# Numerical-solver Assignment Report

for course 'Numerical Solution of Differential Equations I' [4]

Hui Jia Farm

April 10, 2021

# Contents

# Chapter 1

# Introduction

This report describes the work undertaken as an assignment for the Numerical Solution for Differential Equations I course given in the Mathematical Institute by Professor Endre Süli. It describes the implementation of a number of methods for the numerical solution of ordinary differential equations (ODEs) as an open source Python software package that I have developed and made available on GitHub. The package makes extensive use of software engineering techniques to ensure that the software is robust and fully tested.

I have also developed a number of tutorial examples as Jupyter Notebooks to explain how the methods work and to demonstrate their use on some real examples. They are available through links, indicated by underlined italic font, provided throughout the report.

The structure of this report is as follows: in Chapter 2, I describe the domain of ODEs and the numerical methods that I implement, and gives links to the software and tutorial notebooks for a simple test problem; Chapter 3 describes the implementation and testing of the software; Chapter 4 describes the application of the methods to the Fitzhugh-Nagumo model; and Chapter 5 gives some brief conclusions.

# Chapter 2

# Numerical methods

## 2.1 Ordinary differential equation

An ordinary differential equation (ODE) is an equation that contains derivatives of single independent variable functions. Examples of ODEs would be $\frac{df}{dx} = -x$ and $\frac{df}{dx} + \frac{dg}{dx} = 4x$, where $x$ is an independent variable, $f$ and $g$ are functions of $x$. In general, ODEs are used to describe changes. One of the simplest example is speed, change in distance travelled per unit time.

There are different ways to describe an ODE. The order of an ODE is the order of the highest derivative in an ODE. The ODE $\frac{d^3f}{dx^3} = -1$ has order 3. Other than the order, ODEs can be classified into linear or non-linear ODE. Linear ODEs are functions that can be expressed as linear combinations of derivatives, while non-linear ODEs cannot. The equation, $\frac{df}{dx}\frac{dg}{dx} = 1$, for example, is non-linear ODE; the equation $\frac{df}{dx} + \frac{dg}{dx} = 1$ is linear. Linear ODE can further be categorised into homogeneous and non-homogeneous ODEs. In a general linear ODE, $a_0(x)f(x)+a_1(x)f'(x)+...+a_n(x)f^{(n)}(x)+b(x) = 0$, the function is homogeneous if $b(x) = 0$ and non-homogeneous if $b(x) \neq 0$.

ODEs are widely used in biological problem. An example of biological problem would be the SEIR model. The SEIR model describes the spread of a disease in a population with time $t$. The model compartmentalises the population into groups of susceptibles ($S$), exposed ($E$), infectious ($I$) and recovered ($R$). Transitions of individuals from groups are described by a

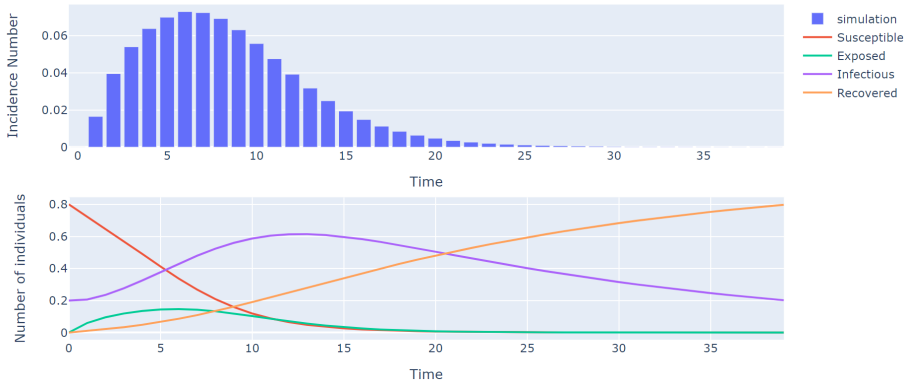


Figure 2.1: Simulation of the SEIR model. The bar graph shows the incidence number, while the line graph shows the number of individuals in each group. All values shown are in fractions of the whole population.

system of ODEs, with independent variable time $t$. It is defined as

$$\frac{dS(t)}{dt} = -\beta S(t)I(t), \tag{2.1}$$

$$\frac{dE(t)}{dt} = \beta S(t)I(t) - \kappa E(t), \tag{2.2}$$

$$\frac{dI(t)}{dt} = \kappa E(t) - \gamma I(t), \tag{2.3}$$

$$\frac{dR(t)}{dt} = \gamma I(t) \tag{2.4}$$

where the parameters $\beta$, $\kappa$ and $\gamma$ are infection rate, incubation rate and recovery rate respectively. The incidence cases, $N$ at time, $t$, is defined as the difference between the current total number of infected and recovered and that of the previous time, according to the equation

$$N(t) = I(t) + R(t) - I(t-1) - R(t-1) \tag{2.5}$$

Figure 2.1 shows the incidence cases simulated by the SEIR model.

Another example is the modelling of excitable systems. The heart muscle cells and nerve cells are excitable systems. The Hodgkin & Huxley model models the ionic currents across the cell membrane, to explain the action potential of nerve cells[3]. Their model is defined as

$$C_m \frac{dV}{dt} = -g_{\text{eff}}(V - V_{\text{eq}}) + I_{\text{app}} \tag{2.6}$$

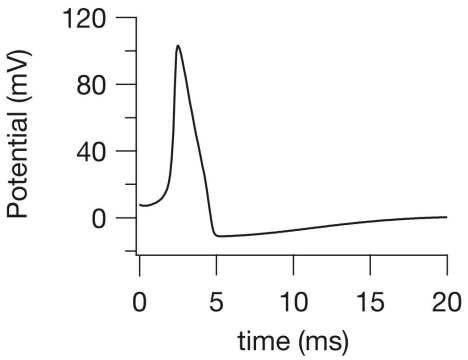


Figure 2.2: The action potential in the Hodgkin and Huxley model, adapted from [3].

where $C_m$ is membrane capacitance, $V$ is membrane potential, $g_{\text{eff}}$ is sum of conductance for all ion channels, $V_{\text{eq}}$ is membrane resting potential and $I_{\text{app}}$ is applied current. Here, the independent variable is also time $t$. The value of potential is obtained by subtracting the membrane resting potential from the membrane potential, that is $V - V_{\text{eq}}$. Figure 2.2 shows the simulated action potential of the Hodgkin and Huxley model.

