

COMPACT NUMERICAL METHODS FOR STIFF
DIFFERENTIAL EQUATIONS

By

EDWARD PURBA

Sarjana S-1

Bandung Institute of Technology

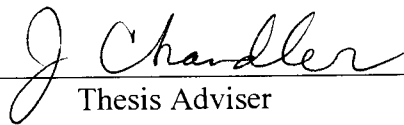
Bandung, Indonesia

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1996

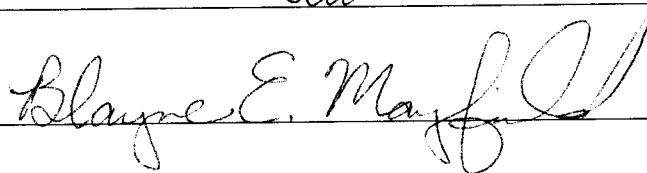
COMPACT NUMERICAL METHODS FOR STIFF
DIFFERENTIAL EQUATIONS

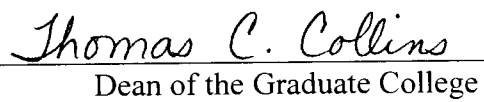
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGMENTS

First, I would like to express my sincere appreciation to my major adviser, Dr. John P. Chandler, for providing continuous support in finishing this thesis, and helping me with his intelligent supervision, constructive guidance, and insightful critiques and suggestions from the early until the late stages of the works. My sincere appreciation is also addressed to my other committee members Dr. Blayne E. Mayfield and Dr. Huizhu Lu, for their cooperation, friendship, and assistance.

So many people have helped me in completing this thesis, it is impossible to acknowledge them all personally. I have gained much value from my relationship with H. Sinaga during my study at Oklahoma State University, and encouragement from my friends including Y. Sofyan, T. Rahardjo and Dr. Robert A. Divall.

I also am indebted to the Government of Indonesia, who sponsored my graduate study through the Ministry for Research and Technology, in the framework of the implementation of the Science And Technology for Industrial Development. Many thanks to Prof. Dr. Ing. B.J. Habibie, State Minister of Research and Technology, Prof. S. Sapiie D.Sc., Mrs. Ir. Ina Juniarti, Vice President for IPTN Computing Center, and Mr. Ir. Muriawan A. Kadir.

Finally, I would like to dedicate this thesis to my mother, late father and late sister Ida Purba.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION -----	1
1.1 Initial Value Problem for First-Order ODEs -----	2
1.2 The Existence and Uniqueness of Solution of Initial Value Problems ---	3
1.3 Taylor's Series -----	4
1.4 Stability and Instability of ODEs -----	6
1.5 Convergence of Euler Method -----	9
1.6 The Multistep Concept -----	12
II. STABILITY ANALYSIS -----	16
2.1 Stability of Euler Methods -----	18
2.2 Stability of Explicit Runge-Kutta Methods -----	24
2.3 Stability of Implicit Runge-Kutta Methods -----	30
2.4 Difference Equations-----	31
III. STABILITY OF MULTISTEP METHODS -----	35
3.1 Linear Difference Equations -----	38
3.2 Adams Methods -----	44
3.3 Backward Differentiation Formula (BDF) -----	48
3.4 Predictor-Corrector Methods -----	51
3.5 Extrapolation Methods for Solving ODEs -----	53
IV. STIFF ORDINARY DIFFERENTIAL EQUATIONS -----	62
4.1 Stiff Differential Problems -----	66
4.2 Stiffness Concepts -----	68
4.3 Stability Concepts of Numerical Stiffness Methods -----	72
4.4 A Modified Euler Method for Solving ODEs -----	75
4.5 An Explicit Exponential Method for Solving ODEs -----	79
4.6 ODE Solvers -----	83
V. ANALYSIS AND DISCUSSION -----	86
5.1 Implementation of a Modified Euler Method -----	87

Chapter	Page
5.2 Implementation of an Exponential Method -----	90
5.3 Analysis of Kidney Problems -----	92
5.4 Analysis of Autocatalytic Problems -----	95
5.5 Analysis of Problem D4 of Enright et al. -----	97
5.6 Analysis of Gupta and Wallace's Problem -----	99
 VI. CONCLUSION AND SUGGESTION -----	 101
 SELECTED BIBLIOGRAPHY -----	 103
 APPENDIXES -----	 116
APPENDIX A-- SOFTWARE AVAILABILITY -----	116
APPENDIX B -- ROOTS OF A COMPLEX POLYNOMIAL -----	120
APPENDIX C -- LINEAR MULTISTEP FORMULAS -----	128
APPENDIX D-- ESTIMATION OF ERROR AND STEPSIZE CONTROL -----	137
APPENDIX E -- STABILITY REGION OF SIMPLE PREDICTOR- CORRECTOR METHODS -----	148
APPENDIX F -- COLLECTION OF PROGRAMS -----	152
APPENDIX G-- COLLECTION OF TABLES -----	174

LIST OF TABLES

Table	Page
2.1. Stability Region of Runge-Kutta for Orders 1 to 10 -----	28
2.2 Butcher's Array of Runge-Kutta Methods -----	29
2.3 Butcher's Array of a Fourth-Order Runge-Kutta method -----	30
3.1 A Trial Solution to Find a Particular Solution of Nonhomogeneous Difference Equations -----	42
3.2 Table of Extrapolation-----	56
3.3 Romberg Representation of $y' = -y, y(0) = 1$ -----	59
3.4 The Computation Result of $y' = -y, y(0)=1$, Using Extrapolation, Euler, and ABM-4 Methods -----	60
G.1 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of Kidney Problem -----	174
G.2 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of Autocatalytic Reaction Pathway Problem -----	184
G.3 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of D4 of Enright et al. -----	186
G.4 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of problem proposed by Gupta and Wallace-----	188
G.5 The Computation Results for Problem 1 of Chapter 4.4 Using Modified Euler Method and Exact Solution -----	190
G.6 The Computation Results for Problem 2 of Chapter 4.4 Using Modified Euler Method and Mathematica Packages-----	191

Table	Page
G.7 The Computation Results for Problem 1 of Chapter 4.4 Using Exponential Method and the Exact Solution -----	192
G.8 The Computation Results for Problem 2 of Chapter 4.4 Using Exponential Method and Mathematica Packages -----	193

LIST OF FIGURES

Figure	Page
1.1 Solution $y = (3+3\varepsilon/5)e^{-2t} + 2\varepsilon/5e^{3t}$ for $\varepsilon = 0, 0.01, \text{ and } 0.1$ -----	7
1.2 Crude Euler Method -----	10
1.3 Numerical Results Using Exact, Euler, and Trapezoidal Methods -----	11
2.1 Stability Region of the Explicit Euler Method -----	19
2.2 Stability Region of the Implicit Euler Method -----	19
2.3 Euler Method with Step size $h = 0.01, 0.03, 0.04, \text{ and } 0.05$ -----	21
2.4 Crude Euler, Implicit Euler Using PC, and Implicit Euler Using Newton's ---	23
2.5 Implicit Euler by Showing the Vector Gradient -----	24
2.6 A Second Order of Runge-Kutta -----	26
2.7 Stability Region of Second order Runge-Kutta -----	27
2.8 Difference Equation Methods Using Forward, Backward and Central for $h = 0.05, 1, 2, \text{ and } 3$.-----	34
3.1 Computational Results of $y_{n+2} = -4y_{n+1} + 5y_n + h(4f_{n+1} + 2f_n)$ Applied to $y' = y, y(0) = 1$ -----	43
3.2 Computation Results of $y' = -y, y(0) = 1$ Using Extrapolation, Euler, and ABM-4 Methods -----	61
4.1 Graphical Solutions of e^{-t} and e^{-100t} -----	64
4.2 A Representation of a Minimum Region of A-Stability -----	73
4.3 A Representation of a Minimum Region of $A(\alpha)$ -Stability-----	74

Figure	Page
4.4 A Representation of a Minimum Region of A_0 -Stability -----	74
4.5 A Representation of a Minimum Region of a Stiffly-Stable Method -----	75
5.1 Problem 1 of Section 4.4 Using Modified Euler Method -----	88
5.2 Problem 2 of Section 4.4 Using Modified Euler Method -----	89
5.3 Problem 1 of Section 4.4 Using Exponential Method -----	90
5.4 Problem 2 of Section 4.4 Using Exponential Method -----	91
5.5 Kidney Problems with $\lambda = 902688359, 0.0990283499,$ 0.9925211341, and 1.0304879856 -----	92
5.6 Kidney Problems with $\lambda = 0.99, 0.9,$ and 0-----	93
5.7 Output of Mathematica for Robertson Problem-----	95
5.8 Robertson Problem Solved with EPSODE -----	96
5.9 Problem D4 of Enright et al. Solved with Mathematica -----	97
5.10 Problem D4 of Enright et al. Solved with MEBDF -----	98
5.11 Gupta and Wallace's Problem Solved with Mathematica -----	99
5.12 Gupta and Wallace's Problem Solved with VODE -----	100
B-1 Stability Region of Runge-Kutta Methods Orders 1, 2, 3, 4, and 5 -----	127
E-1 A PC-Method Where a Trapezoidal as a Corrector and an Euler as a Predictor -----	150
E-2 A PC-Method Where the Implicit Euler Method as Corrector and Crude Euler as a Predictor -----	151

LIST OF SYMBOLS AND TERMS

Symbols	Meanings
$<$	less
\leq	less or equal
$>$	greater
\geq	greater or equal
\neq	not equal
$ $	absolute value
$ $	such that
$\{ \}$	set
\subset	subset
∞	infinity
min	minimum
max	maximum
$!$	factorial
Σ	summation
lim	limit
$x \rightarrow b$	x approaches b
∇	backward difference

Symbols	Meanings
$\text{Re}(z)$	the real part of a complex number z
$\text{Im}(z)$	the imaginary part of a complex number z
$\text{arg}(z)$	argument of a complex number z
\cup	union
\gg	far greater
$\partial/\partial t$	partial derivative over t
$O()$	order
$[]^T$	matrix transpose
\Re	set of real numbers

CHAPTER I

INTRODUCTION

Before computers were involved in human lives, scientific practice only recognized two terminologies; theory and experiment. However, after computers become involved in human lives, computational science becomes one of the terminologies in scientific practice and stands beside of the others as an essential methodology. It is undeniable that most parts of science and engineering can be modeled by mathematical equations. Unfortunately, most of mathematical equations are not easy to solve exactly, so we need to approximate their solutions. The methods used to approximate these solutions are called numerical methods. Scientific computing starts by transforming the real-life problems into mathematical models. In this case, essential features of scientific problems are represented in the form of mathematical equations. The second step is to modify the mathematical models, so that they are suitable for numerical computations. In this stage, usually we have to discretize the domain of problems into steps of computation, so that the formulations can be executed numerically either by computers or by human brains manually, using the operators $+$, $-$, $*$, and $/$. The third stage is to test and validate the solutions. This can be done by: comparing with the exact solutions, comparing with previously validated computations, comparing with laboratory experiments, and analyzing the convergences, the errors, and the diagnostic data. Finally,

the tested and validated codes are ready to use for various scientific and engineering problems.

Some scientific and engineering problems are modeled by ordinary differential equations such as $y' = f(t,y)$, $y(t_0) = y_0$. Among them are chemical reactions (chemistry), heat-flow problems (thermodynamics), electrical circuits (electrical engineering), force problems (mechanics), rate of bacterial growth (biological science), decompositions of radioactive material (atomic physics), population growth (statistics), and simulation and control systems (control engineering). Due to difficulties in finding the exact solutions of ordinary differential equations (ODEs), the study of numerical solution of ODEs becomes important.

The choice of numerical methods used for the approximations depends on the accuracy needed and the characteristic of the problem. Since every n th-order of ODE can be transformed into a system of first-order ODEs, here the discussion will be confined only to systems of first-order ODEs. For a similar reason, the discussion will also be limited to initial value problems (IPVs).

1.1 Initial Value Problem for First-Order ODEs

The general form of an initial value problem for a first-order ODE can be written as $y' = f(t,y)$, where an initial value $y(a)$ is given and t is in the interval $[a,b]$. Every n th-order ODE in the form of $d^n y/dt^n = y^{(n)} = f(t,y,y^{(1)},y^{(2)}, \dots, y^{(n-1)})$ with initial conditions $y^{(i)}(a) = c_i$, $i=0,1, \dots, n-1$, can be converted into a system of first-order ODEs of the form $dy_i/dt = f_i(t,y_1,y_2, \dots, y_n)$, $y_i(a) = c_i$, $i=1,2, \dots, n-1$ by substituting $y_1=y$, $y_2=dy_1/dt, \dots$

.. $y_n = dy_{n-1}/dt$. For example, a third-order IPV $\frac{d^3 y}{dt^3} + \left(\frac{dy}{dt}\right)^2 + 3y = e^t$, $y(0) = 1$, $y'(0) = 0$, $y''(0) = 0$ can be converted into an initial-value problem for the variables y , dy/dt , and d^2y/dt^2 by substituting $y_1=y$, $y_2=dy/dt$, and $y_3=d^2y/dt^2$. Applying these new variables will transform the third-order ODE into a system of first-order ODEs that can be written as

$$\frac{dy_1}{dt} = y_2, \quad \frac{dy_2}{dt} = y_3, \quad \frac{dy_3}{dt} = e^t - y_2^2 - 3y_1$$

where $y_1(0)=0$, $y_2(0)=0$, $y_3(0)=0$. After transforming the problem into a system of first-order ODEs, our problem becomes how to solve this system of first-order ODEs using numerical computing methods. Solving the first-order ODE will lead us to nonstiff or stiff differential systems. These problems affect the stability of the methods used to solve the system of first-order ODEs.

1.2 The Existence and Uniqueness of Solutions of Initial Value Problems

The first interesting question in handling first-order ODEs is to ask about the existence of a solution. By investigating the existence of a solution, we do not waste our time trying to solve a problem that has no solution in a given domain. An understanding of Lipschitz conditions plays an important role in proving the existence and the uniqueness of solutions of first-order ODEs.

A function $f(t,y)$ is said to satisfy a Lipschitz condition with respect to y in a region $D \subset \mathbb{R} \times \mathbb{R}$, if for all (t,y) and (t,z) in D , there exists a constant $L > 0$ such that $|f(t,y) - f(t,z)| \leq L|y-z|$. Here, the constant L is called a Lipschitz constant for f . For example, let $f(t,y) = 1 + y^2$ for $D = \{(t,y) \mid |t| < 1, |y| < 1\}$. It is easy to see that

$|f(t,y)-f(t,z)| = |1+y^2 - (1+z^2)| = |y^2-z^2| = |y+z||y-z|$. Since $|t| < 1$ and $|y| < 1$, than we can take $L=2$ as a Lipschitz constant for f .

Many authors in the field of Differential Equations have proved the existence of solutions of first-order ODEs. Usually, a constructive solution of a first-order ODE can be accomplished using Picard iteration. Braun [10] said that if f and $\partial f / \partial y$ are continuous in the rectangle $D=\{(t,y) \mid t_0 \leq t \leq t_0+a, |y - y_0| \leq b \}$, then the initial value problem $y'=f(t,y)$, $y(t_0)=y_0$ has at least one solution $y(t)$ on the interval $t_0 \leq t \leq t_0+\alpha$, where $\alpha = \min\{a,b/M\}$ and $M = \max_{(t,y) \in D} |f(t,y)|$. Shampine and Gordon [99] used the terminology of Lipschitz condition to show a necessary condition for the initial value problem $y'=f(t,y)$ to have a solution on a given domain.

The second interesting question in handling first-order ODEs is to ask about the uniqueness of solution. The guarantee for uniqueness of solutions becomes important, especially when we want to find an approximate solution for the problem. Otherwise, our computation may never converge. Shampine and Gordon [99] claim that if $f(t,y)$ is continuous and satisfies a Lipschitz condition on an open region $D=\{(t,y) \mid a \leq t \leq b, -\infty < y < \infty \}$, then the problem $y'=f(t,y)$, $y(a)=A$ has a unique solution for all intervals $[a,b]$. Braun [10] used the same premises as those in the existence solution, in order to show the uniqueness of a solution of a first-order ODE.

1.3 Taylor's Series

Newton and Taylor has introduced a monumental concept of approximating a function using a polynomial generated involving the derivatives of the function. This

method of approximation has been used in many applications. Unfortunately, in order to gain a cheap cost of computation, this method is rarely used by itself in solving problems. But, some methods in ODE computations such as Euler, Runge-Kutta, etc. use this Taylor's series expansion. By some mathematical manipulations, the need to calculate derivatives occurring in Taylor's series can be omitted. The expansion of a function f using Taylor's n -series needs the condition that f should have $n+1$ continuous derivatives. Suppose $f(t)$ has $n+1$ continuous derivatives in $[a,b]$. For a point t , and c in $[a,b]$, the Taylor expansion of $f(t)$ around c is given as

$$f(t) = f(c) + f'(c)(t-c) + f^{(2)}(c)(t-c)^2/2! + \dots + f^{(n)}(c)(t-c)^n/n! + R_{n+1}(\xi),$$

where $R_{n+1}(\xi) = f^{(n+1)}(t-c)^{n+1} / (n+1)!$, and ξ is a point between t and c . Taylor's expansion of function f of two variables t and y , where f and all its partial derivatives of order up to $n+1$ are continuous in a neighborhood of the point (a,b) is given as

$$f(t,y) = f(a,b) + \sum_{1 \leq r+s \leq n} \frac{\partial^{r+s} f(a,b)}{\partial t^r \partial y^s} \frac{(t-a)^r}{r!} \frac{(y-b)^s}{s!} + R_{n+1},$$

where

$$R_{n+1} = \sum_{r+s=n+1} \frac{\partial^{r+s} f(\xi,\eta)}{\partial t^r \partial y^s} \frac{(t-a)^r}{r!} \frac{(y-b)^s}{s!},$$

with the value (ξ,η) situated on the line segment joining the points (a,b) and (t,y) .

Barton [8] has used this method to solve stiff problems. He equipped the method with five devices: a device for estimating the local error, a device for predicting a stepsize h , the use of predictor and corrector formulas, a device for deciding the order of Taylor series to be used, and a device to detect when a transient has no effect on the solution.

Corliss and Chang [24] have developed software, ATSMCC (Automatic Taylor Series by Morriss, Chang, and Corliss), using this Taylor approach. The software can be used to solve nonstiff systems and moderately stiff systems of initial value problems of ODEs.

1.4 Stability and Instability of ODEs

All numerical methods are proposed to approximate the true solutions of mathematical problems that are not easy to solve. In order to have accurate solutions, the methods should be stable, so we can approximate the true values by controlling the errors. When trying to solve an ODE problem, sometimes we are facing with instabilities of solutions. The instabilities are not only caused by the approximation method of solution, but are caused by the ODE problem itself. In general, we can classify instabilities of ODEs solutions into two categories: inherent instability and induced instability.

Inherent instability is an instability caused merely by the property of the differential system, and not the method of solution, while induced instability is an instability caused by the use of a particular numerical method in solving the problem. We will discuss the latter in the next chapter. To have a better understanding of inherent instability, let's take a look at the following ODE problem: $y'' - y' - 6y = 0$, with initial conditions $y(0) = 3$, and $y'(0) = -6$. The analytical solution of this problem is $y(t) = 3e^{-2t}$. This solution will tend to zero when t goes to infinity. Suppose we change the initial value $y(0)$ by adding to it a small value $\epsilon > 0$, so that $y(0) = 3 + \epsilon$. The analytical solution for this new initial value is $y(t) = (3+3\epsilon/5)e^{-2t} + (2\epsilon/5)e^{3t}$. It can be shown that no matter how small ϵ is chosen, the solution will tend to infinity as t goes to infinity. In this case,

the solution $y(t) = e^{-2t}$ is said to be unstable. The picture of the solution for $\varepsilon = 0, 0.01,$ and 0.1 are given in Figure 1.1. In numerical analysis, the situation when a little change in the initial condition gives an unstable solution is called an ill-conditioned problem. It is difficult to solve an ill-conditioned ODE problem numerically since the numerical method usually has rounding errors. The approximate solution will always tend to infinity just like the example of changing an initial condition given above. It is important then to look at the stability of the system before performing some numerical computations.

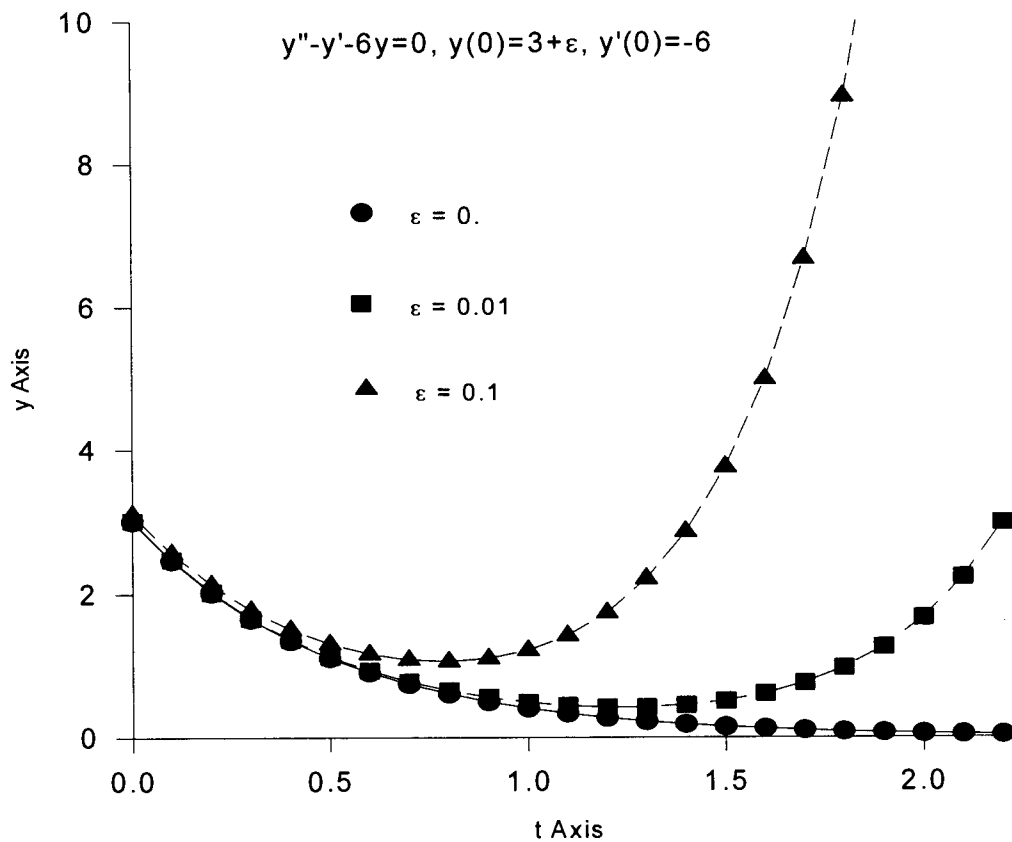


Figure 1.1 Solution $y = (3+3\varepsilon/5)e^{-2t} + 2\varepsilon/5e^{3t}$
for $\varepsilon = 0, 0.01,$ and 0.1

The concept of stability of a system given here is based on the concept written by Braun [10]. The question whether the solution $\phi(t)$ of the differential equation $y'=f(y)$, with $y(0) = y_0$ is stable or unstable is just the same as asking whether every solution $\psi(t)$ of $y' = f(y)$ starting from a value that is very close to $\phi(t)$ at $t=0$ remains close to $\phi(t)$ for all t in the given interval or not. The solution $y=\phi(t)$ of $y' = f(y)$ is said to be stable if every solution $\psi(t)$ of $y' = f(y)$ which starts sufficiently close to $\phi(t)$ at $t=0$ must remain close to $\phi(t)$ for all future times t . The solution $\phi(t)$ is unstable if there exists at least one solution $\psi(t)$ of $y' = f(y)$ which starts near $\phi(t)$ at $t=0$ but which does not remain close to $\phi(t)$ for all future time. In his book, Braun said that if A is a matrix of ODEs $y' = Ay$ then

1. Every solution $y = \phi(t)$ of $y' = Ay$ is stable if all the eigenvalues of A have negative real part.
2. Every solution $y = \phi(t)$ of $y' = Ay$ is unstable if at least one eigenvalue of A has positive real part, and
3. Suppose that all the eigenvalues of A have real part ≤ 0 and $\lambda_1 = i\sigma_1, \dots, \lambda_m = i\sigma_m$ have zero real part. Let $\lambda_j = i\sigma_j$ have multiplicity k_j . This means that the characteristic polynomial of A can be factored into the form $p(\lambda) = (\lambda - i\sigma_1)^{k_1} \dots (\lambda - i\sigma_m)^{k_m} q(\lambda)$ where all the roots of $q(\lambda)$ have negative real parts. Then, every solution $y = \phi(t)$ of $y' = Ay$ is stable if A has k_j linearly independent eigenvectors for each eigenvalue $\lambda_j = i\sigma_j$. Otherwise, every solution $\phi(t)$ is unstable.

1.5 Convergence of Euler Method

Euler method is the simplest approximation method for the initial value problem $y' = f(t,y)$ with $y(t_0) = y_0$. This method is called the simplest method because it is easy to derive directly from Taylor's series or directly from integration of the function by assuming the tangent $f(t,y)$ is constant at each interval. In practice, this method is not recommended. One reason for not recommending this method is because this method has a lack of accuracy compared to other methods at the equivalent stepsize. If $f(t,y)$ is assumed constant over a small interval (t_i, t_{i+1}) , a simple numerical procedure can be derived for calculating numerical values y_1, y_2, \dots, y_n , that approximate the true solution values $y(t_1), y(t_2), \dots, y(t_n)$, of $y' = f(t,y)$ respectively. By integrating the constant value $f(t_i, y_i)$ over the interval (t_i, t_{i+1}) , we have $y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t_i, y(t_i)) dt$. The result of this integration gives the relation $y_{i+1} = y_i + f(t_i, y_i)(t_{i+1} - t_i)$ where $y_0 = y(t_0)$, which is Euler method. If we examine this relation, it is none other than the first two terms of a Taylor's series of $y(t)$. A graphical interpretation of this method is shown in Figure 1.2, where $y' = f(t,y)$ is interpreted as the slope of the integral curve at any point (t,y) on the curve. Here, the solution curve on the range of (t_i, t_{i+1}) is approximated as a straight-line which passes through (t_i, y_i) with the slope $f(t_i, y_i)$. This type of explicit Euler method is sometimes called "the crude Euler method".

The crude Euler method can be corrected by choosing the slope of the curve on the range of (t_i, t_{i+1}) as the average of the slopes of the curve $y(t)$ at the range endpoints. The result of integration over the range of (t_i, t_{i+1}) then will give the approximate solution

of $y' = f(t,y)$ as $y_{i+1} = y_i + h/2[f(t_i,y_i) + f(t_{i+1},y_{i+1})]$. Since y_{i+1} appears implicitly on the right-hand-side, this representation is called an implicit representation. This corrected Euler method itself is sometimes called the Trapezoidal method. The value of y_{i+1} appeared in $f(t_{i+1},y_{i+1})$ is calculated using the crude Euler method. The procedure of finding the value of y_{i+1} at t_{i+1} by the Trapezoidal (corrected Euler) method is given as:

Predict y_{i+1} using the crude Euler method, $y_{i+1}^{\text{old}} = y_i + hf(t_i, y_i)$.

Correct y_{i+1} using the Trapezoidal method, $y_{i+1}^{\text{new}} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_{i+1}^{\text{old}})]$.

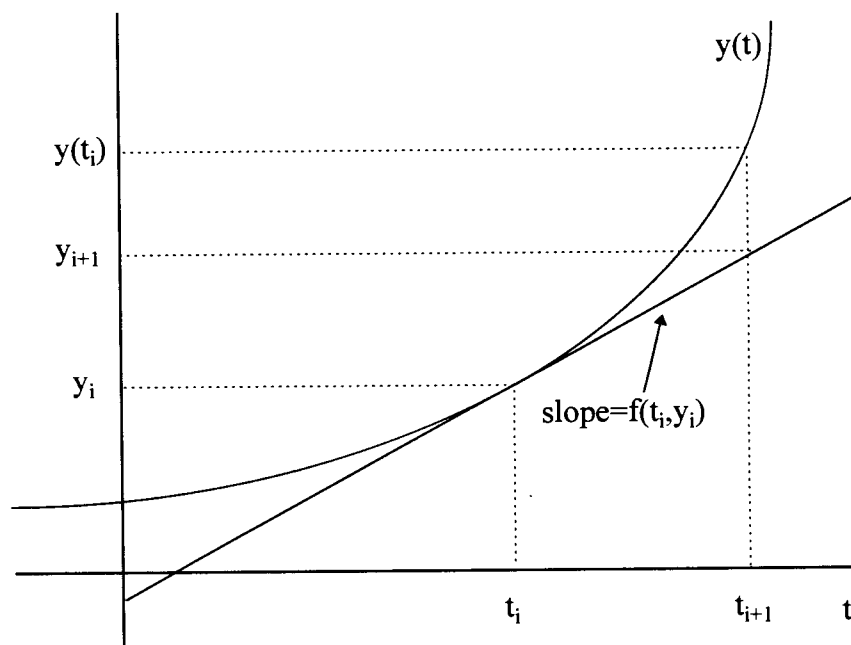


Figure 1.2 Crude Euler Method

The formula for predicting y_{i+1} is usually called a predictor, while the implicit

formula for correcting y_{i+1} is called a corrector. The procedure of predicting a result by one formula and then correcting by another formula is one of the most effective methods for solving initial value problems and is called a predictor-corrector method. To get a better result, the latter procedure can be used repeatedly until $|y_{i+1}^{\text{new}} - y_{i+1}^{\text{old}}| < \epsilon$. Let us take $y' = 4e^{0.8t} - 0.5y$ with $y(0) = 2$ as an example problem solved both by the crude Euler and the Trapezoidal methods. The exact solution for this problem is

$$y(t) = (4e^{0.8t} - 1.4e^{-0.5t})/1.3.$$

Without need to iterate the Trapezoidal method, its result is still better than that of the crude Euler method. Figure 1.3 is a graphical results of computations for stepsize $h = 0.5$. It can be seen that the Trapezoidal method is better than the crude Euler method.

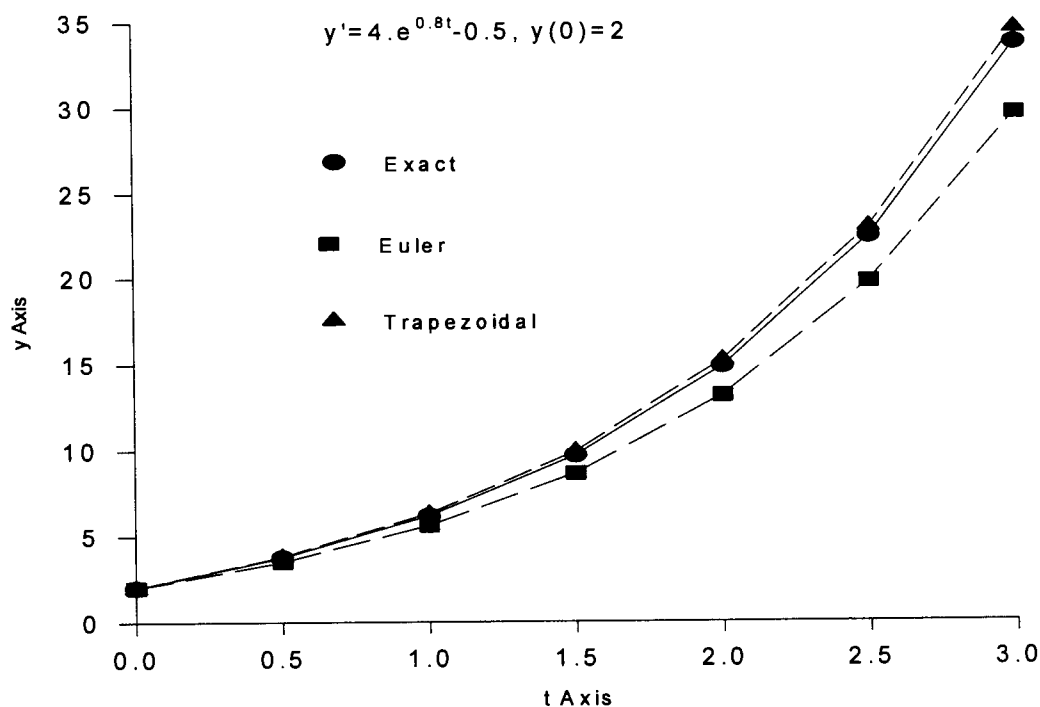


Figure 1.3 Numerical Results Using Exact, Euler, and Trapezoidal Methods

In order to have an acceptable computation, the analysis of errors caused by the numerical solution of ODEs becomes important [66, 78, 80]. Usually the errors are classified into two types of errors: round-off errors and truncation or discretization errors. Round-off errors are errors that occurred because of the limited numbers of significant digits that can be retained by a computer. Truncation errors consist of a local truncation error that results from computations over a single step, and a propagated truncation error that results from the previous steps. The sum of the local truncation error and the propagated truncation error is called a total or global truncation error [15, 44, 57, 66, 80]. Golub and Ortega [49] have shown that if the function f has a bounded partial derivative with respect to its second variable, and if a solution $y' = f(t,y)$ with $y(t_0) = y_0$ has a bounded second derivative, then the Euler approximation converges to the exact solution as $h \rightarrow 0$, and the global discretization error of Euler method satisfies $E(h) = O(h)$.

1.6 The Multistep Concept

The numerical methods of solving an ODE $y' = f(t,y)$ with initial value $y(t_0)=y_0$ are classified into two categories: single step methods and multistep methods. In single step methods, the approximate value of $y(t)$ at t_{i+1} is calculated merely using the values y_i , y_i' and h . In multistep methods, the approximate value of $y(t)$ at t_{i+1} is calculated using the recurrence relation in terms of the function values $y(t)$ and derivatives $y'(t)$ at t_{i+1} and at previous nodal points. Among examples of single step methods [66, 77, 80] are :

$$1. \quad y_{i+1} = y_i + f(t_i, y_i) \quad (\text{Euler method})$$

$$2. \quad y_{i+1} = y_i + \frac{y_i'}{1!}h + \dots + \frac{y_i^{(n)}}{n!}h^n \quad (\text{Taylor's series method})$$

$$3. \quad y_{i+1} = y_i + hf(t_i+h/2, y_i+(h/2)f(t_i, y_i)) \quad (\text{Midpoint rule})$$

Among examples of multistep methods are:

$$1. \quad y_{i+1} = y_i + \frac{h}{2}[3f(t_i, y_i) - f(t_{i-1}, y_{i-1})] \quad (\text{Adams-Bashforth})$$

$$2. \quad y_{i+1} = y_i + \frac{h}{2}[f(t_i, y_i) + f(t_{i+1}, y_{i+1})] \quad (\text{Adams-Moulton})$$

The idea of multistep methods appears when some information from previous points has been gained using single step methods. After gaining the information, the value of $f(t,y)$ at the next step is based on interpolation over that information. Computation using these multistep methods is in general more accurate than those in single step methods. Suppose we already have information of y_1, y_2, \dots, y_n using single step methods. By that information, we can calculate f_0, f_1, \dots, f_n . A multistep method

of solving $y' = f(t,y)$ can be based on the relation $y(t_{n+1}) = y(t_{n-M}) + \int_{t_{n-M}}^{t_{n+1}} f(t,y)dt$, where M

is some non-negative integer. In this case we will approximate the function $f(t,y)$ by a polynomial $P(t)$ generated using information $f_n, f_{n-1}, \dots, f_{n-M}$. Later on we will discuss another way of finding multistep methods. That is, instead of approximating the integrand $f(t,y)$, the methods will be achieved by approximating $y(t)$ using a polynomial and then by differentiating the polynomial. By substituting this interpolation polynomial

$P(t)$ for $f(t,y)$ as the integrand, we have the relation $y(t_{n+1}) = y(t_{n-M}) + \int_{t_{n-M}}^{t_{n+1}} P(t)dt$. In order

to integrate this from t_{n-M} up to t_{n+1} we should extrapolate $P(t)$ through t_{n+1} . The procedure of solving $y' = f(t,y)$ is then similar with that in the Trapezoidal method:

1. Predict y_{n+1} using $y_{n+1}^{\text{old}} = y_{n-M} + \int_{t_{n-M}}^{t_{n+1}} P(t) dt$.
2. Compute f_{n+1} using $f_{n+1} = f(t_{n+1}, y_{n+1}^{\text{old}})$.
3. Interpolate $P(t)$ using $f_{n+1}, f_n, f_{n-1}, \dots$.
4. Correct y_{n+1} using $y_{n+1}^{\text{new}} = y_{n-M} + \int_{t_{n-M}}^{t_{n+1}} P(t) dt$.

This procedure can be executed repeatedly until we satisfy with the desired accuracy of the corrected result. Suppose $\epsilon > 0$ is the accuracy needed for the computation. Then steps 1 through 4 will be executed repeatedly until $|y_{n+1}^{\text{new}} - y_{n+1}^{\text{old}}| < \epsilon$. Usually, $P(t)$ is approximated using Newton's backward-form (NBF). The followings are some examples of predictors and the related correctors [66, 78, 80]:

Milne method:

$$\text{Predictor} \quad : y_{n+1}^{\text{old}} = y_{n-3} + \frac{4h}{3} [2f_n - f_{n-1} + 2f_{n-2}].$$

$$\text{Corrector} \quad : y_{n+1}^{\text{new}} = y_{n-1} + \frac{h}{3} [f_{n+1} + 4f_n + f_{n-1}].$$

Adams-Moulton methods:

1. Second-order:

$$\text{Predictor} \quad : y_{n+1}^{\text{old}} = y_n + \frac{h}{2} [3f_n - f_{n-1}].$$

$$\text{Corrector} \quad : y_{n+1}^{\text{new}} = y_n + \frac{h}{2} [f_{n+1} + f_n].$$

2. Third-order:

$$\text{Predictor} : y_{n+1}^{\text{old}} = y_n + \frac{h}{12} [23f_n - 16f_{n-1} + 5f_{n-2}].$$

$$\text{Corrector} : y_{n+1}^{\text{new}} = y_n + \frac{h}{12} [5f_{n+1} + 8f_n - f_{n-1}].$$

3. Fourth-order:

$$\text{Predictor} : y_{n+1}^{\text{old}} = y_n + \frac{h}{24} [55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}].$$

$$\text{Corrector} : y_{n+1}^{\text{new}} = y_n + \frac{h}{24} [9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}].$$

4. Fifth-order:

$$\text{Predictor} : y_{n+1}^{\text{old}} = y_n + \frac{h}{720} [1901f_n - 2984f_{n-1} + 2616f_{n-2} \\ - 1274f_{n-3} + 251f_{n-4}].$$

$$\text{Corrector} : y_{n+1}^{\text{new}} = y_n + \frac{h}{720} [251f_{n+1} + 646f_n - 264f_{n-1} \\ + 106f_{n-2} - 19f_{n-3}].$$

5. Sixth-order:

$$\text{Predictor} : y_{n+1}^{\text{old}} = y_n + \frac{h}{1440} [4277f_n - 7923f_{n-1} + 9982f_{n-2} \\ - 7298f_{n-3} + 2877f_{n-4} - 475f_{n-5}].$$

$$\text{Corrector} : y_{n+1}^{\text{new}} = y_n + \frac{h}{1440} [475f_{n+1} + 1427f_n - 798f_{n-1} \\ + 482f_{n-2} - 173f_{n-3} + 27f_{n-4}].$$

CHAPTER II

STABILITY ANALYSIS

In the previous chapter the instability caused by the problems themselves has been addressed. The next observations will be about the instability caused by the approximation methods used to solve the problem. This instability is called induced instability. As mentioned in the previous chapter, before using the approximation method, we have to make sure that the problem does not belong to the class of ill-conditioned problems. Due to the inaccuracy of the numerical methods used in solving an initial value problem for an ordinary differential equation, errors occur at each step of the integration. The total error then becomes the summation of the local truncation errors and their propagation. If the method is unstable, the accumulation of these local errors will make the total error becomes larger than it would be expected. This is the reason why before using the approximation methods, we need to make sure that the problem does not have inherent instability. The phenomenon of the growth of this error is called numerically induced instability.

The stability of numerical methods of solving ODEs can be analyzed using the simple linear first order differential equation proposed by Dahlquist [27, 28, 29]

$$y' = \lambda y, y(t_0) = y_0 \tag{2.1}$$

where λ is a constant. This ODE is also called a test problem which was first introduced by Dahlquist [27, 28, 29] and is sometimes called Dahlquist's test problem. The solution of (2.1) is $y(t) = y_0 e^{\lambda(t-t_0)}$. If t_n is written as $t_n = t_0 + nh$, $y(t)$ can be given as $y(t_{n+1}) = e^{\lambda h} y(t_n)$. The value of $e^{\lambda h}$ can be approximated using Taylor's series. If the approximate value $e^{\lambda h}$ is denoted by $E(\lambda h)$, then $y(t_{n+1})$ can be approximated by $y_{n+1} = E(\lambda h)y_n$. Assume $E(\lambda h)$ is taken as a Taylor's series of order p , then

$$E(\lambda h) = 1 + \lambda h + \frac{(\lambda h)^2}{2!} + \dots + \frac{(\lambda h)^p}{p!}. \quad (2.2)$$

Based on the approximate solution given above, it can be said that a numerical method is stable, if the error $y_n - y(t_n) = e_n$ remains bounded as $n \rightarrow \infty$. By calculating the error resulted from the approximate solution, we have

$$\begin{aligned} e_{n+1} &= y_{n+1} - y(t_{n+1}) \\ &= E(\lambda h)[y(t_n) + e_n] - e^{\lambda h} y(t_n) \\ &= [E(\lambda h) - e^{\lambda h}]y(t_n) + E(\lambda h)e_n. \end{aligned} \quad (2.3)$$

From (2.3), it can be shown that the error at t_{n+1} consists of two parts. The first part, $E(\lambda h) - e^{\lambda h}$, is the local truncation error that can be chosen as small as we want by choosing a suitable approximation of $E(\lambda h)$. The second part, $E(\lambda h)e_n$ is the propagation error from the previous step t_n to t_{n+1} that will not grow if $|E(\lambda h)| \leq 1$.

Definition 2.1

For a given h and λ from Dahlquist's test problem $y' = \lambda y$, a numerical method

$y_{n+1} = E(\lambda h)y_n$ is called absolutely stable if $|E(\lambda h)| \leq 1$.

Definition 2.2

For a given h and λ from Dahlquist's test problem $y' = \lambda y$, a numerical method $y_{n+1} = E(\lambda h)y_n$ is called relatively stable if $|E(\lambda h)| \leq e^{\lambda h}$.

The region of stability of numerical methods will be based on the methods used to approximate the solutions [2, 4, 6, 14, 25, 66, 79].

In the case of an ODE in the form of $y' = f(t, y)$, with initial value $y(t_0) = y_0$, the problem can be transformed using the Taylor expansion about (t_n, y_n) , so that the problem can be solved as a linear problem. If the Taylor series is truncated after first order terms, the linearized form of the equation will be given as:

$$y' = \lambda y + Bt + C \quad (2.4)$$

where

$$\lambda = \frac{\partial f}{\partial y}(t_n, y_n), \quad B = \frac{\partial f}{\partial t}(t_n, y_n), \quad C = f_n - y_n \frac{\partial f}{\partial y}(t_n, y_n) - t_n \frac{\partial f}{\partial t}(t_n, y_n).$$

The solution of (2.4) can be found in the form $y = Ae^{\lambda t} + p(t)$, where $p(t)$ is a polynomial of degree at most 2. It can be seen that the solution is dominated by the exponential term.

2.1 Stability of Euler Methods

The crude Euler method is $y_{n+1} = y_n + hf(t_n, y_n)$. If we apply the test problem $y' = \lambda y$, we will have $y_{n+1} = (1 + \lambda h)y_n$. In this case $E(\lambda h) = 1 + \lambda h$. It can be said then that the crude Euler method is absolutely stable if $|1 + \lambda h| \leq 1$. For λ real, the interval of

absolute stability is $-2 \leq \lambda h \leq 0$. For λ a complex number, the region of stability is the region inside a circle with center $(-1,0)$ and radius 1 as shown in Figure 2.1.

The implicit Euler method is $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$. If we apply Dahlquist's test problem $y' = \lambda y$, we will have $y_{n+1} = y_n + h\lambda y_{n+1}$ or $y_{n+1} = (1-\lambda h)^{-1}y_n$. In this case $E(\lambda h) = (1-\lambda h)^{-1}$. It can be said then that the implicit Euler method is absolutely stable if $|1-\lambda h| \geq 1$. For λ real, the interval of absolute stability is $\{\lambda h \mid \lambda h \geq 2 \text{ or } \lambda h \leq 0\}$. For λ complex, the region of stability is the region outside the circle with center $(1,0)$ and radius 1 as shown in Figure 2.2. To have a better understanding of this stability concept, let takes an example $y' = -50y$, $y(0) = 1$. For the implicit Euler method, the numerical computation can be done by applying a predictor function and then applying Newton's iteration.

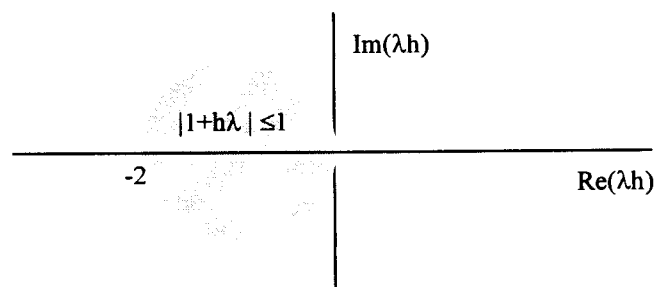


Figure 2.1 Stability Region of the Explicit Euler Method

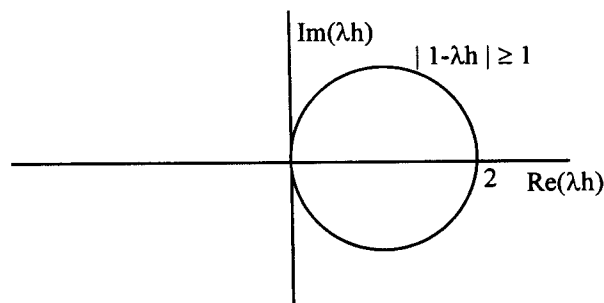


Figure 2.2 Stability Region of the Implicit Euler Method

The algorithm of the implicit Euler method using a predictor-corrector scheme is as follows [46, 72, 94]:

1. Predict y_{n+1} using the crude Euler method, $y_{n+1}^{\text{old}} = y_n + hf(t_n, y_n)$.
2. Correct y_{n+1} using the implicit method, $y_{n+1}^{\text{new}} = y_n + hf(t_{n+1}, y_{n+1}^{\text{old}})$.

The algorithm of the implicit Euler method applying Newton's iteration is as follows [51, 70, 72]:

1. Give the initial value for y_{n+1} , says y_{gues} , so that $y_{n+1}^{\text{old}} = y_{\text{gues}}$.
2. Compute $F(y_{n+1})$, $F(y_{n+1}^{\text{old}}) = y_{n+1}^{\text{old}} - y_n - hf(t_{n+1}, y_{n+1}^{\text{old}})$.
3. Compute $F'(y_{n+1})$, $F'(y_{n+1}^{\text{old}}) = 1 - hf_{y_{n+1}}(t_{n+1}, y_{n+1}^{\text{old}})$.
4. Compute y_{n+1} using Newton's iteration, $y_{n+1}^{\text{new}} = y_{n+1}^{\text{old}} - \frac{F(y_{n+1}^{\text{old}})}{F'(y_{n+1}^{\text{old}})}$.
5. Iterate again from step 2 until $|y_{n+1}^{\text{new}} - y_{n+1}^{\text{old}}| < \text{EPS}$, where EPS is an accuracy requested to a numerical solution.

