# Applications of MATLAB: **Ordinary Differential Equations (ODE)**

David Houcque
Robert R. McCormick School of Engineering and
Applied Science - Northwestern University
2145 Sheridan Road
Evanston, IL 60208-3102

**Abstract**

Textbooks on differential equations often give the impression that most differential equations can be solved in closed form, but experience does not bear this out. It remains true that solutions of the vast majority of first order initial value problems cannot be found by *analytical* means. Therefore, it is important to be able to approach the problem in other ways. Today there are numerous methods that produce numerical approximations to solutions of differential equations. Here, we introduce the oldest and simplest such method, originated by Euler about 1768. It is called the *tangent line method* or the *Euler method*. It uses a fixed step size $h$ and generates the approximate solution.

The purpose of this paper is to show the details of implementing a few steps of Euler's method, as well as how to use *built-in* functions available in MATLAB (2005) [1]. In the first part, we use Euler methods to introduce the basic ideas associated with initial value problems (IVP). In the second part, we use the Runge-Kutta method presented together with the built-in MATLAB solver `ODE45`. The implementations that we develop in this paper are designed to build intuition and are the first step from textbook formula on ODE to production software.

**Key words**: Euler's methods, Euler forward, Euler modified, Euler backward, MATLAB, Ordinary differential equation, ODE, ode45.

## 1 Introduction

The dynamic behavior of systems is an important subject. A mechanical system involves displacements, velocities, and accelerations. An electric or electronic system involves voltages, currents, and time derivatives of these quantities. An equation that involves one or more derivatives of the unknown function is called an *ordinary differential equation*, abbreviated as ODE. The *order* of the equation is determined by the order of the highest derivative. For example, if the first derivative is the only derivative, the equation is called a first-order ODE. In the same way, if the highest derivative is second order, the equation is called a second-order ODE.

The problems of solving an ODE are classified into *initial-value problems* (IVP) and *boundary-value problems* (BVP), depending on how the conditions at the endpoints of the domain are specified. All the conditions of an initial-value problem are specified at the initial point. On the other hand, the problem becomes a boundary-value problem if the conditions are needed for both initial and final points. The ODE in the time domain are initial-value problems, so all the conditions are specified at the initial time, such as $t = 0$ or $x = 0$. For notations, we use $t$ or $x$ as an independent variable. Some literatures use $t$ as time for independent variable.

It is important to note that our focus here is on the practical use of numerical methods in order to solve some typical problems, not to present any consistent theoretical background. There are many excellent and exhaustive texts on these subjects that may be consulted. For example, we would recommend Edwards and Penny (2000) [2], Boyce and DiPrima (2001) [3], Coombes et *al.* (2000) [4], Van Loan (1997) [5], Nakamura (2002) [6], Moler (2004) [7], and Gilat (2004) [8].

## 2 Numerical methods

Numerical methods are commonly used for solving mathematical problems that are formulated in science and engineering where it is difficult or even impossible to obtain exact solutions. Only a limited number of differential equations can be solved analytically. Numerical methods, on the other hand, can give an approximate solution to (almost) any equation. An ordinary differential equation (ODE) is an equation that contains an *independent* variable, a *dependent* variable, and *derivatives* of the dependent variable. Literal implementation of this procedure results in Euler's method, which is, however, not recommended for any practical use. There are other methods more sophisticated than Euler's. Among them, there are three major types of practical numerical methods for solving initial value problems for ODEs: (i) Runge-Kutta methods, (ii) Burlirsch-Stoer method, and (iii) predictor-corrector methods. We will present these three approaches on another occasion. Now, we are interested to talk about Euler's methods.

### 2.1 EULER methods

The Euler methods are simple methods of solving first-order ODE, particularly suitable for quick programming because of their great simplicity, although their accuracy is not high. Euler methods include three versions, namely,

- forward Euler method
- modified Euler method
- backward Euler method

### 2.1.1 Forward Euler method

The forward Euler method for $y' = f(y, x)$ is derived by rewriting the forward difference approximation,

$$(y_{n+1} - y_n)/h \approx y'_n \tag{1}$$

to

$$y_{n+1} = y_n + hf(y_n, x_n) \tag{2}$$

where $y'_n = f(y_n, x_n)$ is used. In order to advance time steps, Eq. 2 is recursively applied as

$$\begin{aligned}
y_1 &= y_0 + hy'_0 \\
y_1 &= y_0 + hf(y_0, x_0) \\
y_2 &= y_1 + hf(y_1, x_1) \\
y_3 &= y_2 + hf(y_2, x_2) \\
&\vdots \\
y_n &= y_{n-1} + hf(y_{n-1}, x_{n-1})
\end{aligned} \tag{3}$$

