An-Najah National University

Faculty of Graduate Studies

Error Analysis and Stability of Numerical Schemes for Initial Value problems "IVP's"

By

Imad Omar Faris Kayid

Supervised

Prof. Naji Qatanani

This Thesis is Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Computational Mathematics, Faculty of Graduate Studies, An-Najah National University, Nablus, Palestine.

2013

Error Analysis and Stability of Numerical Schemes for Initial Value problems "IVP's"

By

Imad Omar Faris Kayid

This thesis was defended successfully on 7 / 2 / 2013 and approved by:

Defense Committee Members		**Signature**
1-Prof. Dr. Naji Qatanani	**(Supervisor)**	N. QATANANI
2-Dr. Yousef Zahaykah	**(External Examiner)**	.....
3-Dr. Subhi Ruzieh	**(Internal Examiner)**	.....

Dedication

I would like to dedicate this thesis to my parents, to my wife Raifa, who supported me each step of the way and to my children Yara, Kareem and Noor.

Acknowledgment

First of all, I would thank my supervisor Prof. Dr. Naji Qatanani for his efforts and important guidance for the completion of this thesis.

الاقرار

أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:

Error Analysis and Stability of Numerical Schemes for Initial

Value problems "IVP's"

تحليل الأخطاء والثباتية للطرق العددية لحل مسائل القيم الابتدائية

أقر بأن ما اشتملت عليه هذه الرسالة إنما هي نتاج جهدي الخاص، باستثناء ما تمت الاشارة اليه حيثما ورد، وأن هذه الرسالة ككل، أو أي جزء منها لم يقدم من قبل لنيل أية درة علمية أو بحث لدى أية مؤسسة تعليمية أو بحثية أخرى.

Declaration

The work provided in this thesis, unless otherwise references, is the researcher's own work, and has not been submitted elsewhere for any other degree or Qualification.

Student's name: **اسم الطالب:**

Signature: **التـوقيـــع:**

Date: **التـاريـــخ:**

Table of Content

Table of Tables

Table of Figures

Error Analysis and Stability of Numerical Schemes

for Initial Value problems "IVP's"

By

Imad Omar Faris Kayid

Supervisor

Prof. Dr. Naji Qatanani

Abstract

Most of initial value problems are natural phenomena written in the language of mathematics. Solving these initial value problems is one of the most challenging fields in mathematics, because of the mathematicians' continuous desire of exactness.

This work focuses mainly on developing algorithms and programs to construct higher order Taylor's methods for approximating the solution of first order initial value problems, systems of first order initial value problems and higher order initial value problems. Moreover, it concentrates on studying error and stability of numerical methods for solving initial value problems. For this purpose, we developed programs to find the error amplification functions of Taylor's and Runge-Kutta methods and to plot boundaries of stability regions for these methods and other methods.

We concluded that with the programs we developed, higher order Taylor's methods could be a good choice for approximating solutions of a wide range

of initial value problems.

Introduction

Many problems in engineering and science can be formulated in terms of differential equations. Many mathematicians have studied the nature of these equations and many complicated systems can be described exactly with compact mathematical expressions.

The techniques for solving differential equations based on numerical approximations were developed before programmable computers existed. The problem of solving ordinary differential equations is classified into initial value and boundary value problems, depending on the conditions specified at the end points of the domain.

There are numerous methods that produce numerical approximations to the solution of initial value problems in ordinary differential equations such as Euler's method which was the oldest and simplest method originated by Leonard Euler in 1768. He was the first who suggested the idea to propagate the solution of an initial value problem by a sequence of small time-steps. In each step, the rate of change of the solution is treated as constant and is found from the formula for the derivative evaluated at the beginning of the step [5]. An improved Euler's method and Runge-Kutta methods described by Carl Runge and Martin Kutta in 1895 and 1905 respectively. The paper by Runge is now recognized as the starting point for modern one-step methods [6].

The 1883 paper of Bashforth and Adams [1] and the 1926 paper of Moulton [14] were the foundation blocks of developing multistep methods. Through their work, the explicit Adams-Bashforth methods, the implicit Adams-Moulton methods and the predictor-corrector methods were established. Milne also contributed in this field by the methods called after him and by the so-called Milne's device, which estimates error in predictor-corrector methods [13].

Numerical methods form an important part of solving initial value problems in ordinary differential equations most especially in cases where there is no closed form solution.

This thesis is organized as follows:

Chapter one gives perspective study of differential equations in particular, initial value problems, systems of first order initial value problems and higher order initial value problems.

In chapter two we introduce some numerical examples for solving initial value problems. These include single step methods and multistep methods.

Chapter three deals with higher order Taylor methods for solving first order IVP's, systems of first order IVP's and higher order IVP's.

In chapter four some error analysis for the numerical methods: Taylor method, Runge-Kutta method, Adams-Bashforth method, Adams-Moulton method, predictor-corrector method and Milne's method will be investigated and presented through same numerical examples.

Chapter One

Initial Value Problem

1.1 Preliminaries

A differential equation: is an equation relating some function f to one or more of its derivatives.

The equation

$$\frac{dy}{dt} = y(t) - t^2 + 3$$

is a differential equation.

The order of a differential equation is the order of the highest derivative that appears in the equation.

The equation

$$y'' - 2y' + y = te^t - t$$

is of order 2.

The degree of an ordinary differential equation: is the greatest number of times the dependent variable appears in any single term. For example, the degree of $y' + (y'')^2 y + 1 = 0$ is 3, whereas the degree of $y''y'y^2 + x^5 y = 1$ is 4.

An ordinary differential equation (ODE): is an equation with the derivatives of a function of one variable.

The general form of an explicit *k*th order ordinary differential equation is given by

$$y^{(k)} = f\left(t, y, y', \ldots, y^{(k-1)}\right) \qquad (1.1)$$

where y, y′, . . . , $y^{(k-1)}$ are functions of t.

The general solution of this equation contains *k* arbitrary constants $c_1, c_2, \ldots, c_k$. These constants can be found by prescribing *k* conditions.

A partial differential equation (PDE) describes a relation between an unknown function and its partial derivatives.

The heat equation

$$\frac{\partial u}{\partial t} - \alpha^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = 0$$

is a second order PDE.

Initial value problem (IVP): is a problem that specifies the initial conditions at the same value of t.

As an example

$$y'' = y + 2e^t, \qquad y(0) = 0, \qquad y'(0) = 1$$

is an initial value problem since both conditions imposed on $t = 0$.

Boundary value problem: is a problem that specifies the boundary conditions at different values of $\boldsymbol{t}$.

As an example

$$y'' = y + 2e^t, \qquad y(0) = 0, \qquad y'(1) = e.$$

is a boundary value problem because the two conditions are specified at different values of *t*.

1.2 Initial Value Problem (IVP)

A first order initial value problem defined on the interval [a, b] can be written as

$$\frac{dy}{dt} = f\big(t, y(t)\big) \qquad a \le t \le b, \qquad y(a) = \alpha \tag{1.2}$$

Since any ordinary differential equation of order *k*

$$y^{(k)} = f(t, y, y', \ldots, y^{(k-1)})$$

can always be transformed into a system of *k* first−order equations, we will focus on solving first order initial value problems and systems of first order initial value problems.

1.3 Existence and Uniqueness of the Solution

Solving differential equations can be done by two major ways. The first way is to find the exact solution analytically. The other way is to approximate the solution by numerical methods at usually equally spaced points, then interpolate the solution to the whole interval of interest. Since most differential equations that represent real nature phenomena cannot be solved analytically, we will focus on solving initial value problems using numerical approximations.

Before we discuss methods for approximating the solution to our basic problem (1.2), we must consider some theory to ensure that our problem has a unique solution and know how small errors on the initial condition can affect the accuracy of the approximated solution.

Definition (1.1)

A function $f(t,y)$ is said to satisfy a Lipschitz condition in the variable *y* on a set $D \subset \mathbb{R}^2$ if a constant $L > 0$ exists with

$$|f(t,y_1) - f(t,y_2)| \leq L|y_1 - y_2|,$$

whenever (t,y_1) and (t,y_2) are in *D*. The constant *L* is called a Lipschitz constant for f [3].

Example (1.1) We can show that $f(t,y) = ty^2$ satisfies a Lipschitz condition on the set $D = \{(t,y): 1 \leq t \leq 2 \; and \; 1 \leq y \leq 3\}$.

For each pair of points (t,y_1) and (t,y_2) in D we have

$$|f(t,y_1) - f(t,y_2)| = |ty_1^2 - ty_2^2| = |t||y_1^2 - y_2^2|$$

$$= |t||y_1 + y_2||y_1 - y_2|$$

$$\leq 2 \times 6 \times |y_1 - y_2| = 12|y_1 - y_2|.$$

Thus, f satisfies a Lipschitz condition on D in the variable *y* with a Lipschitz constant $L = 12$.

Definition (1.2) [16]

A set $D \subset \mathbb{R}^n$ is said to be convex if, whenever $\boldsymbol{x}$ and $\boldsymbol{y}$ belong to *D*, also

$\theta \boldsymbol{x} + (1-\theta)\boldsymbol{y} \in D \quad \forall \, \theta \in [0,1]$.

Thus, a set $D \subset \mathbb{R}^2$ is convex if $(t_1,y_1),(t_2,y_2) \in D$ then the point

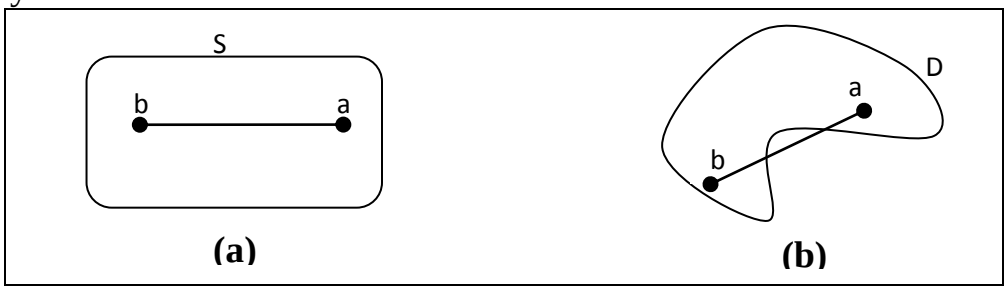
$$\lambda(t_1,y_1) + (1-\lambda)(t_2,y_2) \in D \; with \; \lambda \in [0,1] \, .$$

Since a line segment between (t_1,y_1) and (t_2,y_2) is the set of all points

$(t,y) = \lambda(t_1,y_1) + (1-\lambda)(t_2,y_2) \quad \lambda \in [0,1]$,

then a set *D* in $\mathbb{R}^2$ is called a convex set if the line segment joining any pair of points of *D* lies entirely in *S*.

The set S in Figure (1.1) (a) is convex because the line segment joining any pair of points of S lies entirely in S, while the set D in Figure (1.1) (b) is non-convex, since points a and b lie in D, but the line segment $\overline{ab}$ does not lie entirely in D.



Figure(1.1): (a) Convex Set S. (b) Non-convex Set D.

Example (1.2)

We can show that the set $D = \{(t, y): a \leq t \leq b \text{ and } -\infty \leq y \leq \infty\}$ is convex.

Let$(t_1, y_1), (t_2, y_2) \in D$.

So, $a \leq t_1 \leq b \text{ and } a \leq t_2 \leq b$.

For $\lambda \in [0,1]$

$(1-\lambda)a \leq (1-\lambda)t_1 \leq (1-\lambda)b$ and $\lambda a \leq \lambda t_2 \leq \lambda b$.

These two inequalities give

$a \leq (1-\lambda)t_1 + \lambda t_2 \leq b$.

It is obvious that

$-\infty \leq (1-\lambda)y_1 + \lambda y_2 \leq \infty$.

Therefore, D is a convex set.

Theorem (1.1) [3]

Suppose $f(t,y)$ is defined on a convex set $D \subset \mathbb{R}^2$. If a constant $L > 0$ exists

with

$$\left|\frac{\partial f}{\partial y}(t,y)\right| \leq L, \quad \text{for all } (t,y) \in D,$$

then f satisfies a Lipschitz condition on D in the variable y with a Lipschitz constant L.

Proof: Holding t constant and applying the Mean Value Theorem to the function $f(t,y)$, when $y_1 < y_2$, a number ξ in (y_1, y_2) exists with

$$\frac{f(t,y_2) - f(t,y_1)}{y_2 - y_1} = \frac{\partial f}{\partial y}(t,\xi),$$

$$|f(t,y_2) - f(t,y_1)| = |y_2 - y_1| \left|\frac{\partial f}{\partial y}(t,\xi)\right| \leq |y_2 - y_1| L.$$

Thus, f satisfies a Lipschitz condition on D in the variable y with a Lipschitz constant L.

Example (1.3)

For the set $D = \{(t,y): 1 \leq t \leq 2 \ \ and \ -\infty \leq y \leq \infty\}$ it easy to show, that $f(t,y) = ty$ satisfies a Lipschitz condition in the variable y.

We have shown in Example (1.2) that D is convex. In addition

$$\left|\frac{\partial f}{\partial y}(t,y)\right| = |t| \leq 2.$$

Thus f satisfies a Lipschitz condition on D in the variable y with a Lipschitz constant 2.

Definition (1.3) [3]

The initial value problem

$$\frac{dy}{dt} = f\big(t, y(t)\big), \quad a \le t \le b, \quad y(a) = \alpha,$$

is said to be a well-posed problem if:

- A unique solution, $y(t)$ to the problem exists, and
- There exist constants $\varepsilon_0 > 0$ and $k > 0$ such that for any ε, with $\varepsilon_0 > \varepsilon > 0$, whenever $\delta(t)$ is continuous with $|\delta(t)| < \varepsilon$ for all t in [a, b], and when $|\delta_0| < \varepsilon$ the initial-value problem

$$\frac{dz}{dt} = f(t, z) + \delta(t), \quad a \le t \le b, \qquad z(a) = \alpha + \delta_0.$$

has a unique solution z(t) that satisfies

$$|z(t) - y(t)| < k\varepsilon \text{ for all } t \text{ in } [a, b].$$

Point two in this definition says that small perturbations of the original problem and small perturbations of the initial condition have only small error effects on the approximated solution. To illustrate this we give the following example.

Example (1.4)

Using definition (1.3), we can show that the initial value problem

$$y'(t) = y(t) + 1, \quad 0 \le t \le 1,, \quad y(0) = 0$$

is well-posed.

Consider the perturbed problem with constant δ and δ_0

$$z'(t) = z(t) + 1 + \delta, \qquad 0 \le t \le 1, \qquad z(0) = \delta_0.$$

$$y(t) = e^t - 1 \quad \text{and} \quad z(t) = (\delta + \delta_0 + 1)e^t - (1 + \delta).$$

Suppose that $\varepsilon > 0$. And if $|\delta| < \varepsilon$ and $|\delta_0| < \varepsilon$, then

$$|z(t) - y(t)| = |(\delta + \delta_0)e^t - \delta|$$

$$\leq |(\delta + \delta_0)e^t| + |\delta| \leq 2\varepsilon e^1 + \varepsilon = (2e + 1)\varepsilon,$$

for all t . Therefore, the problem is well-posed with $k = (2e + 1)$ and for all $\varepsilon > 0$.

The next theorem tells us sufficient conditions that guarantee well posedness of our basic problem.

Theorem (1.2) [3]

Suppose that $D = \{(t, y): a \leq t \leq b \ and \ -\infty < y < \infty\}$. If $f(t, y)$ is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial value problem

$$y'(t) = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha,$$

is well-posed.

We note, that this theorem ensures the uniqueness of the solution $y(t)$ for $a \leq t \leq b$.

Example (1.5)

We can show that the IVP

$$y' = t^2 y + 1, \qquad 0 \leq t \leq 2, \qquad y(0) = 1,$$

is well-posed.

The function $f(t, y) = t^2 y + 1$ is continuous. We have shown in Example (1.2) that the set $D = \{(t, y): 0 \leq t \leq 2 \ and \ -\infty < y < \infty\}$ is convex. In addition, we have

$$\left|\frac{\partial f}{\partial y}(t, y)\right| = |t^2| \leq 4.$$

Thus f satisfies a Lipschitz condition on D in the variable y with Lipschitz constant 4. Therefore, according to theorem (1.2) the IVP is well-posed.

Theorem (1.3) [4]

Suppose that f and f_y, its first partial derivative with respect to *y*, are continuous for *t* in [*a*, *b*] and for all *y*. Then the initial value problem

$$\frac{dy}{dt} = f\big(t, y(t)\big), \qquad a \le t \le b \quad \text{and} \quad y(a) = \alpha,$$

has a unique solution *y*(*t*) for $a \le t \le b$, and the problem is well-posed.

Proof: The set $D = \{(t, y): a \le t \le b \ \ and \ -\infty < y < \infty\}$ is convex. Since f_y is continuous on [a, b], then f_y is bounded. Therefore, there exists a real number $L > 0$ such that $|f_y| \le L$. Thus, f satisfies a Lipschitz condition on D in the variable y. Also f is continuous. It follows from theorem (1.2), that the IVP is well-posed.

Example (1.6)

The initial-value problem

$$y' = (t - 0.5t^2) + y, \text{ for } \ 0 \le t \le 1 \text{ and } y(0) = 1$$

is a well-posed initial value problem, since the functions

$$f(t, y) = (t - 0.5t^2) + y, \ and \ \ f_y(t, y) = 1$$

are both continuous for $0 \le t \le 2$ and for all y.

1.4 Systems of First Order Initial Value Problems

A *k*th-order system of first order initial value problems has the general form

$$u_1' = f_1(t, u_1, u_2, \dots, u_k)$$

$$u_2' = f_2(t, u_1, u_2, \dots, u_k)$$

⋮

$u_k' = f_k(t, u_1, u_2, \ldots, u_k)$

for $a \le t \le b$, with the initial conditions

$$u_1(a) = \alpha_1,\ u_2(a) = \alpha_2,\ \ldots,\ u_k(a) = \alpha_k. \qquad (1.3)$$

To solve this system we must find $u_1, u_2, \ldots, u_k$ that satisfy all the differential equations and the initial conditions.

Consider the following system of IVP's

$$u_1'(t) = u_2(t)$$

$$u_2'(t) = 2u_1(t) + u_2(t)$$

together with the initial conditions

$$u_1(0) = 8 \text{ and } u_2(0) = 4.$$

We can show that

$$u_1(t) = 4e^{2t} + 4e^{-t}, \quad u_2(t) = 8e^{2t} + 4e^{-t}$$

is the solution of this system.

Solving this system numerically will be discussed in next chapter.

1.5 Higher Order Initial Value Problem

The first step to solve an IVP of order k

$$y^{(k)} = f(t, y, y', y'', \ldots, y^{(k-1)}), \qquad a \le t \le b$$

$$y(a) = \alpha_1,\ y'(a) = \alpha_2,\ y''(a) = \alpha_3, \ldots,\ y^{(k-1)}(a) = \alpha_k. \qquad (1.4)$$

is to transform it into a system of first order IVP as follows:

Let

$$u_1 = y,$$

$u_2 = y'$,

$u_3 = y''$,

$\vdots$

$u_k = y^{(k-1)}$. (1.5)

Differentiating these equations, we get

$u_1' = y'$,

$u_2' = y''$,

$u_3' = y'''$,

$\vdots$

$u_k' = y^{(k)} = f(t, y, y', y'', \ldots, y^{(k-1)})$ (1.6)

Substituting we get a system of first order IVP

$u_1' = u_2$

$u_2' = u_3$

$\vdots$

$u_{k-1}' = u_k$

$u_k' = f(t, u_1, u_2, \ldots, u_k)$.

$u_1(a) = y(a) = \alpha_1$, $u_2(a) = y'(a) = \alpha_2$, …, $u_k(a) = y^{(k-1)}(a) = \alpha_n$. (1.7)

The second step is to solve this system for $u_1, \ldots, u_k$ satisfying (1.6). Finally, the solution of u_1 is assigned to y because $y = u_1$.

To illustrate this method, consider the following example.

Example (1.7)

We will show how to transform the following second order IVP into a

two-dimensional system of first order IVP's:

$$y'' - 2y' + y = te^t - t, \qquad 0 \le t \le 1,$$

with initial conditions

$$y(0) = 0, \quad y'(0) = 0.$$

First, we rewrite the DE as

$$y'' = 2y' - y + te^t - t.$$

Now let

$$u_1 = y,$$

$$u_2 = y'.$$

Differentiating both equations with respect to t, we get

$$u_1' = y',$$

$$u_2' = y''.$$

Substituting, we get a system of first order IVP's

$$u_1' = u_2,$$

$$u_2' = 2u_2 - u_1 + te^t - t$$

with initial conditions

$u_1(0) = y(0) = 0$, $u_2(0) = y'(0) = 0$.

Solving systems of IVP's numerically will be discussed next chapter.

Chapter Two

Numerical Methods for Initial Value Problems

2.1 Introduction

In this chapter, we shall study some numerical methods for solving IVP's of the form

$$y'(t) = f\big(t, y(t)\big), \qquad a \le t \le b, \qquad y(a) = \alpha,$$

that possess a unique solution on some specified interval, $t \in [a, b]$. In these numerical methods, we will find approximations to the solution of the initial value problem at N particular equally spaced points

$$t_{i+1} = t_i + h, \quad i = 1, \dots, N, \quad N = \left\lfloor \frac{b-a}{h} \right\rfloor, \quad t_1 = a, \; y(t_1) = \alpha,$$

approximations to the numbers $y(t_2), \dots, y(t_{N+1})$, rather than to the curve of y(t).

Methods for approximating the solution of initial value problems can be classified mainly into two types. They are

(i) Single step methods,

(ii) Multistep methods.

Both of these methods can be either implicit or explicit. If the approximate solution w_{i+1} depends only on the previous $w_j, j = 1, \dots, i$, then the method is explicit. However, if w_{i+1} depends on w_{i+1} too, then the method is implicit, that is, we get an algebraic equation for the solution of w_{i+1}.

2.2 Single Step methods

In single step methods, the solution at any point t_{i+1} is obtained by using the solution at only the previous point t_i. Thus, we can write a general implicit single step method as

$$w_{i+1} = w_i + h\,\varphi(t_{i+1}, w_{i+1}, t_i, w_i, h),$$

and a general explicit single step method as

$$w_{i+1} = w_i + h\,\varphi(t_i, w_i, h), \qquad (2.1)$$

where φ is a function of the arguments t_i, w_i, h and depends on $f(t, y(t))$ of the given differential equation. The function φ is called the increment function, see [10].

Definition (2.1) (Local Truncation Error) [9]

The local truncation error τ_{i+1} of a method is defined to be the difference between the exact and the numerical solution of the IVP at time $t = t_{i+1}$:

$$\tau_{i+1} = y(t_{i+1}) - w_{i+1},$$

under the localizing assumption that $w_i = y(t_i)$, i.e. that the current numerical solution w_i is exact. If $\tau_{i+1} = O(h^{p+1})(p > 0)$, the method is said to be of order p.

If we calculate $\widehat{w}_{i+1}$ assuming $w_i = y_i$, that is:

$$\widehat{w}_{i+1} = y_i + h\,\varphi(t_i, y_i, h, f), \qquad (2.2)$$

then the LTE, τ_{i+1}, at t_{i+1} is defined by:

$$\tau_{i+1} = y_{i+1} - \widehat{w}_{i+1}$$

$$\tau_{i+1} = y_{i+1} - y_i - h\,\varphi(t_i, y_i, h, f). \qquad (2.3)$$

Definition (2.2)

The global truncation error**:** The difference

$e_i = y(t_i) - w_i$

is referred to as the global error (GE) at t = t_i.

Definition (2.3) [3]

A one-step difference-equation method with local truncation error $\tau_i(h)$ at the *i*th step is said to be **consistent** with the differential equation it approximates if

$\lim_{h \to 0} \max_{1 \le i \le N} |\tau_i(h)| = 0.$

Definition (2.4) [9]

A numerical method is said to converge to the solution $y(t)$ of a given IVP at $t = t^*$ if the GE $e_i = y(t_i) - w_i \; at \; t_i = t^* = t_0 + ih$ satisfies

$|e_i| \to 0$

as $h \to 0$. It converges at a pth-order rate if $e_i = O(h^p)$ for some $p > 0$.

A numerical method is said to be consistent of order p if

$\tau_{i+1} = O(h^{p+1})$ with $p > 0$.

Before we begin our discussion of the methods, we shall first derive the Taylor series expansion of $y(t_{i+1})$ with remainder.

Theorem (2.1) (Taylor Theorem) [2]

Suppose f has $n + 1$ continuous derivatives on an open interval containing a. Then for each x in the interval,

$$f(x) = \left[\sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!} (x-a)^k \right] + R_{n+1}(x),$$

where the error term $R_{n+1}(x)$ satisfies

$$R_{n+1}(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-a)^{n+1}$$

for some *c* between *a* and *x*.

Now for our Problem (1.2), if $y(t)$ is continuous and has $n+1$ continuous derivatives on an interval about $t = t_i$, then the Taylor series expansion of $y(t)$ about $t = t_i$ is

$$y(t) = \sum_{p=0}^{\infty} \left(\frac{y^{(p)}(t_i)}{p!}(t-t_i)^p \right) \qquad (2.4)$$

Now, for $t = t_{i+1}, t_{i+1} - t_i = h$, therefore (2.4) becomes

$$y_{i+1} = y_i + hy_i' + \frac{1}{2}h^2 y_i'' + \cdots + \frac{1}{n!}h^n y_i^{(n)} + R_{n+1}(t) \qquad (2.5)$$

$$y_{i+1} = y_i + h\left[y_i' + \frac{1}{2}hy_i'' + \cdots + \frac{1}{n!}h^{n-1} y_i^{(n)} \right] + R_{n+1}(t) \qquad (2.6)$$

$$R_{n+1}(t) = \frac{1}{(n+1)!}\, h^{n+1} y^{(n+1)}(c), \quad c \in (t_i, t_{i+1}) \qquad (2.7)$$

When $y'(t) = f(t, y(t))$, we can replace $y_i^{(j+1)}$ by $f_i^{(j)}$ in the previous equations for $j = 0, \ldots, n$.

We note that the expression in the square brackets in (2.6) represents the $(n-1)$ Taylor expansion of $f(t, y)$.

Finally, in (2.6) if we let

$$T^{(n)}(t_i, y_i, h) = \left[y_i' + \frac{1}{2}hy_i'' + \cdots + \frac{1}{n!}h^{n-1} y_i^{(n)} \right], \qquad (2.8)$$

then (2.6) becomes

$$y_{i+1} = y_i + hT^{(n)}(t_i, y_i, h) + R_{n+1}(t). \qquad (2.9)$$

2.2.1 Euler's Method

Euler was the first who suggested the idea to propagate the solution of an initial value problem forward by a sequence of small time-steps. In each step, the rate of change of the solution is treated as constant and is found from the formula for the derivative evaluated at the beginning of the step [5].

One way to derive Euler's method for approximating the solution of the first order IVP (1.2) is achieved by approximating $y_i' = f(t_i, y_i)$ as follows:

$$f(t_i, y_i) \approx \frac{y_{i+1} - y_i}{h}.$$

Solving for y_{i+1}, we get

$$y_{i+1} \approx y_i + hf(t_i, y_i).$$

Using this equation, we get Euler's forward method

$$w_{i+1} = w_i + hf(t_i, w_i). \qquad (2.10)$$

Now, if f is differentiable on (a, b) and f' is continuous on $[a, b]$, then there exists $c \in [t_i, t_{i+1}]$ such that

$$y_{i+1} = y_i + hf(t_i, y_i) + \frac{1}{2}h^2 f'(c, y(c)).$$

To find the LTE of Euler's method, we have $\widehat{w}$ as defined in (2.2)

$$\widehat{w}_{i+1} = y_i + h\, f(t_i, y_i),$$

therefore

$$\tau_{i+1} = y_{i+1} - y_i - h\, f(t_i, y_i),$$

$$\tau_{i+1} = \frac{1}{2}h^2 f'(c, y(c)).$$

Then, if a positive number M exists so that $|f'(t,y)| \leq M$ for all $t \in (a,b)$, then

$$|\tau_{i+1}| \leq \frac{1}{2} M h^2 \qquad \tau_{i+1} = O(h^2).$$

It follows from Definition (2.1) that Euler's method is of order $O(h)$.

We introduce now Algorithm (2.1) for approximating the solution of the problem (1.2) using Euler's method and comparing the approximated solution with the exact solution.

Algorithm (2.1): Euler's Method.

Step 1: $Define\ f(t,y),\ yExact(t)$

Step 2: $Input\ a, b, \alpha, h$

Step 3:

Step 4: $Let\ N = (b-a)/h$

Step 5: $Let\ w_1 = \alpha,\ t_1 = a,\ y_1 = \alpha$

$For\ i = 1\ to\ N$

$w_{i+1} = w_i + hf(t_i, w_i)$

Step 6: $y_{i+1} = yExact(t_{i+1})$

Step 7: $t_{i+1} = t_i + h$

Step 8: $Find\ \ GE = abs(y - w)$

$Output\ t, y, w, GE$

$Stop$

Example (2.1)

Consider Euler's method to approximate the solution of the following initial-value problem:

$$y' = \frac{y}{t} - \left(\frac{y}{t}\right)^2, 1 \le t \le 2, \quad y(1) = 1, \text{using } h = 0.1.$$

This initial value problem has the exact solution

$$y(t) = \frac{t}{log(t) + 1}.$$

Matlab Program (2.1)(See Appendix A) is an implementation of Algorithm (2.1) in computer language. The results of running this program for our Example (2.1) are represented in Table (2.1). In addition to the approximated solution of $y(t)$, at each step the program calculates the absolute values of the global error e (column 4) and the local truncation error τ (column 6). It also plots (Figure (2.1)) the approximated vector w (column 3) and the exact vector y (column 2) against the vector t (column 1).

Table (2.1): Results of Example (2.1) using Euler's method with h=0.1

t_i	y_i	w_i	$abs(y_i - w_i)$	$\widehat{w}_i$	$abs(\tau_i)$
1.0000000	1.0000000	1.0000000	0.0000000	1.0000000	0.0000000
1.1000000	1.0042817	1.0000000	0.0042817	1.0000000	0.0042817
1.2000000	1.0149523	1.0082645	0.0066879	1.0122262	0.0027261
1.3000000	1.0298137	1.0216895	0.0081242	1.0279950	0.0018187
1.4000000	1.0475339	1.0385147	0.0090192	1.0462776	0.0012562
1.5000000	1.0672624	1.0576682	0.0095942	1.0663717	0.0008907
1.6000000	1.0884327	1.0784611	0.0099716	1.0877888	0.0006439
1.7000000	1.1106551	1.1004322	0.0102229	1.1101830	0.0004721
1.8000000	1.1336536	1.1232621	0.0103915	1.1333041	0.0003494
1.9000000	1.1572284	1.1467236	0.0105048	1.1569686	0.0002599
2.0000000	1.1812322	1.1706516	0.0105806	1.1810389	0.0001934

We notice that the results are consistent with the theoretical error estimates for Euler's method, since the global error is of $O(h)$ and the local truncation error is of $O(h^2)$.