## 2.2 Initial value problem

Given an ODE system, we would like to solve for the function. As an example, given an ODE $\frac{df(x)}{dx} = -f(x)$, we would like to know the function $f$. To solve ODE systems, initial conditions are required to pinpoint the solution of interest, as there are infinitely possible solutions to an ODE. The ODEs, together with the initial conditions, set up the initial value problem.

A general form of an initial value problem is as follows:

$$y' = f(x, y) \tag{2.7}$$

$$y(x_0) = y_0 \tag{2.8}$$

for $x \in [x_0, X_M]$, where Eq. (2.8) describes the initial condition. For example, initial values of $S(0) = 0.8, E(0) = 0, I(0) = 0.1$ and $R(0) = 0$ are used to solve the SEIR model in Figure 2.1.

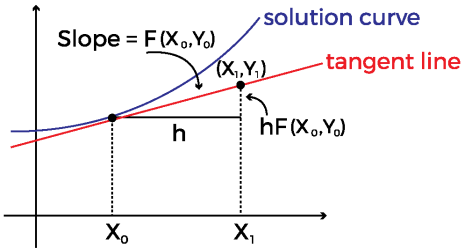


Figure 2.3: Intuition of Euler's method, adapted from [1].

## 2.3 Numerical method

Many ODE systems cannot be solved analytically. Some examples would be the given SEIR Model, Eqs. (2.1)-(2.4) and Hodgkin & Huxley Model, Eq. (2.6). Solutions to such models have to be estimated by numerical methods.

Throughout the report, the following notation will be used:

- $y_n$ - numerical approximation of $y(x_n)$
- $y(x_n)$ - analytical solution at mesh point $x_n$
- $x_n$ - mesh points of defined range, where

$$x_n = x_0 + nh \tag{2.9}$$

$$h = \frac{(X_M - x_0)}{N} \tag{2.10}$$

  for $n = 0, \ldots, N$

- $h$ - step size

### 2.3.1 One-step methods

The simplest numerical method in solving ODEs is Euler's explicit method. Intuitively, Euler's explicit method assumes for a small step size $h$, the solution of the function can be estimated by its tangent line. As seen in Figure 2.3, following the tangent line at point $x_0$, the difference between the $y$ value for point $x_0$ and point $x_1$ is $hF(x_0, y_0)$. So, the estimated value of $y_1$ is $y_0 + hF(x_0, y_0)$, where $y_0$ is the value of solution curve at $x_0$.

Euler's explicit method has the following definition,

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{2.11}$$

Starting from the initial value, the solution at the subsequent mesh point is estimated to follow a straight line with gradient as given.

The implementation is as follows:

**Algorithm 1:** Euler's explicit method

**Output:** mesh points and its numerical solution
solution = [initial value];
meshpoints = [starting point];
**for** *n from 1 to total number of mesh points* **do**
  solution.append(solution[$n-1$] + step size $\times$
    $f$(meshpoints[$n-1$], solution[$n-1$]);
  meshpoints.append(meshpoints[$n-1$] + $n \times$ step size);
**end**
**return** *meshpoints and solution*;

A vector containing the numerical solution and a vector of mesh points are initialised with their respective initial values. At each iteration, the mesh point and the numerical solution at the mesh point are calculated. The series of mesh points and its solution are returned. The mesh points are returned for consistency of the software, where all methods return the same outputs. Adaptive methods does not have fixed step sizes, thus it is important to return the mesh points.

Truncation error of the numerical methods is defined to be the difference between the exact solution and the numerical solution, assuming the exact solution at the previous mesh point is known. We have that the truncation error for Euler's explicit method is

$$T_n := \frac{y(x_{n+1}) - y(x_n)}{h} - f(x_n, y(x_n)) \tag{2.12}$$

According to Taylor's series expansion, we have

$$y(x_n + h) = y(x_n) + hy'(x_n) + \frac{1}{2}hy''(\xi_n) \tag{2.13}$$

for some $\xi_n \in (x_n, x_{n+1})$. Substitute (2.13) to (2.12), noting that $f(x_n, y(x_n)) = y'(x_n)$, we get

$$T_n = \frac{1}{2}hy''(\xi_n) \tag{2.14}$$

Therefore, the truncation error for Euler's explicit method varies linearly with the step size.

A test problem

$$f(x, y) = -y \tag{2.15}$$

$$y(0) = 1 \tag{2.16}$$

for $x \in [0, 5]$ is used throughout this report to check that the implementation is in line with the theory. The analytical solution to this problem is $y = e^{-x}$. Some examples of solutions to the given test problem Eqs. (2.15)-(2.16) can be found via this link: *test problem notebook*. This Jupyter Notebook is a tutorial example that demonstrates the use of the fully tested package I developed on the test problem.