### 2.1.2 Modified Euler method

First, the modified Euler method is more accurate than the forward Euler method. Second, it is more stable. It is derived by applying the trapezoidal rule to the solution of $y' = f(y, x)$

$$y_{n+1} = y_n + \frac{h}{2}[f(y_{n+1}, x_{n+1}) + f(y_n, x_n)] \tag{4}$$

### 2.1.3 Backward Euler Method

The backward Euler method is based on the backward difference approximation and written as

$$y_{n+1} = y_n + hf(y_{n+1}, x_{n+1}) \tag{5}$$

The accuracy of this method is quite the same as that of the forward Euler method.

## 2.2 Steps for MATLAB implementation

The purpose of using an example is to show you the details of implementing the typical steps of Euler's method, so that it will be clear exactly what computations are being executed. For some reasons, MATLAB does not include Euler functions. Therefore, if you really need one, you have to code by yourselves. However, MATLAB has very sophisticated ones using Runge-Kutta algorithms. We will show how to use one of them in the next section.

### 2.2.1 Basic steps

The typical steps of Euler's method are given below.

**Step 1.** define $f(x, y)$

**Step 2.** input initial values $x_0$ and $y_0$

**Step 3.** input step sizes $h$ and number of steps $n$

**Step 4.** calculate $x$ and $y$:

```
for i=1:n
    x=x+h
    y=y+hf(x,y)
end
```

**Step 5.** output $x$ and $y$

**Step 6.** end

### 2.2.2 Example

As an application, consider the following initial value problem

$$\frac{dy}{dx} = \frac{x}{y}, \qquad y(0) = 1 \tag{6}$$

which was chosen *because* we know the analytical solution and we can use it for check. Its exact or analytical solution is found to be

$$y(x) = \sqrt{x^2 + 1} \tag{7}$$

Therefore, we will be able to compare the approximate solutions and the exact solution.

Here we wish to approximate $y(0.3)$ using the Euler's methods with step sizes $h = 0.1$ and $h = 0.05$. We find by hand-calculation,

$$\begin{aligned}
x_0 &= 0, \quad x_1 = 0.1, \quad x_2 = 0.2, \quad x_3 = 0.3 \\
y_0 &= 1 \\
y_1 &= y_0 + hf(x_0, y_0) = y_0 + hx_0/y_0 = 1 \\
y_2 &= y_1 + hx_1/y_1 = 1.01 \\
y_3 &= y_2 + hx_2/y_2 = 1.0298.
\end{aligned}$$

Since $y(0.3) = \sqrt{(0.3)^2 + 1} = 1.044030$, we find that

$$\text{Error} = \frac{|y(0.3) - y_3|}{y(0.3)} \times 100 = 1.36\%$$

Similarly, for the step size $h = 0.05$, we find that the error is

$$\text{Error} = \frac{|y(0.3) - y_6|}{y(0.3)} \times 100 = 0.67\%$$

### 2.2.3 MATLAB codes

- **Step 1**: *Create user-defined function files:* `euler_forward.m`, `euler_modified.m`, and `euler_backward.m`.

```
function [x,y]=euler_forward(f,xinit,yinit,xfinal,n)
% Euler approximation for ODE initial value problem
% Euler forward method
% File prepared by David Houcque - Northwestern U. 5/11/2005

% Calculation of h from xinit, xfinal, and n
h=(xfinal-xinit)/n;

% Initialization of x and y as column vectors
x=[xinit zeros(1,n)]; y=[yinit zeros(1,n)];

% Calculation of x and y
for i=1:n
    x(i+1)=x(i)+h;
    y(i+1)=y(i)+h*f(x(i),y(i));
end

end
```

```
function [x,y]=euler_modified(f,xinit,yinit,xfinal,n)
% Euler approximation for ODE initial value problem
% Euler modified method
% File prepared by David Houcque - Northwestern U. - 5/11/2005

% Calculation of h from xinit, xfinal, and n
h=(xfinal-xinit)/n;

% Initialization of x and y as column vectors
x=[xinit zeros(1,n)]; y=[yinit zeros(1,n)];
```

```
% Calculation of x and y
for i=1:n
    x(i+1)=x(i)+h;
    ynew=y(i)+h*f(x(i),y(i));
    y(i+1)=y(i)+(h/2)*(f(x(i),y(i))+f(x(i+1),ynew));
end

end


function [x,y]=euler_backward(f,xinit,yinit,xfinal,n)
% Euler approximation for ODE initial value problem
% Euler backward method
% File prepared by David Houcque - Northwestern U. - 5/11/2005

% Calculation of h from xinit, xfinal, and n
h=(xfinal-xinit)/n;

% Initialization of x and y as column vectors
x=[xinit zeros(1,n)];
y=[yinit zeros(1,n)];

% Calculation of x and y
for i=1:n
    x(i+1)=x(i)+h;
    ynew=y(i)+h*(f(x(i),y(i)));
    y(i+1)=y(i)+h*f(x(i+1),ynew);
end

end
```