If we repeat solving Example (2.1), but using $h = 0.05$ instead of $h = 0.1$, we get better results. Table (2.2) shows that the global error at the last step

using h = 0.05 is 0.0051070, while it was 0.0105806 (Table 2.1) when we used $h = 0.1$.

In addition, we notice that

$$\frac{0.0051070}{0.0105806} = 0.4826758 \approx \left(\frac{0.05}{0.1}\right)^1 = 0.5,$$

which agrees with our theoretical analysis that Euler's method is of $O(h)$.

If we do the same for the local truncation error, we get

$$\frac{0.0000437}{0.0001934} = 0.2259565 \approx \left(\frac{0.05}{0.1}\right)^2 = 0.25$$

which agrees with our theoretical analysis that local truncation error for Euler's method is of $O(h^2)$.

Table (2.2): Results of Example (2.1) using Euler's method with h=0.05

t_i	y_i	w_i	$abs(y_i - w_i)$	$\hat{w}_i$	$abs(\tau_i)$
1.0000000	1.0000000	1.0000000	0.0000000	1.0000000	0.0000000
1.0500000	1.0011536	1.0000000	0.0011536	1.0000000	0.0011536
1.1000000	1.0042818	1.0022676	0.0020142	1.0033714	0.0009104
1.1500000	1.0089827	1.0063152	0.0026674	1.0082539	0.0007287
1.2000000	1.0149523	1.0117819	0.0031704	1.0143620	0.0005903
1.2500000	1.0219569	1.0183942	0.0035627	1.0214736	0.0004833
1.3000000	1.0298136	1.0259420	0.0038717	1.0294145	0.0003992
1.3500000	1.0383780	1.0342605	0.0041175	1.0380456	0.0003323
1.4000000	1.0475339	1.0432196	0.0043144	1.0472554	0.0002785
1.4500000	1.0571876	1.0527145	0.0044732	1.0569528	0.0002348
1.5000000	1.0672624	1.0626605	0.0046019	1.0670633	0.0001990
1.5500000	1.0776949	1.0729880	0.0047068	1.0775256	0.0001693
1.6000000	1.0884327	1.0836401	0.0047926	1.0882881	0.0001446
1.6500000	1.0994318	1.0945687	0.0048630	1.0993078	0.0001239
1.7000000	1.1106551	1.1057341	0.0049209	1.1105486	0.0001065
1.7500000	1.1220713	1.1171026	0.0049686	1.1219796	0.0000916
1.8000000	1.1336535	1.1286457	0.0050078	1.1335746	0.0000790
1.8500000	1.1453792	1.1403389	0.0050402	1.1453110	0.0000681
1.9000000	1.1572285	1.1521615	0.0050669	1.1571697	0.0000588
1.9500000	1.1691843	1.1640954	0.0050889	1.1691337	0.0000507
2.0000000	1.1812322	1.1761253	0.0051070	1.1811885	0.0000437

Figure (2.1) compares the approximated solutions with the exact solutions of Example (2.1). It is clear that Euler's method is not accurate for our choice of the step size $h = 0.1$ (Figure (2.1):(a)). To get better results, we have to make h smaller, and that means more computation time, or to find another more accurate method. Figure (2.1) (b) shows that the approximated solution gets closer to the exact solution when we use smaller values of h.

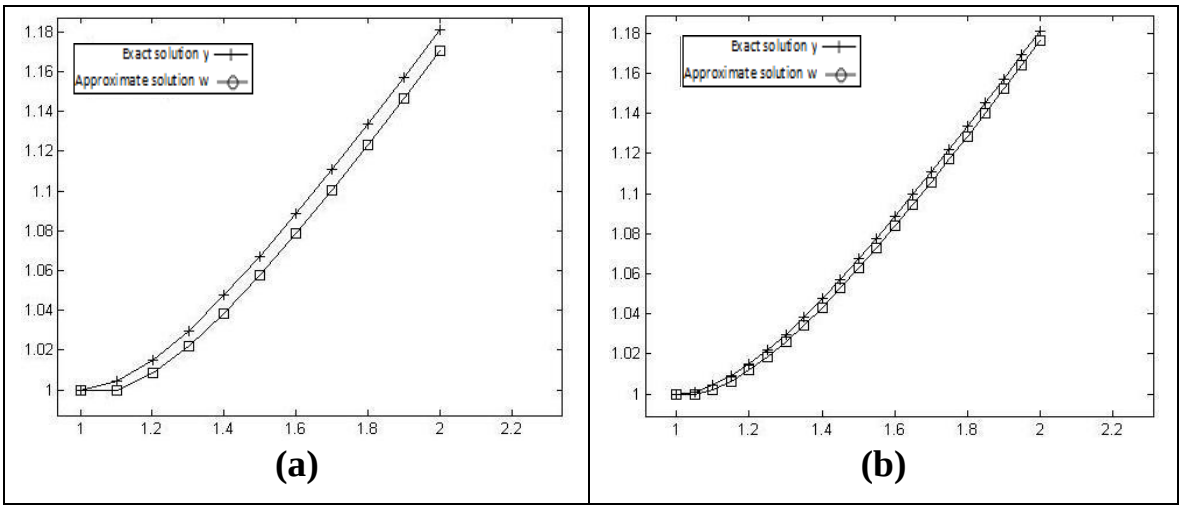


Figure (2.1) Comparison between the approximated solution and the exact solution of Example (2.1) (a): using h=0.1 (b) using h=0.05.

2.2.2 Runge–Kutta Methods

Runge-Kutta methods are based on the 1895 paper of C. Runge [15] and the 1901 paper of W. Kutta [11]. The paper by Runge is now recognized as the starting point for modern one-step methods [6].

The family of explicit Runge–Kutta methods is given by

$$w_{n+1} = w_n + \sum_{i=1}^{s} b_i k_i,$$

where

$$k_1 = hf(t_n, w_n),$$

$$k_2 = hf(t_n + c_2 h, w_n + a_{21}k_1),$$

$$k_3 = hf(t_n + c_3 h, w_n + a_{31}k_1 + a_{32}k_2),$$

$$\vdots$$

$$k_s = hf\left(t_n + c_s h, w_n + a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}\right)\ [7].$$

To specify a particular method, one needs to provide the integer *s* (the number of stages), and the coefficients $a_{ij}(for\ 1 \leq j < i \leq s)$, $b_i(for\ i = 1,2,\ldots,s)$ and $c_i(for\ i = 2,3,\ldots,s)$. The matrix $[a_{ij}]$ is called the *Runge–Kutta matrix*, while the b_i and c_i are known as the *weights* and the *nodes*. These data are usually arranged in a mnemonic device, known as a *Butcher tableau* (after John C. Butcher):

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$			$\ddots$		
c_s	a_{s1}	a_{s2}	$\cdots$	$a_{s,s-1}$	
	b_1	b_2	$\cdots$	b_{s-1}	b_s

Figure (2.2):Butcher tableau for explicit RK methods [7]

The Runge–Kutta method is consistent if (see [7])

$$\sum_{j=1}^{s} b_j = 1 \quad and \quad \sum_{j=1}^{i-1} a_{ij} = c_i\ for\ i = 2, \ldots, s\ .$$

Figures (2.3) contains some Butcher tableaus for Runge-Kutta explicit methods. We can use any of these tableaus in Algorithm (2.2) to generate the desired Runge-Kutta method.

	0	0	0	0

0	0
	1

(a)

0	0	0
1/2	1/2	0
	0	1

(b)

0	0	0
1	1	0
	1/2	1/2

(c)

1/2	1/2	0	0
1	-1	2	0
	1/6	2/3	1/6

(d)

Figure (2.3): Butcher tableaus for some RK explicit methods :
(a) One stage Euler's forward method. (b) Two stages midpoint method.
(c) Two stages Heun's method. (d) Three stages RK method [7]

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

Figure (2.3) (e) Butcher tableau for classical explicit RK4 method [7]

We will use Taylor expansion

$$y_{i+1} = \left(y_i + hy_i' + \frac{1}{2}h^2 y_i'' + \cdots + \frac{1}{6}h^3 y_i''' + \cdots\right)_{t_i}$$

$$= y_i + hf_i + \frac{h^2}{2}\left(f_t + ff_y\right)_{t_i}$$

$$+ \frac{h^3}{6}(f_{tt} + 2ff_{ty} + f^2 f_{yy} + f_t f_y + f f_y^2)_{t_i} + \cdots \qquad (2.11)$$

to study two stages Runge-Kutta methods:

$$y_{i+1} = y_i + b_1 k_1 + b_2 k_2 \qquad (2.12)$$

$$k_1 = hf_i$$

$$k_2 = hf(t_i + c_2 h, y_i + a_{21} hf_i)$$

$$= h[f_i + (c_2 hf_t + a_{21} hff_y)_{t_i}$$

$$+ \frac{1}{2}\{(c_2 h)^2 f_{tt} + 2c_2 h.a_{21} hff_{ty} + (a_{21} hf)^2 f_{yy}\}_{t_i} + \cdots]$$

$$= hf_i + h^2(c_2 f_t + a_{21} ff_y)_{t_i}$$

$$+ \frac{h^3}{2}\{(c_2)^2 f_{tt} + 2c_2 a_{21} ff_{ty} + a_{21}^2 f^2 f_{yy}\}_{t_i} + \cdots$$

Substituting the values of k_1 and k_2 in (2.12), we get

$$y_{i+1} = y_i + (b_1 + b_2)hf_i + h^2(b_2c_2f_x + b_2a_{21}ff_y)_{t_i}$$
$$+\frac{h^3}{2}b_2\{(c_2)^2f_{tt} + 2c_2a_{21}ff_{ty} + a_{21}^2f^2f_{yy}\}_{t_i} + \cdots \qquad (2.13)$$

Comparing the coefficients of h and h^2, we get

$$b_1 + b_2 = 1, \quad b_2c_2 = \frac{1}{2}, \quad b_2a_{21} = \frac{1}{2}.$$

Solving these equations, we get

$$a_{21} = c_2, \quad b_2 = \frac{1}{2c_2}, \quad b_1 = 1 - \frac{1}{2c_2}.$$

Using these values, we get the two stages Runge-Kutta methods

$$w_{i+1} = w_i + \left(1 - \frac{1}{2c_2}\right)k_1 + \frac{1}{2c_2}k_2$$

$$k_1 = hf(t_i, w_i)$$

$$k_2 = hf(t_i + c_2h, w_i + c_2k_1)$$

To find the local truncation error, we first substitute the values of a_{21}, b_1 and b_2 in equation (2.13). So we get

$$y_{i+1} = y_i + hf_i + \frac{h^2}{2}(f_t + ff_y)_{t_i}$$
$$+\frac{h^3}{2}\{\frac{c_2}{2}f_{tt} + c_2ff_{xy} + \frac{c_2}{2}f^2f_{yy}\}_{t_i} + \cdots$$
$$= y_i + hf_i + \frac{h^2}{2}(f_t + ff_y)_{t_i}$$
$$+\frac{c_2h^3}{4}\{f_{tt} + 2ff_{ty} + f^2f_{yy}\}_{t_i} + \cdots \qquad (2.14)$$

Subtracting (2.14) from (2.11), we get

$$\tau_{i+1} = h^3\left[\left(\tfrac{1}{6} - \tfrac{c_2}{4}\right)\{f_{tt} + 2ff_{ty} + f^2f_{yy}\} + \tfrac{1}{6}\left(f_tf_y + ff_y^2\right) + \cdots\right]_{t_i}. \qquad (2.15)$$

Therefore, we have local truncation error equals to $O(h^3)$ and hence the methods are of order two.

If we choose $c_2 = 2/3$, then the first term in brackets in (2.15) vanishes and we get a method with minimum local truncation error:

$$k_1 = hf(t_i, w_i),$$

$$k_2 = hf\left(t_i + \frac{2}{3}h, w_i + \frac{2}{3}k_1\right),$$

$$w_{i+1} = w_i + \frac{1}{4}(k_1 + 3k_2).$$

We employed Algorithm (2.2) to approximate the solution of first order IVPs using any RK explicit method. We enter the IVP, number of stages, s and the desired RK method (Butcher tableau) that agrees with s. In the next example, we will use the classical fourth order RK method (Figure (2.3) (e)).

Algorithm (2.2): Runge-Kutta explicit methods.

Step 1: Define $f(t,y)$, $yEx(t)$

Step 2: Input endpoints a, b; initial condition α; step size h

Step 3: Input number of stages s

Step 4: Input Butcher matrix in A, B, C

Step 5: Let $N = (b-a)/h$

Step 6: Let $w_1 = \alpha$, $t_1 = a$, $y_1 = \alpha$

Step 7: For $n = 1$ to N repeat steps 8-18

Step 8: Let $sum1 = 0$

Step 9:

Step 10: For $i = 1$ to s repeat steps 10-15

Step 11: Let $sum2 = 0$

Step 12:

Step 13: For $j = 1$ to $i-1$ repeat step 12

Step 15: $sum2 = sum2 + A(i,j) * k(j)$

Step 16: $k(i) = h * f(t(n) + C(i) * h, w(n) + sum2)$

Step 17: $sum1 = sum1 + B(i) * k(i)$

Step 18: $w(n+1) = w(n) + sum1$

Step 19: $t(n+1) = t(n) + h$

Step 20: $y(n+1) = fEx(t(n+1))$

Step 21

Find $GE = abs(y - w)$

Output t, y, w, GE

Stop

We translated Algorithm (2.2) into a Matlab Program (2.2), which we will use to approximate the solution of the IVP and compare the approximate solution w with the exact solution y.

Example (2.2)

We can approximate the solution to the initial value problem

$y' = -2ty^2, \ t \in [0,1], \ y(0) = 1.$

using the fourth order classical Runge−Kutta method with $h_1 = 0.1$ and for $h_2 = 0.05$.

This initial value problem has the exact solution

$y(t) = 1/(1 + t^2)$.

We represented the results of running Program (2.2) for this problem in Table (2.3) and Figure (2.4). We see that the results of this method are better than the results of Euler's method.

In addition, Table (2.3) shows that the global errors at $t = 1$ are 6.0221055E-07 and 4.0931106E-08 for h_1=0.1 and h_2=0.05 respectively.

Straight forward computation of the ratio for these global errors we obtain

$$\frac{4.0931106E-08}{6.0221055E-07} = 0.0679681 \approx \left(\frac{0.05}{0.1}\right)^4 = .0625,$$

which agrees with Runge-Kutta of $O(h^4)$.

Figure (2.4) shows how close the approximate solution w and the exact solution are.

Table (2.3): Results of Example (2.2) using RK4 method with h_1=0.1 and h_2=0.05

t_i	y_i	w_i h_1=0.1	$error_i$ h_1=0.1	w_i h_2=0.05	$error_i$ h_2=0.05
0.0000000	1.0000000	1.0000000	0.0000000E+00	1.0000000	0.0000000E+00
0.1000000	0.9900990	0.9900990	8.4950827E-08	0.9900990	5.1539253E-09
0.2000000	0.9615384	0.9615381	3.1788036E-07	0.9615384	1.8421105E-08
0.3000000	0.9174312	0.9174306	5.9514099E-07	0.9174312	3.3211020E-08
0.4000000	0.8620690	0.8620682	7.8202896E-07	0.8620690	4.2046747E-08
0.5000000	0.8000000	0.7999992	7.9098146E-07	0.8000000	4.0618882E-08
0.6000000	0.7352941	0.7352935	6.1736802E-07	0.7352941	2.9229003E-08
0.7000000	0.6711410	0.6711406	3.2007944E-07	0.6711409	1.1436653E-08
0.8000000	0.6097561	0.6097561	2.1627292E-08	0.6097561	8.3141618E-09
0.9000000	0.5524862	0.5524865	3.4201159E-07	0.5524862	2.6452970E-08
1.0000000	0.5000000	0.5000006	6.0221055E-07	0.5000001	4.0931106E-08

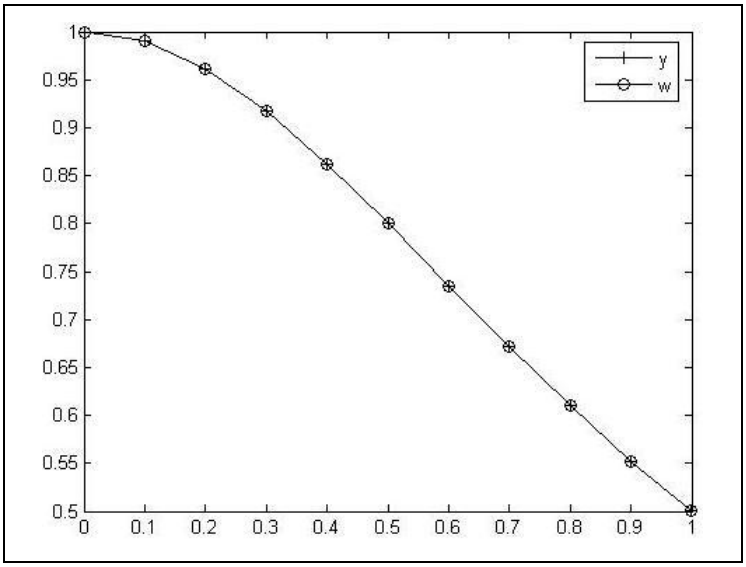


Figure (2.4): Comparison between the approximated solution and exact solution of Example (2.2) using h_1=0.1

2.2.3 Taylor Methods

To derive Taylor methods to solve the initial value problem (1.2), we consider equations (2.5),(2.6), (2.7) and (2.8). These equations give us Taylor methods of order n

$$w_{i+1} = w_i + hT^{(n)}(t_i, w_i, h), \qquad (2.16)$$

where

$$T^{(n)}(t_i, y_i, h) = y_i' + \frac{1}{2}hy_i'' + \cdots + \frac{1}{n!}h^{n-1}y_i^{(n)}$$

and local truncation error at $t = t_{i+1}$

$$\tau_{i+1} = R_{n+1}(c) = \frac{1}{(n+1)!}h^{n+1}y^{(n+1)}(c) \quad c \in (a,b),$$

and, if $|y^{(n+1)}(t)| \leq M$ for all t ∈ (a, b), then

$$|\tau_{i+1}| \leq \frac{M}{(n+1)!}h^{n+1},$$

that is

$\tau_{i+1} = O(h^{n+1})$ and the method is $O(h^n)$.

Example (2.3)

We can apply Taylor's method of orders two and six to the next initial value problem using step size $h = 0.2$.

$$y' = y - t^2 + 1, \ \ 0 \le t \le 2, \ y(0) = 0.5.$$

This initial value problem has the exact solution

$y(t) = (t + 1)^2 - 0.5e^t$.

Results of running Program (3.3) for this problem are represented in Table (2.4) and Figure (2.5). The exact solution y is given in column 2, approximate solution and global error using second order Taylor method in columns 3 and 4, and finally approximated solution and global error using sixth order Taylor method in columns 5 and 6. It is clear we have better results using higher order Taylor methods with the same step size.

In Figure (2.3), we have a closer look to see what happens at the last step. Comparing Figures (2.5) (a) and (b), we see that w and y in (a) are not so close as in (b). That means we have better results with higher order Taylor methods.

Table (2.4): Results of Example (2.3) using Taylor's methods 2 and 6 with h=0.2

t_i	y_i	n=2 w_i	$error_i$	n=6 w_i	$error_i$
0.0000000	0.5000000	0.5000000	0.0000000E+00	0.5000000	0.0000000E+00
0.2000000	0.8292986	0.8300000	7.0137909E-04	0.8292986	1.3023073E-09
0.4000000	1.2140877	1.2158000	1.7123488E-03	1.2140877	3.1812835E-09
0.6000000	1.6489406	1.6520760	3.1354001E-03	1.6489406	5.8284417E-09
0.8000000	2.1272295	2.1323327	5.1031844E-03	2.1272295	9.4918331E-09
1.0000000	2.6408591	2.6486459	7.7868327E-03	2.6408591	1.4491690E-08
1.2000000	3.1799415	3.1913480	1.1406482E-02	3.1799416	2.1240226E-08
1.4000000	3.7324000	3.7486446	1.6244568E-02	3.7324000	3.0266683E-08
1.6000000	4.2834838	4.3061464	2.2662606E-02	4.2834838	4.2248928E-08
1.8000000	4.8151763	4.8462986	3.1122332E-02	4.8151763	5.8053327E-08

2.0000000	5.3054720	5.3476843	4.2212341E-02	5.3054720	7.8784993E-08

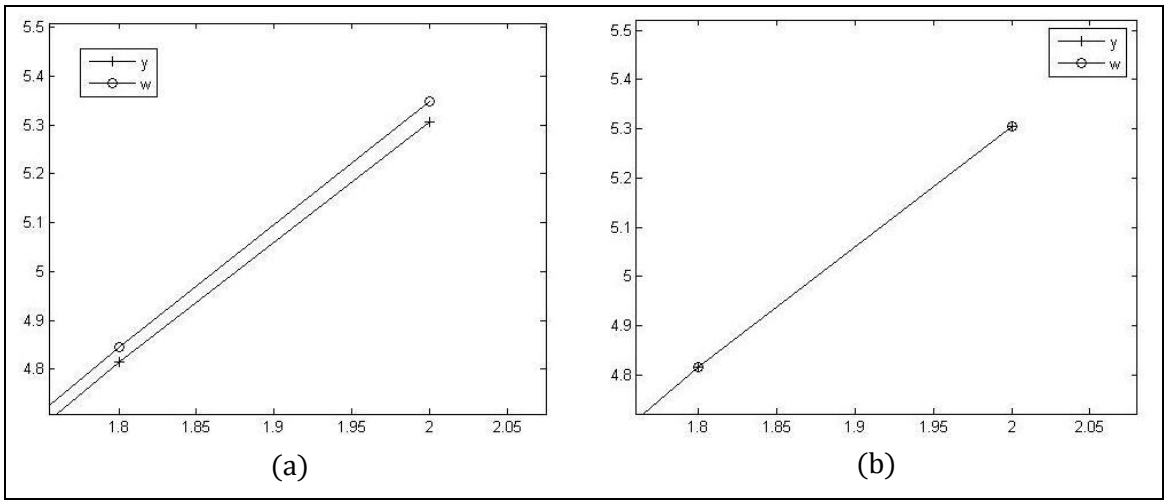


Figure (2.5): Results of Example (2.3) with h=0.2 (a) Taylor 2 (b) Taylor 6

Table (2.4) shows that the global error at $t = 2$ is 7.8784993E-08 for n=6 and h=0.2. We solved the same problem with n=6 and h=0.05. We found that the global error at $t = 2$ is 1.3433343E-009

We notice that

$$\frac{1.3433343E-009}{7.8784993E-08} = 0.017051 \approx \left(\frac{0.1}{0.2}\right)^6 = 0.015625$$

which agrees with the theoretical error estimate for sixth order Taylor method which is equal $O(h^6)$.

2.3 Multistep Methods

The 1883 paper of Bashforth and Adams [1] and the 1926 paper of Moulton [14] were the foundation blocks of developing multistep methods. Through

their work, the explicit Adams-Bashforth methods, the implicit Adams-Moulton methods and the predictor-corrector methods were established. Milne also contributed in this field by the methods called after him and by the so-called Milne's device, which estimates error in predictor-corrector methods, see [13].

Multistep methods use the solution at a number of previous points to find the solution at any point t_{i+1}. If we use $w_i, w_{i-1}, \dots, w_{i-m+1}, f(t_i, w_i)$,

$f(t_{i-1}, w_{i-1}), \dots, f(x_{i-m+1}, w_{i-m+1})$

at m previous points $t_i, t_{i-1}, \dots, t_{i-m+1}$ to find the approximation w_{i+1} to $y(t)$ at t_{i+1}, then we call the method as an m−step multistep method.

For example, the method

$$w_{i+1} = w_i + h\,\varphi(t_i, t_{i-1}, t_{i-2}, w_i, w_{i-1}, w_{i-2}, h),$$

is a three step method.

We can write a general explicit m−step method as [10]

$$w_{i+1} = w_{i-r} + h\varphi(t_{i-m+1}, \dots, t_{i-1}, t_i, w_{i-m+1}, \dots, w_{i-1}, w_i, h),$$

where $r \in \{0, 1, \dots, m-1\}, \quad i = m, m+1, \dots, N.$ (2.17)

If the right hand side contains w_{i+1}, then we have an implicit method.

To construct multi-step methods, we first integrate the differential equation

$$y' = f(t, y)$$

in the interval $[t_{i-r}, t_{i+1}]$, getting

$$\int_{t_{i-r}}^{t_{i+1}} dy = \int_{t_{i-r}}^{t_{i+1}} f(t, y)\,dt$$

$$y(t_{i+1}) = y(t_{i-r}) + \int_{t_{i-r}}^{t_{i+1}} f(t,y)\,dt \ \ [10]. \qquad (2.18)$$

We will use Newton's backward difference interpolating polynomial to approximate the integrand $f(x,y)$.

2.3.1 Predictor Methods

An m-step predictor method is an m-step explicit method defined in (2.17). If we use Newton's backward difference interpolating polynomial of degree $m-1$ to approximate the integrand $f(x,y)$ in the interval $[t_{i-r}, t_{i+1}]$.

For equally spaced m points, $t_i, t_{i-1}, \dots, t_{i-m+1}$, we get degree $m-1$ Newton's backward difference interpolating polynomial of $f(t,y)$, [3]

$$P_{m-1}(t) = \sum_{k=0}^{m-1}(-1)^k \binom{-s}{k} \nabla^k f(t_i, y_i) \qquad (2.19)$$

$$f(t,y) = P_{m-1}(t) + \frac{f^m\left(\xi_i, y(\xi_i)\right)}{m!}\,(t-t_i)(t-t_{i-1})\dots(t-t_{i-m+1}). \quad (2.20)$$

Now, for $t = t_i + sh$ we get

$$(t-t_i) = sh, \ \ (t-t_{i-1}) = h(s+1), \dots, (t-t_{i-m+1}) = h(s+m-1).$$

Therefore,

$$(t-t_i)(t-t_{i-1})\dots(t-t_{i-m+1}) = h^m s(s+1)\dots(s+m-1). \qquad (2.21)$$

Substituting (2.19), (2.20), (2.21) and $r = 0$ in (2.18), we get

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f\big(t, y(t)\big)\,dt$$

$$= y(t_i) + \int_{t_i}^{t_{i+1}} \sum_{k=0}^{m-1} (-1)^k \binom{-s}{k} \nabla^k f(t_i, y_i)\,dt$$

$$+\int_{t_i}^{t_{i+1}} \frac{f^{(m)}\left(\xi_i, y(\xi_i)\right)}{m!}(t-t_i)(t-t_{i-1})\dots(t-t_{i-m+1})dt$$

$$= y(t_i) + h\sum_{k=0}^{m-1}\left[\nabla^k f(t_i, y(t_i)\int_0^1 (-1)^k \binom{-s}{k} ds\right]$$

$$+\frac{h^{m+1}}{m!} f^{(m)}\left(\xi_i, y(\xi_i)\right)\int_0^1 s(s+1)\dots(s+m-1)\,ds$$

$(\nabla f_i = f_i - f_{i-1} \text{ and } \nabla^k f_i = \nabla^{k-1} f_i - \nabla^{k-1} f_{i-1})$.

This leads to the m-step explicit Adams−Bashforth methods,

$$w_{i+1} = w_i + h\sum_{k=0}^{m-1}\left[\nabla^k f(t_i, w_i)\int_0^1 (-1)^k \binom{-s}{k} ds\right]. \qquad (2.22)$$

with local truncation error

$$|\tau_i| < \frac{Mh^{m+1}}{m!}\int_0^1 s(s+1)\dots(s+m-1)\,ds. \qquad (2.23)$$

Therefore, $\tau_i = O(h^{m+1})$ and the methods are $O(h^m)$.

Now, we compute the coefficients $\int_0^1 (-1)^k \binom{-s}{k} ds$.

For $k = 0$, $\int_0^1 (-1)^0 \binom{-s}{0} \mathrm{ds} = \int_0^1 ds = 1$.

For $k = 1$, $\int_0^1 (-1)^1 \binom{-s}{1} \mathrm{ds} = \int_0^1 (-1)(-s)ds = \int_0^1 sds = \frac{1}{2}$.

For $k = 2$, $\int_0^1 (-1)^2 \binom{-s}{2} \mathrm{ds} = \frac{1}{2!}\int_0^1 (-s)(-s-1)ds = \frac{1}{2!}\int_0^1 s(s+1)ds = \frac{5}{12}$.

For $k = 3$, $\int_0^1 (-1)^3 \binom{-s}{3} \mathrm{ds} = \frac{1}{3!}\int_0^1 s(s+1)(s+2)ds = \frac{9}{24}$.

For $k = 4$, $\int_0^1 (-1)^4 \binom{-s}{4} ds = \frac{1}{4!}\int_0^1 s(s+1)(s+2)(s+3)\mathrm{ds} = \frac{251}{720}$.

⋮

For $k = k$, $\int_0^1 (-1)^k \binom{-s}{k} ds = \frac{1}{k!} \int_0^1 s(s+1)(s+2)(s+k-1)\text{ds}$.

