International Journal of Advanced Technology in Engineering and Science www.ijates.com
Volume No 03, Special Issue No. 01, April 2015 ISSN (online): 2348 – 7550

A STUDY OF MATLAB SOLVERS FOR SOLVING INITIAL VALUE PROBLEMS (IVP)

Ameer Hassan Abdullahi

Department Mathematics, Sharda University, Greater Noida, (India)

ABSTRACT

This paper concentrates on solving ordinary differential equations in MATLAB by using build-in solvers such as ode45, ode23 and ode15s. Firstly, we give an introduction of MATLAB Build-in solvers and existence and uniqueness solution of initial value problems (IVP). Next we introduce the general syntax of ode solvers and there implementation solution(text problems). Then, the reason of the stiff system of ordinary differential equation is discussed, we gave the examples of stiff equation, comparisons are made and plotting their numerical solutions which we obtain with the MATLAB using ode solvers (ode45, ode23 and ode15s). Next we provide certain details on the algorithm behind the ode solvers ode45, ode23 and ode15s. Explicit formula for non-stiff systems, implicit formula for stiff systems and implementation of Runge-kutta-Fehlberg method. Finally, we conclude that the solver is more accurate in solving ode.

I. INTRODUCTION

1.1Matlab ODE Suite

The MATLAB ODE suite is a collection of Matlab codes (M-file) developed by lowrence F. Shampine and Mark W.Reichelt for solving initial value problems given by first-order ODE or systems of ordinary differential equations and plotting their numerical solutions. and it contain seven solvers ode45, ode23, ode15s, ode113,ode23s and ode23tb. The three codes ode23, ode45, and ode113 are designed to solve non-stiff problems and the two codes ode23s and ode15s are designed to solve stiff problems. for example, that we want to solve the first order differential equation y'= f(t,y) we can use MATLABs built-in function(solvers).
we shall mainly discuss the general purpose solves ode45, ode23 and ode15s. Although we will not discuss other solvers, it is important to realize that the calling syntax is the same for each solver in ODE suite. The detail are given as follows.

1.2 Existence and Uniqueness of Solution IVP

Before exploring any method of solving the problems, we need to check weather the solution exist and if it exist, is the solution unique?. Then Consider the initial value problem for a system of n ordinary differential equations of first order:

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0, \tag{2.1}$$

on the interval [a,b], Where $y'(t)$ is unknown function that is being sought and y_0 is the value of the function at the initial time t_0 The given function f(t,y) of two variables defines the differential equation. Firstly let have some definitions.

$$f : R \times R^n \longrightarrow R^n,$$

is continuous in t and Lispschitz continuous in y, which it is describe blow.

definition 1 *A continuous function $f(t, y)$ in a set $D \subseteq \Re^2$ is said to satisfied Lipschitz condition on the variable y. If the constant $L \leq 0$ with*

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

where (t, y_1), $(t, y_2) \in D$ *and* L *is called Lipschitz Constant.*

II. BUILT-IN SOLVERS

We shall mainly discuss the general purpose solves ode45, ode23 and ode15s. Although we will not discuss other solvers, it is important to realize that the calling syntax is the same for each solver in ODE suite. The detail are given as follows.

- **Syntax**

$$[T, Y] = \text{solver(yprime,tspan,y0)}$$
$$[T, Y] = \text{solver(yprime,tspan,y0,options)}$$

Where solver is one of ode45, ode23, ode15s, ode113,ode23s and ode23tb. You can call any of these solvers by substituting the placeholder, solver, with any of the function names.

- **yprime**
  A function handle that evaluates the right side of the differential equations.
- **tspan**
  A vector specifying the interval of integration, $[t0, tf]$. The solver imposes the initial conditions at tspan(1) and integrates from tspan(1) to tspan(end).
- **y0**
  A vector of initial conditions. Make sure that the order corresponds to the ordering used to write y, z and their derivatives in terms of t. Also note that, if t consists of 5 variables, then we need an input of 5 initial conditions.
- **Options:-** Optinal Argument integration argument using odeset function. commonly used properties include a scalar relative error tolerance RelTol(le-3 by default) and a vector of absolute error tolerance AbsTol(all component are le-6 by default).

And use helpwin for odeset for detail.

The two parameters RelTol and AbsTol can be use to adjust a desired accuracy.

And MATLAB adaptability chooses the time step such that the error at n^{th} step satisfies

$e_n \leq \max\ (r \mid y_n \mid, a)$

Where a is the relative tolerance and a is the absolute tolerance. To change the relative and and absolute tolerance, odeset function can be employed as follows.

Option = odeset(RelTol',le - 12', Abstol', le - 8)

By adjusting these tolerance levels one can enforce the solver to take small time steps to preserve the desired accuracy.

2.1 Error Tolerance Properties

The solvers use standard local error control techniques for monitoring and controlling the error of each integration step. At each step, the local error e in the i^{th} component of the solution is estimated and is required to be less than or equal to the acceptable error, which is a function of two user-defined tolerances RelTol and AbsTol.

$$|e(i)| <= max(RelTol * abs(y(i)), AbsTol(i))$$

- **RelTol:-** Is the relative accuracy tolerance, a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in the answer. The default, 1e-3, corresponds to 0.1% accuracy.

- **AbsTol:-** Is a scalar or vector of the absolute error tolerances for each solution component. AbsTol(i) is a threshold below which the values of the corresponding solution components are unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. The default value is 1e-6. Set tolerances using odeset, either at the command line or in the ODE file.

III. ALGORITHMS USE IN SOLVERS

MATLAB's ODE solvers.