- **Step 2:** *Solve the problem by putting data and called functions into a script file called* `main1.m`:

```
% Script file: main1.m
% The RHS of the differential equation is defined as
% a handle function
% File prepared by David Houcque - Northwestern U. - 5/11/2005

f=@(x,y) x./y;

% Calculate exact solution
g=@(x) sqrt(x.^2+1);
xe=[0:0.01:0.3];
ye=g(xe);
```

```
% Call functions
[x1,y1]=euler_forward(f,0,1,0.3,6);
[x2,y2]=euler_modified(f,0,1,0.3,6);
[x3,y3]=euler_backward(f,0,1,0.3,6);

% Plot
plot(xe,ye,'k-',x1,y1,'k-.',x2,y2,'k:',x3,y3,'k--')
xlabel('x')
ylabel('y')
legend('Analytical','Forward','Modified','Backward')
axis([0 0.3 1 1.07])

% Estimate errors
error1=['Forward error: ' num2str(-100*(ye(end)-y1(end))/ye(end)) '%'];
error2=['Modified error: ' num2str(-100*(ye(end)-y2(end))/ye(end)) '%'];
error3=['Backward error: ' num2str(-100*(ye(end)-y3(end))/ye(end)) '%'];

error={error1;error2;error3};
text(0.04,1.06,error)
```

- **Step 3:** *Compare the results.*
  The calculated results are displayed in the graphical form below. Reasonably good results are obtained even for a moderately large step size and the approximation can be improved by decreasing the step size. According to the results (Figure 1) and Table 1, forward and backward approaches give identically the same results (less than 1% of error), while modified method give very good result when compared with the exact solution.

| $h$ | **Forward** | **Modified** | **Backward** |
|---|---|---|---|
| 0.05 | 0.67% | 0.04% | 0.67% |

Table 1: Comparison of exact solution with Euler methods

## 2.3 Using built-in function

MATLAB has several different functions (built-ins) for the numerical solution of ordinary differential equations (ODE). In this section, however, we will present one of them. We will also give an example how to use it, instead of writing our own MATLAB codes as we did in the first part. The basic steps, previously defined, are still typically the same. These *solvers* can be used with the following syntax:

```
[x,y] = solver(@odefun,tspan,y0)
```

