Lecture `ode1`: ODE IVPs: introduction to difference methods

November 30, 2022

Summary: Introduction to the concepts of ODE initial value problems (IVPs). Explicit Euler and Trapezoid Methods for approximating solutions to IVPs.

References: T. Sauer's *Numerical Analysis*, Section 6.1, 1st ed. pages 284–295, 2nd ed. pages 282–291; some material here also developed by M. Nitsche.

1 Introduction

If you have taken Math 316 here at UNM, then you are familiar with ODE. The general ODE initial value problem (IVP) is the following:[1]

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0. \tag{1}$$

Here the ODE is the rule for evolving the initial state $\mathbf{y}_0$ forward in time to obtain the value $\mathbf{y}(t)$ of the solution at later times. The independent variable, here t, is not always time, but it often is, and we shall assume so here. Our chief interest is how to *numerically* solve (1), but before we discuss methods (sometimes called schemes) for doing so, we would like to know *theoretically* (i) whether a solution to (1) exists at all, and if so (ii) whether it is unique. Another question is (iii) whether the solution depends continuously on the initial data, the choice of $\mathbf{y}_0$. For certain mathematical problems such as our IVP, if the answer to all three questions is yes, then according to the definition of J. Hadamard the problem is *well-posed*. Although more is required to make (iii) a mathematically precise notion, the idea is that sufficiently small changes $\mathbf{y}_0 \to \mathbf{y}_0 + \delta\mathbf{y}_0$ in the initial data induce correspondingly small changes $\mathbf{y}(t) \to \mathbf{y}(t) + \delta\mathbf{y}(t)$ in the solution at later times: i.e. small $\delta\mathbf{y}_0$ implies small $\delta\mathbf{y}(t)$.

Analysis of these issues is rather involved. For most cases you will encounter the "right-hand side" $\mathbf{f}(t, \mathbf{y})$ and a number of its derivatives will be continuous on some open set containing the initial data point $(t_0, \mathbf{y}_0)$. For example, often at least $\partial f_j/\partial t$ and $\partial f_j/\partial y_k$ are continuous for all j, k on such an open set. For such "smooth" $\mathbf{f}(t, \mathbf{y})$, the IVP (1) is well-posed. For further reference, see [**?**].

1.0.1 Scalar examples

Let us first consider a scalar example:

$$y' = 1 - t + 4y, \qquad y(0) = 1. \tag{2}$$

[1] An important special case is

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0.$$

Here the right hand side $\mathbf{f}(\mathbf{y})$ does not depend on t. That is, the system is autonomous. *Autonomous* means *independent* or *having the power of self-government*. For an autonomous system the solution clearly "controls itself", since the "forcing" $\mathbf{f}(\mathbf{y})$ depends only on the solution itself and not explicitly on time.

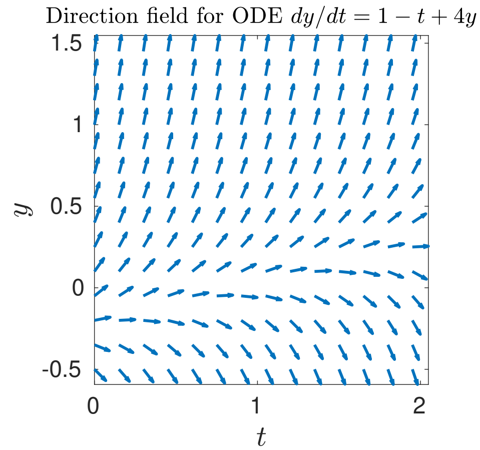

Figure 1: Direction field for $y' = 1 - t + 4y$.

Here we seek a function $y = \phi(t)$, sometimes expressed more simply as $y = y(t)$, which both solves the above differential equation and obeys the *initial condition* $\phi(0) = 1$. Using the method of integrating factors, you can solve the above problem, finding

$$y = \phi(t) = \tfrac{1}{4}t - \tfrac{3}{16} + \tfrac{19}{16}e^{4t}. \tag{3}$$

You may not be familiar with the solution process. However, once you are given the solution, verification that it indeed solves the problem is straightforward: (i) compute $\phi'(t)$ and compare it with $1 - t + 4\phi(t)$, and (ii) compute $\phi(1)$.

