MATH49111/69111 Scientific Computing
Project 1

Nikolaos Theodorakopoulos
ID: 9798612 *

November 20, 2015

*School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (nikolaos.theodorakopoulos@postgrad.manchester.ac.uk).

1 Introduction

We were assigned a task to solve the Falkner-Skan equation (a third order boundary value problem)by converting it from a boundary value problem to an initial value problem through the shooting method.In order to solve an initial value ODE problem we used the Euler integration, the Midpoint integration and the 4th order Runge-Kutta method.We resort to numerical methods that will allow us to approximate solutions to differential equations.

1.1 Prerequisites

1.1.1 The Shooting Method

In numerical analysis, the shooting method is a method for solving a boundary value problem by reducing it to the solution of an initial value problem.The shooting method uses the methods developed for solving initial value problems to solve boundary value problems. The idea is to write the boundary value problem in vector form and begin the solution at one end of the boundary value problem, and shoot to the other end with an initial value solver until the boundary condition at the other end converges to its correct value.The vector form of the boundary value problem is written in the same way as it was for the initial value problems, except all of the initial conditions are not known a-priori. Moreover 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.

1.1.2 The Shooting Method using Newton iteration

The idea of the Newton iteration is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line (which can be computed using the tools of calculus), and one computes the x-intercept of this tangent line (which is easily done with elementary algebra). This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated.We now combine the Newton iteration for root finding with an initial value ODE solver to calculate the solution to boundary problems.We start from some initial guess g_0 and iteratively refine this with successive guesses $g_1, g_2....$ For some guess g_n, the next guess g_{n+1} is given by

$$g_{n+1} = g_n - \frac{f(g_n)}{f'(g_{n+1})}$$

If $f(g)$ is nonlinear this iteration converges quadratically for a sufficiently good initial guess.If $f(g)$ is linear then $f'(g)$ is independent of g and

$$g_{n+1} = g_n - \frac{f(g_n)}{f'(g_{n+1})}$$

solves

$$f(g) = 0$$

exactly for any initial guess.

2 Vectors and Functions

2.1 Vectors and Operators

In order to begin, we need to create a new class MVector that represents a mathematical vector of real numbers and then overload the four arithmetic operators $+, -, /, *$ for our new MVector class.

2.1.1 Functions

In order to solve an initial value ODE problem we need to develop a function that takes one double and one MVector parameter and returns a MVector.Also we need to provide the function with the initial conditions, the initial value of independent variable x,the number of steps *struct* which is class where all members are public

3 ODE solver function

We will use the Euler Method, the Midpoint Method and the 4th order Runge-Kutta Method and try to solve the following simple problem.We need to have a first look about the accuracy of the approximations of each method before we try to solve a more complicated problem.

$$\frac{dY}{dx} = F_2(x, Y) \quad with \quad Y(x = 0) = (0, 1)$$

on the interval $x \in [0, 1]$. The exact solution at $x = 1$ is

$$Y(x = 1) = (0.5, e)$$

For each method we used a **step** of 0.001 and the output **precision** is 16 digits. So, for each method, we have the following results at $x = 1$:

Euler Method: (0.4995000000000002 , 2.716923932235896)

Midpoint Method: (0.4999999999999998 , 2.718281375751763)

4th order Runge-Kutta: (0.4999999999999998 , 2.718281828459025)

Figure 1: All Three approximation methods converging at 0.5 and e

At first glance the graph illustration seems the same for all three methods.But the figures bellow will help as take a closer look.

Figure 2: Yellow colour line for Euler's Method against the other two Methods with red colour approximation at e

Figure 3: Blue colour line for Runge-Kutta Method against the Midpoint method approximation at e

We clearly see that the **4th order Runge-Kutta** Method returns the best approximation to our problem.But why? We will try to explain this by solving the following problem.

Consider the following nonlinear ODE:

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

on the interval $x \in [1, 3]$ with initial conditions

$$y(x = 1) = 17 \quad and \quad y'(x = 1) = 1$$

We will solve this problem using the Euler Method, the Midpoint Method and the 4th order Runge-Kutta Method. We will make comments on all these three numerical methods and try to explain why the 4th order Runge-Kutta is the best approximation method.

For each of the following methods we used a **step** of 0.002 and the output **precision** is 16 digits. So, for each method, we created tables for the last 20 approximations.The output Y_i is a vector with elements: $Y_1(x_i)$ is the approximation of the solution of the above ODE at the point x_i and $Y_2(x_i)$ is the aproximation

of $y'(x_i)$ at the point x_i.

3.0.1 Euler Method

Eulers method is used to solve first-order initial-value problems and is this simplest (and least accurate) method for integrating an ODE.The derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.There are two main reasons why Euler's method is not generally used in scientific computing. Firstly, the truncation error per step associated with this method is far larger than those associated with other, more advanced, methods (for a given value of h). Secondly, Euler's method is too prone to numerical instabilities.In a few words, we are computing approximations based on previous approximations.

We'll use Euler's Method to approximate solutions to a couple of first order differential equations. The differential equations that we'll be using are linear first order differential equations that can be easily solved for an exact solution.Suppose we are applying Eulers method to a given initial-value problem over some interval $[x_0, x_{max}]$. In order to reduce the error we can adjust the step size, (or, equivalently, the number of steps, N , in going from x_0 to x_{max}). By shrinking (increasing N), at least a good thing is typically accomplished: The error in the underlying approximation $y(x_k + \delta x) = y(x_k) + \delta x f(x_k, y(x_k))$ is reduced.Euler's method can be thought of as a first-order Runge-Kutta method.

The algorithm for Euler Method is the following

$$x_i = a + ih$$

$$Y_{i+1} = Y_i + hF(x_i, Y_i)$$

for $i = 0, 1...steps - 1$.