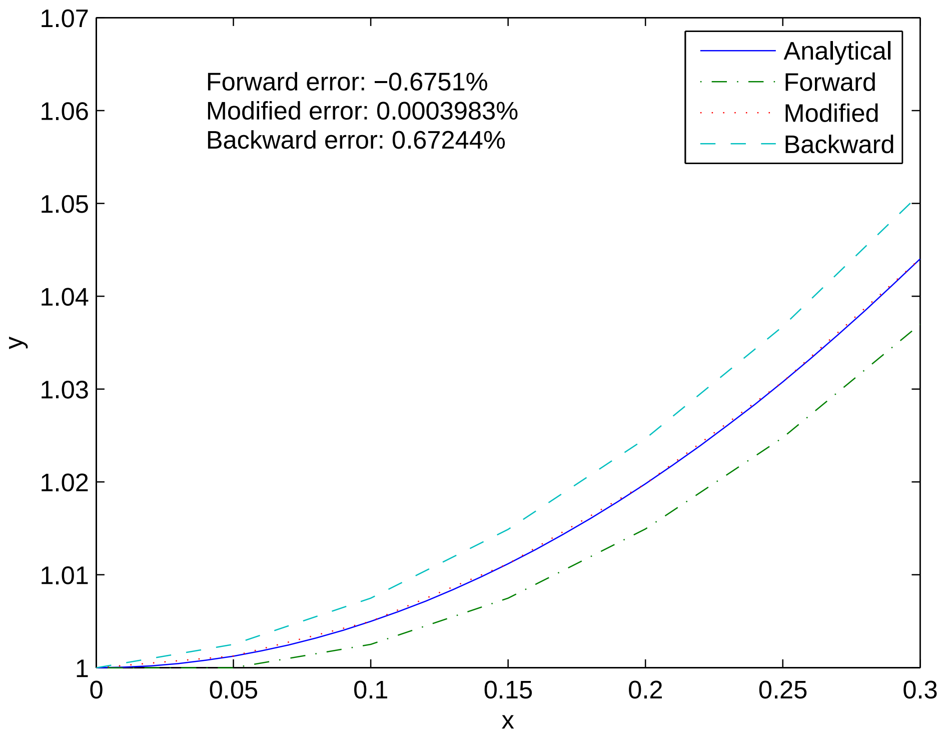

Figure 1: Comparison of exact solution with Euler methods

`solver` is the *solver* you are using, such as name, `ode45` or `ode23`. `odefun` is the function that defines the derivatives, so `odefun` defines $y'$ as a function of the independent parameter (typically $x$ or $t$) as well as $y$ and other parameters. `tspan` a vector that specifies the interval of the solution (e.g., `[t0,tf]`). `y0` is the initial value of $y$. `[x,y]` is the output, which is the solution of the ODE.

### 2.3.1 Runge-Kutta methods

There are many variants of the Runge-Kutta method, but the most widely used one is the following. Given:

$$\begin{aligned} y' &= f(x,y) \\ y(x_n) &= y_n \end{aligned} \tag{8}$$

we compute in turn

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{9}$$

### 2.3.2 Using ode45

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair [9]. That means the numerical solver `ode45` combines a *fourth* order method and a *fifth* order method, both of which are similar to the classical fourth order Runge-Kutta (RK) method discussed above. The *modified* RK varies the step size, choosing the step size at each step in an attempt to achieve the desired accuracy. Therefore, the solver `ode45` is suitable for a wide variety of initial value problems in practical applications. In general, `ode45` is the best function to apply as a "first try" for most problems. Table 2 lists ODE solvers, which are MATLAB built-in functions. A short description of each solver is included.

NOTE - It is important to note that in MATLAB 7.0 (R14), latest version, it is preferred to have `odefun` in the form of a function *handle*. For example, it is recommended to use the following syntax,

```
ode45(@xdot,tspan,y0)
```

rather than

| **Solver** | **Accuracy** | **Description** |
|---|---|---|
| `ode45` | Medium | This should be the first solver you try |
| `ode23` | Low | Less accurate than `ode45` |
| `ode113` | Low to high | For computationally intensive problems |
| `ode15s` | Low to medium | Use if `ode45` failed |

Table 2: Some MATLAB ODE solvers

```
ode45('xdot',tspan,y0)
```

Note the use of `@xdot` and `'xdot'`. Use function handles to pass any function that defines quantities the MATLAB solver will compute, in particular for simple functions.

On the other hand, it is also important to remember that complicated differential equations should be written an M-file instead of using *inline* command or function *handle*.

Following is an example of an ordinary differential equation using a MATLAB ODE solver. First, let's create a script file, called `main2.m`, as follows:

```
% Script file: main2.m
% The RHS of the differential equation is defined as
% a function handle.
% File prepared by David Houcque - Northwestern U. - 5/11/2005

f=@(x,y) x./y;

% Calculate exact solution
g=@(x) sqrt(x.^2+1);
xe=[0:0.01:0.3];
ye=g(xe);

% Call function
[x4,y4]=ode45(f,[0,0.3],1);

% Plot
plot(xe,ye,'k-',x4,y4,'k:')
xlabel('x')
ylabel('y')
legend('Analytical','ode45')
axis([0 0.3 1 1.05])
```

Here, we use the same data as defined in the first part for Euler's methods. The initial conditions and the time steps are the same as before.

The integration proceeds by *steps*, taken to the values specified in `tspan`. Note that the *step size* (the distance between consecutive elements of `tspan`) does not have to be uniform.

