# Stepsize

MATH1090: Directed Study in Differential Equations

http://people.sc.fsu.edu/~jburkardt/classes/math1090_2020/convergence/convergence.pdf

*Can we anticipate the problem areas ahead?*

**Stepsize**

*If we can estimate how "rough" the ODE solution is, we can adjust our stepsize so that we are both accurate and efficient.*

## 1 Sometimes, if it looks wrong, it is wrong!

Last week, we finished with a convergence study of the flame ODE. However, when although I repeatedly decreased the stepsize, the convergence plot showed no improvement at all. It's natural to believe that, when we write a program to do something, that's what it does. If we don't see an error message, we assume things are going well. But the bad convergence plot was a kind of error message, and I was not paying attention to it.

I sat down and tried using a thousand, ten thousand, and a hundred thousand steps, and still the program showed no convergence. Finally, I started to doubt my program. And when I looked at the file *flame_euler_backward.m*, I discovered two mistakes:

```
function [ t, y ] = flame_euler_backward ( n )

  t = zeros ( n + 1, 1 );
  y = zeros ( n + 1, 1 );

  a = 0.0;
  b = 250.0;
  dt = ( b - a ) / n;

  t(1) = a;
  y(1) = 0.0;   % Wrong initial condition

  for i = 1 : n
    t(i+1) = t(i) + dt;
    y(i+1) = ( y(i) + dt * 50.0 * cos ( t(i+1) ) ) / ( 1.0 + 50.0 * dt );   % Wrong derivative
  end

  return
end
```

Listing 1: The incorrect backward Euler code for the flame ODE.

I first noticed that derivative was completely wrong. It's the backward Euler formula we came up with for the stiff equation, not the flame. It's harder to make the flame derivative into a backward formula, so I made a new file, *flame_euler.m*, which uses the forward Euler method instead.

But this program produced a zero solution for all time. That's when I noticed that the line `y(1)=0.0;` should have been `y(1) = delta;`; the flame must start out with an initial nonzero radius, otherwise it stays at zero for ever.

The forward Euler code can be made to work, so let's go back and repeat the investigation that I botched last week. Then we'll go back and see about make a backward Euler solver...

## 2 The forward Euler code

Just to be clear, here's the forward Euler code, which needed the additional input quantity `delta`:

```
function [ t, y ] = flame_euler ( n, delta )

  t = zeros ( n + 1, 1 );
  y = zeros ( n + 1, 1 );

  a = 0.0;
  b = 250.0;
  dt = ( b - a ) / n;

  t(1) = a;
  y(1) = delta;   % Correct initial condition

  for i = 1 : n
    t(i+1) = t(i) + dt;
    y(i+1) = y(i) + dt * ( y(i)^2 - y(i)^3 );  % Correct form for forward Euler
  end

  return
end
```

Listing 2: flame_euler.m the (correct) forward Euler code for the flame ODE.

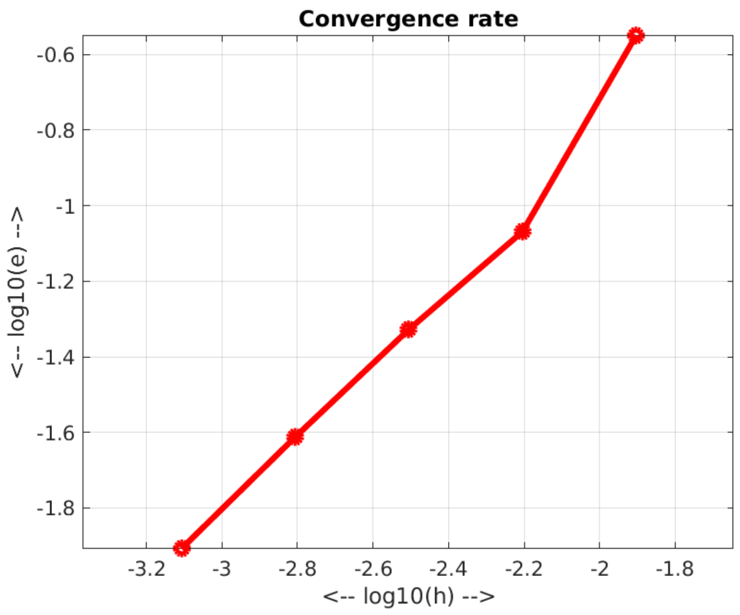
## 3 Convergence of the forward Euler method for the flame