The problem (2) is a bit special in that the solution $y = \phi(t)$ exists for all time $t > 0$. Indeed, $\phi(t) = \frac{1}{4}t - \frac{3}{16} + \frac{19}{16}e^{4t}$ is clearly defined for all t, and solves the given ODE for all t. Therefore, given the problem (2), the questions "what is $y(1)$?" and "what is $y(100)$?" are sensible ones. Generally, we need to be more careful, since the solution to an IVP may not exist for all time. For example, the IVP

$$y' = y^2, \qquad y(0) = 1, \tag{4}$$

has the solution $y = \phi(t) = 1/(1-t)$, as can easily be checked. If we consider only non-negative times, then clearly the solution is only defined on $[0, 1)$. The question "what is $y(100)$?" does not make sense for the problem (4).

The **direction field** for the ODE (2) is shown in Fig. 1. At each point in the (t, y) plane a little line is drawn whose slope equals $1 - t + 4y$. We may put arrows on these lines, if we wish, to indicate the forward flow of time. A solution curve $y = \phi(t)$ is a curve in the plane which runs everywhere tangent to the arrows. Such solution curves are also called **integral curves**. Notice that the integral curve associated with $\phi(0) = 1$ leaves the plotted region quickly. The concept of a direction field will help motivate our numerical methods for ODE.

1.1 System example

Let us consider a system example:

$$\mathbf{y}' = \begin{pmatrix} 1 & 1 \\ 4 & -2 \end{pmatrix} \mathbf{y} + \begin{pmatrix} t \\ 0 \end{pmatrix}; \quad \mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{5}$$

Now the solution is a vector-valued function $\mathbf{y} = \boldsymbol{\phi}(t)$ of time, in this case with two components, $y_1 = \phi_1(t)$ and $y_2 = \phi_2(t)$. Without describing the solution process, we note that the solution is

$$\mathbf{y} = \boldsymbol{\phi}(t) = \begin{pmatrix} \phi_1(t) \\ \phi_2(t) \end{pmatrix} = \frac{1}{9} \begin{pmatrix} 9e^{2t} + 2e^{-3t} - 3t - 2 \\ 9e^{2t} - 8e^{-3t} - 6t - 1 \end{pmatrix}. \tag{6}$$

Verification that this is indeed the solution to (5) is straightforward, if algebraically tedious. The numerical methods that we consider work equally well for a system IVP, although we shall develop them with a scalar IVP in mind.

2 Two explicit difference methods

All of the methods we consider for (1) involve discretization of time by a sequence of $m+1$ discrete points t_k called *mesh points*. Note that m is not n, the system size of the ODE (i.e. $\mathbf{y}, \mathbf{y}_0, \mathbf{f} \in \mathbb{R}^n$). We take these points to be equally spaced,

$$t_k = a + k\Delta t, \quad k = 0, \dots m, \quad \Delta t = (b-a)/m, \tag{7}$$

where $a \equiv t_0$ and $b \equiv T$ are also used for the initial time t_0 and final time T. For the numerical approximation of the exact solution $\mathbf{y}(t)$ at $t = t_k$, we use

$$\mathbf{y}_k \simeq \mathbf{y}(t_k), \tag{8}$$

for $k = 0, \dots, m$. The approximation $\mathbf{y}_k$ will depend on Δt, and is therefore sometimes written as

$$\mathbf{y}_k^{\Delta t} \text{ (precise notation for numerical solution).} \tag{9}$$

Note, for example, that if $\Delta t = 0.25$, then $\mathbf{y}_8^{\Delta t}$ approximates $\mathbf{y}(2)$, whereas if $\Delta t = 0.125$, then $\mathbf{y}_8^{\Delta t}$ approximates $\mathbf{y}(1)$. We will make use of this precise notation below.

The methods we study compute the approximation $\mathbf{y}_k$ in terms of the approximation $\mathbf{y}_{k-1}$ at the previous time t_{k-1}. These methods for computing $\mathbf{y}_k$ are obtained by approximating the derivatives in the ODE as finite difference quotients; therefore, they are called *difference methods*. We shall also restrict our focus to *explicit methods*. For explicit methods, or "marching schemes", we have an explicit formula for $\mathbf{y}_k$ in terms of $\mathbf{y}_{k-1}$ and t_{k-1}. That is, no root-finding procedure is needed to compute $\mathbf{y}_k$.