We used Matlab Program (2.3) to compute these integrals for $k = 1, \ldots, 10$ and represented the results in Table (2.5). Using these results, (2.22) becomes

$$w_{i+1} = w_i + h\left[f_i + \frac{1}{2}\nabla f_i + \frac{5}{12}\nabla^2 f_i + \frac{3}{8}\nabla^3 f_i + \frac{251}{720}\nabla^4 f_i + \cdots\right]$$

For $m = 1, k = 0$

$$w_{i+1} = w_i + hf_i$$

Table (2.5): Coefficients of Adams-Bashforth methods

k	$\int_0^1 (-1)^k \binom{-s}{k} ds$
0	1
1	1/2
2	5/12
3	3/8
4	251/720
5	95/288
6	19087/60480
7	5257/17280
8	1070017/3628800
9	25713/89600
10	26842253/95800320

For $m = 2, k = 0,1$

$$w_{i+1} = w_i + h\left[f_i + \frac{1}{2}\nabla f_i\right] = w_i + h\left[f_i + \frac{1}{2}(f_i - f_{i-1})\right]$$

$$w_{i+1} = w_i + \frac{h}{2}[3f_i - f_{i-1}]$$

For $m = 3, k = 0,1,2$

$$w_{i+1} = w_i + h\left[f_i + \frac{1}{2}\nabla f_i + \frac{5}{12}\nabla^2 f_i\right]$$

$$= w_i + h\left[f_i + \frac{1}{2}(f_i - f_{i-1}) + \frac{5}{12}\nabla(f_i - f_{i-1})\right]$$

$$= w_i + h\left[f_i + \frac{1}{2}(f_i - f_{i-1}) + \frac{5}{12}((f_i - f_{i-1}) - (f_{i-1} - f_{i-2})\right]$$

$$= w_i + h\left[\frac{23}{12}f_i - \frac{16}{12}f_{i-1} + \frac{5}{12}f_{i-2}\right]$$

$$w_{i+1} = w_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}].$$

For $m = 4$, $k = 0,1,2,3$ we find in the same way that

$$w_{i+1} = w_i + h\left[f_i + \frac{1}{2}\nabla f_i + \frac{5}{12}\nabla^2 f_i + \frac{3}{8}\nabla^3 f_i\right]$$

$$w_{i+1} = w_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]. \qquad (2.24)$$

The method in (2.24) is the 4-step Adam-Bashforth method, for which we wrote Algorithm (2.3) to approximate first order IVP (1.2). It uses RK4 to approximate the solution at t_2, t_3, t_4, where $t_1 = a$ and $y(t_1) = \alpha$.

Algorithm (2.3): Adams-Bashforth 4-step method using RK4 to find the starting points

Step 1:	*Define* $f(t,y)$, $yEx(t)$
Step 2:	*Input endpoints* a,b; *initial condition* α; *step size* h
Step 3:	*Let number of steps* $N = (b-a)/h$
Step 4:	*Let* $t_1 = a$;
Step 5:	*Let* $w_1 = \alpha$; *The vector w contains the approximad solution of y*
Step 6:	*Let* $y_1 = \alpha$; *The vector y contains the exact solution of y*
Step 7:	*For* $i = 1$ *to* N *repeat steps* $8-15$
Step 8:	$k_1 = f(t_i, w_i)$
Step 9:	$k_2 = f(t_i + 1/2h, w_i + 1/2hk_1)$
Step 10:	$k_3 = f(t_i + 1/2h, w_i + 1/2hk_2)$
Step 11:	$k_4 = f(t_i + h, w_i + hk_3)$
Step 12:	
Step 13:	

Step 15: $w_{i+1} = w_i + h/6(k_1 + 2k_2 + 2k_3 + k_4)$

Step 16: $t_{i+1} = t_i + h$

Step 17: $y_{i+1} = yEx(t_{i+1})$

For $i = 4$ *to* N *repeat steps* $17 - 19$

Step 18:

$$w_{i+1} = w_i + h/24[55f(t_i, w_i) - 59f(t_{i-1}, w_{i-1})$$

Step 19:

$$+37f(t_{i-2}, w_{i-2}) - 9f(t_{i-3}, w_{i-3})]$$

Step 20: $t_{i+1} = t_i + h$

Step 21: $y_{i+1} = yEx(t_{i+1})$

Step22: *Find* $GE = abs(y - w)$

Output t, y, w, GE

Stop

Example (2.4)

Using explicit Adams−Bashforth four−step method with step size $h = 0.2$, we can approximate the solution of the initial value problem,

$y' = y - t^2 + 1, \quad 1 \leq t \leq 3, \quad y(0) = 4 - 0.5e^1,$

and then refine the solution by using $h = 0.1$,

This initial value problem has the exact solution

$y(t) = (t + 1)^2 - 0.5e^t.$

We represented the results for this problem in Table (2.6).

At $t = 3$, the global error 5.7083601E-03 for $h = 0.2$ and 5.0822471E-04 for $h = 0.1$. We note that

$$\frac{5.0822471E-04}{5.7083601E-03} = 0.089032 \approx \left(\frac{0.1}{0.2}\right)^4 = 0.0625$$

which agrees with the theoretical error estimate for Adams−Bashforth which is equal to $O(h^4)$.

Table (2.6): Results of Example (2.4) using Adams-Bashforth 4-step method with h=0.2 and h=0.1

t_i	y_i	w_i h=0.2	$error_i$ h=0.2	w_i h=0.1	$error_i$ h=0.1
1.0000000	2.6408591	2.6408591	0.0000000E+00	2.6408591	0.0000000E+00
1.2000000	3.1799417	3.1799386	2.9179384E-06	3.1799414	1.8399729E-07
1.4000000	3.7323999	3.7323945	5.6519293E-06	3.7324054	5.4625666E-06
1.6000000	4.2834840	4.2834759	7.9774882E-06	4.2835054	2.1798965E-05
1.8000000	4.8151765	4.8153915	2.1536958E-04	4.8152208	4.4641027E-05
2.0000000	5.3054719	5.3060632	5.9144641E-04	5.3055487	7.6600882E-05
2.2000000	5.7274933	5.7285838	1.0904461E-03	5.7276139	1.2052076E-04
2.4000001	6.0484118	6.0501885	1.7766465E-03	6.0485921	1.8013343E-04
2.5999999	6.2281308	6.2308531	2.7220398E-03	6.2283912	2.6023496E-04
2.8000000	6.2176766	6.2216763	3.9999373E-03	6.2180438	3.6697558E-04
3.0000000	5.9572315	5.9629397	5.7083601E-03	5.9577398	5.0822471E-04

Figure (2.6) compares the curves of exact and approximate solutions of Example (2.4).

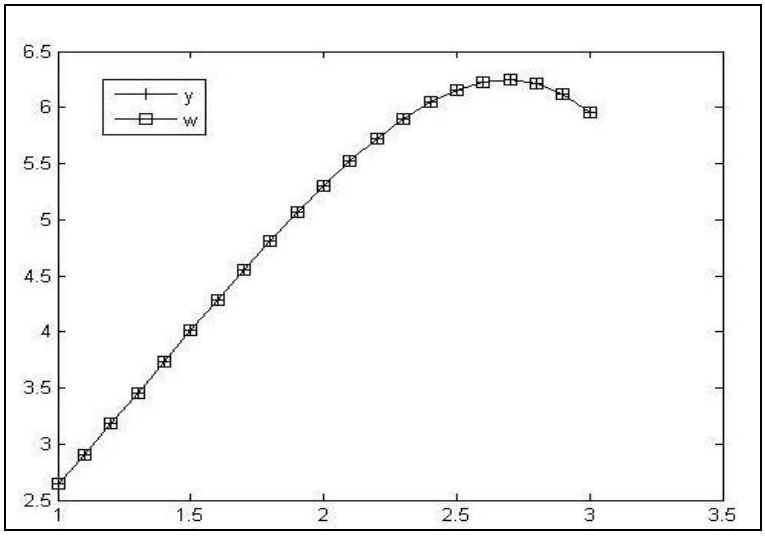


Figure (2.6): Comparing approximate and exact solutions in Example (2.4)

To derive Milne's methods, we let $r = m - 1$ in (2.18), so we get

$$y(t_{i+1}) = y(t_{i-m+1}) + \int_{t_{i-m+1}}^{t_{i+1}} f(t,y)\,dt \qquad (2.25)$$

If the Newton Backward-Difference interpolating polynomial is integrated over $[t_{i-m+1}, t_{i+1}]$ using $m-1$ points; $(t_i, y_i), (t_{i-1}, y_{i-1}) \ldots$ (t_{i-m+2}, y_{i-m+2}), then we get P_{m-2} polynomial and (2.19) becomes

$$P_{m-2}(t) = \sum_{k=0}^{m-2} (-1)^k \binom{-s}{k} \nabla^k f(t_i, y_i) \qquad (2.26)$$

Again, we have $t = t_i + sh$. Therefore, we get $dt = hds$. When $t = t_{i+1}$, $s = 1$. In addition, when $t = t_{i-m+1}, s = -m+1$.

From (2.25) and (2.26), we get

$$w_{i+1} = w_{i-m+1} + h \sum_{k=0}^{m-2} \left[\nabla^k f(t_i, y_i) \int_{-m+1}^{1} (-1)^k \binom{-s}{k} ds \right].$$

For $m = 4$ we get

$$w_{i+1} = w_{i-3} + h \sum_{k=0}^{2} \left[\nabla^k f(t_i, y_i) \int_{-3}^{1} (-1)^k \binom{-s}{k} ds \right]$$

For $k = 0$, $\int_{-3}^{1} (-1)^0 \binom{-s}{0} ds = \int_{-3}^{1} ds = 4$.

We used Program (2.5) to compute $\int_{-3}^{1} (-1)^k \binom{-s}{k} ds$ for $k = 1, .., 4$ and put the results into Table (2.7) together with the coefficient 4 when $k = 0$.

Table (2.7): Coefficients of 4-step Milne's method

K	0	1	2	3	4
$\int_{-3}^{1} (-1)^k \binom{-s}{k} ds$	4	-4	$\frac{8}{3}$	0	$\frac{14}{45}$

Using the data in Table (2.7), we get

$$w_{i+1} = w_{i-3} + h[4f_i - 4\nabla f_i + \frac{8}{3}\nabla^2 f_i],$$

$$w_{i+1} = w_{i-3} + \frac{4h}{3}[2f_i - f_{i-1} + 2f_{i-2}],$$

which is the 4-step explicit Milne's method. Since the coefficient vanishes for $k = 3$, the local truncation error becomes

$$|\tau_i| \leq \frac{14h^{m+1}M}{45 * m!}, \quad m = 4.$$

Therefore, $\tau_i = O(h^{h+1})$ and the method is of O(h^h).

The following Algorithm (2.4) approximates the solution to the IVP (1.2) using RK4 to produce the starting points for Milne's 4-step method.

Algorithm (2.4) Milne's 4-step method

Step 1: *Define* $f(t,y)$, $yEx(t)$

Step 2: *Input endpoints* a,b; *initial condition* α; *step size* h

Step 3: *Let number of steps* $N = (b-a)/h$

Step 4: *Let* $t_1 = a$;

Step 5: *Let* $w_1 = \alpha$; *The vector w contains the approximad solution of y*

Step 6: *Let* $y_1 = \alpha$; *The vector y contains the exact solution of y*

Step 7: *For* $i = 1$ *to* N *repeat steps* $8-15$

Step 8:

Step 9: $k_1 = f(t_i, w_i)$

Step 10: $k_2 = f(t_i + 1/2h, w_i + 1/2hk_1)$

Step 11: $k_3 = f(t_i + 1/2h, w_i + 1/2hk_2)$

Step 12: $k_4 = f(t_i + h, w_i + hk_3)$

Step 13: $w_{i+1} = w_i + h/6(k_1 + 2k_2 + 2k_3 + k_4)$

Step 15: $t_{i+1} = t_i + h$

Step 16: $y_{i+1} = yEx(t_{i+1})$

Step 17:

Step 18: *For* $i = 4$ *to* N *repeat steps* $17-19$

Step 19: $w_{i+1} = w_{i-3} + h/3[2f(t_i, w_i) - f(t_{i-1}, w_{i-1}) + 2f(t_{i-2}, w_{i-2})]$

Step 20: $t_{i+1} = t_i + h$

Step 21: $y_{i+1} = yEx(t_{i+1})$

Step 22: $Find \quad GE = abs(y - w)$

$Output\ t, y, w, GE$

$Stop$

We translated this algorithm into the Matlab Program (2.6).

Example (2.5)

Consider Milne's 4-step method to approximate the solution to the initial value problem

$$y'(t) = -2ty, \qquad 0 \le t \le 1, \qquad y(0) = 0.5,$$

using step size $h = 0.1$.

This initial value problem has the exact solution

$$y(t) = 0.5e^{-t^2}.$$

Table (2.8) contains the results of Example (2.5).

Table (2.8): Results of Example (2.5)

t_i	y_i	w_i	$\lvert GE_i \rvert$
0.0000000	0.5000000	0.5000000	0.0000000E+00
0.1000000	0.4950249	0.4950249	2.0791741E-10
0.2000000	0.4803947	0.4803947	1.9583726E-09
0.3000000	0.4569656	0.4569656	5.6262963E-09
0.4000000	0.4260719	0.4261052	3.3334312E-05
0.5000000	0.3894004	0.3894376	3.7220070E-05
0.6000000	0.3488382	0.3488814	4.3255306E-05
0.7000000	0.3063132	0.3063461	3.2902120E-05
0.8000000	0.2636462	0.2637070	6.0803573E-05
0.9000000	0.2224290	0.2224652	3.6136491E-05
1.0000000	0.1839397	0.1839864	4.6702633E-05

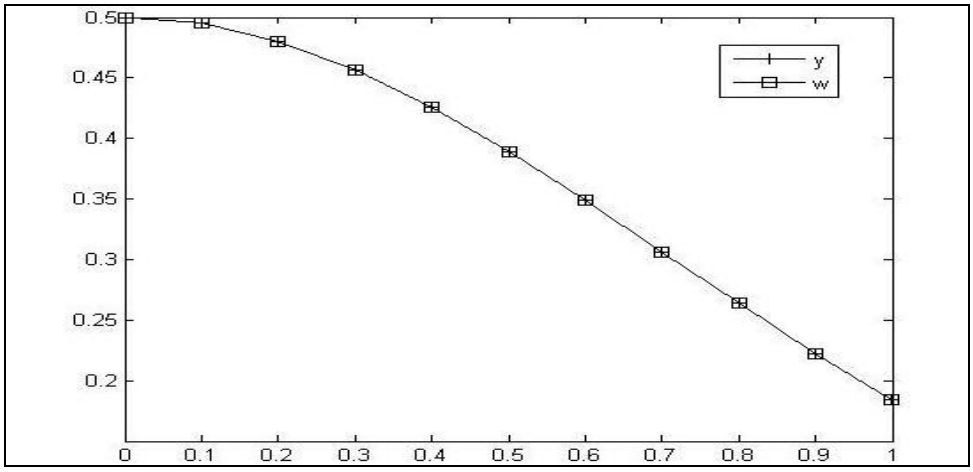


Figure (2.7): Comparing approximate and exact solutions of Example (2.5)

Figure (2.7) compares the curves of exact and approximate solutions of Example (2.5).

2.3.2 Implicit Methods

If we use the point $(t_{i+1}, f(t_{i+1}, y(t_{i+1})))$ together with the m points used in Adams-Bashforth methods to interpolate $f(t,y)$ in the integral

$$\int_{t_i}^{t_{i+1}} f(t,y)dt,$$

then we will get an interpolating polynomial P_m of degree m.

Now, for $t = t_i + sh$, $dt = h.ds$ and $t = t_{i+1} - h + sh = t_{i+1} + (s-1)h$. Therefore, $\binom{-s}{k}$ becomes $\binom{-(s-1)}{k} = \binom{-s+1}{k}$. In addition, for $t = t_i$, $s = 0$ and for $t = t_{i+1}$, $s = 1$.

Applying this to (2.22) and (2.23), we get the implicit Adams−Moulton methods

$$w_{i+1} = w_i + h\sum_{k=0}^{m}\left[\nabla^k f(t_{i+1}, w_{i+1})\int_0^1 (-1)^k \binom{-s+1}{k} ds\right]. \quad (2.27)$$

$$|\tau_i| < \frac{Mh^{m+2}}{(m+1)!}\int_0^1 (s-1)s(s+1)\dots(s+m-1)\,ds.$$

Now for k=0, $\int_0^1 (-1)^0\binom{-s+1}{0}ds=\int_0^1 ds$ =1,

for k=1, $\int_0^1 (-1)^1\binom{-s+1}{1}ds=\int_0^1 (s-1)ds= -\frac{1}{2}$,

for k=2, $\int_0^1 (-1)^2\binom{-s+1}{2}ds=\frac{1}{2}\int_0^1 (s-1)sds= -\frac{1}{12}$,

for k=3, $\int_0^1 (-1)^3\binom{-s+1}{3}ds=\frac{1}{3!}\int_0^1 (s-1)s(s+1)ds= -\frac{1}{24}$,

for k=4, $\int_0^1 (-1)^4\binom{-s+1}{4}ds=\frac{1}{4!}\int_0^1 (s-1)s(s+1)(s+2)ds= -\frac{19}{720}$.

Hence, we have

$$w_{i+1} = w_i + h\left[f_{i+1} - \frac{1}{2}\nabla f_i - \frac{1}{12}\nabla^2 f_{i+1} - \frac{1}{24}\nabla^3 f_{i+1} - \frac{19}{720}\nabla^4 f_{i+1} + \cdots\right]$$

Table (2.9) contains the coefficients $\int_0^1 (-1)^k\binom{-s+1}{k}ds$ for $k = 0,..10$. We computed these coefficients using Program (2.7).

Table (2.9): Coefficients of Adams-Moulton methods

k	$\int_0^1 (-1)^k\binom{-s+1}{k}ds$
0	1
1	-1/2
2	-1/12
3	-1/24
4	-19/720
5	-3/160
6	-863/60480
7	-275/24192
8	-33953/3628800
9	-8183/1036800
10	-3250433/479001600

Now, for $m = 0$, we get

$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1}),$$

which is the backward Euler's method of order 2.

For m=1, we get the 1-step method of order 2

$$w_{i+1} = w_i + h\left[f_{i+1} - \frac{1}{2}\nabla f_{i+1}\right]$$

$$= w_i + h[f_{i+1} - \frac{1}{2}(f_{i+1} - f_i)]$$

$$= w_i + \frac{h}{2}[f_{i+1} + f_i].$$

For $m = 2$, we get the 2-step method of order 3

$$w_{i+1} = w_i + \frac{h}{12}[5f_{i+1} + 8f_i - f_{i-1}].$$

For $m = 3$, we get the 3-step method of order 4 and $\tau_i = O(h^4)$.

$$w_{i+1} = w_i + \frac{h}{24}[9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2}]. \qquad (2.28)$$

Example (2.6)

We can approximate the solution to the initial value problem in Example (2.4), using the 3-step fourth order Adams-Moulton method with step size $h = 0.2$, and compare the results with the results we got in Example (2.4) for the same step size.

At first, we must solve (2.28) algebraically for w_{i+1} , where

$$f(t_{i+1}, w_{i+1}) = w_{i+1} - (t_{i+1})^2 + 1$$

Doing that, we get

$$w_{i+1} = \frac{24}{24-9h}\left[w_i + \frac{h}{24}[9(-t_{i+1}^2 + 1) + 19f_i - 5f_{i-1} + f_{i-2}]\right].$$

Then we insert this explicit formula as a 3-step Adams-Moulton method in Program (2.7), which gives the results illustrated in Table (2.10).

Table (2.10): Results using the 3-step Adams-Moulton method for Example (2.6)

t_i	y_i	w_i	GE_i
1.0000000	2.6408591	2.6408591	0.0000000E+00
1.2000000	3.1799417	3.1799386	2.9179384E-06
1.4000000	3.7323999	3.7323945	5.6519293E-06
1.6000000	4.2834840	4.2834592	2.4484483E-05
1.8000000	4.8151765	4.8151245	5.1853749E-05
2.0000000	5.3054719	5.3053818	9.0046029E-05
2.2000000	5.7274933	5.7273507	1.4260200E-04
2.4000001	6.0484118	6.0481977	2.1401687E-04
2.5999999	6.2281308	6.2278209	3.1006479E-04
2.8000000	6.2176766	6.2172384	4.3815278E-04
3.0000000	5.9572315	5.9566236	6.0776027E-04

The global error at $t = 3$ using Adams-Moulton method is 6.0776027E-04, while it was 5.7083601E-03 in Example (3.1). This means that we have about 9.4 times less error in the 3-step Adams-Moulton method than the error produced by the 4-step Adams-Bashforth method. Although implicit Adams-Moulton methods give better results than the explicit Adams-Bashforth methods of the same order, they have the weakness of that we have to convert them algebraically to an explicit representation for w_{i+1}.

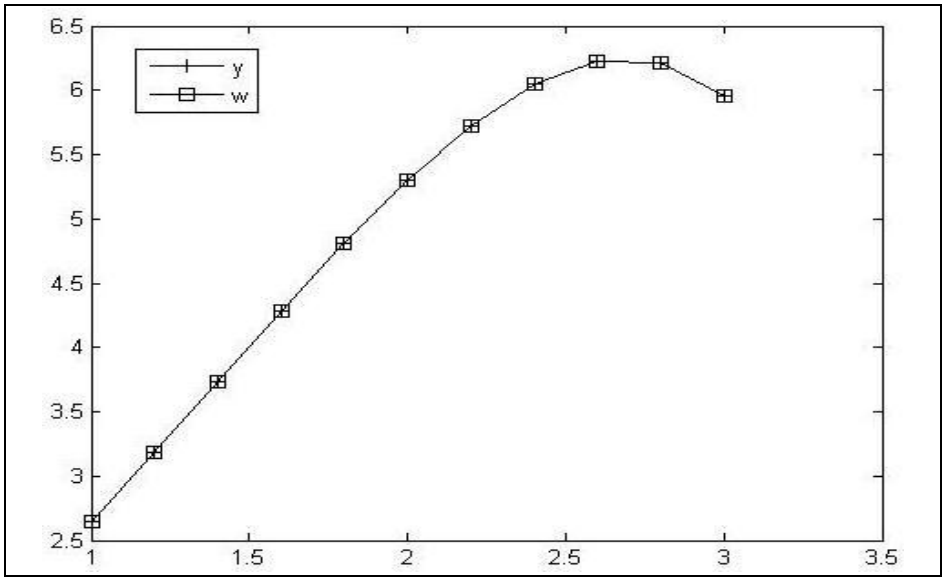


Figure (2.8): Comparing approximate and exact solutions of Example (2.6)

Figure (2.8) compares the approximate and exact solutions of Example (2.6).

2.3.3 Predictor-Corrector Methods

Since implicit multistep methods must be solved algebraically for w_{i+1} before using them to approximate the solution, they are usually not used alone. Rather, they are used with explicit methods to improve the results. Because implicit methods need the value of w_{i+1} at t_{i+1}, explicit methods are used to approximate w_{i+1} for them. This technique of approximating w_{i+1} by explicit methods and improve approximations by implicit methods is called predictor-corrector methods.

As an example, we will use Adams-Bashforth 4th-order 4-step method

$$w_{i+1} = w_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]$$

as a predictor method and Adams-Moulton 4th-order 3-step method

$$w_{i+1} = w_i + \frac{h}{24}[9f_{i+1} + 19f_i - 5f_{i-1} + f_{i-2}]$$

as a corrector method for solving an initial value problem. We first need to calculate the starting values w_1, w_2, w_3, w_4 for the explicit Adams-Bashforth 4-Step method. To do this, we will use the 4th-order 1-step Runge-Kutta method.

Example (2.7)

Consider using the 4-step Adams-Bashforth method as a predictor and the 3-step Adams-Moulton method as a corrector to approximate the solution to the initial-value problem in Example (2.4) with $h = 0.2$. We consider using the fourth order Runge-Kutta method to approximate the starting values.

Table (2.11): Results of Example (2.7)

t_i	y_i	w_i	GE_i
1.0000000	2.6408591	2.6408591	0.0000000E+00
1.2000000	3.1799417	3.1799386	2.9179384E-06
1.4000000	3.7323999	3.7323945	5.6519293E-06
1.6000000	4.2834840	4.2834759	7.9774882E-06
1.8000000	4.8151765	4.8151636	1.2691397E-05
2.0000000	5.3054719	5.3054528	1.9202997E-05
2.2000000	5.7274933	5.7274652	2.7937787E-05
2.4000001	6.0484118	6.0483723	3.9602623E-05
2.5999999	6.2281308	6.2280760	5.5064698E-05
2.8000000	6.2176766	6.2176013	7.5431999E-05
3.0000000	5.9572315	5.9571295	1.0211881E-04

For the predictor-corrector method the global error at $t = 3$ is 1.0211881E-04 which is about six times less than the error (6.0776027E-04) generated by the implicit Adams-Moulton method and about 56 times less than the error (5.7083601E-03) generated by Adams-Bashforth method. Both the Adams-Bashforth and the predictor-corrector methods took advantage at $t = 1.6$ by

the higher accuracy of Runge-Kutta, but this is not the case for Moulton's method.

2.4 Stability and Stability Regions [8, 12]

In this section, we will study absolute stability of one-step numerical schemes. Absolute stability considers the behavior of the numerical scheme when the time step *h* is held fixed and $t \to \infty$.

A numerical method is stable if and only if it is consistent and stable. Moreover, if a numerical is stable and has local truncation error equals to $O(h^{p+1})$ then it has global error equals to $O(h^p)$.

This means that if a method is consistent and stable then $|y(t_i) - w_i| \to 0$ as $t_i \to \infty$. In other words global error vanishes as $t \to \infty$.

Our model for studying stability will be

$$y'(t) = \lambda y(t), \quad y(0) = 1, \quad t > 0, \quad y(t) = e^{\lambda t}, \quad \lambda \in \mathbb{C}. \qquad (2.29)$$

For $z = \alpha h$, the numerical scheme is absolutely stable if [8]

$$|G(z)| < 1,$$

where

$$G(z) = \frac{w_{i+1}}{w_i}.$$

Definition (2.5) [8]

The locus S of points z ∈ C for which $|G(z)| < 1$ is called the (absolute) stability region of the scheme.

Since $z = |z|e^{\theta i}$, the boundary of the region of absolute stability is the roots of the equation

$$G(z) = e^{\theta i}, \qquad 0 \le \theta < 2\pi. \tag{2.30}$$

Now, for Euler's forward method

$$w_{i+1} = w_i + hf(t_i, w_i)$$

$$w_{i+1} = w_i + \lambda h w_i$$

$$\frac{w_{i+1}}{w_i} = 1 + \lambda h$$

$$G(z) = 1 + z.$$

We wrote a Matlab Program (2.9) to find the roots of (2.30) at equally spaced values in the interval $[0, 2\pi[$ and to plot these roots.

To find the boundary of the region of absolute stability for Euler's forward method, we run Program (2.9) for $G(z) = 1 + z$.

Figure (2.9) (a) shows the stability region (shaded) for Euler's forward method. In addition, Figure (2.9) (b) shows the stability region (shaded) for Euler's backward method.

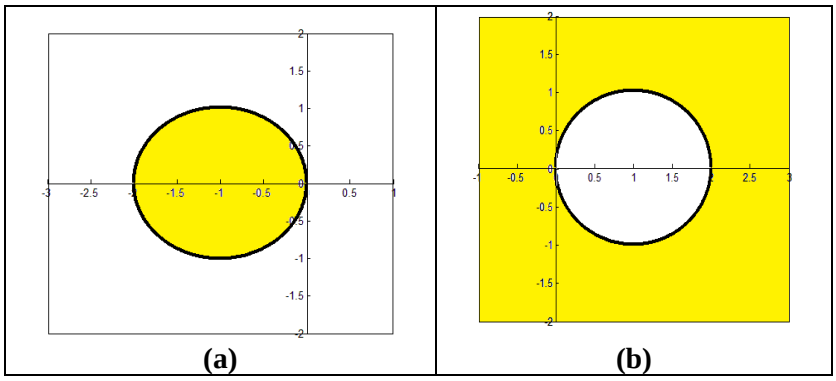


Figure (2.9): (a) Stability region for Euler's Forward method
(b) Stability region for Euler's Backward method

We find that the stability region is the inner of the disk (Figure (2.9) (a)) with center $(-1,0)$ and radius equals one. This means that λh must be inside this disk. For complex $\lambda = a + bi$, we $(ha+1)^2 + (hb-0)^2 < 1$

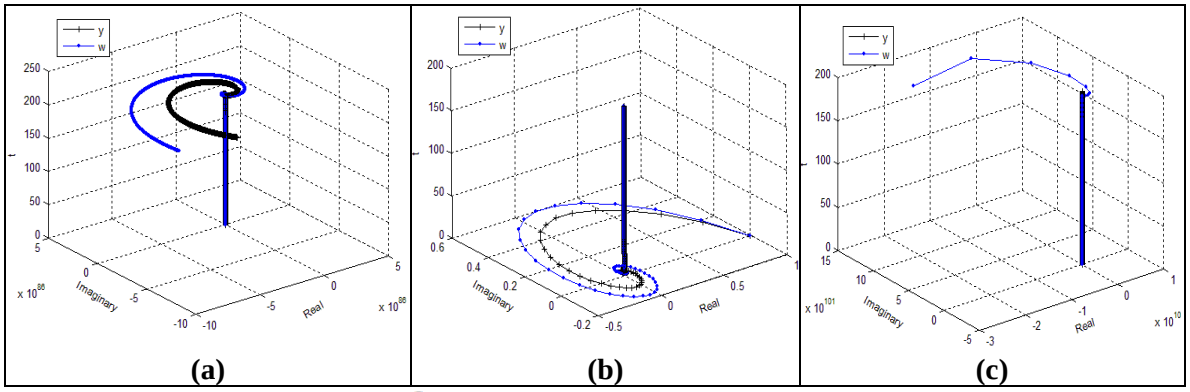
$(ha)^2 + 2ha + 1 + (hb)^2 < 1$ dividing by $h > 0$

$ha^2 + 2a + hb^2 < 0$

$$h < \frac{-2a}{a^2 + b^2}. \qquad (2.31)$$

This means that for $a > 0$, there is no h satisfies (2.31). Therefore, Euler's method is not stable when real$(\lambda) > 0$ and absolutely stable when $real(\lambda) < 0$ and h satisfies (2.31). To make this clearer, we take $\lambda_1 = 1 + 2i$ and $\lambda_2 = -1 + 2i$. For λ_1, there is no $\mathrm{h} < \frac{-2}{5}$, meaning that for any choice of h, Euler's method is unstable. For λ_2, we must choose $h < \frac{2}{5} = 0.4$ so that the method is stable.

We will use the three dimensional Cartesian coordinate system to plot the complex numbers y and w as they vary with t.



(a) (b) (c)

Figure (2.10): Relation between λ, h and stability of Euler's method:

(a) t=200 h=0.001, $\lambda_1 = 1 + 2i$ (b) $\mathrm{t} = 200, \mathrm{h} = 0.1, \lambda_2 = -1 + 2i$

(c) t=200, h=0.5, $\lambda_2 = -1 + 2i$

Figure (2.10) (a) shows that Euler's method is unstable for $\lambda_1 = 1 + 2i$, even though we chose $h = 0.001$. Figure (2.10) (b) shows that Euler's method is stable for $\lambda_2 = -1 + 2i$ and $h = 0.1 < 0.4$. But, Figure (2.10) (c) shows that Euler's method is unstable for $\lambda_2 = -1 + 2i$ and $h = 0.5 > 0.4$.