With the test problem Eqs. (2.15)-(2.16) as a reference, the truncation error versus step size graph is shown in Figure 2.4. The computation and comparison of one-step methods and their truncation error can be found at this link: *error behaviour of one-step methods*. The truncation error is computed by taking the difference between the exact solution and the numerical solution at a randomly chosen $x$ value of 3. The truncation error is computed for solutions obtained by using different step sizes. Figure 2.4 is plotted in logarithmic scale for both variables. It can be observed that $\log |T_n|$ increases linearly with $\log h$. The line gradient of 1 shows that $|T_n| \propto h$, which matches the theoretical prediction.

Other than Euler's explicit method, the other one-step methods implemented are Euler's implicit method, the trapezium rule method and the four-stage explicit Runge-Kutta method. Euler's implicit method is defined to be

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}), \tag{2.17}$$

while the trapezium rule method is

$$y_{n+1} = y_n + \frac{1}{2}h[f(x_n, y_n) + f(x_{n+1}, y_{n+1})]. \tag{2.18}$$

The definition of the four-stage Runge-Kutta method is

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k4) \tag{2.19}$$

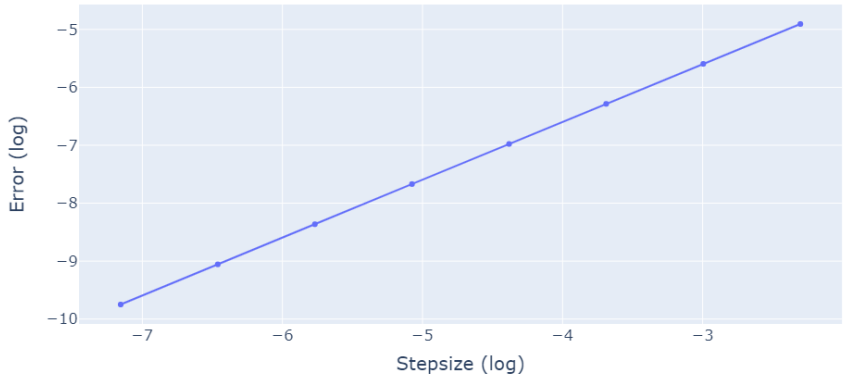


Figure 2.4: Truncation error of Euler's explicit method for various step sizes. The test problem Eqs. (2.15)-(2.16) is solved with Euler's explicit method at different step sizes. The error is the absolute difference between exact solution and numerical solution at $x = 3$.

where

$$k_1 = f(x_n, y_n) \tag{2.20}$$

$$k_2 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1) \tag{2.21}$$

$$k_3 = f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2) \tag{2.22}$$

$$k_4 = f(x_n + h, y_n + hk_3). \tag{2.23}$$

Using the same definition for truncation error as in Eq. (2.12), we have that the truncation error of Euler's implicit method, the trapezium rule method and the four-stage Runge-Kutta method are $T_n = -\frac{1}{2}hy''(\xi_n)$ for $\xi_n \in (x_n, x_{n+1})$, $T_n = -\frac{1}{12}h^2y^{(3)}(\xi_n)$ for $\xi_n \in (x_n, x_{n+1})$ and $T_n = \mathcal{O}(h^4)$ respectively.

These methods are tested on the test problem Eqs. (2.15)-(2.16), and the truncation error behaviour graph is shown in Figure 2.5. Since the graph is plotted in logarithmic scale, the gradient of the lines shows the order of accuracy of the methods. The gradient for Euler's explicit, Euler's implicit, the trapezium rule method and the four-stage Runge-Kutta method are 1, 1, 2 and 4 respectively. The gradients on the graph match the theoretical

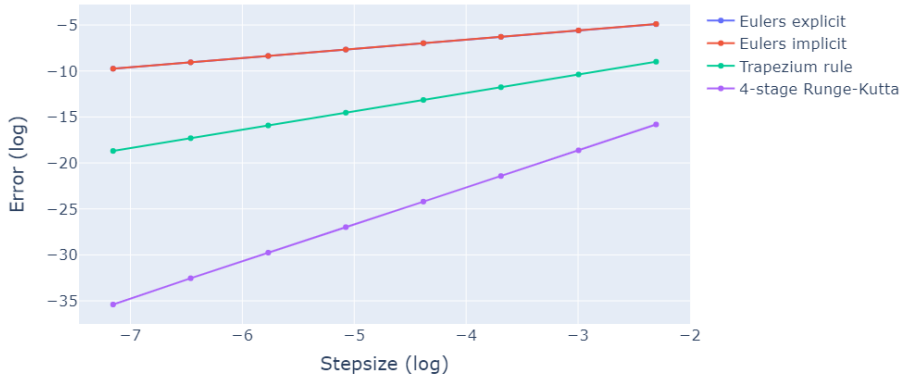


Figure 2.5: Truncation error of Euler's explicit method, Euler's implicit method, the trapezium rule method and the four-stage Runge-Kutta method for various step sizes. The test problem Eqs. (2.15)-(2.16) is solved with Euler's explicit method, Euler's implicit method, the trapezium rule method and the four-stage Runge-Kutta method at different step sizes. The error is the absolute difference between exact solution and numerical solution at $x = 3$. The line for Euler's implicit method overlaps the line for Euler's explicit method because both have the same order of accuracy. The truncation errors follow theoretical prediction.

behaviour of truncation error. In Figure 2.5, the truncation error behaviour of Euler's implicit method overlaps that of Euler's explicit method because both have the same order of accuracy.