Two types of errors are relevant for our discussion: *discretization* (or *truncation*) *error* resulting from approximation of the ODE, and *roundoff error* resulting from the fact that computations used to solve or evaluate the approximate equations are based on finite-precision arithmetic.

2.1 Big-O notation

Before turning to specific time stepping schemes, let us first introduce *big-O* notation. Let E be a function of $h = \Delta t$. We say that $E(h) = O(h^p)$ (read as "E is big-O of h to the p" or "E is of order h to the p") if there exists constants ϵ and C such that

$$|E(h)| < Ch^p, \qquad \forall h \in [0, \epsilon]. \tag{10}$$

This notation gives an upper bound on how fast a function approaches zero in the $h \to 0^+$ limit. If $E(h)$ is the error in a numerical method using time step $h = \Delta t$, then the larger p, the quicker the error approaches zero as $h \to 0^+$. For example, if $p = 1$, then the error is halved every time h is halved. But if $p = 2$ the error is reduced by a factor of 4 every time h is halved.

In the methods described below p is typically an integer. Why? Because the methods are obtained by truncating Taylor series about a basepoint. How do we check numerically how fast the error is decaying, and whether it looks like h^p for some p? One way is to compute the factor by which the error is reduced every time h is halved. A factor of 2 would imply $p = 1$, a factor of 4 would imply $p = 2$, *etc.* The h^p-behavior may only be seen asymptotically, that is in the small-h limit.

Another way is to plot $E(h)$ on a log-log scale. Suppose $E \approx Ch^p$ for $C > 0$, then

$$\log E \approx \log C + p \log h,$$

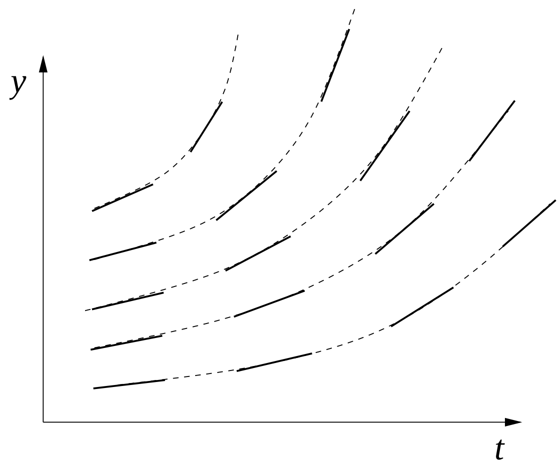

Figure 2: Slope field of $y' = f(t, y)$ depicted as short line segments on top of (dashed) integral curves.

and if you plot $\log E$ versus $\log h$ (e.g. with MATLAB's `loglog` command), then the plot looks approximately linear with slope p. You can estimate the slope by plotting nearby a function with known value p, for example by setting $y = \widetilde{C}h^p$ and choosing $\widetilde{C}$ (which gives you a vertical translation) so that the resulting line (on a log-log scale) is in a good position.

Here is an example where *round-off error* is pertinent. The difference quotient $(f(x+h) - f(x))/h$ is a first order approximation to $f'(x)$:

$$E(h) = \frac{f(t+h) - f(t)}{h} - f'(t) = O(h)$$

Plot this error as a function of h on a log-log scale for $f(t) = \sin t$ at $t = 1$, and $h = \texttt{logspace}(-20, 0, 21)$. Can you explain what you see?

2.2 Forward Euler Method

Consider the scalar problem

$$\frac{dy}{dt} = f(t, y), \qquad y(0) = y_0. \tag{11}$$

The differential equation determines a *direction field*. It tells you what the slope $y'(t)$ of $y(t)$ is at any point in the t-y plane. There are infinitely many *integral curves* tangent to this slope field. The *initial condition* $y(0) = y_0$ determines a unique one of these solution curves, see Fig. 2.