To study stability of backward Euler's method, we have

$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1})$$

$$w_{i+1} = w_i + h\lambda w_{i+1}$$

$$w_{i+1} - h\lambda w_{i+1} = w_i$$

$$\frac{w_{i+1}}{w_i} = \frac{1}{1 - \lambda h}$$

$$G(z) = \frac{1}{1 - z}.$$

We used Program (2.10) to plot the boundary of the region of absolute stability for $G(z) = \frac{1}{1-z}$. Figure (2.9) (b) shows the stability region for this method is the outer side of the disk with centre (1,0) and radius 1.

We can find the function G for other methods in the same way as we did for Euler's methods.

We wrote Program (2.11) to find G(z) for explicit RK methods and to plot the boundary of regions of stability. Table (2.12) contains G(z) for RK methods 1,…,4, generated by Program (2.11). Figure (2.11) contains the regions of stability of RK methods 1,2,3 and 4, generated by Program (2.11).

Table (2.12): Amplification functions G(z) for RK1,…,RK4

Method	G(z)

RK1	1+z
RK2	1+z+1/2*z^2
RK3	1+z+1/2*z^2+1/6*z^3
RK4	1+z+1/2*z^2+1/6*z^3+1/24*z^4

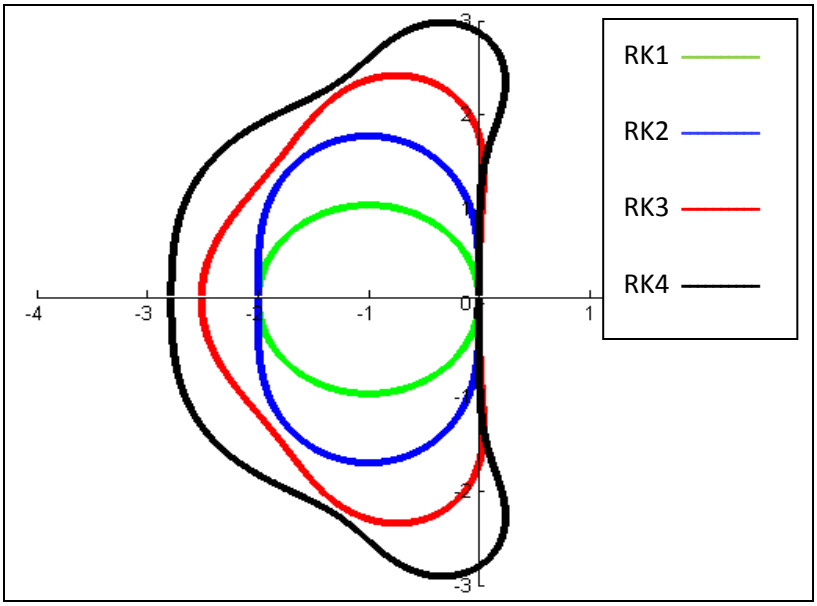


Figure (2.11): Stability regions for RK1,…,RK4

Finally, we will derive the functions G(z) for Taylor methods. First, we note that

$$y' = \lambda y, \quad y'' = \lambda^2 y \ , \dots , \quad y^{(n)} = \lambda^n y \, .$$

Therefore, Taylor's method of order n will be

$$w_{i+1} = w_i + h\,\lambda w_i + \frac{h^2\lambda^2}{2} w_i + \cdots + \frac{(h\lambda)^n}{n!} w_i$$

$$w_{i+1} = w_i + z w_i + \frac{z^2}{2} w_i + \cdots + \frac{z^n}{n!} w_i$$

And hence,

$$G(z) = 1 + z + \cdots + \frac{z^n}{n!} = \sum_{p=1}^{n} \frac{z^p}{p!} .$$

Program (2.12) is designed to plot the boundary of the regions of stability of any range of Taylor's methods. We used this program to plot these boundaries for Taylor's methods $n = 1, \ldots, 6$ in Figure (2.12).

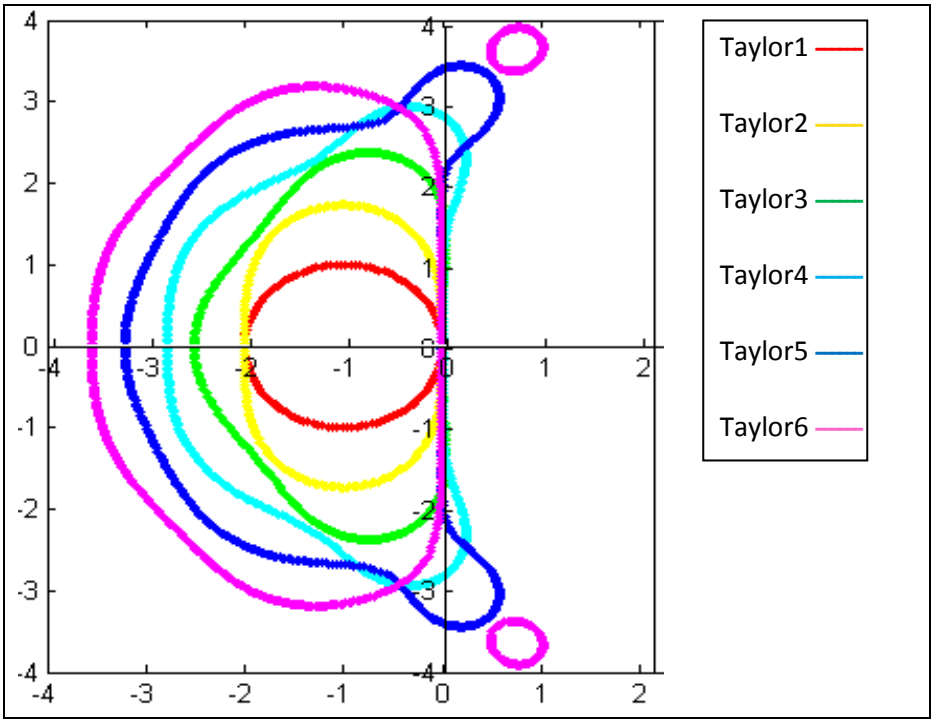


Figure (2.12): Stability regions for Taylor1,…, Taylor6 methods

Figure (2.13) demonstrates stability behavior of fourth order Taylor's method when applied to (2.29) with different values of h and lambda. The method shows absolute stability in (b), (d) and (g). It shows instability in (a), (c) and (f). In (e) the error remains bounded and the method is stable but not absolutely stable.

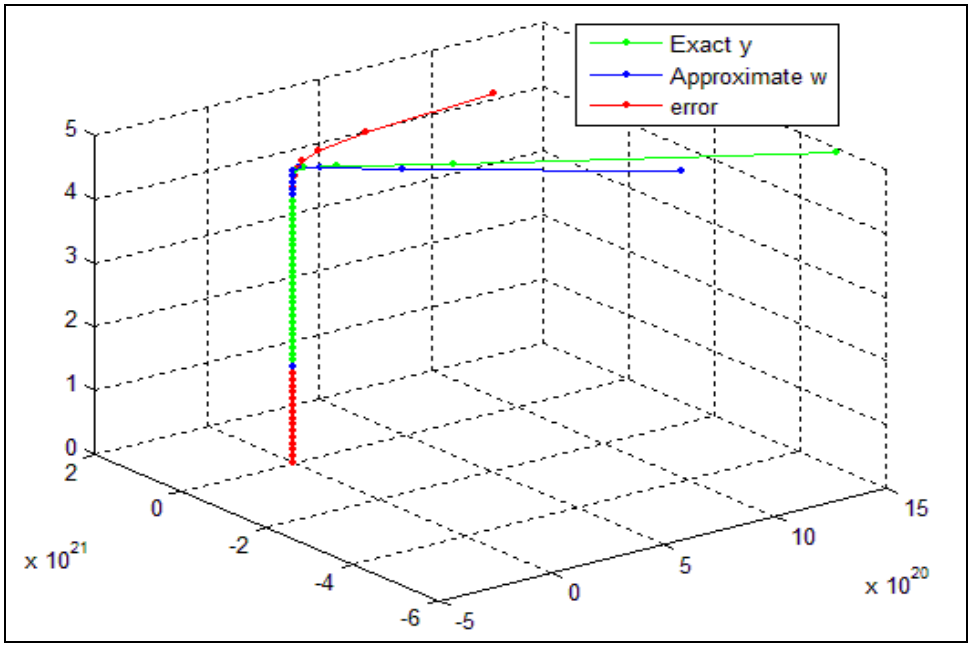


Figure (2.13): Stability behavior of fourth order Taylor method:
(a) lambda=10+1*i; h=.1; b=5

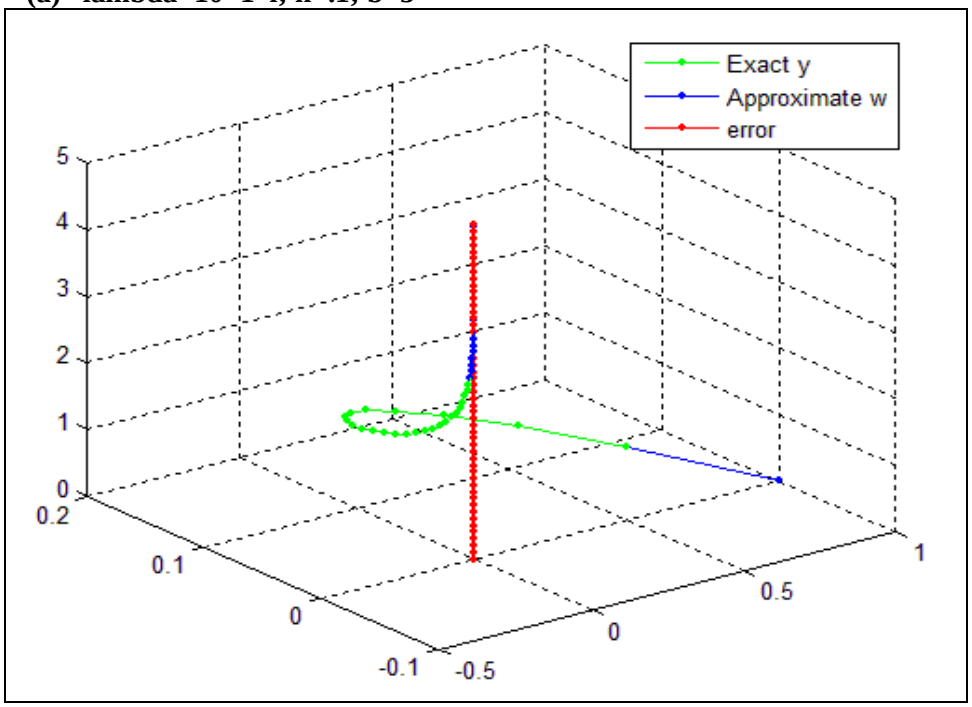


Figure (2.13) :(b) lambda=-2+1*i; h=.1; b=5

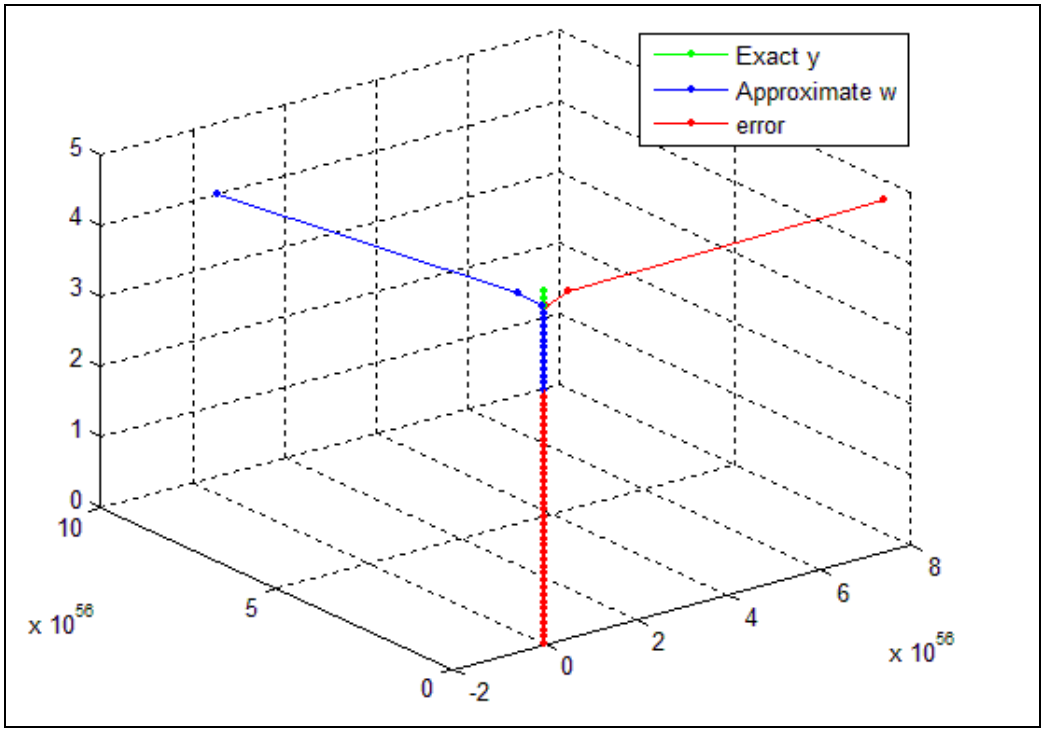


Figure (2.13): (c) lambda=-50+1*i; h=.1; b=5;

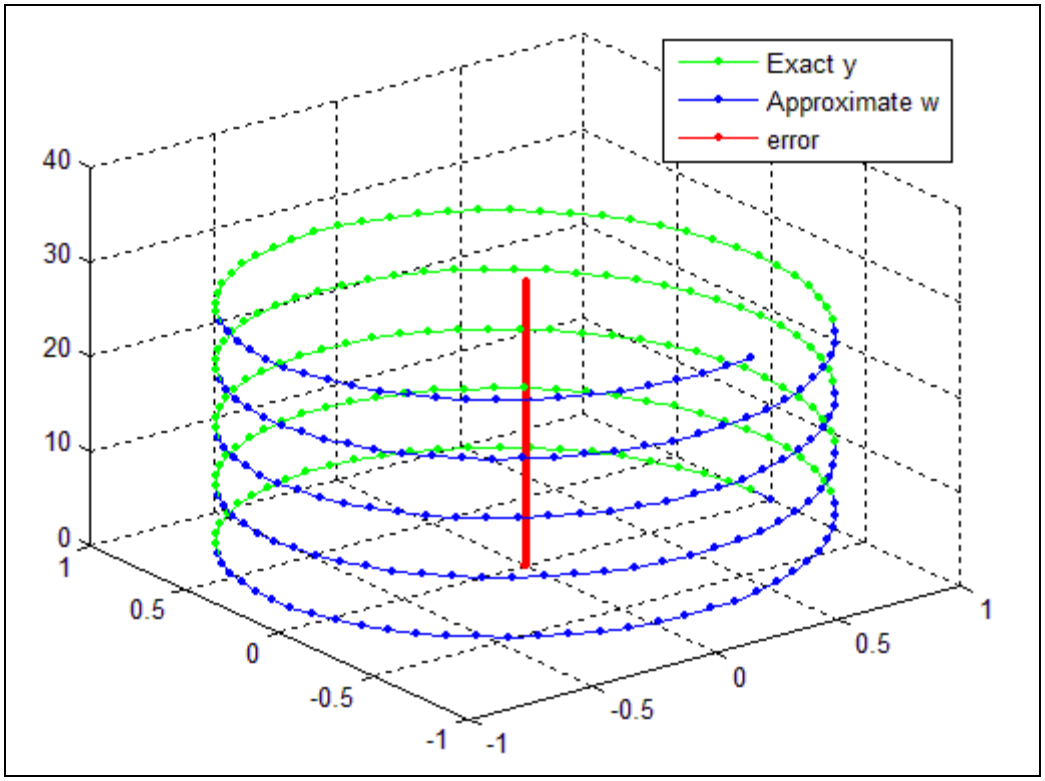


Figure (2.13) :(d) lambda=1*i; h=.1; b=30

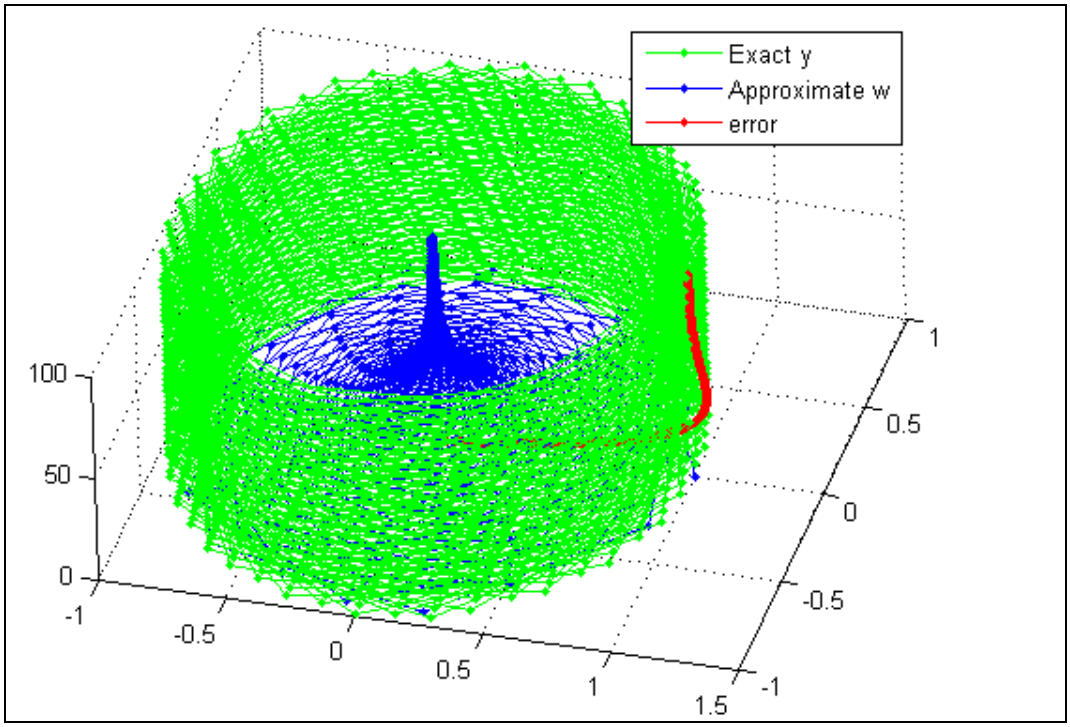


Figure (2.13): (e) lambda=10*i; h=.1; b=100;

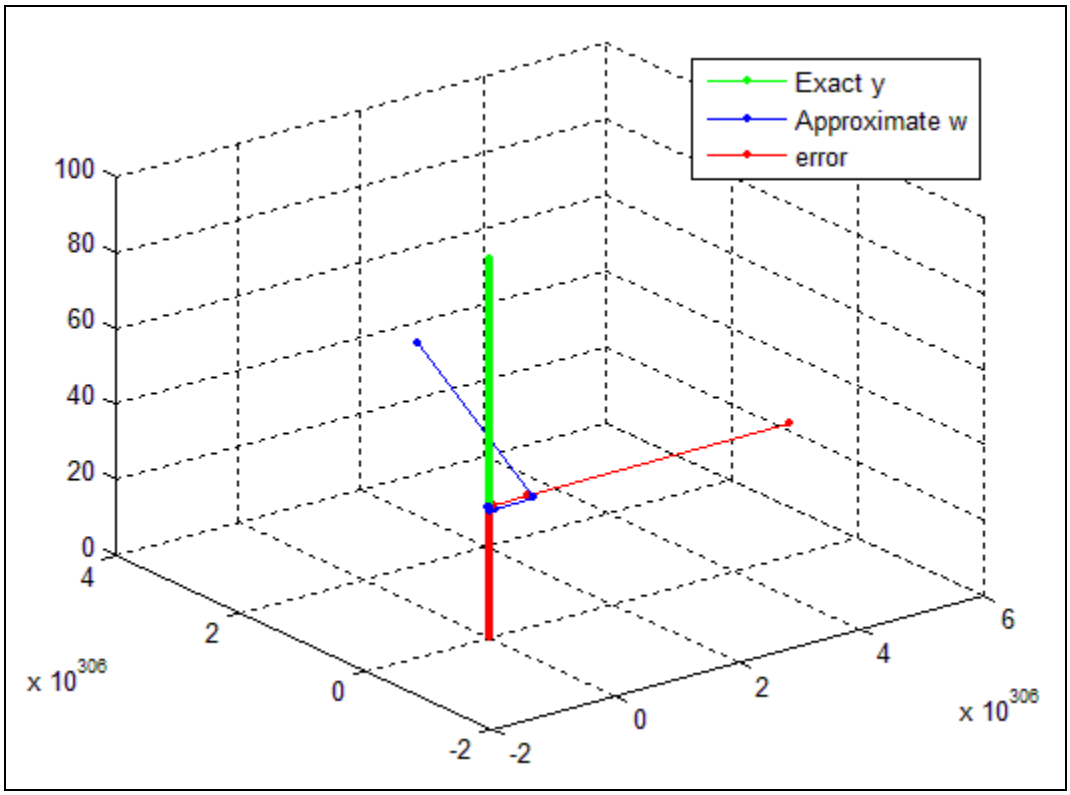


Figure (2.13): (f) lambda=-40*i; h=.1; b=100;

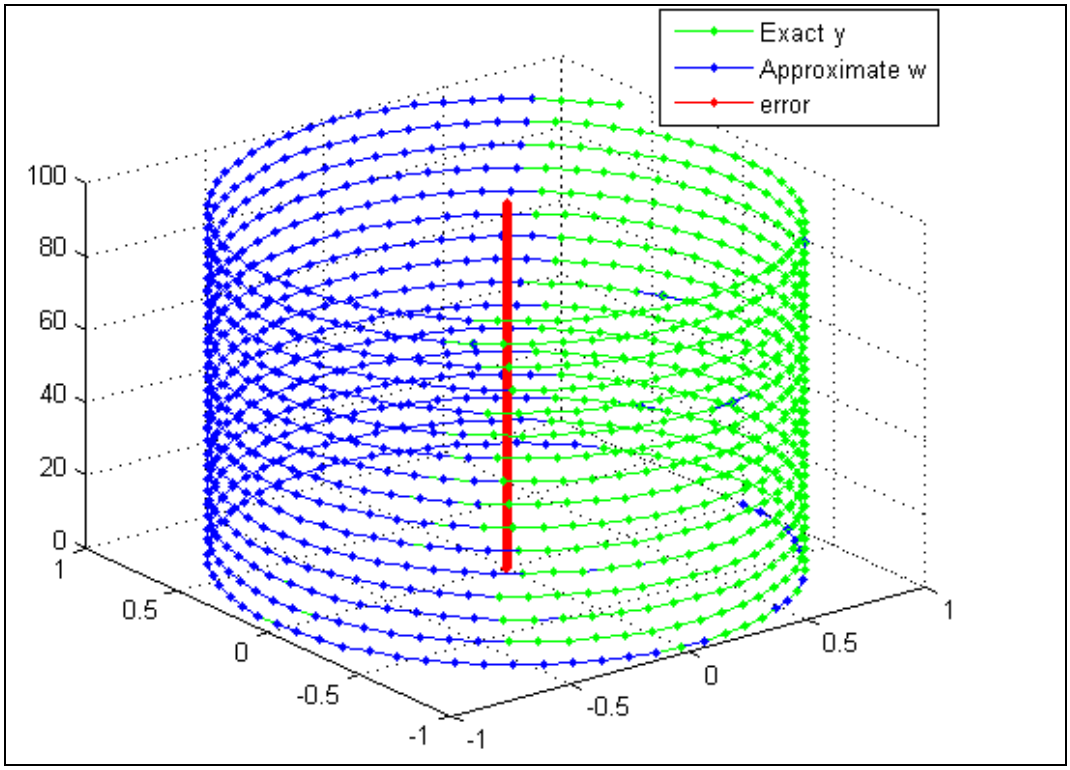


Figure (2.13) (g) lambda=-i; h=.1; b=100;

Chapter Three
Higher Order Taylor Methods

3.1 Introduction

In this chapter, we will be focusing on solving first order initial value problems, systems of first order initial value problems and higher order initial value problems, using higher order Taylor methods. Here, a question arises, since Taylor methods are well known, why we have chosen to investigate Taylor methods. Taylor methods have the weakness of having to find higher order derivatives needed to construct these methods. We thought, it is worth to develop an algorithm and later a computer program to accomplish this task. In this chapter and in the next chapter we will try to answer this question.

3.2 Higher Order Taylor Methods for Solving First Order IVP

In this section, we will develop some numerical algorithms to find $y''(t), \ldots, y^{(n)}(t)$ of our basic problem,

$$y'(t) = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha$$

and construct nth order Taylor methods to solve this problem.

3.2.1 Finding Higher Order Derivatives of First Order IVP's

Theorem (3.1) [17]

If w is a function of $u_1, u_2, \ldots, u_k$ and each is a function of one variable t, then w is a function of t and

$$\frac{dw}{dt} = \frac{\partial w}{\partial u_1}\frac{du_1}{dt} + \frac{\partial w}{\partial u_2}\frac{du_2}{dt} + \cdots + \frac{\partial w}{\partial u_k}\frac{du_k}{dt}.$$

Applying this theorem on

$y'(t) = f(t, y(t))$,

where f, t and y represent w, u_1 and u_2 in the theorem respectively, we get

$$\frac{d\,y'}{dt} = \frac{\partial f}{\partial t}\frac{dt}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}.$$

That is

$$y'' = \frac{\partial y'}{\partial t} + \frac{\partial y'}{\partial y}y'$$

Now let

$$h(t,y) = y''(t).$$

Applying theorem (3.1) on h we get

$$\frac{dy''}{dt} = \frac{\partial h}{\partial t}\frac{dt}{dt} + \frac{\partial h}{\partial y}\frac{dy}{dt},$$

$$y''' = \frac{\partial y''}{\partial t} + \frac{\partial y''}{\partial y}y'.$$

Repeating this process until we get

$$y^{(n)} = \frac{\partial y^{(n-1)}}{\partial t} + \frac{\partial y^{(n-1)}}{\partial y}y'. \qquad (3.1)$$

From (3.1), we get an iterative method to find higher order derivatives y'', $y''', \ldots, y^{(n)}$ of first order IVP's and that is:

$$y^{(i)} = \frac{\partial y^{(i-1)}}{\partial t} + \frac{\partial y^{(i-1)}}{\partial y}y', \; i = 2, 3, \ldots, n. \qquad (3.2)$$

Algorithm (3.1) finds $y'', y''', \ldots, y^{(n)}$ and store them together with y' in the vector y_p, where $y_p(i) = y^{(i)}$. We translated Algorithm (3.1) into a Matlab Program (3.1), which we will use to find the derivatives.

Algorithm (3.1) Finds the first n derivatives of $y(t)$ where $y'(t) = f(t,y)$

To find the first n derivatives of $y(t)$ *where* $y'(t) = f(t,y)$.

We wil use the vector y_p *to hold the derivatives.*

Step 1: *Define* t, y *as symbols*

Step 2: *Define* $f(t,y)$

Step 3: *Input* n

Step 4: *Let* $y^{(1)} = f(t,y)$

Step 5: *Let* $y_p(1) = y^{(1)}$

Step 6: *For* $i = 2$ *to* n *repeat step* $7 - 8$

Step 7: *Let* $y^{(i)} = \frac{\partial y^{(i-1)}}{\partial t} + \frac{\partial y^{(i-1)}}{\partial y} * y^{(1)}$

Step 8: *Let* $y_p(i) = y^{(i)}$

Step 9: *Output* y_p

Step10: *Stop*

Example (3.1)

We can find the first 4 derivatives of $y(t)$ where $y'(t) = \exp(y)$.

Running Program (3.1) for $y'(t) = \exp(y)$ and n = 4, produces the vector y_p, where

$$y_p = [\ \exp(y),\ \exp(2*y), 2*\exp(3*y), 6*\exp(4*y)].$$

That is

y' = exp(y), y'' = exp(2*y), y''' = 2*exp(3*y) and $y^{(4)}$ = 6*exp(4*y).

Example (3.2)

Consider running Program (3.1) for $y' = y\ exp(y)$ and $n = 20$.