The exact solution of this problem is $y = e^{-50t}$. A graphical representation of the explicit Euler method for $h = 0.01, 0.03, 0.04$, and 0.05 is given in Figure 2.3. The computation of $y' = -50y$, using the Euler method gives results as follows :

- The solution converges to the exact solution for every stepsize which is less than 0.04. If the stepsize is close to 0.04, the solution will oscillate but it will converge to the exact solution.
- For stepsize $h=0.04$, the solution will never converge to the exact solution even though the computation is stable. The solution just oscillates between

minus and plus the initial value by turns.

- For every stepsize larger than 0.04, the computation will never converge and it will oscillate and then go to minus or plus infinity by turns.

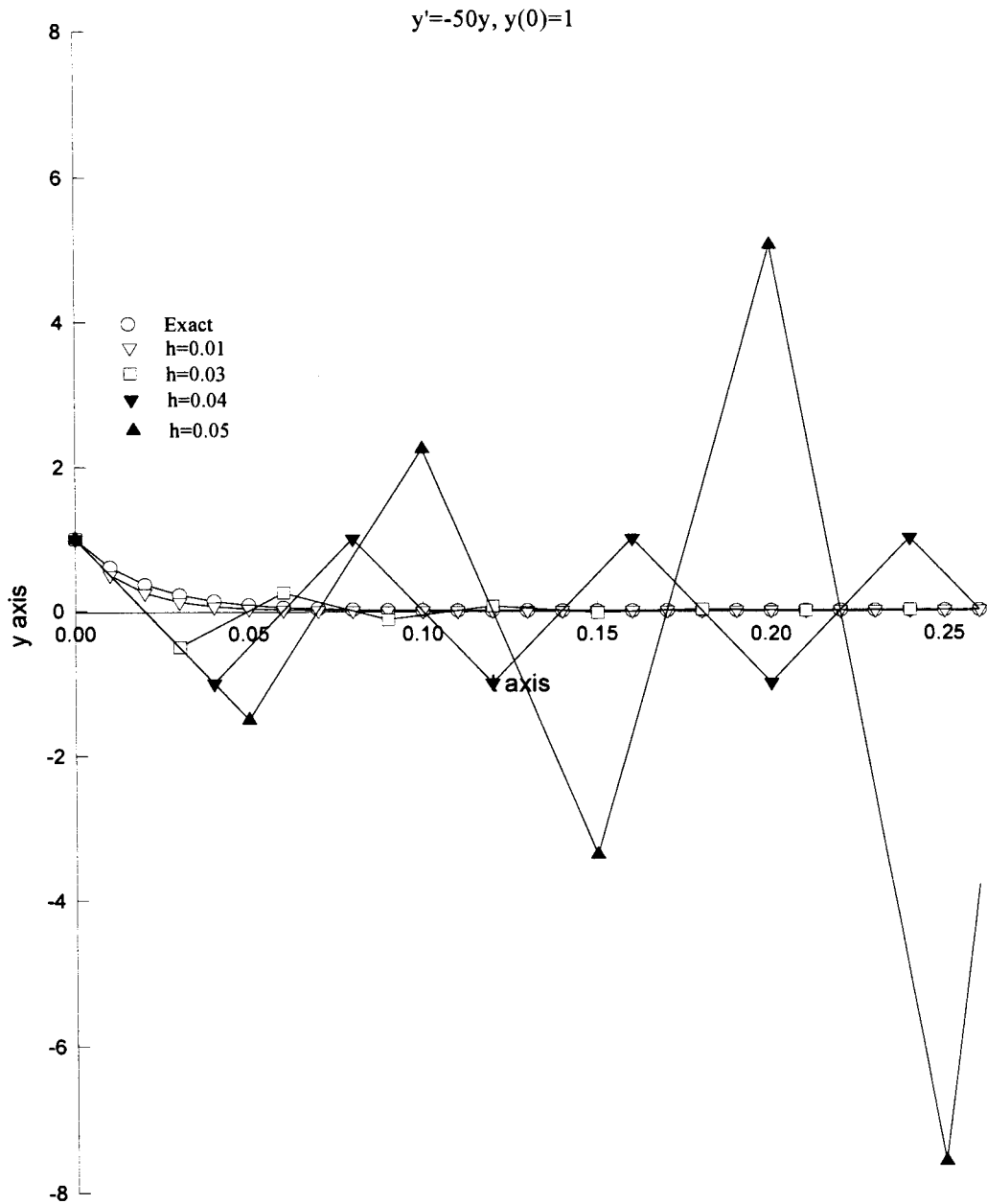


Figure 2.3 Euler Method with Stepsize $h=0.01, 0.03, 0.04,$ and 0.05

The comparison of computations among crude Euler, implicit Euler using predictor-corrector, and implicit Euler using Newton's iteration has been made for the case of problem $y' = -50y$, $y(0) = 1$. The conclusions taken from the computational experiments for the case $y' = -50y$ are:

1. The implicit Euler method using Newton's iteration always converges to the exact solution.
2. As mentioned in the previous experiments, the crude Euler method always converges to the exact solution for a timestep less than $2/50 = 0.04$. For the timestep $h=0.04$, the method is still stable, but does not converge to the right solution.
3. The computation of the implicit Euler method as a corrector and crude Euler as predictor is convergence to the exact solution for a timestep less than $1/50 = 0.02$ (see Appendix E). For the timestep $h = 0.02$, the computation is still stable, but the result is not convergence to the right solution. In the crude Euler method the computation still converges to the exact solution for timestep $h = 0.03$ (even though oscillating), but the computation using this kind of predictor-corrector does not converge for timestep $h \geq 0.03$. It just goes to infinity.

If the computation using the implicit Euler method as corrector and the crude Euler as predictor is stable, then the result converges to the result given from the implicit Euler using Newton's iteration. This PC method may not converge yet for one iteration, but if

we repeatedly correct the computation, the result will be the same with the result from the implicit Euler using Newton's iteration.

Graphical representation of the comparison among the three methods aforementioned for stepsizes $h = 0.01, 0.03, 0.04, 0.05$ are given in Figure 2.4.

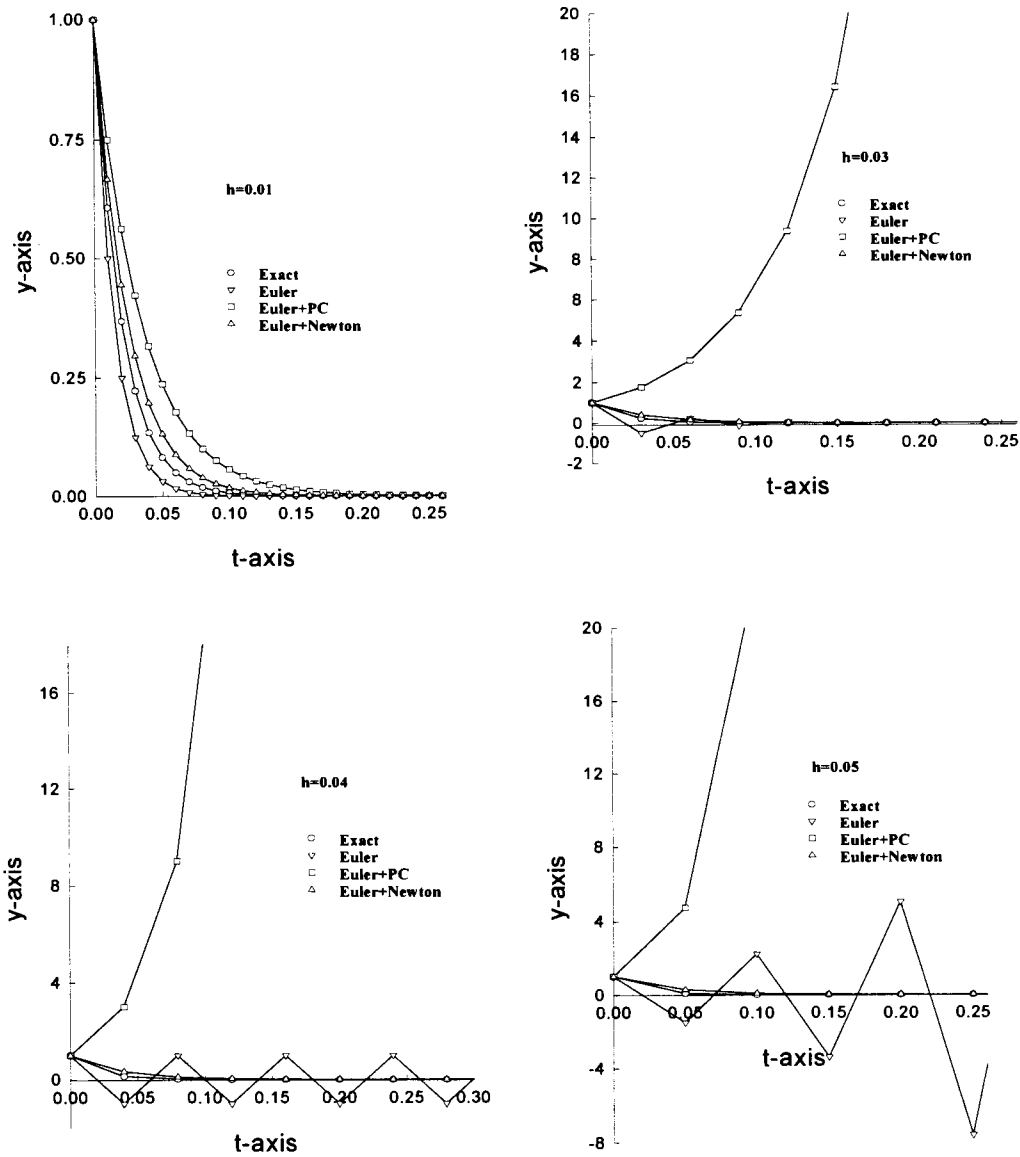


Figure 2.4 Crude Euler, Implicit Euler Using PC, and Implicit Euler Using Newton's

Graphical computations of $y'=-50y$, $y(0)=1$ using implicit Euler with Newton's iteration for $h=0.01$, 0.04 , and 0.07 are given in Figure 2.5, where the gradient is also shown.

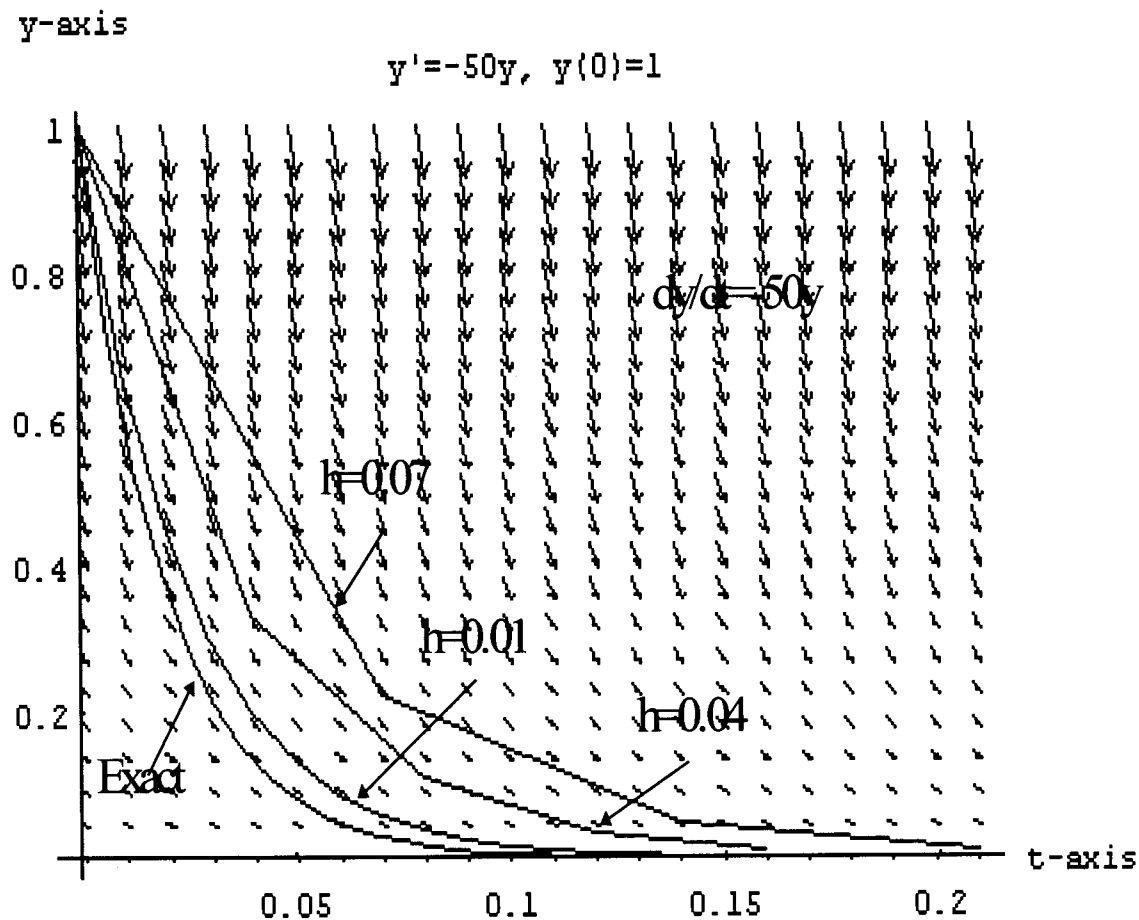


Figure 2.5 Implicit Euler by Showing the Vector Gradient

2.2 Stability of Explicit Runge-Kutta Methods

When Carl Runge and Wilhelm Kutta introduced Runge-Kutta methods, they were motivated by Euler method and Taylor's series. There are three major characteristics of Runge-Kutta methods:

- Single-step methods,
- Derived from Taylor's series with the same order, and
- Avoiding the evaluation of derivatives of f .

While the Euler method only evaluates the function $f(t,y)$ of $y' = f(t,y)$ at the solution points themselves, the Runge-Kutta method is done by evaluating the function $f(t,y)$ at several points within the integration interval. Let us consider the second-order Runge-Kutta method that is given as $y_{n+1} = y_n + ak_1 + bk_2$, where $k_1 = hf(t_n, y_n)$, $k_2 = hf(t_n + \lambda h, y_n + \beta k_1)$. The constants a , b , λ , and β are determined so that the equation will agree with the Taylor expansion. The Taylor expansion of $y(t)$ at t_{n+1} through terms of order h^3 is

$$y(t_{n+1}) = y(t_n) + hf(t_n, y_n) + h^2/2[f_t(t_n, y_n) + f(t_n, y_n)f_y(t_n, y_n)] + O(h^3).$$

By doing the same thing to $f(t,y)$ we get

$$f(t_n + \lambda h, y_n + \beta k_1) = f(t_n, y_n) + \lambda hf_t(t_n, y_n) + \beta k_1 f_y(t_n, y_n) + (\lambda^2 h^2/2)f_{tt}(t_n, y_n) + \lambda \beta h k_1 f_{ty}(t_n, y_n) + (\beta^2 k_1^2/2)f_{yy}(t_n, y_n) + O(h^3).$$

By substituting the latter expansion into $y_{n+1} = y_n + ak_1 + bk_2$, we have

$$y_{n+1} = y_n + (a+b)hf(t_n, y_n) + h^2[b\lambda f_t(t_n, y_n) + b\beta f(t_n, y_n)f_y(t_n, y_n)] + O(h^3).$$

This latter equation will be identical with the Taylor expansion of $y(t)$ at t_{n+1} through terms in h^2 if $a+b = 1$, $b\lambda = 1/2$, and $b\beta = 1/2$. If $a = 1/2$ then $b = 1/2$, $\lambda = 1$, $\beta = 1$, and y_{n+1} is written as

$$y_{n+1} = y_n + 1/2[k_1 + k_2] \tag{2.5}$$

where $k_1 = hf(t_n, y_n)$ and $k_2 = hf(t_n + h, y_n + k_1)$. Equation (2.5) is called a second-order Runge-Kutta method. Its graphical interpretation is shown in Figure 2.6. The stability of

second-order Runge-Kutta method can be examined using Dahlquist's test problem $y' = \lambda y$. By substituting the test function into (2.5), we have

$$y_{n+1} = y_n + 1/2[h\lambda y_n + h\lambda(y_n + h\lambda y_n)] = [1 + \lambda h + (\lambda h)^2/2]y_n.$$

Based on Definition 2.1, it can be said that the second-order Runge-Kutta method will be stable if $|1 + \lambda h + (\lambda h)^2/2| \leq 1$. A graphical representation of the region of stability when $h\lambda$ is complex is given in Figure 2.7. For $h\lambda$ a real number, the interval of stability is $-2 \leq h\lambda \leq 0$.

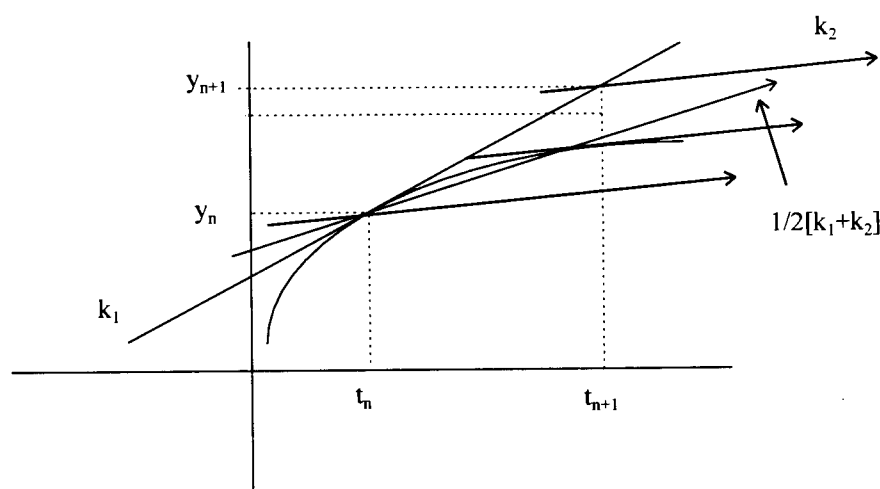


Figure 2.6 A Second Order of Runge-Kutta

If a is chosen $1/4$, the method is called the Heun method and is written as

$$y_{n+1} = y_n + k_1/4 + 3k_2/4 \quad (2.6)$$

where $k_1 = hf(t_n, y_n)$, and $k_2 = hf(t_n + 2h/3, y_n + 2k_1/3)$.

With the same approach, Lapidus and Seinfeld [80] have shown :

1. Third-order Runge-Kutta

$$y_{n+1} = y_n + 1/6 [k_1 + 4k_2 + k_3] \quad (2.7)$$

where $k_1 = hf(t_n, y_n)$, $k_2 = hf(t_n + h/2, y_n + k_1/2)$, and $k_3 = hf(t_n + h, y_n - k_1 + 2k_2)$. The region of absolute stability for λh complex is

$$\{ \lambda h \mid |1 + \lambda h + (\lambda h)^2/2 + (\lambda h)^3/6| \leq 1 \}$$

(see Figure B-1 of Appendix B). For λh a real number, the interval of absolute stability satisfies $-2.512745 \leq \lambda h \leq 0$.

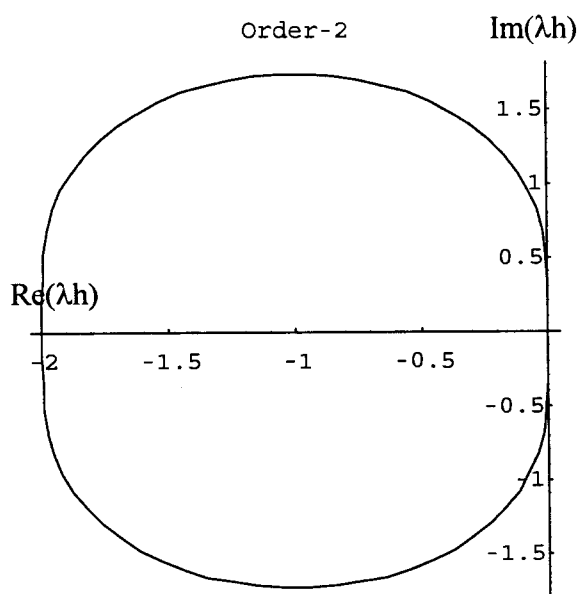


Figure 2.7 Stability Region of Second Order Runge-Kutta

2. Fourth-order Runge-Kutta

$$y_{n+1} = y_n + 1/6 [k_1 + 2k_2 + 2k_3 + k_4] \quad (2.8)$$

where $k_1 = hf(t_n, y_n)$, $k_2 = hf(t_n + h/2, y_n + k_1/2)$, $k_3 = hf(t_n + h/2, y_n + k_2/2)$, $k_4 = hf(t_n + h, y_n + k_3)$.

The region of absolute stability for λh complex is

$$\{ \lambda h \mid |1 + \lambda h + (\lambda h)^2/2 + (\lambda h)^3/6 + (\lambda h)^4/24| \leq 1 \}$$

(see Figure B-1 of Appendix B). For λh a real number, the interval of absolute stability satisfies $-2.78529 \leq \lambda h \leq 0$. For λh real numbers, the intervals of absolute stabilities of

explicit Runge-Kutta methods for orders 1 up to 10 are shown in Table 2.1. Butcher [15] proposed an algorithm to generate data for plotting the stability region of Runge-Kutta methods. We developed FORTRAN (see Appendix B) and Mathematica (see Appendix F under the name of “Mathematica 5”) codes with a different approach. The approach is based on the implementation of Newton-Raphson method for finding a root of a polynomial. Here, the polynomial is evaluated using nested multiplication which is recognized as Horner’s rule.

TABLE 2.1
Stability Region of Runge-Kutta
for Orders 1 to 10.

Order	Interval for λh real numbers
1 st	$-2 \leq \lambda h \leq 0$
2 nd	$-2 \leq \lambda h \leq 0$
3 th	$-2.5127453266 \leq \lambda h \leq 0$
4 th	$-2.78529 \leq \lambda h \leq 0$
5 th	$-3.21705 \leq \lambda h \leq 0$
6 th	$-3.55344 \leq \lambda h \leq 0$
7 th	$-3.95413 \leq \lambda h \leq 0$
8 th	$-4.31363 \leq \lambda h \leq 0$
9 th	$-4.70083 \leq \lambda h \leq 0$
10 th	$-5.06952 \leq \lambda h \leq 0$

The general expression of a s -stage Runge-Kutta method for solving $y' = f(t,y)$, $y(t_0) = t_0$ is given as:

$$y_{n+1} = y_n + \sum_{i=1}^s w_i k_i \quad (2.9)$$

where $k_1 = hf(t_n, y_n)$ and $k_i = hf(t_n + c_i h, y_n + \sum_{j=1}^{i-1} a_{ij} k_j)$, $i = 2, 3, \dots, s$

To maintain consistency, which will be discussed in the next chapter, these following conditions should be satisfied:

$$c_i = \sum_{j=1}^{i-1} a_{ij} \quad \text{and} \quad \sum_{k=1}^s w_k = 1, \quad i = 2, 3, \dots, s \quad (2.10)$$

Here, the w 's represent the weighting coefficients on the slopes, s is the total number of stages, the c 's represent the subinterval locations at which the derivatives are being evaluated, and the a 's represent the slope weightings for the intermediate stages.

Butcher proposed a condensed representation of the Runge-Kutta method as shown in Table 2.2.

TABLE 2.2
Butcher's Array of Runge-Kutta Methods

0				
c_1	a_{21}			
c_2	a_{31}	a_{32}		
.	.	.	.	
c_s	a_{s1}	a_{s2}	a_{ss-1}	
	w_1	w_2		w_s

This representation is recognized as Butcher's array. The Runge-Kutta (2.8) method is represented in Butcher's array as shown in Table 2.3.

TABLE 2.3
Butcher's Array of a Fourth-Order Runge-Kutta Method

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

2.3 Stability of Implicit Runge-Kutta Methods

The general expression of a s -stage implicit Runge-Kutta method for solving $y' = f(t,y)$, $y(t_0) = t_0$ is given in Equation (2.11) as follows:

$$y_{n+1} = y_n + \sum_{i=1}^s w_i k_i \quad (2.11)$$

where

$$k_i = hf(t_n + c_i h, y_n + \sum_{j=1}^s a_{ij} k_j), \quad i = 1, 2, 3, \dots, s$$

and

$$c_i = \sum_{j=1}^s a_{ij} \quad \text{and} \quad \sum_{k=1}^s w_k = 1, \quad i = 1, 2, 3, \dots, s.$$

Some derivations of implicit Runge-Kutta methods have been shown by Jain [66]. A second-order implicit Runge-Kutta method is given as

$$y_{n+1} = y_n + k_1 \quad (2.12)$$

where $k_1 = hf(t_n+h/2, y_n+k_1/2)$. If we apply the test problem $y' = \lambda y$, we have $k_1 = h\lambda(y_n+k_1/2)$. After finding k_1 , its value is substituted into Eq. (2.12) to get

$$y_{n+1} = \frac{1 + \lambda h / 2}{1 - \lambda h / 2} y_n \quad (2.13)$$

For λh real, the interval of absolute stability satisfies $\lambda h \leq 0$, and for λh complex, the region of absolute stability satisfies $|(1 + \lambda h/2)/(1 - \lambda h/2)| \leq 1$. Using a similar approach, a third-order implicit Runge-Kutta method can be written as

$$y_{n+1} = y_n + 1/4 [3k_1 + k_2] \quad (2.14)$$

with $k_1 = hf(t_n+h/3, y_n+k_1/3)$ and $k_2 = hf(t_n+h, y_n+k_1)$. For λh real, the interval of absolute stability satisfies $\lambda h \leq 0$, and for λh complex, the region of absolute stability satisfies $|[1+2\lambda h/3+(\lambda h)^2/6]/[1-\lambda h/3]| \leq 1$.

2.4 Difference Equations

If y is a function of t and t is replaced by $t_k = t_0 + kh$, where k is an element of $\{..,-2, -1,0,1,2,..\}$, then the dependent variable $y(t)$ can be replaced by y_k as an approximation for $y(t_k)$. The forward difference of the y_k value is denoted by Δy_k and defined as $y_{k+1}-y_k$. This definition implies that

$$\Delta^2 y_k = \Delta(\Delta y_k) = \Delta(y_{k+1}-y_k) = y_{k+2} - 2y_{k+1} + y_k = \Delta y_{k+1} - \Delta y_k.$$

The first derivatives of y over t are defined as

$$\frac{dy}{dt} = \lim_{h \rightarrow 0} \begin{cases} \frac{y(t+h) - y(t)}{h} \\ \frac{y(t) - y(t-h)}{h} \\ \frac{y(t+h) - y(t-h)}{2h} \end{cases}$$

The finite difference approximation of the first derivative of y is defined using the first two terms of the Taylor series of y as

$$\frac{dy}{dt} \rightarrow \begin{cases} \frac{y_{k+1} - y_k}{h} & \text{forward Euler.} \\ \frac{y_k - y_{k-1}}{h} & \text{backward Euler} \\ \frac{y_{k+1} - y_{k-1}}{2h} & \text{central difference scheme.} \end{cases}$$

The interpretation of these methods in solving a problem $y' = f(y)$ is as follows:

$$\begin{aligned} \frac{y_{k+1} - y_k}{h} &= f(y_k) && \text{forward Euler} \\ \frac{y_k - y_{k-1}}{h} &= f(y_k) && \text{backward Euler} \\ \frac{y_{k+1} - y_{k-1}}{2h} &= f(y_k) && \text{central difference scheme} \end{aligned}$$

Let's take an example of the decay equation $dy/dt = -y$, $y(0) = y_0$. The correspondence of methods is

$$\begin{aligned} \frac{y_{k+1} - y_k}{h} &= -y_k && \text{forward Euler} \\ \frac{y_k - y_{k-1}}{h} &= -y_k && \text{backward Euler} \\ \frac{y_{k+1} - y_{k-1}}{2h} &= -y_k && \text{central difference scheme} \end{aligned}$$

The exact solution of this problem is $y(t) = y_0 e^{-t}$. The absolute value of the exact solution monotonically decreases to zero as $t \rightarrow \infty$. The forward Euler scheme for this problem is $y_{k+1} = (1-h)y_k$. By substituting the value of y_0 , we have the relation $y_k = (1-h)^k y_0$. The backward scheme for this problem is $y_k = y_{k-1} - hy_k$ or $y_k = [1/(1+h)] y_{k-1}$. By taking $k = k+1$, we can have $y_{k+1} = [1/(1+h)]y_k$. By substituting the value of y_0 , we have the relation $y_k = [1/(1+h)]^k y_0$. The relation for the central difference scheme is $y_{k+1} = y_{k-1} - 2hy_k$. The value of y_1 for this method is calculated using the Euler method. All computations of this problem were done using the three methods, each for $h = 0.05, 1, 2$, and 3 . From the observations shown in Figure 2.8, the backward method is always stable for every $0 < h < \infty$. The central method is always unstable for every $h > 0$. For $h = 1$ and $k > 0$, the value y_k of the forward method is identically equal to zero. If $1 < h < 2$, the value y_k of the forward method decreases to zero with oscillating (change in sign) amplitude. For $h \geq 2$, the y_k of the forward method of this problem does not lead the solution to zero when t goes to infinity. The forward method is always stable for h in the interval $0 < h < 2$. By applying the forward method to the test problem $y' = \lambda y$ we have $y_{k+1} = y_k + h\lambda y_k = (1+\lambda h)y_k$. From the latter equation, for λh real, the region of stability of the forward method is an interval $\{\lambda h \mid |1+\lambda h| \leq 1\} = -2 \leq \lambda h \leq 0$ (similar to the crude Euler method). In a similar way, the region of absolute stability of the backward method for λh real is an interval $\{\lambda h \mid \lambda h \geq 0 \text{ or } \lambda h \leq 0\}$ (similar to the implicit Euler method). For the central method, the formula for finding the region of stability is not simple.

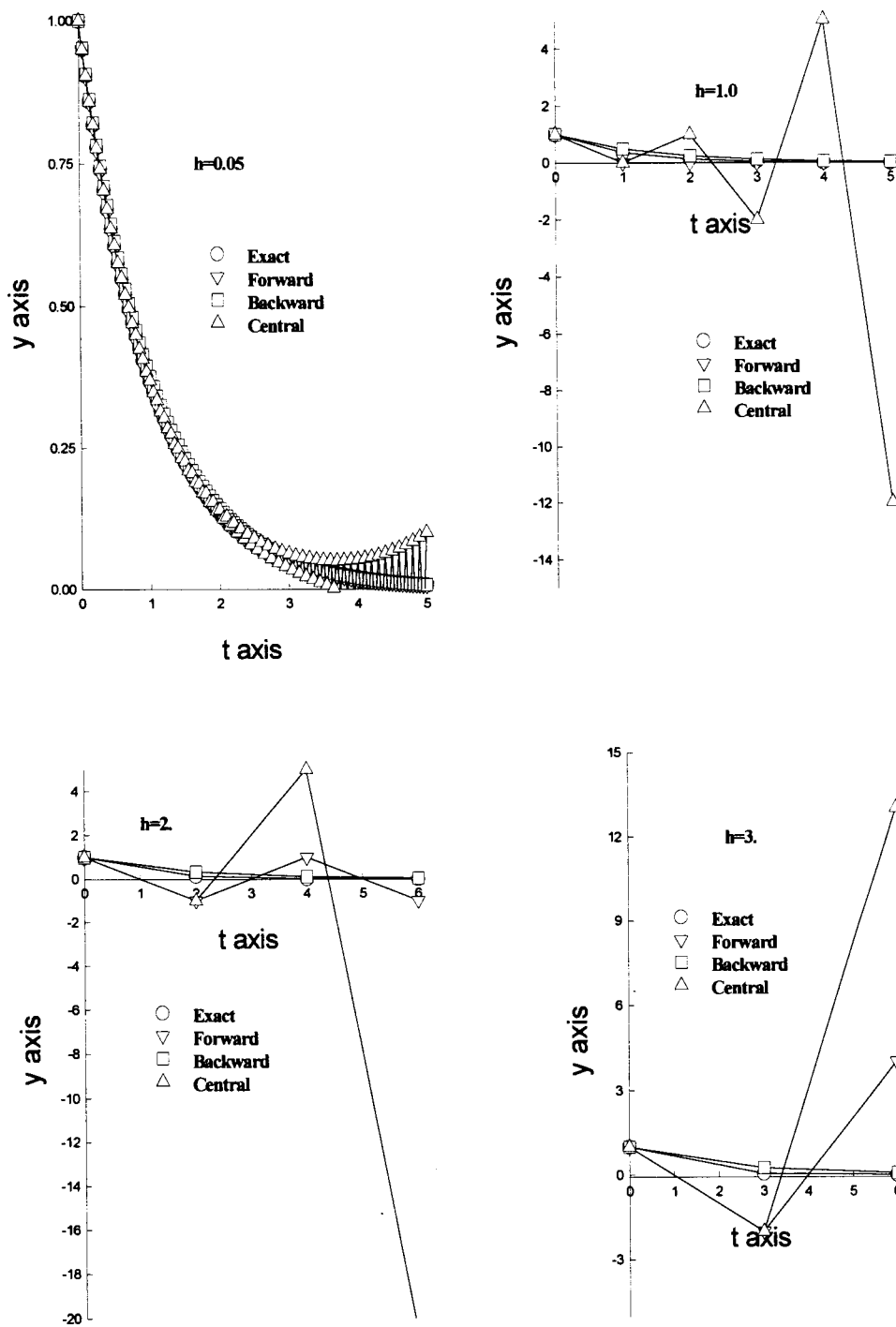


Figure 2.8 Difference Equation Methods Using Forward, Backward and Central for $h=0.05, 1, 2,$ and 3

CHAPTER III

STABILITY OF MULTISTEP METHODS

We have seen that some ideas for solving Ordinary Differential Equations lead to multistep methods. Unfortunately, the computations of multistep methods are not as easy as those in single step methods, because in multistep methods some starting values are needed but they are not given in the starting information. Traditionally, to support this lack of information we can use Runge-Kutta methods with the same order accuracy. There are two possibilities that can be used in order to provide starting values. First, use a single step method to compute all starting values that are not given in the information. Second, use a single step method to support information for a two-step method, then use this two-step method to support information for a three-step method and so on until all information needed to support the chosen multistep method is fulfilled. In general, the multistep methods not only include explicit single step methods, but also implicit single step methods. A general multistep or k -step method for the solution $y' = f(t,y)$, $y(t_0) = y_0$ can be written as

$$a_0 y_n + a_1 y_{n+1} + a_2 y_{n+2} + \dots + a_{k-1} y_{n+k-1} + a_k y_{n+k} = h\phi(t_n, t_{n+1}, \dots, t_{n+k-1}, t_{n+k}, y_n, y_{n+1}, \dots, y_{n+k-1}, y_{n+k}; h) \quad (3.1)$$

where h is the constant stepsize and a_0, a_1, \dots, a_k are real constants. If ϕ independent of y_{n+k} then the general multistep method is called an explicit, open or predictor method;

otherwise it is an implicit, closed or corrector method. A general linear multistep or k-step method can be written as

$$a_0 y_n + a_1 y_{n+1} + a_2 y_{n+2} + \dots + a_{k-1} y_{n+k-1} + a_k y_{n+k} = h(b_0 y'_n + b_1 y'_{n+1} + \dots + b_{k-1} y'_{n+k-1} + b_k y'_{n+k}) \quad (3.2)$$

This formula can be simplified to

$$\rho(E)y_n - h\sigma(E)y'_n = 0 \quad (3.3)$$

where $\rho(\xi) = a_0 + a_1\xi + a_2\xi^2 + \dots + a_{k-1}\xi^{k-1} + a_k\xi^k$, $\sigma(\xi) = b_0 + b_1\xi + b_2\xi^2 + \dots + b_{k-1}\xi^{k-1} + b_k\xi^k$, and E is the shift operator which is defined as $E^k y_n = y_{n+k}$. Dahlquist [1956] was the person who started to study a general theory of multistep methods and was the first person who introduced the polynomials $\rho(\xi)$ and $\sigma(\xi)$.

In the same manner as single step methods, the theory of multistep methods also involves the concepts of consistency, stability, and convergence. Dahlquist [27,28] has defined that Equation (3.3) is said to be consistent if $\rho(1)=0$, and $\rho'(1) = \sigma(1)$. We know that the initial value problem $y' = 0$, $y(0) = 1$ has the exact solution $y(t) = 1$. If we solve this problem using (3.2), we have

$$a_0 1 + a_1 1 + \dots + a_k 1 = h(b_0 0 + b_1 0 + \dots + b_k 0) \text{ or } \rho(1) = 0.$$

Let's take another example, $y' = 1$ with $y(0)=0$. The exact solution of this example is $y(t) = t$. Substituting this exact solution into (3.2), we have

$$\begin{aligned} a_0(nh) + a_2(n+1)h + \dots + a_k(n+k)h &= nh[a_0 + a_1 + \dots + a_k] + h[1a_1 + 2a_2 + \dots \\ &\quad + (k-1)a_{k-1} + ka_k] \\ &= h(b_0 + b_1 + \dots + b_k) \end{aligned}$$

where $y_s = sh$, for $s=0, 1, 2, \dots, n+k$. Simplifying the latter equation and applying the result of $\rho(1)=0$, we have $\rho'(1) = \sigma(1)$. Dahlquist [27,28], Lapidus and Seinfeld [80], and Jain [66] have shown that the multistep method (3.1) is of order p , if and only if one of the following equivalent conditions is satisfied :

$$1. \sum_{i=0}^k a_i = 0, \text{ and } \sum_{i=0}^k a_i i^q = q \sum_{i=0}^k b_i i^{q-1} \text{ for } q = 2, \dots, p, \quad (3.4)$$

$$2. \rho(e^h) - h\sigma(e^h) = O(h^{p+1}) \text{ for } h \rightarrow 0, \text{ and} \quad (3.5)$$

$$3. \frac{\rho(\xi)}{\log \xi} - \sigma(\xi) = O((\xi - 1)^p) \text{ for } \xi \rightarrow 1. \quad (3.6)$$

Definition 3.1

The formula (3.1) will be said to be absolutely stable in the sense of Dahlquist if all roots ξ_i of the characteristic polynomial $\rho(\xi)=0$ are such that $|\xi_i| \leq 1$ and those roots for which $|\xi_i| = 1$ are simple.

Theorem 3.1

Consistency and stability are together necessary and sufficient conditions for the formula (3.1) to be convergence.

Proof: See Dahlquist [27,28].

Example 3.1:

Investigate the consistency and the order of a numerical method

$$y_{n+2} = -4y_{n+1} + 5y_n + h(4f_{n+1} + 2f_n).$$

Solution:

For this case $k = 2$, $a_0 = -5$, $a_1 = 4$, $a_2 = 1$, $b_0 = 2$, $b_1 = 4$, $b_2 = 0$, $\sigma(\xi) = 2 + 4\xi$, and $\rho(\xi) = -5 + 4\xi + \xi^2$. Since $\rho(1) = -5 + (4)(1) + 1^2 = 0$, and $\rho'(1) = 4 + (2)(1) = \sigma(1)$, then the method is consistent. From condition 2 (3.2), we have

$$\begin{aligned} \rho(e^h) - h\sigma(e^h) &= -5 + 4e^h + e^{2h} - h(2 + 4e^h) \\ &= -5 + 4[1 + h + h^2/2 + O(h^3)] + [1 + (2h) + (2h)^2/2 + O((2h)^3)] \\ &\quad - 2h - 4h(1 + h + h^2/2 + O(h^3)) \\ &= -5 + 4 + 4h + 2h^2 + O(h^3) + 1 + 2h + 2h^2 + O(h^3) - 2h - 4h - 4h^2 \\ &\quad - 2h^3 - O(h^3) = -2h^3 + O(h^3) = O(h^3). \end{aligned}$$

So the numerical method is of order 2.

Example 3.2:

Investigate the consistency and the order of the trapezoidal method

$$y_{n+1} = y_n + (h/2)[f_n + f_{n+1}].$$

Solution :

In this case $k = 1$, $a_0 = -1$, $a_1 = 1$, $b_0 = 1/2$, $b_1 = 1/2$, $\sigma(\xi) = 1/2 + (1/2)\xi$, and $\rho(\xi) = -1 + \xi$. Since $\rho(1) = -1 + (1)(1) = 0$, and $\rho'(1) = 1 = \sigma(1) = 1/2 + (1/2)(1)$, then the method is consistent. Let us now use condition 1 to show the order of the trapezoidal method. By applying condition 1(3.1), we can see that the condition will be satisfied only for $p = 2$. So the numerical method is of order 2.

3.1 Linear Difference Equations

Among other purposes of studying difference equations are the use of these equations to formulate and analyze discrete-time systems, to approximate the integrations

of differential equations using finite-difference schemes, and to study deterministic chaos [91, 92]. In general, an ordinary difference equations is a relation of the form

$$y_{n+k} = F(n, y_n, y_{n+1}, \dots, y_{n+k-1}), \quad (3.7)$$

where the order is the difference between the highest and lowest indices that appear in the equation. Based on this definition, the order of Equation (3.7) is k , and shifting the labeling of the indices will not change the order of the equation. If Equation (3.7) is shifted to $y_{n+k+r} = F(n+r, y_{n+r}, y_{n+r+1}, \dots, y_{n+r+k-1})$, its order is still k . A solution of difference equation (3.7) is a function $\varphi(n)$ that reduces the equation to an identity. If we substitute $\varphi(n) = 2^n$ into $y_{n+1} - 2y_n = 0$, we will have an identity in the form of $2^{n+1} - (2)2^n = 0$. In this case, $\varphi(n) = 2^n$ is said a solution of $y_{n+1} - 2y_n = 0$. A difference equation can be derived from a sequence $\{y_n\}$ to which is defined as a function of n and k arbitrary constants c_1, c_2, \dots, c_k . The difference equation of this sequence will be of order k . If we calculate y_{n+1} from $y_n = A2^n$, we will have $y_{n+1} = A2^{n+1} = (2)A(2)^n = 2y_n$. The difference equation of $y_n = A2^n$ is a first-order equation in the form of $y_{n+1} - 2y_n = 0$. By calculating y_{n+1} and y_{n+2} from $y_n = c_12^n + c_22^n$, we will have a second-order of difference equation in the form of $y_{n+2} - 7y_{n+1} + 10y_n = 0$. This gives a difference equation $y_n = An + f(A)$ is $y_n = (y_{n+1} - y_n)n + f(y_{n+1} - y_n)$ which can be found by calculating y_{n+1} and substituting the value of A back into the first equation.

A k th-order linear difference equation with constant coefficients is a relation in the form of

$$y_{n+k} + a_1y_{n+k-1} + a_2y_{n+k-2} + \dots + a_ky_n = R_n \quad (3.8)$$

where a_1, a_2, \dots , and a_k are constants with $a_k \neq 0$, and R_n is a function of n . The equation (3.8) is called a k th-order nonhomogeneous linear difference equation with constant coefficients. If $R_n = 0$, then Equation (3.8) is called homogeneous and it becomes

$$y_{n+k} + a_1 y_{n+k-1} + a_2 y_{n+k-2} + \dots + a_k y_n = 0. \quad (3.9)$$

Using the shift operator E , we can write Equation (3.9) in the form $f(E)y_n = 0$, where $f(E) = E^k + a_1 E^{k-1} + a_2 E^{k-2} + \dots + a_k$ is the operator function. The characteristic equation associated with Equation (3.9) is $f(r) = 0$.

Theorem 3.2

Let r_i be any solution to the characteristic $f(r) = r^k + a_1 r^{k-1} + a_2 r^{k-2} + \dots + a_k = 0$, then $y_n = r_i^n$ is a solution to the homogeneous equation (3.9).

Proof:

Substituting $y_n = r_i^n$ into (3.9) give

$$\begin{aligned} r_i^{n+k} + a_1 r_i^{n+k-1} + \dots + a_{k-1} r_i^{n+1} + a_k r_i^n &= r_i^n (r_i^k + a_1 r_i^{k-1} + \dots + a_{k-1} r_i + a_k) \\ &= r_i^n f(r_i) = r_i^n \cdot 0 = 0. \end{aligned}$$

Hence, $y_n = r_i^n$ is a solution to equation (3.9).

If the k roots of the characteristic equation $f(r) = 0$ are distinct, then the general solution of (3.9) will be $y_n = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n$. In the case where r_i has multiplicity m_i , $i=1, 2, \dots, j$, where $m_1 + m_2 + \dots + m_j = k$, then the solution of Equation (3.9) becomes

$$\begin{aligned} y_n &= r_1^n [a_{1,1} + a_{1,2}n + a_{1,3}n^2 + \dots + a_{1,m_1}n^{m_1-1}] + r_2^n [a_{2,1} + a_{2,2}n + a_{2,3}n^2 + \dots \\ &\quad + a_{2,m_2}n^{m_2-1}] + \dots + r_j^n [a_{j,1} + a_{j,2}n + a_{j,3}n^2 + \dots + a_{j,m_j}n^{m_j-1}]. \end{aligned}$$

The solution y_n of Equation (3.9) is usually written as y_n^H .

A particular solution of the nonhomogeneous linear difference Equation (3.8) can be derived from the relation $y_n^P = f(E)^{-1}R_n$. The general solution of Equation (3.8) then becomes $y_n^G = y_n^H + y_n^P$. When R_n has certain forms, we can use the trial solutions given in Table 3.1 below. The unknown coefficients should be determined by substituting the trial solutions into the difference equation. If a term of the trial solution appeared in the homogeneous solution, then the entire trial solution corresponding to this term should be multiplied by a positive integer power of n . This power should be just large enough so that no term in the new trial solution will appear in the homogeneous solution.