Solver	Problem type	Type of algorithm
ode45	Nonstiff	Explicit Runge Kutta pair, orders 4 and 5
ode23	Nonstiff	Explicit Runge Kutta pair, orders 2 and 3
ode113	Nonstiff	Explicit linear multistep, orders 1 to 13
ode15s	Stiff	Implicit linear multistep, orders 1 to 5
ode23s	Stiff	Modified Rosenbrock pair (one-step), orders 2 and 3
ode23t	Mildly stiff	Trapezoidal rule (implicit), orders 2 and 3
ode23tb	Stiff	Implicit Runge Kutta type algorithm, orders 2 and 3

3.1 The ode45 Method

ode45 it is based on an explicit Runge-Kutta(4,5) formula, the Dormand-prince pair. It is a one-step solver in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, ode45 is the best function to apply as a first try for most problems. It use long step size, the default is to use the interpolated to compute solution values at four points equally spaced within the span of each natural step.

3.2 The ode23 Method

ode23 is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of moderate stiffness. Like ode45, ode23 is a one-step solver. It advances from y_n to y_{n+1} with the third-order method (so called local extrapolation) and controls the local error by taking the difference between the third-order and the second-order numerical solutions(fourth order Runge-kutta).

3.3 The ode15s Method

Is a variable order solver based on (NDFs). Optionally, it uses the (BDFs, also known as Gear's method) that are usually less efficient. Like ode113, ode15s is a multistep solver. Try ode15s when ode45 fails. The code ode15s for stiff systems is a quasi-constant step size implementation of the NDFs of order 1 to 5.

ode45 uses the Dormand-Prince pair. The difference between the two methods is then used as an estimate of the local error in the lower-order method. If a local error estimate seems too large, it is natural to try again with a shorter step based on an asymptotic expansion of the error. This method of step control works well on many problems in practice.

IV. IMPLEMENTATION OF SOLVERS

In some cases, you can improve ODE solver performance by specially coding your ODE file. For instance to improve solver performance, often used in conjunction with a specially coded ODE file, is to tune solver parameters. The default parameters in the ODE solvers are selected to handle common problems. In some cases, however, tuning the parameters for a specific problem can improve performance significantly. You do this by supplying the solvers with one or more property values contained within an options argument.

Examples.1

solve this problem y'=e^{-y}, $o \leq t \leq 10$ y(0)=0 $y_{exact} = int(t+1)$

Solution:

MATLAB code;-

```
function solvesimpleeODE
[T,Ycompute]=ode45(@yprime,[0,20],0);
%code of comparison between computede and exact solution
Yexact=yexact(T);
Yerror=abs(Ycompute-Yexact);
figure (1); plot(T,Ycompute, '-b*', T, yexact(T), '-r' )
title('Analytic solution and computed solution')
xlabel('x-axis')
ylabel('y-axis')
legend('Analytic solution',' computed solution')
figure (2); plot(T,Yerror, '*-r' )
title('Analytic solution and Error')
xlabel('x-axis')
ylabel('y-axis')
legend('Error')
%code for yprime %use as subfunction
function yprime=yprime(t,y);
yprime=exp(-y);
%code for yexact %use as subfunction
function yexact=yexact(t);
yexact=log(t+1);
```