Doing that, we get y_p where

$y_p1 = y*\exp(y)$

$y_p2 = \exp(2*y)*(1+y)*y$

$y_p3 = \exp(3*y)*(4*y+2*y^2+1)*y$

$y_p4 = \exp(4*y)*(18*y^2+6*y^3+11*y+1)*y$

$y_p5 = \exp(5*y)*(96*y^3+24*y^4+98*y^2+26*y+1)*y$

$y_p6 = \exp(6*y)*(600*y^4+120*y^5+874*y^3+424*y^2+57*y+1)*y$

$y_p7 = \exp(7*y)*(4320*y^5+720*y^6+8244*y^4+6040*y^3+1614*y^2$
$+120*y+1)*y$

$y_p8 = \exp(8*y)*(35280*y^6+5040*y^7+83628*y^5+83500*y^4$
$+35458*y^3+5682*y^2+247*y+1)*y$

$y_p9 = \exp(9*y)*(322560*y^7+40320*y^8+915984*y^6+1169768$
$*y^5+701164*y^4+187288*y^3+19022*y^2+502*y+1)*y$

$y_p10 = \exp(10*y)*(1+61584*y^2+920350*y^3+5191412*y^4$
$+13329084*y^5+1013*y+16939800*y^6+10824336*y^7$
$+3265920*y^8+362880*y^9)*y$

$y_p11 = \exp(11*y)*(1+194882*y^2+4297240*y^3+35160560*y^4$
$+131888624*y^5+2036*y+251869440*y^6+255992688*y^7$
$+137636640*y^8+36288000*y^9+3628800*y^10)*y$

$y_p12 = \exp(12*y)*(1+607042*y^2+19332662*y^3+223072440*y^4$
$+1178097904*y^5+4083*y+3213860944*y^6+4818505344$
$*y^7+4054649328*y^8+1876883040*y^9+439084800*y^10$
$+39916800*y^11)*y$

$y_p13 = \exp(13*y)*(1+1870122*y^2+84615152*y^3+1347354144$
$*y^4+9745456704*y^5+8178*y+479001600*y^12$

+36634201456*y^6+77114374080*y^7+94313908080*y^8
+67424622336*y^9+27352529280*y^10+5748019200*y^11)*y

y_p14= exp(14*y)*(1+5716680*y^2+362772194*y^3+7836767696*y^4
+75988344096*y^5+16369*y+80951270400*y^12
+383130347344*y^6+1093159611568*y^7+6227020800
*y^13+1851312035760*y^8+1900327028400*y^9
+1177397912448*y^10+424559111040*y^11)*y

y_p15= exp(15*y)*(1+17379206*y^2+1531122296*y^3+44262649196
*y^4+565644812320*y^5+32752*y+6996194069760*y^12
+3745749248752*y^6+14109101755360*y^7+1220496076800
*y^13+31966042883792*y^8+87178291200*y^14
+44921638784640*y^9+39555955434528*y^10
+21578280106752*y^11)*y

y_p16= exp(16*y)*(1+52628898*y^2+6385177274*y^3
+244280080420*y^4+4057808611860*y^5+65519*y
+414624724508160*y^12+34704916926064*y^6
+1690590527741 60*y^7+122029856121600*y^13
+499330912284528*y^8+19615115520000*y^14
+928707031103280*y^9+1307674368000*y^15
+1108940091549408*y^10+852278692798944*y^11)*y

y_p17= exp(17*y)*(1+20922789888000*y^16+158934998*y^2
+26382771464*y^3+1323563238484*y^4+28255332957880
*y^5+131054*y+19026580503389184*y^12+307859356272208
*y^6+1907751093010304*y^7+8342413577832960*y^13
+7198923054947312*y^8+2246704430745600*y^14
+1727636490758524 8*y^9+334764638208000*y^15

+2705765350469596*y^10+27970385778377856*y^11)*y

y_p18= exp(18*y)*(1+6046686277632000*y^16+479032912*y^2

+108232980822*y^3+7066323307308*y^4+192032572801508

*y^5+262125*y+7228421047764829 44*y^12

+2635356154189416*y^6+20495617800709968*y^7

+44024565864727756 8*y^13+9722207607570097 6*y^8

+17552159728434432 0*y^14+29514534100995678 4*y^9

+43550209534003200*y^15+59133239198060486 4*y^10

+79562473892036572 8*y^11+355687428096000*y^17)*y

y_p19= exp(19*y)*(1+88669743833180160 0*y^16+1441816986*y^2

+441554515704*y^3+37279810191336*y^4

+1279389256340592*y^5+524268*y+23718192662660861376

*y^12+21904079389753056*y^6+2114013531810892 32

*y^7+19174597107038578944*y^13+12439198050940882 08

*y^8+10557245814916161024*y^14+47014507794621854 08

*y^9+38561921036622489 60*y^15+11817272449965875616

*y^10+20191479922695276288*y^11+115242726703104000

*y^17+6402373705728000*y^18)*y

y_p20= exp(20*y)*(1+88341506421223357440*y^16+4335412050*y^2

+1793612585550*y^3+194788586755056*y^4

+8384651931678936*y^5+1048555*y

+69197462314580144736 0*y^12+17763695159874264 0*y^6

+21073883338540219 20*y^7+71909002008909647136 0*y^13

+15211903956287489280*y^8+52267603225747541529 6*y^14

+70648984091409530032*y^9+26228674414200304281 6*y^15

+21931756175940615452 8*y^10+46682593562169495216 0

*y^11+18921620408960102400*y^17+2311256907767808000
*y^18+1216451004088320000*y^19)*y.

3.2.2 Constructing Taylor Expansion for First order IVP's

The next Algorithm (3.1) finds the first n derivatives of $y(t)$, where $y'(t) = f(t,y)$ and constructs the nth order $T^{(n)}(t_i, y_i, h)$ defined in (2.8)

$$T^{(n)}(t_i, y_i, h) = \left[y_i' + \frac{1}{2}hy_i'' + \cdots + \frac{1}{n!}h^{n-1}y_i^{(n)}\right],$$

for the nth order Taylor method (2.16)

$$w_{i+1} = w_i + hT^{(n)}(t_i, w_i, h), \quad \mathrm{i} = 1, \ldots, \mathrm{N}.$$

We translated this algorithm into a Matlab Program (3.2), which we will use to solve the following examples.

Algorithm (3.2) Finds the first n derivatives of $y(t)$ where $y'(t) = f(t,y)$ and constructs T

To construct T defined as $T^{(n)}$ in (2.8) by finding first the derivatives of $y(t)$ where $y'(t) = f(t,y)$

We wil use the vector y_p to hold the derivatives.

Step 1: *Define t, y and h as sympols*

Step 2: *Define $f(t,y)$*

Step 3: *Input n*

Step 4: *Let $fac = 1$*

Step 5: *Let $y^{(1)} = f(t,y)$*

Step 6: *Let $T = y^{(1)}$*

Step 7: *Let $y_p(1) = y^{(1)}$*

Step 8: *For $i = 2$ to n repeat steps $9 - 12$*

Step 9:	*Let* $y^{(i)} = \frac{\partial y^{(i-1)}}{\partial t} + \frac{\partial y^{(i-1)}}{\partial y} * y^{(1)}$
Step 10:	*Let* $fac = fac * i$
Step 11:	*Let* $T = T + \frac{h^{i-1}}{fac} * y^{(i)}$
Step 12:	*Let* $yp(i) = y^{(i)}$
Step 13:	*Output* y_p, T
Step 14:	*Stop*

Example (3.3)

Consider constructing $T^{(4)}$ for $y'(t)$, where $y'(t) = log(t)$.

Entering $n = 4$ and $y'(t) = log(t)$ into Program (3.2), we get

$$T^{(4)} = log(t) + 1/2 * h/t - 1/6 * h\text{^}2/t\text{^}2 + 1/12 * h\text{^}3/t\text{^}3.$$

Example (3.4)

Consider running Program (3.2) for $n = 15$ and $y' = t - y$.

Doing that, we get

$T^{(15)}$ =t-y+1/2*h*(1-t+y)+1/6*h^2*(-1+t-y)+1/24*h^3*(1-t+y)+1/120*h^4
*(-1+t-y)+1/720*h^5*(1-t+y)+1/5040*h^6*(-1+t-y)+1/40320*h^7*(1-t+y)
+1/362880*h^8*(-1+t-y)+1/3628800*h^9*(1-t+y)+1/39916800*h^10
*(-1+t-y)+1/479001600*h^11*(1-t+y)+1/6227020800*h^12*(-1+t-y)
+1/87178291200*h^13*(1-t+y)+1/1307674368000*h^14*(-1+t-y).

3.2.3 Approximating the Solution of First Order IVP's using Higher Order Taylor Methods

Now, we are ready to introduce an algorithm to approximate the solution of the first order initial value problem using higher order Taylor methods.

Algorithm (3.3) enables the user to approximate IVPs with different Taylor orders (n) and step sizes (h). It also, finds numerical approximations and exact values of the solution of the problem at each t_i and stores them in the vectors w and y. In addition, it finds the accumulated global errors at each step and stores them in the vector named 'error'.

We translated Algorithm (3.3) into Matlab Program (3.3) to find $T^{(n)}$, generates nth Taylor's higher order method and uses it to approximate the solution of the first order IVP.

Algorithm (3.3) Solves first order IVPs using higher order Taylor methods and compares with the exact solution

To approximate the solution of the IVP $y'(t) = f(t,y),\ a \le t \le b, y(a) = \alpha$

And compare the approximated solution w with the exact solution y

We will use T constructed by algorithm (3.2)

Step 1: *Define* $f(t,y)$; *and* $yEx(t)$

Step 2: *Input endpoints* a,b; *initial condition* $y(a) = \alpha$

Step 3: *Input Taylors order n; stepsize h*

Step 4: *Let* $N = (b-a)/h$

Step 5: *Let* $t(1) = a;\ w(1) = \alpha; yExact(1) = \alpha$

Step 6: *Let* $y'(t) = f(t,y)$

Step 7: *Use algorithm* (3.2) *to Get T*

Step 8: *For* $i = 1$ *to N repeat steps* $9-11$

Step 9: $w(i+1) = w(i) + h * T(t(i), w(i), h)$

Step 10: $y(i+1) = yEx(t(i+1))$

Step 11: $t(i+1) = t(i) + h$

Step 12: $GE = abs(y - w)$

Step 13: *Output* t, y, w, GE

Step 14: *Stop*

Example (3.5)

We can find numerical approximation of the solution of the next IVP, using step size $h = 0.1$ and Taylor methods of orders $n = 4$ and $n = 10$.

$$y'(t) = \frac{(2 - 2ty)}{t^2 + 1}, \quad 0 \le t \le 1, \quad y(0) = 1.$$

The exact solution of this problem is

$$y(t) = \frac{(2t + 1)}{t^2 + 1}.$$

The results of running Program (3.3) for h = 0.1 and n=4, 10 are represented in Table (3.1) and Figure (3.1).

The generated Taylor iterative method for n=4 and h=0.1 is

$w_{i+1} = w_i + h\,T^{(4)}(t_i, w_i)$; h is replaced by its value 0.1,

where

$T^{(4)}(t,y)$ = (2-2*t*y)/(t^2+1)+1/10*(3*y*t^2-y-4*t)/(t^2+1)^2

-1/50*(2*t^3*y-2*t*y-3*t^2+1)/(t^2+1)^3+1/1000

*(5*y*t^4-10*y*t^2+y-8*t^3+8*t)/(t^2+1)^4

The generated Taylor iterative method for $n = 10$ and $h = 0.1$ is

$w_{i+1} = w_i + h\,T^{(10)}(t_i, w_i)$; h is replaced by its value 0.1,

where

$T^{(10)}$(t, y)=(2-2*t*y)/(t^2+1)+1/10*(3*y*t^2-y-4*t)/(t^2+1)^2

-1/50*(2*t^3*y-2*t*y-3*t^2+1)/(t^2+1)^3+1/1000
*(5*y*t^4-10*y*t^2+y-8*t^3+8*t)/(t^2+1)^4
-59029581035870595/295147905179352825856
*(3*t^5*y+3*t*y-5*t^4+10*t^2-1-10*t^3*y)/(t^2+1)^5
+377789318629571835/3777893186295716170956 8
*(7*y*t^6-35*y*t^4+21*y*t^2-y-12*t^5-12*t
+40*t^3)/(t^2+1)^6
-1208925819614629935/6044629098073145873530 88
*(4*t^7*y-28*t^5*y+28*t^3*y-4*t*y-7*t^6+35
*t^4-21*t^2+1)/(t^2+1)^7
+193428131138340 85/193428131138340667952988 16
*(126*y*t^4-36*y*t^2+y+16*t-84*y*t^6-112*t^3
+112*t^5+9*y*t^8-16*t^7)/(t^2+1)^8
-2475880078570761 4605/123794003928538027489912 4224
*(-1+36*t^2+5*t*y+5*t^9*y-126*t^4-60*t^3*y+84*t^6
+126*t^5*y-60*t^7*y-9*t^8)/(t^2+1)^9
+792281625142643375935439503 36
*(-20*t^9-330*y*t^4+55*y*t^2+11*y*t^10-y-20*t+462
*y*t^6+240*t^3-504*t^5-165*y*t^8+240*t^7)/(t^2+1)^10.

Table (3.1) contains the results produced by Matlab Program (3.3) used to approximate the solution of the IVP in Example (3.5). At $t = 1$, it is clear that the global error with $n = 10$ is highly reduced with the probation 1.06201E-08 to the global error with $n = 4$.

Table (3.1): Results of Example (3.5) using Taylor methods of orders 4 and 10 with step size h=0.1

t_i	y (t_i)	w_i (Taylor 4)	$error_i$ (Taylor4)	w_i (Taylor 10)	$error_i$ (Taylor 10)
0.0000000	1.0000000	1.0000000	0.0000000	1.0000000	0.0000000E-000
0.1000000	1.1881188	1.1881000	0.0000188	1.1881188	1.8811841E-011
0.2000000	1.3461539	1.3461270	0.0000268	1.3461538	1.4418022E-011
0.3000000	1.4678899	1.4678676	0.0000223	1.4678899	3.1070702E-012
0.4000000	1.5517242	1.5517144	0.0000098	1.5517241	1.0866197E-011
0.5000000	1.6000000	1.6000041	0.0000041	1.6000000	6.6371353E-012
0.6000000	1.6176471	1.6176616	0.0000145	1.6176471	5.7087668E-013
0.7000000	1.6107383	1.6107583	0.0000201	1.6107383	1.9819701E-012
0.8000000	1.5853659	1.5853873	0.0000215	1.5853659	1.7961188E-012
0.9000000	1.5469613	1.5469815	0.0000201	1.5469613	8.6997076E-013
1.0000000	1.5000000	1.5000175	0.0000175	1.5000000	1.8585133E-013

Figure (3.1) compares the exact solution y and the approximated solution of the IVP in Example (3.5) using fourth order Taylor method with step size $h = 0.1$. It shows how close the approximated and exact solutions are.

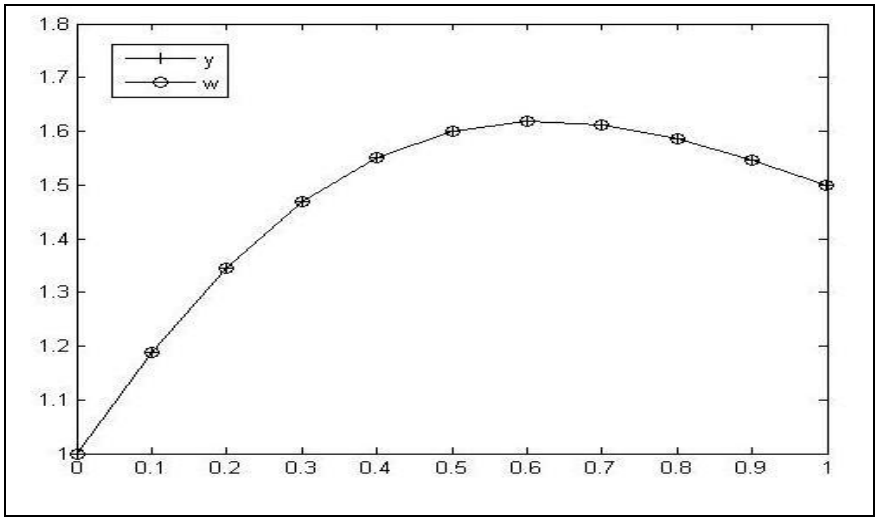


Figure (3.1): Comparing approximated solution w by fourth order Taylor method and exact solution y of the IVP in Example (3.5) with h=0.1

3.3 Higher Order Taylor Methods for Systems of First Order IVP's

A system consisting of k equations of first order ordinary differential equations can be written as

$$u_1'(t) = f_1(t, u_1, \ldots, u_k),$$

$$u_2'(t) = f_2(t, u_1, \ldots, u_k),$$

$$\vdots$$

$$u_k'(t) = f_k(t, u_1, \ldots, u_k),$$

where $u_1, \ldots, u_k$ are functions of t. (3.3)

Reformulating (3.3), we get

$$u_j'(t) = f_j(t, u_1, \ldots, u_k), \quad j = 1, 2, \ldots, k. \tag{3.4}$$

Now, applying theorem (3.1) to each equation in (3.4), where $j = 1, 2, \ldots, k$, we get

$$u_j''(t) = \frac{\partial f_j}{\partial t}\frac{dt}{dt} + \frac{\partial f_j}{\partial u_1}\frac{du_1}{dt} + \frac{\partial f_j}{\partial u_2}\frac{du_2}{dt} + \ldots + \frac{\partial f_j}{\partial u_k}\frac{du_k}{dt}. \tag{3.5}$$

Substituting f_j by u_i', (3.3) becomes

$$u_j''(t) = \frac{\partial u_j'}{\partial t} + \frac{\partial u_j'}{\partial u_1}u_1' \quad \frac{\partial u_j'}{\partial u_2}u_2' + \ldots + \frac{\partial u_j'}{\partial u_k}u_k'.$$

Now, letting

$$g_j(t) = u_j''(t) = \frac{\partial u_j'}{\partial t} + \frac{\partial u_j'}{\partial u_1}u_1' \quad \frac{\partial u_j'}{\partial u_2}u_2' + \ldots + \frac{\partial u_j'}{\partial u_k}u_k'$$

and applying Theorem (3.1) again to $g_j(t)$, then we get

$$g_j'(t) = u_j'''(t) = \frac{\partial g_j}{\partial t}\frac{dt}{dt} + \frac{\partial g_j}{\partial u_1}\frac{du_1}{dt} \quad \frac{\partial g_j}{\partial u_2}\frac{du_2}{dt} + \ldots + \frac{\partial g_j}{\partial u_k}\frac{du_k}{dt}.$$

Substituting g_j by u_j'', we get

$$u_j'''(t) = \frac{\partial u_j''}{\partial t} + \frac{\partial u_j''}{\partial u_1} u_1' + \frac{\partial u_j''}{\partial u_2} u_2' + \ldots + \frac{\partial u_j''}{\partial u_k} u_k'.$$

Repeating this process n-1 times until we reach $u_j^{(n)}$, then we get

$$u_j^{(n)}(t) = \frac{\partial u_j^{(n-1)}}{\partial t} + \frac{\partial u_j^{(n-1)}}{\partial u_1} u_1' + \frac{\partial u_j^{(n-1)}}{\partial u_2} u_2' + \ldots + \frac{\partial u_j^{(n-1)}}{\partial u_k} u_k'.$$

We note that every $u_j^{(i)}$ depends only on $u_j^{(i-1)}$ and the given u_j', where

i = 2,…,n ; j = 1, 2, …, k. So, we can generalize that

$$u_j^{(i)}(t) = \frac{\partial u_j^{(i-1)}}{\partial t} + \frac{\partial u_j^{(i-1)}}{\partial u_1} u_1' + \frac{\partial u_j^{(i-1)}}{\partial u_2} u_2' + \ldots + \frac{\partial u_j^{(i-1)}}{\partial u_k} u_k',$$

where $i = 2, \ldots, n$ and $j = 1, 2, \ldots, k.$ (3.6)

Putting (3.6) into summation form, we get

$$u_j^{(i)}(t) = \frac{\partial u_j^{(i-1)}}{\partial t} + \sum_{p=1}^{k} \left(\frac{\partial u_j^{(i-1)}}{\partial u_p} u_p'\right),$$

where $i = 2, \ldots, n$; $j = 1, 2, \ldots, k.$ (3.7)

In our programs, we will use matrices "up" of dimensions k×n to enter $u_j^{(i)}(t)$ in up(j, i) where i=1,…,n , j = 1, 2, …, k. The notation up(j ,i) and up(p,1) refer to $u_j^{(i)}$ and u_p', where i=1,…,n, j=1, …, k and p=1, …,k. Now for i=1, u(j ,1) is u_j'. Thus, we can write (3.7) as follows:

$u(j, 1) = u_j',$

For $j = 1, \ldots, k$

up(j ,i)= $\frac{\partial up(j,i-1)}{\partial t} + \sum_{p=1}^{k} \left[\frac{\partial up(j,i-1)}{\partial u_p} up(p, 1)\right]$, i=2,…,n. (3.8)

In the following subsections, we will first implement (3.7) in algorithms to find higher order derivatives of u_j' in each equation in the system of first order IVP's (3.4), and then use these derivatives to construct Taylor expansion of each $f_j(t, u_1, \dots, u_k)$ and finally solve the system of IVP using higher order Taylor methods.

3.3.1 Finding Higher Order Derivatives of Systems of First Order IVP's

We put the iterative method (3.7) into Algorithm (3.4). In addition, we translated it into a Matlab Program (3.4). This program uses the alternative form (3.8) instead of (3.7) for finding the first n derivatives of $u_j'(t)$; j $= 2, \dots,$ k.

Algorithm (3.4) Finds higher order derivatives of $u_j'(t)$ in the system of first order IVP's: $u_j'(t) = f_j(t, u_1, \dots, u_k), j = 1, \dots, k$

To find higher order derivatives of $u_j'(t)$ *in the system of first order ODEs*:

$u_j'(t) = f_j(t, u_1, \dots, u_k), j = 1, \dots, k$

Step 1: *Define number of equations k;*

Step 2: *Define the system* $u_j^{(1)}(t) = f_j(t, u_1, \dots, u_k),\ j = 1, \dots, k$

Step 7: *For* $j = 1$ *to k;* *Let* $u_p(j, 1) = u_j^{(1)}(t)$

Step 3: *For* $i = 2$ *to n repeat steps* $4 - 8$

Step 4: *For* $j = 1$ *to k repeat steps* $5 - 8$

Step 5: *Let* $u_j^{(i)} = \partial u_j^{(i-1)} / \partial t$

Step 6: *For* $p = 1$ *to k repeat steps* $7 - 8$

Step 7: *Let* $u_j^{(i)} = u_j^{(i)} + (\partial u_p^{(i-1)} / \partial u_p) * u_p^{(1)}$

Step 8: $Let\ u_p(j,i) = u_j^{(i)}$

Step 9: *Output* u_p

Step 10: *Stop*

Example (3.6)

For n = 3 and 6, we can find the first n-1 derivatives of each u_j' of the following system with respect to t, where u_j are functions of t and j = 1, 2 , 3.

$u_1' = u_2 - u_3 + t$,

$u_2' = 3t^2$,

$u_3' = u_2 + e^{-t}$.

Running Program (3.4), entering the system and three for n, gives the following results:

[u_1' , u_1'' , u_1'''] = [u2-u3+t, 1+3*t^2-u2-exp(-t), 6*t+exp(-t)-3*t^2]

[u_2' , u_2'' , u_2''']= [3*t^2, 6*t, 6]

[u_3' , u_3'' , u_3'''] = [u2+exp(-t), -exp(-t)+3*t^2, 6*t+exp(-t)]

Repeating running the program for the same system but for n=4, gives the following results:

[u_1' , u_1'' , u_1''' , $u_1^{(4)}$] = [u2-u3+t, 1+3*t^2-u2-exp(-t),

6*t+exp(-t)-3*t^2, 6-exp(-t)-6*t]

[u_2' , u_2'' , u_2''', $u_2^{(4)}$] = [3*t^2, 6*t, 6, 0]

[u_3', u_3'', u_3''', $u_3^{(4)}$] = [u2+exp(-t), -exp(-t)+3*t^2, 6*t+exp(-t), 6-exp(-t)]

3.3.2 Constructing Taylor Expansion for a System of First Order IVP's

Taylor expansion for $f(t,y)$ of first order IVP as defined in (2.8)

$$T^{(n)}(t, y, h) = y' + \frac{h}{2} y'' + \ldots + \frac{h^{n-1}}{n!} y^{(n)}.$$

Now for the system (3.3), the nth Taylor expansion of $f_j(t, u_1, \ldots, u_k)$ will be

$$T_j^{(n)}(t, u_1, \ldots, u_k, h) = u_j' + \frac{h}{2!} u_j'' + \cdots + \frac{h^{n-1}}{n!} u_j^{(n)}; \quad j = 1, \ldots, k. \qquad (3.9)$$

Algorithm (3.5) constructs the array T, where T_j refers to $T_j^{(n)}$ defined in the system (3.3). It includes the steps used in Algorithm (3.4), in addition to the steps needed to generate (3.9).

We translated Algorithm (3.5) into Matlab Program (3.5). It constructs the array T, where $T(j)$ refers to $T_j^{(n)}$ in equation (3.9).

Algorithm (3.5) Constructs T_j's as Taylor expansions of each $u_j'(t)$ in the system of first order IVPs: $u_j'(t) = f_j(t, u_1, \ldots, u_k), j = 1, \ldots, k$

To construct the array T, where T_j is the Taylor expansion of $u_j'(t)$ by first finding higher order derivatives of $u_j'(t)$ in the system of first order ODEs:

$u_j'(t) = f_j(t, u_1, \ldots, u_k), j = 1, \ldots, k$

Step 1: Define number of the equations k;

Step 2: Define the system $u_j^{(1)}(t) = f_j(t, u_1, \ldots, u_k)$

Step 3: For $j = 1$ to k repeat steps $5 - 6$

Step 5: Let $up(j, 1) = u_j^{(1)}(t)$

Step 6: Let $T_j(t, u_1, \ldots, u_k, h) = u_j^{(1)}(t)$

Step 7: Let $fac = 1$

Step 8: For $i = 2$ to n repeat steps $9 - 15$

Step 9: Let $fac = fac * i$

Step 10: For $j = 1$ to k repeat steps $11 - 15$

Step 11: $Let\ u_j^{(i)} = \partial u_j^{(i-1)}/\partial t$

Step 12: $For\ p = 1\ to\ k\ repeat\ step\ 13$

Step 13: $Let\ u_j^{(i)} = u_j^{(i)} + (\partial u_p^{(i-1)}/\partial u_p) * u_p^{(1)}$

Step 14: $Let\ up(j,i) = u_j^{(i)}$

Step 15: $T_j(t,u_1,\dots,u_k,h) = T_j(t,u_1,\dots,u_k,h) + (h^{i-1}/fac) * up(j,i)$

Step 16: *Output T*

Step 16: *Stop*

Example (3.7)

Consider constructing T for the system in Example (3.6) using n= 2 and n = 4.

Running Program (3.4), entering the system and n, results:

For n=2

T(1) = (u2-u3+t)+1/2*h*(1+3*t^2-u2-exp(-t)),

T(2) = 3*t^2+3*h*t,

T(3) = (u2+exp(-t))+1/2*h*(-exp(-t)+3*t^2).

For n=4

T(1) = (u2-u3+t)+1/2*h*(1+3*t^2-u2-exp(-t))+1/6*h^2
*(6*t+exp(-t)-3*t^2)+1/24*h^3*(6-exp(-t)-6*t),

T(2) = 3*t^2+3*h*t+h^2,

T(3) = (u2+exp(-t))+1/2*h*(-exp(-t)+3*t^2)+1/6*h^2
*(6*t+exp(-t))+1/24*h^3*(6-exp(-t)).

3.3.3 Higher Order Taylor Methods for Systems of First Order IVP's

From previous discussions, we have the iterative method (2.16) for a single equation IVP

$$w_{i+1} = w_i + hT(t_i, w_i, h), \quad i = 1, 2, \dots, N$$

and Taylor expansion "equation (3.9)" for a system of IVP

$$T_j^{(n)}(t, u_1, \dots, u_k, h) = u_j' + \frac{h}{2!} u_j'' + \dots + \frac{h^{n-1}}{n!} u_j^{(n)}; \quad j = 1, \dots, k.$$

Combining these equations, we get the iterative method for approximating the solution of a system of first order IVP.

$$w_{j,i+1} = w_{j,i} + hT_j^{(n)}(t_i, w_{1,i}, .., w_{k,i})$$

$$i = 1, 2, \dots, N, \quad j = 1, 2, \dots, k \qquad (3.10)$$

In Algorithm (3.6), we changed Taylor method we used in Algorithm (3.3), by using equation (3.10), to deal with systems of equations instead of one equation. Algorithm (3.6) approximates the solution (w)of a system of first order IVP, finds the exact solution (uN) and global error (error), using different orders n and different step sizes h. We put this algorithm into a Matlab Program (3.6) that we will use in approximating the solution of systems of first order ordinary differential equations initial value problems.

Algorithm (3.6): Solves the system $u_j'(t) = f_j(t, u_1, \dots, u_k),\ j = 1, \dots, k$ by higher order Taylor methods

To solve the system $u_j'(t) = f_j(t, u_1, \dots, u_k),\ j = 1, \dots, k$

by higher order Taylor method

Step 1: *Define number of equations k;*

Step 2: *Define the system* $u_j^{(1)}(t) = f_j(t, u_1, \dots, u_k),\ j = 1, \dots, k$

Step 3: Define the functions $fuEx_1(t), \dots, fuEx_k(t)$

Step 4: Define endpoints a, b

Step 5: Define initial conditions $uN_{1,1} = \alpha_1, \dots, uN_{k,1} = \alpha_k$

Step 6: Define step size h; Taylor's order n

Step 7: Use algorithm (3.5) to get $T(t, u_1, \dots, u_k, h)$

Step 8: Let $t_1 = a$

Step 9: Let $N = (b-a)/h$

Step10: For $j = 1$ to k repeat Let $w_{j,1} = uN_{j,1}$

Step 11: For $i = 1$ to N repeat steps $12 - 15$

Step 12: For $j = 1$ to k repeat steps $13 - 14$

Step 13: Let $w_{j,i+1} = w_{j,i} + h * T_j(t, w_{1,i}, \dots, w_{k,i}, h)$

Step 14: Let $uN_{j,i+1} = fuEx_j(t_{i+1})$

Step 15: $t_{i+1} = t_i + h$

Step 16: $GE = abs(uN - w)$

Step 17: Output t, uN, w, GE

Step 18: Stop

Example (3.8)

We can find an approximate solution to the following system using h=0.1 and Taylor order n=4:

$$u_1' = -4u_1 + 3u_2 + 6,$$

$$u_2' = -2.4u_1 + 1.6u_2 + 3.6,$$

$$u_1(0) = 0, \quad u_2(0) = 0, \quad 0 < t < 1.$$

Actual solution of the system

$$u_1(t) = -3.375e^{-2t} + 1.875e^{-0.4t} + 1.5,$$

$$u_2(t) = -2.25e^{-2t} + 2.25e^{-0.4t}.$$

The generated Taylor method (h is replaced by its value 0.1)

$w_{1,\,i+1} = w_{1,\,i} + hT_1(t_i, w_{1,i}, w_{2,i})$ for approximation of u_1, i = 1, …, N.

$w_{2,\,i+1} = w_{2,\,i} + hT_2(t_i, w_{1,i}, w_{2,i})$ for approximation of u_2, i = 1, …, N

where

$T_1(t, u_1, u_2)$ = -224273/62500*u1+41618/15625*u2+672819/125000,

$T_2(t, u_1, u_2)$ = -166472/78125*u1+1297121/937500*u2+249708/78125.

Table (3.2) contains exact values, approximated values and errors at each t_i for each u in the system of Example (3.8).

Table (3.2): Results of Example (3.8) using fourth order Taylor's method using h=0.1

t_i	$u_{1,i}$	$w_{1,i}$	$error_{1,i}$	$u_{2,i}$	$w_{2,i}$	$error_{2,i}$
0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
0.1000000	0.5382639	0.5382552	0.0000087	0.3196321	0.3196262	0.0000058
0.2000000	0.9685130	0.9684988	0.0000143	0.5687917	0.5687822	0.0000095
0.3000000	1.3107365	1.3107190	0.0000175	0.7607448	0.7607331	0.0000117
0.4000000	1.5812844	1.5812652	0.0000191	0.9063333	0.9063206	0.0000127
0.5000000	1.7935270	1.7935075	0.0000196	1.0144155	1.0144024	0.0000130
0.6000000	1.9583968	1.9583776	0.0000192	1.0922257	1.0922129	0.0000128
0.7000000	2.0848298	2.0848114	0.0000184	1.1456703	1.1456580	0.0000122
0.8000000	2.1801288	2.1801114	0.0000172	1.1795682	1.1795567	0.0000114
0.9000000	2.2502594	2.2502437	0.0000158	1.1978493	1.1978387	0.0000105
1.0000000	2.3000934	2.3000791	0.0000144	1.2037157	1.2037061	0.0000096

Figure (3.2) contains plotted results of Example (3.8). It contains four curves; u_1 and u_2 represent the exact solutions, while w_1 and w_2 represent the approximated solutions of u_1 and u_2. We notice that the curves of w_1 and w_2 coincide with the curves of u_1 and u_2 respectively. That means that we have good results.