Example 3.3:

$$\text{Solve } y_{n+2} - 4y_{n+1} + 4y_n = 3(2)^n + 5(4)^n$$

Solution:

A homogeneous solution of this difference equation can be found by finding the roots of the characteristic equation $f(r) = r^2 - 4r + 4 = 0$. Since the roots are $r_1 = 2$ and $r_2 = 2$, the homogeneous solution is $y_n^H = c_1 2^n + c_2 n(2)^n$. The particular solution for the nonhomogeneous solution can be found by choosing the trial solution $y_n^P = A_1 n^2(2)^n + A_2(4)^n$. By substituting this y_n^P into the difference equation, we have $y_n^P = 3n^2(2)^{n-3} + 5(4)^{n-1}$. The general solution of the difference equation is

$$y_n^G = y_n^H + y_n^P = c_1 2^n + c_2 n(2)^n + 3n^2(2)^{n-3} + 5(4)^{n-1}.$$

Example 3.4:

$$\text{Solve numerically } y_{n+2} = -4y_{n+1} + 5y_n + h(4f_{n+1} + 2f_n) \quad \text{and apply to}$$

$$y' = y, y(0) = 1.$$

TABLE 3.1
A Trial Solution to Find a Particular Solution of Nonhomogeneous
Difference Equations

Terms in T_n	Trial solution y_n^p
β^n	$A\beta^n$
$\sin \alpha n$ or $\cos \alpha n$	$A \cos \alpha n + B \sin \alpha n$
polynomial $P(n)$ of degree m	$A_0 n^m + A_1 n^{m-1} + \dots + A_m$
$\beta^n P(n)$	$\beta^n (A_0 n^m + A_1 n^{m-1} + \dots + A_m)$
$\beta^n \sin \alpha n$ or $\beta^n \cos \alpha n$	$\beta^n (A \cos \alpha n + B \sin \alpha n)$

Solution:

Calculating the polynomials ρ and σ , we have $\rho(\xi) = \xi^2 + 4\xi - 5$, $\sigma(\xi) = 4\xi + 2$, and $\rho'(\xi) = 2\xi + 4$. The multistep method is consistent because $\rho(1) = 1 + 4 - 5 = 0$ and $\rho'(1) = 2(1) + 4 = \sigma(1) = 4(1) + 2 = 6$. Since $a_0 = -5$, $a_1 = 4$, $a_2 = 1$, we have $a_0 + a_1 + a_2 = -5 + 4 + 1 = 0$. By observation of formula (3.4), the condition is satisfied for $q=2$ and $q=3$. It can be said then that the multistep method is of order 3. Since the roots of characteristic $\rho(\xi) = 0$ are $\xi_1 = -5$, and $\xi_2 = 1$, then the multistep method is unstable. If the computation satisfies the stability condition, then the method is said to have convergence with order 3. The linear difference equation relation is $y_{n+2} + 4(1-h)y_{n+1} - (5+2h)y_n = 0$. As starting values, we can take $y_0 = 1$, and $y_1 = e^h$. The roots of the characteristic of $r^2 + 4(1-h)r - (5+2h)$ are

$$r_1 = 2h - 2 + \sqrt{4h^2 - 6h + 9} \quad \text{and} \quad r_2 = 2h - 2 - \sqrt{4h^2 - 6h + 9}.$$

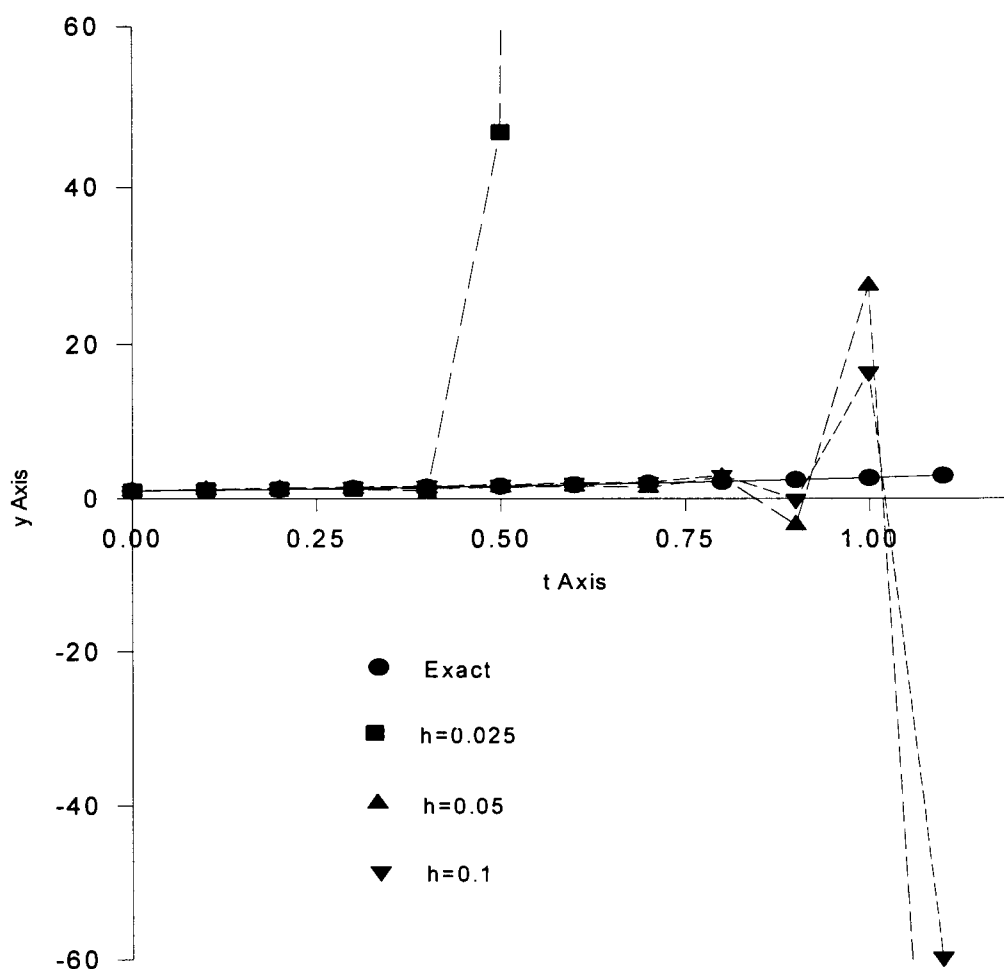


Figure 3.1 Computational Results of $y_{n+2} = -4y_{n+1} + 5y_n + h(4f_{n+1} + 2f_n)$ Applied to $y' = y, y(0) = 1$.

The general solution of the difference equation is $y_n^H = Ar_1^n + Br_2^n$. The computations are done using $h=0.025, 0.05, \text{ and } 0.1$ as shown in Figure 3.1. In this case, the smaller the stepsize is chosen, the worse result is achieved. This occurred because as $h \rightarrow 0, y_n^H$ will oscillate to infinity for r_1 and r_2 equal to 1 and -5, respectively. We can see that the second part of y_n^H has a very strong growth in the solution for a small stepsize. If we approximate the squareroot in r_1 and r_2 using the first two terms of Taylor's series, we

have $r_1 = 1 + h + O(h^2)$, and $r_2 = -5 + O(h)$. Since r_1 is an approximation of e^h , the first term of y_n^H approximates the exact solution e^t . The second part of y_n^H is often called a parasitic solution.

3.2 Adams Methods

John Couch Adams was motivated by a problem given by F. Bashforth when Adams proposed Adams-Bashforth methods. The Adams-Newton Backward Difference formula is used as approximation functions to the integrands. If y is a function of t and t is replaced by $t_k = t_0 + kh$, the backward difference of the y_k value is denoted by ∇y_k and defined as $y_k - y_{k-1}$. It can be shown then that $\nabla^n y_k = \nabla^{n-1} y_k - \nabla^{n-1} y_{k-1}$. From the k known values $y_{n-k+1}, y_{n-k+2}, \dots, y_{n-1}, y_n$, the k points $(t_{n-k+1}, f_{n-k+1}), (t_{n-k+2}, f_{n-k+2}), \dots, (t_n, f_n)$ can be determined. The Newton Backward Difference formula interpolates the function $f(t, y)$ of $y' = f(t, y)$ as

$$p^*(t) = p^*(t_n + sh) = \sum_{j=0}^{k-1} (-1)^j \binom{-s}{j} \nabla^j f_n \quad (3.10)$$

where $\binom{-s}{j} = (-1)^j \frac{s!}{j!(s-j)!}$, $s = \frac{t - t_n}{h}$, and $t_n \leq t \leq t_{n+1}$. The solution of $y' = f(t, y)$ at

t_{n+1} is given by $y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t)) dt$, which in the numerical result is given by

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p^*(t) dt \quad (3.11)$$

By substituting (3.10) into (3.11) and using a new variable s such that the integration will be from 0 to 1, Lambert [78] has shown that

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \gamma_j^* \nabla^j f_n \quad (3.12)$$

where the coefficients γ_i^* satisfy

$$\gamma_j^* = (-1)^j \int_0^1 \binom{-s}{j} ds \quad (3.13)$$

The multistep method (3.12) is called an explicit Adams-Bashforth method. Recurrence relations for coefficients γ_i^* (see Appendix C) are given as

$$\gamma_i^* + \frac{1}{2} \gamma_{i-1}^* + \frac{1}{3} \gamma_{i-2}^* + \frac{1}{4} \gamma_{i-3}^* + \dots + \frac{1}{i+1} \gamma_0^* = 1, \quad i = 0, 1, 2, \dots \quad (3.14)$$

The family of explicit Adams-Bashforth methods then can be written as

$$y_{n+1} = y_n + h \left(f_n + \frac{1}{2} \nabla f_n + \frac{5}{12} \nabla^2 f_n + \frac{3}{18} \nabla^3 f_n + \dots \right). \quad (3.15)$$

By considering the multistep method (3.15), special cases of (3.12) can be shown to be

$$k = 1: \quad y_{n+1} = y_n + h f_n.$$

$$k = 2: \quad y_{n+1} = y_n + h \left(\frac{3}{2} f_n - \frac{1}{2} f_{n-1} \right).$$

$$k = 3: \quad y_{n+1} = y_n + h \left(\frac{23}{12} f_n - \frac{16}{12} f_{n-1} + \frac{5}{12} f_{n-2} \right).$$

$$k = 4: \quad y_{n+1} = y_n + h \left(\frac{55}{24} f_n - \frac{59}{24} f_{n-1} + \frac{37}{24} f_{n-2} - \frac{9}{24} f_{n-3} \right).$$

These methods are very interesting because γ_i^* 's do not depend on k . They can be determined directly from their recurrence relations (3.14). If we know the values of

$\gamma_0^*, \gamma_1^*, \gamma_2^*, \gamma_3^*$, we can create all k-step Adams-Bashforth methods with $k=1, 2, 3, 4$. By observing Equation (3.12) or (3.15), we can see that from a k-step Adams-Bashforth method, we can create a (k-1)-step Adams-Bashforth method just by throwing out the last term of the series. We also can create a (k+1)-step method by adding an extra term after calculating coefficient γ_{k+1}^* using their recurrence relations (3.14).

Thus, we have already shown that the k-step Adams-Bashforth methods only involve k terms (excluding y_n) of series(3.12), and is given as

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \gamma_j^* \nabla^j f_n.$$

which has order k and error constant γ_k^* . By investigating the Adams-Bashforth methods using test equation $y' = \lambda y$, Hairer and Wanner [56] said that the Adams-Bashforth methods are not appropriate for stiff problems.

Implicit Adams methods can be derived by interpolating the function $f(t,y)$ of $y' = f(t,y)$ over k points given in the explicit methods and one unknown point (t_{n+1}, f_{n+1}) . In this case the approximation function of $f(t,y)$ becomes

$$p(t) = p(t_n + sh) = \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j f_{n+1}. \quad (3.16)$$

By substituting $p^*(t)$ in (3.11) with $p(t)$, we have the implicit method

$$y_{n+1} = y_n + h \sum_{j=0}^k \gamma_j \nabla^j f_{n+1}. \quad (3.17)$$

where the coefficients γ_j satisfy

$$\gamma_j = (-1)^j \int_0^1 \binom{-s+1}{j} ds. \quad (3.18)$$

The multistep methods (3.17) are called implicit Adams-Moulton methods. Recurrence relations for coefficients γ_j (see Appendix C) are given as

$$\gamma_i + \frac{1}{2}\gamma_{i-1} + \frac{1}{3}\gamma_{i-2} + \frac{1}{4}\gamma_{i-3} + \dots + \frac{1}{i+1}\gamma_0 = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } i = 1, 2, \dots \end{cases} \quad (3.19).$$

The family of the implicit Adams-Moulton methods then can be written as

$$y_{n+1} = y_n + h(f_{n+1} - \frac{1}{2}\nabla f_{n+1} - \frac{1}{12}\nabla^2 f_{n+1} - \frac{1}{24}\nabla^3 f_{n+1} + \dots). \quad (3.20)$$

By considering the multistep method (3.20), special cases of Eq. (3.17) can be obtained:

$$k = 1: \quad y_{n+1} = y_n + h\left(\frac{1}{2}f_{n+1} + \frac{1}{2}f_n\right).$$

$$k = 2: \quad y_{n+1} = y_n + h\left(\frac{5}{12}f_{n+1} + \frac{8}{12}f_n - \frac{1}{12}f_{n-1}\right).$$

$$k = 3: \quad y_{n+1} = y_n + h\left(\frac{9}{24}f_{n+1} + \frac{19}{24}f_n - \frac{5}{24}f_{n-1} + \frac{1}{24}f_{n-2}\right).$$

$$k = 4: \quad y_{n+1} = y_n + h\left(\frac{251}{720}f_{n+1} + \frac{646}{720}f_n - \frac{264}{720}f_{n-1} + \frac{106}{720}f_{n-2} - \frac{19}{720}f_{n-3}\right).$$

The same arguments as in the Adams-Bashforth methods also prevail in implicit Adams-Moulton methods. But here, the k -step Adams-Moulton methods involve $(k+1)$ terms (excluding y_n) of series (3.17); That is, truncation is done after $(k+1)$ terms (excluding y_n) of series (3.17). For $k=1$, the backward Euler method $y_{n+1} = y_n + hf_{n+1}$ is also called a 1-step Adams-Moulton method. It also can be shown that a k -step Adams-Moulton has order $(k+1)$ and error constant γ_{k+1} . Shampine and Gordon [99] used these

implicit Adams-Moulton methods using PECE (will be discussed later) in developing ODEs packages to solve non-stiff differential equations. Byrne and Hindmarsh used Adams methods to solve nonstiff differential equations in the EPSODE [17].

The consistency, stability, and convergence of Adams-Bashforth and Adams-Moulton methods can be examined using the stability concept proposed by Dahlquist. Let us take an example for $k=2$. The explicit two-step Adams-Bashforth method is $y_{n+1} = y_n + (h/2)(3f_n - f_{n-1})$. By writing this formula in the form of Equation (3.3), we have $\rho^*(r) = r^2 - r$ and $\sigma^*(r) = (1/2)(3r - 1)$. Since $\rho^*(1) = 1 - 1 = 0$, and $\rho^{*'}(1) = 2(1) - 1 = (1/2)(3 \cdot 1 - 1) = \sigma^*(1)$, then the explicit two-step Adams-Bashforth method is consistent. The method is stable because the roots of the characteristic equation $\rho(r)$ are $r_1 = 0$ and $r_2 = 1$. By applying Equation (3.4) to this method, it can be said that this method has order 2. The implicit two-step Adams-Moulton method is $y_{n+1} = y_n + (h/12)(5f_{n+1} + 8f_n - f_{n-1})$. As was done for the explicit two-step Adams-Bashforth method, we have $\rho(r) = r^2 - r$, and $\sigma(r) = (1/12)(5r^2 + 8r - 1)$. Since $\rho(1) = 1 - 1 = 0$, and $\rho'(1) = 2(1) - 1 = (1/12)[5(1)^2 + 8(1) - 1] = \sigma(1)$, then the implicit two-step Adams-Moulton method is consistent. The method is stable because the roots of the characteristic $\rho(r)$ are $r_1 = 0$ and $r_2 = 1$. By applying Equation (3.4) to this method, it can be said that this method has order 3

3.3 Backward Differentiation Formulae (BDF)

So far, we have discussed multistep methods based on difference equations and integrations. We will now derive multistep methods which are based on numerical differentiation of a given function. These methods are called the backward differentiation formulas (BDF) and can be applied to solve stiff differential systems. These implicit multistep methods are the first methods used in solving stiff differential equations and become the most widely and predominantly used for all stiff computations since Gear [44] proposed the computer codes in his book. The EPSODE created by Byrne and Hindmarsh [17] also used these methods to handle stiff system problems. The MEBDF created by Cash [20] to solve stiff problems is also an example of package that implements BDF concepts.

The Newton Backward difference interpolations of k points $(t_{n+1}, y_{n+1}), (t_n, y_n), \dots, (t_{n-k+1}, y_{n-k+1}), (t_{n-k}, y_{n-k})$ are given as

$$p(t) = p(t_n + sh) = P_k(s) = \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j y_{n+1}. \quad (3.21)$$

The values $y_{n-k}, y_{n-k+1}, \dots, y_n$ are known, and we would like to determine the value of y_{n+1} . The methods are done by differentiating equation (3.21) and substituting the value into $y' = f(t, y)$. The function of y is approximated by a polynomial $p(t)$ of degree k . Since the approximation of $y' = f(t, y)$ at t_{n+1} is $p'(t_{n+1}) = f_{n+1}$, we have (see Appendix C)

$$\frac{1}{h} \frac{d}{ds} \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j y_{n+1} = f_{n+1},$$

which can be written as

$$\sum_{j=0}^k \delta_j \nabla^j y_{n+1} = hf_{n+1}.$$

By direct differentiating and substituting $s=1$, the coefficients δ_j' are obtained as

$$\delta_j = \begin{cases} 0, & \text{for } j = 0 \\ \frac{1}{j}, & \text{for } j \geq 1 \end{cases}.$$

The multistep methods then can be written as

$$\sum_{j=1}^k \frac{1}{j} \nabla^j y_{n+1} = hf_{n+1}. \quad (3.22)$$

These multistep formulas are known as backward differentiation formulae (BDF methods). Special cases of BDF can be obtained:

$$k=1: \quad y_{n+1} = y_n + hf_{n+1}.$$

$$k=2: \quad y_{n+1} = (4/3)y_n - (1/3)y_{n-1} + (2/3)hf_{n+1}.$$

$$k=3: \quad y_{n+1} = (18/11)y_n - (9/11)y_{n-1} + (2/11)y_{n-2} + (6/11)hf_{n+1}.$$

$$k=4: \quad y_{n+1} = (48/25)y_n - (36/25)y_{n-1} + (16/25)y_{n-2} - (3/25)y_{n-3} + (12/25)hf_{n+1}.$$

$$k=5: \quad y_{n+1} = (300/137)y_n - (300/137)y_{n-1} + (200/137)y_{n-2} - (75/137)y_{n-3} + \\ (12/137)y_{n-4} + (60/137)hf_{n+1}.$$

$$k=6: \quad y_{n+1} = (360/147)y_n - (450/147)y_{n-1} + (400/147)y_{n-2} - (225/147)y_{n-3} + \\ (72/147)y_{n-4} - (10/147)y_{n-5} + (60/147)hf_{n+1}.$$

Since finding the roots of the polynomial $\rho(r)$ of BDF methods (3.22) is not simple, Definition 3.1 can not be used directly to investigate the stability of BDF methods. By manipulating $\rho(r)$, Hairer and Wanner [56] have shown that k -step BDF-methods are stable only for $k \leq 6$.

3.4 Predictor-Corrector Methods

We have seen that the iterative procedures for the implicit multistep methods require initial guess of the solution at each timestep. The simple way to handle this problem is to use an explicit method to supply the required information. In this case, the explicit method used to supply the information is called a predictor and the implicit method used to correct the result is called a corrector. The combination of both predictor and corrector is called a predictor-corrector (PC) method. The predictor-corrector method consists of three processes: predicting (P) the next value using the explicit method, evaluating (E) the derivative based on the latest value of y , and correcting (C) the result using the implicit method. Normally, the algorithm of the predictor-corrector methods are in the form of $P(EC)^m$ if they end with a correction or $P(EC)^mE$ if they end with an evaluation. The important guidelines [78] that should be followed are:

- Avoid choosing the order of predictor greater than that of the corrector, because the local truncation error will still follow that of the corrector and
- Manage to choose the order of predictor equal to the order of the corrector, so we can avoid unnecessary computations.

Suppose we use an implicit linear multistep method to solve the standard initial value problem $y' = f(t, y)$. An implicit linear multistep method can be written as

$$y_{n+k} + \sum_{j=0}^{k-1} a_j y_{n+j} = h b_k f(t_{n+k}, y_{n+k}) + h \sum_{j=0}^{k-1} b_j f_{n+j} \quad (3.23)$$

and a linear multistep predictor as

$$y_{n+k} + \sum_{j=0}^{k-1} a_j y_{n+j} = h \sum_{j=0}^{k-1} b_j f_{n+j} \quad (3.24)$$

We are required to determine y_{n+k} from the formulas. Since we cannot solve y_{n+k} directly, we can use the following procedure to calculate y_{n+k} :

P : Predict the value $y_{n+k}^{(0)}$ for y_{n+k} using (3.24).

E : Evaluate $f(t_{n+k}, y_{n+k}^{(0)})$.

C : Correct $y_{n+k}^{(0)}$ to obtain a new $y_{n+k}^{(1)}$ for y_{n+k} using (3.23).

E : Evaluate $f(t_{n+k}, y_{n+k}^{(1)})$.

C : Correct $y_{n+k}^{(1)}$ to obtain a new $y_{n+k}^{(2)}$ for y_{n+k} using (3.21)

.

.

The sequence of operations PECECE . . . to determine the value of y_{n+k} is given as $y_{n+k}^{(0)}, y_{n+k}^{(1)}, y_{n+k}^{(2)}, \dots$. Lambert [78] proposed a single mode $P(EC)^m E^{1-t}$ instead of two modes $P(EC)^m E$ or $P(EC)^m$ as follows:

$P(EC)^m E^{1-t}$:

$$P: y_{n+k}^{[0]} + \sum_{j=0}^{k-1} a_j y_{n+j}^{[m]} = h \sum_{j=0}^{k-1} b_j f_{n+j}^{[m-t]}$$

$$f_{n+k}^{[v]} = f(t_{n+k}, y_{n+k}^{[v]})$$

$(EC)^m$:

$$y_{n+k}^{[v+1]} + \sum_{j=0}^{k-1} a_j y_{n+j}^{[m]} = h b_k f_{n+k}^{[v]} + h \sum_{j=0}^{k-1} b_j f_{n+j}^{[m-t]}$$

} $v = 0, 1, \dots, m-1$

$$E^{(1-t)}: f_{n+k}^{[m]} = f(t_{n+k}, y_{n+k}^{[m]}), \text{ if } t = 0$$

where $t = 0$ or 1 . He also has shown that if p and p^* are the order of corrector and predictor respectively, then

1. if $p^* \geq p$ (or if $p^* < p$ and $m > p - p^*$), the PC method and the corrector have the same order and the same principal local truncation error,
2. if $p^* < p$ and $m = p - p^*$, the PC method and the corrector have the same order but different principal local truncation error, and
3. if $p^* < p$ and $m \leq p - p^* - 1$, the order of the PC method is $p^* + m$ ($< p$).

Based on Lambert [78], most of modern predictor-corrector codes for non-stiff problems use Adams-Moulton methods as correctors and Adams-Bashforth methods as predictors. These methods are sometimes called ABM methods. Considering the principal local truncation error given above, it is important then to choose the predictors and correctors with the same order. To satisfy this condition, the ABM methods should be:

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \gamma_j^* \nabla^j f_n, \quad p^* = k.$$

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \gamma_j \nabla^j f_{n+1}, \quad p = k$$

3.5 Extrapolation Methods for Solving ODEs

The meaning of extrapolation in the sense of integration methods is the process of deriving an improved estimate for the value of a definite integral based on two or more applications of a formula using different stepsizes of integration. The following example

is an illustration of the meaning of extrapolation. Assume $y_n(h)$ is the approximate solution of $y' = f(t,y)$ which is integrated using stepsize h , then y_n can be written as $y_n(h) = y(t_n) + h^p \varepsilon(t_n) + O(h^{p+1})$, where $y(t_n)$ is the true solution, $\varepsilon(t)$ is called the magnified error function. We would like to manipulate h so that we have a more accurate estimate to the true value of $y(t_n)$. Let us take a new stepsize λh ($0 < \lambda < 1$), and apply to the formula of y_n . Then, we have $y_n(\lambda h) = y(t_n) + (\lambda h)^p \varepsilon(t_n) + O(h^{p+1})$. By multiplying $y_n(h)$ with $-\lambda^p$ and then adding to $y_n(\lambda h)$, we have

$$y_n(\lambda h) - \lambda^p y_n(h) = (1 - \lambda^p) y(t_n) + O(h^{p+1}),$$

and also

$$y_n(\lambda h) + [y_n(\lambda h) - y_n(h)] / [(1/\lambda)^p - 1] = y(t_n) + O(h^{p+1}).$$

If we choose $y_n(\lambda h) + [y_n(\lambda h) - y_n(h)] / [(1/\lambda)^p - 1]$ as the approximate value of $y(t_n)$, the error becomes smaller. This approximate value is called extrapolation or, sometimes Richardson's extrapolation. Assume that a certain numerical formula has a truncation error proportional to h^3 . Suppose a computation of this formula with $h = 0.10$ gives an output 4.5800 and with $h = 0.05$ gives an output 4.4350. What is the best approximation to the true value based upon this information? In this problem $\lambda = 1/5$ and $p = 3$. Using the formula given above, the approximate solution is

$$\begin{aligned} y(0.05) + [y(0.05) - y(0.10)] / [(1/5)^3 - 1] &= 4.4350 + [4.4350 - 4.5800] / [124] \\ &= 4.4288. \end{aligned}$$

In most of implementation, usually the factor λ is chosen $1/2$, so the Richardson's extrapolation at point t_n is

$$y(t_n) = y_n\left(\frac{h}{2}\right) + \frac{y_n\left(\frac{h}{2}\right) - y_n(h)}{2^p - 1} + O(h^{p+1}) \quad (3.25)$$

We have seen that the extrapolation depends on the time step h . This procedure can be done repeatedly until $h \rightarrow 0$ so that the approximate value tends to the exact solution. Deuflhard [32,33] has shown some extrapolation methods including control order and stepsize.

Let $y(t)$ be the true solution of $y' = f(t,y)$, $y(t_0) = y_0$, and $y(t,h)$ be an approximate solution satisfied by choosing the stepsize h to the appropriate numerical method. Assume $y(t,h)$ can be written as an asymptotic error expansion in powers of h ,

$$y(t,h) = y(t) + a_1 h^{r_1} + a_2 h^{r_2} + a_3 h^{r_3} + \dots + a_m h^{r_m} + a_{m+1} h^{r_{m+1}} + \dots$$

where $0 < r_1 < r_2 < \dots < r_m$, a_1, a_2, a_3, \dots are independent of h and determined by evaluating $y(t,h)$ over stepsize h_i , $i=0, 1, 2, \dots$. One approach can be obtained by taking $r_i = ir$ and the stepsize sequence h_i such that $h_0 > h_1 > h_2 > \dots$. Substituting values r_i , we have

$$y(t,h) = y(t) + a_1 h^r + a_2 h^{2r} + a_3 h^{3r} + \dots + a_m h^{mr} + \dots$$

If the superscript k is related to the approximate value with stepsize h_k and the subscript m is related to the number of times the linear combination has been applied to eliminate a_1, a_2, \dots, a_m , and $Y_m^{(k)}$ is defined as

$$Y_m^{(k)} = \frac{h_k^r Y_{m-1}^{(k+1)} - h_{k+m}^r Y_{m-1}^{(k)}}{h_k^r - h_{k+m}^r}, \quad (3.26)$$

then we will have

$$Y_m^{(k)} = y(t) + (-1)^m h_k^r h_{k+1}^r h_{k+2}^r \dots h_{k+m}^r (a_{m+1} + O(h_k^r)).$$

For $m \rightarrow \infty$, $Y_m^{(k)}$ will close to $y(t)$. The representation of this extrapolation method is shown in Table 3.2. The Table is expressed in a graphical table as

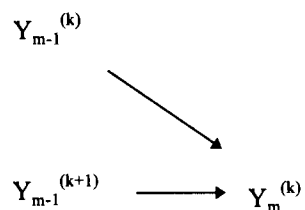


TABLE 3.2
Table of Extrapolation

$Y_0^{(0)} = y(t, h_0)$					
$Y_0^{(1)} = y(t, h_1)$	$Y_1^{(0)}$				
$Y_0^{(2)} = y(t, h_2)$	$Y_1^{(1)}$	$Y_2^{(0)}$			
$Y_0^{(3)} = y(t, h_3)$	$Y_1^{(2)}$	$Y_2^{(1)}$	$Y_3^{(0)}$		
$Y_0^{(4)} = y(t, h_4)$	$Y_1^{(3)}$	$Y_2^{(2)}$	$Y_3^{(1)}$	$Y_4^{(0)}$	
.
.

Equation (3.26) can be written as

$$Y_m^{(k)} = Y_{m-1}^{(k+1)} + \frac{h_{k+m}^r (Y_{m-1}^{(k+1)} - h_{k+m}^r Y_{m-1}^{(k)})}{h_k^r - h_{k+m}^r} \quad (3.27)$$

For $h = h_0 \lambda^i$ with $0 < \lambda < 1$, we have

$$Y_m^{(k)} = Y_{m-1}^{(k+1)} + \frac{b^{mr} (Y_{m-1}^{(k+1)} - Y_{m-1}^{(k)})}{1 - b^{mr}}.$$

For $b=1/2$, we have

$$Y_m^{(k)} = \begin{cases} Y_{m-1}^{(k+1)} + \frac{Y_{m-1}^{(k+1)} - Y_{m-1}^{(k)}}{2^m - 1}, & \text{if } r = 2 \\ Y_{m-1}^{(k+1)} + \frac{Y_{m-1}^{(k+1)} - Y_{m-1}^{(k)}}{4^m - 1}, & \text{if } r = 4 \end{cases} \quad (3.28)$$

Equation (3.28) is sometimes called Romberg integration.

Let us take an example $y' = -y$, $y(0) = 1$. We will interpolate this problem until $t = 1$ using Euler extrapolation with $r = 2$ and the number of points in the interval chosen to be $m = 10$. After that we approximate the problem using Euler extrapolation with $m = 5$, Euler method and ABM method of order 4 with $h = 0.25$, and integrate until $t = 2.5$. For the first Euler extrapolation, we choose $h_i = (1-0) \cdot 2^{-i}$, $i=0,1, \dots,9$. Using Euler method, we have a relation $y_{n+1} = y_n + hf(t_n, y_n) = (1-h)y_n$. Substituting y_0 , we have a relation $y_n = (1-h)^n$. To reduce the use of storage, we can use variable y with dimension as much as m . A procedure of the Romberg extrapolation is represented as:

$$\begin{array}{lll} y_0 = y_0^{(0)} & & \\ y_1 = y_0^{(1)} & y_1 = y_1 + [y_1 - y_0]/[2^1 - 1] & \\ y_2 = y_0^{(2)} & y_2 = y_2 + [y_2 - y_1]/[2^1 - 1] & y_2 = y_2 + [y_2 - y_1]/[2^2 - 1] \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ y_{(m-1)} = y_0^{(m-1)} & y_{(m-1)} = y_{(m-1)} + [y_{(m-1)} - y_{(m-2)}]/[2^1 - 1] & y_{(m-1)} = y_{(m-1)} + [y_{(m-1)} - y_{(m-2)}]/[2^2 - 1] \dots \end{array}$$

An algorithm of Romberg integration using Euler method is given as:

1. $h_0 = (b-a)$ /* a and b are starting and final times respectively */
2. Compute $y(i)$, $i = 0, 1, 2, \dots, m-1$
 - for $i = 0$ to $m-1$ do
 - $h = h_0 * 2^{-i}$
 - $n = 2^i$
 - $y(i) = (1-h)^n$ /* With stepsize h, $y(1)$ is achieved after $n=2^i$ steps */
 - end do
3. Extrapolate the values
 - for $i = 1$ to $m-2$ do
 - $k = m-1$
 - for $j=i+1$ to $m-1$ do
 - $y(k) = y(k) + [y(k) - y(k-1)]/[2^i - 1]$
 - $k = k-1$
 - end do
 - end do

The computation table of the algorithm for Romberg integration using Euler method given above for the case of $y' = -y$, $y(0) = 1$ is given in Table 3.3.

The table is derived as follows: Let us take $m = 4$ and $i = 5$.

$$\begin{aligned}
 Y_m^{(i)} &= Y_{m-1}^{(i)} + [Y_{m-1}^{(i)} - Y_{m-1}^{(i-1)}] / [2^m - 1] \\
 &= Y_3^{(5)} + [Y_3^{(5)} - Y_3^{(4)}] / [2^4 - 1]
 \end{aligned}$$

$$= 0.3678811228 + [0.3678811228 - 0.367920598]/15 = 0.367878603.$$

TABLE 3.3
Romberg Representation of $y' = -y$, $y(0)=1$

i	h	m	$y_i, m=0$	$y_i, m=1$	$y_i, m=2$	$y_i, m=3$
0	$h_0 2^0$	2^0	0.0			
1	$h_0 2^{-1}$	2^1	0.25	0.500000000		
2	$h_0 2^{-2}$	2^2	0.316406250	0.382812500	0.343750000	
3	$h_0 2^{-3}$	2^3	0.343608916	0.370811582	0.366811275	0.370105743
4	$h_0 2^{-4}$	2^4	0.356074130	0.368539345	0.367781933	0.367920598
5	$h_0 2^{-5}$	2^5	0.362055289	0.368036448	0.367868816	0.367881228
6	$h_0 2^{-6}$	2^6	0.364986524	0.367917759	0.367878196	0.367879536
7	$h_0 2^{-7}$	2^7	0.366437716	0.367888908	0.367879290	0.367879447
8	$h_0 2^{-8}$	2^8	0.367159755	0.367881794	0.367879423	0.367879442
9	$h_0 2^{-9}$	2^9	0.367519891	0.367880028	0.367879439	0.367879441

$y_i, m=4$	$y_i, m=5$	$y_i, m=6$	$y_i, m=7$

0.367774922			
0.367878603	0.367881947		
0.367879424	0.367879441	0.367879410	
0.367879441	0.367879441	0.367879441	0.367879441
0.367879441	0.367879441	0.367879441	0.367879441
0.367879441	0.367879441	0.367879441	0.367879441

For the second computation of Euler extrapolation, we choose $m = 5$, with stepsize for all methods $h = 0.25$. The computation results is given in Table 3.4. It can be shown that we can always control the accuracy of results produced by extrapolation methods, just by controlling the stepsizes over the entire range of computations. Graphical representations of these computations compared to those resulted by Euler method and Adams method of order 4 are shown in Figure 3.2. The error vs time t of this extrapolation method tends to be accumulated as shown in Figure 3.2.

TABLE 3.4
The Computation Results of $y' = -y$, $y(0)=1$
Using Extrapolation, Euler, and ABM-4 Methods

t	Exact	Extrapolation	Euler	ABM-4
0.0	1.000000000	1.000000000	1.000000000	1.000000000
0.25	0.778800783	0.778800747	0.750000000	0.778808594

0.50	0.606530660	0.606528620	0.562500000	0.606542826
0.75	0.472366553	0.472345756	0.421875000	0.472380765
1.00	0.367879441	0.367774922	0.316406250	0.385532323
1.25	0.286504797	0.286148404	0.237304688	0.299920011
1.50	0.223130160	0.222179660	0.177978516	0.233222895
1.75	0.173773943	0.171635077	0.133483887	0.181836888
2.00	0.135335283	0.131086662	0.100112915	0.141601456
2.25	0.105399225	0.097729534	0.075084686	0.110261898
2.50	0.082084999	0.069250702	0.056313515	0.085867179

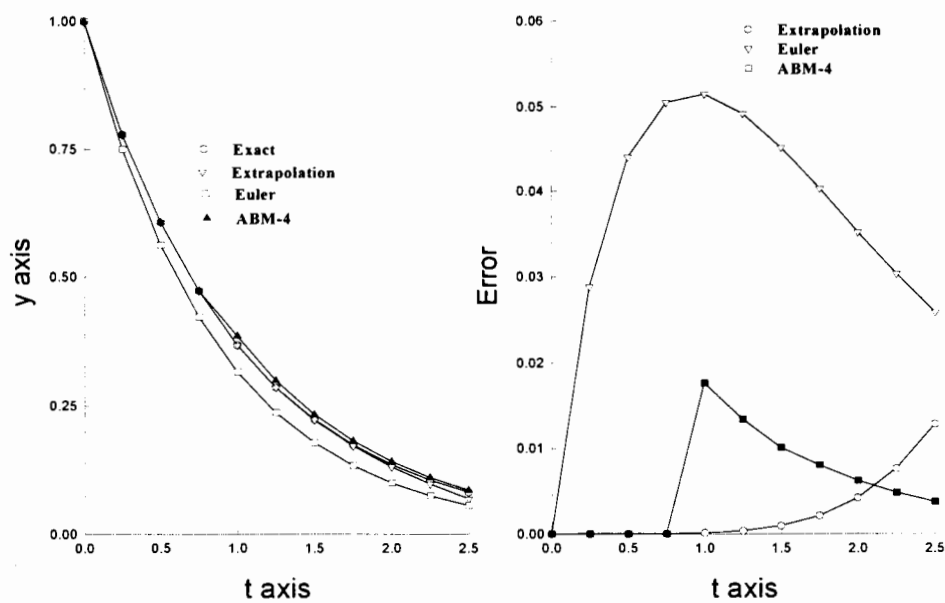


Figure 3.2 Computation Results of $y' = -y$, $y(0)=1$ Using Extrapolation, Euler, and ABM-4 Methods

CHAPTER IV

STIFF ORDINARY DIFFERENTIAL EQUATION

Dealing with system of differential equations will sometimes lead us into stiffness problem. This possibility occurs especially when the solution contains a fast mode which decreases very fast in response to initial conditions or a changing input and a slow mode which needs longer computing time for its transient to decay. This situation commonly occurs when standard numerical methods are implemented to find the approximate solutions of a differential equation having the exact solution containing a term of the form $e^{\lambda t}$, where λ is a complex number with negative real part. As mentioned before, this term will tend to zero when t increases to infinity, but unfortunately its approximation generally will not always have the same property, except when the stepsize is chosen such that the method used is still stable. The problem will become serious when the exact solution has a steady-state term that does not grow significantly with t , and a transient term that decays fast to zero. In this case, the numerical methods can approximate the steady state solution, but they need attention when they are applied to approximate the decaying transient term that can dominate the computation, and can produce incorrect results. A standard numerical method can be used to solve stiff problems, but the cost will be very high since we have to maintain an

extremely small stepsize for the entire of integration [13,105]. The two following problems will help us to understand the stiffness concepts in ordinary differential equations.

Problems 4.1:

Consider the following set of equations

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -100 & -101 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Discussion:

The eigenvalues of the matrix

$$A = \begin{pmatrix} 0 & 1 \\ -100 & -101 \end{pmatrix}$$

are $\lambda = -1$ and $\lambda = -100$. These eigenvalues and the initial conditions give solution

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} e^{-t} \\ -e^{-t} \end{pmatrix}$$

From this exact solution, we cannot see any problem at all, but when the problem is solved numerically, the unstable computation arises unless a small stepsize is used. The corresponding solution resulting from the eigenvalues are e^{-t} and e^{-100t} as shown in Figure 4.1. The initial condition has thrown the e^{-100t} term out from the exact solution. By considering the eigenvalues, we know that the dependent variables contain both fast and slow components. In this case, e^{-100t} decays fast to zero, while e^{-t} decreases to zero very slowly. To maintain the stability of a simple Euler method, the stepsize should be chosen such that $|1+\lambda_{\min}h| \leq 1$. Therefore, the stepsize of the computation should be maintained within the interval $|1-100h| \leq 1$ or $0 \leq h \leq 2/100 = 0.020$.

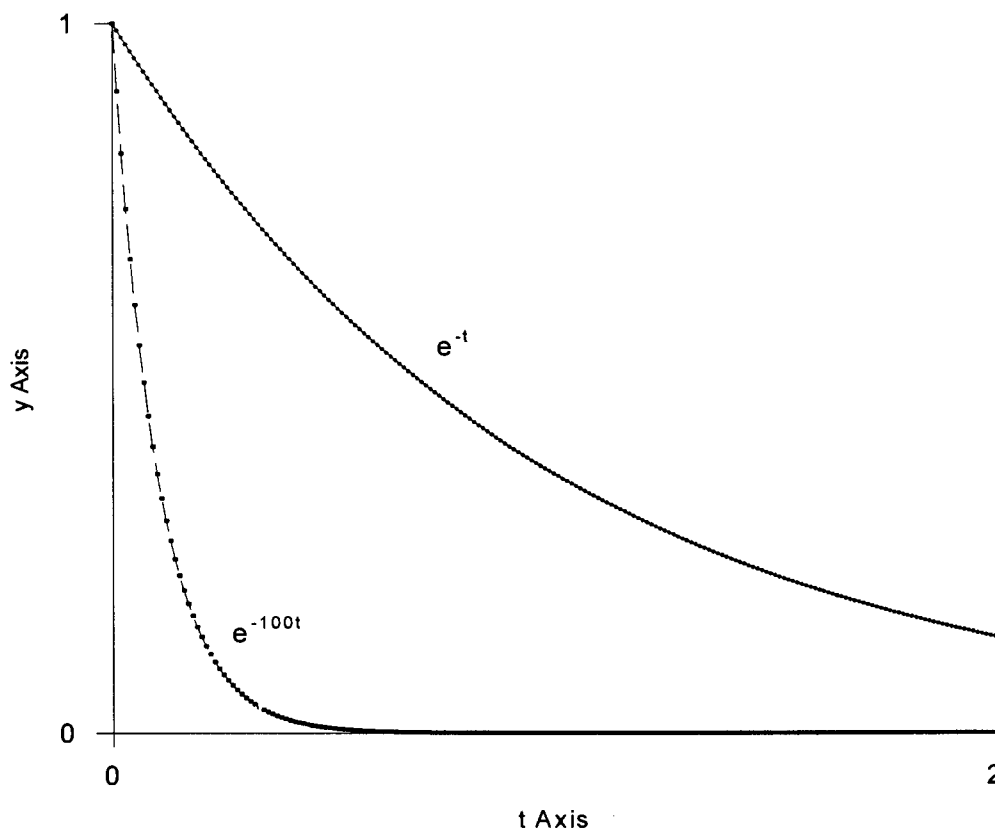


Figure 4.1 Graphical Solutions of e^{-t} and e^{-100t}

Problem 4.2:

Consider the following set of equations

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -4 & -5 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Discussion:

The eigenvalues of the matrix

$$A = \begin{pmatrix} 1 & 2 \\ -4 & -5 \end{pmatrix}$$

are $\lambda = -1$ and $\lambda = -3$. The corresponding solutions resulting from the eigenvalues are e^{-t} and e^{-3t} respectively. These eigenvalues and the initial conditions give solution

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} e^{-t} \\ -e^{-t} \end{pmatrix}$$

Using Euler method, the absolute stability can be maintained if $|1 + \lambda_{\min}h| \leq 1$. Thus, the stepsize of computation should be maintained in the interval $|1 - 3h| \leq 1$ or $0 \leq h \leq 2/3$.

Both problems have the same exact solutions, but they behave very differently when they are solved numerically. In this case, we need to apply different stepsizes to compute the two problems using Euler method. The cost of Problem 4.1 is far more than that of Problem 4.2 when they are solved numerically. The cost for integrating Problem 4.1 from $t_0 = 0$ to $t_f = 50.0$ is at least 2500 steps (for $h = 0.02$), while the cost for integrating Problem 4.2 from $t_0 = 0$ to $t_f = 50.0$ is at least 75 steps (for $h=2/3$). The phenomenon illustrated in these two problems is known as stiffness. Problem 4.1 is stiff, while Problem 4.2 is non-stiff. Since both problems have the same exact solutions, the phenomenon should be a property of the differential system itself.

Even though the exact solution for eigenvalue $\lambda = -100$ of Problem 4.1 contributes practically nothing to the solution of y_1 and y_2 , the criterion of absolute stability forces us to use an extremely small value of h over the entire range of integration. Therefore, it can be said that using the same numerical method, the computation time needed to solve a highly stiff differential system is more expensive than that of non-stiff one. The main objective of a good ODE solver is to minimize the computing cost with subject to the required tolerance and the stability of computation. In

order to gain this objective, an ODE solver is usually equipped with tools for: detection stiffness, determining a starting stepsize, and controlling stepsizes and formulae, by considering the required tolerance.

4.1 Stiff Differential Problems

The study of stiff systems has become popular in the last twenty years because they arise in many applications such as: radio technology, chemical kinetics, reactor and radiation physics, hydrodynamics, process dynamic and control, electrical circuits, diffusion, beam, and lasers. Several methods have been proposed and analyzed carefully, but only few of the codes have been developed until recently. In this thesis, the investigation will be done based on stiff differential equation packages such as MEBDF, LSODE, EPSODE, and VODE, using the test cases:

1. Kidney problems introduced by Scott and Watts in the form [17, 21, 79]

$$\begin{aligned}
 y_2 y_1' &= a y_1 (y_3 - y_1), & y_1(0) &= 1 \\
 y_2' &= -a (y_3 - y_1) & y_2(0) &= 1 \\
 y_4 y_3' &= b - c (y_3 - y_5) - a y_3 (y_3 - y_1) & y_3(0) &= 1 \\
 y_4' &= a (y_3 - y_1) & y_4(0) &= -10 \\
 y_5' &= -c (y_5 - y_3) / d & y_5(0) &= \lambda
 \end{aligned}$$

where $a = 100$, $b = 0.9$, $c = 1000$, $d = 10$, $0 \leq t \leq 1$, with initial conditions

Problem	λ
G1	0.9902688359
G2	0.9902834990

G3	0.9925211341
G4	1.0304879856
G5	0.99
G6	0.9
G7	0

2. An autocatalytic reaction pathway proposed by Robertson [21, 56]

$$\begin{aligned}
 y_1' &= -0.04 y_1 + 10^4 y_2 y_3, & y_1(0) &= 1 \\
 y_2' &= 0.04 y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2, & y_2(0) &= 0 \\
 y_3' &= 3 \cdot 10^7 y_2^2, & y_3(0) &= 0
 \end{aligned}$$

for $0 \leq t \leq 4 \cdot 10^8$.

3. Problem D4 of Enright et al. [39, 103]

$$\begin{aligned}
 y_1' &= -0.013 y_1 - 1000 y_1 y_3 & y_1(0) &= 1 \\
 y_2' &= -2500 y_2 y_3 & y_2(0) &= 1 \\
 y_3' &= 0.013 y_1 - 1000 y_1 y_3 - 2500 y_2 y_3 & y_3(0) &= 0
 \end{aligned}$$

where $0 \leq t \leq 50$.

4. Problem proposed by Gupta and Wallace [53]

$$\begin{aligned}
 y_1' &= v y_1 - w y_2 + (-v + w + 1)e^t & y_1(0) &= 1 \\
 y_2' &= w y_1 + v y_2 + (-v - w + 1)e^t & y_2(0) &= 1
 \end{aligned}$$

The exact solution is

$$\begin{aligned}
 y_1 &= c_1 e^{vt} \cos(wt + c_2) + e^t \\
 y_2 &= c_1 e^{vt} \sin(wt + c_2) + e^t
 \end{aligned}$$

where $v = -80$, $w = 8$, $0 \leq t \leq 10.0$.

4.2 Stiffness Concepts

There are two approaches that have been taken in order to define stiffness. The first approach is to define stiffness based on various aspects of phenomena of stiffness. The second approach is to define stiffness in mathematical terms. Even though several definitions relating to mathematical terms of stiffness have been proposed, but they are all still in debate until now. Stiffness can not be defined precisely in mathematical terms [1, 62, 78], even for the class of linear constant coefficient systems. The following differential equation concepts are useful in order to understand the stiffness concepts.

The homogeneous solutions of ODE problems are also called transient solutions, and the particular solutions are called steady-state solutions. For the cases of linear systems, the homogeneous solutions correspond to the characteristic roots λ_i of their matrices. If $\text{Re}(\lambda_i) < 0$ and $|\text{Re}(\lambda_i)|$ is large, the corresponding homogeneous solution will decay fast as t increases and is thus called a fast transient, but if the $|\text{Re}(\lambda_i)|$ is small, the corresponding homogeneous solution will decay slowly and is called a slow transient. Suppose $\bar{\lambda}$ and $\underline{\lambda}$ in $\{\lambda_i, i = 1, \dots, n\}$ are defined by $|\text{Re}(\underline{\lambda})| \leq |\text{Re}(\lambda_i)| \leq |\text{Re}(\bar{\lambda})|, \forall i$. We will have trouble when $|\text{Re}(\bar{\lambda})|$ is very large and $|\text{Re}(\underline{\lambda})|$ is very small, because we are forced to integrate over the whole range of integration with a very small stepsize.

The following discussion of stiffness concepts will be based on Lambert's explanations [78]. If we take the ratio $|\operatorname{Re}(\bar{\lambda})|/|\operatorname{Re}(\underline{\lambda})|$ as the stiffness ratio, the statement as a candidate for the definition of stiffness is given as follows:

Statement 1:

A linear constant coefficient system is stiff if all its eigenvalues have negative real part and the stiffness ratio is large ($\gg 1$).