Now our results will make some sense:

```
Stepsize and error

    n          h              e

   20       12.5000            NaN
   40        6.2500            NaN
   80        3.1250         0.2826
  160        1.5625         0.0854
  320        0.7812         0.0470
```

```
   640         0.3906         0.0244
  1280         0.1953         0.0124

Step ratio and error ratio

     n         hold/h         eold/e

    40         2.0000            NaN
    80         2.0000            NaN
   160         2.0000         3.3094
   320         2.0000         1.8155
   640         2.0000         1.9265
  1280         2.0000         1.9747

Log(h) versus log(error)

     n        log10(h)      log10(e)
    20         1.0969            NaN
    40         0.7959            NaN
    80         0.4949        -0.5489
   160         0.1938        -1.0687
   320        -0.1072        -1.3277
   640        -0.4082        -1.6124
  1280        -0.7093        -1.9079
```

The step ratio versus error ratio table makes it easy to see that the error and stepsize are decreasing together, so that we can guess that $e \propto h^1$.



*Convegence plot for Euler on the Flame problem.*

Looking at the convergence plot, we can also feel comfortable in assessing the convergence rate as 1.

## 4 Making a stab at a backward Euler solver for the flame ODE

The problem with writing a backward Euler solver for the flame is that the right hand side is nonlinear in the solution $y$. Thus, formally, the backward Euler solver would be

```
  y(i+1) = y(i) + dt * flame_deriv ( t(i+1), y(i+1) );
```

but when we write out the flame derivative and move it to the right hand side we get

```
  y(i+1) - dt * ( y(i+1)^2 - y(i+1)^3) = y(i);
```

and we can't see an easy way to "solve" for `y(i+1)`. As a quick fix, we can write instead:

```
  y(i+1) - dt * y(i+1) * ( y(i) - y(i)^2) = y(i);
```

resulting in a "implicit-explict" or "semi-backward" Euler method:

```
  y(i+1) = y(i) / ( 1.0 - dt * ( y(i) - y(i)^2) );
```

When the derivative is nonlinear, we can either try a method like this, to allow us to extract a linear factor. A better and more reliable solution is to try to solve the nonlinear equation using a numerical technique. We will put that idea off until next time.

## 5 Heun's method: a second order ODE solver

The Heun ODE solver can be written in the following Runge-Kutta form:

$k_1 = dt\, y'(t_n, y_n)$

$k_2 = dt\, y'(t_{n+1}, y_n + k_1)$

$y_{n+1} = y_n + 0.5\, k_1 + 0.5\, k_2$

Thus, the method involves the computation of two stages, $k_1$ and $k_2$, which are then combined to form the solution estimate. Heun's algorithm has order 2, so it is more accurate than the Euler method.

```
Name: rk2
Input: yprime, n, tspan, y0
Output: t, y

yprime evaluates the right hand side of the ODE.
n is the number of steps to take.
tspan is a vector containing first and last times.
y0 is a vector containing the initial condition.

t is a vector of computed times.
y is a vector of computed ODE solutions.

BEGIN FUNCTION
  Set t to equally spaced values in tspan(1) to tspan(2).
  Set dt to the size of single time step.
  Set the first entry of y to y0.

  LOOP N TIMES ON I
    Set k1 to dt times the derivative at t(i) and y(i)
    Set k2 to dt times the derivative at t(i)+dt and y(i)+k1.
    Set the next y(i+1) to the previous y plus (k1+k2)/2.
  END LOOP

END FUNCTION
```

Listing 3: Pseudocode for rk2.m

We could do a convergence study on Heun's method to verify that it has an order 2 convergence rate.