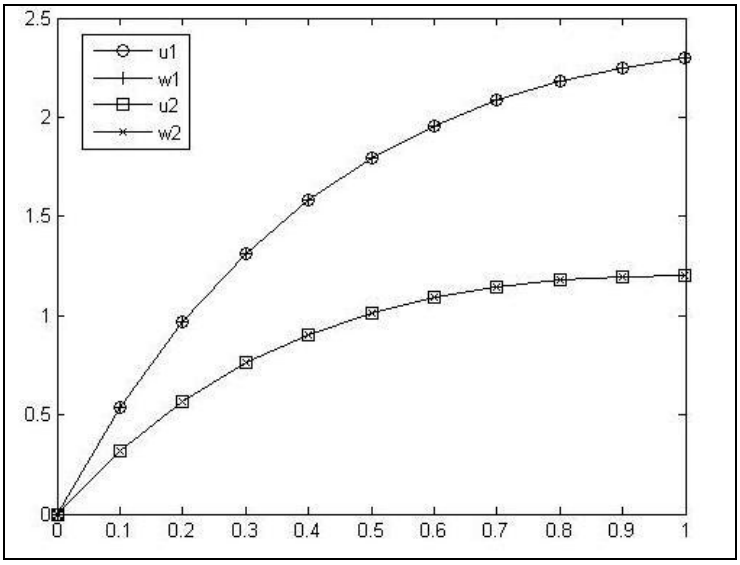


Figure (3.2): Results of Example (3.8)

Example (3.9)

We can find an approximate solution of the following system of IVP.

Take n=9 and h= 0.1.

$$u_1' = u_2 - u_3 + t,$$

$$u_2' = 3t^2, \qquad 0 \le t \le 1$$

$$u_3' = u_2 + e^{(-t)}, \quad h = 0.1;$$

with initial conditions

$$u_1(0) = 1, \quad u_2(0) = 1 \quad \text{and} \quad u_3(0) = -1.$$

This system has the exact solution

$$u_1(t) = -0.05t^5 + 0.25t^4 + t + 2 - e^{-t},$$

$$u_2(t) = t^3 + 1, \text{ and}$$

$$u_3(t) = 0.25t^4 + t - e^{-t}.$$

We ran Program (3.6) for this problem and it produced the following results:

- The generated Taylor method (h is replaced by its value 0.1)

$w_{1,\,i+1} = w_{1,\,i} + hT_1(t_i, w_{1,i}, w_{2,i}, w_{3,i})$ for approximation of u_1, i = 1, …, N,

$w_{2,\,i+1} = w_{2,\,i} + hT_2(t_i, w_{1,i}, w_{2,i}, w_{3,i})$ for approximation of u_2, $i = 1, \ldots, N$,

$w_{3,\,i+1} = w_{3,\,i} + hT_2(t_i, w_{1,i}, w_{2,i}, w_{3,i})$ for approximation of u_3, $i = 1, \ldots, N$,

where

$T_1(t, u_1, u_2, u_3)$ = 19/20*u2-u3+4039/4000*t+1482970649573658273438 9

/29514790517935282585600 0+29/200*t^2

-12264311753678031670870441241325 3

/25353012004564588029934064107520 00*exp(-t),

$T_2(t, u_1, u_2, u_3)$ = 3*t^2+3/10*t+1/100,

$T_3(t, u_1, u_2, u_3)$ =u2+24126580829196784862847019983387 47

/25353012004564588029934064107520 00

*exp(-t)+3/20*t^2+1/100*t+1/4000.

- Table (3.3) contains exact values, approximated values and errors at each t_i for each u in the system of Example (3.9). We have very good results, since the global errors at the final step is about 10^{-15}, which is better than the expected $O(h^9)$.
- Figure (3.3) contains plotted results of Example (3.9). It contains six curves; u_1, u_2 and u_3 represent the exact solutions, while w_1, w_2 and w_3 represent the approximated solutions of u_1, u_3 and u_3. We notice that the curves of w_1, w_2 and w_3 coincide with the curves of u_1, u_2 and u_3 respectively. That means that we have good results.

Table (3.3): Results of Example (3.9), using Taylor method with n=9 and h=0.1

t_i	$u_{1,i}$	$w_{1,i}$	$error_{1,i}$	$u_{2,i}$	$w_{2,i}$	$error_{2,i}$

0.0	1.0000000	1.0000000	0.0000000E+00	1.0000000	1.0000000	0.0000000E+00
0.1	1.1951871	1.1951871	0.0000000E+00	1.0010000	1.0010000	0.0000000E+00
0.2	1.3816532	1.3816532	4.4408921E-16	1.0080000	1.0080000	4.4408921E-16
0.3	1.5610853	1.5610853	2.2204460E-16	1.0270000	1.0270000	4.4408921E-16
0.4	1.7355680	1.7355680	2.2204460E-16	1.0640000	1.0640000	8.8817842E-16
0.5	1.9075318	1.9075318	2.2204460E-16	1.1250000	1.1250000	8.8817842E-16
0.6	2.0797004	2.0797004	4.4408921E-16	1.2160000	1.2160000	1.1102230E-15
0.7	2.2550362	2.2550362	4.4408921E-16	1.3430000	1.3430000	1.1102230E-15
0.8	2.4366870	2.4366870	4.4408921E-16	1.5120000	1.5120000	1.3322676E-15
0.9	2.6279308	2.6279308	4.4408921E-16	1.7290000	1.7290000	1.1102230E-15
1.0	2.8321206	2.8321206	8.8817842E-16	2.0000000	2.0000000	1.1102230E-15

$u_{3,i}$	$w_{3,i}$	$error_{3,i}$
-1.0000000	-1.0000000	0.0000000E+00
-0.8048124	-0.8048124	1.1102230E-16
-0.6183308	-0.6183308	1.1102230E-16
-0.4387932	-0.4387932	1.6653345E-16
-0.2639200	-0.2639200	5.5511151E-17
-0.0909057	-0.0909057	0.0000000E+00
0.0835884	0.0835884	0.0000000E+00
0.2634397	0.2634397	1.1102230E-16
0.4530710	0.4530710	5.5511151E-17
0.6574553	0.6574553	1.1102230E-16
0.8821206	0.8821206	2.2204460E-16

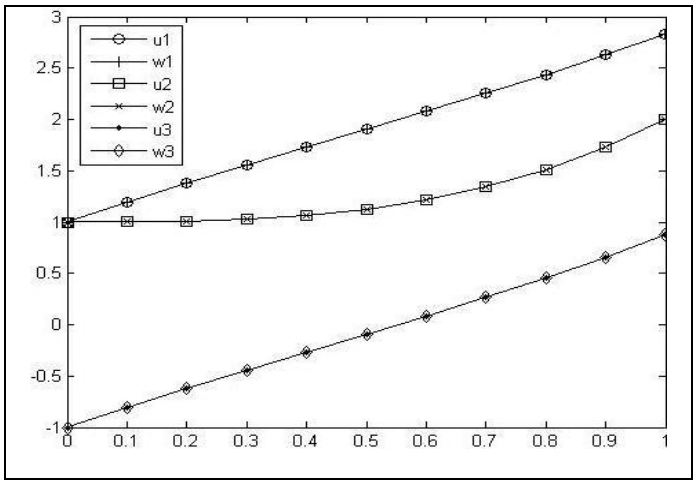


Figure (3.3): Results of Example (3.9)

3.4 Higher Order Taylor Methods for Higher Order IVP's

In this section, we will develop an algorithm, which we will translate into a Matlab program to approximate the solution of the IVP (1.4)

$y^{(k)}(t) = f(t, y, y', y'', \ldots, y^{(k-1)}), \quad a \le t \le b,$

$y(a) = \alpha_1 , y'(a) = \alpha_2, y''(a) = \alpha_3, \ldots, y^{(k-1)}(a) = \alpha_k.$

To approximate the solution of a higher order IVP, we first transform it into a system (1.7) of first order IVP's as discussed in chapter one. Then, we approximate the solution of the system satisfying the initial conditions. Finally, we treat the approximated w_1 of u_1 as the approximation of y.

Algorithm (3.7) solves higher order ordinary differential equations initial value problem

$$y^{(k)}(t) = f\left(t, y, y', y'', \ldots, y^{(k-1)}\right), \quad a \le t \le b,$$

$$y(a) = \alpha_1, \qquad y'(a) = \alpha_2, \ldots, y^{(k-1)}(a) = \alpha_k,$$

following the steps mentioned in the last paragraph.

Algorithm (3.7): Solves higher order IVP's by higher order Taylor methods

To solve the ODE $y^{(k)}(t) = f\left(t, y, y', y'', \ldots, y^{(k-1)}\right)$, *by converting it into a system of first order ODEs and then solve the system by higher order Taylor methods*

Step 1: *Define order of the ODE* k;

Step 2: *Define the ODE* $y^{(k)}(t) = f(t, y, y', y'', \ldots, y^{(k-1)})$;

Step 3: *Define the exactfunction* $fyEx(t)$

Step 4: *Define endpoints* a, b

Step 5: *Define initial conditions* $y(a) = \alpha_1,\ y'(a) = \alpha_2, \ldots, y^{(k-1)}(a) = \alpha_k$

Step 6: *Define step size* h; *Taylor's order* n

Step 7: *Let* $y = u_1$; $uN_1 = y(a)$

Step 8: *For* $i = 1$ *to* $(k-1)$ *repeat steps 9-11*

Step 9: *Let* $u_i' = u_{i+1}$

Step 10: *Let* $y^{(i)} = u_{i+1}$

Step 11: $uN_{i+1} = y^{(i)}(a)$

Step 12: *Let* $u_k' = f(t, y, y', y'', \dots, y^{(k-1)})$; *Note:This results* $u_k' = f(t, u_1, \dots, u_k)$

Step 13: *Use algorithm* (3.5) *to get* $T(t, u_1, \dots, u_k, h)$

Step 14: *Let* $t_1 = a$

Step 15: *Let* $N = (b-a)/h$

Step 16: *For* $j = 1$ *to* k *repeat* *Let* $w_{j,1} = uN_j$

Step 17: *Let* $yN_1 = uN_1$

Step 18: *For* $i = 1$ *to* N *repeat steps* $19 - 22$

Step 19: *For* $j = 1$ *to* k *repeat step* 20

Step 20: *Let* $w_{j,i+1} = w_{j,i} + h * T_j(t, w_{1,i}, \dots, w_{k,i}, h)$

Step 21: $t_{i+1} = t_i + h$

Step 22: *Let* $yN_{i+1} = fyEx(t_{i+1})$

Step 23: $GE = abs(yN - w_{1,1:N+1})$

Step 24: *Output* t, yN, $w_{1,1:N+1}$, GE

Step 25: *Stop*

We translated Algorithm (3.7) into the Matlab Program (3.7), which we will use in solving the following examples.

Example (3.10)

We can transform the following IVP into a system of first order s, and then approximate the solution of y(t) using n = 4 and h = 0.1.

$y^{(10)}(t) = 2y^{(9)} + y^{(8)}(t) - 3y^{(7)}(t) + 2y^{(3)}(t) + y'(t) + y(t) - (\exp(-t) + t + 1)$, $0 \le t \le 1$,

with initial conditions

y(0) = -1, $y'(0) = 2$, $y''(0) = -1$, $y'''(0) = 1$, $y^{(4)}(0) = -1$, $y^{(5)}(0) = 1$, $y^{(6)}(0) = -1$, $y^{(7)}(0) = 1$, $y^{(8)}(0) = -1$ and $y^{(9)}(0) = 1$.

Actual solution is

$$y(t) = t\text{-}e^{-t}.$$

Running Program (3.7) for this problem gives the following results:

The generated System

$$u1' = u2; \quad u1(0) = -1 \qquad u2' = u3; \quad u2(0) = 2$$

$$u3' = u4; \quad u3(0) = -1 \qquad u4' = u5; \quad u4(0) = 1$$

$$u5' = u6; \quad u5(0) = -1 \qquad u6' = u7; \quad u6(0) = 1$$

$$u7' = u8; \quad u7(0) = -1 \qquad u8' = u9; \quad u8(0) = 1$$

$$u9' = u10; \quad u9(0) = -1$$

$$u10' = 2 * u10 + u9 - 3 * u8 + 2 * u4 + u2 + u1 - exp(-t) - t - 1;$$

u10(0)=1.

Table (3.4) contains t, exact solution y, $w_1, \ldots, w_{10}$ approximating $u_1, \ldots, u_{10}$ and "error". The vector error is the absolute value of column two minus column one. We are interested only on the vector w_1, since it approximates u_1, which is equal to y. That means w_1 approximates y.

Table (3.4): Result of Example (3.10) using Taylor method of order n=4 and step size h=0.1

t_i	y_i	$w_{1,i}$	$w_{2,i}$	$w_{3,i}$	$w_{4,i}$	$w_{5,i}$

0.0000000	-1.0000000	-1.0000000	2.0000000	-1.0000000	1.0000000	-1.0000000
0.1000000	-0.8048374	-0.8048375	1.9048375	-0.9048375	0.9048375	-0.9048375
0.2000000	-0.6187308	-0.6187309	1.8187309	-0.8187309	0.8187309	-0.8187309
0.3000000	-0.4408182	-0.4408184	1.7408184	-0.7408184	0.7408184	-0.7408184
0.4000000	-0.2703200	-0.2703203	1.6703203	-0.6703203	0.6703203	-0.6703203
0.5000000	-0.1065307	-0.1065309	1.6065309	-0.6065309	0.6065309	-0.6065309
0.6000000	0.0511884	0.0511881	1.5488119	-0.5488119	0.5488119	-0.5488119
0.7000000	0.2034147	0.2034144	1.4965856	-0.4965856	0.4965856	-0.4965856
0.8000000	0.3506710	0.3506707	1.4493293	-0.4493293	0.4493293	-0.4493293
0.9000000	0.4934303	0.4934300	1.4065700	-0.4065700	0.4065700	-0.4065700
1.0000000	0.6321206	0.6321202	1.3678798	-0.3678798	0.3678798	-0.3678798

$w_{6,i}$	$w_{7,i}$	$w_{8,i}$	$w_{9,i}$	$w_{10,i}$	$error_i=\|y_i-w_{1,i}\|$
1.0000000	-1.0000000	1.0000000	-1.0000000	1.0000000	0.0000000E+00
0.9048375	-0.9048375	0.9048375	-0.9048375	0.9048375	8.1964040E-08
0.8187309	-0.8187309	0.8187309	-0.8187309	0.8187309	1.4832827E-07
0.7408184	-0.7408184	0.7408184	-0.7408184	0.7408184	2.0131946E-07
0.6703203	-0.6703203	0.6703203	-0.6703203	0.6703203	2.4288185E-07
0.6065309	-0.6065309	0.6065309	-0.6065309	0.6065310	2.7471075E-07
0.5488119	-0.5488119	0.5488119	-0.5488119	0.5488121	2.9828229E-07
0.4965856	-0.4965856	0.4965856	-0.4965856	0.4965858	3.1487982E-07
0.4493293	-0.4493293	0.4493293	-0.4493292	0.4493296	3.2561721E-07
0.4065700	-0.4065700	0.4065700	-0.4065699	0.4065704	3.3145947E-07
0.3678798	-0.3678798	0.3678798	-0.3678796	0.3678803	3.3324104E-07

Figure (3.4) shows that the curve of w_1 coincides with the curve of y. This Figure shows only three of the nine remaining curves of $w_2, \ldots, w_{10}$, since $u_3' = u_5' = u_7' = u_7'$ and $u_4' = u_6' = u_8' = u_{10}'$. In addition we used circles in plotting all curves of $w_2,\ldots,w_{10}$.

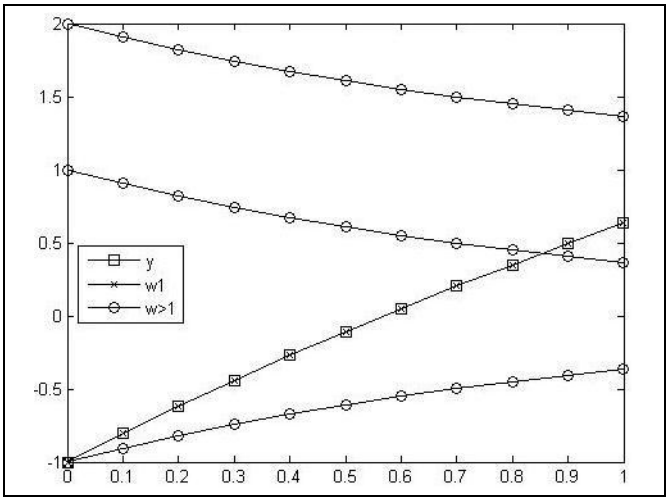


Figure (3.4): Results of Example (3.10)

Example (3.11)

We can find an approximate solution to following IVP using step size h=0.1 and Taylor of order n=15:

$$y^{(20)}(t) = y(t), \quad 0 \le t \le 2,$$

with initial conditions

$$y^{(i)}(0)=(-1)^i, \quad i=0, \ldots,19.$$

Exact solution for the IVP is

$$y(t)= e^{-t}.$$

Running Program (3.7) for this problem gives the following results:

The generated System

$$u1' = u2; \quad u1(0) = -1 \qquad u2' = u3; \quad u2(0) = 1$$

$$u3' = u4; \quad u3(0) = -1 \qquad u4' = u5; \quad u4(0) = 1$$

$$u5' = u6; \quad u5(0) = -1 \qquad u6' = u7; \quad u6(0) = 1$$

$$u7' = u8; \quad u7(0) = -1 \qquad u8' = u9; \quad u8(0) = 1$$

$$u9' = u10; \quad u9(0) = -1 \qquad u10' = u11; \quad u10(0) = 1$$

$$u11' = u12; \quad u11(0) = -1 \qquad u12' = u13; \quad u12(0) = 1$$

$$u13' = u14; \quad u13(0) = -1 \qquad u14' = u15; \quad u14(0) = 1$$

$$u15' = u16; \quad u15(0) = -1 \qquad u16' = u17; \quad u16(0) = 1$$

$$u17' = u18; \quad u17(0) = -1 \qquad u18' = u19; \quad u18(0) = 1$$

$$u19' = u20; \quad u19(0) = -1 \qquad u20' = u1; \quad u20(0) = 1.$$

Table (3.5) contains the vectors t, y, w_1 and GE. Since we are interested only on w_1, we haven't presented $w_2, \dots, w_{20}$ in Table (3.5). We still have good results, but not as good as in the previous example, since we have used order fifteen.

Table (3.5): Results of Example (3.11), using Taylor of order n=15 and step size h=0.1

t_i	y_i	$w_{1,i}$	error= $\lvert y_i - w_{1,i} \rvert$
0.0000000	1.0000000	1.0000000	0.0000000E+00
0.1000000	0.9048374	0.9048374	0.0000000E+00
0.2000000	0.8187308	0.8187308	0.0000000E+00
0.3000000	0.7408182	0.7408182	1.1102230E-16
0.4000000	0.6703200	0.6703200	0.0000000E+00
0.5000000	0.6065307	0.6065307	0.0000000E+00
0.6000000	0.5488116	0.5488116	1.1102230E-16
0.7000000	0.4965853	0.4965853	5.5511151E-17
0.8000000	0.4493290	0.4493290	5.5511151E-17
0.9000000	0.4065697	0.4065697	5.5511151E-17
1.0000000	0.3678794	0.3678794	0.0000000E+00
1.1000000	0.3328711	0.3328711	5.5511151E-17
1.2000000	0.3011942	0.3011942	0.0000000E+00
1.3000000	0.2725318	0.2725318	0.0000000E+00
1.4000000	0.2465970	0.2465970	2.7755576E-17
1.5000000	0.2231302	0.2231302	2.7755576E-17
1.6000000	0.2018965	0.2018965	2.7755576E-17
1.7000000	0.1826835	0.1826835	5.5511151E-17
1.8000000	0.1652989	0.1652989	8.3266727E-17
1.9000000	0.1495686	0.1495686	5.5511151E-17
2.0000000	0.1353353	0.1353353	5.5511151E-17

Figure (3.5) shows that the curve of w_1 coincides with the curve of y. As earlier explained, we plotted only y and w_1.

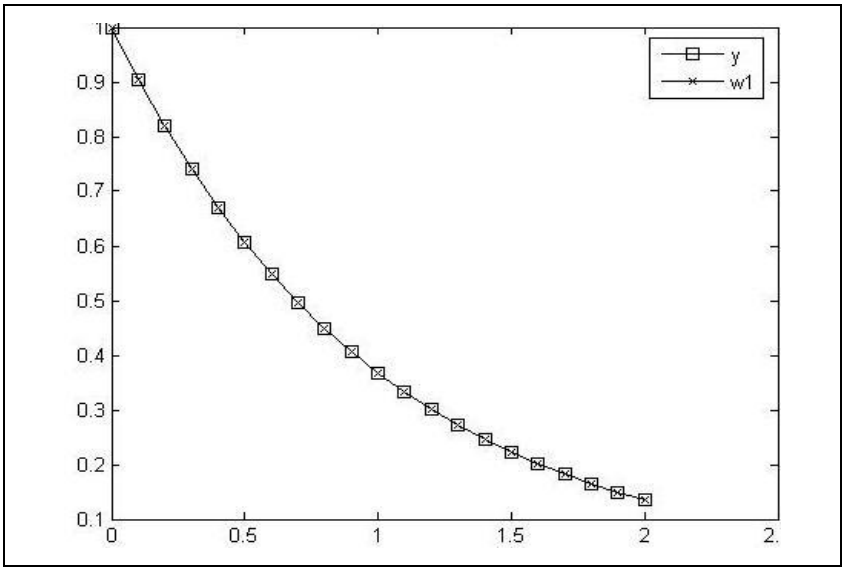


Figure (3.5): Comparison between the approximated solution w_1 and the exact solution y in Example (3.11) with n=15, h=0.1

Chapter Four

Error Analysis

4.1 Error Analysis for Numerical Methods

In this section, we will study the global error (GE) generated by fourth order numerical methods studied in chapter two. These are Taylor method, Runge-Kutta method, Adams-Bashforth method, Adams-Moulton method, predictor-corrector method and Milne's method. Using step size $h = 0.1$, we will apply all of these methods to the initial value problem in the next example.

Example (4.1)

Using step size $h = 0.1$, we can approximate the solution to the IVP

$$y'(t) = \frac{-2y}{t} + 4t, \quad 0.5 \le t \le 1.5, \quad y(0.5) = 4.25. \qquad (4.1)$$

This initial value problem has the exact solution

$$y(t) = t^2 + \frac{1}{t^2}.$$

For $t \in [0.5,1.5]$ and $h = 0.1$, we have $t_1 = 0.5$ and $t_i = 0.5 + 0.1(i-1)$, $i = 2, \dots, 11$. We will approximate the solution and compare it with the given exact solution of (4.1) at these values of t.

Table (4.1) contains the numerical results for fourth order Taylor method generated for initial value problem (4.1).

Table (4.1): Results of problem (4.1) using Taylor method

t_i	y_i	w_i	$\|GE_i\|$
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1440001	6.2222220E-03
0.7	2.5308163	2.5371852	6.3688587E-03
0.8	2.2025001	2.2080023	5.5023138E-03
0.9	2.0445678	2.0491660	4.5979898E-03
1.0	2.0000000	2.0038359	3.8358041E-03
1.1	2.0364463	2.0396702	3.2240110E-03
1.2	2.1344445	2.1371815	2.7370052E-03
1.3	2.2817159	2.2840633	2.3474416E-03
1.4	2.4702041	2.4722369	2.0328776E-03
1.5	2.6944444	2.6962206	1.7761366E-03

In Figure (4.1), we plotted the numerical results of the fourth order Taylor method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

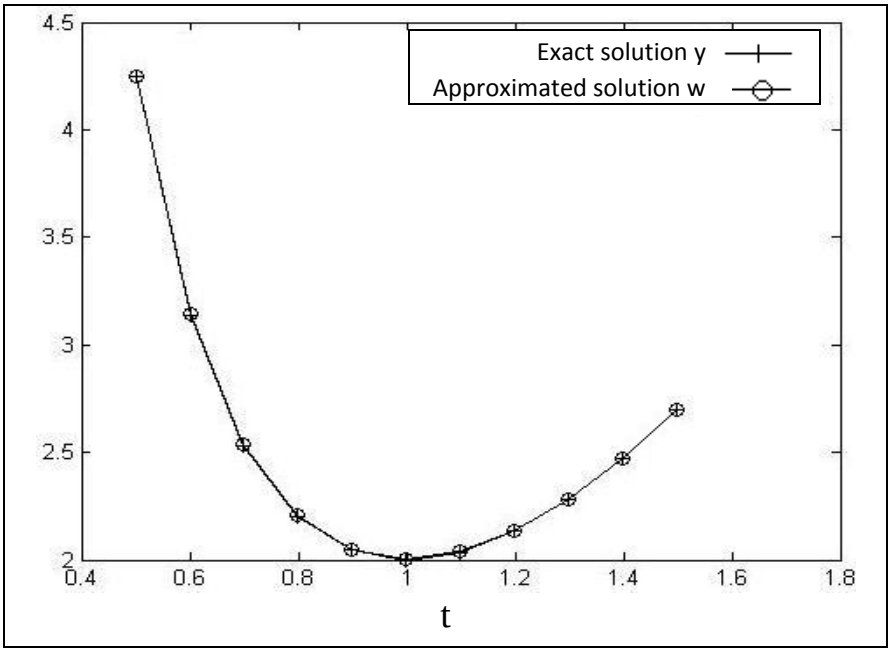


Figure (4.1): Approximate and exact solutions for problem (4.1) using fourth order Taylor method

Table (4.2) contains the numerical results for fourth order Runge-Kutta method generated for initial value problem (4.1).

Table (4.2): Results of problem (4.1) using Runge-Kutta method

t_i	y_i	w_i	GE_i
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1379659	1.8824609E-04
0.7	2.5308163	2.5310133	1.9703139E-04
0.8	2.2025001	2.2026730	1.7295216E-04
0.9	2.0445678	2.0447142	1.4627952E-04
1.0	2.0000000	2.0001233	1.2321006E-04
1.1	2.0364463	2.0365508	1.0438789E-04
1.2	2.1344445	2.1345336	8.9226342E-05
1.3	2.2817159	2.2817929	7.6985394E-05
1.4	2.4702041	2.4702711	6.7025467E-05
1.5	2.6944444	2.6945033	5.8843481E-05

In Figure (4.2), we plotted the by fourth order Runge-Kutta method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

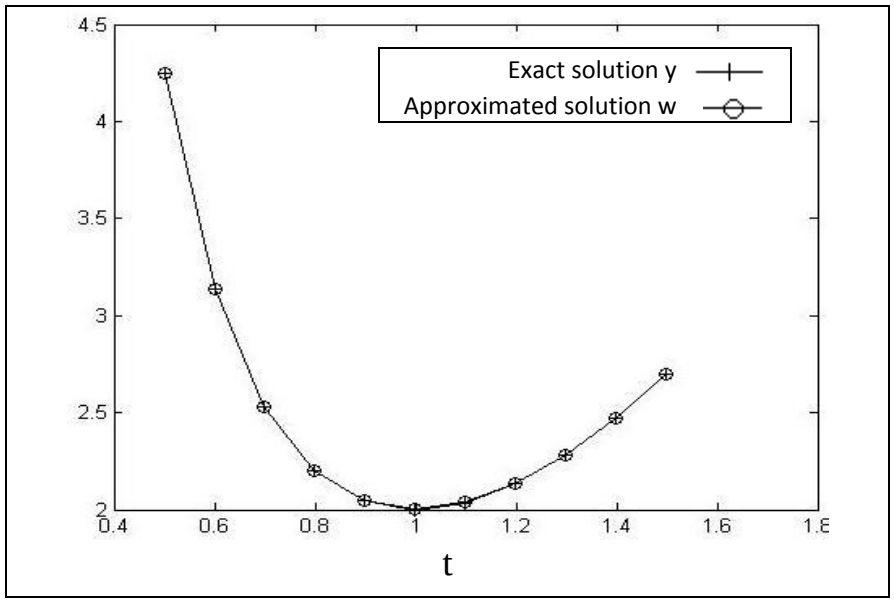


Figure (4.2): Approximate and exact solutions for problem (4.1) using fourth order Runge-Kutta method

Table (4.3) contains the numerical results for fourth order Adams-Bashforth method generated for initial value problem (4.1).

Table (4.2): Results of problem (4.1) using fourth order Runge-Kutta method

t_i	y_i	w_i	GE_i
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1379659	1.8824609E-04
0.7	2.5308163	2.5310133	1.9703139E-04
0.8	2.2025001	2.2026730	1.7295216E-04
0.9	2.0445678	2.0833945	3.8826704E-02
1.0	2.0000000	2.0335274	3.3527330E-02
1.1	2.0364463	2.0818779	4.5431580E-02
1.2	2.1344445	2.1669915	3.2546941E-02
1.3	2.2817159	2.3164694	3.4753364E-02
1.4	2.4702041	2.4965878	2.6383726E-02
1.5	2.6944444	2.7205029	2.6058473E-02

In Figure (4.3), we plotted the by fourth order Adams-Bashforth method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

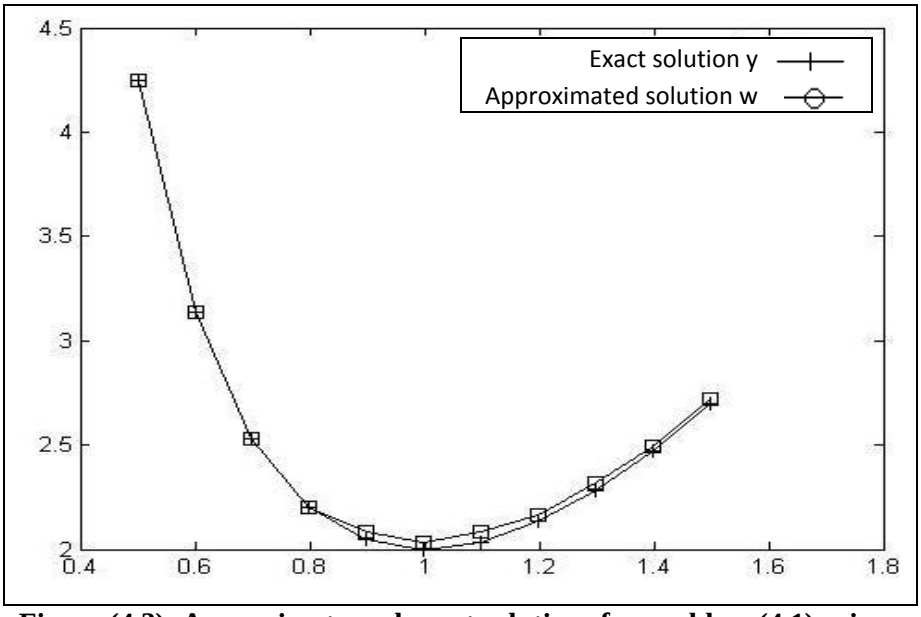


Figure (4.3): Approximate and exact solutions for problem (4.1) using fourth order Adams-Bashforth method

Table (4.4) contains the numerical results for fourth order Milne's method generated for initial value problem (4.1).