Table 1: Euler Method

$Y_1(x_i)$	$Y_2(x_i)$	x_i
21.28344918831749	3.342271426538865	2.962
21.29013373117057	3.345481131082789	2.964
21.29682469343274	3.348694504585823	2.966
21.30352208244191	3.351911546979817	2.968
21.31022590553587	3.355132258180948	2.97
21.31693617005223	3.358356638089751	2.972
21.32365288332841	3.361584686591143	2.974
21.33037605270159	3.364816403554448	2.976
21.3371056855087	3.368051788833419	2.978
21.34384178908637	3.371290842266268	2.98
21.3505843707709	3.374533563675686	2.982
21.35733343789825	3.377779952868872	2.984
21.36408899780399	3.381030009637554	2.986
21.37085105782326	3.384283733758019	2.988
21.37761962529078	3.38754112499113	2.99
21.38439470754076	3.390802183082357	2.992
21.39117631190693	3.394066907761801	2.994
21.39796444572245	3.397335298744216	2.996
21.40475911631994	3.400607355729034	2.998
21.4115603310314	3.403883078400393	3

3.0.2 Midpoint Method

The Midpoint Method is very similar to the Euler's Method. The midpoint method is a one-step method for numerically solving the differential equation.The name of the method comes from the fact that in the formula bellow the function f giving the slope of the solution, is evaluated at $x = x_n + \frac{h}{2}$, which is the midpoint between x_n at which the value of $y(x)$ is known and x_{n+1} at which the value of $y(x)$ needs to be found.Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval.Midpoint method can be thought of as a second-order Runge-Kutta method.

The algorithm for the Midpoint Method is the following

$$x_i = a + ih$$

$$Y_{i+1} = Y_i + hF(x_i + \frac{h}{2}, Y_i + \frac{h}{2}F(x_i, Y_i))$$

for $i = 0, 1...steps - 1$.

Table 2: Midpoint Method

$Y_1(x_i)$	$Y_2(x_i)$	x_i
21.2852918026364	3.342354195103837	2.962
21.29197971875109	3.345563755347702	2.964
21.29867405765155	3.348776980855621	2.966
21.30537482666826	3.351993871561683	2.968
21.31208203313153	3.355214427384317	2.97
21.31879568437152	3.358438648226318	2.972
21.32551578771816	3.361666533974867	2.974
21.33224235050114	3.364898084501563	2.976
21.33897538004987	3.368133299662446	2.978
21.34571488369348	3.371372179298018	2.98
21.35246086876075	3.374614723233272	2.982
21.35921334258009	3.377860931277716	2.984
21.36597231247953	3.381110803225396	2.986
21.37273778578666	3.384364338854923	2.988
21.37950976982861	3.387621537929497	2.99
21.38628827193203	3.39088240019693	2.992
21.39307329942304	3.394146925389674	2.994
21.39986485962723	3.397415113224841	2.996
21.40666295986957	3.400686963404233	2.998
21.41346760747446	3.403962475614362	3

3.0.3 Runge-Kutta Method

Runge-Kutta 4th order method is a numerical technique used to solve ordinary differential equation.Also is generally considered to be an accurate, stable numerical integration method.This method requires four evaluations of the righthand side per step h.In each step the derivative is evaluated four times:once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value is calculated.

The algorithm for Runge-Kutta Method is the following:

$$x_i = a + ih$$

$$k_1 = F(x_i, Y_i)$$

$$k_2 = F(x_i + \frac{h}{2}, Y_i + \frac{h}{2}k_1)$$

$$k_3 = F(x_i + \frac{h}{2}, Y_i + \frac{h}{2}k_2)$$

$$k_4 = F(x_i + h, Y_i + hk_3)$$

for $i = 0, 1...steps - 1$.

Table 3: Runge-Kutta Method

$Y_1(x_i)$	$Y_2(x_i)$	x_i
21.28529209134954	3.342353762941858	2.962
21.29198000782373	3.345563322128238	2.964
21.29867434708156	3.348776546581681	2.966
21.30537511645352	3.351993436236275	2.968
21.31208232326991	3.355213991010444	2.97
21.3187959748609	3.358438210806982	2.972
21.32551607855641	3.361666095513068	2.974
21.33224264168613	3.364897645000299	2.976
21.33897567157949	3.368132859124711	2.978
21.34571517556561	3.371371737726805	2.98
21.35246116097327	3.374614280631572	2.982
21.35921363513087	3.377860487648516	2.984
21.36597260536646	3.381110358571684	2.986
21.37273807900761	3.384363893179681	2.988
21.37951006338147	3.387621091235707	2.99
21.38628856581468	3.390881952487572	2.992
21.39307359363337	3.394146476667724	2.994
21.39986515416311	3.397414663493274	2.996
21.4066632547289	3.400686512666022	2.998
21.41346790265512	3.403962023872476	3

3.0.4 Comparison of the three methods

The Runge-Kutta method yields better results than the Midpoint and Euler method, although for those step sizes are chosen accordingly smaller to have a comparable effort in computation. This will be superior to the midpoint method if at least twice as large a step is possible with for the same accuracy.Euler's Method has a local truncation error of $O(h^2)$ while the global truncation error is $O(h)$.Midpoint's Method has a local truncation error of $O(h^3)$ while the global truncation error is $O(h^2)$.

Figure 4: Euler Method with blue colour against the other two methods at the approximation of the solution $Y_1(x_i)$

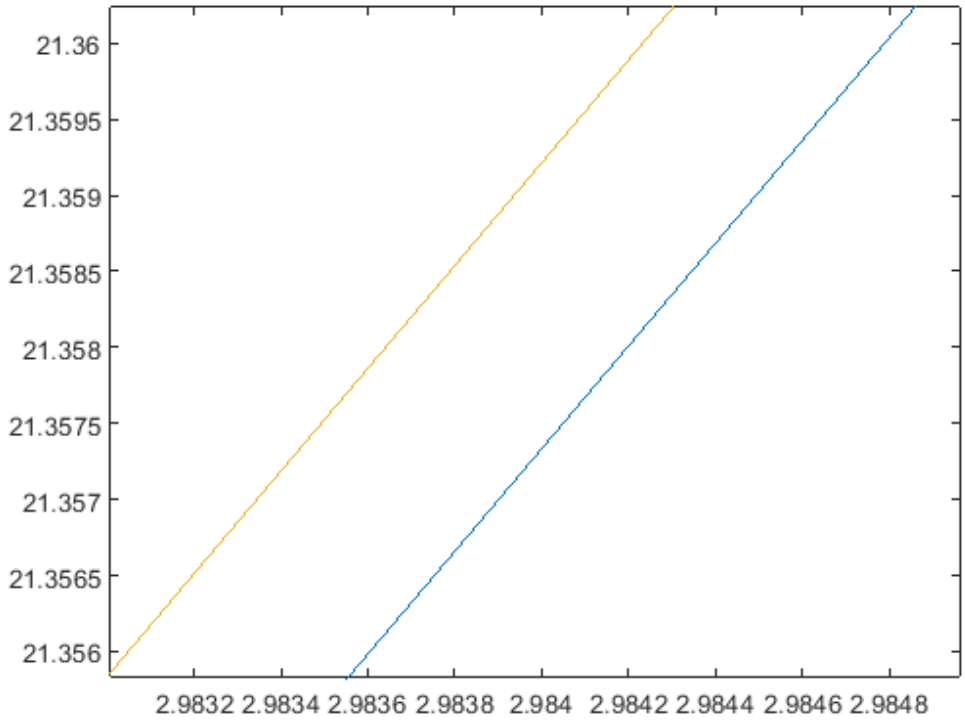


Figure 5: Runge-Kutta Method with blue colour against the Midpoint Method at the approximation of the solution $Y_1(x_i)$