### 2.3.2 Predictor-corrector methods

For implicit one-step methods, the fixed point iteration algorithm is used to obtain the numerical solution. Implicit functions cannot be solved with a general method, therefore a general algorithm, the fixed point iteration, is used to estimate the solution to the implicit functions. Initial guesses for this algorithm are taken as the numerical solution at previous mesh point. However, in practice, the computational cost of the fixed point algorithm is too high. The predictor-corrector method suggests a more carefully chosen initial guess for the implicit methods. An explicit numerical method is used as a predictor for the initial guess of an implicit method. The initial guess

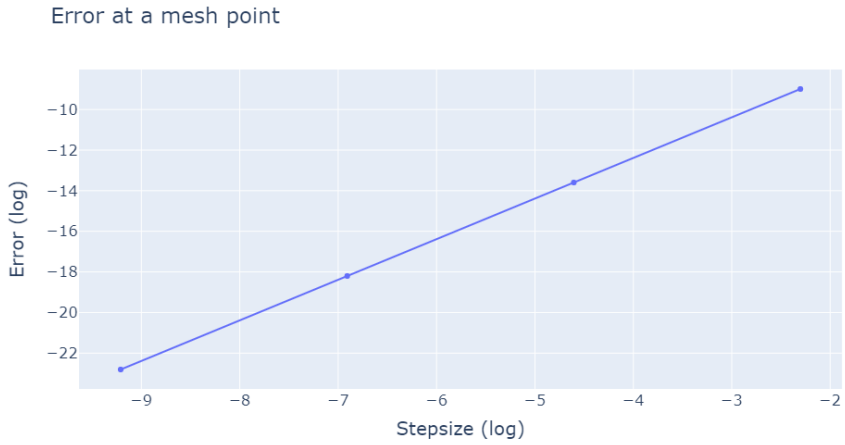


Figure 2.6: Error of Euler-Trapezoidal method for various step sizes. The test problem Eqs. (2.15)-(2.16) is solved with Euler-Trapezoidal method at different step sizes. The error is the absolute difference between exact solution and numerical solution at $x = 3$.

is then used for the iterations in the algorithm to solve an implicit function. The implicit method that refines the solution is known as the corrector method.

The Euler-Trapezoidal method is a predictor-corrector method that uses Euler's explicit method as the predictor and the trapezium rule method as the corrector. In this implementation, the trapezium rule method corrector is iterated until a set of conditions are satisfied. The conditions are set to be the difference between the current iteration and the previous iteration is lesser than a given threshold value or the number of iterations exceeds a certain amount. Figure 2.6 shows the graph of $\log |T_n|$ against $\log h$ for the Euler-Trapezoidal method.

### 2.3.3 Adaptive method

In some types of problem, the solution to the problem exhibits a behaviour where step sizes have to be sufficiently small to obtain a stable solution. In other words, the solution changes rapidly as a function of the independent variable in some regions of the solution domain. Such problems are called stiff problems. In order to obtain a stable solution for stiff problems, the

computational cost would be high. Moreover, such a stable solution would have resolution higher than required for practical purposes in some regions.

Adaptive methods try to achieve the desired accuracy with low computational cost. The main idea of an adaptive method is to control the precision at each mesh point and from there assumes the precision of the solution is within the desired range. To achieve this, the error at each mesh point is estimated. If the error is larger than a threshold value, a smaller step size is chosen. These steps are repeated until the error is smaller than the given threshold value.

The adaptive methods implemented in my package are based on the BS23 (Bogacki and Shampine) and RKF45 (Runge-Kutta-Fehlberg) method. A lower order method is used to estimate the solution, while a higher order method is used to estimate the error.

Absolute tolerance and relative tolerance are used in the implementation of the adaptive methods to control the error. Similarly, the implemented adaptive method is used to solve the test problem Eqs. (2.15)-(2.16) and tested for convergence. Two comparisons were made, one based on absolute tolerance and the other on relative tolerance. As shown in Figure 2.7 and Figure 2.8, the error of the method is smaller for smaller tolerance, regardless of absolute tolerance or relative tolerance.

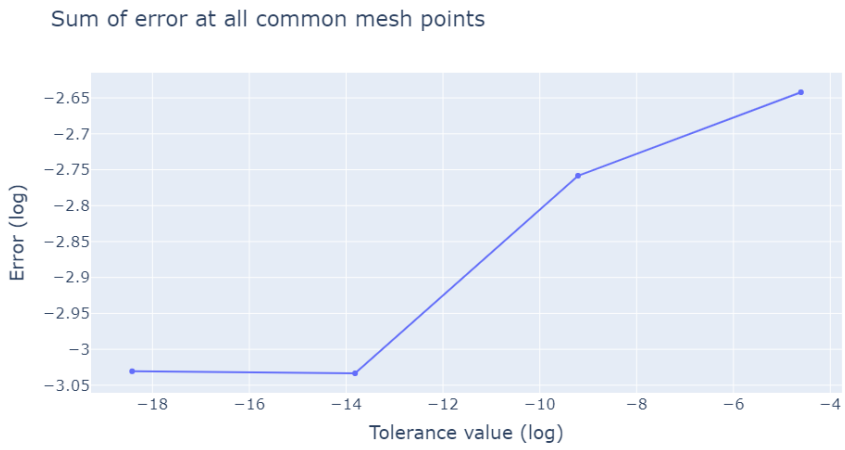


Figure 2.7: Total error of RKF45 method for various absolute tolerance values. The test problem Eqs. (2.15)-(2.16) is solved with RKF45 method at different absolute tolerance values. The relative tolerance is fixed at $1^{-15}$. The error is the sum of absolute difference between exact solution and numerical solution at all mesh points.