This statement has been used by Lambert [1973] as a definition for stiffness, but now he realizes that the definition is not correct any longer. He used the following three systems to show that the definition is not satisfactory with the phenomenon.

System 4.1:

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} -2 & 1 \\ 998 & -999 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + \begin{pmatrix} 2 \sin t \\ 999(\cos t - \sin t) \end{pmatrix}$$

The general solution of this system is given as

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = c_1 e^{-t} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + c_2 e^{-1000t} \begin{pmatrix} 1 \\ -998 \end{pmatrix} + \begin{pmatrix} \sin t \\ \cos t \end{pmatrix}$$

The stepsize interval of stability of this system using an Euler method is $0 \leq h \leq 2/1000$.

System 4.2:

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} -2 & 1 \\ -1.999 & 0.999 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + \begin{pmatrix} 2 \sin t \\ 0.999(\sin t - \cos t) \end{pmatrix}$$

The general solution of this system is given as

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = c_1 e^{-t} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + c_2 e^{-0.001t} \begin{pmatrix} 1 \\ 1.999 \end{pmatrix} + \begin{pmatrix} \sin t \\ \cos t \end{pmatrix}$$

The stepsize interval of stability of this system using an Euler method is $0 \leq h \leq 2$.

System 4.3:

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} -0.002 & 0.001 \\ 0.998 & -0.999 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + \begin{pmatrix} 0.002 \sin(0.001t) \\ 0.999[\cos(0.001t) - \sin(0.001t)] \end{pmatrix}$$

The general solution of this system is given as

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = c_1 e^{-t} \begin{pmatrix} 1 \\ -998 \end{pmatrix} + c_2 e^{-0.001t} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} \sin(0.001t) \\ \cos(0.001t) \end{pmatrix}$$

The stepsize interval stability of this system using an Euler method is $0 \leq h \leq 2$.

It is easy to show that the three systems have the same stiffness ratios: 1000. Even though they have the same stiffness ratios, System 4.1 and System 4.2 have different stepsizes in order to maintain the absolute stability of computations. No matter how we choose an initial condition for System 4.1, the numerical method needs a very small stepsize for integration. The effect of stiffness is that we have to continue using that small stepsize long after the fast transient solution has no influence. In this case, System 4.1 is called a genuine stiff [78]. Suppose we equip System 4.2 with initial conditions: $y(0) = [2, 3.999]^T$ causing $c_1 = c_2 = 1$. It is obvious that if we want to reach steady state solutions, the transient solutions should vanish. For System 4.2, this is only possible if the solution $e^{-0.001t}$ is close to zero. In this case, the total computation step of System 4.2 is comparable to that of System 4.1, because we still need to compute $e^{-0.001t}$ until it has no effect on the solution of System 4.2. In this situation, the definition of stiffness in Statement 1 seems to be consistent. The conclusion will be different when the initial conditions for System 4.2 is changed to be: $y(0) = [2, 3]^T$ causing $c_1 = 2$, and

$c_2=0$. Using these initial conditions, the slow transient term disappears from the exact solution so there is no need to integrate a long way until the steady state condition is reached. From these two initial conditions, we see that the definition of Statement 1 is not consistent any longer, because it depends also on initial conditions imposed by a particular problem. In this situation, System 4.2 is not genuinely stiff, but we can call them pseudo-stiff. By doing the same evaluation, it can be said that System 4.1 is also pseudo-stiff. System 4.3 can be changed to System 4.1 by making the transformation $t = 1000\tau$. In this case, System 4.3 is stiff in exactly the same sense as System 4.1.

Statement 2:

Stiffness occurs when stability is more of a constraint than accuracy.

This statement also fails, because the stability also depends on the accumulation of errors. By examining the local error at the very first step, it can be said that a stiff problem has a substantially higher local error than that of a non-stiff problem.

Statement 3:

Stiffness occurs when some components of the solution decay much more rapidly than others.

This statement also fails, because it is merely based on comparison of the rate of change of the fastest transient solution with that of the steady-state solution. From the discussion of Problem 4.1, this statement fails totally.

Definition 4.1:

If a numerical method with a finite region of absolute stability, applied to a system with any initial conditions, is forced to use in a certain interval of

integration a stepsize which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be stiff in that interval.

This definition can distinguish between the genuine stiff and the pseudo-stiff systems, and it offers the idea that stiffness can be considered to change over the interval of integration. To get a good understanding of this definition, we have to do some computations, otherwise it will be difficult to understand.

Statement 4:

A system is said to be stiff in a given interval of t if in that interval the neighboring solution curves approach the solution curve at a rate which is very large in comparison with the rate at which the solution varies in that interval.

The same case with Definition 4.1 occurs in Statement 4.

4.3 Stability Concepts of Numerical Stiffness Methods

Several definitions of stability related to stiff systems are given in order to satisfy the needs of smaller stepsize more for stability than for accuracy.

Definition 4.2:

A method is said to be A-stable in the sense of Dahlquist if when it is applied to the test problem $y' = \lambda y$, the solutions converge to 0 as $n \rightarrow \infty$ or the region of stability is a superset of $\{\lambda h \mid \text{Re}(\lambda h) < 0\}$. A minimum region of this stability is shown in Figure 4.2.

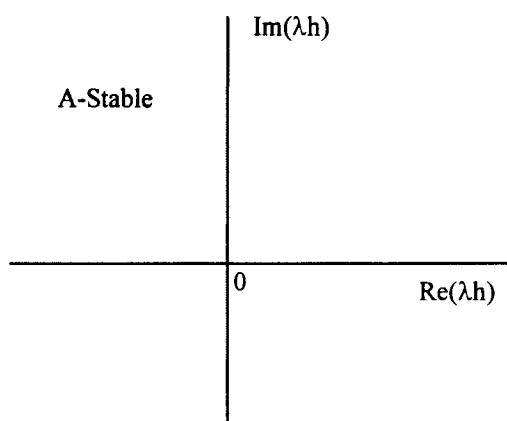


Figure 4.2 A representation of a Minimum Region of A-stability

It can be shown that a trapezoidal formula, a second-order Adams-Moulton method $y_{n+1} = y_n + (h/2)(y_{n+1}' + y_n')$, is A-stable. By substituting the test function $y' = \lambda y$ into the trapezoidal formula, we have $y_n = [(2+\lambda h)/(2-\lambda h)]^n y_0$. The region of stability is reached if $|2 + \lambda h|/|2 - \lambda h| \leq 1$. Here, the region is the set of points z such that the distance between z and the point $z = 2$ is greater than the distance between z and the point $z = -2$. This region is $\{z \mid \text{Re}(z) \leq 0\}$. That is, the trapezoidal formula is A-stable. The backward (implicit) Euler method $y_{n+1} = y_n + h y_{n+1}'$ is also A-stable (see Chapter 2.1).

Definition 4.3:

A method is said to be $A(\alpha)$ -stable, $\alpha \in (0, \pi/2)$ in the sense of Widlund if when it is applied to the test problem $y' = \lambda y$, the solutions converge to 0 as $n \rightarrow \infty$ with h fixed for all $|\arg(\lambda h)| < \alpha$, $|\lambda| \neq 0$ or the region of stability is a superset of $\{\lambda h \mid -\alpha < \pi - \arg(\lambda h) < \alpha\}$; it is said to be $A(0)$ -stable if it is $A(\alpha)$ -stable for some $\alpha \in (0, \pi/2)$. Figure 4.3 illustrates a minimum region of this stability.

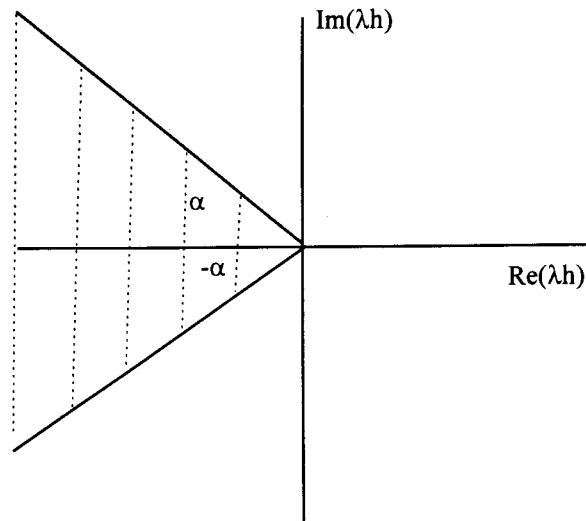


Figure 4.3 A Representation of a Minimum Region of $A(\alpha)$ -Stability

Definition 4.4:

A method is said to be A_0 -stable if the region of stability is a superset of $\{\lambda h \mid \text{Re}(\lambda h) < 0, \text{Im}(\lambda h) = 0\}$. Figure 4.4 illustrates a minimum region of this definition.

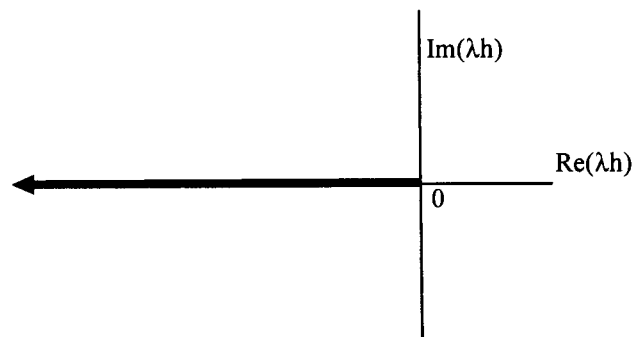


Figure 4.4 A Representation of a Minimum Region of A_0 -Stability

Definition 4.5:

A method is said to be stiffly stable if $R_1 \cup R_2$ is a subset of the region of stability, where $R_1 = \{\lambda h \mid \operatorname{Re}(\lambda h) < -a\}$ and $R_2 = \{\lambda h \mid -a \leq \operatorname{Re}(\lambda h) < 0, -c \leq \operatorname{Im}(\lambda h) \leq c\}$, and a and c are positive real numbers. Figure 4.5 illustrates this definition.

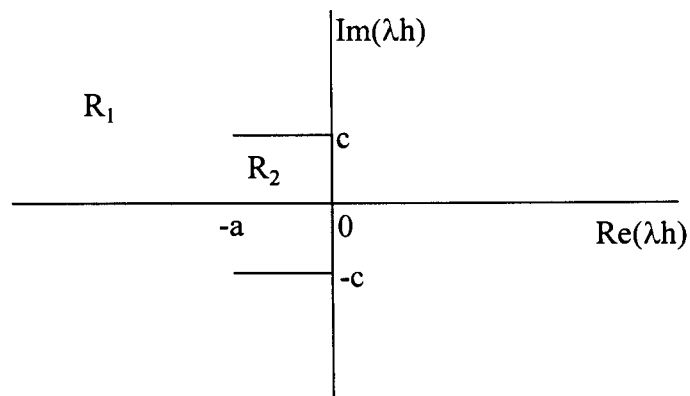


Figure 4.5 A Representation of a Minimum Region of a Stiffly-Stable Method

Definition 4.6

A one-step method is said to be L-stable if it is A-stable and, in addition, when applied to the scalar test equation $y' = \lambda y$, λ a complex constant with $\operatorname{Re}(\lambda) < 0$, it yields $y_{n+1} = E(\lambda h)y_n$, where $|E(\lambda h)| \rightarrow 0$ as $|\lambda h| \rightarrow -\infty$.

From the definitions, the hierarchy of stability is drawing as follows: L-stable \rightarrow A-stable \rightarrow Stiffly-stable \rightarrow $A(\alpha)$ -stable \rightarrow $A(0)$ -stable \rightarrow A_0 -stable.

4.4 A Modified Euler Method for Solving ODEs

Since most of the numerical methods used to solve stiff differential problems are implicit methods, it is almost impossible to have a cheap low-accuracy code for stiff ODEs. The meaning of cheap here is that the methods are easy to program and the computing time for solving the problems is not too expensive. It is true that if one persistently solves stiff problems, one should use one of the better codes which are now accessible. But if one needs to compute stiff problems where the accuracy is not taken into consideration especially for a quick low-accuracy solution, Lambert [77] proposed a method to satisfy that requirement.

The idea that Lambert suggested is to apply an automatically determined sequence of variable stepsizes with Euler's rule. Unfortunately, there is no guarantee that this method can be successfully used in order to satisfy the meaning of cheap. Lambert claims that this method is stable and the solution is not unimportant, but sometimes the computation is trapped into unacceptably slow progress. Even though Euler's rule using the automatic sequence of stepsizes seems not to be a likely method for solving stiff systems, this method can provide a usable solution with an acceptable computing time for some stiff problems.

Here, this method is introduced in order to have a method for solving stiff problems that is easy to program and gives relatively lower computing time. The test problems that will be used are:

$$\begin{aligned}
 1. \quad & y_1' = -y_1 - 0.5y_2 - 0.5y_3 & y_1(0) &= -1 \\
 & y_2' = -0.5y_1 - 1000.75y_2 + 999.25y_3 & y_2(0) &= 1 \\
 & y_3' = -0.5y_1 + 999.25y_2 - 1000.75y_3 & y_3(0) &= 3;
 \end{aligned}$$

$$\lambda_1 = -2000, \lambda_2 = -2, \lambda_3 = -1/2$$

with the exact solution

$$y_1(t) = e^{\lambda_2 t} - 2e^{\lambda_3 t}$$

$$y_2(t) = -e^{\lambda_1 t} + e^{\lambda_2 t} + e^{\lambda_3 t}$$

$$y_3(t) = e^{\lambda_1 t} + e^{\lambda_2 t} + e^{\lambda_3 t}$$

$$2. \quad y_1' = 0.01 - [1+(y_1+1000)(y_1+1)](0.01+y_1+y_2) \quad y_1(0) = 0$$

$$y_2' = 0.01 - (1+y_2^2)(0.01+y_1+y_2) \quad y_2(0) = 0$$

The derivation of the method for solving $y' = f(t,y)$, $y(t_0) = y_0$ is done by introducing a parameter θ , such that the method called the θ -method, and is written as $y_{n+1} = y_n + h[(1-\theta)f_{n+1} + \theta f_n]$. The parameter θ then should be determined such that the method is exact when applied to the test problem $y' = \mu y$. By modifying Euler's rule the method becomes $y_{n+1} = y_n + hvf_n$, where $v = 1+O(h)$ to guarantee the consistency of the method, and our task then is to find v such that the modified Euler method holds exactly when applied to the test problem $y' = \mu y$. The exact solution of the test problem is $y = y_0 e^{\mu t}$. After substituting $y' = \mu y$ into the modified Euler's rule, we have $v = [(y_{n+1}/y_n)-1]/h\mu = [e^{h\mu}-1]/h\mu$, where $y_{n+1} = y_0 e^{(n+1)h\mu}$ and $y_n = e^{nh\mu}$. Since the Taylor's series of $e^{h\mu} = 1 + h\mu + O(h)$, we have $v = 1 + O(h)$.

Consider the initial value problem $y' = f(t,y)$, $y(t_0) = y_0$, $y, f \in R^m$ and the test problem $y' = Ay$, $y(t_0) = y_0$, where A is a constant $m \times m$ matrix with real eigenvalues λ_t , $t = 1, 2, \dots, m$ satisfying $\lambda_1 \leq \lambda_2 \leq \lambda_1 \leq \dots \leq \lambda_{m-1} \leq \lambda_m \leq 0$. By adopting the idea given above, for the ODE system where $A_n = \partial f / \partial y|_n$, we have

$$\left. \begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + h\nu_n \mathbf{f}_n \\ \nu_n &= [e^{h\mu_n} - 1] / h\mu_n \\ \mu_n &= (\mathbf{f}_n^T \mathbf{A}_n \mathbf{f}_n) / (\mathbf{f}_n^T \mathbf{f}_n) \end{aligned} \right\}. \quad (4.1)$$

On the irregular-spaced point set $\{ t_n \mid t_{n+1} = t_n + h_n, h_n = h\nu_n \}$, the equation (4.1)

becomes

$$\left. \begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + h_n \mathbf{f}_n \\ h_n &= [e^{\theta\mu_n} - 1] / \mu_n \\ \mu_n &= (\mathbf{f}_n^T \mathbf{A}_n \mathbf{f}_n) / (\mathbf{f}_n^T \mathbf{f}_n) \end{aligned} \right\}. \quad (4.2)$$

Since Equation (4.2) may have a large step, which would not be good for the accuracy,

Equation (4.2) is modified as

$$\left. \begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + h_n \mathbf{f}_n \\ h_n &= \min(h_0, h_n^*) \\ h_n^* &= [e^{\theta\mu_n} - 1] / \mu_n \\ \mu_n &= (\mathbf{f}_n^T \mathbf{A}_n \mathbf{f}_n) / (\mathbf{f}_n^T \mathbf{f}_n) \end{aligned} \right\}. \quad (4.3)$$

The value θ should be chosen positive so that h_n^* is always positive. Since nothing is lost if θ is chosen very large, the modified Euler method with the automatically determined sequence of stepsizes can be written as

$$\left. \begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + h_n \mathbf{f}_n \\ h_n &= \min(h_0, h_n^*) \\ h_n^* &= (\mathbf{f}_n^T \mathbf{f}_n) / |\mathbf{f}_n^T \mathbf{A}_n \mathbf{f}_n| \end{aligned} \right\} \quad (4.4)$$

4.5 An Explicit Exponential Method for Solving ODEs

Even though some authors define stiffness equations as problems that cannot be solved by explicit methods [56, 62], there are others that have still tried to solve stiffness problems using explicit methods. Among them are Treanor [79], Lawson [79], Osborn [79], Lambert [77], and Ashour and Hanna [3]. The effort for solving stiff problems using explicit methods is still worthwhile because implicit methods need space and the cost of computing time per step is relatively higher than that of explicit methods. Treanor [79] modified the fourth-order Runge-Kutta method for solving stiff problems, Lawson [79] proposed a transformation of the stiff system into a nonstiff system, and then applied a Runge-Kutta method, Osborn [79] used Pade fractions to approximate $e^{h\lambda}$, Lambert [77] modified Euler method (Chapter 4.4) and Ashour and Hanna [3] used an exponential method by applying Watson's lemma of asymptotic expansion of Laplace integrals. The following explicit exponential method was a method that was proposed by Ashour and Hanna to solve stiff ODEs.

The explicit exponential method is developed based on the idea of detecting any exponential decaying variables at each step of the computation and to approximate them such that they are not exponential decaying any longer. To achieve this idea, Ashour and

Hanna [3] applied Watson's lemma. We know that integration of $y' = f(t,y)$ from t to $t+h$ is written as

$$y(t+h) = y(t) + \int_t^{t+h} f(s,y(s))ds. \quad (4.5)$$

Computations will decay exponentially if the local Jacobian $f_y(t_0,y_0)$ is negative and $|f_y(t_0,y_0)| \gg 1$. If this situation occurs, the solutions are approximated with functions such that computations tend not to decay exponentially any longer. The following expansion in Laplace integrals will be used to implement the idea of choosing solution methods aforesaid,

$$I = \int_0^A e^{-ts} F(s) ds, \quad t > 0 \quad (4.6)$$

where $F(s)$ is of exponential order. It means that function F is of exponential order β if there exist positive real numbers T , c , and β such that $|F(s)| < ce^{\beta s}$, $s > T$. Based on Watson's lemma, the integration (4.6) can be approximated by expanding $F(s)$ using a Maclaurin series and integrating it. In the case where the upper limit of A is too small relative to t , we use an expansion that is asymptotic either for $t \rightarrow \infty$ with A fixed, or for $A \rightarrow 0$ with t fixed. Using a mathematical manipulation involving the Maclaurin series, we have

$$\begin{aligned} F(s) &= e^{-ms} F(s) e^{ms} \\ &= e^{-ms} [F(0)e^0 + s\{F'(0)e^0 + mF(0)e^0\} + O(s^2)] \\ &= e^{-ms} [F(0) + s\{F'(0) + mF(0)\} + O(s^2)]. \end{aligned}$$

Let us take $\{F'(0) + mF(0)\} = 0$ such that the $O(s)$ term in a Maclaurin expansion is zero.

By substituting the latter results, from (4.6) we have

$$I = \int_0^A e^{-(t+m)s} F(0) ds = \frac{-F(0)}{t+m} [e^{-(t+m)A} - e^0] = \frac{F(0)}{t+m} [1 - e^{-(t+m)A}]. \quad (4.7)$$

By taking $s = t + u$, Equation (4.5) can be transformed into

$$y(t+h) = y(t) + \int_{u=0}^{u=h} e^{-mu} y'(t+h) e^{mu} du. \quad (4.8)$$

This transformation can only be used when $m > 0$; otherwise we solve the problem using an explicit method that converges to the result of transformation (4.8). Our next task is to find m by adopting the same idea we used in finding Equation (4.7). If the function $y'(t+h)e^{mu}$ is expanded using a Maclaurin series, we have

$$y'(t+h)e^{mu} = y'(t) + u[y''(t) + my'(t)] + O(h^2).$$

Now, we choose m such that $y''(t) + my'(t) = 0$ and $O(s)$ becomes zero. From these conditions, we have to choose $m = -y''(t)/y'(t)$ or $my'(t) = -y''(t)$. By considering the last conditions and applying them to (4.8), we have

$$y(t+h) = y(t) + [y'(t)/m][1 - e^{-mh}] + [O(1/m^3) \text{ or } O(h^3)]. \quad (4.9)$$

Now, we investigate Equation (4.9) to find an appropriate explicit method used for solving problems for $m \leq 0$. Let us take h small enough. We know perfectly well that e^{-mh} can be approximated using a Taylor series such as $[1 - mh + (mh)^2/2 + O(h^3)]$. By substituting this result into (4.9), we have

$$\begin{aligned} y(t+h) &= y(t) - y'(t)mh^2/2 + O(h^3) \\ &= y(t) + y''(t)h^2/2 + O(h^3). \end{aligned}$$

This last result is a Taylor series which is equivalent to a second-order explicit Runge-Kutta (RK2) method.

The numerical computation then is done as follows: If $m > 0$ we apply Equation [4.9], otherwise we apply RK2. Suppose $dold = f(t, y(t))$ and $dnew = f(t+h, y(t+h))$. An algorithm for an explicit exponential method can be given as follows:

```

For i = 1 to N                                     /* N is number of variables*/

    Compute  $dold_i = f_i(t, y(t))$ 

    Compute  $dnew_i = f_i(t+h, y(t+h))$ 

    If  $dold_i \neq 0$  then

        Calculate  $m_i = (dold_i - dnew_i)/(h*dold_i)$ 

        If  $m_i > 0$  then

            /* Calculate  $y(t+h)$  using an explicit exponential method */

             $y_i(t+h) = y_i(t) + dold_i[1 - \exp(-m_i*h)]/m_i$ 

        endif

    endif

    if  $dold_i = 0$  or  $m_i \leq 0$  then

        /* Calculate  $y(t+h)$  using RK2 */

         $y_i(t+h) = y_i(t) + 0.5*(dold_i + dnew_i)*h$ 

    endif

end do

```

In order to adjust the stepsize so the computation results will compromise with the required tolerance, we use a formula

$$h_{\text{new}} = \left(\frac{\text{relative error tolerance}}{\|\text{current relative local error}\|} \right)^{1/2} * h_{\text{current}}, \quad (4.10)$$

where $\|\text{current relative local error}\|$ can be written as

$$\max_{i=1}^N \left\{ \frac{|\text{estimated local error for } i\text{th variable}|}{|y_i(t+h)| + \varepsilon} \right\},$$

and ε is a positive parameter. If $|y_i(t+h)| > \varepsilon$ then the relative error TOL is controlled, otherwise the absolute error εTOL is controlled. The integration stepsize then is chosen as $h = \min\{h_{\text{new}}, 2h_{\text{current}}\}$. Usually the local error can be taken as the difference between two methods with different orders [see Appendix D]. The local error for RK2 in this method can be taken as the difference between the solutions obtained by RK2 and those of the Euler method. For an exponential method, the local error is estimated as the difference between the solutions obtained by the exponential method (4.9) and those of Equation (4.11) given below

$$y_i(t+h) = y_i(t) * \exp[y_i'(t)h/y_i(t)] + O(h^2). \quad (4.11)$$

4.6 ODE Solvers

Several ODE solvers that are available in the public domain (see Appendix A) will be investigated using test cases such as: Kidney problems proposed by Scott and Watts [17, 21, 79], an autocatalytic reaction proposed by Robertson [21, 56], problem D4 of Enright [39], and a problem proposed by Gupta and Wallace [53]. Among the ODE solvers used are MEBDF, LSODE, EPSODE, and VODE. All of the software including the explicit methods are coded in the FORTRAN language.

Gear was the first person who developed a widely-used code for solving stiff ODE problems. His early code named DIFSUB [1, 17, 44, 57, 99, 105] was developed using BDF methods for stiff problems and Adams methods for nonstiff problems. Among other variants of DIFSUB are STIFF [17] and GEAR (modification of STIFF and created by Gear and Hindmarsh [17]). GEAR itself has many variants including GEARB, GEARS, GEARBI, GEARBIL, BEARIB, GEARV, GEARST [17].

LSODE (Livermore Solver for ODEs) is a package created by Hindmarsh [<http://www.netlib.org/odepack/index.html>] that is based on combinations of GEAR and GEARB. It can be used to solve stiff or nonstiff problems. The Jacobian for the case of stiff problems is treated as either full or banded, and as either user-supplied or internally approximated by difference equations [1].

EPISODE (Experimental Package for Integration of Systems of Ordinary Differential Equations) was created based on DIFSUB [16]. It uses implicit BDF methods of orders one through five for stiff ODEs and Adams-Moulton methods of orders one through twelve for nonstiff ODEs. Both BDF and Adams-Moulton methods are in Nordsieck forms. Among variants of EPISODE are EPSODE, EPISODEB, and EPISODEIB [17]. EPSODE [<http://www.netlib.org/ode/index.html>] is an early version of EPISODE modified by Byrne and Hindmarsh.

MEBDF is a package program for stiff ODEs created by Cash and Cosidine [20, 21, <http://netlib.att.com/netlib/ode/index.html>, <http://www.netlib.org/ode/index.html>]. It is developed using modified extended BDF methods. The advantage of this approach is

that the methods is A-stable up to order 4 and $A(\alpha)$ -stable up to order 9, while pure BDF methods themselves are A-stable only up to order 2 [28].

VODE is a package program for ODEs created by Brown, Byrne, and Hindmarsh [11, <http://netlib.att.com/netlib/ode/index.html>, <http://www.netlib.org/ode/index.html>]. This program is a combination of EPISODE and EPISODEB. It uses variable-coefficient Adams-Moulton and BDF methods in Nordsieck form. It treats Jacobian matrices as full or banded.

STINT is a package program created by J.M. Tandler, T.A. Bickart, and Z. Picel [17, <http://www.netlib.org/toms/index.html>]. It was the first ODE solver developed based on cyclic composite linear multistep methods. This solver can solve ODE problems that belong to stiffly stable.

CHAPTER V

ANALYSIS AND DISCUSSION

The criteria of evaluation will be based on the efficiency of the codes using parameters such as computing time (CT), number of integration steps (NS), number of evaluations of $f(t,y)$ (NF), number of evaluations of the Jacobian (NJ), actual stepsize (H), and the actual order of the methods (P). The graphical interpretations for all solutions using methods and input test cases mentioned in section 4.1, will be compared with the graph of solutions of the same problem solved using the NDSolve function of Mathematica. $\text{NDSolve}\{\text{equation}_1, \text{equation}_2, \dots, \text{equation}_n\}, y, \{t, t_{\min}, t_{\max}\}$ is the function used in Mathematica to solve ODE problems numerically. The NDSolve function will automatically apply the Adams predictor-corrector method if it detects that the problem is non-stiff and will apply the backward differentiation formulae (Gear's method) if it detects that the problem is stiff. Since NDSolve can detect the stiffness of an ODE problem automatically, users do not need to know what stiffness is or even to be aware of its existence. The selection method for solving stiff and non-stiff systems of ODE for NDSolve is based on the method proposed by Petzold [95].

The modified Euler method will be implemented without the facility to calculate the Jacobian matrix numerically; users should be able to supply the Jacobian

matrix analytically. Since the Jacobian matrix of an ODE problem is not always easy to find, it is recommended that users use Mathematica or Maple or Matlab to get the Jacobian aforementioned. In this thesis, the Jacobian matrix of system of ODE is calculated using Mathematica. One example of Mathematica instructions for finding the Jacobian matrix of problem 2 of section 4.4 is given as follows:

```
f[y1_,y2_]=0.01-(1+(y1+1000)(y1+1))(0.01+y1+y2);
g[y1_,y2_]=0.01-(1+y2^2)(0.01+y1+y2);
jacob={{D[f[y1,y2],y1],D[f[y1,y2],y2]},
        {D[g[y1,y2],y1],D[g[y1,y2],y2]}};
MatrixForm[jacob]
```

Output is given as follows:

```
-1-(1+y1)(1000+y1)-(1001+2y1)(0.01+y1+y2)    -1-(1+y1)(1000+y1)
-1-y22                                          -1-y22-2y2(0.01+y1+y2)
```

5.1 Implementation of Modified Euler Method

The program for this method is given in Appendix F under the name of “Program 5”. For the case of problem 1 mentioned in Chapter 4.4, the solutions are compared with the exact solution, while for problem 2 of Chapter 4.4, the results will be compared with the output resulted from Mathematica.

To calculate problem 1 of Chapter 4.4 from 0 to 10, this program took 520 steps, while Lambert [77] claims that his program with the same modified Euler method only

took 501 steps and Euler's rule with the maximum allowable steplength 0.001 (the maximum steplength that is allowable for stability of the explicit Euler method for this problem) took 10,000 steps. The comparison between the solution of problem 1 that resulted from this modified Euler method and the exact solution is given in Table G.5 of Appendix G.

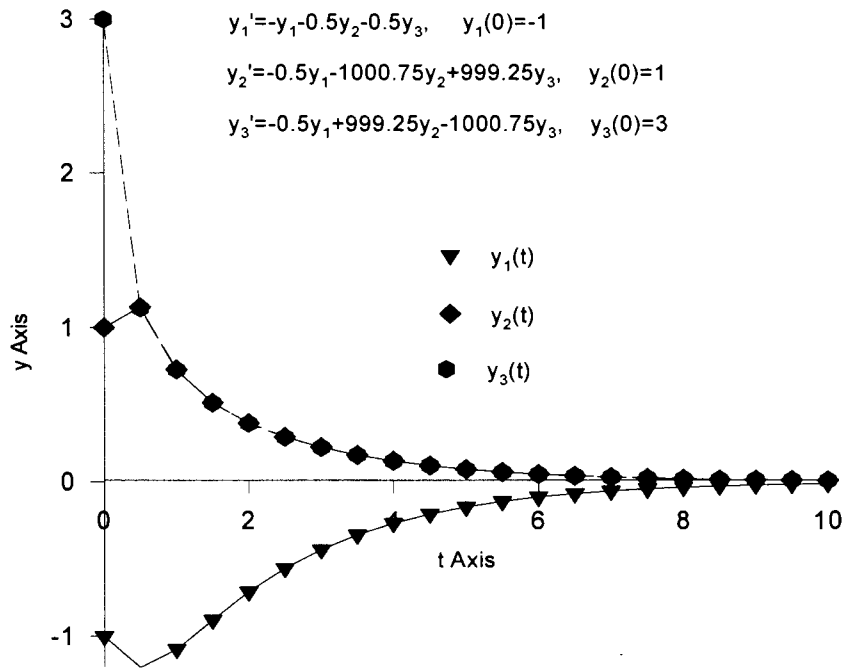


Figure 5.1 Problem 1 of Section 4.4
Using Modified Euler Method

This "Program 5" needs about 548 steps to integrate problem 2 of section 4.4 from 0 to 10, while Lambert [77] claims that his program with the same modified Euler

method only needs 379 steps to solve the same cases, and a fourth-order Runge-Kutta method took 3296 steps for a tolerance of 0.01 (the maximum steplength that is allowable for stability for the fourth-order Runge-Kutta method for this problem). So far, there is no known analytic solution of this problem, so that solutions of problem 2 of section 4.4, will be compared with the solutions from Mathematica. The comparison table resulted from this program and Mathematica software is given in Table G.6 of Appendix G. The graphical solutions this problem 2 produced by the modified Euler method is shown in Figure 5.2.

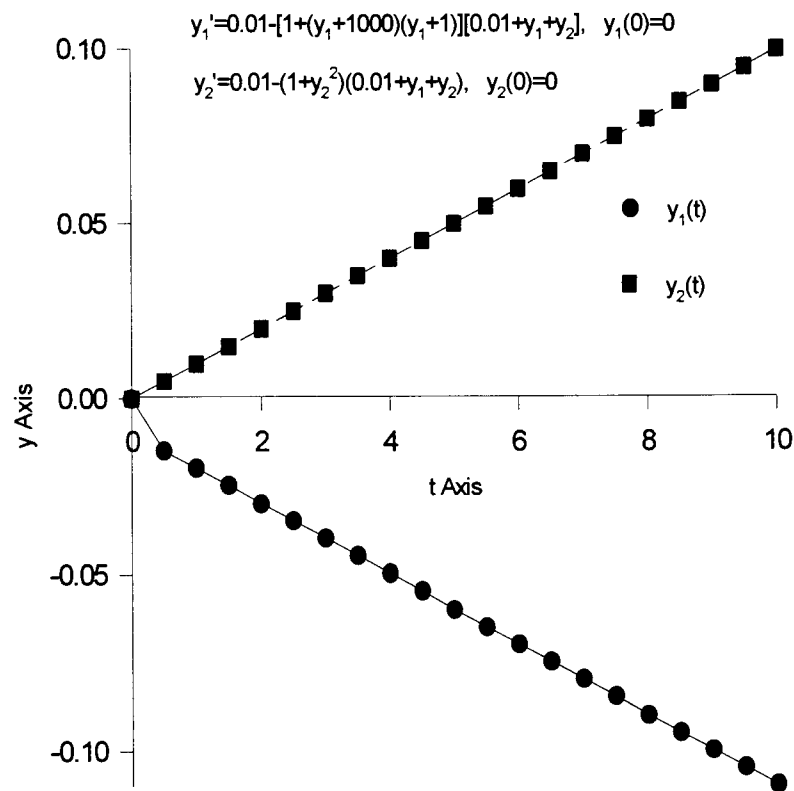


Figure 5.2 Problem 2 of Section 4.4
Using Modified Euler Method

5.2 Implementation of an Exponential Method

This method does not need the Jacobian matrix. A method for selecting an initial stepsize proposed by Gladwell et al. [48] has been implemented in this program. The FORTRAN coding of this method is given in Appendix F under the name of "Program 6". The two problems used in section 5.1 are also used as examples solved with this method.

To solve the problem 1 of section 4.4 from 0 to 10, this program took 9659 steps for a local tolerance of 10^{-4} . The comparison table between solutions of this problem 1 solved by the exponential method and the exact solution is given in Table G.7 of Appendix G. The solutions of this problem solved with the exponential method are shown in Figure 5.3.

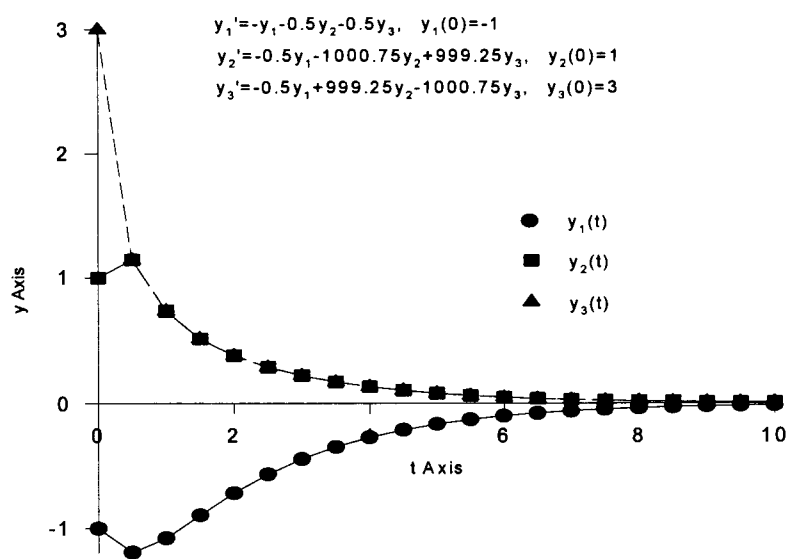


Figure 5.3 Problem 1 of Section 4.4
Using Exponential Method

This "Program 6" takes 4369 steps to integrate problem 2 of section 4.4 from 0 to 10 with a local tolerance of 10^{-6} . The comparison between solutions of this problem 2 solved with the exponential method and solutions resulted from Mathematica is given in Table 5.4. The solutions of this problem 2 solved with the exponential method is shown in Figure 5.4.

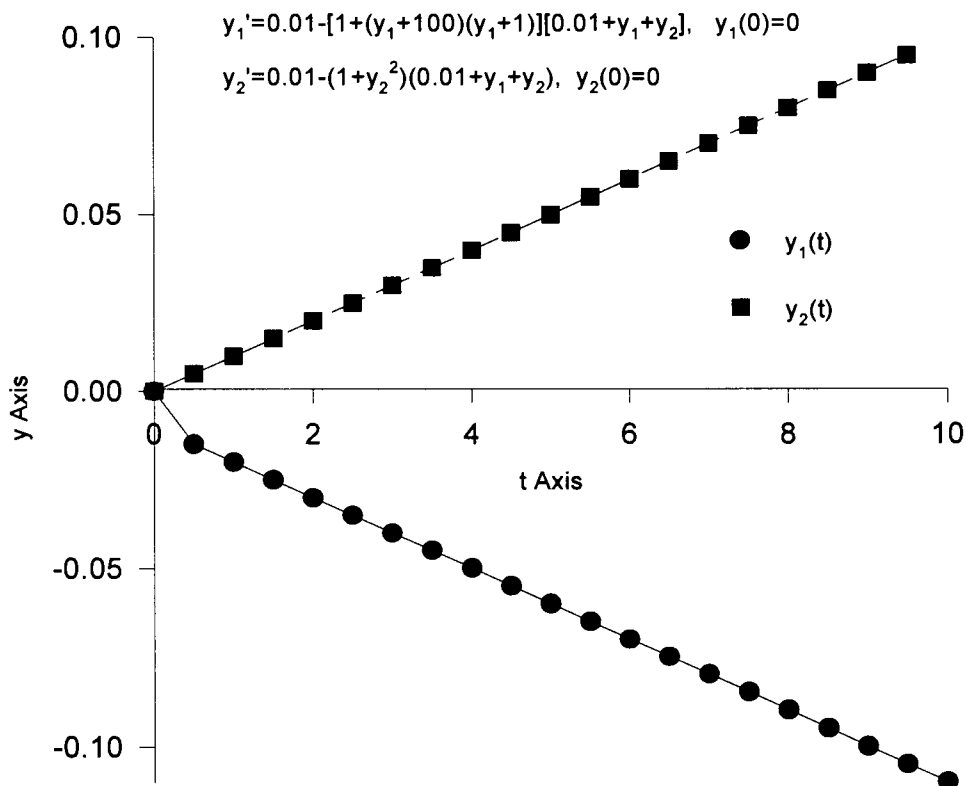


Figure 5.4 Problem 2 of Section 4.4
Using Exponential Method

5.3 Analysis of Kidney Problems

Due to Scott and Watts [20] and Byrne and Hindmarsh [16], kidney problems with several initial conditions given in Problem 1 of section 4.1 are considered as non-stiff and stiff problems. The graphical solutions of these problems using Mathematica with coding instructions given in Appendix F under the name "Mathematica 1", are shown in Figure 5.5 and Figure 5.6.

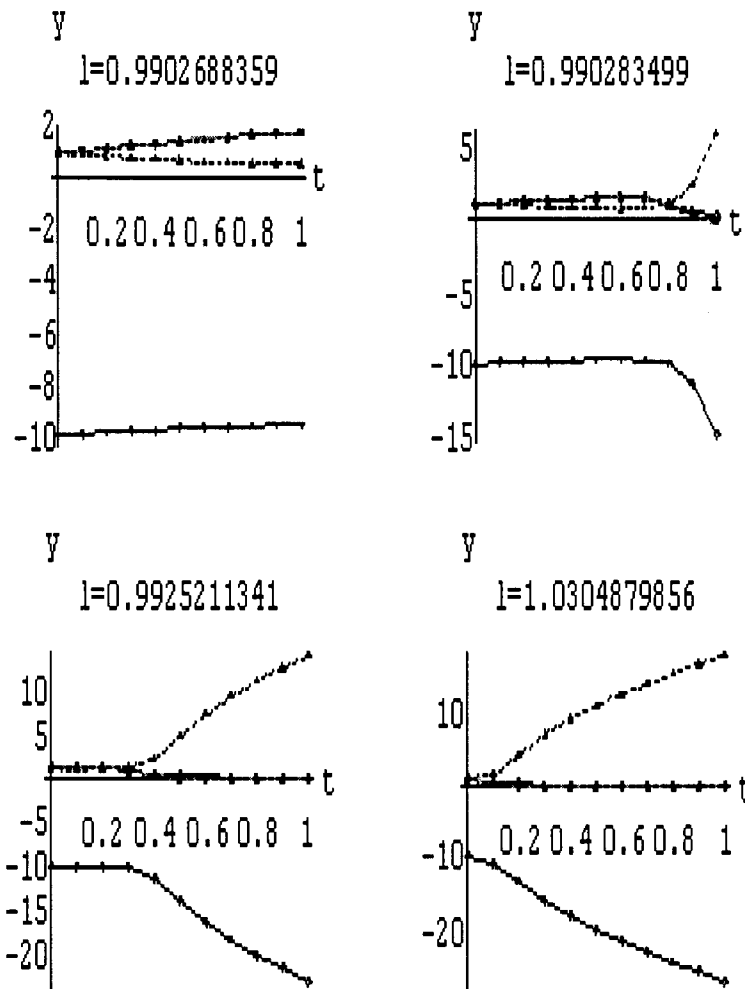


Figure 5.5 Kidney Problems with $\lambda = 0.9902688359, 0.990283499, 0.9925211341, \text{ and } 1.0304879856$

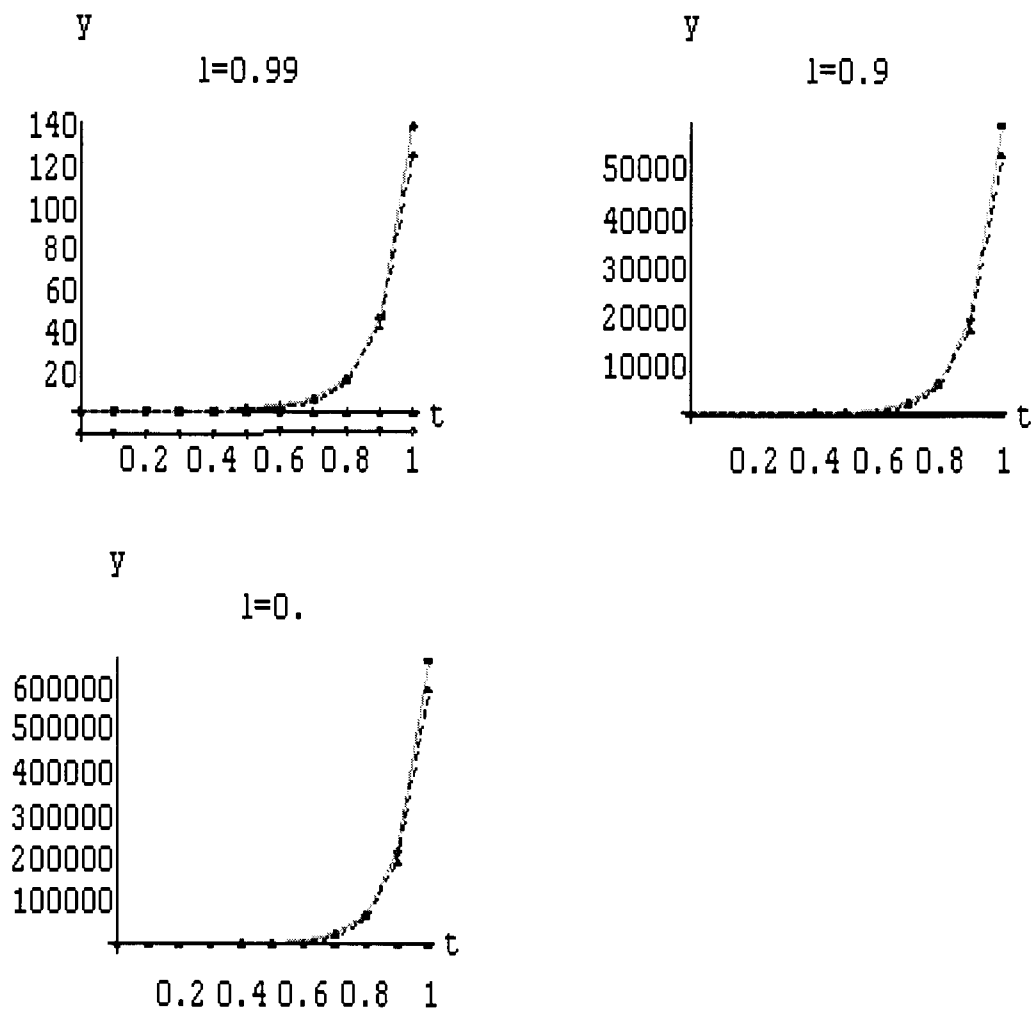


Figure 5.6 Kidney Problems with $\lambda = 0.99, 0.9,$ and 0

Main programs for calling MEBDF, VODE, LSODE, and EPSODE are given in Appendix F with names "Program 1", "Program 2", "Program 3", and "Program 4" respectively. All programs were run on a Sequent machine with a tolerance of 10^{-6} . The performances of these four routines for the case of kidney problems are given in Table G.1 of Appendix G.

Byrne and Hindmarsh [17] solved this problem using LSODE for the choices of $\lambda = 0.99026833, 0.99, 0.9$. They stated that the last two choices are stiff problems, and the first choice is non-stiff problem. By investigating Table G.1, we can conclude that kidney problems belong to stiff problems for the choices of $\lambda = 0.99, 0.9, \text{ and } 0$.

It is difficult to analyze the performances of these four routines based on pure CPU time of computations, because the DTIME routine supplied by Sequent can produce different execution time for different executions of the same program with the same problem. This occurs because Sequent uses symmetric multiprocessing (SMP) computing concept; Multiprocessing design in which any CPU can be assigned any application task. In all computations using MEBDF, LSODE, EPSODE, and VODE, we tried to run each problem as many as 20 times and took the average of execution times. It is more convenient to analyze the performances based on NS, NF, and NJ. For the case of $\lambda=0.9902688359$, to compute this problem from 0 to 1, EPSODE needs 68 steps, 128 evaluations of $f(t,y)$, and 23 evaluations of the Jacobian matrix of $f(t,y)$, VODE needs 74 steps, 105 evaluations of $f(t,y)$, and 2 evaluations of the Jacobian matrix of $f(t,y)$, LSODE needs 7 steps, 15 evaluations of $f(t,y)$, and 7 evaluations of the Jacobian matrix of $f(t,y)$, and MEBDF needs 77 steps, 147 evaluations of $f(t,y)$, and 2 evaluations of the Jacobian matrix of $f(t,y)$. Since evaluations of matrices are relatively expensive, the investigation performances will be based on the number of evaluations of the Jacobian matrix. For the choices of $\lambda = 0.99, 0.9, \text{ and } 0.0$, LSODE seems to have the best performance among the four packages. For other choices of λ , conclusions cannot be taken merely based on NJ, since NS and NF are varying.