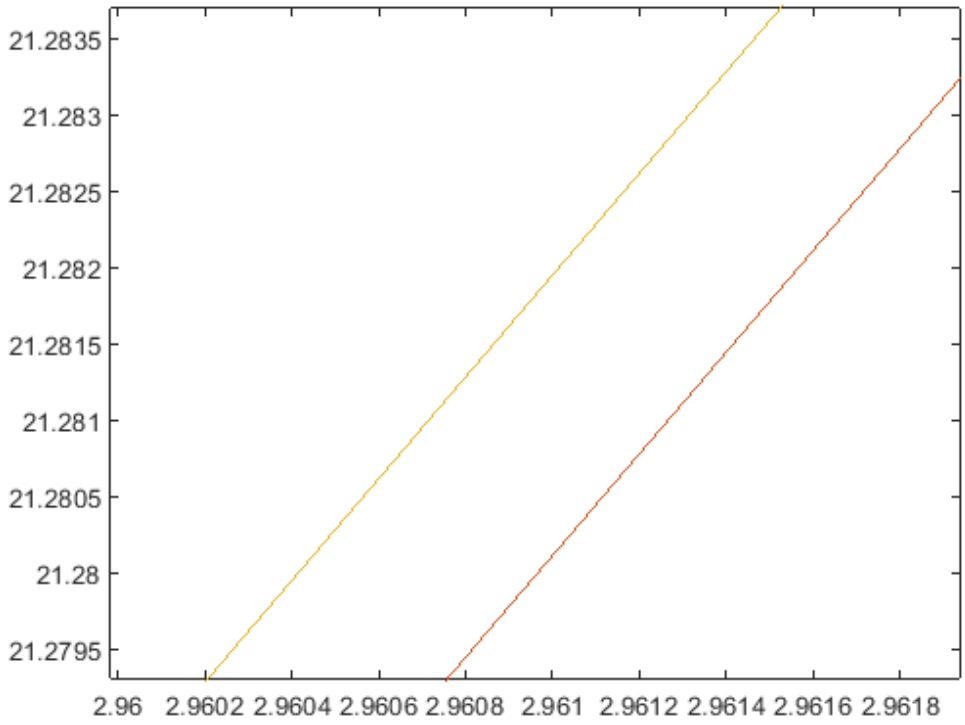
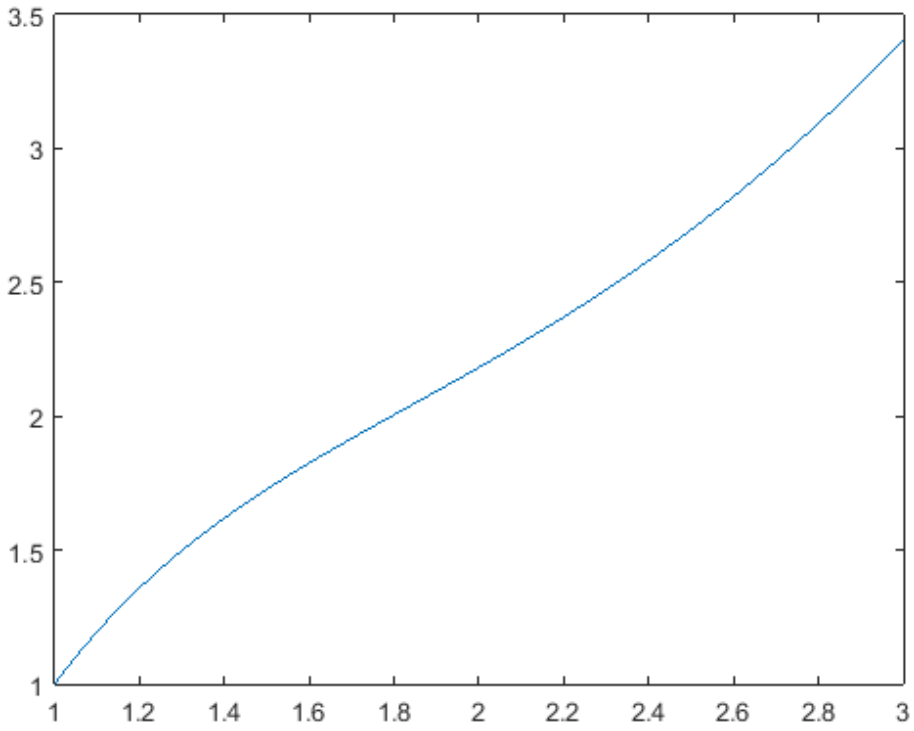


Figure 6: All methods at $Y_2(x_i)$ which is the approximation of $y'(x_i)$



4 Falkner Skan Equation

Flow past a wedge is governed by the Falkner-Skan equation. This equation admits only numerical solution, which requires the application of the shooting technique we described in the prerequisites section.

The Falkner-Skan equations

$$f''' + ff'' + \beta(1 - f'^2) = 0$$

$$f(0) = f'(0) = 0$$

$$f' \to 1 \quad as \quad \eta \to \infty$$

We write the Falkner equation as a first order ODE as following:

$$\frac{df}{d\eta} = f'$$

$$\frac{df'}{d\eta} = f''$$

$$\frac{df''}{d\eta} = -ff'' - \beta(1 - f'^2)$$

We can use the Runge-Kutta Method to obtain a numerical solution to the initial value problem with the two boundary conditions at $\eta = 0$ and a guess $f''(0) = g$.

But our purpose is to solve the Falkner-Skan boundary problem.To do that we will use the shooting method with Newton iterations with the boundary condition

$$f' \to 1 \quad as \quad \eta \to \infty$$

being satisfied.