2.2.1 Method and example

Suppose you know the approximate solution y_k at a point t_k. To obtain y_{k+1}, Euler's method approximates the exact solution curve through (t_k, y_k) by the tangent to the curve at that point. That is, we move from (t_k, y_k) to (t_{k+1}, y_{k+1}) along a line with slope $f(t_k, y_k)$. See Fig. 3. From the picture, we see that the algorithm is

$$\frac{y_{k+1} - y_k}{\Delta t} = f(t_k, y_k) = \text{slope at } (t_k, y_k), \qquad k = 0, \ldots, m-1, \tag{12}$$

that is

$$y_{k+1} = y_k + \Delta t f(t_k, y_k), \qquad k = 0, \ldots, m-1. \tag{13}$$

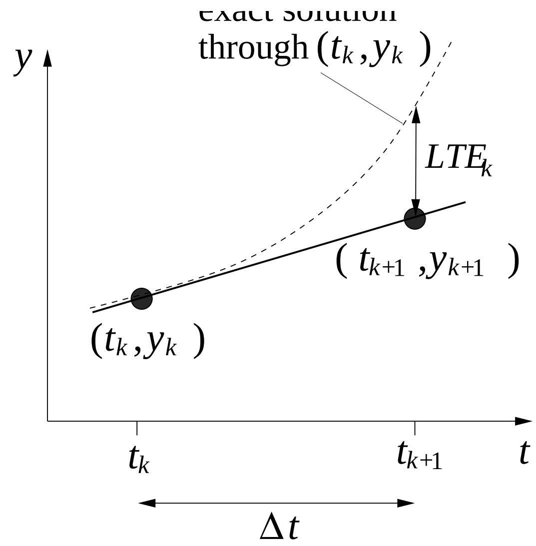

Figure 3: Advancing from t_k to t_{k+1} with Euler's method.

Together with the initial condition, this rule determines a sequence of values y_k for $k = 0, \ldots, m$. Actually, (12) is the *forward* Euler method. The *backward* Euler method is

$$\frac{y_{k+1} - y_k}{\Delta t} = f(t_{k+1}, y_{k+1}) = \text{slope at } (t_{k+1}, y_{k+1}), \qquad k = 0, \ldots, m-1. \tag{14}$$

The backward Euler method is not an explicit method, rather an *implicit* one: obtainment of y_{k+1} from the above formula typically involves root-finding. Our focus is on the forward Euler method, for which we have an explicit formula $y_{k+1} = y_k + \Delta t f(t_k, y_k)$ for y_{k+1}. For systems (1) of equations in which $\mathbf{y}$ is a vector, we simply advance each of the components of $\mathbf{y}$ in the same fashion. Thus, Euler's method (for one or higher-dimensional systems) is

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \Delta t f(t_k, \mathbf{y}_k), \qquad \mathbf{y}_0 \text{ given}, \qquad k = 0, \ldots, m-1, \tag{15}$$

where t_k are given by Eq. (7).

Here is a MATLAB function that uses the forward Euler method to solve (1) on an interval $t \in [t_0, t_F = \texttt{MAXTIME}]$, where t_F is the final time and the function $\mathbf{f}$ is passed with the `@` notation.

```
%  UNM CS/Math 375, Fall Semester 2022
%  function [y] = ForwardEuler(f,t0,MAXTIME,y0,nsteps)
   function [y] = ForwardEuler(f,t0,MAXTIME,y0,nsteps)
h = (MAXTIME-t0)/nsteps;
y = y0; t = t0;
for n = 1:nsteps
    y = y + h*f(t,y);
    t = t0 + n*h;
end
```

Let us use the Euler method to numerically integrate the problem (2), in order to compute an approximation to $y(2)$. First, we code a function for the righthand side of the ODE:

```
% function [f] = ExampleODERightSide(t,y)
  function [f] = ExampleODERightSide(t,y)
  f = 1 - t + 4*y;
```