5.4 Analysis of Autocatalytic Problems

Autocatalytic reaction pathway problems are frequently called Robertson problems to honor Robertson who was the first man who proposed these problems for chemical kinetics problem [21, 56, 79, 105]. These Robertson problems will be solved numerically using Mathematica, LSODE, MEBDF, VODE, and EPSODE, from 0 to 4.0×10^{10} . The graphical solutions will be given as functions of \log_{10} of t . Figure 5.7 shows graphical solutions that resulted by solving these problems using the Mathematica instructions given in Appendix F under the name "Mathematica 2".

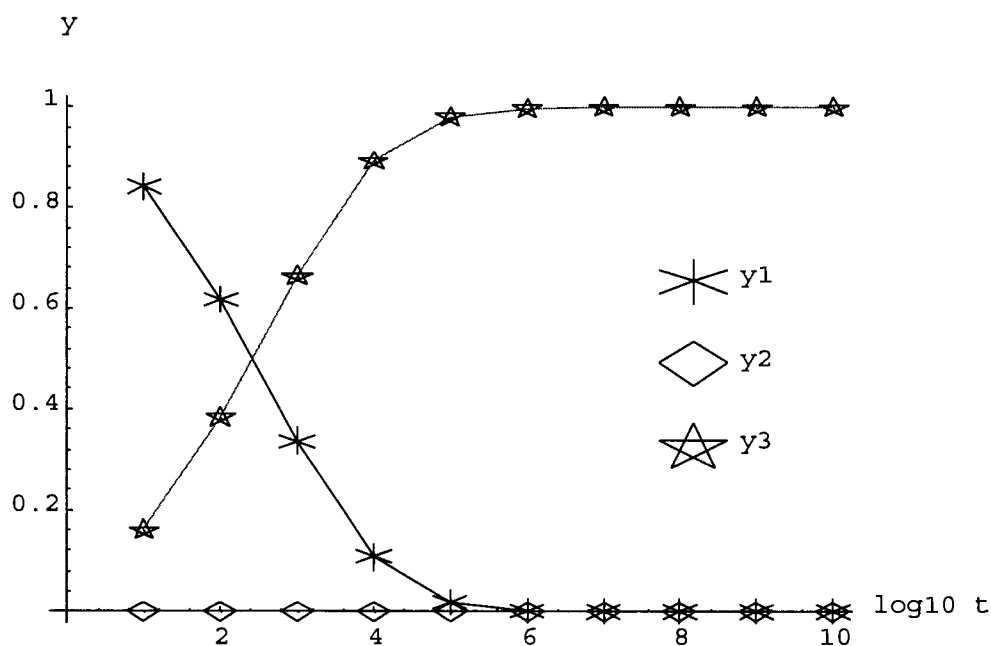


Figure 5.7 Output of Mathematica for Robertson Problem

The performances of the MEBDF, LSODE, VODE, and EPSODE routines when solving Robertson problems are given in Table G.2 of Appendix G. Figure 5.8 is a graphical solutions of Robertson Problems solved with EPSODE.

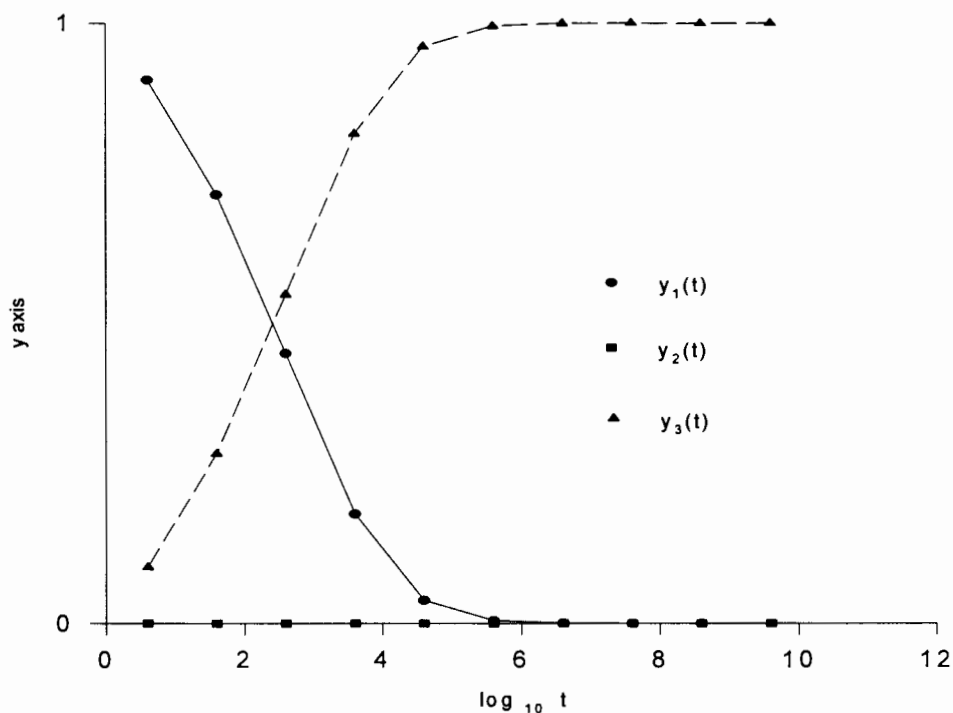


Figure 5.8 Robertson Problem Solved with EPSODE

The graphs that are given in Figure 5.7 and Figure 5.8 are plotted from $t=10$ to $4.0E10$.

Hindmarsh and Byrne [79] claimed that these problems are stiff because for the very early stages of computation, the stepsizes should be maintained small, even though gradually the stepsize can become larger and larger as t goes to infinity. Since NS, NF and NJ among the four routines are varying from one to another, we cannot comment on the best method for solving these problems among the four routines.

5.5 Analysis Problem D4 of Enright et al.

Enright et al. [39] proposed this problem as one example of a stiff problem. They gave the third equation as

$$y_3' = -0.013 y_1 - 1000 y_1 y_3 - 2500 y_2 y_3.$$

Later on, Shampine [103] commented that logically, this equation cannot match with the initial conditions y_1 , y_2 , and y_3 at $t=0$ that are given as 1, 1, and 0 respectively. In his opinion, for those conditions, $y_3'(0) = -0.013 < 0$, so that $y_3(t)$ should be negative for a very small value of t . Shampine then improved the problem by suggesting the third equation as

$$y_3' = 0.013 y_1 - 1000 y_1 y_3 - 2500 y_2 y_3.$$

Using the Mathematica instructions given in Appendix F under the name "Mathematica 3", we get graphical solutions as shown in Figure 5.9.

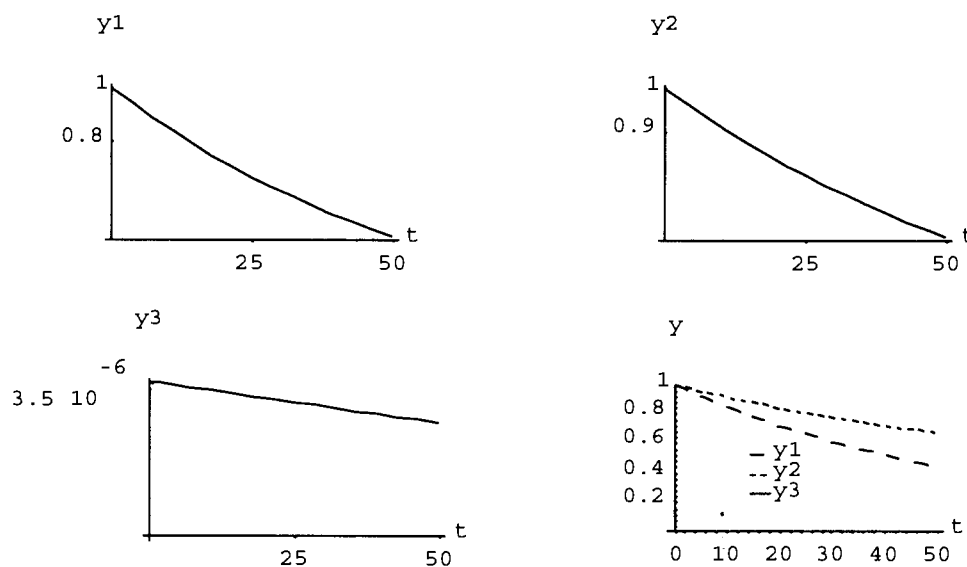


Figure 5.9 Problem D4 of Enright et al. Solved with Mathematica

Table G.3 of Appendix G is a table of performances of the MEBDF, VODE, LSODE, and EPSODE routines. A graphical solutions of this D4 problems solved with MEBDF is shown in Figure 5.10.

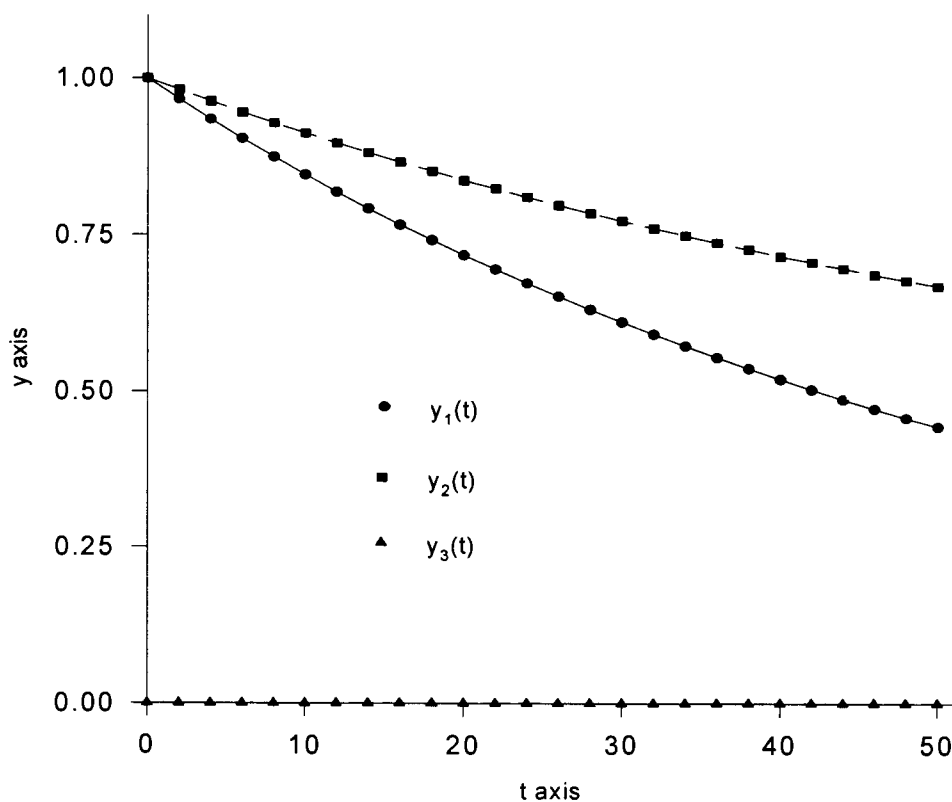


Figure 5.10 Problem D4 of Enright et al. Solved with MEBDF

By investigating Table G.3 and assuming that evaluating the Jacobian matrix is relatively more expensive than evaluating the function $f(t,y)$, it can be concluded that for these problems, MEBDF gives the best performance among the four packages.

5.6 Analysis of Gupta and Wallace's Problem

Mathematica needs 1523 steps to solve these problems from 0 to 10. In order to get Mathematica successfully to integrate the problems from 0 to 10, we have to supply to the NDSolve routine of Mathematica, a parameter MaxStep with a value not less than 1523. Otherwise, Mathematica will automatically stop the execution, and gives the message:

NDSolve :: mxst:

Maximum number of 500 steps reached at the point 3.27871.

Without supplying the parameter MaxStep, Mathematica will automatically give 500 steps as a default value for MaxStep. Figure 5.11 shows graphical solutions resulting from Mathematica for these problems.

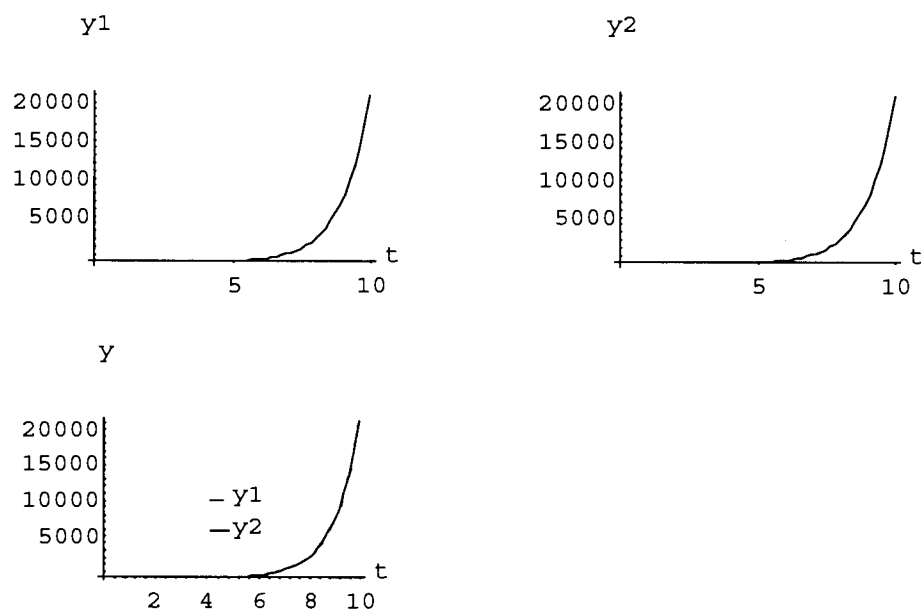


Figure 5.11 Gupta and Wallace's Problem Solved with Mathematica

Gupta and Wallace [53] stated that for the choices of $v = -80$ and $w = 8$, these problems are stiff. Whatever the choices of v and w , the exact solution will be found as $y_1 = e^t$ and $y_2 = e^t$. The problems is non-stiff if v and w are chosen to be 1 and 0 respectively.

These problems were also solved using the MEBDF, LSODE, VODE, and EPSODE routines. The performances of the four routines in solving these problems is given in Table G.4 of Appendix G. By assuming that evaluation of the Jacobian matrix will take more computing time than evaluation of the function $f(t,y)$, it can be said that of the four routines, MEBDF gives the best performance. Figure 5.12 shows graphical solutions from VODE on these problems.

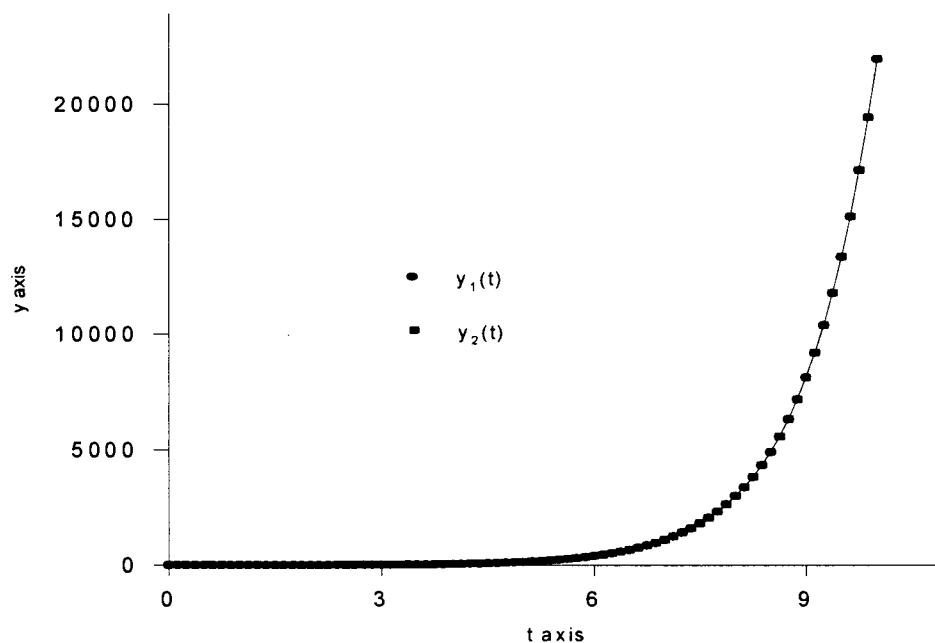


Figure 5.12 Gupta and Wallace's Problem Solved with VODE

CHAPTER VI

CONCLUSIONS AND SUGGESTIONS

It is not easy to define stiffness of ODEs precisely in mathematical terminology. The common approaches to understanding stiffness are to investigate problems based on various aspects of phenomena. If the paces of decay of the past and the slow mode of solutions are too disparate, then problems are potentially stiff. Kidney problems and the problems proposed by Gupta and Wallace are among examples that can be either stiff or nonstiff. In these situations, the classification depends on the choice of initial conditions.

Users who solve ODE problems using high-level languages should be aware of stiffness' existence, since otherwise they can get solutions that do not converge to the correct ones. Since all well-known explicit methods are not suitable for solving stiff ODE problems, the ODE solvers are recommended to have tools for detecting stiffness. These tools will be very useful especially when we need to minimize the CPU time of computation. It is undeniable that explicit methods are not recommended for handling stiff problems. The modified Euler and exponential methods are only recommended for solving stiff ODE problems where accuracies are not taken into consideration.

From solving several stiff problems using the MEBDF, LSODE, VODE, and EPSODE, we can see that the stepsizes vary over the whole range of computations. And so do the orders of the methods. This occurs because in some intervals the problems can

be very stiff, stiff, mildly stiff, or even nonstiff. This is one reason why a good ODE solver will support both implicit and explicit methods. Backward differentiation formulae and Adam predictor-corrector methods are among the most popular methods used to solve stiff and nonstiff problems respectively.

Recently, Shampine and Reichelt [Information taken from Netscape with address <http://www.mathworks.com/paper.html>] have developed tools for solving ODE problems using Matlab. They have introduced a new family of formulas for the solution of stiff problems. They call these formulas numerical differentiation formulas (NDF). They also claimed that these NDF are more efficient than the BDF, even though they agree that a couple of the higher order formulas are rather less stable. Since Matlab, Maple and Mathematica can access routines that are coded in high-level languages like C and FORTRAN, it is recommended that ordinary users of ODEs learn how to use one of those three software packages. From our experiences using Mathematica for solving ODE problems, we highly recommend that ordinary users try to learn how to use Mathematica. The NDSolve function of Mathematica will automatically choose an Adam predictor-corrector method if it detects the problem is nonstiff, and will choose BDF if the problem is stiff.

For the future, we think researchers who work in the field of numerical computation should consider developing numerical tools that can be attached to packages such as Matlab, Mathematica, or Maple.

SELECTED BIBLIOGRAPHY

1. Aiken, R.C., *Stiff Computation*, Oxford University Press, Inc., 1985.
2. Alexander, R.K., Stability of Runge-Kutta Methods for Stiff Ordinary Differential Equations, *SIAM J. Numer. Anal.*, Vol. 31, No. 4, pp. 1147-1168, August 1994.
3. Ashour, S.S., and Hanna, O.T., Explicit Exponential Method for the Integration of Stiff Ordinary Differential Equations, *J. Guidance, Control, and Dynamics*, Vol. 14., No. 6, December 1991, pp. 1234-1239.
4. Auzinger, W., On the Error Structure of the Implicit Euler Scheme Applied to Stiff Systems of Differential Equations, *Computing* 43, 115-131 (1989).
5. Auzinger, W., Frank, R., and Kirlinger, G., Modern Convergence Theory for Stiff Initial-Value Problems, *J. Comput. Appl. Math.* 45 (1993) 5-16.
6. Auzinger, W., Frank, R., and Macsek, F., Asymptotic Error Expansion for Stiff Equations: The Implicit Euler Scheme, *SIAM J. Numer. Anal.*, Vol. 27, No. 1, pp. 67-104, February 1990.
7. Babuska, I., Prager, M., and Vitasek, E., *Numerical Processes in Differential Equations*, John Wiley & Sons, Ltd., 1966.
8. Barton, D., On Taylor Series and Stiff Equations, *ACM Transaction on Mathematical Software*, Vol. 6., No. 3, September 1980, pages 280-294.
9. Birta, L.G., Yang, M., and Abou-Rabia, O., An Adaptive Approach to Stepsize Control in ODE Solvers, *Mathematics and Computers in Simulation* 35 (1993) 63-78.

10. Braun, M., *Differential Equations and Their Applications*, Springer-Verlag New York Inc., 1983.
11. Brown, P.N., Byrne, G.D., and Hindmarsh, A.C., VODE: A Variable-Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, Vol. 10, No. 5, pp. 1033-1051, September 1989.
12. Bui, T.D., and Poon, S.W., On the Computational Aspects of Rosenbrock Procedures with Built-in Error Estimates for Stiff Systems, *BIT* 21 (1981) 168-174.
13. Burrage, K., The Dichotomy of Stiffness : Pragmatism Versus Theory, *J. Comput. Appl. Math.* 31 (May 1989) 92-111.
14. Butcher, J.C., Optimal Order and Stepsize Sequences, *IMA J. Numer. Anal.* (1986) 6, 433-438.
15. Butcher, J.C., *The Numerical Analysis of Ordinary Differential Equations*, John Wiley & Sons Ltd., 1987.
16. Byrne, G.D., and Hindmarsh, A.C., A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations, *ACM Transactions on Mathematical Software*, Vol. 1, No. 1, March 1975, pages 71-96.
17. Byrne, G.D., and Hindmarsh, A.C., Stiff ODE Solvers: A Review of Current and Coming Attractions, *J. Comput. Physics* 70, 1-62(1987).
18. Calvo, M., Lisbona, F., and Montijano, J., On the Stability of Variable-Stepsize Nordsieck BDF Methods, *SIAM J. Numer. Anal.*, Vol. 24, No. 4, August 1987.
19. Carroll, J., A Composite Integration Scheme for the Numerical Solution of Systems of Ordinary Differential Equations, *J. Comput. Appl. Math.* 25 (1989) 1-13.

20. Cash, J.R., and Considine, S., An MEBDF Code for Stiff Initial Value Problems, *ACM Transactions on Mathematical Software*, Vol. 18, No. 2, June 1992, Pages 142-155.
21. Cash, J.R., and Considine, S., MEBDF: A FORTRAN Subroutine for Solving First-Order Systems of Stiff Initial Value Problems for Ordinary Differential Equations, *ACM Transactions on Mathematical Software*, Vol. 18, No. 2, June 1992, pages 156-158.
22. Chang, Y.F., Solving STIFF Systems by Taylor Series, *J. Comput. Appl. Math.* 31 (May 1989) 251-269.
23. Chen, T.C., Automatic Computation of Exponential, Logarithms, Ratios and Square Roots, *IBM J. Res. Develop.*, July 1972, pp. 380-388.
24. Corlis, G., and Chang, Y.F., Solving Ordinary Differential Equations Using Taylor Series, *ACM Transactions on Mathematical Software*, Vol. 8, No. 2, June 1982, pages 114-144.
25. Crane, R.L., and Klopfenstein, R.W., A Predictor-Corrector Algorithm with an Increased Range of Absolute Stability, *J. ACM*, Vol. 12, No. 2 (April 1965), pp. 227-241.
26. Dahlquist, G., Bjorck, A., *Numerical Methods*, Prentice-Hall, Inc., 1974.
27. Dahlquist, G., Convergence and Stability In the Numerical Integration of Ordinary Differential Equations, *Math. Scand.* 4 (1956), 33-53.
28. Dahlquist, G.G., A Special Stability Problem For Linear Multistep Methods, *BIT* 3 (1963), 27-43.

29. Dahlquist, G.G., Liniger, W., and Nevanlinna, O., Stability of Two-Step Methods for Variable Integration Steps, *SIAM J. Numer. Anal.*, Vol. 20, No. 5, October 1983, pp. 1071-1085.
30. Davis, M.E., *Numerical Methods & Modeling for Chemical Engineers*, John Wiley & Sons, 1984.
31. Derr, L., Outlaw, C., and Sarafyan, D., A New Method for Derivation of Continuous Runge-Kutta Formulas, *Computers Math. Applic.*, Vol. 26, No. 3, pp. 7-13, 1993.
32. Deuflhard, P., Order and Step-size Control in Extrapolation Methods, *Numer. Math.* 41, 399-422(1983).
33. Deuflhard, P., Recent Progress in Extrapolation Methods for Ordinary Differential Equations, *SIAM Review*, Vol. 27, No. 4, December 1985.
34. Dieci, L., and Estep D., Some Stability Aspects of Schemes for the Adaptive Integration of Stiff Initial Value Problems, *SIAM J. Sci. Stat. Comput.*, Vol. 12, No. 6, pp. 1284-1303, November 1991.
35. Distefano, G. P., Causes of Instabilities in Numerical Integration Techniques, *International J. Comp. Math.*, 1968, Vol. 2, pp. 123-142.
36. Enright, W.H., A New Error-control for Initial Value Solvers, *J. Comput. Appl. Math.* 31 (May 1989) 288-301.
37. Enright, W.H., and Pryce, J.D., Two FORTRAN Packages for Assessing Initial Value Methods, *ACM Transactions on Mathematical Software*, Vol. 13, No. 1, March 1987, pp. 1-27.

38. Enright, W.H., and Seward, W.L., Achieving Tolerance Proportionality in Software for Stiff Initial-Value Problems, *Computing* 42, 341-352 (1989).
39. Enright, W.H., Hull, T.E., and Lindberg, B., Comparing Numerical Methods For Stiff Systems of O.D.E:s, *BIT* 15(1975), 10-48.
40. Fatunla, S.O., An Implicit Two-Point Numerical Integration Formula for Linear and Nonlinear Stiff Systems of Ordinary Differential Equations, *Mathematics of Computation*, Vol. 32, No. 141, January 1978, pp. 1-11.
41. Fox, L., and Mayers, D.F., On the Numerical Solution of Implicit Ordinary Differential Equations, *IMA J. Numer. Anal.* (1981) 1, 377-401.
42. Gear, C.W., and Tu, K.W., The Effect of Variable Mesh on the Stability of Multistep Methods, *SIAM J. Numer. Anal.*, Vol. 11, No. 5, October 1974.
43. Gear, C.W., and Watanabe, D.S., Stability and Convergence of Variable Order Multistep Methods, *SIAM J. Numer. Anal.*, Vol. 11, No. 5, October 1974.
44. Gear, C.W., *Numerical Initial Value Problem In Ordinary Differential Equations*, Prentice-Hall, Inc., 1971.
45. Gear, C.W., Runge-Kutta Starters for Multistep Methods, *ACM Transactions on Mathematical Software*, Vol. 6, No. 3, September 1980, pages 263-279.
46. Gear, C.W., The Automatic Integration of Stiff Ordinary Differential Equations, *Information Processing* 68 - North-Holland Publishing Company - Amsterdam, 187-193 (1969).
47. Gerlach, J., Accelerated Convergence in Newton's Method, *SIAM Review*, Vol. 36, No. 2, pp. 272-276, June 1994.

48. Gladwell, I., Shampine, L.F., and Brankin, R.W., Automatic Selection of the Initial Step Size for an ODE Solver, *J. Comput. Appl. Math.* 18(1987) 175-192.
49. Golub, G.H., and Ortega, J.M., *Scientific Computing and Differential Equations*, Academic Press, Inc., 1981, 1992.
50. Gottwald, B.A., and Wanner, G., A Reliable Rosenbrock Integrator for Stiff Differential Equations, *Computing* 26, 355-360 (1961).
51. Gragg, W.B., and Stetter, H.J., Generalized Multistep Predictor-Corrector Methods, *J. ACM*, Vol. 11, No. 2 (April 1964), pp. 188-209.
52. Groeneweg, J., and Spijker, M.N., On the Error Due to the Stopping of the Newton Iteration in Implicit Linear Multistep Methods, *Proceeding of the 15th Dundee Conference*, June-July 1993.
53. Gupta, G.K., and Wallace, C.S., Some New Methods for Solving Ordinary Differential Equations, *Mathematics of Computation*, Vol. 29, No. 130, April 1975, pp. 489-500.
54. Gupta, G.K., Sacks-Davis, R., and Tischer, P.E., A Review of Recent Developments in Solving ODEs, *Computing Surveys*, Vol. 17, No. 1, March 1985, pp. 5-47.
55. Gustafsson, K., Ludh, M., and Soderlind, G., A PI Stepsize for the Numerical Solution of Ordinary Differential Equations, *BIT* 28 (1988), 270-287.
56. Hairer, E., and Wanner, G., *Solving Ordinary Differential Equations II*, Springer-Verlag Berlin Heidelberg, 1991.
57. Hairer, E., Norsett, S.P., and Wanner, G., *Solving Ordinary Differential Equations I*, Springer-Verlag Berlin Heidelberg 1987, 1993.

58. Hall, G., A New Step-size Strategy for Runge-Kutta Codes, *Numerical Analysis Report No. 245. University of Manchester. Manchester Centre for Computational Mathematics*, March 1994.
59. Hall, G., Stability Analysis of Predictor-Corrector Algorithm of Adams Type, *SIAM J. Numer. Anal.*, Vol. 11, No. 3, June 1974.
60. Havie, T., Romberg Integration As a Problem in Interpolation Theory, *BIT* 17 (1977), 418-429.
61. Henrici, P., *Discrete Variable Methods In Ordinary Differential Equations*, John Wiley & Sons, Inc., 1962.
62. Higham, D.J., and Trefethen, L.N., Stiffness of ODEs, *BIT* 33 (1993), 285-303.
63. Hindmarsh, A.C., and Petzold, L.R., Algorithms and Software for Ordinary Differential Equations and Differential Algebraic Equations, Part I: Euler Methods and Error Estimation, *Computers in Physics*, Vol. 9, No. 1, Jan/Feb 1995, pp. 34-41.
64. Hindmarsh, A.C., and Petzold, L.R., Algorithms and Software for Ordinary Differential Equations and Differential Algebraic Equations, Part II: Higher-Order Methods and Software Packages, *Computers in Physics*, Vol. 9, No. 1, Mar/Apr 1995, pp. 148-155.
65. Hull, T.E., Enright, W.H., Fellen, B.M., and Sedgwick, A.E., Comparing Numerical Methods For Ordinary Differential Equations, *SIAM J. Numer. Anal.*, Vol. 9., No. 4, December 1972, pp. 603-637.
66. Jain, M.K., *Numerical Solution of Differential Equations*, Wiley Eastern Limited, 1979.

67. Jeltsch, R., and Nevanlinna, O., Dahlquist's First Barrier for Multistage Multistep Formulas, *BIT* 24 (1984), 538-555.
68. Jeltsch, R., and Nevanlinna, O., Stability and Accuracy of Time Discretizations for Initial Value Problems, *Numer. Math.* 40, 245-296 (1982).
69. Jeltsch, R., and Nevanlinna, O., Stability of Explicit Time Discretizations for Solving Initial Value Problems, *Numer. Math.* 37, 61-91 (1981).
70. Kirchgraber, U., Multi-Step Methods are Essentially One-Step Methods, *Numer. Math.* 48, 85-90(1986).
71. Kockler, N., *Numerical Methods and Scientific Computing Using Software Libraries for Problem Solving*, Clarendon Press Oxford, 1994.
72. Kohfeld, J.J., and Thompson, G.T., Multistep Methods with Modified Predictors, *J. ACM*, Vol. 14, No. 1, January 1967, pp. 155-166.
73. Krogh, F.T., A Test for Instability in the Numerical Solution of Ordinary Differential Equations, *J. ACM* Vol. 14, No. 2, April 1967, pp. 351-354.
74. Krogh, F.T., A Variable Step Variable Order Multistep Method for the Numerical Solution of Ordinary Differential Equations, *Information Processing* 68 - North-Holland Publishing Company-Amsterdam (1969), pp. 194-199.
75. Krogh, F.T., and Stewart, K., Asymptotic ($h \rightarrow \infty$) Absolute Stability for BDFs Applied to Stiff Differential Equations, *ACM Transactions on Mathematical Software*, Vol. 10, No. 1, March 1984, pages 45-57.
76. Krogh, F.T., On Testing a Subroutine for the Numerical Integration of Ordinary Differential Equations, *J. ACM* Vol. 20. No. 4, October 1973, pp. 545-562.

77. Lambert, J.D., A Stable Sequence of Steplengths for Euler's Rule Applied to Stiff Systems of Differential Equations, *Comp. & Maths. with Appl.*, Vol. 12B, No. 5/6, pp. 1141-1151, 1986.
78. Lambert, J.D., *Numerical Methods For Ordinary Differential Systems*, John Wiley & Sons, Ltd., 1991.
79. Lapidus, L., and Schiesser, W.E., *Numerical Methods for Differential Systems*, Academic Press, Inc., 1976.
80. Lapidus, L., and Seinfeld, J.H., *Numerical Solution of Ordinary Differential Equations*, Academic Press, Inc., 1971.
81. Lindberg, B., A Simple Interpolation Algorithm for Improvement of the Numerical Solution of a Differential Equation, *SIAM J. Numer. Anal.*, Vol. 9, No. 4, 1972, pp. 662-668.
82. Lindberg, B., Characterization of Optimal Stepsize Sequences for Methods for Stiff Differential Equations, *SIAM J. Numer. Anal.*, Vol. 14, No. 5, October 1977.
83. Lindberg, B., On a Dangerous Property of Methods for Stiff Differential Equations, *BIT* 14 (1974), 430-436.
84. Lindberg, B., On Smoothing and Extrapolation for The Trapezoidal Rule, *BIT* 11 (1971), 29-52.
85. Liniger, F., and Odeh, F., A-Stable, Accurate Averaging of Multistep Methods for Stiff Differential Equations, *IBM J. Res. Develop.*, July 1972, pp. 335-348.

86. Liniger, W., and Willoughby, R.A., Efficient Integration Methods for Stiff Systems of Ordinary Differential Equations, *SIAM J. Numer. Anal.*, Vol. 7., No. 1, March 1970.
87. Lu, L., The Stability of the Block θ -Methods, *IMA J. Numer. Anal.* (1993) 13, 101-114.
88. Lyness, J.N., An Algorithm For Gauss-Romberg Integration, *BIT* 12(1972), 194-203.
89. Maeder, A.J., A General Purpose ODE Solver Implementation, *J. Comput. Appl. Math.* 31 (May 1989) 316-327.
90. Marchuck, G.I., *Numerical Methods and Applications*, CRC Press, Inc., 1994.
91. Mickens, R.E., *Difference Equations*, Van Nostrand Reinhold Company Inc., 1987.
92. Mickens, R.E., *Nonstandard Finite Difference Models of Differential Equations*, World Scientific, Co., Pte., 1994.
93. Neto, R., and Rao, R., A Stochastic Approach to Global Error Estimation in ODE Multistep Numerical Integration, *J. Comput. Appl. Math.* 30 (1990) 257-281.
94. Odeh, F., Some Stability Techniques for Multistep Methods, *IBM J. Res. Develop.*, Vol. 31, No. 2, March 1987, pp. 178-185.
95. Petzold, L., Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, *SIAM J. Sci. Sta. Comput.*, Vol. 4., No. 1, March 1983.
96. Rahme, H.S., Stability Analysis of a New Algorithm Used for Integrating a System of Ordinary Differential Equations, *J. ACM*, Vol. 17, No. 2, April 1970, pp. 284-293.

97. Riggs, J. B., *An Introduction To Numerical Methods for Chemical Engineers*, Texas Tech University Press, 1994.
98. Seifert, P., Computational Experiments with Algorithms for Stiff ODEs, *Computing* 38, 163-176 (1987).
99. Shampine, L.F., and Gordon, M.K., *Computer Solution of Ordinary Differential Equations*, W.H. Freeman and Company, 1975.
100. Shampine, L.F., Control of Step Size and Order in Extrapolation Codes, *J. Comput. Appl. Math.* 18 (1987) 3-16.
101. Shampine, L.F., Diagnosing Stiffness for Runge-Kutta Methods, *SIAM J. Sci. Stat. Comput.*, Vol. 12, No. 2, pp. 260-272, March 1991.
102. Shampine, L.F., Efficient Extrapolation Methods for ODEs, *IMA J. Numer. Anal.* (1983) 3, 383-395.
103. Shampine, L.F., Evaluation of a Test Set for Stiff ODE Solvers, *ACM Transactions on Mathematical Software*, Vol. 7, No. 4, December 1981, pages 409-420.
104. Shampine, L.F., Implementation of Rosenbrock Methods, *ACM Transactions on Mathematical Software*, Vol. 8, No. 2, June 1982, pages 93-113.
105. Shampine, L.F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, Inc., 1994.
106. Shampine, L.F., Type-Incentive ODE Codes Based on Implicit A-Stable Formulas, *Mathematics of Computations*, Vol. 36, No. 154, April 1981, pp. 499-510.
107. Shub, M., The Implicit Function Theorem Revisited, *IBM J. Res. Develop.*, Vol. 38, No. 3, May 1994, pp. 259-263.

- 108.Skeel, R.D., Thirteen Ways to Estimate Global Error, *Numer. Math.* 48, 1-20(1986).
- 109.Stetter, H.J., Global Error Estimation in Adams PC-Codes, *ACM Transactions on Mathematical Software*, Vol. 5, No. 4, December 1979, pages 415-430.
- 110.Stewart, K., Avoiding Stability-induced Inefficiencies in BDF Methods, *J. Comput. Appl. Math.* 29 (1990), 357-367.
- 111.Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, New York, Inc., 1980.
- 112.Stoffer D, and Nipp, K., Invariant Curves For Variable Step Size Integrators, *BIT* 31(1991), 169-180.
- 113.Stuart, A.M., and Humphries, A.R., Model Problems in Numerical Stability Theory for Initial Value Problems, *SIAM Review* Vol. 36, No. 2, pp. 226-257, June 1994.
- 114.Vichnevetsky, R., New Stability Theorems Concerning One-Step Numerical Methods for Ordinary Differential Equations, *Mathematics and Computers in Simulation* 25 (1983) 199-205.
- 115.Walston, D.E., and Waddell, E.R., Acceleration of Convergence of One-Step Methods for the Numerical Solution of Ordinary Differential Equations, *International Journal of Computer Mathematics*, 1968, Vol.2, pp. 23-33.
- 116.Walz, G., Error Bounds and Stopping Rules for Extrapolation Methods, *IMA J. Numer. Anal.* (1989) 9, 185-198.
- 117.Wille, D. R., Experiments in Step-size Control for Adams Linear Multistep Methods, *Numerical Analysis Report No. 253. University of Manchester. Manchester Centre for Computational Mathematics*, October 1994.

118. Wille, D. R., New Step Size Estimators for Linear Multistep Methods, Numerical Analysis Report No. 247, University of Manchester, Manchester Centre for Computational Mathematics, March 1994.
119. Zhang, W., The Starting Procedure in Variable-stepsize Variable-order PECE Codes, J. Comput. Appl. Math. 53 (1994) 73-86.
120. Zlatev, Z., Advances in The Theory of Variable Step Size Variable Formula Methods for Ordinary Differential Equations, J. Comput. Appl. Math. 31 (May 1989) 209-249.
121. Zlatev, Z., and Thomsen, P.G., Automatic Solution of Differential Equations Based on the Use of Linear Multistep Methods, ACM Transactions on Mathematical Software, Vol. 5., No. 4, December 1979, pages 401-414.

APPENDIX A

SOFTWARE AVAILABILITY

Information about how to access software from the IMSL, NAG, and NETLIB libraries, and a brief overview of some products for solving ODEs will be given in this Appendix. All software in both the IMSL and NAG libraries is commercial. So, these routines are not free of charge such as the software that is offered in the NETLIB library:

The information about the products and services available from NAG can be obtained by sending e-mail to infodesk@nag.com or accessing the NAG pages at <http://www.nag.co.uk:70/>. Among the products of NAG for solving ODEs are:

- D02AGF : Solving boundary value problem, shooting and matching technique, allowing interior matching point, general parameters to be determined.
- D02BAF : Solving IVP, Runge-Kutta-Merson method, over a range (simple driver).
- D02BBF : Solving IVP, Runge-Kutta-Merson method, over a range, intermediate output (simple driver).
- D02BDF : Solving IVP, Runge-Kutta-Merson method, over a range, global error estimate, stiffness check (simple driver).
- D02BGF : Solving IVP, Runge-Kutta-Merson method, until a component attains given value (simple driver).
- D02BHF : Solving IVP, Runge-Kutta-Merson method, until function of solution is zero (simple driver).
- D02CAF : Solving IVP, Adams method, over a range (simple driver)
- D02CBF : Solving IVP, Adams method, over a range, intermediate output (simple driver).
- D02CGF : Solving IVP, Adams method, until a component attains given value (simple driver).
- D02CHF : Solving IVP, Adams method, until function of solution is zero (simple driver).

- D02CJF : Solving IVP, Adams method, until function of solution is zero, intermediate output (simple driver).
- D02EAF : Solving stiff IVP, BDF method, over a range (simple driver).
- D02EBF : Solving stiff IVP, BDF method, over a range, intermediate output (simple driver).
- D02EGF : Solving stiff IVP, BDF method, until a component attains given value (simple driver).
- D02EHF : Solving stiff IVP, BDF method, until function of solution is zero (simple driver).
- D02EJF : Solving stiff IVP, BDF method, until function of solution is zero, intermediate output (simple driver).
- D02GAF : Solving boundary value problem, finite difference technique with deferred correction, simple nonlinear problem.
- D02GBF : Solving boundary value problem, finite difference technique with deferred correction, general linear problem.
- D02HAF : Solving boundary value problem, shooting and matching, boundary values to be determined.
- D02HBF : Solving boundary value problem, shooting and matching, general parameters to be determined.
- D02JAF : Solving boundary value problem, collocation and least-squares, single nth order linear equation.
- D02JBF : Solving boundary value problem, collocation and least-squares, system of 1st order linear equations.
- D02QFF : Solving Adams method with root-finding (forward communication, comprehensive).
- D02QGF : Solving IVP, Adams method with root-finding (reverse communication, comprehensive).
- D02YAF : Solving IVP, Runge-Kutta-Merson method, integration over one step.

Information about the products and services available from IMSL can be obtained by sending e-mail to mktg@imsl.com or accessing the IMSL pages at <http://www.vni.com/>. Among the products of IMSL for solving ODEs are:

- DASPG : Solving stiff and mixed algebraic-ODEs, the Petzold--Gear BDF method.
- DIVPAG : Solving stiff or mildly stiff IVP, Adams-Moulton or Gear method.
- DIVPRK : Solving nonstiff or mildly stiff IVP, Runge-Kutta-Verner fifth- and sixth-order method.

Public domain software for numerical methods can be accessed from the NETLIB library by sending e-mail to netlib@ornl.gov with the body message

send index.

After receiving the index file, the next instruction will be given in the index file. This library can also be accessed using anonymous ftp at netlib.att.com or through the NETLIB pages at <http://www.netlib.org/>. Directories /netlib/toms, /netlib/ode, and /netlib/odepack contain some programs that can be used to solve ODEs.

Among the programs that are given in /netlib/ode are:

- RKSUITE : Created by R.W. Brankin (NAG), I. Gladwell and L.F. Shampine (SMU), solving IVP including an error assessment facility and a sophisticated stiffness checker, file
- DDASRT : Created by Petzold, solving stiff differential-algebraic system with root stopping, backward differentiation formulae.
- DDASSL : Created by Petzold, solving stiff differential-algebraic system, backward differentiation formulae.
- DP12 : Created by Cash, solving stiff IVP, Cash's extended backward differentiation formulae.
- DRESOL : Created by Dieci, solving stiff/nonstiff matrix differential Riccati equations (DREs) with reference SIAM J. Numerical Analysis 29-3 and symmetric and unsymmetric DREs, Adams' and backward differentiation formulae based on LSODE by Hindmarsh.
- DVERK : Created by Jackson, Hull, and Enright, solving IVP with global error control, Verner's fifth and sixth order Runge-Kutta pair.
- EPSODE : Created by Byrne and Hindmarsh, solving stiff IVP, backward differentiation formulae (variable coefficient formulae).
- ODE : Created by Shampine and Gordon, solving IVP, Adam's methods.
- RKC : Created by Sommeijer, solving nearly-stiff IVP, second-order explicit Runge-Kutta formulae.
- RKF45 : Created by Watts and Shampine, solving IVP, Runge-Kutta Fehlberg fourth-fifth order.
- SDERoot : Created by Shampine, Gordon, and Allen, solving IVP with root stopping, Adam's methods.
- SODE : Created by Shampine and Gordon, solving IVP, Adam's methods.
- VODE : Created by Brown, Byrne and Hindmarsh, solving non-stiff or stiff IVP, backward differentiation formulae (variable coefficient formulae) with reference SIAM J. Sci. Stat. Comput. 10 (1989) 1038 - 1091, updated version of EPSODE.
- VODPK : Created by Brown, Byrne and Hindmarsh, solving large non-stiff or stiff IVP, backward differentiation formulae (variable coefficient formulae) with GMRES with user-supplied preconditioner, updated version of EPSODE.

Among the programs that are given at /netlib/odepack are :

- LSODE : Created by Alan C. Hindmarsh, solving stiff and nonstiff IVP, Adams methods and BDF methods, based on GEAR and GEARB packages.
- LSODA: Created by Linda R. Petzold Alan C. Hindmarsh, solving IVP with automatic selection between stiff and nonstiff IVP, variant of LSODE with reference SIAM J. Sci. Stat. Comput. 4 (1983) 136-148.
- SODE : Solving stiff and nonstiff systems of IVP, Adams method or Backward - Differentiation Formula with user supplied or difference quotient Jacobians.
- LSODAR : Created by Linda R. Petzold Alan C. Hindmarsh, solving stiff and nonstiff IVP with automatic selection, variant of LSODE with reference SIAM J. Sci. Stat. Comput. 4 (1983) 136-148.

Among the programs that are given at /netlib/toms are :

- GERK : Created by L.F. Shampine and H.A. Watts, solving IVP with global error estimate, Runge-Kutta-Fehlberg methods of 4th and 5th order with reference ACM TOMS 2 (1976) 200-203, under name toms/504.
- STINT : Created by J.M. Tandler, T.A. Bickart, and Z. Picel, solving stiff IVP, stiffly stable and cyclic composite linear multistep methods with reference ACM TOMS 4 (1978) 399-403, under name toms/534.
- ODESSA : Created by J. R. Leis and M. A. Kramer, solving IVP with explicit simultaneous sensitivity analysis, modified from LSODE with reference ACM TOMS 14 (1988) 61-67, under name toms/658.
- MEBDF : Created by Cash, Considine, solving stiff IVP, modified backward differentiation formulae with reference ACM Transaction on Math. Soft., vol. 18, No. 2 (June 1992) 142-155, under name toms/703.

APPENDIX B

ROOTS OF A COMPLEX POLYNOMIAL

All roots of a complex polynomial $p(z) = a_0 + a_1z + a_2z^2 + \dots + a_nz^n$, where a_0, a_1, \dots, a_n , and z are complex numbers, can be found easily using the Newton-Raphson method (Newton's method). The polynomials $p(z)$ and $p'(z)$ will be evaluated using nested multiplication, which is also known as Horner's rule. The nested multiplication form of the polynomial $p(z)$ is $((\dots((a_nz + a_{n-1})z + a_{n-2})z + \dots + a_2)z + a_1)z + a_0$. One of the algorithms to evaluate this function can be given as follows:

```
p = a_n
for i = n-1 downto 0
    p = p*z + a_i
next i
```

Using this algorithm, the need to perform any exponentiation can be avoided, so that we only need n multiplications and n additions.