The idea is to write the boundary value problem in vector form and begin the solution at one end of the boundary value problem, and shoot to the other end with an initial value solver(4th order Runge-Kutta) until the boundary condition at the other end converges to its correct value($f' \to 1$).

By using the Shooting method with Newton iterations we obtain the following set of first order ODEs:

$$\frac{dZ_1}{dh} = Z_2$$

$$\frac{dZ_2}{dh} = Z_3$$

$$\frac{dZ_3}{dh} = -f''Z_1 + 2'Z_2 - fZ_3$$

Now we need to define a class-function and set the f, f', Z_2, Z_3.

After running our code for 100 Newton iterations, $\eta = 100$ and 8000 Steps for the Runge-Kutta method we obtain the following tables:

Table 4: $6.175 \leq \eta \leq 6.2875$

f	f'	f''	η
5.3704514	0.99999999	3.9979267e-08	6.175
5.3829514	0.99999999	3.7298974e-08	6.1875
5.3954514	0.99999999	3.4793095e-08	6.2
5.4079514	0.99999999	3.2450644e-08	6.2125
5.4204514	0.99999999	3.0261306e-08	6.225
5.4329514	1	2.8215393e-08	6.2375
5.4454514	1	2.6303807e-08	6.25
5.4579514	1	2.4518008e-08	6.2625
5.4704514	1	2.2849979e-08	6.275
5.4829514	1	2.1292197e-08	6.2875

Table 5: Last 20 terms with $\eta = 100$

f	f'	f''	η
98.957951	1	2.8050713e-16	99.7625
98.970451	1	2.8047169e-16	99.775
98.982951	1	2.8043627e-16	99.7875
98.995451	1	2.8040085e-16	99.8
99.007951	1	2.8036544e-16	99.8125
99.020451	1	2.8033004e-16	99.825
99.032951	1	2.8029465e-16	99.8375
99.045451	1	2.8025927e-16	99.85
99.057951	1	2.802239e-16	99.8625
99.070451	1	2.8018853e-16	99.875
99.082951	1	2.8015318e-16	99.8875
99.095451	1	2.8011783e-16	99.9
99.107951	1	2.8008249e-16	99.9125
99.120451	1	2.8004717e-16	99.925
99.132951	1	2.8001185e-16	99.9375
99.145451	1	2.7997654e-16	99.95
99.157951	1	2.7994124e-16	99.9625
99.170451	1	2.7990594e-16	99.975
99.182951	1	2.7987066e-16	99.9875
99.195451	1	2.7983538e-16	100

Also when $\eta = 50$ then $f(50) = 49.195451$. And when $\eta = 100$ then $f(100) = 99.195451$

So we have three important facts:

- $f' \to 1$ when $\eta \gtrapprox 6.225$ and $\beta \geq 0$
- $f'' \to 0$ as $\eta \to \infty$
- $2f(\eta) \approx f(2\eta)$

We present the following plots of $f(\eta), f'(\eta), f''(\eta)$ against η for $\beta = 0, 1/2$ and 1.