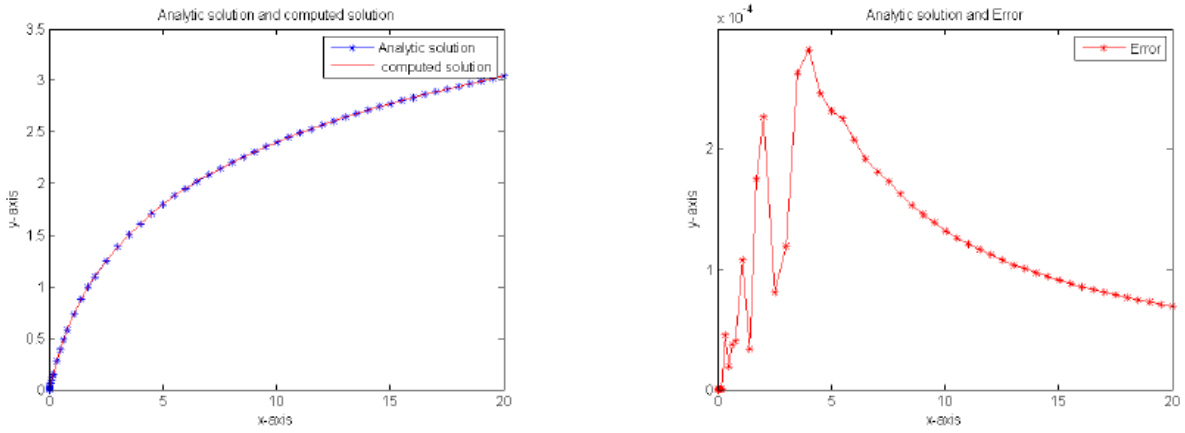

Figure 1: comparison between computede and exact solution

In the same problem above we see the solver of ode23 solution

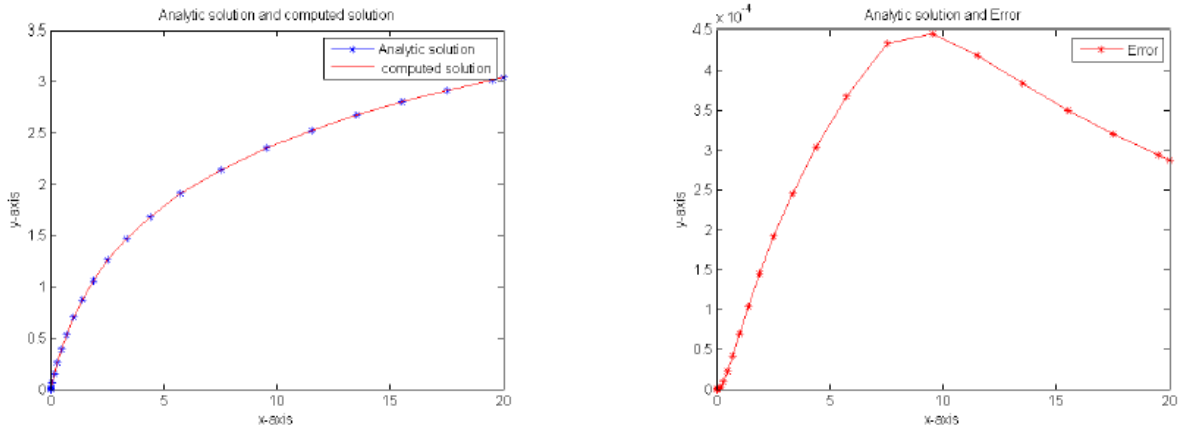

Figure 2: comparison between computede and exact solution

4.1 System of Equations

Example 2.

Consider the well-known lorentz equation given by

$$\begin{aligned} y_1' &= -\beta y_1 + y_2 y_3 \\ y_2' &= -\rho y_2 + \rho y_3 \\ y_1' &= y_1 y_2 + \sigma y_2 - y_3 \end{aligned}$$

Where β =3/8, $\rho = 10$, $\sigma = 28$. the initial value are given by $y_1(0)$=$y_2(0)$=$y_3(0)$=ε, ε is very small positive number i.e $\varepsilon = 10^{-10}$.

Solution: In the process of solving ODEs Using ODE solver available in MATLAB, it may require to introduce additional argument so that when this parameter are changed in ODEs the can be modified through these additional arguments for instant in the given lorentz equation example involve β, ρ, σ can be consider as additional arguments, the are passed by declaring them global variable. following is the code for solving this system using ode45

Solution:

MATLAB code.

```
function lorentz1
global beta rho sigma
rho=10;beta=8/3;sigma=28;
y0=[0 0 1e-10];
[T Y]=ode45(@loren,[0 100],y0);
disp([T Y])
figure(1);
plot(T,Y)
figure(2);
plot3(Y(:,1),Y(:,2),Y(:,3))
axis([10 42 -20 20 -20 25])
figure(3);
comet3(Y(:,1),Y(:,2),Y(:,3))
function dy=loren(t,y)
global beta rho sigma
dy=zeros(3,1);
dy(1)=-beta*y(1)+y(2)*y(3);
dy(2)=-rho*y(2)+rho*y(3);
dy(3)=-y(1)*y(2)+sigma*y(2)-y(3);
```

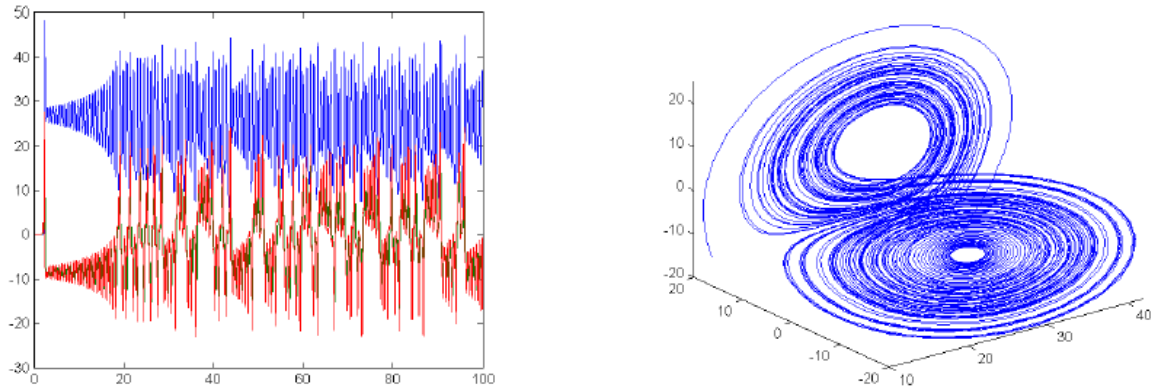

Figure 3: comparison between computede and exact solution

5.2 Converting n^{th} Order Ode to a System of Equations

Consider a general form of n^{th} order ordinary differential equation given by

$$y^{(n)} = f(t, y, y', y'', ...y^{n-1})$$

$$\text{with initial condition } y(0), y'(0)...y^(n-1)(0)$$

The ODE solver available in MATLAB can only deal with first order explicit differential equations thus. before solving a higher order ODE. then one has to transform it into a set of first order ODEs as follows set.

$u_1 = y$, $u_2 = y', ..., u_n = y^{n-1}$ and will be converted to first order explicit ODEs

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= u_3 \\ &\vdots \\ u_1' &= f(t, u_1, u_2, ..., u_n) \end{aligned}$$

With initial condition $u_1(0) = y_1(0), u_2(0) = y_0', ..., u_n(0) = y^{(n-1)}(0)$

Example 3.