A plot comparing the computed $y$ versus $x$ is shown in Figure 2. According to the plot, the

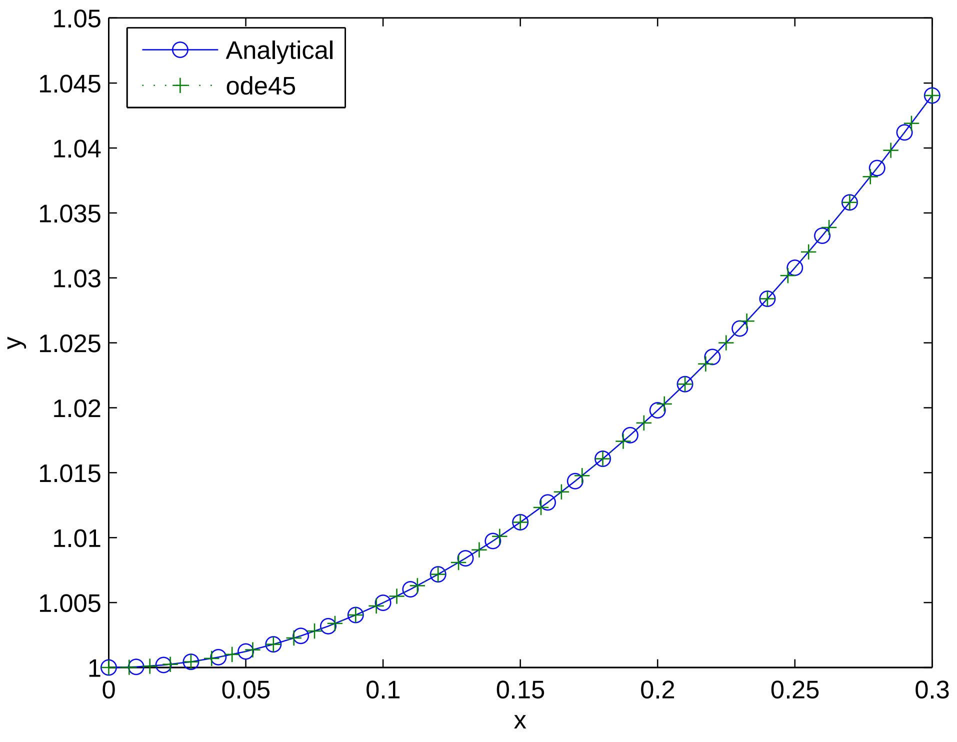

Figure 2: Comparison of exact solution with `ode45` solution

calculated results using the built-in function (`ode45`) give a very good result when compared with the analytical solution.

Additional on all of these solvers can be found in the online help. For addition information on numerical methods, we refer to Shampine (1994) [10] and Forsythe et *al.* (1977) [11].

## 3 Conclusion

The above plots show the results obtained from different algorithms. Consequently, we can see better that Runge-Kutta algorithm is even more accurate at large step size ($h = 0.1$) than Euler algorithms at small step size ($h = 0.05$). One can easily adapt these MATLAB codes as needed for a different type of problem. In using numerical procedure, such as Euler's method, one must always keep in mind the question of whether the results are accurate enough to be useful. In the preceding examples, the accuracy of the numerical results could be ascertained directly by a comparison with the solution obtained analytically. Of course, usually the analytical solution is not always available to compare.

**Acknowledgement-** The author is sincerely grateful to Associate Dean Steve Carr for his support. Thanks also go to Professor MacIver for his support and discussions. He is grateful to EA3 students (Honors Section) for their bright ideas and suggestions.

## References

[1] The MathWorks Inc. *MATLAB: SP2 R14*. The MathWorks Inc., 2005.

[2] C. H. Edwards and D. E. Penny. *Differential Equations and Boundary Value Problems: Computing and Modeling*. Prentice Hall, 2000.

[3] W. E. Boyce and R. C. DiPrima. *Elementary Differential Equations and Boundary Value Problems*. John Wiley and Sons, 2001.

[4] K. R. Coombes, B. R. Hunt, R. L. Lipsman, J. E. Osborn, and G. J. Stuck. *Differential Equations with MATLAB*. John Wiley and Sons, 2000.

[5] C. F. Van Loan. *Introduction to Scientific Computing*. Prentice Hall, 1997.

[6] S. Nakamura. *Numerical Analysis with MATLAB*. Prentice Hall, 2002.

[7] C. B. Moler. *Numerical Computing with MATLAB*. Siam, 2004.

[8] A. Gilat. *MATLAB: An introduction with Applications*. John Wiley and Sons, 2004.

[9] J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *J. Comp. Appl. Math*, 6:19–26, 1980.

[10] L. F. Shampine. *Numerical Solution of Ordinary Equations*. Chapman and Hall, 1994.

[11] G. Forsythe, M. Malcolm, and C. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, 1977.