Figure 7: solution $f(\eta)$

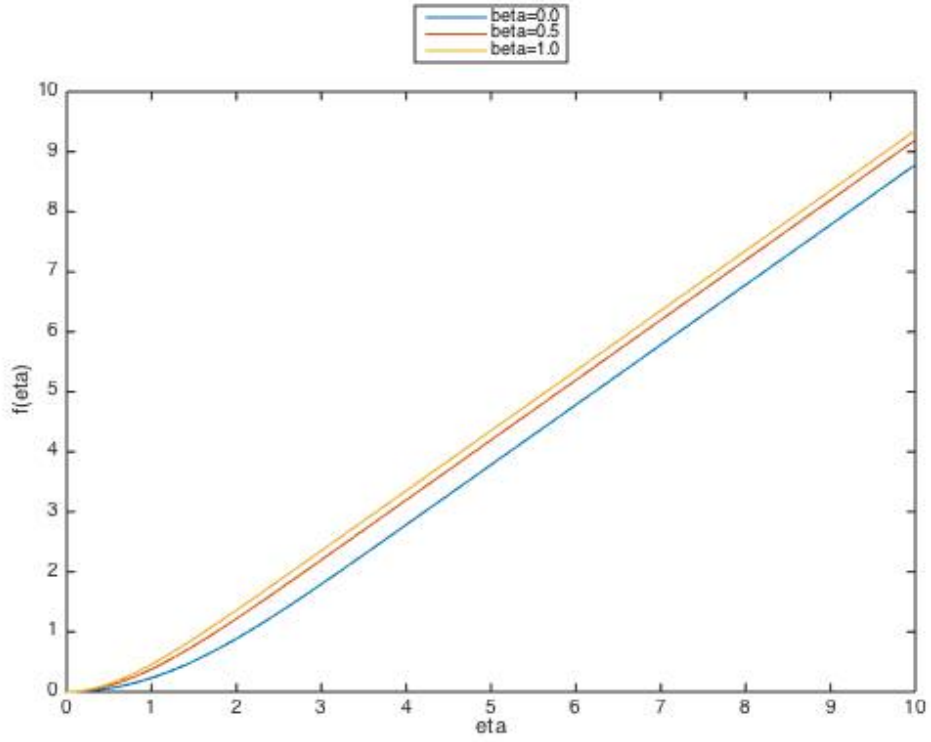


Figure 8: $f'(\eta)$

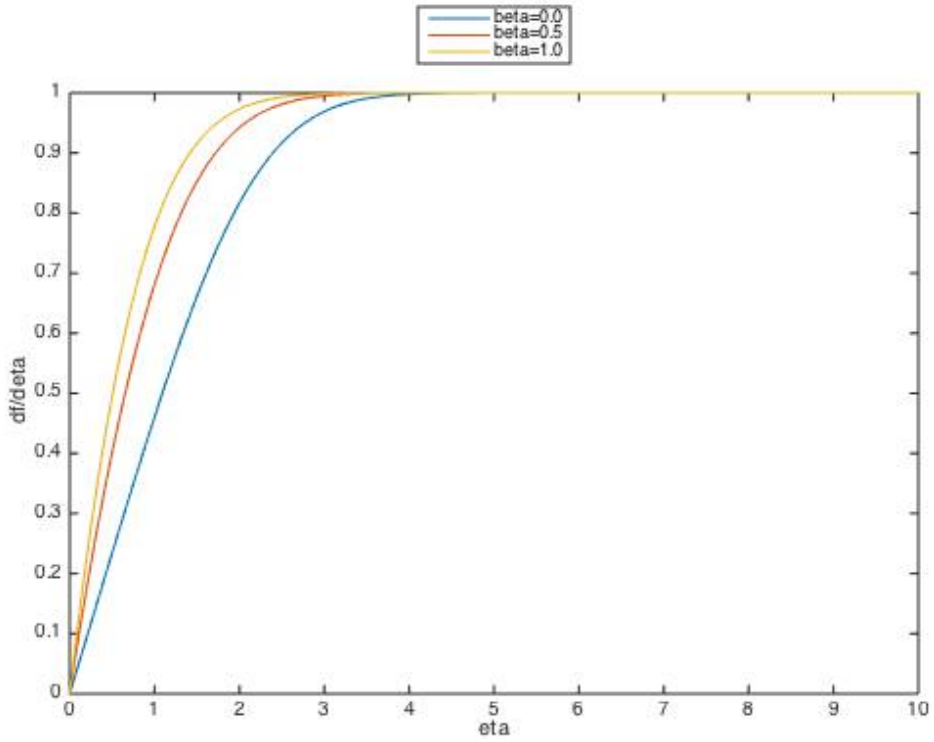
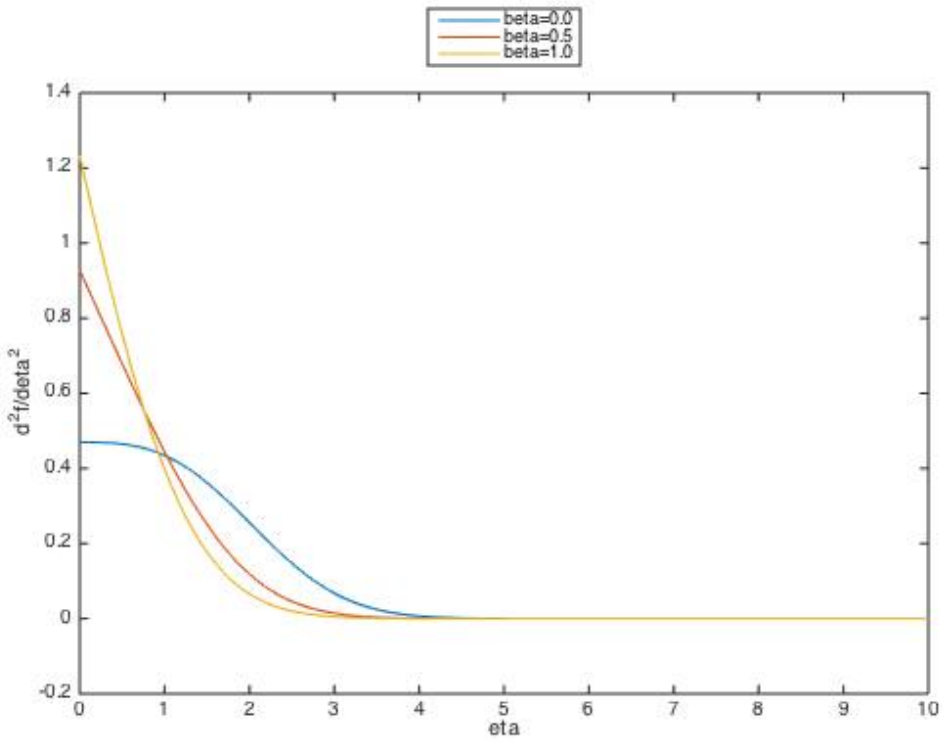
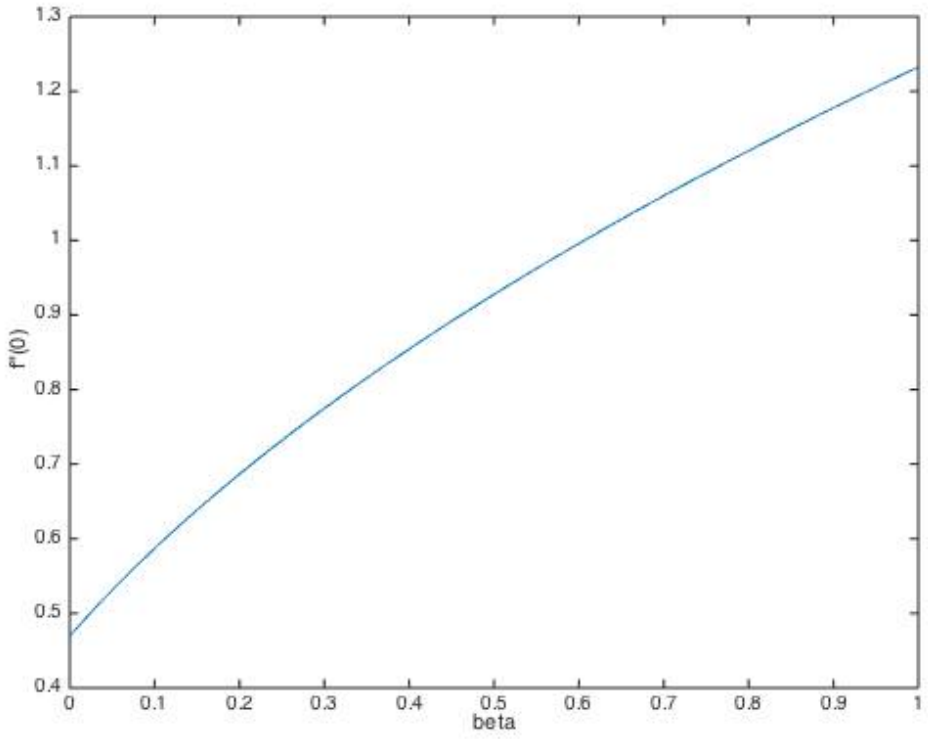


Figure 9: $f''(\eta)$



- $f''(0) = 0.4695999914$ when $\beta = 0$
- $f''(0) = 0.9276800374$ when $\beta = 1/2$
- $f''(0) = 1.232587597$ when $\beta = 1$

Figure 10: $f''(0)$ against β for $0 \leq \beta \leq 1$



What if β is negative?