Plot the solution of the initial value problem

$$y'' + yy' + y = 0, \qquad y(0)=0, \quad y'(0) = 1$$

on the interval [0.10] therefore

$$y' = -yy' - y$$

introducing the new variable for y and y'

$$u_1 = y \qquad u_2 = y'$$

then we have

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= -u_1 u_2 - u_1 \end{aligned}$$

Solution:

MATLAB code

```
function initialvprob
[t,u] = ode45(@yprime,[0 10],[0,1]);
```

```
plot(t,u(:,1))
title('y'''' + yy'' + y = 0, y(0) = 0, y''(0) = 1')
xlabel('t'), ylabel('y'), grid
function du = yprime(t,u)
du = zeros(2,1);
du(1) = u(2);
du(2) = -u(1)*u(2) - u(1);
```

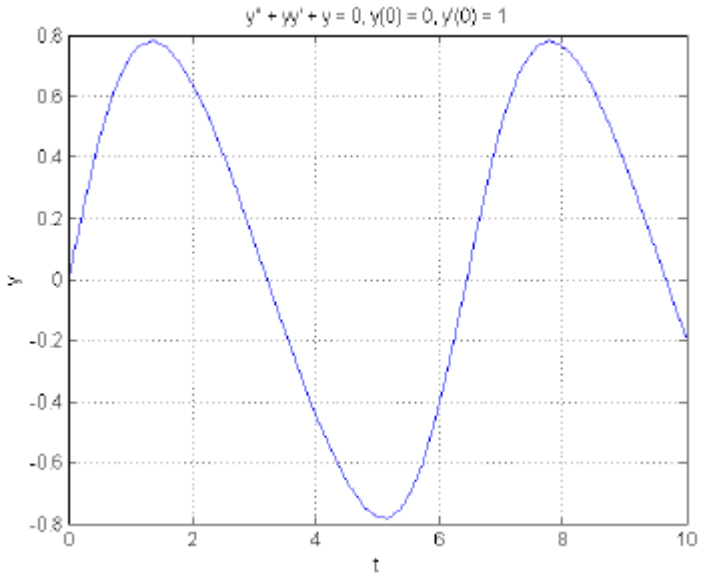

Figure 4: Solution of second order ODE

4.3 System for Higher Order Coupled Equation

Consider the set of higher order couple differential equation given as

$$x^{(m)} = f(t, x, x' ..., x^{(m-1)}, y, y', ..., y^{n-1})$$
$$y^{(m)} = g(t, x, x' ..., x^{(m-1)}, y, y', ..., y^{n-1})$$

set the variable $z_1 = x, z_2 = x', ..., z_m = x^{(m-1)}$ and $z_{m+1} = y, z_{m+2} = y', ..., z_{m+n} = y^{(n-1)}$.Thus, the original higher-order coupled ODEs can be converted to.

$$\begin{aligned} z_1' &= x_2 \\ &\vdots \\ z_m' &= f(t, z_1, z_2, ..., z_{m+n}) \\ z_{m+1}' &= z_{m+2} \\ &\vdots \\ z_{m+n}' &= g(t, z_1, z_2, ..., z_{m+n}) \end{aligned}$$

Which is desirable form to use solvers.
Finally, note that there are many different ways of coding the same thing. Play about with MATLAB and you will probably discover multiple ways of doing the same thing, some more efficient or easier than others.

Example 4.

Consider an example of motion of spacecraft in the gravitational field of the earth and the moon. The governing equations are given by

$$x'' = 2y' + x - \frac{E(x+M)}{r_1^3} - \frac{M(x-E)}{r_2^3}$$
$$y'' = -2x' + y - \frac{Ey}{r_1^3} - \frac{My}{r_2^3}$$

Where $r_1 = \sqrt{(x+M)^2 + y^2}$, $r_2 = \sqrt{(x-E)^2 + y^2}$ and E = 1 - M. For simplification, it has been assumed that the earth does not move and is located at (-M,0) and the moon is at (E,0). the initial values are given as $x(0) = 1.15$, $\frac{dx}{dt(0)} = 0$, $y(0) = 0$, and $\frac{dy}{dt} = 0.008688$. Let us introduce following auxiliary function

$$z_1 = x,\ z_2 = x',\ z_3 = y,\ z_4 = y',$$

This yields

$$\begin{pmatrix} z_1' \\ z_2' \\ z_3' \\ z_4' \end{pmatrix} = \begin{pmatrix} x' \\ x'' \\ y' \\ y'' \end{pmatrix} = \begin{pmatrix} z_2 \\ 2z_4 + z_1 - \frac{E(z_1+M)}{r_1^3} - \frac{M(z_1-E)}{r_2^3} \\ z_4 \\ -2z_2 + z_3 - \frac{Ez_3}{r_1^3} - \frac{Mz_3}{r_2^3} \end{pmatrix}$$

Where $r_1 = \sqrt{(z_1+M)^2 + z_3^2}$, $r_2 = \sqrt{(z_1-E)^2 + z_3^2}$

initial values are$z_1(0), z_2(0), z_3(0), z_4(0), = [1.15, 0, 0, 0.008688]$

Solution:

MATLAB code.

```
function spacecraft
M=0.012277; E=1-M;
tspan=[0 24];
z0=[1.15,0,0,0.008688];
[t,z]=ode23(@zprime, tspan, z0);
%phase plot with local moon and earth
subplot(211); plot(-M, 0, 'ko', E, 0, 'ko')
hold on
plot(z(:,1), z(:,3));
%for animated plot
subplot(212); comet(z(:,1),z(:,3));
function dz = zprime(t,z)
M = 0.012277;  E = 1-M;
%to simplefy the notation
z1 = z(1); z2 = z(2); z3 = (3); z4 = z(4);
R1 = sqrt((z1 + M)^2 + (z3)^2);
R2 = sqrt((z1 - E)^2 + (z3)^2);
dz = zeros(4,1);
dz(1) = z2;
dz(2) = 2*z4 + z1 - E*(z1+M)/R1^3 - M*(z1-E)/R2^3;
dz(3) = z4;
dz(4) = -2*z2 + z3-E*z3/R1^3-M*z3/R2^3;
```

