rate  $2^4$ . So, if  $f''(0)$  is the true value, and  $f''_{0.08}(0)$  and  $f''_{0.04}(0)$  are the estimated values at  $h = 0.08$  and  $h = 0.04$  respectively, then we expect the error ratio to be

$$\frac{E_{0.08}}{E_{0.04}} = \frac{f''(0) - f''_{0.08}(0)}{f''(0) - f''_{0.04}(0)} = 2^4 \quad (32)$$

We then solve for  $f''(0)$  and find the difference between that and our estimate.

$$f''(0) = \frac{16f''_{0.04}(0) - f''_{0.08}(0)}{15} \quad (33)$$

The error for each  $h$  can then be calculated. For each  $h$ , we expect the error  $\epsilon_h$  to be in the region of  $h^4$ , or

$$\epsilon_h = h^4, \text{ with } f''(0) = f''_h(0) \pm \epsilon_h \quad (34)$$

so, for example, at  $h = 0.01$  and  $\beta = 0$  the true value for  $f''(0)$  is

$$\begin{aligned} f''(0) &= f''_{0.01}(0) \pm \epsilon_{0.01} \\ &= 0.46959999 \pm 0.01^4 \end{aligned} \quad (35)$$

The uncertainty lies in the 8th decimal, so we can trust this value up to 7 decimal places and then round to 6.

The guess  $g$  for each value of  $\beta$  needs to be reasonably close to the true value to find convergence. Provided a change  $\delta$  to  $\beta$  is relatively small, we can use a previously calculated value for  $f''(0)$  for  $\beta$  as a starting guess for  $\beta + \delta$ .

Starting with  $\beta_0 = 0$ ,  $g_0 = 0.47$ , and  $\delta = 0.01$  we can produce figure 5 by using

$$\beta_{i+1} = \beta_i + \delta, \text{ and } g_{i+1} = (f''(0); \beta_i) \quad (36)$$

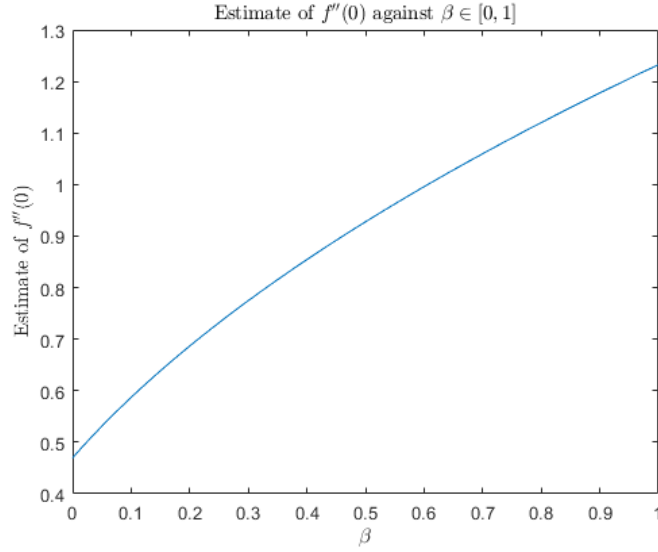


Figure 5: Estimates for  $f''(0)$  for values of  $\beta$  between 0 and 1

The code to apply functions/classes and collect the data used in this report can be find in Appendix E *main*.

## References

- [1] E. Hairer, S. P. Nørsett, G. Wanner *Solving Ordinary Differential Equations I*. Pages 143-154 Springer, 2008.
- [2] Morris Tenenbaum, Harry Pollard *Ordinary Differential Equation*. Pages 20-21 Harper and Row, 1963.
- [3] Deyi Shang *Review of Falkner-Skan Type Transformation for Laminar Forced Convection Boundary Layer*. Springer, 25 August 2010.
- [4] Stefan M.Filipova, Ivan D.Gospodinova, IstvánFaragó *Shooting-projection method for two-point boundary value problems*. Pages 10-15 Elsevier, October 2017.
- [5] Schlichting *Boundary-layer Theory*. Page 164 (Flow Past a Wedge) McGraw, Hill, 1955.  
Numerical Solution of the FalknerSkan Equation Using Third-Order and High-Order-Compact Finite Difference Schemes
- [6] Carlos Duque-Daza, Duncan Lockerby, Carlos Galeano *Numerical Solution of the FalknerSkan Equation Using Third-Order and High-Order-Compact Finite Difference Schemes* . Eq. 54