## 6 Error estimation

Since Euler's method has order 1 and Heun has order 2, we could consider the following idea for estimating the error we make at each step:

1. `y1(i+1)` = Euler step from `y1(i)`;
2. `y2(i+1)` = Heum step from `y1(i)`;
3. Error `e = ||y2(i+1)-y1(i+1)||`;

In our textbook examples, we are able to compute the error exactly, because we already know the exact colution. In real life, that information is not given. By using a pair of solvers whose accuracy differs, we have a chance of roughly estimating the error. In other words, the higher order solver is presumably so much closer to the right answer that we can take the difference between our two solvers as a good error estimate.

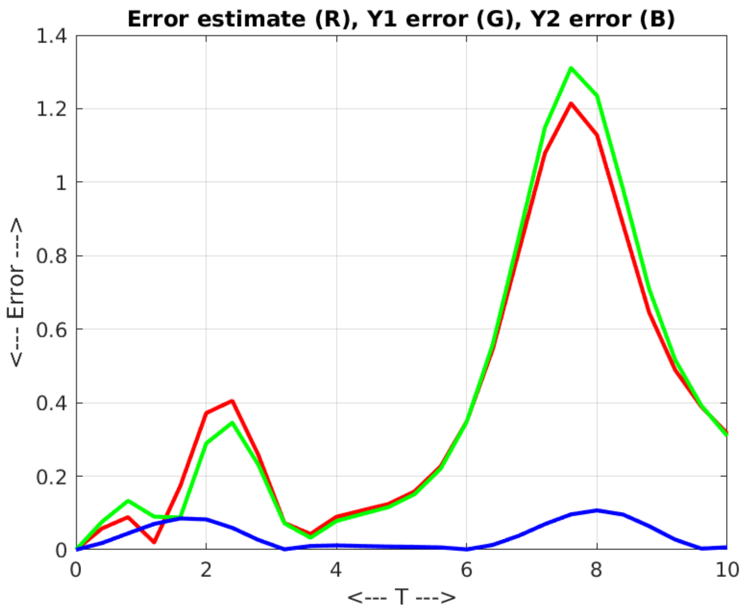
The file *rk12.m* implements this idea, calling *rk1.m* and *rk2.m* and returning the error estimate `e` along with the solution:

```
function [ t, y, e ] = rk12 ( yprime, n, tspan, y0 )
```

Let's examine the reliability of this idea for a particular example, comparing our error estimate to the true error. Consider the problem

$$y'(t) = \cos(t) * y$$
$$y(0) = 1.0$$
$$y_{\text{exact}} = e^{\sin(t)}$$

For this case, I will actually run the two solvers separately, so I can plot both errors.



*The Euler error (G), Heun error (B), Estimated error (R)*

## 7 Adaptive stepsizes

We have seen problems where the error is large over some time intervals, and very small elsewhere. To control error, we reduce the stepsize. But an efficient algorithm would reduce the stepsize in places where the errors seem large, and increase it when the errors are so small that a larger stepsize seems safe.