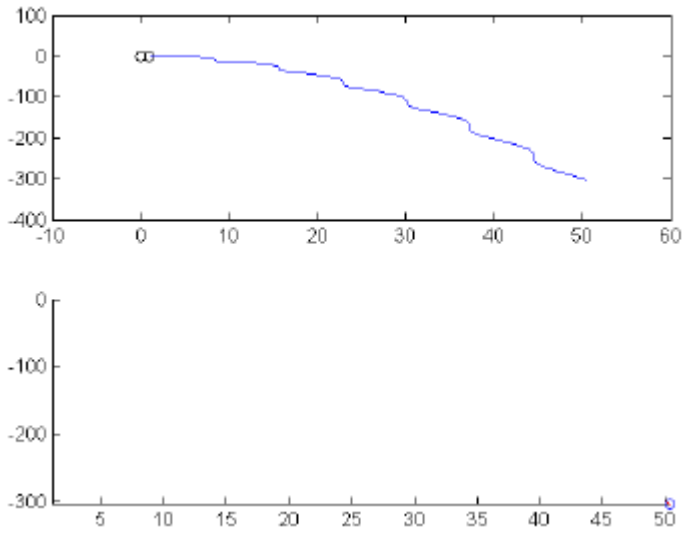

Figure 5: Solution y(t) of proton transfer problem, semilogx plot.

4.4 Sti Equations

In many differential equation, some variable changed very rapidly while other change very slowly. This type of differential equations is usually referred to as stiff equation.The term "stiff" as applied to ODE's does not have a precise definition. Loosely, it means that there is a very wide range between the most rapid and least rapid (with) changes in solution components. For such type of differential equations the solver ode45 may not be suitable, an alternative solver ode15s may be preferred. To understand the stiffness is to make the following observation.

1. From the explicit Runge-Kutta and Adams-Bashforth methods or other method, when the require much smaller step for the solution accuracy then the system is probably stiff.
2. When the height order performs even more poorly than the low order method then this problem are called stiff ODE

5.4.1 Sti Problems and the Choice of Solver

By a stiff ODE we mean an ODE for which numerical errors compound dramatically over time. For example, consider the ODE.

$$y^{'} = 100y + 100t + 1; y(0) = 1$$

Since the dependent variable, y, in the equation is multiplied by 100, small errors in our approximation will tend to become magnified. In general, we must take considerably smaller steps in time to solve stiff ODE, and this can lengthen the time to solution dramatically. Often, solutions can be computed more efficiently using one of the solvers designed for stiff problems.

Example 9.

The Robertson ODE system The Robertson ODE system

$y_1' = -0.04y_1 + 10^4 y_2 y_3, \quad y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2, \quad y_3 = 3 \times 10^7 y_2^2$

Models a reaction between three chemicals, for $0 \leq t \leq 3$ with initial condition [1,0,0]

Solution:

MATLAB code.

```
function chem1
tspan = [0 3]; yzero = [1;0;0];
tic, [ta,ya] = ode45(@chem,tspan,yzero); toc
subplot(121), plot(ta,ya(:,2),'-*')
ax = axis;
ax(1) = -0.2; axis(ax) % Make initial transient clearer.
xlabel('t'), ylabel('y_2(t)'), title('ode45','FontSize',14)
tic, [tb,yb] = ode15s(@chem,tspan,yzero); toc
```

```
subplot(122), plot(tb,yb(:,2),'-*'), axis(ax)
xlabel('t'), ylabel('y_2(t)'), title('ode15s','FontSize',14)
function yprime = chem(t,y)
%CHEM Robertson's chemical reaction model.
% YPRIME = CHEM(T,Y).
yprime = [-0.04*y(1) + 1e4*y(2)*y(3);
0.04*y(1) - 1e4*y(2)*y(3) - 3e7*y(2)^2;
3e7*y(2)^2];
```