Table (4.4): Results of problem (4.1) using Milne's method

t_i	y_i	w_i	GE_i
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1379659	1.8824609E-04
0.7	2.5308163	2.5310133	1.9703139E-04
0.8	2.2025001	2.2026730	1.7295216E-04
0.9	2.0445678	2.0764439	3.1875897E-02
1.0	2.0000000	1.9933140	6.6860020E-03
1.1	2.0364463	2.0547066	1.8260250E-02
1.2	2.1344445	2.1075168	2.6927656E-02
1.3	2.2817159	2.3347795	5.3063568E-02
1.4	2.4702041	2.4275715	4.2632643E-02
1.5	2.6944444	2.7521725	5.7727974E-02

In Figure (4.4), we plotted the by fourth order Milne's method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

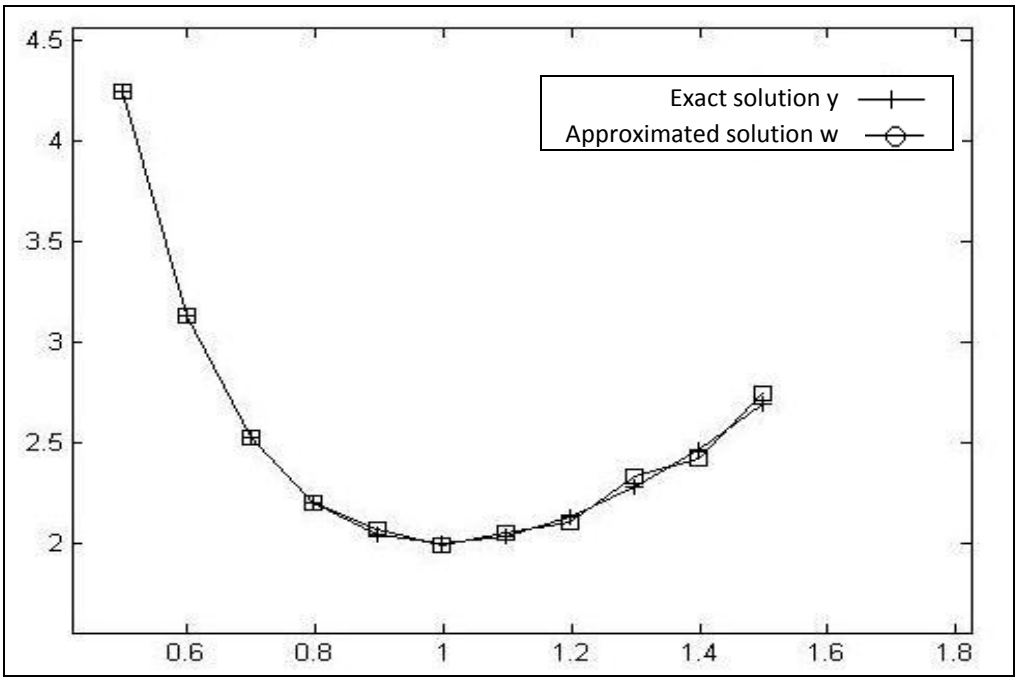


Figure (4.4): Approximate and exact solutions for problem (4.1) using fourth order Milne's method

Table (4.5) contains the numerical results for fourth order predictor-corrector method generated for initial value problem (4.1).

Table (4.5): Results of problem (4.1) using predictor-corrector method

t_i	y_i	w_i	GE_i
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1379659	1.8824609E-04
0.7	2.5308163	2.5310133	1.9703139E-04
0.8	2.2025001	2.2026730	1.7295216E-04
0.9	2.0445678	2.0401838	4.3841591E-03
1.0	2.0000000	1.9949298	5.0701834E-03
1.1	2.0364463	2.0316632	4.7831568E-03
1.2	2.1344445	2.1301129	4.3315729E-03
1.3	2.2817159	2.2778773	3.8385985E-03
1.4	2.4702041	2.4668176	3.3863666E-03
1.5	2.6944444	2.6914520	2.9923916E-03

In Figure (4.5), we plotted the by fourth order predictor-corrector method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

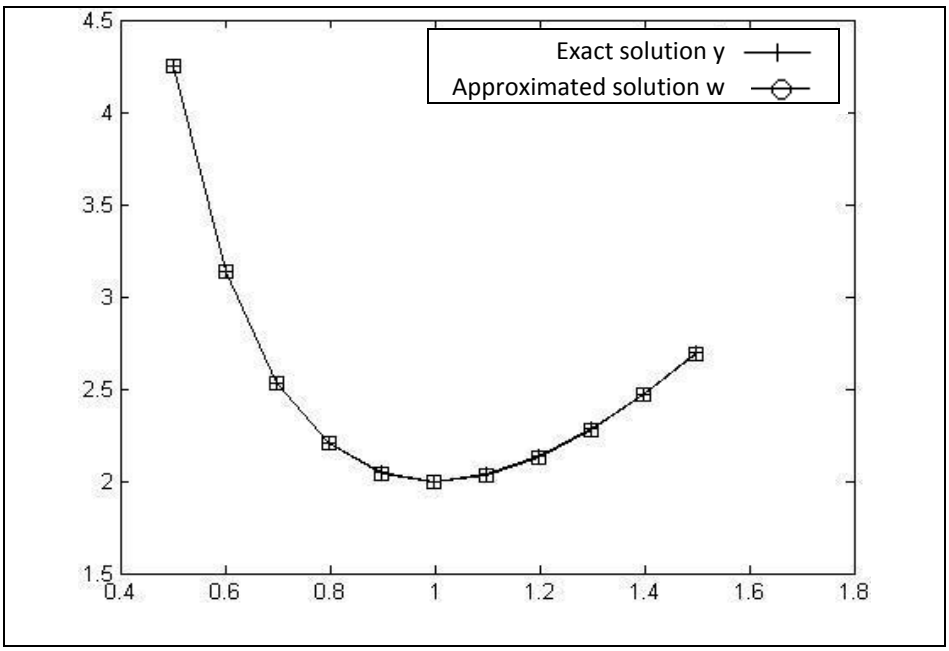


Figure (4.5): Approximate and exact solutions for problem (4.1) using fourth order predictor-corrector method

Since Adams-Moulton method is an implicit method, we had to solve manually for w_{i+1}. We found that

$$w_{i+1} = \frac{4t_{i+1}}{4t_{i+1} + 3h}\left[w_i + \frac{3ht_{i+1}}{2} + \frac{h}{24}(19f_i - 5f_{i-1} + f_{i-2})\right]$$

Table (4.6) contains the numerical results for fourth order Adams-Moulton method generated for initial value problem (4.1).

Table (4.6): Results of problem (4.1) using Adams-Moulton method

t_i	y_i	w_i	GE_i
0.5	4.2500000	4.2500000	0.0000000E+00
0.6	3.1377778	3.1379659	1.8824609E-04
0.7	2.5308163	2.5310133	1.9703139E-04
0.8	2.2025001	2.1994088	3.0913462E-03
0.9	2.0445678	2.0410907	3.4770828E-03
1.0	2.0000000	1.9966804	3.3195824E-03
1.1	2.0364463	2.0334775	2.9686904E-03
1.2	2.1344445	2.1318364	2.6081272E-03
1.3	2.2817159	2.2794333	2.2827196E-03
1.4	2.4702041	2.4682019	2.0022437E-03
1.5	2.6944444	2.6926804	1.7641560E-03

In Figure (4.6), we plotted the by fourth order Adams-Moulton method approximated solution w and the exact solution y for the initial value problem (4.1) against t.

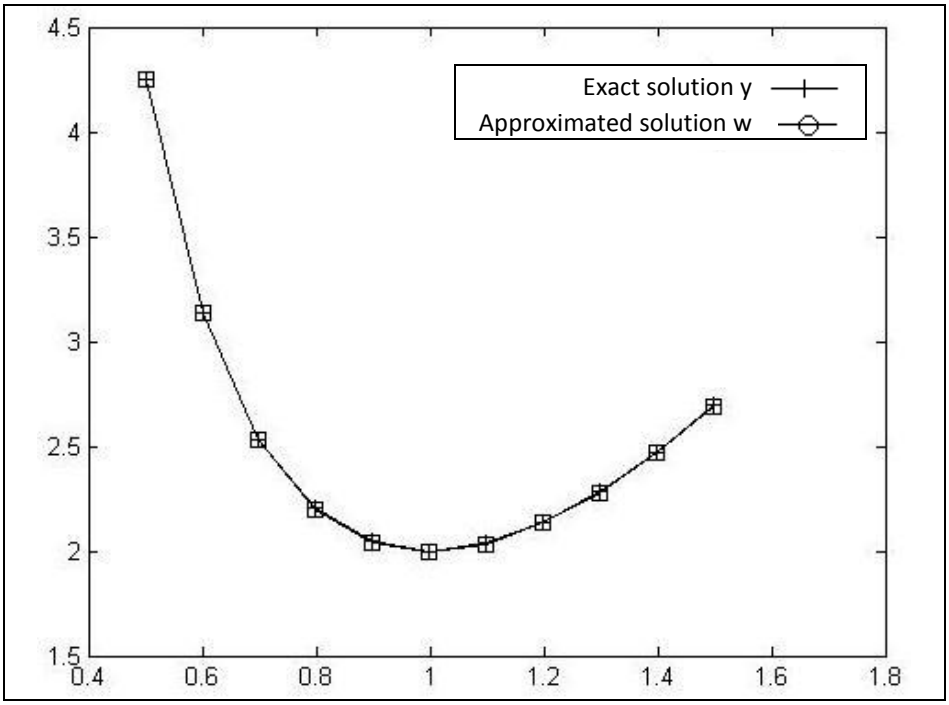


Figure (4.6): Approximate and exact solutions for problem (4.1) using fourth order Adams-Moulton method

Since we used the RK method to approximate the starting values for the multistep methods, we thought that it would be not accurate to compare them with other methods in such manner. Therefore, we modified the programs to assign exact values to w_i at the first four steps$(t = 0.5, 0.6, 0.7, 0.8)$ and let the methods approximate the solution at the remaining points.

Table (4.7) contains global errors for these six methods. Since we assigned w_1, w_2, w_3, w_4 exact values, global errors at t_1, t, t_3, t_4 will be zero for all methods.

Table (4.7): Errors generated by the methods used to solve Example (4.1)

t_i	RK	Taylor	Adams-Moulton	Predictor-Corrector	Adams-Bashforth	Milne
0.5	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.6	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.7	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.8	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.9	9.6250E-06	2.4960E-04	1.1967E-03	4.5224E-03	3.8711E-02	3.2083E-02
1.0	1.2520E-05	3.1322E-04	1.4195E-03	5.1824E-03	3.3428E-02	6.9048E-03
1.1	1.2908E-05	3.1260E-04	1.4029E-03	4.8760E-03	4.5360E-02	1.8357E-02
1.2	1.2358E-05	2.9051E-04	1.2919E-03	4.4096E-03	3.2480E-02	2.7329E-02
1.3	1.1488E-05	2.6280E-04	1.1614E-03	3.9051E-03	3.4701E-02	5.3589E-02
1.4	1.0550E-05	2.3537E-04	1.0354E-03	3.4437E-03	2.6335E-02	4.3203E-02
1.5	9.6474E-06	2.1029E-04	9.2199E-04	3.0423E-03	2.6018E-02	5.8328E-02

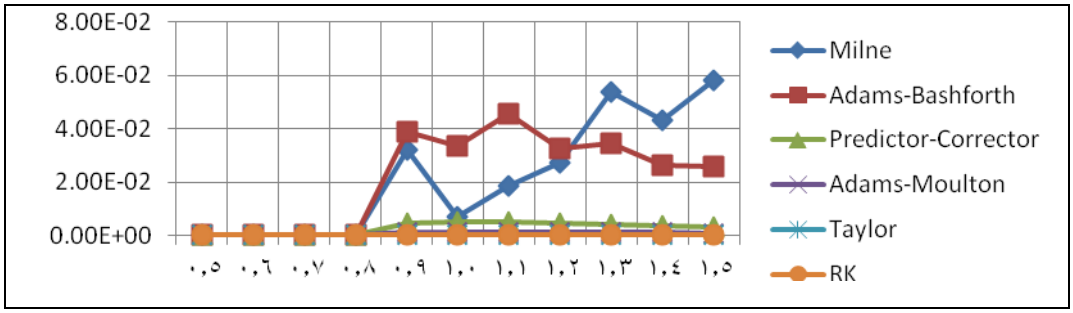


Figure (4.7): (a) Propagation of GE by the six methods in study for Example (4. 1)

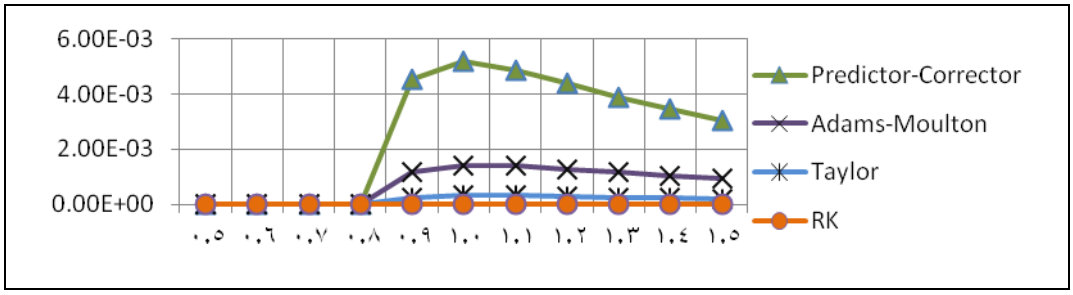


Figure (4.7): (b) Closer look into the first 4 methods with least error

Figure (4.7) (a) shows error propagation of the six methods under study. It is clear that Milne's method is the method with greatest error. To have a better comparison for the methods, we excluded in Figure (4.7) (b) the curves of

Milne's and Adams-Bashforth. Figure (4.7) (a) shows that RK method generated the least error.

To do more analysis to the data in Table (4.7), we took the absolute global errors, accumulated by using each of these six methods, at the last step $t = 1.5$, sorted them in ascending order and put them into Table (4.8) column 2. We found that RK method has the smallest error, while Milne's method has the greatest error. In addition, Table (4.8) contains the error ratios of these methods. For example, the ratio of the error generated by Adams-Bashforth method to the error generated by Adams-Moulton method is about 28:1.

Table (4.8): Error ratios for the methods under study

order	error at t=1.5	Method	Runge-Kutta	Taylor	Adams-Moulton	Predictor-Corrector	Adams-Bashforth	Milne
1	9.6474E-06	Runge-Kutta	1					
2	2.1029E-04	Taylor	21.798	1				
3	9.2199E-04	Adams-Moulton	95.569	4.384	1			
4	3.0423E-03	Predictor-Corrector	315.349	14.467	3.300	1		
5	2.6018E-02	Adams-Bashforth	2696.897	123.723	28.219	8.552	1	
6	5.8328E-02	Milne	6045.966	277.364	63.263	19.172	2.242	1

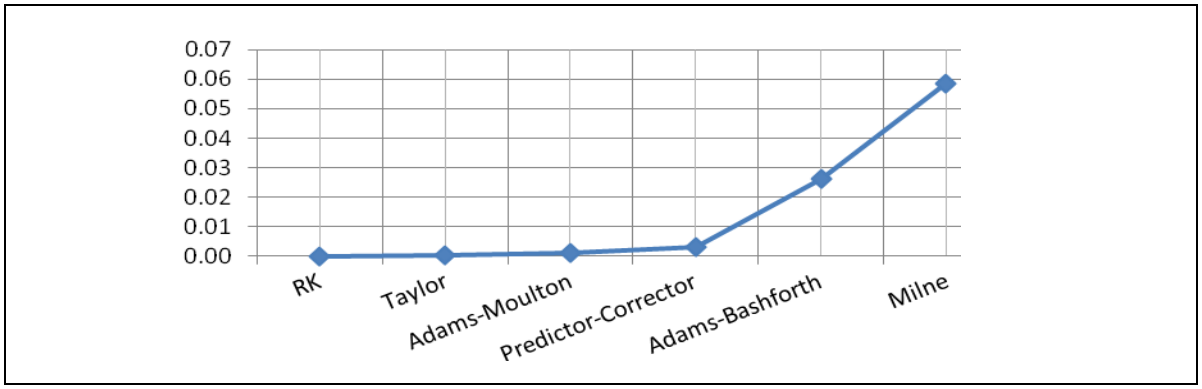


Figure (4.8): Error accumulated at the last step in the methods under study in Example (4.1)

Table (4.8) shows that the error ratio is $1:3.300:28.219$ for Adams-Moulton, Predictor-Corrector and Adams-Bashforth methods respectively. This means, using the implicit Adams-Moulton method as a corrector for the explicit Adams-Bashforth method has reduced the error by 8.552 times.

Surprisingly, both Table (4.8) and Figure (4.8) shows that one-step methods (RK and Taylor) are supreme to multistep methods of the same order.

To confirm our results, we will solve another example, using exact values at the first four steps as we done in the previous example.

Example (4.2)

Using step size $h = 0.1$, approximate the solution to the IVP

$$y'(t) = \frac{2 - 2ty}{t^2 + 1} + 4t, \quad 0 \le t \le 1, \quad y(0) = 1.$$

This initial value problem has the exact solution

$$y(t) = \frac{2t + 1}{t^2 + 1}.$$

We represented global errors generated by the methods in Table (4.9), plotted this data in Figure (4.9) and calculated error ratios in Table (4.10).

Table (4.9): Global error generated in Example (4.2)

t_i	Adams-Bashforth	Milne	Predictor-Corrector	Adams-Moulton	Taylor	RK
0.5	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.6	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.7	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.8	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.9	1.9204E-05	3.9552E-05	2.5176E-05	2.4057E-05	1.1170E-05	4.3123E-07
1.0	3.9721E-04	3.4727E-04	7.3954E-05	6.0292E-05	2.3500E-05	8.4084E-07
1.1	8.4929E-04	3.9845E-04	1.2262E-04	9.1653E-05	3.2397E-05	1.1590E-06
1.2	1.2234E-03	3.5059E-04	1.5530E-04	1.0972E-04	3.6408E-05	1.3584E-06
1.3	1.4192E-03	2.0552E-04	1.6775E-04	1.1415E-04	3.6282E-05	1.4462E-06
1.4	1.4510E-03	3.8881E-04	1.6374E-04	1.0876E-04	3.3558E-05	1.4472E-06

1.5	1.3660E-03	2.7794E-04	1.4979E-04	9.8060E-05	2.9654E-05	1.3900E-06

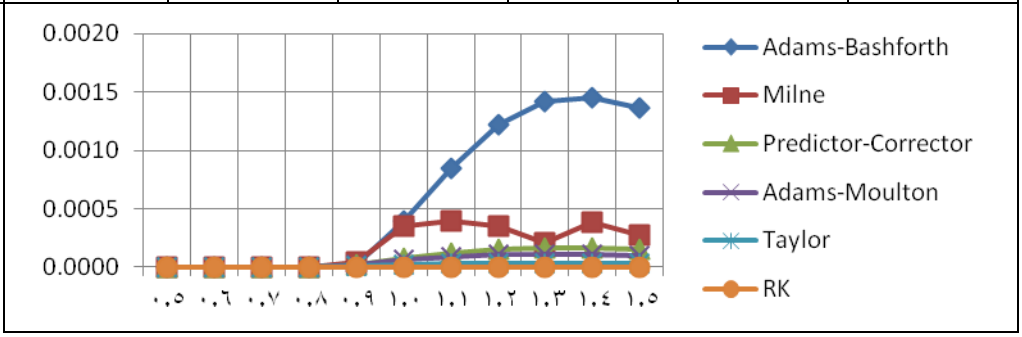


Figure (4.9): (a) Propagation of GE by the methods under study for Example (4.2)

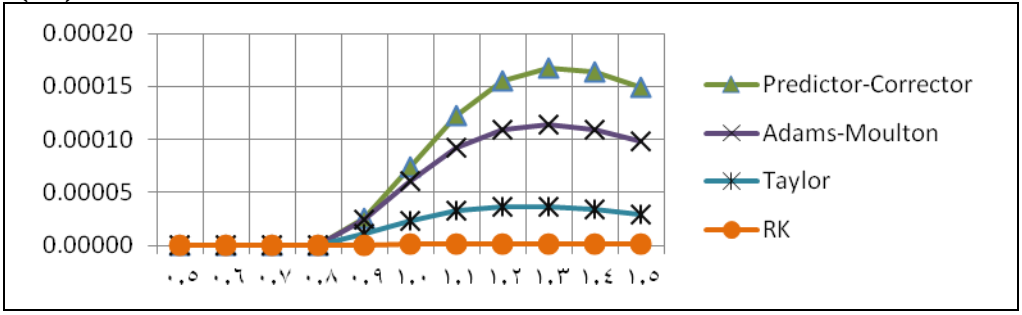


Figure (4.9): (b) Closer look into the first 4 methods with least error

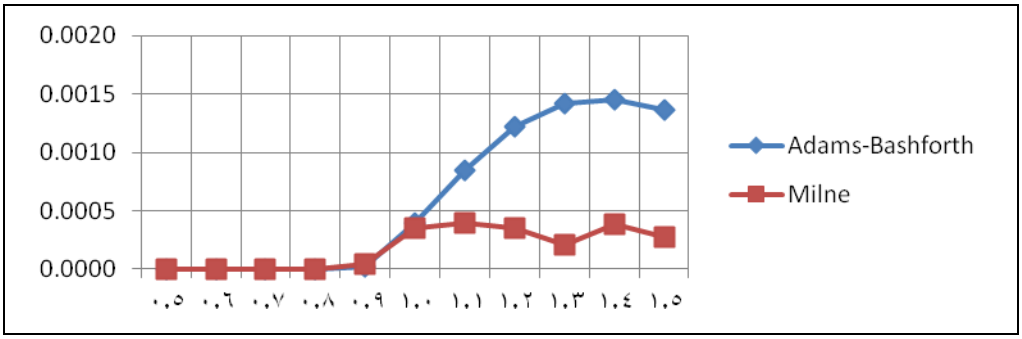


Figure (4.9): (c) Closer look into the 2 methods with greatest error

Table (4.10): Error ratios of the methods under study for Example (4.2)

order	error at t=1.5	method	RK	Taylor	Adams-Moulton	Predictor-Corrector	Milne	Adams-Bashforth
1	1.39E-06	RK	1					
2	2.97E-05	Taylor	21.334	1				
3	9.81E-05	Adams-Moulton	70.547	3.307	1			
4	1.50E-04	Predictor-Corrector	107.761	5.051	1.528	1		
5	2.78E-04	Milne	199.956	9.373	2.834	1.856	1	
6	1.37E-03	Adams-Bashforth	982.730	46.064	13.930	9.120	4.915	1

Table (4.10) and Figure (4.9) confirm the results we got in Example (4.1)

with one exception; Adams-Bashforth method has generated the greatest error instead of Milne's method.

Finally, we will compare these methods according to CPU time. We measured CPU time for these methods used in Example (4.2) with different step sizes and represented the results in Table (4.11). We found that the CPU time for the Taylor method is the highest, and that is due to the time cost of constructing Taylor expansion of $f(t,y(t))$. We noted also, that with decreasing step size the CPU time differences between Taylor method and other methods decreases in favor of Taylor method.

Table (4.11) Comparing CPU time

Step size	RK CPU time/s	Taylor CPU time/s	Adams-Moulton CPU time/s	Predictor-Corrector CPU time/s	Milne CPU time/s	Adams-Bashforth CPU time/s
h=0.5	0.0017888	0.0481321	0.0017486	0.0020422	0.002092	0.0022194
h=0.2	0.0049428	0.049220	0.0041848	0.0079606	0.005129	0.0060896
h=0.1	0.0127556	0.052088	0.0103906	0.0215922	0.010619	0.0155052
h=0.05	0.0290058	0.057133	0.0228816	0.0477258	0.024353	0.0305398

4.2 Error by Higher Order Taylor Methods

In this section, we will compare global error generated by different higher order Taylor methods $(\text{Taylor}4,\dots,\text{Taylor}10)$. We will see how error is reduced by increasing Taylor's order.

Table (4.11) contains errors accumulated after each step using Taylor methods $n=4,\dots 10$. We plotted this data in Figure (4.10). Figure (4.10) contains the methods (a) for $n=4,\dots 6$, (b) for $n=6,\dots 8$ and (c) for $n=8,\dots 10$. We intentionally repeated the method $n=6$ in (b) and the method $n=8$ in (c), to have better comparison.

Table (4.12): Global error generated by Taylor (n=4,…,10) methods

t_i	Taylor4	Taylor5	Taylor6	Taylor7	Taylor8	Taylor9	Taylor10
0.5	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00
0.6	1.881E-05	1.188E-06	1.881E-07	1.188E-08	1.881E-09	1.188E-10	1.881E-11
0.7	2.682E-05	3.179E-06	2.273E-07	3.248E-08	1.854E-09	3.232E-10	1.442E-11
0.8	2.228E-05	4.810E-06	1.167E-07	4.421E-08	2.971E-10	3.790E-10	3.108E-12
0.9	9.768E-06	5.356E-06	3.884E-08	4.104E-08	1.056E-09	2.780E-10	1.087E-11
1.0	4.070E-06	4.882E-06	1.432E-07	2.994E-08	1.341E-09	1.628E-10	6.638E-12
1.1	1.454E-05	3.897E-06	1.708E-07	1.966E-08	9.182E-10	1.120E-10	5.711E-13
1.2	2.011E-05	2.885E-06	1.469E-07	1.380E-08	4.060E-10	1.098E-10	1.982E-12
1.3	2.147E-05	2.096E-06	1.056E-07	1.163E-08	9.640E-11	1.193E-10	1.796E-12
1.4	2.014E-05	1.574E-06	6.816E-08	1.124E-08	1.170E-11	1.221E-10	8.697E-13
1.5	1.751E-05	1.266E-06	4.195E-08	1.126E-08	1.164E-11	1.166E-10	1.859E-13

Generally, the higher the Taylor's order, the better results we get. Exceptions are possible. We see that the results for $n = 8$ are better than for $n = 9$.

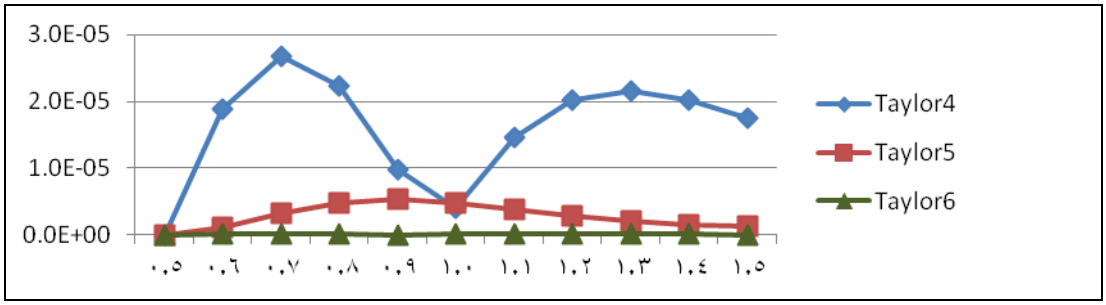


Figure (4.10) (a) Error propagation for Taylor Methods n=4,…,6 for Example (4.2)

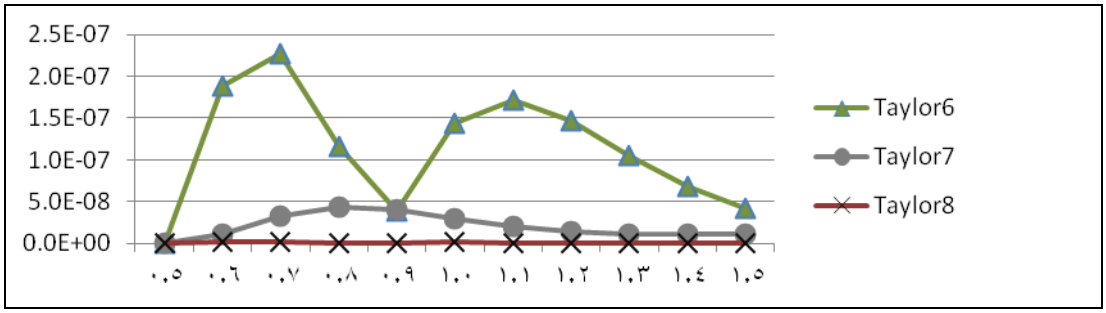


Figure (4.10): (b) Error propagation for Taylor Methods n=6,…,8 for Example (4.2)

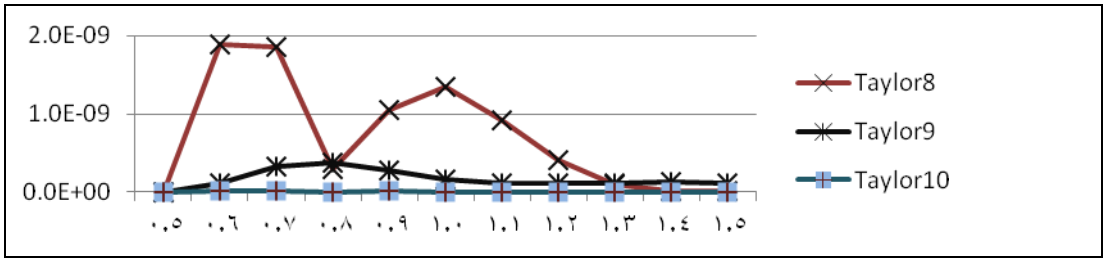


Figure (4.10): (c) Error propagation for Taylor Methods n=8,…,10 for Example (4.2)

Finally, we can say that Taylor methods are a good choice to approximate the solution of initial value problem ordinary differential equations, since with algorithms we developed, all we need is to enter a higher value of n to get more accurate solutions to a single first order IVP, a system of first order IVP's or a higher order IVP's.