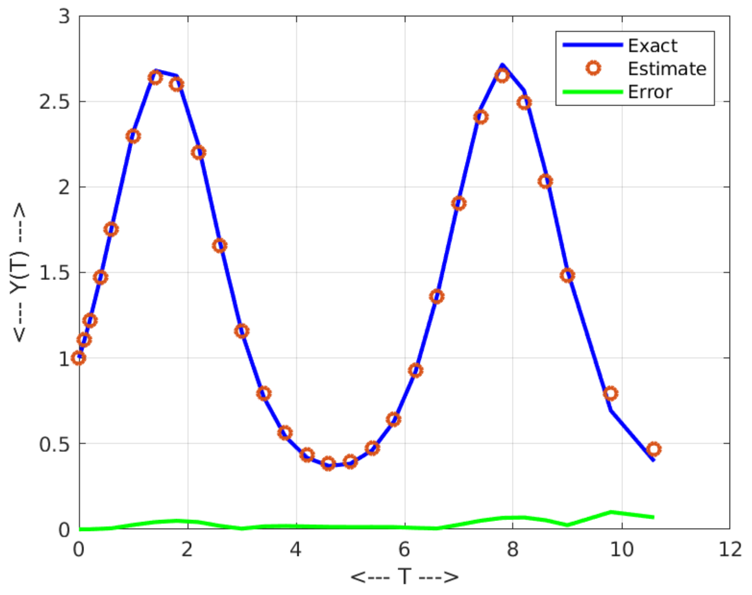
Although we can't know the true error, we can use our error estimate for this process. A reasonable adaptive scheme to estimate the next step, might be

```
dt: current step
e:  current error estimate
t:  tolerance

if ( tol * dt < e )
  dt = dt / 2
else if ( e < tol * dt / 16 )
  dt = dt * 2
else
  dt = dt
end
```

The code *rk12_adapt.m* uses this idea to vary the stepsize according to the error estimate. Note that there is no input quantity `n`, because we no longer ask for a fixed number of steps. Instead, we give an initial stepsize `dt` and an error tolerance `tol`:

```
function [ t, y ] = rk12_adapt ( yprime, tspan, y0, dt, tol )
```

The code *expsin_rk12_adapt.m* solves our test problem using `rk12_adapt`. We plot the approximate solution as dots, rather than a line, so that we can see how the stepsize varies.



*Exact solution (B), RK12 estimate (R), Error (G)*

## 8 Report:

Consider the pendulum problem:

$$
\begin{aligned}
u'' &= -\frac{g}{l}u \\
u(0) &= \frac{\pi}{3} \\
u'(0) &= 0 \\
u(t)_{\text{exact}} &= \frac{\pi}{3}\sin(\sqrt{\frac{g}{l}}t)
\end{aligned}
$$

Here the gravitational force $g = 9.8$, and the length of the pendulum $l = 1$. This implies that a single period is $p = 2\pi\sqrt{\frac{l}{g}} \approx 2.0071$. The problem is to be solved over $0 \le t \le 10$. The following files are available on the web page:

- *pendulum_deriv.m*: `duvdt=pendulum_deriv(t,uv)` returns the right hand side of the pendulum problem;
- *pendulum_exact.m*: `uv=pendulum_exact(t)` returns the exact solution (u,u') of the pendulum problem as a pair of values;
- *pendulum_euler_backward.m*: `[t,u,v]=pendulum_euler_backward(n)` solves the pendulum ODE using the backward Euler method;
- *pendulum_midpoint.m* `[t,u,v]=pendulum_midpoint(n)` solves the pendulum ODE using the midpoint method;
- *pendulum_rk4.m*: `[t,u,v]=pendulum_rk4(n)` solves the pendulum ODE using a 4th order Runge-Kutta method;
- *rk4.m*: a fourth order Runge Kutta ODE solver, needed by `pendulum_rk4`.

Note that the derivative $u'$ will be stored as the variable `v`, and that in some cases, we cram both `u` and `v` into a pair `uv`.

The backward, midpoint, and rk4 codes are set up to solve the pendulum problem over 4 periods using `n` equal steps. Do a convergence study to each of these methods. You might start with $n = 50$, followed by 100, 200, 400 steps. Compute the error ratio, and also the ratio of the log of the error ratio to the log of the stepsize ratio:

```
nold/n        hold/h      eold/e    log10(eold/e) / log10(hold/h)
--------      ------      ------    -----------------------------
  50/100      2.0000      ......    .............................
 100/200      2.0000      ......    .............................
 200/400      2.0000      ......    .............................
 400/800      2.0000      ......    .............................
```

The last column should approximate the order of convergence of each method.

Also create a convergence plot of $\log 10(h)$ versus $\log 10(e)$ for each case.

Bring your tables and plots to our next meeting, at 1:00pm, Thursday, 27 February, room Thackery 624.