## A Function of $\frac{dY}{dx}$

```
1 #ifndef FUNCTIONF2_H
2 #define FUNCTIONF2_H
3
4 #include "mFunction.h"
5
6 // Specific function class is derived from generic function class
7 /*
8 Function is in the form of FunctionF1(x, y) = [x, y[1]]
9 */
10 class FunctionF2 : public MFunction
11 {
12 public:
13
14     // overload pure virtual function from MFunction
15     virtual MVector operator()(const double& x, const MVector& y)
16     {
17         // create vector to return calculated values
18         MVector temp(2);
19
20         // calculate values and return them
21         temp[0] = x;
22         temp[1] = y[1];
23
24         return temp;
25     }
26 };
27
28 #endif
```

## B ODE Solvers

```
1 #ifndef EULERSOLVE_H
2 #define EULERSOLVE_H
3
4 #include "mFunction.h"
5 #include <string>
6 #include <fstream>
7 #include <string>
8
9 /*
10 Data structure to contain outputs from methods for solving ODEs
11
12 error (int): a value for file reading/writing (0 if successful)
13 x (double): The value at x
14 y (MVector): value(s) at y
15
16 */
17 struct SolverOutput
18 {
19     int error;
20     double x;
21     MVector y;
22 };
23
24 /*
25 Function to estimate the value(s) of a function(s) at x = b
26
27 INPUTS:
28
29 steps (int): integer for number of steps wanted between a and b
30 a (double): value of x to start estimates at
31 b (double): value of x to end estimates at
32 y (MVector): vector to hold estimated values of y at each step
33 f (MFunction): function(s) to evaluate
34 save (bool): whether or not to save outputs to file
35
36 OUTPUTS:
37
38 result (SolverOutput): A struct holding estimated values for Y at x
39                       and an
40 error code for file writing
41 */
42 SolverOutput EulerSolve(int steps, double a, double b, MVector& y,
43                         MFunction& f, bool save = false)
44 {
45     std::ofstream myFile;
46     // Struct to hold return values and error value
47     SolverOutput result;
48     result.error = 0;
49     // declare the initial starting point for x
50     double x;
51     // calculate the size of each step
52     double h = abs(a - b) / steps;
53
54     // if user wants to save output, save to txt file
55     if (save == true)
```

```

55 {
56     myFile.open("eulerSolver" + std::to_string(steps) + ".txt");
57
58     // check opened file successfully. If not, return error code 1
59     if (!myFile.is_open())
60     {
61         result.error = 1;
62     }
63 }
64
65 // loop over
66 for (int i = 0; i < steps; i++)
67 {
68     x = a + i * h;
69
70     if (myFile.is_open())
71     {
72         myFile << "step: " << i << "\t" << x << "\t"
73             << y << std::endl;
74     }
75
76     y = y + h * f(x, y);
77 }
78
79 x = b;
80
81 // write final values to file
82 if (myFile.is_open())
83 {
84     myFile << "step: " << steps << "\t" << x << "\t"
85         << y << std::endl;
86 }
87
88
89 myFile.close();
90
91 result.x = x;
92 result.y = y;
93
94 return result;
95 }
96
97
98 /*
99 Function to estimate the value(s) of a function(s) at x = b
100
101 INPUTS:
102
103 steps (int): integer for number of steps wanted between a and b
104 a (double): value of x to start estimates at
105 b (double): value of x to end estimates at
106 y (MVector): vector to hold estimated values of y at each step
107 f (MFunction): function(s) to evaluate
108 save (bool): whether or not to save outputs to file
109
110 OUTPUTS:
111