4.3 Conclusions

The main aim of this thesis was to develop algorithms to get highly accurate approximations to the solution of initial value problem

$$y'(t) = f(t,y), \quad a \leq t \leq b, \quad y(a) = \alpha,$$

and to study the concepts of stability and error propagation when a numerical method is applied to an initial value problem. Through our work, we found that higher order Taylor's methods give highly accurate approximations. However, the main obstacle was the tedious work of finding higher order derivatives of the initial value problem. We managed to write algorithms to find higher order derivatives of initial value problems and to construct Taylor's methods for solving these problems numerically. We also managed to develop these algorithms to deal in the same way with systems of first order initial value problems. In addition, we wrote an algorithm to convert higher order initial value problems into systems of first order initial value problems and to solve these systems. We translated all of these algorithms into Matlab programs. For these programs, all we need is to enter the problem, choose the desired order (n), and step size (h).

We wrote also algorithms, which we translated into Matlab programs for the fourth order Runge-Kutta, Adams-Bashforth, Adams-Moulton, Milne's and predictor-corrector (4-step fourth order Adams-Bashforth method as predictor and 3-step fourth order Adams-Moulton method as corrector) methods. We used these programs to compare errors generated by these methods and the fourth order Taylor's method. We found that single step methods (Runge-Kutta and Taylor) generated the best results. Since we can increase accuracy of Taylor's methods only by selecting a higher value of n, we concluded that with the programs we developed, higher order Taylor's methods could be a good choice for approximating the solution of initial value problems.

We used the test initial value problem (2.29)

$$y'(t) = \lambda y(t), \qquad t > 0, \quad y(0) = 1, \quad \lambda \in \mathbb{C},$$

to study absolute stability of Taylor's methods. For this purpose, we wrote a Matlab program to find the error amplification functions and to plot the boundaries of stability regions of any Taylor's method.

To compare stability of Taylor's methods with other single step methods, we also wrote the Matlab Program (2.11) to find the error amplification functions and to plot the boundaries of stability regions of any explicit Runge-Kutta method. In addition, we wrote the Matlab Program (2.9) to plot the boundaries of stability regions of any explicit method given we have the error amplification functions. Matlab Program (2.10) plots the boundaries of

stability regions of implicit methods (only of orders less than five) given we have the error amplification functions.

Using results of these programs, we found that Taylor's methods, compared with other explicit methods, have similar stability regions. The most part of these stability regions lies at the imaginary axis and in the left part of the complex plane. That means, with enough small step size h and with non positive real part of λ, Taylor' methods are stable.

Finally, we conclude that we can get higher accuracy of a wide range of initial value problems applying Taylor's methods to them.

References

[1] F. Bashforth, J. Adams, **An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid, with an explanation of the method of integration employed in constructing the tables which give the theoretical forms of such drops**, Cambridge University Press, Cambridge, 1883.

[2] R. Buck, **Advanced Calculus**, McGraw-Hill, 1978.

[3] R. Burden and J. Faires, **Numerical Analysis**, Brooks/Cole, Cengage Learning, 2011.

[4] R. Burden and J. Faires, **Numerical Methods**, Brooks Cole, 2002.

[5] J. Butcher, "***A history of Runge-Kutta methods"***, **Applied Numerical Mathematics**, **20**, 1996/ 247-260.

[6] J. Butcher, "***Numerical methods for ordinary differential equations in the 20th century***", **Journal of Computational and Applied Mathematics, 125,** 2000/ 1-29.

[7] J. Butcher, **Numerical Methods for Ordinary Differential Equations**, John Wiley & Sons Ltd,2008.

[8] I. Danaila, P. Joly, S. Kaber and M. Postel, **An Introduction to Scientific Computing**, Springer Science & Business Media, LLC, 2007.

[9] D. Griffiths and D. Higham, **Numerical Methods for Ordinary Differential Equations Initial Value Problems**, Springer-Verlag London Limited, 2010.

[10] S. Iyengar and R. Jain, **Numerical Methods**, New Age International (P) Ltd., Publishers, 2009.

[11] W. Kutta, *"**Beitrag zur Näherungsweisen Integration Totaler Differentialgleichungen"***, **Z. Math. Phys. , 46,** 1901/ 435-453.

[12] R. LeVeque, **Finite Difference Methods for Ordinary and Partial Differential Equations**, Society for Industrial and Applied Mathematics, 2007.

[13] W. Milne, ***"A note on the numerical integration of differential equations"***, **J. Res. Nat. Bur. Standards, 43,** 1949/ 537-542.

[14] F. Moulton, **New Methods in Exterior Ballistics**, University of Chicago, Chicago, 1926.

[15] C. Runge, ***"Über die numerische Auflösung von Differentialgleichungen",* Math. Ann. , 46,** 1895/ 167-178.

[16] E. Süli and D. Mayers, **An Introduction to Numerical Analysis**, Cambridge University Press, 2003.

[17] E. Swokowski, **Calculus with Analytic Geometry**, Prindle, Weber & Schmidt, 1979.

Appendices

Appendex (A)

My Matlab Programs

```
function Euler_2_1
% ================================================
% Entering The IVP
%================================================
a=1;
b=2;
y(1)=1;
fyp=inline('y/t-(y/t)^2','t','y');  %y'=f(t,y)
fy=inline('t/(log(t)+1)','t'); %Exact Solution
h=.1;
%================================================
% Initializing For Euler
%================================================
N=(b-a)/h;
t(1)=a;
w(1)=y(1);
wHat(1)=y(1);
%================================================
% Euler Method
%================================================
for i=1:N;
w(i+1)=w(i)+h*fyp(t(i),w(i));% Approximated Solution
wHat(i+1)=y(i)+h*fyp(t(i),y(i));% w hat
t(i+1)=t(i)+h;
y(i+1)=fy(t(i+1));% Exact y
end
%================================================
% Finding Global And LTEs
%================================================
gEr=abs(y-w);% Absolute Value of Global Error
locEr=abs(y-wHat);% Absolute Value of LTE
%================================================
% Displaying And Plotting The Results
%================================================
format long
disp(single([t' y' w' gEr' wHat' locEr' ]))
plot(t,y,'k+-',t,w,'ks-')
legend('y','w')
```

Program (2.1) Euler's Method

```
function ExplicitRK
%===============  IVP  ============================
f=inline('-2*t*y^2','t','y');% y'(t)
a=0;  b=1;% end points
alpha=1; % initial condition
%================================================
yEx=inline('1/(1+t^2)','t');% Actual solution y(t)
h=.1;% Step size
%================= Butcher tableau ================
s=4
A=[ 0   0   0  ;
  1/2   0   0  ;
   0   1/2  0  ;
   0    0   1 ];
B=[1/6 1/3 1/3 1/6];
```

```
C=[0 1/2 1/2 1];
%==================================================

w(1)=alpha;
y(1)=alpha;
t(1)=a;
N=(b-a)/h;
%================== RK method ======================
for n=1:N
 sum=0;
   for i=1:s;
      m=0;
      for j=1:i-1
        m=m+A(i,j)*k(j);
      end
      k(i)=h*f(t(n)+C(i)*h,w(n)+m);
      sum=sum+B(i)*k(i);
   end
 w(n+1)=w(n)+sum;
 t(n+1)=t(n)+h;
 y(n+1)=yEx(t(n+1));
end

error=abs(y-w);

 format long ;
 plot(t,y,'k+-',t,w,'ko-')
 legend('y','w')
 out=single([t' y' w' error'])
 xlswrite('test.xls',out,'sheet1','d2');
```

Program 2.2 Fourth order Runge-Kutta method

```
function CoeffAdam_Bashforth
syms s
k=10;
x=1;
y(1)=sym('1');
fac=1;
for i=1:1:k
   x=x*(s+i-1);
   fac=fac*i;
  y(i+1)=(int(x,0,1)/fac );
end
y'
```

Program (2.3) Coefficients of Adam-Bashforth methods

```
function Adams_Bashforth
f=inline('y-t^2+1','t','y');
yEx=inline('(t+1)^2-0.5*exp(t)','t');
a=1; b=3
alpha=4-0.5*exp(1);
h=.1

N=(b-a)/h
t(1)=a
w(1)=alpha;
y(1)=alpha;

for i=1:3
    k1=f(t(i),w(i))
    k2=f(t(i)+.5*h,w(i)+.5*h*k1)
    k3=f(t(i)+.5*h,w(i)+.5*h*k2)
```

```
    k4=f(t(i)+h,w(i)+h*k3)
    w(i+1)=w(i)+(h/6)*(k1+2*k2+2*k3+k4)
    t(i+1)=t(i)+h
    y(i+1)=yEx(t(i+1))
end

for i=4:N
   w(i+1)=w(i)+h/24*(55*f(t(i),w(i))-59*f(t(i-1),...
     w(i-1))+37*f(t(i-2),w(i-2))-9*f(t(i-3),w(i-3)))
    t(i+1)=t(i)+h
    y(i+1)=yEx(t(i+1))
end
 error=abs(y-w);

format long
out=single([t' y' w' error' ]);
disp(out)
plot(t,y,'k+-',t,w,'ks-')
legend('y','w')
xlswrite('test.xls',out,'sheet1','d2')
```

Program (2.4) Fourth order Adams-Bashforth method

```
function CoeffMilne
syms s x
k=4;
x=1;
y(1)=sym('4');
fac=1;
for i=1:1:k
   x=x*(s+i-1);
   fac=fac*i;
  y(i+1)=(int(x,-3,1)/fac );
end
y'
```

Program (2.5) Coefficients of the four step Milne's method

```
function Milne
f=inline('y-t^2+1','t','y');
yEx=inline('(t+1)^2-0.5*exp(t)','t');
a=1; b=3
alpha=4-0.5*exp(1);
h=.1

N=(b-a)/h
t(1)=a
w(1)=alpha;
y(1)=alpha;

for i=1:3
    k1=f(t(i),w(i))
    k2=f(t(i)+.5*h,w(i)+.5*h*k1)
    k3=f(t(i)+.5*h,w(i)+.5*h*k2)
    k4=f(t(i)+h,w(i)+h*k3)
    w(i+1)=w(i)+(h/6)*(k1+2*k2+2*k3+k4)
    t(i+1)=t(i)+h
    y(i+1)=yEx(t(i+1))
end

for i=4:N
   w(i+1)=w(i-3)+4*h/3*(2*f(t(i),w(i))-f(t(i-1),w(i-1))...
           +2*f(t(i-2),w(i-2)))
    t(i+1)=t(i)+h
end
```

```
format long
out=single([t' y' w' error' ]);
disp(out)
plot(t,y,'k+-',t,w,'ks-')
legend('y','w')
xlswrite('test.xls',out,'sheet1','d2')
```

Program (2.6) Fourth order Milne's method

```
function CoeffMoulton
syms s x
k=10;
x=1;
y(1)=sym('1');
fac=1;
for i=1:1:k
   x=x*(s+i-2);
   fac=fac*i;
  y(i+1)=(int(x,0,1)/fac );
end
y'
```

Program (2.7) Coefficients of .Adams-Moulton methods

```
function Moulton
f=inline('y-t^2+1','t','y');
yEx=inline('(t+1)^2-0.5*exp(t)','t');
a=1; b=3
alpha=4-0.5*exp(1);
h=.2
N=(b-a)/h
t(1)=a
w(1)=alpha;
y(1)=alpha;
for i=1:2
   k1=f(t(i),w(i))
   k2=f(t(i)+.5*h,w(i)+.5*h*k1)
   k3=f(t(i)+.5*h,w(i)+.5*h*k2)
   k4=f(t(i)+h,w(i)+h*k3)
   w(i+1)=w(i)+(h/6)*(k1+2*k2+2*k3+k4)
   t(i+1)=t(i)+h
   y(i+1)=yEx(t(i+1))
end
for i=3:N
     t(i+1)=t(i)+h
    w(i+1)=24/(24-9*h)*(w(i)+h/24*(9*(-(t(i+1))^2+1)...
              +19*f(t(i),w(i))-5*f(t(i-1),w(i-1))+f(t(i-2),w(i-2))))
  y(i+1)=yEx(t(i+1))
end
error=abs(y-w);
format long
out=single([t' y' w' error' ]);
disp(out)
plot(t,y,'k+-',t,w,'ks-')
legend('y','w')
xlswrite('test.xls',out,'sheet1','d2')
```

Program (2.8) 3-step Adams-Moulton method

```
function OtherExplicitStabilityRegions
syms   z th Gz
%=======================================================
Gz=1+z+z^2/2+z^3/6+z^4/24+z^5/120;

%=======================================================
 HHH=sym2poly(Gz);
 order=length(HHH)-1;
 HHH=sym(HHH);
 HHH(order+1)=HHH(order+1)-exp(i*th);

%================   Finding roots and plottig =========
count=0;
for th=0:.01:2*pi
       clear z
       z=roots(eval(HHH));
       for j=1:length(z)
         count=count+1;
         y(count)=(z(j));
       end
end
hand =plot(real(y),imag(y),'k.');
set(hand, 'MarkerSize', 5);
hold all
```

Program (2.9) Plots the boundary of stability regions of explicit methods

```
function ImplicitStabilityRegions
syms   z th
%=======================================================

Gz='z*(1/(z-1))';

%=======================================================

B=strcat(Gz,'-exp(i*th)=0');
s=0;
for th=0:.1:2*pi
       clear z
       z=solve(B,'z');
       for j=1:length(z)
         s=s+1;
         y(s)=eval(z(j));
       end
end
hand =plot(real(y),imag(y),'k.');
set(hand, 'MarkerSize', 5);
```

Program (2.10) Plots the boundary of stability regions for implicit methods

```
function ExplicitRK_StabilityRegions
syms   sum m z th
%===============   IVP  ============================
f=inline('z*y','y','z');% y'(t)
s=3;
%================ Butcher tableaux ================

switch s
  case 1
B=[1];
case 2
A=[1/2];
B=[0 1];% midpoint method
case 3
A=[1/2   0 ;
   -1   2];
B=[1/6 2/3 1/6];
```

```
case 4
A=[1/2   0   0  ;
   0   1/2  0  ;
   0    0   1 ];
B=[1/6 1/3 1/3 1/6];
end
%====================== Find Gz ========================
format rat
 Gz=1;
   for p=1:s;
      m=0;
      for j=1:p-1
       m=m+A(p-1,j)*k(j);
      end
      k(p)=f(1+m,z);
      Gz=Gz+B(p)*k(p);
   end
char(simplify(Gz))

 HHH=sym2poly(Gz);
 HHH=sym(HHH);
 HHH(s+1)=HHH(s+1)-exp(i*th);

%================  Finding roots and plottig =========
count=0;
for th=0:.01:2*pi
       clear z
       z=roots(eval(HHH));
       for j=1:length(z)
         count=count+1;
         y(count)=(z(j));
       end
end
hand =plot(real(y),imag(y),'k.');
set(hand, 'MarkerSize', 5);
hold all
```

Program (2.11) Finds G(z) for RK1,...,RK4 and plots the boundary of the stability regions

```
function TaylorStabilityRegions

kk=5;
kkk=8;

%============ ====  Finding G(z)  =============
for k=kk:kkk
n=k;
clear B
Gzz=sym('Gzz');
clear th
th=sym('th','real');
Gzz(n+2)=1-exp(i*th);
fact=1;
for j=1:n+1
   fact=fact*j;
   Gzz(n+2-j)=1/fact;
end
Gzz

%=============Finding the roots and plotting===========
s=0;
for th=0:.05:2*pi
       clear z
       z=roots(eval(Gzz));
       for j=1:length(z)
         s=s+1;
         y(k,s)=(z(j));
       end
```

```
end
end
colors = hsv(kkk);
for v=kk:kkk
hand =plot(real(y(v,:)),imag(y(v,:)),'.','color',colors(v,:));
legendmatrix{v-kk+1,1}=strcat('Taylor ',num2str(v));
hold on
end
set(hand, 'MarkerSize', 5);
legend(legendmatrix)
```

Program (2.12) Finds G(z) for any Taylor method and plots the boundary of stability regions

```
1  function Taylor_3_1
2  syms t y
3  %============================================
4  % This Program finds the first n derivatives
5  %  of y, yp(i)refers to y prime(i)
6  %============================================
7  n=input('n=');
8  yp(1)=input('y''=');  %ODE y'(t,y).
9  % To find the derivatives of y'
10 for i=2:n
11    pt=diff(yp(i-1),t);
12    py=diff(yp(i-1),y);
13    yp(i)=simplify(pt+py*yp(1));
14 end
15 yp
```

Program (3.1) Finds the first n derivatives of y in $y' = f(t,y)$

```
1  function Taylor_3_2
2  syms t y h
3  %=================================================================
4  %This Program finds the first n derivatives of y' yp(i)
5  %refers to y prime(i)and construct Taylor series of order n
6  %=================================================================
7  n=input('n=');
8  yp(1)=input('y''=');  %ODE y'(t,y).
9  %=================================================================
10 %Finding the derivatives of y'and constructing Taylor expantion
11 %=================================================================
12 fac=1;
13 T=yp(1);
14 for i=2:n
15    pt=diff(yp(i-1),t);
16    py=diff(yp(i-1),y);
17    yp(i)=simplify(pt+py*yp(1));
18    fac=fac*i;
19    T=T+h^(i-1)/fac*yp(i);
20 end
21 yp
22 T
```

Program (3.2) Finds the first n derivatives of y in $y' = f(t,y)$ and constructs $T^{(n)}$

```
1   function Taylor_3_3
2   syms t y h
3   %=====================================================================
4   %Entering the IVP and saving it with the ability to solve it with
5   %different order and step size of Taylor method
6   %=====================================================================
7   Q=input('Do you want to enter a new problem y/n: ','s');
8   if Q=='y'|Q=='Y'
9   yp(1)=input('ODE y''(t)= ');% yp(i)is the i-th derivative of y
10  yExact=inline(input('exact solution of ODE y(t)=','s'));
11  a=input('start of interval a= ');
12  b=input('end of interval b= ');
13  yN(1)=input('initial condition y(a)= ');
14    save data_mat yp yExact a b  yN
15  else
16     load data_mat yp yExact a b  yN
17  end
18  n=input('Taylor order n=');
19  hN=input('step size h= ');
20  %=====================================================================
21  % Constructing Taylor Expansion of f(t,y)
22  %=====================================================================
23  fac=1; %factorial
24  T=yp(1);% Taylor series
25  for i=2:n
26     pt=diff(yp(i-1),t);% partial derivative of y prime(i-1)
27                  % with respect to t
28     py=diff(yp(i-1),y);% partial derivative of y prime(i-1)
29                  % with respect to y
30     yp(i)=simplify(pt+py*yp(1));% construct y prime(i)
31     fac=fac*i;
32     T=T+h^(i-1)/fac*yp(i);% construct Taylor series
33  end
34  %=====================================================================
35  %Taylor Method
36  %=====================================================================
37  N=(b-a)/hN;
38  tN(1)=a;
39  w(1)=yN(1);
40  h=hN;
41  for i=1:N;
42     t=tN(i);
43     y=w(i);
44     w(i+1)=w(i)+h*eval(T);
45     tN(i+1)=tN(i)+h;
46     yN(i+1)=yExact(tN(i+1));
47  end
48  error=abs(yN- w)
49  format long
50  plot(tN,yN,'k+-',tN,w,'ko-')
51  legend('y','w')
52  out=single([tN'  yN' w' error'])
53  xlswrite('test.xls',out,'sheet1','d2')
54
```

Program (3.3) Finds the first n derivatives of y in $y' = f(t,y)$, constructs $T^{(n)}$ and solves the IVP

```
function SystDerivFind
syms t
%=================================================================
%Entering the System of First Order  ODEs and saving it for
%further use
%=================================================================
Q=input('Do you want to enter a new problem y/n: ','s');
if Q=='y'|Q=='Y'
   k=input('No. of Equations k=');
  for i=1:k
     u(i)=sym(strcat('u',num2str(i)),'real');
  end
  for z=1:k
      up(z,1)=input( strcat('ODE',num2str(z),' u',num2str(z),...
              'prime(t)='));
  end
  save data_mat up  u k
else
    load data_mat up u  k
end
%=================================================================
%Finding the derivatives of (uj)'
%=================================================================
n=input('Taylor order n=');
for i=2:n
    for j=1:k
      up(j,i)= diff(up(j,i-1),t);
       for p=1:k
        pu(j,p)=diff(up(j,i-1),u(p));% partial derivative of
                              %u prime(i-1) with respect to u
        up(j,i)= up(j,i)+pu(j,p)*up(p,1);% construct u prime(i)
       end
    end
end
disp(up)
```

Program (3.4) Finds the first n derivatives every y_j in $y_j' = f_j(t, y)$ and constructs $T_j^{(n)}$

```
function SystTaylorConstr
syms t  h
%=========================================================
%Entering the System of First Order  ODEs and saving it for
%further use
%=========================================================
Q=input('Do you want to enter a new problem y/n: ','s');
if Q=='y'|Q=='Y'
   k=input('No. of Equations k=');
  for j=1:k
     u(j)=sym(strcat('u',num2str(j)),'real');
   end
     for j=1:k
       up(j,1)=input( strcat('ODE',num2str(j),' u',num2str(j),...
               'prime(t)=') );% up(i)is the i-th derivative of
     end
  save data_mat up  u k
else
 load data_mat up u k
end
%=================================================================
%Finding the derivatives of u(j)' and constructing Taylor expansion
%=================================================================
n=input('Taylor order n=');
for j=1:k
T(j)=up(j,1);% Taylor series
end
fac=1; %factorial
for i=2:n
```

```
    fac=fac*i;
    for j=1:k
     up(j,i)= diff(up(j,i-1),t);
       for p=1:k
         pu(j,p)=diff(up(j,i-1),u(p));% partial derivative of
                        % u prime(i-1) with respect to u
         up(j,i)= up(j,i)+pu(j,p)*up(p,1);% construct u prime(i)
       end
     T(j)=T(j)+h^(i-1)/fac*simplify(up(j,i));
    end
end
disp(T)
```

Program (3.5) Finds the first n derivatives every y_j in $y_j' = f_j(t, y)$ and constructs $T_j^{(n)}$

```
function SystTaylorSolve
syms t  h
%=========================================================================
%Entering the IVP system and saving it with the ability to
%solve it with different Taylor order and step size
%=========================================================================
Q=input('Do you want to enter a new problem y/n: ','s');
if Q=='y'|Q=='Y'
   k=input('No. of Equations k=');
  for i=1:k
     u(i)=sym(strcat('u',num2str(i)),'real');
    end
    for z=1:k
      up(z,1)=input( strcat('ODE',num2str(z),' u',num2str(z),...
             'prime(t)=') );% up(i)is the i-th derivative of
    end
    for e=1:k
      uN(e,1)=input(strcat('initial condition u',num2str(e),'(a)= '));
    end
for i=1:k
 uExact(i)=input(strcat('exact solution of ODE u',num2str(i),'(t)='));
end
 a=input('start of interval a= ');
 b=input('end of interval b= ');
 save data_mat up  a b u k uN uExact
else
   load data_mat up u a b k  uN uExact
end
n=input('Taylor order n=');
hN=input('step size h =');
%=========================================================================
%Finding the derivatives of u' and constructing Taylor expansion
%=========================================================================
fac=1; %factorial
for i=1:k
T(i)=up(i,1);% Taylor series
end
for i=2:n
   fac=fac*i;
   for j=1:k
     pt(j)=diff(up(j,i-1),t);% partial derivative of u prime(i-1)
     up(j,i)=pt(j);
      for p=1:k
       pu(j,p)=diff(up(j,i-1),u(p));% partial derivative of
                        %u prime(i-1) with respect to u
       up(j,i)= up(j,i)+pu(j,p)*up(p,1);% construct u prime(i)
      end
     T(j)=T(j)+h^(i-1)/fac*simplify(up(j,i));% construct Taylor series
   end
end
D=cell(1,k);
for i=1:k
```

```
  D(i)=(T(i));
  for j=k:-1:1
  D{i}=strrep(D{i},strcat('u',num2str(j)),strcat('v(',num2str(j),')'));
  end
 end
%=====================================================================
%Taylor Method
%=====================================================================
N=(b-a)/hN;
tN(1)=a;
w(1:k,1)=uN(1:k,1);
h=hN;
 t=tN(1);
for i=1:N;
   v(1:k)=w(1:k,i);
   for s=1:k
   w(s,i+1)=w(s,i)+h*eval(D{s});
   end
   tN(i+1)=tN(i)+h;
   t=tN(i+1);
   for s=1:k
       uN(s,i+1)=eval(uExact(s));
      (s,i+1)=abs(uN(s,i+1)-w(s,i+1));
   end
 error=abs(uN-w);
end
 out=[tN]';
 for i=1:k
  out=[out uN(i,:)' w(i,:)' error(i,:)'];
 end
format long e
 single(out)
plot(tN,uN(1,:),'ko-',tN,w(1,:),'k+-',tN,uN(2,:),'ks-',...
     tN,w(2,:),'kx-')
legend('u1','w1','u2','w2')
xlswrite('test.xls',out,'sheet1','d2')
```

Program (3.6) Finds the first n derivatives every y_j in $y_j' = f_j(t, y)$ and constructs $T_j^{(n)}$ and solves the system of IVP

```
% ===============================================================
%     THIS PROGRAM APPROXIMATES IVP ODE's OF ORDER k  BY
%    CONVERTING THE ODE INTO A SYSTEM OF FIRST ORDER ODE's
% ===============================================================
function TaylorForHighOrderODEs
syms t
%===============================================================
%              Entering The IVP  ODE
%     Transforming It Into A System Of First Order ODEs
%       Saving It With The Ability To Solve
%   It with Different Order and step size of Taylor method
%===============================================================
Q=input('Do you want to enter a new problem y/n: ','s');

if Q=='y'|Q=='Y'
   k=input('order of ODE=');

      for j=1:k
       u(j)=sym(strcat('u',num2str(j)),'real');
```

```
20      end
21
22      y=u(1);
23      for j=2:k
24        yp(j-1)=u(j);
25      end
26
27      for j=1:k-1
28        up(j,1)=u(j+1);
29      end
30      up(k,1)=input( strcat('ODE y prime',...
31                      num2str(k),'(t)=') );
32
33      uN(1)=input(strcat('initial condition y','(a)= '));
34      for e=2:k
35        uN(e)=input(strcat('initial condition y prime',...
36                        num2str(e-1),'(a)= '));
37      end
38
39   uExact=input(strcat('exact solution of ODE y(t)='));
40
41   a=input('start of interval a= ');
42   b=input('end of interval b= ');
43
44   save data_mat u up  a b k uN    uExact
45  else
46    load data_mat u up  a b k  uN  uExact
47  end
48  % ================================================================
49  %           Displaying The Generated System
50  % ================================================================
51  disp('The generated System')
52  for j=1:k
53   sys=['u' num2str(j) '''=' char(up(j))];
54   init=[';   u'  num2str(j) '(' num2str(a) ')=' num2str(uN(j))];
55
56  out1=strcat(sys,init);
57  disp(out1)
58  end
59  %================================================================
60  %           Constructing Taylor expansion
61  %================================================================
62  n=input('Taylor order n=');
63  h=input('step size h= ');
64  fac=1; %factorial
65  for i=1:k
66  T(i)=up(i,1);% Taylor series
67  end
68  for i=2:n
69   fac=fac*i;
70    for j=1:k
71     pt(j)=diff(up(j,i-1),t);% partial derivative of u prime(i-1)
72     up(j,i)=pt(j);
73
74     for p=1:k
75      pu(j,p)=diff(up(j,i-1),u(p));% partial derivative of
76                          %u prime(i-1) with respect to u
77        up(j,i)= up(j,i)+pu(j,p)*up(p,1);% construct u prime(i)
78     end
79    T(j)=T(j)+h^(i-1)/fac*simplify(up(j,i));%construct Taylor
80     end
81  end
82
83  D=cell(k,1);
84  for i=1:k
85     D(i)=(T(i));
86     for j=k:-1:1
87        D{i}=strrep(D{i},strcat('u',num2str(j)),...
88                    strcat('v(',num2str(j),')'));
```

```
   end
end
%=============================================================
%               Taylor Method
%=============================================================
N=(b-a)/h;
tN(1)=a;
w(1:k,1)=uN(1:k);
t=tN(1);
yN(1)=uN(1);
for i=1:N;
   v(1:k)=w(1:k,i);
   for s=1:k
   w(s,i+1)=w(s,i)+h*eval(D{s});
   end
   tN(i+1)=tN(i)+h;
   t=tN(i+1);
   yN(i+1)=eval(uExact);
   error(i+1)=abs(yN(i+1)-w(1,i+1));
end
%=============================================================
%               Results Output
%=============================================================
out=[tN' yN' w' error'];
format short
single(out)
plot(tN,yN,'ks-',tN,w(1,:),'kx-',tN,w(2:k,:),'ko-')
legend('y','w1', 'w>1')
xlswrite('test.xls',out,'sheet1','d2')

```

Program (3.7) Solves higher order IVP's

جامعة النجاح الوطنية

كلية الدراسات العليا

تحليل الأخطاء والثباتية للطرق العددية لحل مسائل القيم الابتدائية

إعداد

عماد عمر فارس كايد

إشراف

أ. د. ناجي قطناني

قدمت هذه الاطروحة استكمالا لمتطلبات الحصول على درجة الماجستير في الرياضيات المحوسبة بكلية الدراسات العليا في جامعة النجاح الوطنية في نابلس، فلسطين.

2013

تحليل الأخطاء والثباتية للطرق العددية
لحل مسائل القيم الابتدائية

إعداد
عماد عمر فارس كايد
إشراف
أ. د. ناجي قطناني

الملخص

مسائل القيم الابتدائية في معظمها هي ظواهر طبيعية كتبت بلغة الرياضيات. إن حل مسائل القيم الابتدائية هو أحد اكثر حقول الرياضيات تحديا بسبب الرغبة المستمرة للدقة عند علماء الرياضيات.

يركز هذا العمل بشكل أساسي على تطوير خوارزميات وبرامج لتكوين طرق تيلر العليا لتقريب حل مسائل القيم الابتدائية من الدرجة الاولى ولتقريب حل انظمة مسائل القيم الابتدائية من الدرجة الاولى ولتقريب حل مسائل القيم الابتدائية من الدرجات العليا. بالاضافة إلى ذلك يركز هذا العمل على دراسة الأخطاء والثباتية للطرق العددية لحل مسائل القيم الابتدائية. ولهذا الغرض قمنا بتطوير برامج لايجاد اقترانات تكبير الاخطاء لطرق تيلر وطرق رنجي-كتا الصريحة ولرسم حدود مناطق الثباتية لهذه الطرق وطرق أخرى.

لقد استنتجنا انه وباستخدام البرامج التي طورناها يمكن أن تكون طرق تيلر العليا خيارا جيدا لتقريب حلول مجموعة واسعة من مسائل القيم الابتدائية.