K. Stewartson says that" It is found that when $\beta < 0.1988$ there are no solutions satisfying $\|f'(\eta)\| \leq 1$ and that if $0 > \beta > 0.1988$ there are two acceptable solutions, one with $f(0) < 0$. The new ones are computed to three places of decimals for various values of β. In addition, it is shown that if $05 < \beta < 0$ there is a family of solutions corresponding to boundary layers bounded on one side by free streamlines".(Further solutions of the Falkner-Skan equation,California Institute of Technology Pasadena 4, California, U.S.A)

5 C++ Code

```
#include<iostream>
#include<cmath>
#include<ostream>
#include<fstream>
#include<string>
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2
```

```
8
9  #include <vector>
10 using namespace std;
11
12
13 // Class that represents a mathematical vector
14 class MVector
15 {
16 public:
17         // constructors
18         MVector() {}
19         explicit MVector(int n) : v(n) {}
20         MVector(int n, double x) : v(n, x) {}
21
22         // access element (lvalue)
23         double &operator[](int index) { return v[index]; }
24
25         // access element (rvalue)
26         double operator[](int index) const { return v[
              index]; }
27
28         int size() const { return v.size(); } // number of
               elements
29         void push_back(double x) { v.push_back(x); }
30 private:
31         std::vector<double> v;
32 };
33
34 #endif
35
36 // Operator overload for "scalar * vector"
37 inline MVector operator*(const double& lhs, const MVector&
       rhs)
38 {
39         MVector temp(rhs);
40         for (int i = 0; i<temp.size(); i++) temp[i] *= lhs
              ;
41         return temp;
42 }
43 // Operator overload for vector / scalar
44 inline MVector operator/(const MVector& rhs, const double&
       lhs)
45 {
```

```
46          if (lhs == 0) cout << "The denominator is 0" <<
               std::endl;
47
48          else {
49                  MVector temp(rhs);
50                  for (int i = 0; i < temp.size(); i++) temp
                       [i] /= lhs;
51                  return temp;
52          }
53  }
54  // Operator overload for "vector+vector"
55  inline MVector operator+(const MVector& rhs, const MVector
       & lhs)
56  {
57          if (rhs.size() != lhs.size()) cout << "These
               vectors have different size" << std::endl;
58
59          else {
60                  MVector temp(rhs), temp1(lhs);
61                  for (int i = 0; i < temp.size(); i++)
62                          temp[i] += temp1[i];
63
64                  return temp;
65          }
66  }
67
68  // Operator overload for "vector-vector"
69  inline MVector operator-(const MVector& rhs, const MVector
       & lhs)
70  {
71          if (rhs.size() != lhs.size()) cout << "These
               vectors have different size" << std::endl;
72
73          else {
74                  MVector temp(rhs), temp1(lhs);
75                  for (int i = 0; i < temp.size(); i++)
76                          temp[i] -= temp1[i];
77                  return temp;
78          }
79  }
80
81  // Operator overload for "vector*vector"
82  double operator*(const MVector& rhs, const MVector& lhs)
```

```
83
84  {
85          if (rhs.size() != lhs.size()) cout << "These
               vectors have different size" << std::endl;
86
87
88          else
89          {
90                  MVector temp(rhs), temp2(lhs);
91                  double sum = 0.0;
92                  for (int i = 0; i < temp.size(); i++) {
93                          sum += temp[i] * temp2[i];
94
95                  }
96                  return sum;
97          }
98  }
99
100 ostream& operator<<(ostream& os, const MVector& v)
101 {
102         int n = v.size();
103         //os << "(";
104         for (int i = 0; i < n; i++) {
105
106                 os << v[i] << " ";
107         }
108
109         os << ")";
110         return os;
111 }
112 struct MFunction
113 {
114         virtual MVector operator()(const double& x,
115                 const MVector& y) = 0;
116 };
117 class FunctionF1 : public MFunction
118 {
119 public:
120         MVector operator()(const double& x, const MVector&
                y)
121         {
122                 MVector temp(2);
123                 temp[0] = y[0] + x*y[1];
```

```
124                 temp[1] = x*y[0] - y[1];
125                 return temp;
126         }
127 };
128
129 class FunctionF2 : public MFunction
130 {
131 public:
132         MVector operator()(const double& x, const MVector&
                y)
133         {
134                 MVector temp(2);
135                 temp[0] = x;
136                 temp[1] = y[1];
137                 return temp;
138         }
139 };
140
141 class FunctionF3 : public MFunction
142 {
143 public:
144         MVector operator()(const double& x, const MVector&
                y)
145         {
146                 MVector temp(2);
147                 temp[0] = y[1];
148                 temp[1] = (32 + 2 * pow(x, 3) - y[0] * y
                    [1]) / 8;
149                 return temp;
150         }
151 };
152
153
154
155 // Declaration for an Euler scheme ODE solver
156
157 int EulerSolve(int steps, double a, double b, MVector &y,
    MFunction &f)
158 {
159         ofstream outFile("Euler.txt");
160         double x, h = (b - a) / steps;
161         MVector yy;
162
```

```
163
164         for (int i = 0; i <= steps; i++) {
165
166                 x = a + i*h;
167                 yy = y;
168                 y = yy + h*f(x, y);
169                 outFile.precision(16);
170                 outFile.width(10);  outFile << yy << "\t";
171                 outFile.width(10);  outFile << x << endl;
172
173
174
175         }
176
177         cout << yy << endl;
178         return 0;
179 }
180
181 // Declaration for Midpoint scheme ODE solver
182 int Midpoint(int steps, double a, double b, MVector &y,
      MFunction &f)
183 {
184         ofstream outFile("Midpoint.txt");
185         double x, h = (b - a) / steps;
186         MVector yy;
187
188         for (int i = 0; i <= steps; i++) {
189
190                 x = a + i*h;
191                 yy = y;
192                 y = yy + h*f(x + h / 2, yy + h*f(x, yy) /
                      2);
193                 outFile.precision(16);
194                 outFile.width(10);  outFile << yy << "\t";
195                 outFile.width(10);  outFile << x << endl;
196         }
197         cout << yy << endl;
198         return 0;
199
200
201 }
202
203 //Declaration for Runge Kutta scheme ODE solver
```

```
204
205 int RungeKutta(int steps, double a, double b, MVector &y,
        MFunction &f, string filename = "")
206 {
207         {
208
209                 bool writeToFile = filename.size()>0;
210                 std::ofstream myFile;
211                 if (writeToFile)
212                 {
213                         myFile.open(filename.c_str());
214                         if (!myFile)
215                         {
216                                 std::cout << "Could not
                                    open " << filename <<
                                    std::endl;