```

```

112 result (SolverOutput): A struct holding estimated values for Y at
113 x and an error code (0 if no error)
114 */
115
116 SolverOutput RungeKutta(int steps, double a, double b, MVector& y,
117 MFunction& f, bool save = false, bool print = false)
118 {
119     std::ofstream myFile;
120     // Struct to hold return values and error value
121     SolverOutput result;
122     result.error = 0;
123     // declare the initial starting point for x
124     double x;
125     // calculate the size of each step
126     double h = abs(a - b) / steps;
127
128     MVector k1, k2, k3, k4;
129
130     // if user wants to save output, save to txt file
131     if (save == true)
132     {
133         myFile.open("rungeKutta" + std::to_string(steps) + ".txt");
134
135         // check opened file successfully. If not, return error code 1
136         if (!myFile.is_open())
137         {
138             result.error = 1;
139         }
140     }
141
142     // loop over
143     for (int i = 0; i < steps; i++)
144     {
145         x = a + i * h;
146
147         if (myFile.is_open())
148         {
149             myFile << "step: " << i << "\t" << x << "\t"
150                 << y << std::endl;
151         }
152
153         k1 = f(x, y);
154         k2 = f(x + h / 2, y + (h / 2) * k1);
155         k3 = f(x + h / 2, y + (h / 2) * k2);
156         k4 = f(x + h, y + h * k3);
157
158         y = y + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
159
160     }
161
162     x = b;
163
164     if (myFile.is_open())
165     {
166         myFile << "step: " << steps << "\t" << x << "\t"
167             << y << std::endl;
168     }

```

```
169
170
171     myFile.close();
172
173     if (print == true)
174     {
175         std::cout << "x: " << x << "\ny: "
176             << y << std::endl;
177     }
178
179     result.x = x;
180     result.y = y;
181
182     return result;
183 }
184
185 #endif
```



## C Falkner Skan Equation

```
1 #ifndef FALKNERSKANEQ_H
2 #define FALKNERSKANEQ_H
3
4 #include "mFunction.h"
5
6 // Specific function class is derived from generic function class
7 /*
8 Function is in the form of  $\text{FunctionF1}(x, y) = [y[1],$ 
9      $y[2],$ 
10      $-y[0] * y[2] - \text{beta} * (1 - y[1] * y[1]),$ 
11      $y[4],$ 
12      $y[5],$ 
13      $-y[0] * y[5] - y[2] * y[3] + 2 * \text{beta} * y[1] * y[4]]$ 
14 */
15 class FalknerSkanEq : public MFunction
16 {
17 public:
18
19     FalknerSkanEq() { beta = 0; }
20
21     // overload pure virtual function from MFunction
22     virtual MVector operator()(const double& x, const MVector& y)
23     {
24         // create vector to return calculated values
25         MVector temp(6);
26
27         // calculate values and return them
28         temp[0] = y[1];
29         temp[1] = y[2];
30         temp[2] = -y[0] * y[2] - beta * (1 - y[1] * y[1]);
31         temp[3] = y[4];
32         temp[4] = y[5];
33         temp[5] = -y[0] * y[5] - y[2] * y[3] + 2 * beta * y[1] * y[4];
34
35         return temp;
36     }
37
38     void SetBeta(double b) { beta = b; } // change beta
39
40 private:
41     // variable describing the angle of the wedge
42     double beta;
43 };
44
45 #endif
```

## D Newton Shooting Method for FS

```
1 #ifndef NEWTONSHOOTING_H
2 #define NEWTONSHOOTING_H
3
4 #include "falknerSkanEq.h"
5 #include "EulerSolve.h"
6 /*
7 INPUT:
8 beta (double): parameter for angle of the wedge fluid is flowing
9               past
10 guess (float): initial guess for f''(0)
11 bound (int): upper bound for eta
12 maxNewtonSteps (int): Maximum number to get solution
13 RungeKuttaSteps (int): Number of RK$ steps for each iteration of
14                       the
15                       shooting method
16 */
17 double FSBoundary(double guess = 0, double beta = 0, int bound = 5,
18                  int maxNewtonSteps = 100, int RungeKuttaSteps = 100)
19 {
20     FalknerSkanEq f;
21     MVector y(6);
22     f.SetBeta(beta);
23     double tol = 1e-8;
24     double phi;
25     double phidash;
26
27     for (int i = 0; i < maxNewtonSteps; i++)
28     {
29
30         y[0] = 0; y[1] = 0; y[2] = guess;
31         y[3] = 0; y[4] = 0; y[5] = 1;
32
33         SolverOutput result = RungeKutta(RungeKuttaSteps,
34                                         0, bound, y, f);
35
36         phi = result.y[1] - 1;
37         phidash = result.y[4];
38
39         if (abs(phi) < tol) break;
40
41         guess -= phi / phidash;
42     }
43
44     return guess;
45 }
46
47 #endif
```