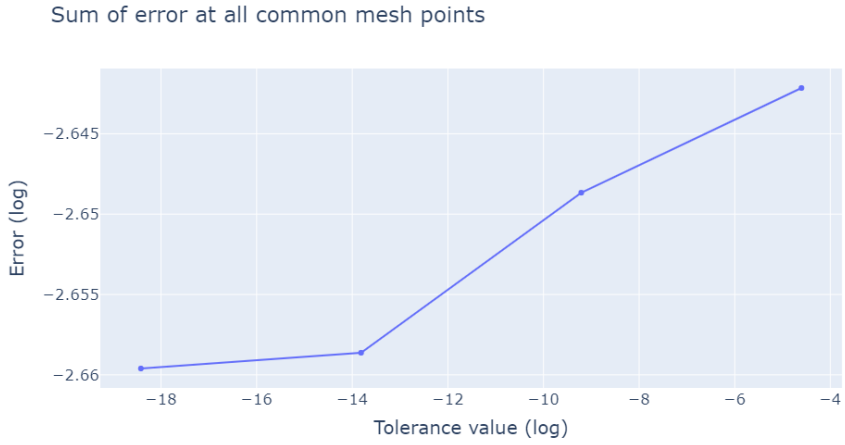


Figure 2.8: Total error of RKF45 method for various relative tolerance values. The test problem Eqs. (2.15)-(2.16) is solved with RKF45 method at different relative tolerance values. The absolute tolerance is fixed at $1^{-15}$. The error is the sum of absolute difference between exact solution and numerical solution at all mesh points.

# Chapter 3

# Implementation

## 3.1 Software overview

The code of the ODE solvers software that I have developed is available on GitHub via this link: *GitHub repository*. GitHub is a good platform for version control and collaborations. Software with version control can keep track of changes made to the code. It is very useful for large projects with collaborations. Aside from having version control, the software is fully documented. Details on its structure and functions are available at the link: *software documentation*. Finally, the software is also full-tested to ensure the correctness of the code. More details of the testing infrastructure are given in later sections.

To showcase the features of the code and to give a brief summary of the quality of the code, badges are displayed on the main page of the GitHub repository, as shown in Figure 3.1. These badges indicate that the code is tested to work using several python versions and operating systems. The badge 'codecov' shows the code coverage of the software, which will be described in details in the Section 3.3, together with the testing infrastructure. Finally, there is a status badge 'Doctest' to verify that the documentation of the software is built successfully.

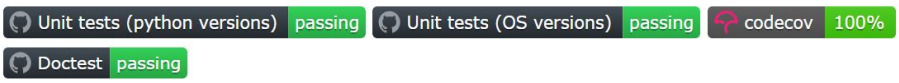


Figure 3.1: Badges on Github repository.

## 3.2 Implemented numerical methods

The numerical methods implemented in this software are classified into three classes: one-step methods, predictor-corrector methods and adaptive methods. The methods included are:

1. One-step methods
    - Euler's explicit method
    - Euler's implicit method
    - Trapezium rule method
    - Four-stage Runge-Kutta method
2. Predictor-corrector method
    - Euler-Trapezoidal method
3. Adaptive methods
    - BS23 algorithm
    - RKF45 algorithm

The main code containing these numerical methods can be found at the link: *methods code*.

## 3.3 Unit testing

In the process of developing the software, a unit testing infrastructure was put in place. The purpose of the unit testing is to create a robust and sustainable software. All codes are fully tested whilst being written (this is known as test-driven development) to ensure correctness and making future maintenance of the code much easier.

Code coverage is a measurement of the percentage of codes covered in the unit testing process. It is usual to aim for 100% code coverage. However, a 100% code coverage does not necessarily mean that the code is correct and free of errors. Nevertheless, it provides some confidence that the code is implemented correctly.

Every method in this software is tested using the 'unittest' Python library, a unit testing framework. The numerical solution produced by each method is tested to be the same solution as the manually calculated solution. The methods are mostly tested against the test problem Eqs. (2.15)-(2.16).

## 3.4 Testing initialisation of problem

The initialisations of each class are tested to ensure that variables are initialised correctly and input type satisfies the requirements. For example,

```
1  def test__init__(self):
2
3      def func(x, y):
4          return [-y[0]]
5      x_min = 0
6      x_max = 1
7      initial_value = [1]
8      mesh_points = 10
9
10     problem = solver.OneStepMethods(
11         func, x_min, x_max, initial_value, mesh_points)
12
13     # Test initialisation
14     self.assertEqual(problem.x_min, 0)
15     self.assertEqual(problem.mesh_points, 10)
16
17     # Test raised error for callable function
18     with self.assertRaises(TypeError):
19         solver.OneStepMethods(
20             x_min, x_min, x_max, initial_value, mesh_points)
21
22     # Test raised error if initial_value not list
23     with self.assertRaises(TypeError):
24         solver.OneStepMethods(
25             func, x_min, x_max, 1, mesh_points)
```

Testing initialisation of problem

A simple model with known solution is first initialised. Required inputs were checked to make sure the problem is set up properly, in line 14 and 15 of the code snippet above. To ensure that the inputs to the function are of the desired data type, errors are raised whenever the user inputs a wrong data type. The unit testing also tests that these errors are raised appropriately (line 17 to 25) whenever the data type does not satisfy the requirements.

## 3.5 Testing function

After making sure the problem is properly initialised, we then test that the numerical methods are working correctly. Take the adaptive method, BS23 algorithm, as an example, the testing of a method is as follows:

```
def test_ode23(self):

    def func(x, y):
        return [-y[0]]
    x_min = 0
    x_max = 1
    initial_value = [1]

    problem = solver.AdaptiveMethod(
        func, x_min, x_max, initial_value, initial_mesh=0.5)
    mesh, soln = problem.ode23()

    # Test end point of mesh
    self.assertGreaterEqual(mesh[-1], 1.0)

    # Test mesh point
    self.assertAlmostEqual(mesh[1], 0.3483788976565)

    # Test solution at first stepsize
    self.assertAlmostEqual(soln[1][0], 0.7052580305097)
```

Testing execution of method

The method is first tested that it executes the computations up to the maximum mesh value indicated. Then, it is checked that the first adaptive mesh point and its solution, obtained from the software, matches the value computed manually.