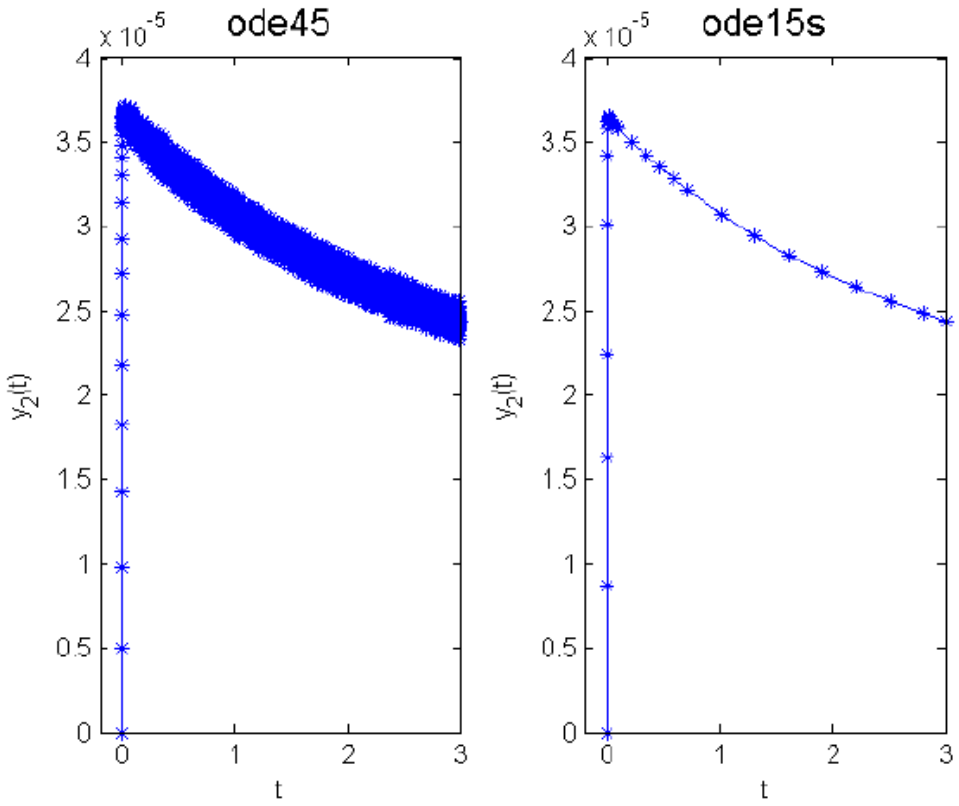

Figure 6: Chemical reaction solutions. Left: ode45. Right: ode15s.

Solver	produce the result in seconds	Time-step requires
ode45	0.609172	8209
ode15s	0.383132	34

The solutions agree to within a small absolute tolerance (note the scale factor 10^{-5} for the y-axis labels). However, the left-hand solution from ode45 has been returned at many more time values than the right-hand solution from ode15s and seems to be less smooth. To emphasize these points, plots ode45's $y_2(t)$ for $2 : 0 \leq t \leq 2.1$. We see that the t values are densely packed and spurious oscillations are present at the level of the default absolute error tolerance, 10^{-6}. The Robertson problem is a classic example of a stiff ODE.

Many solvers behave inefficiently on stiff ODEs they take an unnecessarily large number of intermediate steps in order to complete the integration and hence make an unnecessarily large number of calls to the ODE function (in this case, chem). We can obtain statistics on the computational cost of the integration by setting.

The behavior of ode45 typifies what happens when an adaptive algorithm designed for non stiff ODEs operates in the presence of stiffness. The solver does not break down or compute an inaccurate solution, but it does behave non smoothly and extremely inefficiently in comparison with solvers that are customized for stiff problems. This is one reason why MATLAB provides a suite of ODE solvers.

V.COMPARISON OF SOLVERS

How to use ode45 instead of ode23 or ode15s. This program displays the solution to the screen as it is computed and displays some statistics at the end of the run, we will find that the solution increases slowly from its initial value of tspan, at a time the reactant ignites and increases rapidly to a value near some point. This increase takes place in an interval for the remainder of the interval of integration, the solution is very near to its limit of point.

Modify the program to solve the IVP with the solver ode45 based on an explicit RungeKutta pair: all you must do is change the name of the solver, you will find that the numerical integration stalls after ignition, despite the fact that the solution is very nearly constant then, to quantify the difference, use tic and toc to measure the run times of the two solvers over the whole interval and over the first half of the interval we will find that the non stiff solver ode45 is rather faster on the first half of the interval because of the superior accuracy of its formulas this despite the minimal linear algebra costs in ode15s due to the ODE having only one unknown. You will find that the stiff solver ode15s is much faster on the whole interval because the explicit RungeKutta formulas of ode45 must use a small step size (to keep the integration stable in the last half of the interval) and the BDFs of ode15s do not. The statistics show that the RungeKutta code has many failed steps in the second half of the integration, this is typical when solving a stiff problem with a method that has a finite stability region. so the IVP can be stiff only on long intervals, in the first part of the integration the solution is positive, slowly varying

ode23 is a three-stage, third-order, Runge-Kutta method. ode45 is a six-stage, fifth-order, Runge-Kutta method. ode45 does more work per step than ode23, but can take much larger steps. For differential equations with smooth solutions, ode45 is often more accurate than ode23. In fact, it may be so accurate that the interpolant is required to provide the desired resolution.

5.1 MATLAB Code and Time in Second, Time-Step Required

Consider example 1 above

```
function solvesimpleode2
tic,[T,Ycompute]=ode45(@yprime,[0,20],0); toc
length(T)
tic,[T1,Y1compute]=ode23(@yprime,[0,20],0); toc
length(T1)
%code of comparison between computede and exact solution
Yexact=yexact(T); Yerror=abs(Ycompute-Yexact);
Y1exact=yexact(T1); Y1error=abs(Y1compute-Y1exact);
figure (1); plot(T,Ycompute, '-b*', T, yexact(T), '-r' )
title('Analytic solution and computed solution'); xlabel('x-axis'); ylabel('y-axis')
legend('Analytic solution',' computed solution')
figure (2); plot(T,Yerror, '*-r' )
title('Analytic solution and Error'); xlabel('x-axis'); ylabel('y-axis')
legend('Error')
figure (3); plot(T1,Y1compute, '-b*', T1, yexact(T1), '-r' )
title('Analytic solution and computed solution'); xlabel('x-axis'); ylabel('y-axis')
legend('Analytic solution',' computed solution')
figure (4); plot(T1,Y1error, '*-r' )
title('Analytic solution and Error'); xlabel('x-axis'); ylabel('y-axis')
legend('Error')
%code for yprime %use as subfunction
function yprime=yprime(t,y)
yprime=exp(-y);
%code for yexact %use as subfunction
function yexact=yexact(t)
yexact=log(t+1);
```

Solver	produce the result in seconds	Time-step requires
ode45	0.162406	56
ode23	0.091169	25