Assume $p(z)$ can be written as $(z - zk)q(z)$, where $q(z)$ is a polynomial of degree $(n-1)$. If $q(z)$ is written as $b_{n-1}z^{n-1} + b_{n-2}z^{n-2} + \dots + b_2z^2 + b_1z + b_0$, then $p(z)$ can be written as $(z - zk)(b_{n-1}z^{n-1} + b_{n-2}z^{n-2} + \dots + b_2z^2 + b_1z + b_0) = b_{n-1}z^n + (b_{n-2} - zk*b_{n-1})z^{n-2} + \dots + (b_2 - zk*b_3)z^2 + (b_1 - zk*b_2)z + b_0$. From equating coefficients of power of z , we have the recursive relations $b_{n-1} = a_n$ and $b_{i-1} = a_i + zk*b_i$, for $i=n-1, n-2, \dots, 1$. Differentiating $p(z)$, we find $p'(z) = q(z) + (z - zk)q'(z)$, which gives $p'(zk) = q(zk)$.

A root of $p(z)$ can be found using the recursive relation of Newton's method:
 $z_{k+1} = z_k - p(z_k)/p'(z_k)$. The polynomial $p'(z_k)$ here can also be evaluated using nested multiplication since it is none other than the polynomial $q(z_k)$. One of the algorithms for evaluating $p(z)$ and $p'(z)$, for $z = z_k$ is given as follows :

```

p = an
q = an          /* Since bn-1 = an */
for i = n-1 down to 1
    p = p*zk + ai    /* Evaluate p(z) */
    q = q*zk + p      /* Evaluate p'(z) */
next i
p = p*zk + a0      /* The last evaluation of p(z) */

```

Here, we need only $2n$ multiplications and $2n$ additions in order to evaluate the polynomial $p(z)$ and the polynomial $p'(z)$. We can avoid the need for performing any exponentiations when evaluating $p(z)$ and $p'(z)$.

One of the algorithms for finding all complex roots of the complex polynomial $p(z)$ is given below:

```

z0 = 1 + j          /* j = √-1 imaginer number */
while (n > 0) do
    z = z0 + 2*ε
    while ( |z - z0| > ε ) do
        z0 = z
        p = an
        pz = an
        for i = n-1 downto 1
            p = p*z0 + ai    /* Evaluate p(z) */
            pz = pz*z0 + p    /* Evaluate p'(z) */
        next i
        p = p*z0 + a0      /* The last evaluation of p(z) */
        z = z0 - p/pz
    end while
    root(n) = z
    bn-1 = an
    n = n - 1
    for i = n-1 downto 0

```

```

                bi = ai+1 + z*bi+1           /* Evaluate coefficients of g(z) */
next i
for i = 0 to n
    ai = bi                               /* Rewrite p(z) with g(z) */
next i
if ( im(z) < ε ) then
    z = conjugate (z)
else
    z = 1 + j
endif
end while

```

The implementation of this algorithm using FORTRAN for the case $p(z) = z^3 - 3z^2 - z + 9$

is given as follows :

```

*-----
*   This is a main program for finding roots of polynomial p(z)
*-----
PROGRAM POLYNOM
IMPLICIT NONE
INTEGER          N
PARAMETER        (N=3)
COMPLEX *16      ROOT(N), A(N+1), B(N)
REAL *8          EPS
INTEGER          I, ORDER
DATA             EPS/1.D-9/
DATA             A/(9.D0,0.D0),(-1.D0,0.D0),(-3.D0,0.D0),
*               (1.D0,0.D0)/
ORDER = N
CALL RTPOL(ORDER,A,ROOT,EPS,B)
DO 10 I=1,N
    WRITE(6,100) I, DREAL(ROOT(I)), DIMAG(ROOT(I))
10  CONTINUE
100 FORMAT(5X,'z(',I2.2,') = ',F12.9,X,SP,F12.9,'J')
STOP
END

SUBROUTINE RTPOL(ORDER, A, ROOT, EPS,B)
IMPLICIT NONE
*-----
*   This is the May 12, 1995 version of
*   RTPOL ..ELSP Solver For finding root of polynomial
*   p(z) = anz^n + ....+ a2z^2 + a1z + a0.

```

```

*      This version is in double precision and written using F77.
*-----
*      Reference ..
*      Edward Purba, Compact Numerical Methods for Stiff Differential Equations,
*      Master Thesis, Computer Science, Oklahoma State University,
*      1996.
*-----
*      ORDER Order of the polynomial
*      ROOT  Roots of the polynomial
*      A     Coefficient of polynomial p(z)
*      B     Coefficient of g(z) where p(z)=(z-zst)g(z)
*-----
      COMPLEX *16      ROOT(1), A(1), B(1)
      INTEGER          ORDER
      REAL *8          EPS
      COMPLEX *16      Z0, ZST, P, PZ
      INTEGER          I
      Z0 = DCMPLX(1.D0,1.D0)
      DO WHILE( ORDER .GT. 1)
          ZST = Z0 + DCMPLX(2.D0*EPS,0.D0)
          DO WHILE (CDABS(ZST - Z0) .GT. EPS)
              Z0 = ZST
              P = A(ORDER+1)
              PZ= A(ORDER+1)
              DO I=ORDER,2,-1
*      Calculate polynomial p(z) using Horner's rule.
                  P = P*Z0 + A(I)
*      Calculate polynomial p'(z) using Horner's rule.
                  PZ = PZ*Z0 + P
              END DO
*      Calculate the outermost part of p(z)
                  P = P*Z0 + A(1)
*      Calculate the root using Newton-Raphson
                  ZST = Z0 - P/PZ
              END DO
*      The root is found.
                  ROOT(ORDER) = ZST
                  B(ORDER) = A(ORDER+1)
*      Deflate polynomial by division (z - zst) so that p(z)=(z-zst)g(z)
                  ORDER = ORDER - 1
*      Assign coefficients of b's
                  DO I=ORDER,1,-1
                      B(I) = A(I+1) + ZST*B(I+1)
                  END DO
*      Renew coefficient A with coefficient B

```

```

      DO I=1,ORDER+1
        A(I) = B(I)
      END DO
*   If the root is complex, then its conjugate is possible as a root.
      IF(DABS(DIMAG(ZST)) .GT. EPS) THEN
        ZST = DCONJG(ZST)
      ELSE
        ZST = DCMPLX(1.D0,1.D0)
      ENDIF
    END DO
    ROOT(1) = -A(1)/A(2)
    RETURN
  END

```

The output of this program is given as follows:

```

z(01) = -1.525102255 + 0.000000000J
z(02) = 2.262551127 -0.884367598J
z(03) = 2.262551127 +0.884367598J

```

The output here is given in the form of complex number.

The idea of finding a root of the polynomial $p(z)$ given above, can be applied in order to find the region of stability of an explicit Runge-Kutta method. A FORTRAN program for plotting the stability region of an explicit Runge-Kutta method is given below:

```

*-----
*   This is a main program for STABLE-SUBROUTINE
*   This program creates data for plotting of the stability region
*-----
PROGRAM REGSTAB
IMPLICIT NONE
INTEGER          N, M
PARAMETER        (N=50, M=20)
COMPLEX *16      Z(N*M), COEFF(M+1)
REAL *8          EPS
INTEGER          I, ORDER, OPT
DATA             OPT/1/, EPS/1.D-9/
READ(*,*) ORDER
CALL STABLE(Z, COEFF, EPS, ORDER, OPT, N)
DO 10 I=1,N*ORDER

```

```

          PRINT 100, DREAL(Z(I)), DIMAG(Z(I))
10      CONTINUE
100     FORMAT(2(F10.5,2X))
        STOP
        END

SUBROUTINE STABLE(Z, COEFF, EPS, ORDER, OPT, N)
IMPLICIT NONE
*-----
*      This is the May 12, 1995 version of
*      STABLE .. ELSP Solver For Creating The data for plotting the region of
*      stability of Explicit Runge-Kutta Methods.
*      This version is in double precision and written using F77.
*      STABLE is a package based on Polynomial Algorithm for zero complex
*-----
*      Reference ..
*      Edward Purba, Compact Numerical Methods for Stiff Differential Equations,
*      Master Thesis, Computer Science, Oklahoma State University,
*      1996.
*-----
*      N          Number of dividing angle steps
*      ORDER      Runge-Kutta ORDER >= 1
*      Z          Complex variables with, dimension N*ORDER
*      COEFF      Variable for storing the coefficient of polynomial,
*                dimension ORDER+1.
*      OPT        Option of stability function
*                OPT = 1,  Explicit Runge-Kutta with the form
*                 $|r(Z)| = |1 + (lh) + (lh)^2/2 + \dots + (lh)^s/s!| < 1.$ 
*-----
      COMPLEX *16    Z(1), COEFF(1)
      REAL *8        EPS
      INTEGER        N, OPT, ORDER
      COMPLEX *16    Z0, FZ, DFZ
      REAL *8        PI, THETA, T
      INTEGER        I, K
      DATA          PI /3.14159265359D0/

      IF(ORDER .LT. 1) THEN
          PRINT*, '***ERROR ----ORDER SHOULD BE GREATER THAN 0'
          STOP
      ENDIF
      IF(OPT .EQ. 1) CALL CRCOEF(COEFF,ORDER)
      THETA = 2.D0*PI/FLOAT(N)
      Z0 = DCMLPX(0.D0,0.D0)
      DO 10 K = 1, N*ORDER

```



```

          T = DFLOAT(K) * THETA
15  CONTINUE
          FZ = COEFF(ORDER+1)
          DFZ = COEFF(ORDER+1)
*   Horner's rule is used to represent the polynomial
          DO I = ORDER, 2, -1
*   Calculate  $F(Z) = r(Z) - e^{iT}$ 
          FZ = FZ*Z0 + COEFF(I)
*   Calculate  $DF(Z)/DZ$ 
          DFZ = DFZ*Z0 + FZ
          END DO
          FZ = FZ*Z0 + COEFF(1)
          FZ = FZ - DCMLPX(DCOS(T),DSIN(T))
          Z(K) = Z0 - FZ/DFZ
          IF(CDABS(Z(K)-Z0) .LT. EPS) GOTO 10
          Z0 = Z(K)
          GOTO 15
10  CONTINUE
RETURN
END

SUBROUTINE CRCOEF(COEFF,ORDER)
IMPLICIT NONE
*-----
*   This subroutine is used to create the coefficients of the polynomial
*   of the stability of an explicit Runge-Kutta Method.
*   ORDER      Runge-Kutta ORDER >= 1
*   COEFF      Variable for storing the coefficient of polynomial,
*             dimension ORDER+1.
*-----
COMPLEX  *16  COEFF(1)
INTEGER  ORDER
INTEGER  I, NFACT
NFACT = 1
COEFF(1) = 1.D0
DO I=1,ORDER
    NFACT = NFACT*I
    COEFF(I+1) = DCMLPX(1.D0/DFLOAT(NFACT), 0.D0)
END DO
RETURN
END

```

A graphical example of this program for input ORDER = 1, 2, 3, 4, and 5 is given in Figure B-1 in the next page.

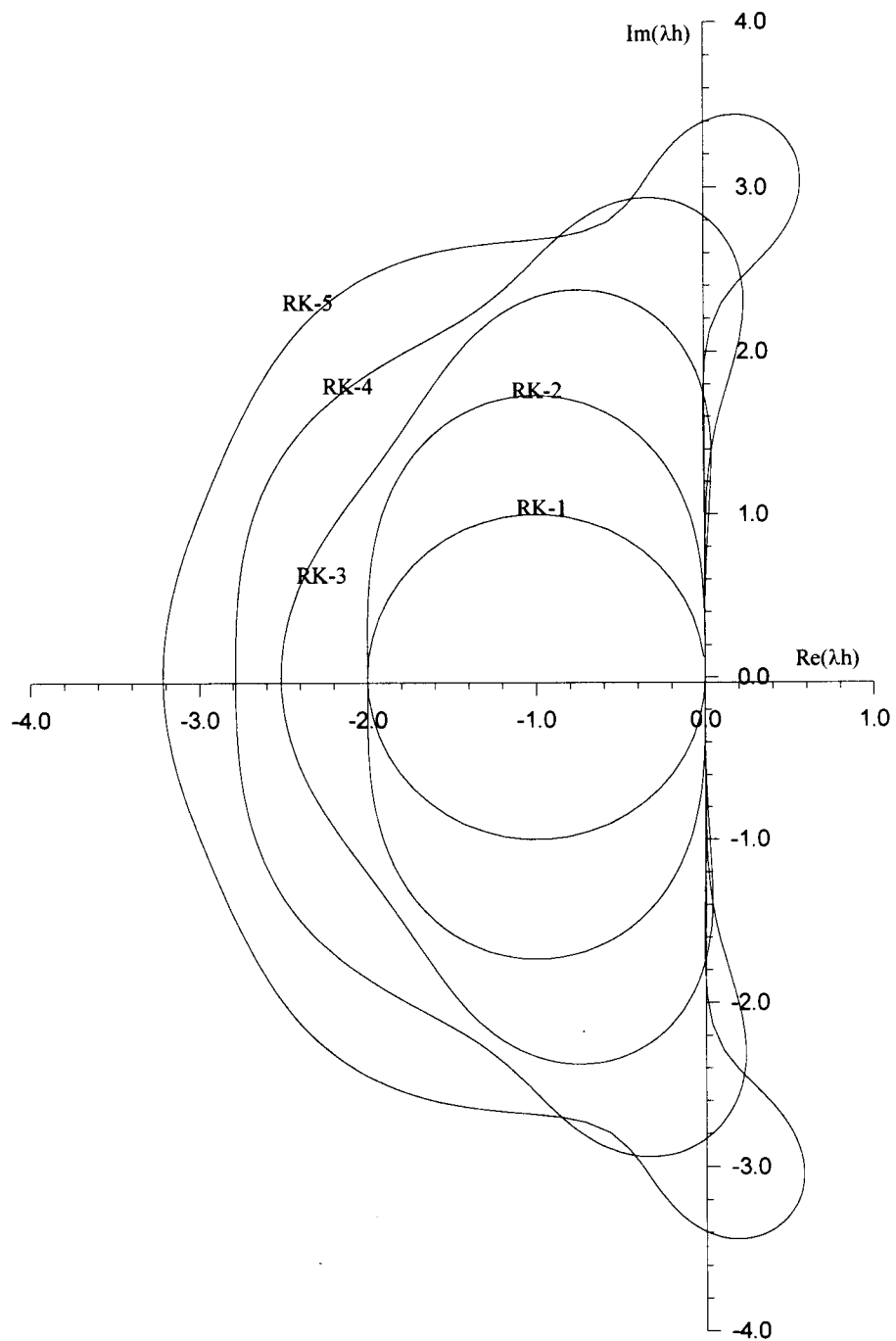


Figure B-1 Stability Region of Runge-Kutta Methods
Orders 1, 2, 3, 4, and 5

APPENDIX C

LINEAR MULTISTEP FORMULAS

There are two approaches for multistep methods that will be given in this appendix related to ODE problems $y'=f(t,y)$. The first is to approximate the function $f(t,y)$ with a polynomial and then integrate both sides. The second is to approximate the function $y(t)$ with a polynomial and then differentiate the approximating function. All polynomial functions that are used in these approximations are generated by the Newton Backward Difference formula. The backward difference of the y_k is denoted by ∇y_k and defined as $y_k - y_{k-1}$. The form of $\nabla^0 y_k$ is defined as y_k ,

$$\nabla^2 y_k = \nabla y_k - \nabla y_{k-1} = y_k - 2y_{k-1} + y_{k-2}, \text{ and } \nabla^n y_k = \nabla^{n-1} y_k - \nabla^{n-1} y_{k-1}.$$

The Newton backward difference interpolation of the k known values $y_{n-k+1}, y_{n-k+2}, \dots, y_{(n-1)}, y_n$ is written as

$$y(t) = y(t_n + sh) = y_n + s\nabla y_n + \frac{s(s+1)}{2!} \nabla^2 y_n + \dots + \frac{s(s+1) \dots (s+k-1)}{(k-1)!} \nabla^{k-1} y_n \quad (\text{C-1})$$

where $s = (t - t_n)/h$, and $t_n \leq t \leq t_{n+1}$. From discrete mathematics, we know that

$$\binom{-s}{j} = \frac{(-s)(-s-1) \dots (-s-j+1)(-s-j)!}{j!(-s-j)!} = (-1)^j \frac{s(s+1) \dots (s+j-1)}{j!}.$$

Equation (C-1) then can be written as

$$y(t) = y(t_n + sh) = \sum_{j=0}^{k-1} (-1)^j \binom{-s}{j} \nabla^j y_n. \quad (\text{C-2})$$

Adams-Bashforth Methods

John Couch Adams used the Newton backward difference interpolation to approximate $f(t,y)$ by assuming that he already had k known points (t_{n-k+1}, y_{n-k+1}) , (t_{n-k+2}, y_{n-k+2}) , \dots , (t_{n-1}, y_{n-1}) , (t_n, y_n) . The polynomial approximation $p^*(t)$ of $f(t,y)$ is written as

$$p^*(t) = p^*(t_n + sh) = \sum_{j=0}^{k-1} (-1)^j \binom{-s}{j} \nabla^j f_n.$$

By integrating both sides, we have

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} \sum_{j=0}^{k-1} (-1)^j \binom{-s}{j} \nabla^j f_n dt. \quad (\text{C-3})$$

We know that $dt = hds$, for $t = t_n$, $s = 0$, and for $t = t_{n+1}$, $s = 1$. By integrating (C-3) over s from 0 to 1, we have

$$y_{n+1} = y_n + h \int_0^1 \sum_{j=0}^{k-1} (-1)^j \binom{-s}{j} \nabla^j f_n ds$$

Since the $\nabla^j y_n$'s do not depend on s , we can move the summation out of the integration, so that we have

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \left[(-1)^j \nabla^j f_n \int_0^1 \binom{-s}{j} ds \right].$$

Suppose that

$$\gamma_j^* = (-1)^j \int_0^1 \binom{-s}{j} ds,$$

then the methods can be written as

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \gamma_j^* \nabla^j f_n. \quad (\text{C-4})$$

These methods are called Adams methods or Adams-Bashforth methods

The coefficients γ_j^* are determined using Euler's method of generating functions.

The way to find these coefficients is by seeking $G(t)$ such that $G(t)$ can be written as

$$\begin{aligned} G(t) &= \sum_{j=0}^{\infty} \gamma_j^* t^j = \sum_{j=0}^{\infty} (-t)^j \int_0^1 \binom{-s}{j} ds = \int_0^1 \sum_{j=0}^{\infty} (-t)^j \binom{-s}{j} ds = \int_0^1 (1-t)^{-s} ds. \\ &= \int_0^1 e^{\ln(1-t)^{-s}} ds = \int_0^1 e^{-s \ln(1-t)} ds = \frac{-1}{(1-t)^s \ln(1-t)} \Big|_{s=0}^{s=1} = \frac{-t}{(1-t) \ln(1-t)}. \end{aligned} \quad (\text{C-5})$$

We know from Taylor series that

$$\frac{-\ln(1-t)}{t} = 1 + \frac{t}{2} + \frac{t^2}{3} + \dots$$

and

$$\frac{1}{1-t} = 1 + t^2 + t^3 + t^4 + \dots$$

From (C-5), we can have a relation

$$(\gamma_0^* + \gamma_1^* t + \gamma_2^* t^2 + \dots)(1 + t/2 + t^2/3 + t^3/4 \dots) = 1 + t + t^2 + t^3 + \dots$$

By comparing the coefficients of t^i of both sides, we can get the recurrence relation,

$$\gamma_i^* + \frac{1}{2} \gamma_{i-1}^* + \frac{1}{3} \gamma_{i-2}^* + \dots + \frac{1}{i+1} \gamma_0^* = 1, \quad i = 0, 1, 2, \dots \quad (\text{C-6})$$

It can be easily shown that:

For $i = 0$, $\gamma_0^* = 1$.

For $i = 1$, $\gamma_1^* + \gamma_0^*/2 = 1$ and $\gamma_1^* = 1 - \gamma_0^*/2 = 1/2$.

For $i = 2$, $\gamma_2^* + \gamma_1^*/2 + \gamma_0^*/3 = 1$ and $\gamma_2^* = 1 - \gamma_1^*/2 - \gamma_0^*/3 = 1 - 1/4 - 1/3 = 5/12$.

For $i = 3$, $\gamma_3^* + \gamma_2^*/2 + \gamma_1^*/3 + \gamma_0^*/4 = 1$ and

$$\gamma_3^* = 1 - \gamma_2^*/2 - \gamma_1^*/3 - \gamma_0^*/4 = 1 - 5/24 - 1/6 - 1/4 = 3/8.$$

For $i = 4$, $\gamma_4^* + \gamma_3^*/2 + \gamma_2^*/3 + \gamma_1^*/4 + \gamma_0^*/5 = 1$ and

$$\gamma_4^* = 1 - \gamma_3^*/2 - \gamma_2^*/3 - \gamma_1^*/4 - \gamma_0^*/5 = 1 - 3/16 - 5/36 - 1/8 - 1/5 = 251/720.$$

For $i = 5$, $\gamma_5^* + \gamma_4^*/2 + \gamma_3^*/3 + \gamma_2^*/4 + \gamma_1^*/5 + \gamma_0^*/6 = 1$ and

$$\gamma_5^* = 1 - \gamma_4^*/2 - \gamma_3^*/3 - \gamma_2^*/4 - \gamma_1^*/5 - \gamma_0^*/6 = 1 - 251/1440 - 3/24 - 5/48 - 1/10 - 1/6 = 95/288.$$

By substituting the values of γ_j^* 's into (C-4), we can write the family of Adams-Bashforth methods as

$$\begin{aligned} y_{n+1} &= y_n + h(\gamma_0^* f_n + \gamma_1^* \nabla f_n + \gamma_2^* \nabla^2 f_n + \gamma_3^* \nabla^3 f_n + \gamma_4^* \nabla^4 f_n + \dots) \\ &= y_n + h(f_n + \frac{1}{2} \nabla f_n + \frac{5}{12} \nabla^2 f_n + \frac{3}{8} \nabla^3 f_n + \frac{251}{720} \nabla^4 f_n + \dots) \end{aligned} \quad (C-7)$$

For the cases $k = 1, 2, 3, 4, 5$, and 6 , Equation (C-7) gives

$$k = 1, \quad y_{n+1} = y_n + hf_n \quad (\text{Explicit Euler})$$

$$k = 2, \quad y_{n+1} = y_n + h/2(3f_n - f_{n-1})$$

$$k = 3, \quad y_{n+1} = y_n + h/12(23f_n - 16f_{n-1} + 5f_{n-2})$$

$$k = 4, \quad y_{n+1} = y_n + h/24(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})$$

$$k = 5, \quad y_{n+1} = y_n + h/720(1901f_n - 2774f_{n-1} + 2616f_{n-2} - 1274f_{n-3} + 251f_{n-4})$$

$$k = 6, \quad y_{n+1} = y_n + h/1440(4277f_n - 7923f_{n-1} + 9982f_{n-2} - 7298f_{n-3} + 2877f_{n-4} - 475f_{n-5})$$

The interesting thing in this derivation of formulas is that the γ_j^* 's do not depend on k .

If we already know the formula for Adams-Bashforth methods for $k = m$, we can create

the next $(k+1)$ form of the Adams-Bashforth methods, just by adding the new term with the new coefficient γ_k^* . Since we already know the values of $\gamma_0^*, \gamma_1^*, \dots, \gamma_{m-1}^*$, the value of γ_m^* can be found just by applying the recurrence relation (C-6) with $i = k$.

Adams-Moulton Methods

The implicit Adams methods are generated using k known values $y_{n-k+1}, y_{n-k+2}, \dots, y_{n-1}, y_n$ and one unknown value y_{n+1} . These methods are called implicit because the value y_{n+1} will be approximated using the previous values y_{n-k+1}, \dots, y_n , and y_{n+1} itself. The Newton backward difference interpolation to approximate $f(t, y)$ using these $k+1$ values $y_{n-k+1}, y_{n-k+2}, \dots, y_n, y_{n+1}$ is written as

$$p(t) = p(t_n + sh) = \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j f_{n+1}.$$

By analogy with the Adams-Bashforth methods, we have

$$y_{n+1} = y_n + h \sum_{j=0}^k \gamma_j \nabla^j f_{n+1}, \quad (\text{C-8})$$

where the coefficients γ_j satisfy

$$\gamma_j = (-1)^j \int_0^1 \binom{-s+1}{j} ds. \quad (\text{C-9})$$

By using Euler's method of generating functions, we take $G(t)$ as a function in the form

$$G(t) = \sum_{j=0}^{\infty} \gamma_j t^j.$$

As was done for the Adams-Bashforth methods, we have

$$G(t) = \frac{-1}{(1-t)^{s-1} \ln(1-t)} \Big|_{s=0}^{s=1} = \frac{-t}{\ln(1-t)}. \quad (\text{C-10})$$

From Equation (C-10), we have

$$(\gamma_0 + \gamma_1 t + \gamma_2 t^2 + \dots)(1 + t/2 + t^2/3 + t^3/4 \dots) = 1.$$

By comparing the coefficients of t^i of both sides, we can get the recurrence relation

$$\gamma_i + \frac{1}{2}\gamma_{i-1} + \frac{1}{3}\gamma_{i-2} + \dots + \frac{1}{i+1}\gamma_0 = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } i = 1, 2, \dots \end{cases} \quad (\text{C-11})$$

Equation (C-8) with the recurrence relation (C-11) are also called Adams-Moulton methods.

It can be easily shown that:

$$\text{For } i = 0, \gamma_0 = 1.$$

$$\text{For } i = 1, \gamma_1 + \gamma_0/2 = 0 \text{ and } \gamma_1 = -\gamma_0/2 = -1/2.$$

$$\text{For } i = 2, \gamma_2 + \gamma_1/2 + \gamma_0/3 = 0 \text{ and } \gamma_2 = -\gamma_1/2 - \gamma_0/3 = +1/4 - 1/3 = -1/12.$$

$$\text{For } i = 3, \gamma_3 + \gamma_2/2 + \gamma_1/3 + \gamma_0/4 = 0 \text{ and } \gamma_3 = -\gamma_2/2 - \gamma_1/3 - \gamma_0/4 = 1/24 + 1/6 - 1/4 = -1/24.$$

$$\text{For } i = 4, \gamma_4 + \gamma_3/2 + \gamma_2/3 + \gamma_1/4 + \gamma_0/5 = 1 \text{ and}$$

$$\gamma_4 = -\gamma_3/2 - \gamma_2/3 - \gamma_1/4 - \gamma_0/5 = 1/48 + 1/36 + 1/8 - 1/5 = -19/720.$$

$$\text{For } i = 5, \gamma_5 + \gamma_4/2 + \gamma_3/3 + \gamma_2/4 + \gamma_1/5 + \gamma_0/6 = 0 \text{ and}$$

$$\gamma_5 = -\gamma_4/2 - \gamma_3/3 - \gamma_2/4 - \gamma_1/5 - \gamma_0/6 = 19/1440 + 1/72 + 1/48 + 1/10 - 1/6 = -3/160.$$

By substituting the values of the γ_j 's into (C-8), we can write the family of Adams-Moulton methods as

$$\begin{aligned} y_{n+1} &= y_n + h(\gamma_0 f_{n+1} + \gamma_1 \nabla f_{n+1} + \gamma_2 \nabla^2 f_{n+1} + \gamma_3 \nabla^3 f_{n+1} + \gamma_4 \nabla^4 f_{n+1} + \dots) \\ &= y_n + h(f_{n+1} - \frac{1}{2} \nabla f_{n+1} - \frac{1}{12} \nabla^2 f_{n+1} - \frac{1}{24} \nabla^3 f_{n+1} - \frac{19}{720} \nabla^4 f_{n+1} + \dots) \end{aligned} \quad (\text{C-12})$$

For the cases $k = 0, 2, 3, 4,$ and $5,$ Equation (C-8) gives

$$k = 0, \quad y_{n+1} = y_n + hf_{n+1} \quad (\text{Implicit Euler})$$

$$k = 1, \quad y_{n+1} = y_n + h/2(f_{n+1} + f_n) \quad (\text{Trapezoidal})$$

$$k = 2, \quad y_{n+1} = y_n + h/12(5f_{n+1} + 8f_n - f_{n-1})$$

$$k = 3, \quad y_{n+1} = y_n + h/24(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2})$$

$$k = 4, \quad y_{n+1} = y_n + h/720(251f_{n+1} + 646f_n - 264f_{n-1} + 106f_{n-2} - 19f_{n-3})$$

$$k = 5, \quad y_{n+1} = y_n + h/1440(475f_{n+1} + 1427f_n - 798f_{n-1} + 482f_{n-2} - 173f_{n-3} + 27f_{n-4})$$

Backward Differentiation Formulae (BDF)

The approach of finding a discrete solution using these methods are different from the Adams methods. Here, the function that will be approximated is $y(t)$, and the procedure of solving the solution is done by differentiating $y(t)$. Similar to Adams methods, the polynomial Newton backward difference interpolation will be used to approximate $y(t)$. The interpolation is done by considering k known values $y_{n-k+1}, y_{n-k+2}, \dots, y_{n-1}, y_n,$ and one unknown values y_{n+1} . The polynomial approximation $q(t)$ of $y(t)$ then is written as

$$q(t) = q(t_n + sh) = \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j y_{n+1}. \quad (\text{C-13})$$

We know that $dq/dt = (dq/ds)(ds/dt) = (1/h)dq/ds,$ so if we substitute this into $y' = f(t,y),$ we have

$$\frac{1}{h} \frac{d}{ds} \sum_{j=0}^k (-1)^j \binom{-s+1}{j} \nabla^j y_{n+1} = f_{n+1}.$$

$$\frac{1}{h} \sum_{j=0}^k (-1)^j \nabla^j y_{n+1} \frac{d}{ds} \binom{-s+1}{j} = f_{n+1}. \quad (\text{C-14})$$

Since

$$\binom{-(s-1)}{j} = (-1)^j \frac{(s-1)(s)(s+1) \dots (s-1+j-1)}{j!},$$

and $t_{n+1} = t_n + sh$ for $s=1$, then

$$\begin{aligned} \frac{d}{ds} \binom{-(s-1)}{j} \Big|_{s=1} &= (-1)^j \frac{d}{ds} \frac{(s-1)(s)(s+1) \dots (s+j-1)}{j!} \Big|_{s=1} \\ &= \frac{(-1)^j}{j!} \{ (s)(s+1) \dots (s+j-2) + (s-1)(s+1) \dots (s+j-2) \\ &\quad + \dots + (s-1)s(s+1) \dots (s+j-3) \} \Big|_{s=1} \end{aligned}$$

For case $j=0$,

$$\frac{d}{ds} \binom{-(s-1)}{0} \Big|_{s=1} = \frac{d}{ds} \frac{(-s+1)!}{0!(-s+1)!} \Big|_{s=1} = \frac{d}{ds} 1 \Big|_{s=1} = 0.$$

For case $j=1$,

$$\frac{d}{ds} \binom{-(s-1)}{1} \Big|_{s=1} = \frac{d}{ds} \frac{(-s+1)!}{0!(-s+1-1)!} \Big|_{s=1} = \frac{d}{ds} (-s+1) \Big|_{s=1} = -1 = (-1)^1 0!.$$

Let us take $s=1$. Since for $j=2$, $(s)(s+1) \dots (s+j-2) = (s) = 1!$, for $j=3$, $(s)(s+1) \dots (s+j-2) = (s)(s+1) = 2!$, for $j=k$, $(s)(s+1) \dots (s+j-2) = (1)(2) \dots (k-1) = (k-1)!$, $k \geq 2$, then

$$\frac{d}{ds} \binom{-(s-1)}{j} \Big|_{s=1} = \frac{(-1)^j}{j!} \begin{cases} 0 & \text{for } j = 0 \\ (j-1)! & \text{for } j \geq 1 \end{cases} = (-1)^j \begin{cases} 0 & \text{for } j = 0 \\ \frac{1}{j} & \text{for } j \geq 1 \end{cases}. \quad (\text{C-15})$$

By substituting (C-15) into (C-14), we have

$$\sum_{j=1}^k (-1)^{2j} \binom{1}{j} \nabla^j y_{n+1} = hf_{n+1}.$$

Since $(-1)^{2j} = 1$ for every $j \geq 1$, then

$$\sum_{j=1}^k \binom{1}{j} \nabla^j y_{n+1} = hf_{n+1}. \quad (\text{C-16})$$

These multistep methods are called backward differentiation formulae (BDF) methods.

For the case $k = 1, 2, 3, 4, 5$, and 6 , the equation (C-16) gives

$$k = 1, \quad y_{n+1} = y_n + hf_{n+1} \quad (\text{Implicit Euler})$$

$$k = 2, \quad y_{n+1} = (4/3)y_n - (1/3)y_{n-1} + (2/3)hf_{n+1}$$

$$k = 3, \quad y_{n+1} = (18/11)y_n - (9/11)y_{n-1} + (2/11)y_{n-2} + (6/11)hf_{n+1}.$$

$$k = 4, \quad y_{n+1} = (48/25)y_n - (36/25)y_{n-1} + (16/25)y_{n-2} - (3/25)y_{n-3} + (12/25)hf_{n+1}.$$

$$k = 5, \quad y_{n+1} = (300/137)y_n - (300/137)y_{n-1} + (200/137)y_{n-2} - (75/137)y_{n-3} + \\ (12/137)y_{n-4} + (60/137)hf_{n+1}.$$

$$k = 6, \quad y_{n+1} = (360/147)y_n - (450/147)y_{n-1} + (400/147)y_{n-2} - (225/147)y_{n-3} + \\ (72/147)y_{n-4} - (10/147)y_{n-5} + (60/147)hf_{n+1}.$$

APPENDIX D

ESTIMATION OF ERROR AND STEPSIZE CONTROL

Economically, it is suggested to choose a stepsize not to be constant for the whole range of computation. The stepsize control policy becomes important not only for deciding the next stepsize but also for deciding the starting stepsize. The objective of selecting the stepsize is to minimize the computing time of numerical solution of ODEs by considering a user-specified accuracy requirement. In this case, the stepsize should be chosen small if $f(t,y)$ or $y' = f(t,y)$ is varying fast; on the other hand, if f is changing slowly, the stepsize can be chosen larger but still satisfy the accuracy of solution. In choosing the stepsize, the instability of the numerical computation should be considered. Usually, techniques for adaptive stepsize selection hinge on the local error at each step. Since users of ODE solvers do not want to be involved in designating an initial stepsize for the computation, but can be asked to supply a tolerance for the numerical results, the ODE solvers should support the automatic selection of stepsizes.

The series solutions of $y' = f(t,y)$ from t_0 to t_f will be given as $(t_0, y_0), (t_1, y_1), \dots, (t_N, y_N)$ with $t_0 < t_1 < t_2 < \dots < t_N = t_f$. For every stepsize h , there are always two solutions; the approximate solution y_{n+1} , and local exact solution $u(t_{n+1})$. The local error at step $n+1$ is defined as $l_{n+1} = u(t_{n+1}) - y_{n+1}$. Here, $u(t_{n+1})$ is the exact solution of $u' = f(t,u)$ at t_{n+1} , where $u(t_n) = y_n$ and $t_n \leq t \leq t_{n+1}$. The true error or global error is defined as

$g_{n+1} = y(t_{n+1}) - y_{n+1} = y(t_{n+1}) - u(t_{n+1}) + l_{n+1}$, where $y(t_{n+1})$ is the exact solution of $y' = f(t,y)$, $y(t_0) = y_0$, $t_0 \leq t \leq t_f$. Since at the first step the local exact solution is the same as the true solution, the local discretization error for the first step will also be the same as the global error.

Accumulative Error of One-Step Methods

The general one-step method for solving $y' = f(t,y)$ is defined as $y_{n+1} = y_n + h\phi(t_n, y_n; h)$, where ϕ is known as the increment function and always depends on $f(t,y)$. The Euler method is one example of one-step methods where ϕ is f itself. The local discretization error of the one-step method is

$$l_{n+1}(h) = u(t_n+h) - y_n - h\phi(t_n, y_n; h) = u(t_n+h) - u(t_n) - h\phi(t_n, y_n; h),$$

where $u(t_{n+1})$ is the local exact solution of $u' = f(t,u)$ at t_{n+1} , for $u(t_n) = y_n$. The Taylor series of u about $h=0$ is given as

$$u(t_{n+1}) = u(t_n) + hu'(t_n) + h^2u''(t_n)/2! + \dots + h^N u^{(N)}(t_n)/N! + O(h^{N+1}).$$

Let $\psi(h) = h\phi(t_n, y_n; h)$. We can find that $\psi^{(m)}(h) = m\phi_{(m-1),h}(t_n, y_n; h) + h\phi_{(m),h}(t_n, y_n; h)$, where $\phi_{(k),h}(t_n, y_n; h)$ is the k th partial derivative of $\phi(t_n, y_n; h)$. The Taylor series of ψ about $h=0$ is

$$\begin{aligned} \psi(h) = & h\phi(t_n, y_n; 0) + 2h^2\phi_{1,h}(t_n, y_n; 0)/2! + 3h^3\phi_{2,h}(t_n, y_n; 0)/3! + \dots + \\ & Nh^N\phi_{(N-1),h}(t_n, y_n; 0)/N! + O(h^{N+1}). \end{aligned}$$

By substituting the Taylor series of u and ψ about $h=0$ into the local discretization error l_{n+1} , we have the Taylor series of l_{n+1} about $h=0$ as

$$l_{n+1}(h) = h[u'(t_n) - \phi(t_n, y_n; 0)] + h^2/2![u''(t_n) - 2\phi_{1,h}(t_n, y_n; 0)] + \dots$$

$$\begin{aligned}
& + h^N/N! [u^{(N)}(t_n) - N\phi_{(N-1),h}(t_n, y_n; 0)] + O(h^{N+1}) \\
& = h \left\{ \sum_{k=1}^N [h^{k-1} G_k(t_n)] + O(h^N) \right\}, \\
G_k(x) & = \frac{u^{(k)}(x)}{k!} - \frac{\phi_{(k-1),h}(x, y; 0)}{(k-1)!}.
\end{aligned}$$

From the local discretization error, we have

$$\frac{u(t_n + h) - u(t_n)}{h} - \phi(t_n, y_n; h) = \sum_{k=1}^N h^{k-1} G_k(t_n) + O(h^N). \quad (D-1)$$

A one-step method is said to be of order $p \geq 1$, if $|h^{-1}l_{n+1}| = O(h^p)$. From (D-1), a one-step method is of order $p \geq 1$ if $G_1(x) = G_2(x) = \dots = G_p(x) = 0$, $G_{p+1}(x) \neq 0$. A one-step method $y_{n+1} = y_n + h\phi(t_n, y_n; h)$ is said to be convergent if for arbitrary $t_n \in [t_0, t_f]$, $\lim_{h \rightarrow 0} y(t; h) = y(t)$. A one-step method $y_{n+1} = y_n + h\phi(t_n, y_n; h)$ is said to be consistent if $\phi(t, y; 0) = f(t, y)$. It has been shown [61, 66, 115] that the necessary and sufficient condition for convergence of a one-step method of order $p \geq 1$ is consistency.

Theorem (D-1): The accumulated error of a one-step method of order p is of order h^p .

Proof:

Suppose $e_n = y(t_n) - y_n$; then $e_{n+1} = y(t_{n+1}) - y_{n+1}$. By manipulating $y(t_{n+1})$, we have

$$e_{n+1} = \left[y(t_n) - h \left(\frac{y(t_{n+1}) - y(t_n)}{h} \right) \right] - [y_n + h\phi(t_n, y_n; h)]. \quad (D-2)$$

By doing a little manipulation on (D-2) and considering the local truncation error, we have

$$e_{n+1} = y(t_n) - y_n + h \left(\frac{y(t_{n+1}) - y(t_n)}{h} - \phi(t_n, y(t_n); h) + \phi(t_n, y(t_n); h) - \phi(t_n, y_n; h) \right)$$

$$= e_n + hO(h^p) + h(\phi(t_n, y(t_n); h) - \phi(t_n, y_n; h))$$

Since $y(t_n) = y_n + e_n$, we can have $e_{n+1} = e_n + hF_n(e_n) + hO(h^p)$, where

$$F_n(e_n) = \phi(t_n, y_n + e_n; h) - \phi(t_n, y_n; h).$$

Expanding F_n using Taylor series about $e_n = e_0 = 0$, we have

$$F_n(e_n) = e_n \phi_{1, e_n}(t_n, y_n + d; h),$$

where $e_0 \leq d \leq e_n$. Let $K = \max \{|\phi_{1, y}(t, y; h)|\}$, then $|F_n(e_n)| \leq K|e_n|$. It has been shown by

Walston and Waddel [115] that

$$|y(t_n) - y_n| = |e_n| \leq \frac{e^{nhK} - 1}{K} O(h^p). \quad (D-3)$$

Examples :

1. Consider the Euler method $y_{n+1} = y_n + hf(t_n, y_n)$.

Here, $u' = f(t, y)$, and $\phi(t, y; h) = f(t, y)$.

$$u' = f, u'' = f_t + ff_y, u''' = f_{tt} + 2ff_{ty} + f^2f_{yy} + f_y(f_t + ff_y).$$

$$\phi_{1, h}(t, y; h) = f(t, y), \phi_{2, h}(t, y; h) = 0.$$

$$G_1(t) = u'(x)/1! - \phi_{0, h}(t, y; 0)/0! = f(t, y) - f(t, y) = 0.$$

$$G_2(t) = u''(t)/2! - \phi_{1, h}(t, y; 0)/1! = (f_t + ff_y)/2 - 0 \neq 0.$$

Therefore the Euler method is of order 1.

2. Consider the Runge-Kutta method $y_{n+1} = y_n + 1/2[hf(t_n, y_n) + hf(t_n + h, y_n + hf(t_n, y_n))]$.

Here, $u' = f(t, y)$ and $\phi(t, y; h) = 1/2f(t, y) + 1/2f(t+h, y+hf(t, y))$.

Let $v = t + h$, $w = y + hf(t, y)$, and $\psi(v, w) = f(v, w)$.

$$\phi_{1, h}(t, y; h) = 1/2[v_h \cdot \psi_{1, v}(v, w) + w_h \cdot \psi_{1, w}(v, w)] = 1/2[f_v(v, w) + f(v, w)f_w(v, w)].$$

$$\phi_{1,h}(t,y;0) = 1/2[f_t(t,y) + f(t,y)f_y(t,y)].$$

$$\phi_{2,h}(t,y;h) = 1/2[f_{vv}(v,w) + 2f(v,w)f_{vw}(v,w)f(v,w)^2f_{ww}(v,w) + f_y(v,w)(f_v(v,w) + f(v,w)f_v(v,w))].$$

$$\phi_{2,h}(t,y;0) = 1/2[f_{tt} + 2ff_{ty} + f^2f_{yy} + f_y(f_t + ff_t)].$$

$$G_1(t) = u'(x)/1! - \phi_{0,h}(t,y;0)/0! = f(t,y) - 1/2[f(t,y) + f(t,y)] = 0.$$

$$G_2(t) = u''(t)/2! - \phi_{1,h}(t,y;0)/1! = 1/2[f_t + ff_y] - 1/2[f_t + ff_y] = 0.$$

$$G_3(t) = u'''(t)/3! - \phi_{2,h}(t,y;0)/2! = 1/6[f_{tt} + 2ff_{ty} + f^2f_{yy} + f_y(f_t + ff_y)] - 1/4[f_{tt} + 2ff_{ty} + f^2f_{yy} + f_y(f_t + ff_y)] \neq 0.$$

Therefore the Runge-Kutta $y_{n+1} = y_n + 1/2[hf(t_n, y_n) + hf(t_n+h, y_n+h(f(t_n, y_n)))]$ is of order 2.

Initial Stepsize

Frequently, users do not know how to give a suitable initial stepsize for an ODE solver. That is one reason why ODE solvers should support a facility to generate an initial stepsize. Gladwell et al. [48], Hairer et al. [57], and Shampine [105] have discussed algorithms to generate the initial stepsize for ODE solvers. The idea is based on the hypothesis that the local error (from the expansion of Taylor series) of a method of order p is approximately $Ch^{p+1}y^{(p+1)}(t_0)$. The initial stepsize will be calculated if users do not give the initial stepsize for the ODE computation, but are willing to give a tolerance for the numerical results. Let sc be a vector that represents the desired tolerance for the numerical results, where $sc_i = Atol_i + \max\{|y_{0i}|, |y_{1i}|\}Rtol_i$ is the i th element of sc . Here, $Rtol_i$ is the relative error for the i th element, and $Atol_i$ is the absolute error for the i th

element. Since calculating $y^{(p+1)}(t_0)$ is not easy and there is no guarantee that $y^{(p+1)}(t_0)$ is not close to zero, in order to have the local error not exceed the desired tolerance, the initial stepsize should be computed using the following algorithm [57] :

Calculate $sc_i = A_{tol_i} + |y_{0i}| (R_{tol_i})$

$$\text{Calculate } d_0 = \|y_0\| = \sqrt{\frac{1}{m} \sum_{i=1}^m \left(\frac{y_{0,i}}{sc_i}\right)^2}.$$

$$\text{Calculate } d_1 = \|f(t_0, y_0)\| = \sqrt{\frac{1}{m} \sum_{i=1}^m \left(\frac{f_{0,i}}{sc_i}\right)^2}.$$

If $d_0 \leq 10^{-5}$ or $d_1 \leq 10^{-5}$ then $h_0 = 10^{-6}$ else $h_0 = 0.01(d_0/d_1)$.

Take $h_0 = \min\{|t_f - t_0|, h_0\}$.

Perform explicit Euler, $y_1 = y_0 + h_0 f(t_0, y_0)$.

Calculate $f_1 = f(t_0 + h_0, y_1)$.

$$\text{Estimate the second derivative, } d_2 = \|f_1 - f_0\|/h_0 = \frac{1}{h_0} \sqrt{\frac{1}{m} \sum_{i=1}^m \left(\frac{f_{1,i} - f_{0,i}}{sc_i}\right)^2}.$$

Calculate $d_{\max} = \max\{d_1, d_2\}$.

$$\text{If } d_{\max} \leq 10^{-15} \text{ then } h_0 = \max\{10^{-6}, 10^{-3} h_0\} \text{ else } h_0 = \left(\frac{0.01}{d_{\max}}\right)^{\frac{1}{p+1}}.$$

Usually, an initial stepsize that is computed using the algorithm given above can give a good guess for the initial stepsize, but it takes an additional cost as shown above. If users can give initial stepsize that is obtained from computational experience, it is suggested that they give an initial stepsize to ODE solvers.

Stepsize Strategies

There are five steps related to the process for choosing an adaptive stepsize: (i) determine an error tolerance sc for the numerical results, (ii) determine an appropriate value for h_n , (iii) calculate a numerical approach y_{n+1} , (iv) probe the quality of y_{n+1} with respect to the local solution $u(t_{n+1})$ of $u' = f(t,u)$, $u(t_n) = y_n$, and (v) improve the numerical results y_{n+1} which has a local error l_{n+1} is greater than sc . A PI stepsize control has been proposed by Gustafsson et al. [55, 57] and has been implemented by Hairer et al. [57] in the DOPRI5 solver.