To investigate how the error in the approximation depends on the step size $h = \Delta t = 2/\texttt{nsteps}$, we compute a sequence of approximations

$$y^{\Delta t}_{\texttt{nsteps}} = y^{2/\texttt{nsteps}}_{\texttt{nsteps}} \text{ for } \texttt{nsteps} = 4096, 8192, 16384, 32768, 65536,$$

and also the corresponding errors $|y(2) - y^{\Delta t}_{\texttt{nsteps}}|$. The relevant script follows.

```
% Script: TestEulerConvergence
t0 = 0; MAXTIME = 2; y0=1;
hs = transpose([1/4096 1/8192 1/16384 1/32768 1/65536]);
errs = zeros(size(hs));
t = MAXTIME; yExact = 0.25*t - 0.1875 + 1.1875*exp(4*t);
for k = 1:length(hs)
    h = hs(k);
    nsteps = round(MAXTIME/h);
    yNumerical = ForwardEuler(@ExampleODERightSide,t0,MAXTIME,y0,nsteps);
    errs(k) = abs(yNumerical-yExact);
end
errs
```

The output from this script

```
>> TestEulerConvergence
errs =

      13.792
      6.9049
      3.4547
      1.7279
     0.86409
```

shows that the forward Euler method is an order-1 method. Indeed, clearly the error appears to drop by a factor of $\frac{1}{2}$ each time we halve the step size (that is, double the number of time steps taken). To confirm, we compute the following ratios.

```
>> errs(2:end)./errs(1:end-1)
ans =

     0.50065
     0.50033
     0.50016
     0.50008
```

This is an empirical verification that $|y(2) - y^{2/\texttt{nsteps}}_{\texttt{nsteps}}| = O(h)$. We say that the forward Euler method has order of convergence 1.

2.2.2 Discretization error and order of convergence

Here we wish to justify *theoretically* that the the forward Euler method has order of convergence 1. Let ℓ mean "local" and consider the following initial value problem (IVP):

$$\frac{dy^\ell}{dt} = f(t, y^\ell), \qquad y^\ell(t_k) = y_k. \tag{16}$$

The ℓ here is, in some sense, superfluous; it simple serves to remind us that this IVP is started at time t_k, i.e. with the initial time taken as the kth timestep t_k rather than t_0 as is customary.

The *local truncation error* for the forward Euler method is (see Fig. 3)

$$\mathrm{LTE}_k \equiv y^\ell(t_{k+1}) - y_{k+1}, \tag{17}$$

where $y_{k+1} = y_k + \Delta t f(t_k, y_k)$ is the forward Euler step. Since in our notation $y^\ell(t_{k+1})$ is the *exact* solution to the IVP (16) at time t_{k+1}, the local truncation error is the error made in taking a single time step. Notice that

$$\begin{aligned} \mathrm{LTE}_k &= y^\ell(t_{k+1}) - y_k - \Delta t f(t_k, y_k) \\ &= y^\ell(t_{k+1}) - y^\ell(t_k) - \Delta t f(t_k, y^\ell(t_k)), \end{aligned} \tag{18}$$

with the last equality stemming from the initial condition in (16).

Using Taylor's theorem with quadratic remainder, we have

$$\begin{aligned} y^\ell(t_{k+1}) &= y^\ell(t_k + \Delta t) \\ &= y^\ell(t_k) + (y^\ell)'(t_k)\Delta t + \frac{1}{2}(y^\ell)''(\eta)\Delta t^2 \\ &= y^\ell(t_k) + f(t_k, y^\ell(t_k))\Delta t + \frac{1}{2}(y^\ell)''(\eta)\Delta t^2, \end{aligned} \tag{19}$$

where $\eta \in [t_k, t_{k+1}]$. Substitution of (19) into (18) yields

$$\mathrm{LTE}_k = \frac{1}{2}(y^\ell)''(\eta)\Delta t^2. \tag{20}$$

We now assume that $|(y^\ell)''(\eta)| \le M$ (constant) for all $\eta \in [t_k, t_{k+1}]$. Then $|\mathrm{LTE}_k| \le \frac{1}{2}M\Delta t^2$, so that $\mathrm{LTE}_k = O(\Delta t^2)$, which shows that the forward Euler method is *consistent*: $\mathrm{LTE}_k \to 0$ as $\Delta t \to 0^+$.