5.2 MATLAB Code and Time in Second, Time-Step Required

Consider example 2 above

```
function lorentz12
global beta rho sigma
rho=10;beta=8/3;sigma=28;
y0=[0 0 1e-10];
tic, [T Y]=ode45(@loren,[0 100],y0); toc
figure(1); plot(T,Y)
figure(2); plot3(Y(:,1),Y(:,2),Y(:,3))
axis([10 42 -20 20 -20 25])
figure(3); comet3(Y(:,1),Y(:,2),Y(:,3))
length(T)
tic, [T2 Y2]=ode45(@loren,[0 100],y0); toc
figure(1); plot(T2,Y2)
figure(2); plot3(Y2(:,1),Y2(:,2),Y2(:,3))
axis([10 42 -20 20 -20 25])
figure(3); comet3(Y2(:,1),Y2(:,2),Y2(:,3))
```

```
length(T2)
tic, [T1 Y1]=ode15s(@loren,[0 100],y0); toc
figure(1); plot(T1,Y1)
figure(2); plot3(Y1(:,1),Y1(:,2),Y1(:,3))
axis([10 42 -20 20 -20 25])
figure(3); comet3(Y1(:,1),Y1(:,2),Y1(:,3))
length(T1)
function dy=loren(t,y)
global beta rho sigma
dy=zeros(3,1);
dy(1)=-beta*y(1)+y(2)*y(3);
dy(2)=-rho*y(2)+rho*y(3);
dy(3)=-y(1)*y(2)+sigma*y(2)-y(3);
```

Solver	produce the result in seconds	Time-step requires
ode45	0.229720	5601
ode23	0.203336	5601
ode15s	0.003357	12

VI. EXPLICIT FORMULAS FOR STI SYSTEMS

6.1 The ode45 Program

Ode45 introduced in the late 1990s is based on an algorithm of Dormand and Prince. It uses six stages, employs the FSAL(first same as last) strategy, provides fourth and fifth order formulas, has local extrapolation and a companion interpolant.

The Dormand-Prince method has seven stages, but it uses only six function evaluations per step because it has the FSAL(first same as last)property, the last stage is evaluated at the same point as the first stage of the next step. Dormand and Prince chose the coefficient of their method to minimize the error of the fifth-order solution. This is the main difference with the Fehlberg method, which was constructed so that the fourth-order solution has a small error. for this reason, Dormand-Prince is more suitable when the higher-order solution is used to continue the integration, a practice known as local extrapolation and solution components can change substantially in the course of a single step, the values computed at the end of each natural step may not provide adequate resolution for graphical display of the solution.

This is remedied by computing intermediate values by interpolation, specifically by computing four values spaced evenly within the span of each natural step.

6.2 The ode23 Program

The new version of ode23 is based on the Bogacki- Shampine (2,3) pair [3] (see also [37]). This FSAL pair was constructed for local extrapolation. In the standard measures, the pair is of high quality and significantly more efficient than the pair used in the previous version of ode23. Accurate solution values can be obtained throughout a step for free by cubic Hermite interpolation to the values and slopes computed at the ends of the step.

At the default tolerances ode23 is generally more expensive than ode45, but not by a great deal, and it is to be preferred at cruder tolerances. The advantage it enjoys at crude tolerances is largely because a step in ode23 is about half as expensive as a step in ode45, hence the step size is adjusted more often. When a step fails in ode23, fewer evaluations of F are wasted. This is particularly important in the presence of mild stiffness because of the many failed steps then. The stability regions of the (2,3) pair are rather bigger than those of the (4,5) pair when scaled for equal cost, so ode23 is advantageous in the presence of mild stiffness.

VII IMPLICIT FORMULAS FOR STI SYSTEMS

7.1 The ode15s Program

The ode15s code is a quasi-constant step size implementation in terms of backward differences of the Klopfenstein-Shampine family of NDF's, but some options apply only to the codes for stiff problems or even only to this particular code. Therefore the derivation of the NDF's, it is easy to accommodate the BDF's in this framework. The user is, then, given the option of integrating with the classic BDF's rather than the default choice of the NDF's. then also, the user can reduce the maximum order from the default value of 5, should this appear desirable for reasons of stability. To implement the NDF formula, we rewrite the Klopfenstein form to make its evaluation efficient.

VIII. IMPLEMENTATION OF RUNGE-KUTTA-FEHLBERG METHOD

Before today's version of ode45, there was an earlier one. In a 1969, Erwin Fehlberg introduced called six stage Runge-Kutta method that requires six function evaluations per step. These function values can be combined with one set of coefficients to produce a fifth-order accurate approximation and with another set of coefficients to produce an independent fourth-order accurate approximation. Comparing these two approximations provides an error estimate and resulting step size control and it takes six stages to get fifth order. It is not possible to get fifth order with only five function evaluations per step.

In the early 1970's Shampine and his colleague H. A. (Buddy) Watts at Sandia Laboratories published a Fortran code, RKF45, based on Fehlberg's algorithm. In 1977, made RKF45 the ODE solver in text book (Computer Methods for Mathematical Computations, by Forsythe, Malcolm and Moler). a link to it is *Fortransourcecode* for RKF45 is still available from netlib. RKF45 became the basis for the first version of ODE45 in MATLAB in the early 1980s and for early versions of Simulink. The Felhberg (4,5) pair did a terrific job for almost fifteen years until the late 1990s when Shampine and MathWorker Mark Reichelt modernized the suite and introduced a more efficient algorithm.