In general, the approach for controlling the stepsize is based on the hypothesis that the local error l_{n+1} of a p th-order method at t_n can be approximated as

$$l_{n+1} = u(t_n + h_n) - y_{n+1} = h_n^{p+1} \psi(t_n, y_n),$$

where $u(t)$ is an exact solution of $u' = f(t,u)$, $u(t_n) = y_n$. The Problem now is that the exact solution is unknown so that l_{n+1} is not known. This difficulty can be handled by finding an approximate local error ϵ_{n+1} for l_{n+1} . Among methods that can be used to find ϵ_{n+1} are:

1. Merson's method [71] using Richardson's extrapolation [26, 33, 57, 66, 78]. This is done by executing the method using two stepsizes $h/2$ and h . Lets y^* be the result for the stepsize $h/2$, and y^{**} be the result for the stepsize h . We can have $\epsilon_{n+1} = l_{n+1} + O(h^{p+2}) = (y^{**} - y^*) / (2^p - 1)$, where p is the order of the method and
2. Embedded Runge-Kutta methods as developed by Fehlberg [71, 112]. This is done by executing the method using two methods of order p and $p+1$. If y^* is the result from the method of order p and y^{**} is the result from the method of order $p+1$, then

$$\epsilon_{n+1} = l_{n+1} + O(h^{p+2}) = (y^{**} - y^*).$$

One example of determining the control stepsize is using crude Euler and implicit Euler [30]. If the local truncation error of a p th-order method is $l_{n+1} = \psi(t_n, y_n) h_n^{p+1}$, and we wish the error to equal sc , then for the stepsize h^* we have the relation

$$sc = \psi(t_n, y_n)(h^*)^{p+1}. \text{ By eliminating } \psi, \text{ we have a relation } h^* = h_n \left(\frac{sc}{l_{n+1}} \right)^{\frac{1}{p+1}}.$$

If for stepsize h_n the problem is solved using crude Euler, we have the numerical result $u_{n+1} = u_n + h_n f(t_n, u_n)$, with local error $u_{n+1} - y(t_{n+1}) = -h_n^2 f'_n / 2 + O(h_n^3)$. If for stepsize h_n the problem is solved using implicit Euler, we have the numerical result $w_{n+1} = w_n + h_n f(t_{n+1}, w_{n+1})$, with local error $y(t_{n+1}) - w_{n+1} = h_n^2 f'_n / 2 + O(h_n^3)$. The estimated error ϵ_{n+1} can be taken as $\epsilon_{n+1} \approx (u_{n+1} - w_{n+1})/2$. In order to avoid a small error,

$$h^* \text{ can be taken as } h^* = \mu h_n \left(\frac{sc}{\epsilon_{n+1}} \right)^{\frac{1}{p+1}}, \text{ where } 0 < \mu < 1 \text{ (}\mu \text{ can be taken equal to 0.9).}$$

Birta et al. [9] proposed several algorithms for finding the stepsizes. All algorithms will be based on the assumption that $R_{n+1} = \max_j \{ |r_{n+1}|_j \}$, where $r_{n+1} = \epsilon_{n+1}/rc$ and for m -vectors a and b , a/b is the vector whose j th component is a_j/b_j . The following algorithms will be named S1, S2, and S3.

Algorithm S1 :

Determine μ , and let $R_T = 1$.

Determine ϵ_n , and compute sc_n , where $sc_{ni} = Atol_i + \max \{ |y_{(n-1)i}|, |y_{(n)i}| \} (Rtol_i)$.

Compute $g_n = \|\epsilon_n/sc_n\|$, $R_n = h_n^{p+1} g_n$, and $h_{n+1} = \left(\frac{\mu}{R_n} \right)^{\frac{1}{p+1}} h_n$.

Compute $R_{n+1} = h_{n+1}^{p+1} g_n$.

If $R_{n+1} > R_T$ then $h_{n+1} = \left(\frac{\mu}{R_{n+1}}\right)^{\frac{1}{p+1}} h_{n+1}$, and check R_{n+1} again until $R_{n+1} \leq R_T$.

Algorithm S2 :

Determine λ , and $\mu = \min\{0.9, -0.1 \log \lambda\}$.

Determine ε_n , and compute sc_n , where $sc_{ni} = Atol_i + \max\{|y_{(n-1)i}|, |y_{(n)i}|\}(Rtol_i)$.

Compute R_n , and $h_{n+1} = \left(\frac{\mu R_{n-1}}{R_n^2}\right)^{\frac{1}{p+1}} \left(\frac{h_n}{h_{n-1}}\right) h_n$.

Algorithm S3 :

Let $\mu = 0.5$, $A_0 = 1$, and let $R_T = 2$.

Determine ε_n , and compute sc_n , where $sc_{ni} = Atol_i + \max\{|y_{(n-1)i}|, |y_{(n)i}|\}(Rtol_i)$.

Compute $g_n = \|\varepsilon_n/sc_n\|$, $R_n = h_n^{p+1} g_n$.

Compute $A_n = \left(0.4 + 0.6 \min\left\{2, \frac{1}{R_n}\right\}\right) A_{n-1}$, $n > 0$

Compute $\alpha_n = 0.5(1 + A_n) \left(\frac{\mu}{R_n}\right)^{\frac{1}{p+1}}$.

Calculate $h_{n+1} = \alpha_n h_n$, and $R_{n+1} = h_{n+1}^{p+1} g_n$.

If $R_{n+1} > R_T$ then

Compute $A_n = 0.5 \left(1 + \frac{1}{R_{n+1}}\right) A_n$.

Compute $\alpha_{n+1} = \min\{1, 0.5(1 + A_n)\} \left(\frac{\mu}{R_{n+1}}\right)^{\frac{1}{p+1}}$.

Calculate $h_{n+1} = \alpha_{n+1} h_n$.

Calculate R_{n+1} and check until $R_{n+1} \leq R_T$.

endif

Birta et al. [17] also claimed that the algorithm S3 has superior performance, and demonstrated this in their several numerical experiments. Wille [117, 118] introduced procedures for finding a new stepsize for linear multistep methods and for Adams linear multistep methods. Hall [58] in his report introduced a procedure for finding a new stepsize for Runge-Kutta codes.

Optimal Order Related to Step Size Sequences

In modern standard methods for solving ODEs, the strategy is based on the possibility of computing the solution over the whole range of integration using variable methods and variable step sizes. A classic theory of constant step size and constant formula methods (CSCFM) has been successfully implemented in several fields of ODE problems. Nowadays, the challenge is to solve ODE problems using a theory of variable step size and variable formula methods (VSVFM). Some ideas for VSVFM have been introduced by Shampine and Gordon [99], Kockler [71], Zlatev [120], Butcher [14], Shampine [105], Hairer et. al [56, 57], and Zhang [119].

Butcher [14, 15] proposed an algorithm for selecting the optimal order and step size from a collection of numerical methods with different orders as follows :

$$h_{\max} = t_f - t$$

$$r_0 = h_{\max}/h_n$$

$$u = 0$$

```

for i ∈ ℑ do
  if (i)e[i] r0i+1 < (sc)w[i] then
    r = r0
  else
    r =  $\left( \frac{(sc)w[i]}{(i)e[i]} \right)^{\frac{1}{i+1}}$ 
  endif
  if (i)(r) > (i+1)w[i]u then
    u =  $\frac{(i)(r)}{(i+1)w[i]}$ 
    p = i
    hn+1 = (r)hn
  endif
enddo.

```

In this algorithm, \mathfrak{I} is a set of available orders of numerical methods, i is the order of a method, r is a factor for the next stepsize, sc is a user-specified tolerance, $w[i]$ is the relative cost using the order i , h_n is the current stepsize, p is the chosen order, h_{n+1} is the chosen stepsize, and u is an auxiliary variable. The relative cost $w[i]$ can be the cpu time needed to calculate one step for a method of order i . This can be done from the first computation with stepsize h_0 for all method of order i 's in \mathfrak{I} by investigating the cpu time for each order.

APPENDIX E

STABILITY REGION OF SIMPLE PREDICTOR-CORRECTOR METHODS

The test problem $y' = f(t,y) = \lambda y$, has been successfully used by Dahlquist in analyzing the stability region of numerical methods of ODEs. By substituting this test problem into the Euler method $y_{n+1} = y_n + hf(t_n, y_n)$, we have $y_{n+1} = y_n + \lambda h y_n = (1+\lambda h)y_n$. Since the latter can be simplified into $y_n = (1+\lambda h)^n y_0$, it can be said that the Euler method will absolutely be stable if $|1+\lambda h| \leq 1$. For λh real, the interval of absolute stability is $-2 \leq \lambda h \leq 0$. In case λh complex, the region of absolute stability is the region inside a circle with center $(-1,0)$ and radius 1. By applying the same procedure to the implicit Euler method $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$, we have $y_{n+1} = y_n + \lambda h y_{n+1}$. From grouping the same variables, we have $y_{n+1} = (1-\lambda h)^{-1} y_n$, and then $y_n = (1-\lambda h)^{-n} y_0$. The implicit Euler method will be absolutely stable if $|1-\lambda h| \geq 1$. For λh real, the interval of absolute stability is $\{\lambda h \mid \lambda h \geq 2 \text{ or } \lambda h \leq 0\}$. For the trapezoidal method $y_{n+1} = y_n + h/2[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$, we get $y_{n+1} = y_n + h/2[\lambda y_n + \lambda y_{n+1}]$. By grouping the same variables, we have $y_{n+1} = [(1+\lambda h/2)/(1-\lambda h/2)]y_n$, and $y_n = [(1+\lambda h/2)/(1-\lambda h/2)]^n y_0$. The trapezoidal method is said to be absolutely stable if $|(1+\lambda h/2)/(1-\lambda h/2)| \leq 1$. In case λh is real, the region stability of the trapezoidal method is an interval $\lambda h < 0$. For the case of λh complex, the region of absolute stability is the set of complex z so that the

distance between z and the point $z=-2$ is less than or equal to that of the distance between z and the point $z=2$. This region is none other than half-plane of $\text{Re}(\lambda h) \leq 0$. Let us examine the case when the trapezoidal method is used as a corrector and the crude Euler method is used as a predictor. This predictor-corrector method is also called a PC method. The stability of this PC method will depend on both the predictor and the corrector. The stability will not be dominated by the corrector alone, but the predictor will also contribute to the stability of this PC method. Let's analyze what will happen if we iterate the corrector up to m times. The situation is as follows:

$$\begin{aligned}
 y_{n+1}^{[0]} &= y_n + hf_n = y_n + \lambda h y_n = (1 + \lambda h) y_n && \text{(corrector)} \\
 y_{n+1}^{[1]} &= y_n + (h/2)(f_n + f_{n+1}^{[0]}) = y_n + (h/2)[\lambda y_n + \lambda(1 + \lambda h)y_n] \\
 &= [1 + \lambda h + (\lambda h)^2/2] y_n && \text{(1st execution of corrector)} \\
 y_{n+1}^{[2]} &= y_n + (h/2)(f_n + f_{n+1}^{[1]}) = y_n + (h/2)[\lambda y_n + \lambda\{ [1 + \lambda h + (\lambda h)^2/2] y_n \}] \\
 &= [1 + \lambda h + (\lambda h)^2/2 + (\lambda h)^3/2^2] y_n && \text{(2nd execution of corrector)} \\
 &\cdot && \cdot \\
 &\cdot && \cdot \\
 y_{n+1}^{[m]} &= y_n + (h/2)(f_n + f_{n+1}^{[m-1]}) = E(\lambda h) y_n && \text{(mth execution of corrector)}
 \end{aligned}$$

where $E(\lambda h) = 1 + \lambda h + (\lambda h)^2/2 + (\lambda h)^3/2^2 + \dots + (\lambda h)^{m+1}/2^m$. In order to have an absolutely stable computation, first, the $E(\lambda h)$ should converge as m goes to infinity. This is only possible for $|\lambda h/2| < 1$ or $|\lambda h| < 2$. The region of stability here is a region inside a circle with center 0 and radius 2. By calculating the series for $E(\lambda h)$, we have

$$E(\lambda h) = 1 + \lambda h \left[\frac{1 - (\lambda h/2)^{m+1}}{1 - \lambda h/2} \right] = \frac{1 + \lambda h/2 - 2(\lambda h/2)^{m+2}}{1 - \lambda h/2}$$

The second condition is $|E(\lambda h)| \leq 1$. For $m \rightarrow \infty$, $E(\lambda h) = [1 + \lambda h/2]/[1 - \lambda h/2]$. By considering the two conditions together, for λh complex, we have that the region of

stability is $\{ \lambda h \mid \text{Re}(\lambda h) \leq 0 \text{ and } |\lambda h| < 2 \}$, which is the left part of a region inside a circle with center 0 and radius 2. For λh real, the interval of stability is $-2 < \lambda h < 0$. It is clear then that for the case $y' = -50y$, the timestep should be less than $2/50=0.04$ in order to have a stable computation. The region of stability of a PC method where the trapezoidal rule is used as a corrector and crude Euler is used as a predictor, is shown in Figure E-1.

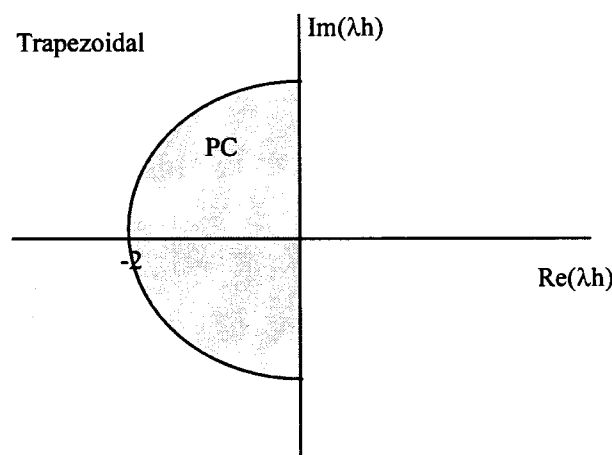


Figure E-1 A PC-Method Where a Trapezoidal as a Corrector and an Euler as a Predictor

Let us now investigate the case when the implicit Euler method is used as a corrector and the crude Euler method is used as a predictor. Let us analyze what will happen if we iterate the corrector up to m times. The situation is as follows:

$$\begin{aligned}
 y_{n+1}^{[0]} &= y_n + hf_n = y_n + \lambda h y_n = (1 + \lambda h) y_n && \text{(corrector)} \\
 y_{n+1}^{[1]} &= y_n + hf_{n+1}^{[0]} = y_n + h\lambda(1 + \lambda h)y_n \\
 &= [1 + \lambda h + (\lambda h)^2] y_n && \text{(1st execution of corrector)} \\
 y_{n+1}^{[2]} &= y_n + hf_{n+1}^{[1]} = y_n + h\lambda \{ [1 + \lambda h + (\lambda h)^2] y_n \} \\
 &= [1 + \lambda h + (\lambda h)^2 + (\lambda h)^3] y_n && \text{(2nd execution of corrector)}
 \end{aligned}$$

$$y_{n+1}^{[m]} = y_n + hf_{n+1}^{[m-1]} = E(\lambda h)y_n \quad (m^{\text{th}} \text{ execution of corrector})$$

where $E(\lambda h) = 1 + \lambda h + (\lambda h)^2 + (\lambda h)^3 + \dots + (\lambda h)^{m+1}$. In order to have absolutely stable computation, first, $E(\lambda h)$ should converge as m goes to infinity. This is only possible for $|\lambda h| < 1$. The region of stability here is a region inside a circle with center 0 and radius 1.

By calculating the series of $E(\lambda h)$, we have

$$E(\lambda h) = \left[\frac{1 - (\lambda h)^{m+2}}{1 - \lambda h} \right]$$

The second condition is $|E(\lambda h)| \leq 1$. For $m \rightarrow \infty$, $E(\lambda h) = [1 - \lambda h]^{-1}$. In the case where λh is real, from the two conditions, the region of stability is an interval $-1 < \lambda h < 0$. It is clear then that for the case $y' = -50y$, the timestep should be less than $1/50 = 0.02$ in order to have a stable computation. For λh complex, we have that the region of stability is $\{ \lambda h \mid |1 - \lambda h| \geq 1 \text{ and } |\lambda h| < 1 \}$. A graphical representation of this region of stability is given in Figure E-2.

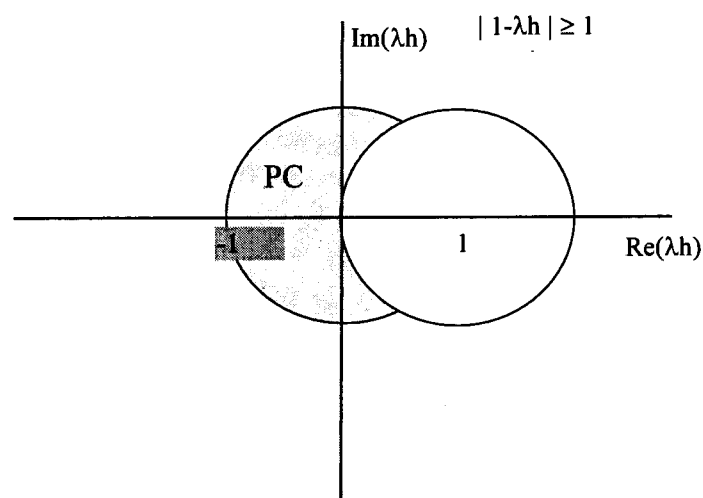


Figure E-2 A PC-Method Where the Implicit Euler Method as a Corrector and Crude Euler as a Predictor

APPENDIX F

COLLECTION OF PROGRAMS

Mathematica 1. Kidney Problems

```
(*Loading Graphics and MultiplePlot packages *)
<<Graphics`Graphics`
<<Graphics`MultipleListPlot`
(* Create module for solving the problems *)
Clear[sols]
sols[l_ ,title_]:=Module[{dsys,y1,y2,y3,y4,y5,
  y1out,y2out,y3out,y4out,y5out,t},
  endtime=1;
  a=100;
  b=0.9;
  c=1000;
  d=10;
  tmin=0;
  tmax=1;
  dt=0.1;
  eqone=y1'[t]==a*y1[t]*(y3[t]-y1[t])/y2[t];
  eqtwo=y2'[t]==-a*(y3[t]-y1[t]);
  eqthree=y3'[t]==(b-c*(y3[t]-y5[t])-a*y3[t]*(y3[t]-y1[t]))/y4[t];
  eqfour=y4'[t]==a*(y3[t]-y1[t]);
  eqfive=y5'[t]==-c*(y5[t]-y3[t])/d;
  dsys=NDSolve[{eqone,eqtwo,eqthree,eqfour,eqfive,
    y1[0]==1,y2[0]==1,y3[0]==1,y4[0]==-10,y5[0]==1},
    {y1[t],y2[t],y3[t],y4[t],y5[t]},{t,0,endtime},MaxSteps->5000];
  y1out=Table[{t,y1[t]/.dsys[[1]]},{t,tmin,tmax,dt}];
  y2out=Table[{t,y2[t]/.dsys[[1]]},{t,tmin,tmax,dt}];
  y3out=Table[{t,y3[t]/.dsys[[1]]},{t,tmin,tmax,dt}];
  y4out=Table[{t,y4[t]/.dsys[[1]]},{t,tmin,tmax,dt}];
  y5out=Table[{t,y5[t]/.dsys[[1]]},{t,tmin,tmax,dt}];
  MultipleListPlot[y1out,y2out,y3out,y4out,y5out,
    PlotRange->All,AxesOrigin->{0.,0.},AxesLabel->{"t","y"},
    PlotLabel->title,PlotJoined->True]]
(* Plot the outputs *)
```

```

graph1=sols[0.9902688359,"l=0.9902688359"];
graph2=sols[0.990283499,"l=0.990283499"];
graph3=sols[0.9925211341,"l=0.9925211341"];
graph4=sols[1.0304879856,"l=1.0304879856"];
graph5=sols[0.99,"l=0.99"];
graph6=sols[0.9,"l=0.9"];
graph7=sols[0.,"l=0."];
Show[GraphicsArray[{{graph1,graph2},{graph3,graph4},
                    {graph5,graph6,graph7}}]]

```

Mathematica 2. Autocatalytic Reaction Pathway

```

(*Loading Graphics, MultiplePlot, Legend packages *)
<<Graphics`Graphics`
<<Graphics`MultipleListPlot`
<<Graphics`Legend`
(* Create symbols for legend *)
NewMakeSymbol[{a_,Line[x_]}]:=
Module[{yugh,y}, y=Line[(Scaled[#1,yugh] & ) /@ x];
        y=y /. yugh->#1;Evaluate[{a,y}]&]
NewMakeSymbol[Line[x_]:=
NewMakeSymbol[{AbsoluteThickness[0.1],Line[x]}]
diamond=RegularPolygon[4,0.02];
triangle=RegularPolygon[3,0.02];
star=RegularPolygon[5,0.02,{0,0},0,2];
snowflake=Line[{{0,0},{0,0.025},{0,0},
                {Sqrt[3]/80,0.0125},{0,0},
                {Sqrt[3]/80,-0.0125},{0,0},{0,-0.025},
                {0,0},{-Sqrt[3]/80,-0.0125},{0,0},
                {-Sqrt[3]/80,0.0125},{0,0}}];
square=Line[{{-0.0125,-0.0125},{0.0125,-0.0125},
             {0.0125,0.0125},{-0.0125,0.0125},
             {-0.0125,-0.0125}}];
(* end of creating symbols *)
(* Module for selecting outputs to be plotted *)
BeginPackage["OutPlot`"];
OutPlot::usage="OutPlot[f]
                selectively choose the output to be plotted.";
Begin["`Private`"];
OutPlot[f_,t0_,tf_]:=
Module[{},
        l={};
        For[x=t0,x<=tf,x=x 10,
            l=Append[l,N[{Log[10,x],f[x]}]]];

```

```

Return[l];
];
End[];
EndPackage[];
(* end of module *)
(* Compute the ODE problem numerically *)
endtime=4 10^10;
Clear[y1,y2,y3,t]
dsys=NDSolve[
  {y1'[t]==-0.04 y1[t]+10^4 y2[t] y3[t],
   y2'[t]==0.04 y1[t]-10^4 y2[t] y3[t]-
     3 10^7 y2[t] y2[t],
   y3'[t]==3 10^7 y2[t] y2[t],
   y1[0]==1,y2[0]==0,y3[0]==0},
  {y1[t],y2[t],y3[t]},{t,0,endtime}];
y1[t_]=y1[t]/.dsys[[1]];
y2[t_]=y2[t]/.dsys[[1]];
y3[t_]=y3[t]/.dsys[[1]];
(* end of computation *)
(* Selecting output to be plotted *)
y1out=OutPlot[y1,10,4 10^10];
y2out=OutPlot[y2,10,4 10^10];
y3out=OutPlot[y3,10,4 10^10];
(* Plot the output *)
ShowLegend[
  MultipleListPlot[y1dat,y2dat,y3dat,
    DotShapes->{NewMakeSymbol[snowflake],
      NewMakeSymbol[diamond],
      NewMakeSymbol[star]},
    PlotJoined->True,
    AxesLabel->{"t","y"},
    DisplayFunction->Identity],
  {{{Graphics[snowflake],"y1"},
   {Graphics[diamond],"y2"},
   {Graphics[star],"y3"}},
  LegendSize->{0.5,0.5},
  LegendShadow->{0,0},
  LegendBorder->None,
  LegendPosition->{0.3,-0.2}}]

```

Mathematica 3. Problem D4 of Enright et al.

(*Loading Graphics and Legend packages *)

```

<<Graphics`Graphics`
<<Graphics`Legend`
(* Define the ODE problem to be solved *)
endtime=50;
Clear[y1,y2,y3,eqone,eqtwo,eqthree,t];
eqone=y1'[t]==-0.013 y1[t] - 10^3 y1[t] y3[t];
eqtwo=y2'[t]==-2500 y2[t] y3[t];
eqthree=y3'[t]==0.013 y1[t] - 10^3 y1[t] y3[t] -
      2500 y2[t] y3[t];
(* Compute the ODE problem numerically *)
dsys=NDSolve[{eqone,eqtwo,eqthree, y1[0]==1,y2[0]==1,y3[0]==0},
      {y1[t],y2[t],y3[t]},{t,0,endtime}];
y1[t_]=y1[t]/.dsys[[1]];
y2[t_]=y2[t]/.dsys[[1]];
y3[t_]=y3[t]/.dsys[[1]];
(* Plot the results *)
y1out=Plot[y1[t],{t,0,endtime},PlotRange->All,AxesLabel->{"t","y1"},
      AxesOrigin->{0,y1[endtime]-0.02},Ticks->{{0,25,50},{0,0.8,1}},
      PlotStyle->{AbsoluteThickness[1],AbsoluteDashing[{5,5}]},
      PlotPoints->15];
y2out=Plot[y2[t],{t,0,endtime},PlotRange->All,AxesLabel->{"t","y2"},
      AxesOrigin->{0,y2[endtime]-0.01},Ticks->{{0,25,50},{0,0.9,1}},
      PlotStyle->{AbsoluteThickness[1.2],AbsoluteDashing[{2,2}]},
      PlotPoints->15];
y3out=Plot[y3[t],{t,0,endtime},PlotRange->All,AxesLabel->{"t","y3"},
      Ticks->{{0,25,50},{0,3.5 10^-6}},PlotStyle->{AbsoluteThickness[2]},
      PlotPoints->15];
yall=Plot[{y1[t],y2[t],y3[t]},{t,0,endtime},PlotRange->{Automatic,{0,1}},
      PlotStyle->{{AbsoluteThickness[1],AbsoluteDashing[{5,5}]},
      {AbsoluteThickness[1.2],AbsoluteDashing[{2,2]}}},
      AbsoluteThickness[2]},
      LegendSize->{0.3,0.3},LegendShadow->None,LegendBorder->{0},
      LegendPosition->{-0.2,-0.3},AxesLabel->{"t","y"},
      PlotLegend->{"y1","y2","y3"}];
Show[GraphicsArray[{{y1out,y2out},{y3out,yall}}]]

```

Mathematica 4. Problem Proposed by Gupta and Wallace

```

(*Loading Graphics and Legend packages *)
<<Graphics`Graphics`
<<Graphics`Legend`
(* Define the ODE problem to be solved *)
endtime=10;
Clear[y1,y2,eqone,eqtwo,t];

```

```

v=-80;
w=8;
eqone=y1'[t]==v y1[t] - w y2[t] +(-v+w+1) Exp[t];
eqtwo=y2'[t]==w y1[t] + v y2[t] +(-v-w+1) Exp[t];
(* Solve the ODE problem Numerically *)
dsys=NDSolve[{eqone,eqtwo,y1[0]==1,y2[0]==1},{y1[t],y2[t]},
             {t,0,endtime},MaxSteps->2000];
y1[t_]=y1[t]/.dsys[[1]];
y2[t_]=y2[t]/.dsys[[1]];
y1out=Plot[y1[t],{t,0,endtime},PlotRange->All,AxesLabel->{"t","y1"},
           AxesOrigin->{0,0},Ticks->{{0,5,10},Automatic},
           PlotStyle->{AbsoluteThickness[1],AbsoluteDashing[{5,5}]},
           PlotPoints->15];
y2out=Plot[y2[t],{t,0,endtime},PlotRange->All,AxesLabel->{"t","y2"},
           Ticks->{{0,5,10},Automatic},PlotStyle->{AbsoluteThickness[2]},
           PlotPoints->15];
yall=Plot[{y1[t],y2[t]},{t,0,endtime},PlotRange->All,
          PlotStyle->{{AbsoluteThickness[1],AbsoluteDashing[{5,5]}},
                     AbsoluteThickness[2]},LegendSize->{0.3,0.3},LegendShadow->None,
          LegendBorder->{0},LegendPosition->{0,-0.3},AxesLabel->{"t","y"},
          PlotLegend->{"y1","y2"}];
Show[GraphicsArray[{{y1out,y2out},{yall}}]]

```

Mathematica 5. Stability Region of Explicit Runge-Kutta

```

(* Loading FilledPlot package *)
<<Graphics`FilledPlot`;
StabilityRungeKutta::usage="StabilityRungeKutta[n,name]
    Plot stability region of Runge-Kutta";
(* Created by Edward Purba
    Computer-Science, Oklahoma State University, 1996 *)
StabilityRungeKutta[n_]:=Module[{coeff,theta,z0,k,t,l,i,fz,dfz,z,nangle,sdt,tbl},
  If[n<1,
    (* then *)
    Print["The order should be >=1\n"],
    (* else *)
    If[n==1,
      (* then *)
      FilledPlot[{Sqrt[1-(x+1)^2],-Sqrt[1-(x+1)^2]},
                 {x,-2,0},AspectRatio->1,Fills->GrayLevel[0.95],
                 PlotStyle->Dashing[{0.01}],
                 PlotLabel->StringJoin["Order-",ToString[n]]],
      (* else *)
      coeff=Table[1/i!,{i,0,n,1}];

```



```

PROGRAM RUNMEB
IMPLICIT NONE

C
C THE REAL WORK ARRAY NEEDS TO BE AT LEAST 2*N*N + 37*N WORDS LONG
C THE INTEGER WORK ARRAY NEEDS TO BE AT LEAST N WORDS LONG
C
REAL *8   Y(5), WORK(235), HUSED
INTEGER   IWORK(5), NQUSED, NSTEP, NFE, NJE, NDEC, NBSOL
INTEGER   NPSET, NCOSET, MAXORD, LOUT, LWORK, LIWORK, N
EXTERNAL  FCN
EXTERNAL  FEX, JEX, CLOCK

COMMON/MEBDF2/HUSED,NQUSED,NSTEP,NFE,NJE,NDEC,NBSOL,NPSET,
+          NCOSET,MAXORD

C
C THE INCLUSION OF THE COMMON BLOCK /MEBDF2/ ALLOWS THE USER TO
C ACCESS VARIOUS COUNTERS THAT MAY BE OF INTEREST TO HIM.
C THESE VARIABLES ARE EXPLAINED IN THE COMMENTS IN SUBROUTINE
C MEBDF.
C THE USER NEEDS TO SET THE UNIT ROUND-OFF ERROR FOR THE MACHINE
C BEING USED IN THE BLOCK DATA STATEMENT FOLLOWING SUBROUTINE
C MEBDF.
C
REAL *8   ALPHA, DELTA, T0, T, TEND, TOUT, HSTART, H0, TOL
REAL *4   DTIME, ETIME, TOTAL, TIMES(2)
INTEGER   INDEX
DATA      LOUT / 6 /, LWORK/235/, LIWORK/5/, N/5/
DATA      T0/0.D0/, TEND/1.D0/, DELTA/0.1D0/
DATA      Y(1),Y(2),Y(3),Y(4)/1.D0,1.D0,1.D0,-10.D0/
DATA      INDEX/1/, HSTART/1.D-6/, TOL/1.D-6/
DATA      TIMES/0.0,0.0/

READ(*,*)ALPHA
Y(5)=ALPHA
T = T0
H0 = HSTART
PRINT*,'ALPHA =', ALPHA

C
C THESE ARE THE INITIAL STEPS CHOSEN BY THE DETEST PACKAGE
C FOR THIS PROBLEM
C
TOUT=T0+DELTA
TOTAL=DTIME(TIMES)
10  CONTINUE
IF(TOUT.LE.TEND) THEN
20  CONTINUE
CALL MEBDF(N, T, H0, Y, TOUT, TEND, TOL, 21, INDEX, LOUT,
+          LWORK, WORK, LIWORK, IWORK, FCN)
IF ((INDEX.NE.0).AND.(INDEX.NE.3)) THEN
IF(INDEX.EQ.1) THEN
INDEX=0
GO TO 20
ENDIF
WRITE( 6 ,15) INDEX

```

```

                STOP
            END IF
C
C HAVE COMPLETED ONE STEP
C
C
C HAVE WE FINISHED YET?
C
                IF( INDEX.EQ.0) THEN
                    WRITE(6,250)T,NSTEP,NFE,NJE,NQUSED,HUSED,Y(1),
*                      Y(2),Y(3),Y(4),Y(5)
C
C THEN WE HAVE EFFECTIVELY HIT TOUT
C
                T=TOUT
                ELSE
                    INDEX=3
                    GOTO 20
                END IF
                TOUT=TOUT+DELTA
                GOTO 10
            ENDIF
            TOTAL=DTIME(TIMES)
            WRITE(6, 690) TOL
690          FORMAT(1X, 'REQUESTED TOLERANCE',5X, 1PE10.3)
            PRINT*,' User      Sys      Total'
            PRINT*, TIMES(1),TIMES(2), TOTAL
C
C THESE VERY SMALL CONSTANTS SHOULD BE SET TO ZERO IF THERE
C ARE LIKELY TO BE DIFFICULTIES DUE TO UNDERFLOW.
C
250          FORMAT(1X,D11.6,I4,I4,I4,I3,D14.6,5(1X,D12.6))
15          FORMAT(' ***INTEGRATION HAS FAILED*** WITH INDEX=',I3)
            STOP
            END

            SUBROUTINE FCN(T,Y,YDOT)
            IMPLICIT DOUBLE PRECISION(A-H,O-Z)
            DIMENSION Y(5),YDOT(5)
            A=100.D0
            B=0.9D0
            C=1000.D0
            D=10.D0
            YDOT(1)= A*Y(1)*(Y(3)-Y(1))/Y(2)
            YDOT(2)=-A*(Y(3)-Y(1))
            YDOT(3)=(B-C*(Y(3)-Y(5))-A*Y(3)*(Y(3)-Y(1)))/Y(4)
            YDOT(4)=A*(Y(3)-Y(1))
            YDOT(5)=-C*(Y(5)-Y(3))/D
            RETURN
            END

            SUBROUTINE PDERV(T,Y,PW)
            IMPLICIT DOUBLE PRECISION(A-H,O-Z)

```

```

DIMENSION Y(5),PW(5,5)
A=100.D0
B=0.9D0
C=1000.D0
D=10.D0
PW(1,1)= (-A*Y(1) + A*(Y(3)-Y(1)))/Y(2)
PW(1,2)= -A*Y(1)*(Y(3)-Y(1))/(Y(2)*Y(2))
PW(1,3)=A*Y(1)/Y(2)
PW(1,4)=0.D0
PW(1,5)=0.D0
PW(2,1)=A
PW(2,2)=0.D0
PW(2,3)=-A
PW(2,4)=0.d0
PW(2,5)=0.d0
PW(3,1)=A*Y(3)/Y(4)
PW(3,2)=0.D0
PW(3,3)=(-C-A*Y(3)-A*(Y(3)-Y(1)))/Y(4)
PW(3,4)=- (B-A*Y(3)*(Y(3)-Y(1))-C*(Y(3)-Y(5)))/(Y(4)*Y(4))
PW(3,5)=C/Y(4)
PW(4,1)=-A
PW(4,2)=0.D0
PW(4,3)=A
PW(4,4)=0.D0
PW(4,5)=0.D0
PW(5,1)=0.D0
PW(5,2)=0.D0
PW(5,3)=C/D
PW(5,4)=0.D0
PW(5,5)=-C/D
RETURN
END

```

Program 2. Main Program for VODE with the Case of Autocatalytic Reaction Problems.

```

PROGRAM RUNVODE
IMPLICIT NONE
*-----
* RWORK = Real work array of length at least..
*   22 + 9*NEQ + 2*NEQ**2      for MF=21 or 22,
* IWORK = Integer work array of length at least ..
*   30 + NEQ                    for MF=21, 22, 24, or 25
* LRW  = Declared length of RWORK (in User's DIMENSION statement)
* LIW  = Declared length of IWORK (in user's DIMENSION statement)
*-----
REAL *8    ATOL(3), RPAR, RTOL, RWORK(67), T, TOUT, Y(3)
REAL *8    DELTA, T0, TEND, HU
INTEGER    IWORK(33), NEQ, ITOL, ISTATE, ITASK, IOPT, LRW, LIW, MF
INTEGER    NCFN, NETF, NFE, NJE, NLU, NNI, NQU, NST, IPAR
EXTERNAL   FEX, JEX, CLOCK
REAL *4    DTIME, ETIME, TOTAL, TIMES(2)

```

```

INTEGER      SYSTEM
COMMON      /DVOD02/HU, NCFN, NETF, NFE, NJE, NLU, NNI, NQU, NST

DATA        TIMES/0.0,0.0/, DELTA/0.4D0/, NEQ/3/, T0/0.D0/
DATA        Y/1.D0,0.D0,0.D0/
DATA        ATOL/1.D-6,1.D-6,1.D-6/, RTOL/1.D-6/
DATA        ITOL/2/, ITASK/1/, ISTATE/1/, IOPT/0/, LRW/67/, LIW/33/
DATA        MF/21/, TEND/4.D10/
T = T0
TOUT = T0 + DELTA
TOTAL = DTIME(TIMES)
10 CONTINUE
IF(TOUT. LE. TEND) THEN
    CALL DVODE(FEX,NEQ,Y,T,TOUT,ITOL,RTOL,ATOL,ITASK,ISTATE,
1          IOPT,RWORK,LRW,IWORK,LIW,JEX,MF,RPAR,IPAR)
    WRITE(6,250)DLOG10(T), NST, NFE, NJE, NQU, HU, Y(1),
1          Y(2), Y(3)
    TOUT = TOUT * 10.D0
    GOTO 10
ENDIF
TOTAL = DTIME(TIMES)
PRINT*
PRINT*,' user      sys      total'
PRINT*,' seconds  seconds  seconds'
250 PRINT*,TIMES(1),TIMES(2),TOTAL
FORMAT(D12.4,I4,I4,I3,I3,D12.4,3D14.6)
STOP
END

SUBROUTINE FEX (NEQ, T, Y, YDOT, RPAR, IPAR)
DOUBLE PRECISION RPAR, T, Y, YDOT
DIMENSION Y(NEQ), YDOT(NEQ)
YDOT(1) = -.04D0*Y(1) + 1.D4*Y(2)*Y(3)
YDOT(3) = 3.D7*Y(2)*Y(2)
YDOT(2) = -YDOT(1) - YDOT(3)
RETURN
END

SUBROUTINE JEX (NEQ, T, Y, ML, MU, PD, NRPD, RPAR, IPAR)
DOUBLE PRECISION PD, RPAR, T, Y
DIMENSION Y(NEQ), PD(NRPD,NEQ)
PD(1,1) = -.04D0
PD(1,2) = 1.D4*Y(3)
PD(1,3) = 1.D4*Y(2)
PD(2,1) = .04D0
PD(2,3) = -PD(1,3)
PD(3,2) = 6.D7*Y(2)
PD(3,3)=0.D0
PD(2,2) = -PD(1,2) - PD(3,2)
RETURN
END

```

Program 3. Main Program for LSODE with the Case of Problem D4 of Enright et al.

```

PROGRAM      RUNLSODE
IMPLICIT     NONE
EXTERNAL    FEX, JEX, CLOCK
REAL *8     ATOL(3), RPAR, RTOL, RWORK(58), T, TOUT, Y(3)
INTEGER     IWORK(33), NEQ, ITOL, IASTATE, IOPT, LRW, LIW, ITASK
REAL *8     DELTA, TEND, T0

REAL *4     DTIME, ETIME, TOTAL, TIMES(2)
INTEGER     SYSTEM, MF

DATA  NEQ/3/, TIMES/0.,0./, DELTA/5.D0/, IOPT/0/,ITASK/1/
DATA  Y/1.D0,1.D0,0.D0/, T0/0.D0/, TEND/50.D0/, ITOL/2/
DATA  ATOL/1.D-6,1.D-6,1.D-6/, IASTATE/1/, RTOL/1.D-6/
DATA  LRW, LIW/58, 23/, MF/21/
T = T0
TOUT = T0 + DELTA
TOTAL = DTIME(TIMES)
10  CONTINUE
    IF(TOUT.LE.TEND) THEN
        CALL LSODE(FEX,NEQ,Y,T,TOUT,ITOL,RTOL,ATOL,ITASK,IASTATE,
1          IOPT,RWORK,LRW,IWORK,LIW,JEX,MF)
        WRITE(6,20)T,IWORK(11),IWORK(12),IWORK(13),IWORK(14),
1          RWORK(11),Y(1),Y(2),Y(3)
        IF(IASTATE.LT.0) GOTO 80
        TOUT = TOUT + DELTA
        GOTO 10
    ENDIF
TOTAL = DTIME(TIMES)
PRINT*
PRINT*,' user      sys      total'
PRINT*,' seconds  seconds  seconds'
PRINT*,times(1),times(2),total
STOP
80  WRITE(6,90)IASTATE
90  FORMAT(/// Error halt.. ISTATE =',I3)
20  FORMAT(D12.4,2I4,2I3,D12.4,3D14.6)
STOP
END

SUBROUTINE FEX (NEQ, T, Y, YDOT, RPAR, IPAR)
DOUBLE PRECISION RPAR, T, Y, YDOT
DIMENSION Y(NEQ), YDOT(NEQ)
YDOT(1) = -.013*Y(1) - 1.D3*Y(1)*Y(3)
YDOT(2) = -2500D0*Y(2)*Y(3)
YDOT(3) = 0.013*Y(1) - 1.D3 * Y(1)*Y(3) - 2500.D0*Y(2)*Y(3)
RETURN
END

SUBROUTINE JEX (NEQ, T, Y, ML, MU, PD, NRPD, RPAR, IPAR)

```

```

DOUBLE PRECISION PD, RPAR, T, Y
DIMENSION Y(NEQ), PD(NRPD,NEQ)
PD(1,1) = -.013D0 - 1.D3*Y(3)
PD(1,2) = 0.d0
PD(1,3) = -1.D3*Y(1)
PD(2,1) = 0.D0
PD(2,2) = -2500.D0*Y(3)
PD(2,3) = -2500.D0*Y(2)
PD(3,1) = 0.013D0 - 1.D3*Y(3)
PD(3,2) = -2500.D0*Y(3)
PD(3,3) = -1.D3*Y(1) - 2500.D0*Y(2)
RETURN
END

```

Program 4. Main Program for EPSODE with the Case of Problem proposed by Gupta and Wallace.

```

PROGRAM RUNEPSODE
IMPLICIT NONE
REAL *8      Y0(2), HUSED, EPS, DELTA, H0, T0, TEND, TOUT
INTEGER      NQUSED, NSTEP, NFE, NJE, N, IERROR, MF, INDEX
COMMON      /EPCOM9/ HUSED, NQUSED, NSTEP, NFE, NJE
EXTERNAL    DIFFUN, JACOB, CLOCK
REAL *4      DTIME, ETIME, TOTAL, TIMES(2)
INTEGER      SYSTEM
DATA        N/2/, EPS/1.D-6/, DELTA/1.D0/, IERROR/1/, TEND/10.D0/
DATA        TIMES/0.,0./, MF/21/, T0/0.D0/, H0/1.D-6/
DATA        Y0/1.D0,1.D0/, INDEX/1/
TOUT = T0 + DELTA
TOTAL = DTIME(TIMES)
10 CONTINUE
IF(TOUT. LE. TEND) THEN
    CALL EPSODE(DIFFUN, JACOB, N, T0, H0, Y0, TOUT, EPS,
*          IERROR, MF, INDEX)
    WRITE(6,1001)T0,NSTEP,NFE,NJE,NQUSED,H0,Y0(1), Y0(2)
    IF(INDEX.NE.0) STOP
    TOUT = TOUT + DELTA
    GOTO 10
ENDIF

TOTAL = DTIME(TIMES)
PRINT*
PRINT*,' user      sys      total'
PRINT*,times(1),times(2),total
STOP
1001 FORMAT(1X,1PD9.1,3I6,I4,D10.2,3D12.4)
END

SUBROUTINE DIFFUN(N, T, Y, YDOT)

```

```

IMPLICIT NONE
INTEGER N
REAL *8      Y(1), YDOT(1), V, W, T
V = -80.D0
W = 8.D0
YDOT(1) = V*Y(1)-W*Y(2)+(-V+W+1.D0)*DEXP(T)
YDOT(2) = W*Y(1)+V*Y(2)+(-V-W+1.D0)*DEXP(T)
RETURN
END

```

```

SUBROUTINE JACOB(N, T, Y, PD, N0)
IMPLICIT NONE
INTEGER N, N0
REAL *8 Y(1), PD(N0,2), V, W, T
V = -80.D0
W = 8.D0
PD(1,1) = V
PD(1,2) = -W
PD(2,1) = W
PD(2,2) = V
RETURN
END

```

Program 5. Explicit Euler with the case of problem 1 of section 4.4

```

*-----
*----- This is a main program for SEULER-SUBROUTINE -----
*----- Set in the PARAMETER N=3 for of dimension array -----
*
PROGRAM SCEULER
IMPLICIT NONE
INTEGER N
PARAMETER(N=3)
REAL *8  A(N,N), Y(N), Y0(N), F(N), WRK(N), H0, HN, T, T0, TF
REAL *8  TOUT
INTEGER  NUSED, NSTEP
*-----
* Set the input parameters
* A   Jacobian matrix of dof/dot at tn
* F   Function f of y'=f(t,Y).
* Y0  The initial values of the dependent variable
* T0  The initial values of the independent variable
* TF  The final values of the independent variable
* TOUT The value of t at which output is desired next.
* WRK The working storage that is set as an array N dimension
* Y   The outcome values of the dependent variable
* H0  The initial value of stepsize
* HN  The real subsequent stepsize
* NUSED The number of dependent variable Y in the problem
* NSTEP The number of steps of integration from T0 to TF
*-----

```

```

EXTERNAL FUNCT
EXTERNAL JACOBY
*-----
* FUNCT The external subroutine for function f(t,Y) of y'=f(t,Y)
* JACOBY The external subroutine for function A(t,Y) of Jacobian.
*-----
      REAL *8   YEX(N), L1,L2,L3
*-----
* YEX   The independent variable for the exact solution
* L1,L2,L3 Roots of characteristic of A.
*-----
      INTEGER I
      REAL *8 DELTA
*-----
* INITIALIZE DATA
*-----
      DATA T0,H0,HN,TF/0.D0,0.04,0.04,20.D0/, NUSED/3/,NSTEP/0/
      DATA Y0/-1.D0,1.D0,3.D0/
      DATA L1, L2, L3/-2000.D0,-2.D0,-0.5D0/
      DATA DELTA/0.5/
*-----
      PRINT 50,(I, I=1,NUSED)
      PRINT 100,T0,0.D0,(Y0(I),I=1,NUSED),NSTEP
      PRINT 100,T0,0.D0,(Y0(I),I=1,NUSED),NSTEP
* SET TIME STEP T EQUAL TO T0.
      T = T0
      TOUT = T0 + DELTA
10    CONTINUE
      IF(TOUT .LE. TF) THEN
*-----
* CALL SUBROUTINE SEULER
          CALL SEULER(FUNCT,JACOBY,Y,Y0,A,F,WRK,T,TOUT,H0,HN,
              I              NUSED,N,NSTEP)
* ADD T WITH THE REAL SUBSEQUENT STEPSIZE HN.
* COMPUTE THE EXACT SOLUTION
          YEX(1) = DEXP(L2*TOUT) - 2.D0*DEXP(L3*TOUT)
          YEX(2) = -DEXP(L1*TOUT) + DEXP(L2*TOUT) + DEXP(L3*TOUT)
          YEX(3) = DEXP(L1*TOUT) + DEXP(L2*TOUT) + DEXP(L3*TOUT)
* PRODUCE THE OUTPUT FOR TIME STEP T
          PRINT 100,TOUT,HN,(Y(I),I=1,NUSED), NSTEP
          PRINT 100,TOUT,HN,(YEX(I),I=1,NUSED),NSTEP
* COUNT THE NUMBER OF STEPS
          TOUT = TOUT + DELTA
          GOTO 10
      ENDIF
      PRINT 200,T0, TF, NSTEP
50    FORMAT(' TIME ',2x,' STEPSIZE ',3(1x,' Y',11,' '),
1     ' STEP/70('-))
100   FORMAT(1X,F8.5,3X,F7.5,2X,3(1X,F13.6),I5)
200   FORMAT(64('-)/# of steps of integration from ',F5.2,' to ',
1     F7.3,' is ',14)
      STOP
      END

```



```

SUBROUTINE SEULER(FUNG,JAC,Y,Y0,A,F,WRK,T,TOUT,
1           H0,HN,NUSED,NDIM,NSTEP)
IMPLICIT NONE

```

```

-----
* This is the April 13, 1995 version of
* SEULER .. ELSP Solver For Ordinary Differential Equations.
* This version is in double precision.
*
* SEULER solves the initial value problem For Stiff ODE-s
*  $Dy/Dt = F(t,Y)$  , or, in component Form,
*  $DY(i)/Dt = F(i) = F(i,t,Y(1),Y(2), \dots, Y(NEQ))$ , (i=1, ...,NEQ).
* SEULER is a package based on Lambert's Algorithm.
-----
* Reference ..
* J.D. Lambert, A Stable Sequence of Steplengths For Euler's Rule Applied
* to Stiff Systems Of Differential Equations,
* comp.Math. With Appls. Vol. 12B, No5/6,pp 1141-1151, 1986.
-----
* Author and Contact .. Edward Purba
* Computer Science, Oklahoma State University
-----
* A Jacobian matrix of dof/dot at tn
* F Function f of  $y'=f(t,Y)$ .
* Y0 The initial values of the dependent variable
* T0 The initial values of the independent variable
* TOUT The value of t at which output is desired next.
* TF The final values of the independent variable
* WRK The working storage that is set as an array N dimension
* Y The outcome values of the dependent variable
* H0 The initial value of stepsize
* HN The real subsequent stepsize
* NDIM The dimension of array (size of declaration vectors or matrix)
* NUSED The actual number of vectors used the calculation
* NSTEP The number of steps of integration from T0 to TF
-----

      REAL *8 A(1), Y(1), Y0(1), F(1), WRK(1), H0, HN, T
      REAL *8 DIST, TOUT
      REAL *8 FINDHN
      INTEGER NUSED, NDIM, NSTEP
      INTEGER I, J
5     CONTINUE
      DIST = TOUT - T
-----
* CALL FUNCTION THAT CALCULATE  $f(t,Y)$  of  $y'=f(t,Y)$  THAT IS DEFINED AS
* SUBROUTINE MENTIONED IN THE MAIN PROGRAM AS EXTERNAL FUNCTION.
      CALL FUNG(F,Y0,T)
-----
* CALL FUNCTION THAT CALCULATE Jacobian A(t,Y) of  $f(t,Y)$  THAT IS DEFINED AS
* SUBROUTINE MENTIONED IN THE MAIN PROGRAM AS EXTERNAL FUNCTION.
      CALL JAC(T,Y0,A,NDIM)
-----
* SET SUBSEQUENT STEPSIZE SN EQUAL TO THE VALUE OF FUNCTION FINDHN.
      HN = FINDHN(A,F,WRK,NUSED,NDIM)

```

```

*-----
      IF (HN .GT. H0) HN=H0
*-----
* CALCULATE THE VALUE OF DEPENDENT VARIABLE Y FOR TIME STEP T
  NSTEP = NSTEP + 1
  IF(DIST.GE.HN) THEN
    DO 10 I=1,NUSED
      Y(I) = Y0(I) + HN*F(I)
      Y0(I)=Y(I)
10    CONTINUE
      T = T + HN
      GOTO 5
    ELSE
      DO 30 I=1,NUSED
        Y(I) = Y0(I) + DIST*F(I)
30    CONTINUE
      HN=DIST
    ENDIF
*-----
      RETURN
      END

      REAL *8 FUNCTION FINDHN(A,F,WRK,NUSED,NDIM)
*-----
*                                     T T
* Function FINDH .. Calculate Hn=(fn fn)/|fn A fn|
*-----
      IMPLICIT NONE
      REAL *8 A(1), F(1), WRK(1)
      REAL *8 TEMP, TPU, TPD
*-----
* A   Jacobian matrix  of dof/dot at tn
* WRK The working storage that is set as an array N dimension
* F   Function f of y'=f(t,Y).
* NDIM The dimension of array (size of declaration vectors or matrix)
* NUSED The actual number of vectors used the calculation
*-----
      INTEGER NUSED, NDIM
      INTEGER I, J, IND
      TPU = 0.0D0
      TPD = 0.0D0
*-----
* CALCULATE AF
      DO 10 I=1,NUSED
        WRK(I)= 0.D0
        DO 20 J=1,NUSED
          IND = I + (J-1)*NDIM
          WRK(I)= WRK(I) + A(IND)*F(J)
20    CONTINUE
*-----
*                                     T
* CALCULATE F F
      TPU = TPU+F(I)*F(I)
*-----

```

```

*           T
* CALCULATE F AF
          TPD = TPD + F(I)*WRK(I)
*-----
10  CONTINUE
    TPD = DABS(TPD)
    FINDHN = TPU/TPD
    RETURN
    END

*-----
      SUBROUTINE FUNCT(F,Y,T)
      IMPLICIT NONE
*-----
* Subroutine Derive is a subroutine to calculate f(t,Y) from y' = f(t,Y)
      REAL*8  T, Y(1), F(1)
*      DY1/DT = F1(T,Y1,Y2,Y3,...)
*      DY2/DT = F2(T,Y1,Y2,Y3,...)
*
*
      F(1) = -Y(1) - 0.5D0*Y(2) - 0.5D0*Y(3)
      F(2) = -0.5D0*Y(1)-1000.75D0*Y(2)+999.25D0*Y(3)
      F(3) = -0.5d0*Y(1)+999.25d0*Y(2)-1000.75d0*Y(3)
      RETURN
END

*-----
      SUBROUTINE JACOBY(T,Y,A,NDIM)
      IMPLICIT NONE
* SUBROUTINE JACOBY .. To initiate Jacobian matrix A = dof/dot
      INTEGER  NDIM
      REAL *8  T, Y(1), A(NDIM,NDIM)
      A(1,1) = -1.D0
      A(1,2) = -0.5D0
      A(1,3) = -0.5D0
      A(2,1) = -0.5D0
      A(2,2) = -1000.75D0
      A(2,3) = 999.25D0
      A(3,1) = -0.5D0
      A(3,2) = 999.25D0
      A(3,3) = -1000.75D0
      RETURN
      END

```

Program 6. Exponential Method with the case of problem 2 of section 4.4

```

*-----
* ----- This is a main program for episod-Subroutine
* ----- Set in the PARAMETER N=2 for the number of variable.