217                                 writeToFile = false;
218                         }
219                 }
220
221
222                 double x, h = (b - a) / steps;
223                 MVector yy, k1, k2, k3, k4;
224                 if (writeToFile)
225                 {
226                         myFile.precision(8);
227                         myFile.width(10);       myFile <<
                            y << "\t";
228                         myFile.width(10);       myFile <<
                            a << endl;
229                 }
230
231
232                 for (int i = 0; i < steps; i++) {
233
234                         x = a + i*h;
235                         yy = y;
236                         k1 = f(x, y);
237                         k2 = f(x + h / 2, y + (h / 2)*k1);
238                         k3 = f(x + h / 2, y + (h / 2)*k2);
239                         k4 = f(x + h, y + h*k3);
240                         y = yy + (h / 6)*(k1 + 2 * k2 + 2
                            * k3 + k4);
```

```
241
242                         if (writeToFile)
243                         {
244
245                                 myFile.width(10);
                                                    myFile
                                    << y << "\t";
246                                 myFile.width(10);
                                    myFile << x + h << endl
                                    ;
247                         }
248
249
250                 }
251
252                 cout.precision(5);
253
254
255                 if (writeToFile)
256                 {
257                         myFile.close();
258                 }
259         }
260         return 0;
261 }
262
263
264
265 class Eqn1p5Derivs : public MFunction //Class Representing
        the function derivatives (1.2.4)
266 {
267 public:
268         // constructor to initialise kappa
269         Eqn1p5Derivs() { kappa = 1.0; }
270         MVector operator()(const double& x, const MVector&
            y)
271         {
272                 MVector temp(4);
273                 temp[0] = y[1];
274                 temp[1] = -kappa*y[1] - x*y[0];
275                 temp[2] = y[3];
276                 temp[3] = -kappa*y[3] - x*y[2];
277                 return temp;
```

```
278         }
279         void SetKappa(double k) { kappa = k; } // change
               kappa
280 private:
281         double kappa; // class member variable, accessible
               within
282                                     // all Eqn1p5Derivs
                                           member functions
283 };
284
285 class FunctionF4 : public MFunction
286 {
287 public:
288         MVector operator()(const double& x, const MVector&
               y)
289         {
290
291                 MVector temp(4);
292                 temp[0] = y[1];
293                 temp[1] = (32 + 2 * pow(x, 3) - y[0] * y
                       [1]) / 8;
294                 temp[2] = y[3];
295                 temp[3] = -(y[0] * y[3]) / 8 - (y[1] * y
                       [2]) / 8;
296                 return temp;
297
298         }
299 };
300
301 class FunctionF5 : public MFunction
302 {
303 public:
304         MVector operator()(const double& x, const MVector&
               y)
305         {
306                 double b = 1.0 / 2.0;
307                 MVector temp(3);
308                 temp[0] = y[1];//=y'
309                 temp[1] = y[2];//=y"
310                 temp[2] = -y[0] * y[2] - b*(1 - pow(y[1],
                       2));
311                 return temp;
312         }
```

```
313 };
314
315
316 class FunctionF6 : public MFunction
317 {
318 public:
319         FunctionF6() { beta = 0.5; }
320         MVector operator()(const double& x, const MVector&
                y)
321         {
322                 MVector temp(6);
323                 temp[0] = y[1];//=y'
324                 temp[1] = y[2];//=y"
325                 temp[2] = -y[0] * y[2] - beta*(1 - pow(y
                    [1], 2));
326                 temp[3] = y[4]; //z2
327                 temp[4] = y[5];  //z3
328                 temp[5] = -y[2] * y[3] + 2 * beta*y[1] * y
                    [4] - y[0] * y[5];
329
330                 return temp;
331         }
332         void SetBeta(double b) { beta = b; } // change
              beta
333 private:
334         double beta;
335 };
336
337 class FunctionF7 : public MFunction //
338 {
339 public:
340
341         MVector operator()(const double& x, const MVector&
                y)
342         {
343                 double b; //try for b=0,1/2,1
344                 MVector temp(6);
345                 temp[0] = y[1];//=y'
346                 temp[1] = y[2];//=y"
347                 temp[2] = -y[0] * y[2] - b*(1 - pow(y[1],
                    2));
348                 temp[3] = y[4]; //z2
349                 temp[4] = y[5];  //z3
```

```
350                 temp[5] = -y[2] * y[3] + 2 * b*y[1] * y[4]
                        - y[0] * y[5];
351
352                 return temp;
353         }
354 };
355
356 double FalknerSkan(double b, double k)
357 {
358
359
360         FunctionF6 f;
361         f.SetBeta(b);
362         int maxNewtonSteps = 100;
363         double tol = pow(10.0, -8);
364         for (int i = 0; i < maxNewtonSteps; i++)
365         {
366
367                 MVector y(6);
368
369                 y[0] = 0; y[1] = 0; y[2] = k; y[3] = 0; y
                        [4] = 0; y[5] = 1.0;
370                 RungeKutta(1000, 0, 1, y, f, "natasa.txt")
                        ; // solve IVP
371                 double phi = y[1] - 1.0; // calculate
                        residual
372                 double phidash = y[4]; // 'Jacobian'
                        phidash = z_1(x=1)
373                 cout << i << endl;
374                 if (std::abs(phi) <= tol) break; // exit
                        if converged/
375                 k -= phi / phidash;
376                 //if (std::abs(phi) > tol){
377                 //      maxNewtonSteps = maxNewtonSteps +
                        1;
378                 cout << k << "    " << b << endl;
379         }
380         cout << k << "    " << b << endl;
381         return k;
382 }
383
384
385
```

```
386 int main()
387 {
388
389
390
391         double h = 0.1, x = 0.5;
392
393         MVector u;
394                 u.push_back(1);
395                 u.push_back(2);
396
397
398
399
400                 cout << "u=" << u << endl;
401
402
403          MVector v, y(2); // initialise y with 2 elements
404                         FunctionF1 f; // f has order 2 by
                                definition
405                         y[0] = 1.4; y[1] = -5.7; // assign
                                element values in y
406                         v = f(2., y); // evaluate function
                                f as required
407
408                         std::cout << "v=" << v << "; y="
                                << y << std::endl;
409                         v = u + f(2.0, y);
410                         std::cout << "v=" << v << "; y="
                                << y << std::endl;
411