Given the system of *ode's* below can be solve using the Runge-Kutta-Fehlberg Method

$$\begin{aligned} y_1' &= 10y_5 - 5y_1^{0.5} \\ y_2' &= 5y_1^{0.5} - 10y_2^{0.5} \\ y_3' &= 2y_2^{0.5} - 1.25y_3^{0.5} \\ y_4' &= 8y_2^{0.5} - 5y_4^{0.5} \\ y_5' &= 0 \end{aligned}$$

The implementation code is:

```
function sadis1
y0=[2, 0.25, 0.64, 0.64, 0.5];
[T45,Y45]=ode45(@sadis,[0,10],y0);
figure(1);
plot(T45,Y45,'.')
title('Computed solution of Ode45 System')
xlabel('t-axis')
ylabel('y-axis')
legend('Y1','Y2','Y3','Y4','Y5')
axis tight
function dy=sadis(t,y)
dy=zeros(5,1);
dy(1)= 10*y(5)- 5*y(1)^0.5;
dy(2)= 5*y(1)^0.5 - 10*y(2)^0.5;
dy(3)= 2*y(2)^0.5 - 1.25 *y(3)^0.5;
dy(4)= 8*y(2)^0.5 - 5*y(4)^0.5;
dy(5)=0;
```

The solution from the common window how the values of independent variables are changing from the initial vector condition.

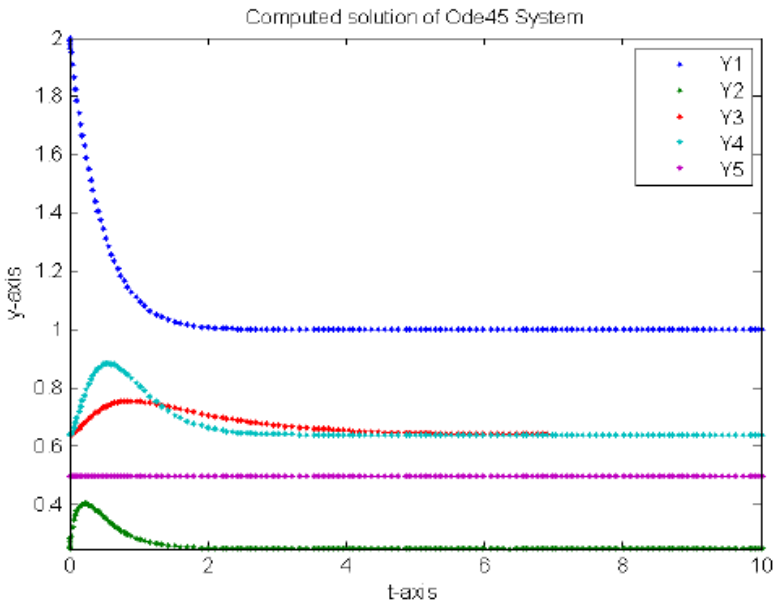

Figure 7: Solution of system of ODE's as Runge-kutta-fehlberg method

IX. CONCLUSION

1 In this dissertation, we have discussed different methods for solving first order, second order ordinary differential equations and systems of ordinary differential equations upto the higher order using build-in solvers in MATLAB.

2 Ode23 is a three-stage, third-order, Runge-Kutta method. ode45 is a six-stage, fifth-order, Runge-Kutta method. ode45 does more work per step than ode23, but can take much larger steps. For differential equations with smooth solutions, ode45 is often more accurate than ode23, it may be so accurate that the interpolant is required to provide the desired resolution. ode45 is the anchor of the differential equation suite. The MATLAB documentation recommends ode45 as the first choice. And Simulink blocks set ode45 as the default solver.

3 Some necessary conditions, algorithms used in build-in solvers, codes in MATLAB, implementation of build-in solvers and definitions are given to examine the numerical solution graphically. After that, by considering these conceptions and definitions the numerical results suggest that, the ode15s solver is suitable for solving stiff problems and perform competitively with the BDF method. The ode15s solver is faster as shown in table 3.5 and 5.6 because the equation is stiff. The method has shown the efficiency in terms of execution time and maximum and in all the table discussed for non-stiff equation, ode23 faster in producing the results in seconds and in some equation of non stiff ode15s has lest time-step than other solvers.also in table 3.6 ode45 is failed to some value of μ, therefore ode15s if the best solver to apply for stiff equations.

4 The implementation of Runge-kutta-fehlberg method adapts the number of position of the grid point during the course of the iteration in attempt to keep the local error withing some specified bound, the Runge-kutta-fehlberg method is an example of adapting time stepping method. it use fourth-order and fifth-order Runge-kutter method that share some valuation of $f(t, y)$, in order to reduce the number of evaluations of f per time step to six.

5 Finally, section (4) contains very useful descriptions of all of the build-in solvers used in the MATLAB (program)

REFERENCES

[1] Behind and Beyond the MATLAB ODE Suite.Ryuichi Ashino Michihiro Nagase Remi Vaillancourt CRM-2651 January 2000 (PAPER)

[2] Bindel, Spring 2012 Intro to Scientific Computing (CS 3220). Week 13: Wednesday, Apr 25 (PAPER)

[3] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," Appl. Math. Letters, Vol. 2, 1989, pp. 321325.

[4] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," J. Comp. Appl. Math., Vol. 6, 1980, pp. 1926.

[5] Elementary mathematics and computational tools for electrical and computer engineering using Matlab Jamal T. Manassah City College of New York.

[6] MATLAB Guide1 Desmond J. Higham and Nicholas J. Higham Version of June 27, 2000

[7] Numerical Analysis. Richard L.Burden J.Douglas faires

[8] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," SIAM Journal on Scientific Computing, Vol. 18, 1997, pp. 122.

[9] Shampine, L. F. , Numerical Solution of Ordinary Differential Equations, Chapman and Hall, New York, 1994.

[10] Solving ODEs with MATLAB. L.F.SHAMPINE, I.GLADWELL, S.THOMPSON

[11] The MATLAB ODE suite. Lawrence F.Shampine and Mark W. Reichelty