```

*

```

PROGRAM EPISOD
IMPLICIT NONE
INTEGER      N
PARAMETER(N=2)
REAL*8      T, T0, TF, Y0(N), Y(N), F(N), WRK(N),FN(N),HN
REAL *8     H0, TOL
EXTERNAL    DERIVE
INTEGER     NUSED, NSTEP, NF

```

```

* Set input parameters
* T   Time independent variable
* T0  Initial values of T
* TF  Final values of T
* Y   The outcome values of the dependent variables
* WRK The working storage that is set as an array of N.
* F   Function f of  $y' = f(t, Y)$ 
* H0  Initial stepsize
* HN  The actual stepsize
* TOL Tolerance; user specified
* TOUT The value of t at which output desired next.
* NUSED # of independent variable Y in the problem
* NSTEP # of steps of integration from T to TF
* NF   # of evaluation f(t,y)

```

```

INTEGER      I
REAL *8     DELTA, TOUT

DATA        T0/0.d0/, TF /10.D0/, H0/0.0D0/, HN/0.0D0/
DATA        TOL/1.D-6/, NF/0/, NSTEP/0/
DATA        Y0/0.D0, 0.D0/
DATA        NUSED/2/, DELTA/0.5/

INTEGER     NPRINT

OPEN(UNIT=3,FILE='expis2.out')
WRITE(3,50)(I, I=1,NUSED)
WRITE(3,100) T0,0.D0, (Y0(I), I=1,NUSED)

```

```

* Set timestep T equal to T0
  T = T0
  TOUT = T0 + DELTA
10  CONTINUE
   IF(TOUT. LE. TF) THEN
* Call subroutine EXPISOD
      CALL EXPISOD(DERIVE, T, H0, HN, Y0, Y, F, FN, WRK, TOL,
1      NUSED,NSTEP,NF,TOUT,TF,T0)
      WRITE(3,100) TOUT,HN,(Y(I),I=1,NUSED)
      TOUT = TOUT + DELTA
      GOTO 10
   ENDIF
   WRITE(3,200)T0, T-H0, NSTEP, NF
   50  FORMAT(' TIME ',2X,'STEP SIZE ',2(1X,' Y',11,' ')

```

```

1 64('-')
100 FORMAT(1x,F8.1,3x,F7.5,2x,2(1x,F13.6))
200 FORMAT(64('-')/'# of steps of integration from ',F5.2,' to ',
1 F7.1,' is ',I8,"# of evaluation of f :",I8)
STOP
END

SUBROUTINE INIT(FUNG,T0,TF,Y0,Y,F,H0,WRK,TOL,NUSED)
IMPLICIT NONE
*-----
* This subroutine is used to determined initial stepsize of ODE
* Solvers.
* The algorithm is based on concepts given on :
* 1. Gladwell, I., Shampine, L.F., and Brankin, R. W.
* "Automatic Selection of the Initial Step Size for an ODE Solver"
* J. Comput. Appl. Math. 18(1987) 175-192.
* 2. Hairer, E., Norsett, S. P., and Wanner, G.
* "Solving Ordinary Differential Equations I,
* Springer-Verlag, Berlin Heidelberg 1987, 1993.
* 3. Purba, Edward
* "Compact Numerical Methods for Stiff Differential Equations"
* Master Thesis, Computer Science, Oklahoma State University,
* 1996.
*-----
* T0 Starting time
* Y0 The initial value of the dependent variables
* WRK The working storage that is set as an array of N.
* F Function f of  $y' = f(t, Y)$ 
* H0 Initial stepsize
* TOL Desired local relative error; user specified.
* NUSED # of independent variable Y in the problem
* N size of the declaration of dimension
REAL*8 T0, TF, H0, Y0(1), Y(1), F(1), WRK(1), TOL
INTEGER NUSED
REAL*8 D0, D1, D2, TEMP0, TEMP1, TEMP
INTEGER 1
EXTERNAL FUNG

TEMP0 = 0.D0
TEMP1 = 0.D0
TOL = DMAX1(TOL, 1.D-10)
TEMP = TOL*TOL
* Calculate f(t0,y0)
CALL FUNG(T0, Y0, F)
DO I=1,NUSED
* Calculate D0 and D1
TEMP0 = TEMP0 + Y0(I)*Y0(I)
TEMP1 = TEMP1 + F(I)*F(I)
END DO
TEMP0 = TEMP0/TEMP
TEMP1 = TEMP1/TEMP
D0 = DSQRT(TEMP0/DFLOAT(NUSED))
D1 = DSQRT(TEMP1/DFLOAT(NUSED))
IF(D0 .LE. 1.D-5 .OR. D1 .LE. 1.D-5) THEN

```

```

        H0 = 1.D-6
    ELSE
        H0 = 0.01D0*(D0/D1)
    ENDIF
    H0 = DMIN1(DABS(TF-T0),H0)
*   Perform explicit Euler
    DO I=1, NUSED
        Y(I) = Y0(I) + H0*F(I)
    END DO
*   Calculate f(t1, y1)
    CALL FUNG(T0+H0, Y, WRK)
*   Estimate the second derivative
    DO I=1, NUSED
        TEMP = (WRK(I)-F(I))*(WRK(I)-F(I))
    END DO
    TEMP = TEMP/(TOL*TOL)
    D2 = DSQRT(TEMP/DFLOAT(NUSED))/H0
    D0 = DMAX1(D1,D2)
    IF(D0 .LE. 1.D-15) THEN
        H0 = DMAX1(1.D-6, 1.D-3*H0)
    ELSE
        H0 = (0.01D0/D0)**(1.D0/3.D0)
    ENDIF
    RETURN
END

```

```

SUBROUTINE EXPISOD(FUNCT, T, H0, HN, Y0, Y, F, FN, WRK, TOL,
1  NUSED,NSTEP,NF,TOUT,TF,T0)
IMPLICIT NONE

```

- *-----
- * This is the August 7, 1995 version of
 - * EXPISOD .. ELSP Solver for Ordinary Differential Equations.
 - * This version is in double precision.
 - *-----
 - * EXPISOD solves the initial value problem for stiff ODE-s
 - * $DY/Dt = f(t,Y)$, or, in component form,
 - * $DY(i)/Dt = F(i) = F(i,t,Y(1),Y(2), \dots, Y(NEQ))$, (i=1, ..., NEQ)
 - * EXPISOD is a package program for solving stiff and nonstiff ODEs
 - * based on algorithm written by Ashour, S.S, and Anna, O.T
 - * Reference ..
 - * Sami S. Ashour and Owen T. Anna
 - * "Explicit Exponential Method fo Integration of Stiff
 - * Ordinary Differential Equations"
 - * J. Guidance, Vol. 14, No. 6, pp.1234-1239.
 - *-----
 - * Author and Contact .. Edward Purba
 - * Computer Science, Oklahoma State University
 - *-----
 - * T Time independent variable
 - * Y The outcome values of the dependent variables
 - * WRK The working storage that is set as an array of N.
 - * F Function f of $y' = f(t,Y)$
 - * FN $-y''(t)/y'(t)$
 - * H Initial stepsize

```

* HN  The actual stepsize
* TOL Desired local relative error; user specified
* TOUT The value of t at which output desired next.
* NUSED # of independent variable Y in the problem
      REAL*8      T, H0,HN, Y0(1), Y(1), F(1), FN(1), WRK(1),TF, T0
      REAL*8      TOL, TOUT
      INTEGER     NUSED, NF, NSTEP
      REAL *8     DIST, FUNCT
      EXTERNAL    FUNCT
      INTEGER     I

      IF(DABS(H0).LT. 1.D-33) THEN
          CALL INIT(FUNCT,T0,TF,Y0,Y,F,H0,WRK,TOL,NUSED)
          NF = 2
          HN=H0
      ENDIF
10    CONTINUE
      DIST=TOUT-T
      NSTEP = NSTEP + 1
      NF = NF + 2
      IF(DIST.GE.HN)THEN
          CALL EPSOD(FUNCT, T, H0, HN, Y0, Y, F, FN, WRK, TOL,
1          NUSED)
          DO 20 I=1,NUSED
              Y0(I)=Y(I)
20    CONTINUE
          T = T + HN
          HN=H0
          GOTO 10
      ELSE
          CALL EPSOD(FUNCT, T, H0, DIST, Y0, Y, F, FN, WRK, TOL,
1          NUSED)
      ENDIF
      RETURN
      END

      SUBROUTINE EPSOD(FUNCT, T, H0, HN, Y0, Y, F, FN, WRK, TOL,
1          NUSED)
      IMPLICIT NONE

```

```

* T  Time independent variable
* Y  The outcome values of the dependent variables
* WRK The working storage that is set as an array of N.
* F  Function f of  $y' = f(t, Y)$ 
* FN  $-y''(t)/y'(t)$ 
* H  Initial stepsize
* HN The actual stepsize
* TOL Desired local relative error; user specified
* NUSED # of independent variable Y in the problem
      REAL*8      T, H0,HN, Y0(1), Y(1), F(1), FN(1), WRK(1)
      REAL*8      TOL
      INTEGER     NUSED, NF

      REAL*8      CURLERR

```

```

INTEGER      I
EXTERNAL     FUNCT

CURLERR = -1.D0
CALL FUNCT(T, Y0, F)
DO 10 I=1,NUSED
      FN(I) = 1.D0
      Y(I) = Y0(I) + H0*F(I)
10  CONTINUE

CALL FUNCT(T+HN, Y, WRK)
DO 20 I=1,NUSED
      IF(DABS(F(I)) .GE. 1.D-16) THEN
          FN(I) = (F(I)-WRK(I))/(HN*F(I))
          IF(FN(I) .GT. 0.D0) THEN
*-----Calculate Exponential approximation
              Y(I) = Y0(I) + F(I)*(1.D0-DEXP(-FN(I)*HN))/FN(I)
              WRK(I)=Y0(I)*DEXP(F(I)*HN/(Y0(I)+1.D-15))
          ENDIF
      ENDIF
      IF(DABS(F(I)).LT. 1.D-16 . OR. FN(I) .LE. 0.D0) THEN
*-----Calculate using RK-2
*   Calculate coefficient of K2's
          FN(I) = HN*WRK(I)
          WRK(I) = Y(I)
          F(I) = HN*F(I)
          Y(I)=Y0(I)+0.5D0* (FN(I)+F(I))
      ENDIF
      CURLERR = DMAX1(DABS(Y(I)-WRK(I))/(Y(I)+1.D-15),CURLERR)
20  CONTINUE
H0 = DSQRT((TOL/CURLERR))*HN
H0 = DMIN1(H0, 2*HN)
RETURN
END

C -----
C   Define the function of the problem in terms of the F's and the X's
C -----
SUBROUTINE  DERIVE(T,Y,F)
IMPLICIT   NONE
REAL*8     T, Y(1), F(1)

F(1) = 0.01D0-(1.D0+(Y(1)+1000.D0)*(Y(1)+1.D0))*
*          (0.01D0+Y(1)+Y(2))
F(2) = 0.01D0 - (1.D0+Y(2)*Y(2))*(0.01D0+Y(1)+y(2))
RETURN
END

```


APPENDIX G

COLLECTION OF TABLES

Table G.1 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of Kidney Problems

Name	λ	CT (sc)	Time	H	NS	NF	NJ	P
			0.0	0	0	0	0	0
			0.1	1.23D-02	33	56	13	5
			0.2	1.84D-02	38	65	14	5
			0.3	2.77D-02	43	72	15	4
			0.4	5.80D-02	46	76	17	4
	0.9902688359	0.008	0.5	5.80D-02	47	78	17	4
			0.6	1.62D-02	52	88	19	4
			0.7	3.64D-02	55	93	21	4
			0.8	2.63D-02	59	103	22	4
			0.9	2.46D-02	64	115	23	5
			1.0	2.46D-02	68	123	23	5
			0.0	0	0	0	0	0
			0.1	1.23D-02	33	56	13	5
			0.2	1.85D-02	38	65	14	5
			0.3	2.77D-02	43	73	15	4
			0.4	2.77D-02	46	79	15	4
	0.990283499	0.007	0.5	1.95D-02	51	91	16	4
			0.6	1.91D-02	57	105	17	5
			0.7	1.42D-02	63	119	18	5

			0.8	8.02D-03	72	141	20	5
			0.9	6.07D-03	86	171	21	5
			1.0	6.02D-03	108	209	23	5
			0.0	0	0	0	0	0
			0.1	1.11D-02	30	52	11	5
			0.2	1.62D-02	36	63	14	5
			0.3	1.23D-02	44	81	15	5
			0.4	6.63D-03	56	109	17	5
	0.9925211341	0.008	0.5	6.63D-02	71	139	18	5
			0.6	6.63D-03	86	154	18	5
			0.7	1.61D-02	96	165	20	5
			0.8	1.61D-02	102	176	20	5
			0.9	2.79D-02	108	182	21	5
			1.0	2.79D-02	111	188	22	5
			0.0	0	0	0	0	0
			0.1	8.24D-03	36	63	12	5
			0.2	6.10D-03	52	97	13	5
			0.3	6.10D-03	68	118	14	5
			0.4	1.06D-02	81	131	16	5
EPSODE	1.0304879856	0.008	0.5	1.90D-02	87	140	17	5
			0.6	1.90D-03	92	150	17	5
			0.7	1.9D-03	97	155	17	5
			0.8	3.44D-02	101	161	19	5
			0.9	3.44D-02	104	167	19	5
			1.0	3.44D-02	106	171	19	5
			0.0	0	0	0	0	0
			0.1	9.62D-03	38	64	9	4
			0.2	1.94D-02	44	75	11	4
			0.3	1.40D-02	49	87	12	4
			0.4	2.00D-02	55	99	12	5
	0.99	0.013	0.5	1.49D-02	61	113	13	5
			0.6	6.41D-03	71	137	17	4
			0.7	3.29D-03	95	185	25	4
			0.8	4.72D-03	114	226	29	4

			0.9	2.71D-03	168	333	47	4
			1.0	5.85D-03	207	407	69	5
			0.0	0	0	0	0	0
			0.1	4.88D-03	50	91	10	5
			0.2	7.85D-03	69	132	19	5
			0.3	3.86D-03	103	204	35	4
			0.4	6.39D-03	118	237	38	5
	0.9	0.013	0.5	3.04D-03	164	339	55	4
			0.6	3.84D-03	222	441	89	5
			0.7	1.14D-03	298	599	108	4
			0.8	2.83D-03	335	677	115	5
			0.9	1.44D-03	417	855	136	5
			1.0	1.60D-03	490	1031	156	5
			0.0	0	0	0	0	0
			0.1	3.63D-03	113	198	32	4
			0.2	5.88D-03	147	270	46	5
			0.3	4.27D-03	166	324	51	5
			0.4	2.66D-03	217	444	64	5
	0.0	0.008	0.5	1.51D-03	264	549	78	4
			0.6	5.7D-04	336	715	96	4
			0.7	8.29D-08	458	967	111	4
			0.8	1.79D-04	554	1181	131	5
			0.9	1.24D-03	659	1415	161	5
			1.0	1.08D-03	799	1708	190	5
			0.0	0	0	0	0	0
			0.1	0.92D-02	35	48	1	4
			0.2	0.38D-01	40	54	1	3
			0.3	0.38D-01	43	58	1	3
			0.4	0.38D-01	45	60	1	3
	0.9902688359	0.007	0.5	0.38D-01	48	68	2	3
			0.6	0.38D-01	51	71	2	3
			0.7	0.23D-01	54	77	2	3
			0.8	0.14D-01	59	85	2	3
			0.9	0.14D-01	66	93	2	3

			1.0	0.88D-01	74	105	2	3
			0.0	0	0	0	0	0
			0.1	0.92D-02	35	48	1	4
			0.2	0.21D-01	41	55	1	3
			0.3	0.21D-01	45	59	1	3
			0.4	0.21D-01	50	65	1	3
	0.990283499	0.008	0.5	0.13D-01	58	78	1	3
			0.6	0.73D-02	67	96	2	3
			0.7	0.11D-01	77	108	2	4
			0.8	0.77D-02	87	122	2	4
			0.9	0.53D-02	104	145	2	4
			1.0	0.56D-02	131	175	3	5
			0.0	0	0	0	0	0
			0.1	0.12D-01	35	49	1	4
			0.2	0.12D-01	44	59	1	4
			0.3	0.81D-02	53	72	1	4
			0.4	0.55D-02	66	89	2	4
	0.9925211341	0.003	0.5	0.55D-02	85	110	2	4
			0.6	0.84D-02	100	126	2	5
			0.7	0.84D-02	112	138	2	5
			0.8	0.15D-01	121	148	3	4
			0.9	0.24D-01	127	155	3	4
			1.0	0.24D-01	131	159	3	4
			0.0	0	0	0	0	0
			0.1	0.69D-02	43	58	1	5
			0.2	0.69D-02	57	72	1	5
			0.3	0.69D-02	72	88	2	5
			0.4	0.11D-01	82	100	2	5
VODE	1.0304879856	0.008	0.5	0.16D-01	89	108	2	5
			0.6	0.16D-01	95	114	2	5
			0.7	0.16D-01	101	121	2	5
			0.8	0.16D-01	107	128	2	5
			0.9	0.16D-01	113	136	2	5

			1.0	0.16D-01	120	147	2	5
			0.0	0	0	0	0	0
			0.1	0.11D-01	33	48	1	4
			0.2	0.17D-01	41	57	1	4
			0.3	0.17D-01	47	63	1	4
			0.4	0.11D-01	53	76	2	4
	0.99	0.015	0.5	0.11D-01	62	86	2	4
			0.6	0.76D-02	74	116	4	4
			0.7	0.22D-02	104	171	6	4
			0.8	0.24D-02	153	257	10	3
			0.9	0.24D-02	195	348	15	3
			1.0	0.24D-02	238	444	21	3
			0.0	0	0	0	0	0
			0.1	0.45D-02	52	74	2	4
			0.2	0.96D-03	100	176	6	4
			0.3	0.62D-03	175	313	10	3
			0.4	0.17D-02	240	441	16	3
	0.9	0.012	0.5	0.17D-02	300	564	23	3
			0.6	0.26D-02	358	689	29	3
			0.7	0.26D-02	397	777	35	3
			0.8	0.26D-02	435	855	41	3
			0.9	0.51D-02	473	932	47	4
			1.0	0.37D-02	531	1032	48	3
			0.0	0	0	0	0	0
			0.1	0.98D-03	141	214	8	3
			0.2	0.70D-03	259	426	14	3
			0.3	0.12D-02	363	626	20	3
			0.4	0.12D-02	450	796	27	3
	0.0	0.027	0.5	0.32D-02	505	919	33	3
			0.6	0.32D-02	536	989	38	3
			0.7	0.32D-02	567	1057	43	3
			0.8	0.14D-02	622	1137	47	3
			0.9	0.60D-02	657	1199	51	4

			1.0	0.12D-02	885	1650	69	3
			0.0	0	0	0	0	0
			0.1	3.39D-01	5	10	5	1
			0.2	3.39D-01	5	10	5	1
			0.3	3.39D-01	5	10	5	1
			0.4	3.39D-01	5	10	5	1
	0.9902688359	0.015	0.5	3.39D-01	6	14	6	1
			0.6	3.39D-01	6	14	6	1
			0.7	3.39D-01	6	14	6	1
			0.8	3.39D-01	6	14	6	1
			0.9	3.39D-01	6	14	6	1
			1.0	1.35D+00	7	15	7	1
			0.0	0	0	0	0	0
			0.1	3.94D-01	5	10	5	1
			0.2	3.94D-01	5	10	5	1
			0.3	3.94D-01	5	10	5	1
			0.4	3.94D-01	5	10	5	1
	0.990283499	0.01	0.5	3.94D-01	6	14	6	1
			0.6	3.94D-01	6	14	6	1
			0.7	3.94D-01	6	14	6	1
			0.8	3.94D-01	6	14	6	1
			0.9	3.94D-01	7	15	7	1
			1.0	1.36D+00	7	15	7	1
			0.0	0	0	0	0	0
			0.1	3.69D-01	5	10	5	1
			0.2	3.69D-01	5	10	5	1
			0.3	3.69D-01	5	10	5	1
			0.4	3.69D-01	5	10	5	1
	0.9925211341	0.017	0.5	3.69D-01	6	14	6	1
			0.6	3.69D-01	6	14	6	1
			0.7	3.69D-01	6	14	6	1
			0.8	3.69D-01	6	14	6	1
			0.9	3.69D-01	7	15	7	1

			1.0	1.72D+00	7	15	7	1
			0.0	0	0	0	0	0
			0.1	1.32D-01	5	10	5	1
			0.2	1.32D-01	6	13	6	1
			0.3	1.32D-01	6	14	6	1
			0.4	8.11D-01	7	14	7	1
LSODE	1.0304879856	0.011	0.5	8.11D-01	7	14	7	1
			0.6	8.11D-01	7	14	7	1
			0.7	8.11D-01	7	14	7	1
			0.8	8.11D-01	7	14	7	1
			0.9	8.11D-01	7	14	7	1
			1.0	8.11D-01	7	14	7	1
			0.0	0	0	0	0	0
			0.1	3.86D-01	5	10	5	1
			0.2	3.86D-01	5	10	5	1
			0.3	3.86D-01	5	10	5	1
			0.4	3.86D-01	5	10	5	1
	0.99	0.008	0.5	3.86D-01	6	14	6	1
			0.6	3.86D-01	6	14	6	1
			0.7	3.86D-01	6	14	6	1
			0.8	3.86D-01	6	14	6	1
			0.9	3.17D-01	7	17	8	1
			1.0	3.17D-01	7	17	8	1
			0.0	0	0	0	0	0
			0.1	8.02D-02	5	10	5	1
			0.2	4.12D-01	7	14	7	1
			0.3	4.12D-01	7	14	7	1
			0.4	4.12D-01	7	14	7	1
	0.9	0.014	0.5	4.12D-01	7	14	7	1
			0.6	4.12D-01	7	14	7	1
			0.7	1.03D-01	8	20	9	1
			0.8	5.15D-02	11	29	13	2
			0.9	5.15D-02	13	35	15	2

			1.0	1.09D-01	14	36	16	3
			0.0	0	0	0	0	0
			0.1	3.31D-01	7	14	7	1
			0.2	3.31D-01	7	14	7	1
			0.3	3.31D-01	7	14	7	1
			0.4	3.31D-01	7	14	7	1
	0.0	0.012	0.5	3.31D-01	8	18	8	1
			0.6	3.31D-01	8	18	8	1
			0.7	3.31D-01	8	18	8	1
			0.8	1.37D+00	9	19	9	1
			0.9	1.37D+00	9	19	9	1
			1.0	1.37D+00	9	19	9	1
			0.0	0	0	0	0	0
			0.1	0.60D-2	38	60	1	5
			0.2	0.32D-01	46	71	1	4
			0.3	0.32D-01	49	77	1	4
			0.4	0.5D-01	52	85	2	4
	0.9902688359	0.005	0.5	0.32D-01	55	92	2	4
			0.6	0.32D-01	58	98	2	4
			0.7	0.32D-01	61	107	2	4
			0.8	0.22D-01	65	117	2	4
			0.9	0.15D-01	70	132	2	4
			1.0	0.14D-01	77	147	2	5
			0.0	0	0	0	0	0
			0.1	0.60D-02	38	60	1	5
			0.2	0.28D-01	46	72	1	4
			0.3	0.28D-01	50	84	1	4
			0.4	0.11D-01	56	94	1	5
	0.990283499	0.013	0.5	0.11D-01	65	117	1	4
			0.6	0.10D-01	74	138	1	4
			0.7	0.91D-02	83	161	1	4
			0.8	0.82D-02	94	182	1	5
			0.9	0.59D-02	108	224	2	6

			1.0	0.55D-02	125	251	2	6
			0.0	0	0	0	0	0
			0.1	0.67D-02	37	59	1	5
			0.2	0.13D-01	46	72	1	5
			0.3	0.13D-01	53	92	1	6
			0.4	0.77D-02	63	120	1	6
	0.9925211341	0.014	0.5	0.60D-02	78	145	1	5
			0.6	0.80D-02	94	171	1	5
			0.7	0.86D-02	106	189	2	4
			0.8	0.15D-01	114	203	2	4
			0.9	0.21D-01	119	210	2	5
			1.0	0.33D-01	123	215	2	5
			0.0	0	0	0	0	0
			0.1	0.67D-02	43	68	1	6
			0.2	0.61D-02	60	100	1	6
			0.3	0.74D-02	76	122	1	6
			0.4	0.84D-02	88	144	1	5
MEBDF	1.0304879856	0.027	0.5	0.11D-01	98	158	2	4
			0.6	0.18D-01	105	166	2	4
			0.7	0.25D-01	110	175	2	5
			0.8	0.25D-01	113	181	2	5
			0.9	0.41D-01	116	186	2	5
			1.0	0.41D-01	119	189	2	5
			0.0	0	0	0	0	0
			0.1	0.60D-02	38	60	1	5
			0.2	0.20D-01	47	74	1	5
			0.3	0.20D-01	52	89	1	5
			0.4	0.64D-02	61	118	1	6
	0.99	0.012	0.5	0.28D-02	88	176	1	4
			0.6	0.17D-02	111	245	3	5
			0.7	0.24D-02	138	299	5	5
			0.8	0.16D-02	210	498	9	3
			0.9	0.31D-02	251	597	14	3

			1.0	0.14D-02	305	728	19	4
			0.0	0	0	0	0	0
			0.1	0.27D-02	58	113	2	5
			0.2	0.26D-02	95	184	5	4
			0.3	0.34D-02	135	286	9	3
			0.4	0.31D-02	186	402	14	3
	0.9	0.005	0.5	0.21D-02	229	505	19	3
			0.6	0.22D-02	276	620	24	3
			0.7	0.28D-02	324	728	29	3
			0.8	0.10D-02	384	861	35	2
			0.9	0.13D-02	434	985	41	3
			1.0	0.19D-02	482	1095	44	3
			0.0	0	0	0	0	0
			0.1	0.41D-02	142	318	6	4
			0.2	0.16D-02	190	434	12	3
			0.3	0.89D-03	239	551	17	3
			0.4	0.58D-02	303	705	22	4
	0.0	0.008	0.5	0.27D-02	351	810	26	4
			0.6	0.27D-02	402	925	31	4
			0.7	0.25D-02	452	1050	36	3
			0.8	0.13D-02	513	1183	40	4
			0.9	0.31D-02	590	1390	47	3
			1.0	0.64D-02	620	1456	51	4

Table G.2 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of Autocatalytic Reaction Pathway Problem

Name	CT (sc)	Time (in Log ₁₀)	H	NS	NF	NJ	P
		6.D-01	4.17D-01	36	59	16	5
		1.6D00	1.73D00	84	164	33	3
		2.6D00	2.22D01	126	244	40	5
		3.6D00	3.11D02	171	336	53	5
		4.6D00	1.30D03	214	421	58	4
EPSODE	0.012	5.6D00	2.88D04	253	490	66	4
		6.6D00	4.25D05	286	545	83	3
		7.6D00	6.43D06	305	574	90	5
		8.6D00	3.51D07	331	622	98	3
		9.6D00	4.79D08	344	641	104	4
		1.1D01	1.00D10	366	681	121	1
		6.D-01	2.64D-01	66	101	3	3
		1.6D00	6.85D-01	129	213	5	3
		2.6D00	1.28D01	216	410	7	3
		3.6D00	2.02D02	291	544	8	4
		4.6D00	1.82D03	346	640	9	4
VODE	0.007	5.6D00	2.40D04	390	693	10	4
		6.6D00	1.25D05	453	784	11	4
		7.6D00	3.29D06	480	816	11	4
		8.6D00	9.90D07	494	843	12	2
		9.6D00	3.68D09	500	854	13	1
		1.1D01	1.79D10	508	869	14	1
		6.D-01	3.05D-01	41	57	11	4
		1.6D00	2.25D00	80	105	16	4
		2.6D00	1.98D01	138	179	24	4
		3.6D00	1.72D02	185	245	30	4
		4.6D00	3.76D03	234	312	38	5
LSODE	0.012	5.6D00	2.76D04	282	383	45	4

		6.6D00	3.31D05	313	420	51	4
		7.6D00	6.34D06	336	448	56	3
		8.6D00	1.08D08	349	464	60	3
		9.6D00	1.34D09	358	477	64	1
		1.1D01	2.36D10	363	483	66	1
		6.D-01	3.08D-01	61	121	4	4
		1.6D00	1.57D00	98	185	6	4
		2.6D00	1.61D01	212	494	8	4
		3.6D00	1.89D02	284	646	11	5
		4.6D00	1.82D03	346	755	13	5
MEBDF	0.010	5.6D00	2.35D04	396	842	14	6
		6.6D00	3.28D05	435	909	15	3
		7.6D00	4.87D06	460	949	17	3
		8.6D00	6.14D07	476	972	17	2
		9.6D00	9.83D08	485	986	18	1
		1.1D01	2.64D10	493	1003	20	1

Table G.3 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of D4 of Enright et al.

Name	CT (sc)	Time	H	NS	NF	NJ	P
		0.0D00	0.0D0	0	0	0	0
		5.0D00	2.21D00	25	29	21	3
		1.0D01	3.05D00	27	31	22	4
		1.5D01	4.52D00	28	32	23	4
		2.0D01	4.52D00	29	34	23	4
EPSODE	0.003	2.5D01	4.52D00	30	36	23	4
		3.0D01	6.04D00	31	38	23	5
		3.5D01	6.04D00	32	39	24	5
		4.0D01	9.06D00	33	40	25	5
		4.5D01	9.06D00	33	40	25	5
		5.0D01	9.06D00	34	42	25	5
		0.0D00	0	0	0	0	0
		5.0D00	1.10D00	22	36	1	2
		1.0D01	2.32D00	26	44	1	3
		1.5D01	2.32D00	28	46	1	3
		2.0D01	2.32D00	30	48	1	3
VODE	0.012	2.5D01	2.32D00	32	50	1	3
		3.0D01	4.39D00	33	52	1	4
		3.5D01	4.39D00	35	55	1	4
		4.0D01	4.39D00	36	56	1	4
		4.5D01	4.39D00	37	57	1	4
		5.0D01	4.39D00	38	58	1	4
		0.0D00	0	0	0	0	0
		5.0D00	1.54D00	17	27	9	3
		1.0D01	2.06D00	21	32	10	3
		1.5D01	2.06D00	23	34	10	3
		2.0D01	4.86D00	25	39	11	4
LSODE	0.010	2.5D01	4.86D00	26	40	11	4

		3.0D01	4.86D00	27	41	11	4
		3.5D01	4.86D00	28	42	11	4
		4.0D01	4.86D00	29	43	11	4
		4.5D01	4.86D00	30	44	11	4
		5.0D01	4.86D00	31	45	11	4
		0.0D00	0	0	0	0	0
		5.0D00	1.70D00	23	34	1	3
		1.0D01	3.87D00	26	38	1	4
		1.5D01	3.87D00	27	39	1	4
		2.0D01	3.87D00	28	40	1	4
MEBDF	0.018	2.5D01	3.87D00	29	41	1	4
		3.0D01	3.87D00	31	43	1	4
		3.5D01	5.58D00	32	46	1	5
		4.0D01	5.58D00	33	48	1	5
		4.5D01	5.58D00	34	50	1	5
		5.0D01	5.58D00	35	52	1	5

Table G.4 Table of Performances of MEBDF, VODE, LSODE, and EPSODE for the Case of Problems Proposed by Gupta and Wallace.

Name	CT (sc)	Time	H	NS	NF	NJ	P
		0.0D00	0.0D0	0	0	0	0
		1.0D00	9.10D-02	31	57	10	5
		2.0D00	9.10D-02	42	78	11	5
		3.0D00	9.10D-02	53	100	12	5
		4.0D00	6.90D-02	66	124	13	5
EPSODE	0.012	5.0D00	6.90D-02	80	151	14	5
		6.0D00	5.26D-02	99	191	15	5
		7.0D00	4.02D-02	121	229	17	5
		8.0D00	4.02D-02	146	265	18	5
		9.0D00	3.06D-02	177	320	21	5
		1.0D01	2.34D-02	212	367	23	5
		0.0D00	0.0D0	0	0	0	0
		1.0D00	9.12D-01	28	45	1	4
		2.0D00	9.12D-01	39	56	1	4
		3.0D00	9.12D-01	50	69	1	4
		4.0D00	9.12D-01	61	80	1	4
VODE	0.007	5.0D00	9.12D-01	72	93	2	4
		6.0D00	9.12D-01	83	104	2	4
		7.0D00	9.12D-01	94	117	2	4
		8.0D00	9.12D-01	105	129	2	4
		9.0D00	9.12D-01	116	141	2	4
		1.0D01	9.12D-01	127	154	3	4
		0.0D00	0.0D0	0	0	0	0
		1.0D00	9.79D-02	27	37	6	5
		2.0D00	1.15D-01	37	53	7	5
		3.0D00	1.15D-01	45	61	7	5
		4.0D00	1.15D-01	54	71	8	5

LSODE	0.010	5.0D00	1.15D-01	63	80	8	5
		6.0D00	1.15D-01	71	88	8	5
		7.0D00	1.15D-01	80	98	9	5
		8.0D00	1.15D-01	89	107	9	5
		9.0D00	1.15D-01	97	116	10	5
		1.0D01	1.15D-01	106	125	10	5
		0.0D00	0.0D0	0	0	0	0
		1.0D00	9.47D-01	25	38	1	6
		2.0D00	1.27D-01	33	52	1	6
		3.0D00	1.38D-01	43	69	1	6
		4.0D00	1.46D-01	50	80	1	6
MEBDF	0.010	5.0D00	1.46D-01	57	97	1	6
		6.0D00	1.38D-01	64	106	1	6
		7.0D00	1.46D-01	71	117	1	6
		8.0D00	1.36D-01	78	126	1	6
		9.0D00	1.46D-01	85	138	1	6
		1.0D01	1.46D-01	92	154	2	6

Table G.5 The Computation Results for Problems 1 of Chapter 4.4
Using Modified Euler Method and Exact Solution

Time t	Modified Euler			Exact solution		
	y1	y2	y3	y1	y2	y3
0.0	-1.0	1.0	3.0	-1.0	1.0	3.0
0.5	-1.200338	1.134247	1.126747	-1.189722	1.146680	1.146680
1.0	-1.082320	0.729724	0.727378	-1.077726	0.741866	0.741866
1.5	-0.893776	0.513017	0.513006	-0.894946	0.522154	0.522154
2.0	-0.713064	0.380666	0.379127	-0.717443	0.386195	0.386195
2.5	-0.560552	0.288582	0.288459	-0.566272	0.293243	0.293243
3.0	-0.437826	0.222322	0.221329	-0.443782	0.225609	0.225609
3.5	-0.340953	0.171567	0.171442	-0.346636	0.174686	0.174686
4.0	-0.265178	0.133271	0.132632	-0.270335	0.135671	0.135671
4.5	-0.206110	0.103236	0.103131	-0.210675	0.105523	0.105523
5.0	-0.160163	0.080331	0.079922	-0.164125	0.082130	0.082130
5.5	-0.124439	0.062276	0.062194	-0.127839	0.063945	0.063945
6.0	-0.096680	0.048476	0.048215	-0.099568	0.049793	0.049793
6.5	-0.075110	0.037587	0.037527	-0.077546	0.038776	0.038776
7.0	-0.058352	0.029260	0.029094	-0.060394	0.030198	0.030198
7.5	-0.045333	0.022688	0.022645	-0.047035	0.023518	0.023518
8.0	-0.035218	0.017662	0.017556	-0.036631	0.018316	0.018316
8.5	-0.027361	0.013695	0.013665	-0.028528	0.014264	0.014264
9.0	-0.021256	0.010661	0.010594	-0.022218	0.011109	0.011109
9.5	-0.016514	0.008267	0.008247	-0.017303	0.008652	0.008652
10.	-0.012829	0.006436	0.006393	-0.013476	0.006738	0.006738

Table G.6 The Computation Results for Problems 2 of Chapter 4.4
Using Modified Euler Method and Mathematica Software

Time t	Modified Euler		Mathematica	
	y1	y2	y1	y2
0.0	0.0	0.0	0.0	0.0
0.5	-0.014776	0.004980	-0.0149596	0.00497987
1.0	-0.019692	0.009970	-0.0199493	0.0099697
1.5	-0.024607	0.014960	-0.024939	0.0149595
2.0	-0.029935	0.019949	-0.0299286	0.0199492
2.5	-0.034852	0.024939	-0.0349182	0.0249389
3.0	-0.039768	0.029929	-0.0399077	0.0299285
3.5	-0.044686	0.034918	-0.0448972	0.0349181
4.0	-0.049604	0.039908	-0.0498866	0.0399076
4.5	-0.054522	0.044898	-0.0548759	0.044897
5.0	-0.059855	0.049887	-0.0598651	0.0498864
5.5	-0.064775	0.054876	-0.0648543	0.0548757
6.0	-0.069696	0.059865	-0.0698435	0.0598649
6.5	-0.074617	0.064855	-0.0748326	0.0648541
7.0	-0.079539	0.069844	-0.0798216	0.0698433
7.5	-0.084462	0.074833	-0.0848105	0.0748323
8.0	-0.089799	0.079822	-0.0897994	0.0798213
8.5	-0.094723	0.084811	-0.0947882	0.0848102
9.0	-0.099648	0.089800	-0.0997769	0.0897991
9.5	-0.104573	0.094788	-0.104766	0.0947879
10.	-0.109499	0.099777	-0.109754	0.0997766

Table G.7 The Computation Results for Problems 1 of Chapter 4.4
Using Exponential Method and the Exact Solution

Time	Exponential method			Exact solution		
t	y1	y2	y3	y1	y2	y3
0.0	-1.0	1.0	3.0	-1.0	1.0	3.0
0.5	-1.189366	1.45198	1.145291	-1.189722	1.146680	1.146680
1.0	-1.076826	0.740041	0.739956	-1.077726	0.741866	0.741866
1.5	-0.893643	0.520202	0.520259	-0.894946	0.522154	0.522154
2.0	-0.715909	0.384379	0.384339	-0.717443	0.386195	0.386195
2.5	-0.564626	0.291533	0.291501	-0.566272	0.293243	0.293243
3.0	-0.442145	0.224069	0.224091	-0.443782	0.225609	0.225609
3.5	-0.345085	0.173349	0.173364	-0.346636	0.174686	0.174686
4.0	-0.268921	0.134550	0.134539	-0.270335	0.135671	0.135671
4.5	-0.209407	0.104548	0.104558	-0.210675	0.105523	0.105523
5.0	-0.163005	0.081310	0.081302	-0.164125	0.082130	0.082130
5.5	-0.126863	0.063247	0.063253	-0.127839	0.063945	0.063945
6.0	-0.098727	0.049213	0.049209	-0.099568	0.049793	0.049793
6.5	-0.076828	0.038290	0.038293	-0.077546	0.038776	0.038776
7.0	-0.059785	0.029797	0.029795	-0.060394	0.030198	0.030198
7.5	-0.046523	0.023518	0.023185	-0.047035	0.023518	0.023518
8.0	-0.036203	0.018044	0.018042	-0.036631	0.018316	0.018316
8.5	-0.028172	0.014039	0.014041	-0.028528	0.014264	0.014264
9.0	-0.021923	0.010925	0.010926	-0.022218	0.011109	0.011109
9.5	-0.017059	0.008502	0.008502	-0.017303	0.008652	0.008652
10.	-0.013275	0.006616	0.006616	-0.013476	0.006738	0.006738

Table G.8 The Computation Results for Problems 2 of Chapter 4.4
Using Exponential Method and Mathematica Software

Time	Exponential method		Mathematica	
t	y1	y2	y1	y2
0.0	0.0	0.0	0.0	0.0
0.5	-0.014960	0.004982	-0.0149596	0.00497987
1.0	-0.019951	0.009972	-0.0199493	0.0099697
1.5	-0.024941	0.014961	-0.024939	0.0149595
2.0	-0.029935	0.019951	-0.0299286	0.0199492
2.5	-0.034928	0.024941	-0.0349182	0.0249389
3.0	-0.039904	0.029932	-0.0399077	0.0299285
3.5	-0.044894	0.034923	-0.0448972	0.0349181
4.0	-0.049885	0.039914	-0.0498866	0.0399076
4.5	-0.054876	0.044906	-0.0548759	0.044897
5.0	-0.059867	0.049898	-0.0598651	0.0498864
5.5	-0.064858	0.054890	-0.0648543	0.0548757
6.0	-0.069850	0.059882	-0.0698435	0.0598649
6.5	-0.074842	0.064875	-0.0748326	0.0648541
7.0	-0.079834	0.069867	-0.0798216	0.0698433
7.5	-0.084826	0.074860	-0.0848105	0.0748323
8.0	-0.089818	0.079853	-0.0897994	0.0798213
8.5	-0.094811	0.084846	-0.0947882	0.0848102
9.0	-0.099803	0.089839	-0.0997769	0.0897991
9.5	-0.104796	0.094832	-0.104766	0.0947879
10.	-0.109789	0.099826	-0.109754	0.0997766

VITA

Edward Purba

Candidate for the Degree of

Master of Science

Thesis: COMPACT NUMERICAL METHODS FOR STIFF
DIFFERENTIAL EQUATIONS

Major Field: Computer Science

Biographical:

Personal Data: Born in Rantau Prapat, Propinsi Sumatera Utara, Indonesia, on October 10, 1958, the son of Daulat Purba and Siti Minar Siringo-ringo.

Graduated from SMA Negri I Medan, in December 1976; received Sarjana S-1 in Applied Mathematics from Bandung Institute of Technology, Bandung, Indonesia, March 1984. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 1996.

Experience: Lecturer, Polytechnic of the Bandung Institute of Technology, from July 1984 to June 1986; Researcher, Agency for the Assessment and Application of Technology (BPP Teknologi), from 1989 to 1991; Engineering System Analyst, Indonesian Aircraft Industries (IPTN), from 1984 to 1988; Head of Numerical Intensive Computing, IPTN Computing Center, from 1989 to 1991; Teaching Assistant, Department of Computer Science, Oklahoma State University, in Spring 1995.

Professional Memberships: Former Representative for West Java to the Indonesian Remote Sensing Society (MAPIN); Organizer of Indonesian Association of Artificial Intelligence (HAI AI); Member of Association for Computing Machinery (ACM).