## E Code to Collect Data for Euler Method, RK4, and FS

```
1 #include <iostream>
2 #include <cmath>
3 #include <fstream>
4 #include <string>
5 #include <iomanip>
6
7 #include "EulerSolve.h"
8 #include "falknerSkanEq.h"
9 #include "functionF2.h"
10 #include "functionF1.h"
11 #include "NewtonShooting.h"
12
13 int main()
14 {
15     /*
16     Find errors for euler and RK4 methods for different step sizes
17     */
18     FunctionF2 f;
19     SolverOutput data_euler, data_rungekutta;
20
21     std::ofstream eulerErrorFile, rungekuttaErrorFile;
22
23     eulerErrorFile.open("eulerErrors2.txt");
24     rungekuttaErrorFile.open("rungekuttaErrors2.txt");
25
26     double act_y0 = 0.5;
27     double act_y1 = exp(1);
28
29     for (int i = 1; i < 1025; i = i * 2)
30     {
31         MVector y1({ 0, 1 });
32         data_euler = EulerSolve(i, 0, 1, y1, f, true);
33         MVector y2({ 0, 1 });
34         data_rungekutta = RungeKutta(i, 0, 1, y2, f, true);
35
36         if (eulerErrorFile.is_open())
37         {
38             eulerErrorFile << i << "\t" << data_euler.y[0] <<
39                 "\t" << data_euler.y[1] << "\t" <<
40                 abs(data_euler.y[0] - act_y0) <<
41                 "\t" << abs(data_euler.y[1] - act_y1) <<
42                 std::endl;
43         }
44
45         if (rungekuttaErrorFile.is_open())
46         {
47             rungekuttaErrorFile << i << "\t" <<
48                 data_rungekutta.y[0] << "\t" <<
49                 data_rungekutta.y[1] << "\t" <<
50                 abs(data_rungekutta.y[0] - act_y0) <<
51                 "\t" << abs(data_rungekutta.y[1] - act_y1) <<
52                 std::endl;
53         }
54     }
```

```

54     }
55
56     eulerErrorFile.close();
57     rungekuttaErrorFile.close();
58
59     /*
60     Calculate f''(0) for some values of beta
61     */
62
63     std::cout << std::setprecision(16);
64     double guess = 0.47;
65     double beta = 0.0;
66
67     double fdash_est = FSBoundary(guess, beta);
68
69     std::cout << fdash_est << std::endl;
70
71     guess = 0.7;
72     beta = 1.0/3.0;
73
74     fdash_est = FSBoundary(guess, beta);
75
76     std::cout << fdash_est << std::endl;
77
78     guess = 1.2;
79     beta = 10.0 / 11.0;
80
81     fdash_est = FSBoundary(guess, beta);
82
83     std::cout << fdash_est << std::endl;
84
85     /*
86     Calculate f''(0) for values of beta between 0 and 1
87     */
88
89     guess = 0.47;
90     beta = 0.0;
91     std::ofstream myFile;
92     myFile.open("fdashdash0.txt");
93
94     while (beta < 1.01)
95     {
96         fdash_est = FSBoundary(guess, beta);
97
98         myFile << beta << "\t" << fdash_est << std::endl;
99
100        guess = fdash_est;
101        beta += 0.01;
102    }
103
104    myFile.close();
105
106    return 0;
107
108 }

```