412                         v = u + h*f(x,u+h*y);
413
414                         std::cout << "v=" << v << "; y="
                                << y << std::endl;
415
416
417                 MVector v, y(2); // initialise y with 2
                        elements
418                 FunctionF2 f;
419
420                         y[0] = 0; y[1] = 1;
421
```

```
422
423             EulerSolve(1000, 0, 1, y, f); // If f and
                    y where diferent numbers then we would
                    have different size ventors to
                    multiplay and so we would have an error
424                 y[0] = 0; y[1] = 1;
425                 Midpoint(1000, 0, 1, y, f);
426                 y[0] = 0; y[1] = 1;
427             RungeKutta(1000, 0, 1, y, f,"rk88.txt");
428
429         MVector v, y(2); // initialise y with 2 elements
430         FunctionF3 f;
431
432         y[0] = 17; y[1] = 1;
433
434
435         RungeKutta(1000, 1.0, 3.0, y, f, "Rk88.txt");
436         y[0] = 17; y[1] = 1;
437         EulerSolve(1000, 1.0, 3.0, y, f); // If f and y
                where diferent numbers then we would have
                different size ventors to multiplay and so we
                would have an error
438         y[0] = 17; y[1] = 1;
439         Midpoint(1000, 1.0, 3.0, y, f);
440
441
442
443
444             Eqn1p5Derivs f;
445             int maxNewtonSteps = 100;
446             double guess = 0;
447             double tol = 1e-8;
448             for (int i = 0; i < maxNewtonSteps; i++)
449             {
450             MVector y(4);
451                     // y[0] = y, y[1] = dy/dx, y[2] =
                            Z_1, y[3] = Z_2
452                     y[0] = 0; y[1] = guess; y[2] =
                            0.0; y[3] = 1.0;
453                     RungeKutta(100, 0.0, 1.0, y, f);
                            // solve IVP
454                     double phi = y[0] - 1; //
                            calculate residual
```

```
455                         double phidash = y[2]; // '
                               Jacobian' phidash = z_1(x=1)
456                         if (std::abs(phi) < tol) break; //
                                exit if converged
457                         guess -= phi / phidash; // apply
                               newton step
458                 }
459
460
461                 return 0;
462
463
464                 FunctionF4 f;
465                 int maxNewtonSteps = 1;
466                         double guess = 0;
467                         double tol = pow(10.0,-8);
468                         for (int i = 0; i < maxNewtonSteps
                               ; i++)
469                         {
470
471                                 MVector y(4);
472                         //y[0] = y, y[1] = dy / dx, y[2] =
                                Z_1, y[3] = Z_2
473                                 y[0] = 17.0; y[1] = guess;
                                     y[2] = 0.0; y[3] =
                                    1.0;
474
475                    RungeKutta(100, 1.0, 3.0, y, f); //
                          solve IVP
476                                 double phi = y[0] -
                                    43.0/3.0; // calculate
                                    residual
477                                 double phidash = y[2]; //
                                    'Jacobian' phidash =
                                    z_1(x=1)
478                                 if (std::abs(phi) <= tol)
                                    break; // exit if
                                    converged
479                          guess -= phi / phidash;
480                                 if (std::abs(phi) > tol){
481                                 maxNewtonSteps =
                                    maxNewtonSteps + 1;
482         }
```

```
483
484
485
486
487
488                         }
489
490
491                 MVector u,w; //VECTOR*VECTOR TEST
492                 u.push_back(2);
493         u.push_back(2);
494                 u.push_back(2);
495                 w.push_back(3);
496            w.push_back(4);
497
498                 cout << u << w << endl;
499                 cout << u+w << endl;
500
501       MVector v, y(3); // initialise y with 2 elements
502           FunctionF5 f;
503         double guess = 0.92;
504                 y[0] = 0; y[1] = 0; y[2] = guess;
505            RungeKutta(8000, 0, 100, y, f, "poutsa2.txt");
506
507         MVector v, y(6);
508         FunctionF6 f;
509     double guess = 0.92;
510
511                 y[0] = 0; y[1] = 0; y[2] = guess; y[3] =
                       0; y[4] = 0; y[5] = 1;
512         RungeKutta(8000, 0, 100, y, f, "poutsa1.txt");
513
514         FunctionF6 f; //last Shit
515                 int maxNewtonSteps = 100;
516                 double guess = 0.92;
517         double tol = pow(10.0, -8);
518         for (int i = 0; i < maxNewtonSteps; i++)
519         {
520
521                         MVector y(6);
522
523                         y[0] = 0; y[1] = 0; y[2] = guess;
                           y[3] = 0; y[4] = 0; y[5] = 1.0;
```

```
524         RungeKutta(8000, 0, 100, y, f, "poutsa.txt"); //
               solve IVP
525                         double phi = y[1] - 1.0; //
                               calculate residual
526         double phidash = y[4]; // 'Jacobian' phidash = z_1
               (x=1)
527                         cout << i << endl;
528                         if (std::abs(phi) <= tol) break;
                               // exit if converged/
529                         guess -= phi / phidash;
530                         if (std::abs(phi) > tol) {
531                                 //      maxNewtonSteps =
                                       maxNewtonSteps + 1;
532                         }
533                 }
534         FalknerSkan(0.5, 0.92);
535         ofstream outFile("Falkner88.txt");
536
537         double step = 0.0001;
538         double out;
539         double Equation = FalknerSkan(0.5, 0.92);
540     for (double b = 0.000; b <= 1.0; b = b + step)
541         {
542                 out = FalknerSkan(b, Equation);
543                 Equation = out;
544                 cout << out << "    " << b << endl;
545                 outFile.precision(10);
546                 outFile.width(10);  outFile << out << "\t
                       ";
547         outFile.width(10);  outFile << b << endl;
548         }
549
550         FalknerSkan(-1, 0.92);
551         ofstream outFile("natasa88.txt");
552
553         double step = 0.01;
554         double out;
555         double Equation = FalknerSkan(-1, 0.92);
556         for (double b = -1; b <= -0.2; b = b + step)
557         {
558                 out = FalknerSkan(b, Equation);
559                 Equation = out;
560                 cout << out << "    " << b << endl;
```

```
561                     outFile.precision(10);
562                     outFile.width(10);   outFile << out << "\t
                            ";
563                     outFile.width(10);   outFile << b << endl;
564             }
565
566             return 0;
567     }
```