## 3.6 Conclusion

The software can solve any initial value problem of the general form Eqs. (2.7)-(2.8) using the implemented numerical methods as listed in Section 3.2. The software can also handle initial value problems of higher dimension. An example of the use of the software can be found at the link:
*Example use of software*. All methods are tested, not only making sure all lines are tested, but also testing their functionalities. Moreover, the results from all notebooks (see Appendix A) behaves as expected by theory. The

time efficiency of the software is not a major concern of the project, thus some methods might take long periods of time to run.

The purpose of setting up a software with version control, documentation and testing is to create a robust and reusable software. These software engineering methods can give confidence in the correctness of the code and aid in the future maintenance of the code. By implementing these methods to this software, it will allow me to gain experience in developing the software engineering techniques that I will use throughout my D.Phil..

# Chapter 4

# Fitzhugh-Nagumo Model Example

## 4.1 Background

We will look into Fitzhugh-Nagumo model as an application of the Python package. The Fitzhugh-Nagumo model describes an excitable system, such as the action potential of cardiac cells. The action potential was first described by Hodgkin and Huxley. Their model were then simplified to the Fitzhugh-Nagumo model, retaining the fast-slow phase and the excitability of the Hodgkin & Huxley model. [3]

This model is relevant to my D.Phil. project as my project will be related to action potential of cardiac muscle cells, an excitable system. I will be working on sodium ion channels of cardiac muscle cells, studying the effect of the flow of sodium ions across the cell membrane on the cell's action potential. Moreover, Fitzhugh-Nagumo model captures all the important features of an action potential, the excitability and the fast-slow phase. Therefore, it is a good simple model to start with.

## 4.2 Fitzhugh-Nagumo model

The definition of Fitzhugh-Nagumo model is

$$\epsilon \frac{dv}{dt} = f(v) - w + I_{app} \tag{4.1}$$

$$\frac{dw}{dt} = v - \gamma w \tag{4.2}$$

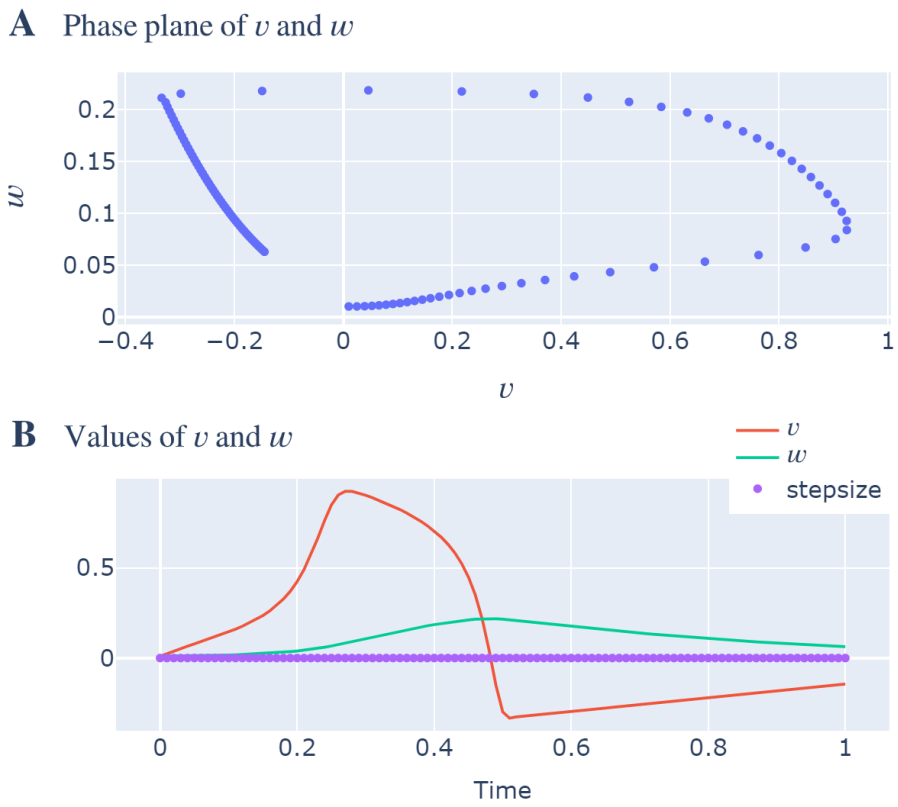


Figure 4.1: $(A)$ Phase plane of $v$ and $w$ by Euler's explicit method. $(B)$ Graph of $v$ and $w$ against time by Euler's explicit method.

where $f(v) = v(1-v)(v-\alpha)$, $0 < \alpha < 1$, $\epsilon \ll 1$, $I_{app}$ is the applied current and $t$ is time. The fast $v$ is the excitation variable, while the slow $w$ is the recovery variable.

In this implementation, the parameters are chosen to be $\alpha = 0.1$, $\gamma = 0.5$, $\epsilon = 0.01$ and $I_{app} = 0.026$, taking reference from [2]. The initial values are taken to be near the origin, which are $(v_0, w_0) = (0.01, 0.01)$. The model is solved for time $t$ from 0 to 1.

Fitzhugh-Nagumo model is solved with the various numerical methods implemented in the software that I have developed. The solutions of the model are in a notebook at the link: *Fitzhugh-Nagumo model notebook*.

From Figure 4.1 B, we can see that $v$, the excitation variable is excited in the early stage. While $v$ increases significantly, the change in $w$ is small.