The local truncation error is the crime committed in taking one time step. To integrate from t_0 to $t_F = \texttt{MAXTIME}$ we take many steps, more precisely $\texttt{nsteps} = \texttt{MAXTIME}/\Delta t$. We therefore expect that the *global truncation error*, or total accumulated error will be

$$|y(t_F) - y^{t_F/\texttt{nsteps}}_{\texttt{nsteps}}| \lesssim \texttt{nsteps}(\tfrac{1}{2}M\Delta t^2) = \texttt{MAXTIME}(\tfrac{1}{2}M\Delta t) = O(\Delta t).$$

Careful analysis of the global truncation error confirms this result. In general we expect that a difference method will have order of convergence p if the local truncation error for the method is $O(h^{p+1})$.

2.3 Explicit trapezoid method

The forward Euler method is order-1. We now consider an order-2 explicit method: the explicit trapezoid method. If we integrate the ODE $y' = f(t, y)$ over an interval $[t_k, t_{k+1}]$ we get

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(s, y(s))ds.$$

This is an integral equation, and we might approximate this equation with a difference quotient on the left side and the trapezoid rule to handle the quadrature on the right side:

$$\frac{y_{k+1} - y_k}{\Delta t} = \frac{\Delta t}{2}\big[f(t_k, y_k) + f(t_{k+1}, y_{k+1})\big].$$

The problem with this formula is that y_{k+1} appears both on the left and right-hand sides. It is an *implicit equation* for y_{k+1}, for which one would typically need a root-finding algorithm to find y_{k+1}.

This *implicit trapezoid rule* is a viable ODE method, but we seek a simpler one. By simpler we mean an *explicit* one, that is a direct "marching scheme". Therefore, we replace the last formula with

$$\frac{y_{k+1} - y_k}{\Delta t} = \frac{\Delta t}{2}\left[f(t_k, y_k) + f(t_{k+1}, \overbrace{y_k + \Delta t f(t_k, y_k)}^{\text{forward Euler step}})\right].$$

This is an example of a *Runge-Kutta scheme*, here a two-stage one. The time step is taken as follows:

$$k_1 = f(t_k, y_k), \qquad k_2 = f(t_{k+1}, y_k + \Delta t k_1), \qquad y_{k+1} = y_k + \frac{\Delta t}{2}(k_1 + k_2). \tag{21}$$

This process is explicit, and involves no root-finding.

Here is a MATLAB function that uses the explicit trapezoid method to solve (1) on an interval $t \in [t_0, t_F = \texttt{MAXTIME}]$.

```
%  UNM CS/Math 375, Fall Semester 2022
%  function [y] = ExplicitTrapezoid(f,t0,MAXTIME,y0,nsteps)
   function [y] = ExplicitTrapezoid(f,t0,MAXTIME,y0,nsteps)
h = (MAXTIME-t0)/nsteps;
y = y0; t = t0;
for n = 1:nsteps
    k1 = f(t,y);
    k2 = f(t+h,y+h*k1);
    y = y + 0.5*h*(k1+k2);
    t = t0 + n*h;
end
```

We again consider approximation of $y(2)$ for the problem (2). The script `TestEulerConvergence` is modified to replace the forward Euler method with the explicit trapezoid method:

```
% Sctipt: TestExplicitTrapConvergence
t0 = 0; MAXTIME = 2; y0=1;
hs = transpose([1/4096 1/8192 1/16384 1/32768 1/65536]);
errs = zeros(size(hs));
t = MAXTIME; yExact = 0.25*t - 0.1875 + 1.1875*exp(4*t);
for k = 1:length(hs)
    h = hs(k);
    nsteps = round(MAXTIME/h);
    yNumerical = ExplicitTrapezoid(@ExampleODERightSide,t0,MAXTIME,y0,nsteps);
    errs(k) = abs(yNumerical-yExact);
end
errs
```

The output

```
>> TestExplicitTrapConvergence
errs =

   0.0044979
   0.0011249
  0.00028127
  7.0325e-05
  1.7582e-05
```

from the new script indicates that the error decreases by a factor of $4 = 2^2$ as the each time the time step $h = \Delta t$ is halved. To confirm, we compute the following ratios.

```
>> errs(2:end)./errs(1:end-1)
ans =

     0.25009
     0.25005
     0.25002
     0.25001
```

The errors are indeed going down by a factor of 4. Whence

$$|y(t_F) - y_{\texttt{nsteps}}^{t_F/\texttt{nsteps}}| = O(\Delta t^2),$$

and the explicit trapezoid rule has order of convergence 2.