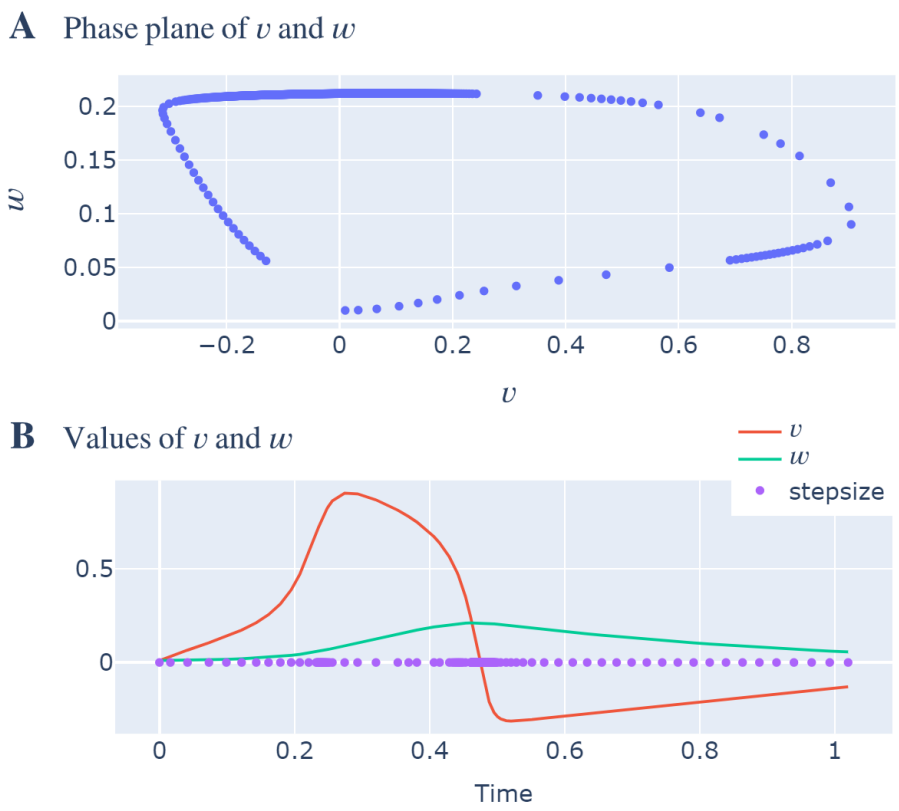


Figure 4.2: $(A)$ Phase plane of $v$ and $w$ for adaptive method BS23. $(B)$ Graph of $v$ and $w$ against time for adaptive method BS23

After $v$ reaches its peak and starts to reduce, $w$ increases slowly. This can be observed in both the phase plane (Figure 4.1 A) and the variable graph (Figure 4.1 B). When the variable $v$ starts to recover to its original value, $w$ is at its maximum. The scattering of points in the phase plane captures the feature of the model, where the change in $v$ is rapid while the change in $w$ is slow. When the change in $v$ is significantly larger than the change in $w$, the points are sparse. On the other hand, the points are packed when $w$ increases or decreases more than $v$. However, in the adaptive methods, such insights cannot be interpreted directly from the phase plane. Therefore, green triangles were plotted in Figure 4.2 B to indicate the adapted mesh points. The mesh points are adapted towards large change in $v$ or $w$ over a short period of time.

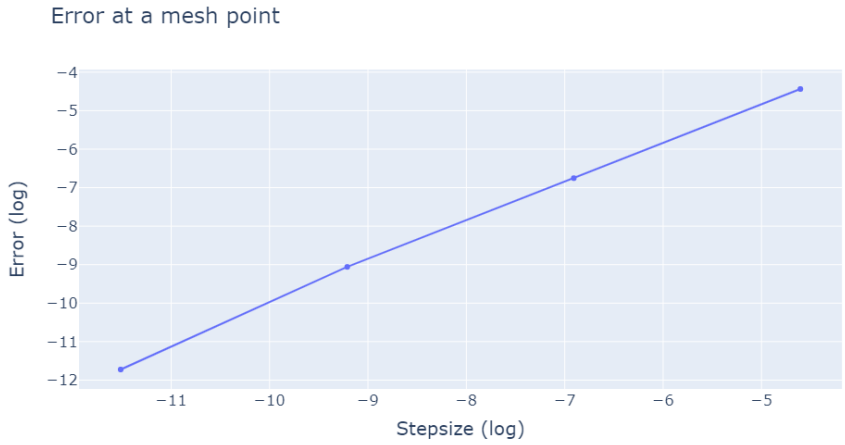


Figure 4.3: Error at a mesh point for Euler's explicit method. The Fitzhugh-Nagumo model Eqs. (4.1)-(4.2) is solved with Euler's explicit method at different step sizes. The error is the absolute difference between reference solution and numerical solution at $x = 0.7$. The reference solution is taken at step size of $1^{-7}$. Note that $\log(1^{-7}) \simeq -16.118$.

## 4.3 Convergence of Fitzhugh-Nagumo model

A notebook (accessible from the link: *Fitzhugh-Nagumo convergence notebook*) is created to test the convergence of the solution of the Fitzhugh-Nagumo model. Since the model has no analytical solution, it is tested against a reference solution. The reference solutions are assumed to be sufficiently accurate. For methods with fixed step size, which are the one-step methods and predictor-corrector method, the reference solutions are constructed by using a much smaller step size of $1^{-7}$, as compared to $1^{-5}$, the smallest step size for other numerical solutions. For methods with adaptive step size, reference solutions are obtained by using a much smaller tolerance value of $1^{-8}$, as compared to $1^{-5}$, the smallest tolerance value for other numerical solutions. This notebook shows the numerical solution computed for different methods at different step sizes or tolerance values. The numerical solutions are then compared with their respective reference solutions. In both methods, the error decreases as the step size or the tolerance value decreases, as shown in Figure 4.3 and Figure 4.4.

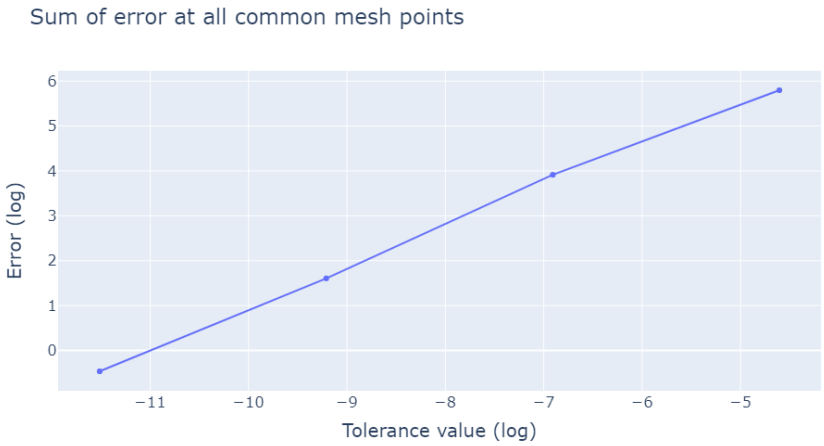


Figure 4.4: Sum of error for adaptive method BS23. The Fitzhugh-Nagumo model Eqs. (4.1)-(4.2) is solved with BS23 method at different absolute tolerance values. The relative tolerance is fixed at $1^{-15}$. The error is the sum of absolute difference between reference solution and numerical solution at all mesh points. The reference solution is taken at relative tolerance of $1^{-8}$. Note that $\log(1^{-8}) \simeq -18.421$.

# Chapter 5

# Conclusion

One-step methods including Euler's explicit method, Euler's implicit method, the trapezium rule method and the four-stage Runge-Kutta method were implemented in the software. Other than one-step methods, a predictor-corrector method and two adaptive methods were implemented. The example notebooks provided in Appendix A show that the numerical methods converge for the test problem Eqs. (2.15)-(2.16). Moreover, the orders of accuracy of the one-step methods computed from the test problem Eqs. (2.15)-(2.16) are in line with their respective theoretical order of accuracy. While there are no orders of accuracy for the adaptive methods, the accuracy of the method increases with the decrease in tolerance values. These also show that the implemented methods are convergent.

A robust software of ODE solvers is constructed, which includes documentation, version control and testing. The details of the package are documented and available at one of the links in Appendix A. The version control tracks all changes made to the code. The unit testing of the software achieved 100% code coverage. It tests the initialisation of problem and functionality of methods. These features will allow easier re-use and maintenance of the code, and provide confidence in the correctness of the code.

The software is applied to Fitzhugh-Nagumo model, a model of excitable system. Solutions from various numerical methods exhibit the fast-slow phase and the excitability property of the model. A convergence analysis shows that the numerical solutions are convergent. The adaptive method

adapts the mesh points according to the changes in the variable. Moreover, the smaller the tolerance value, the higher the accuracy for adaptive methods.

From this course, I learned about the numerical methods of solving ODEs, especially the theory behind the methods and their orders of accuracy. It provides a basis to choose appropriate methods while solving ODE systems. This knowledge will also give me insight when problems arise in numerical solutions of ODE system. Moreover, while implementing the numerical methods, I realised the difference and difficulty in actual implementation, such as usage of fixed point iteration in solving implicit methods and computation of truncation error. Additionally, the reading on Fitzhugh-Nagumo model helps me in the background knowledge of my D.Phil. project. I understood the features and properties of Fitzhugh-Nagumo model and excitable systems in general. Finally, having experience in creating a software is really helpful in practicing the software engineering techniques for my D.Phil. project.

# Appendix A

# Links

## Codes

Github repository:
https://github.com/FarmHJ/numerical-solver

Software documentation:
https://numerical-solver.readthedocs.io/en/latest/

Numerical methods implementation:
https://github.com/FarmHJ/numerical-solver/blob/main/solver/methods.py

## Notebooks

Solving test problem Eqs. (2.15)-(2.16) with various numerical methods:
https://nbviewer.jupyter.org/github/FarmHJ/numerical-solver/blob/main/examples/solver_convergence.ipynb

Truncation error of one-step methods:
https://nbviewer.jupyter.org/github/FarmHJ/numerical-solver/blob/main/examples/Onestep_methods_convergence.ipynb

Convergence of numerical methods on test problem Eqs. (2.15)-(2.16):
https://nbviewer.jupyter.org/github/FarmHJ/numerical-solver/blob/main/examples/solver_convergence_comparison.ipynb

Solving Fitzhugh-Nagumo model with various numerical methods:
https://nbviewer.jupyter.org/github/FarmHJ/numerical-solver/blob/main/examples/fitzhugh_nagumo.ipynb

Convergence of numerical solutions of Fitzhugh-Nagumo model:
https://nbviewer.jupyter.org/github/FarmHJ/numerical-solver/blob/main/examples/fhn_model_convergence.ipynb

# References


[1] Calc Workshop. How to do euler's method? simply explained in 3 powerful examples. `https://calcworkshop.com/first-order-differential-equations/eulers-method-table/`, 2019. Accessed: 2021-04-03.

[2] M Chapwanya, O. A Jejeniwa, A. R Appadu, and J. M.-S Lubuma. An explicit nonstandard finite difference scheme for the fitzhugh-nagumo equations. *International journal of computer mathematics*, 96:1993–2009, 2018.

[3] J. P Keener and J. Sneyd. *Mathematical physiology*. Springer, 2nd ed. edition, 2009.

[4] E. Süli. Numerical solution for differential equations 1. lecture notes for Numerical Solution for Differential Equations I, 2020.