Error
Estimating the error in an ODE solution

MATH1902: Numerical Solution of Differential Equations

http://people.sc.fsu.edu/~jburkardt/classes/math1902_2020/error/error.pdf

Without error estimation, your path can go very wrong!

Error

In real life ODE problems, we have no exact solution to compare to our approximation. Is there any way to estimate how far off we are?

1 Every approximation is wrong. Can we estimate how wrong?

For most of the ODE examples we have looked at, we have known a formula for the exact solution, so we can actually see whether our approximation is close. But if you are doing a real computational problem, you almost certainly don't know the exact solution. When you run your solver, you get an answer, and you know it is wrong - that is, it is only an approximation. How do you know whether the errors you made Is it possible, though, to determine whether your approximation is close, or perhaps very far from the correct solution?

It might seem impossible to determine errors; certainly, if we could compute the error exactly, then we could compute the exact solution, which we can't really do. But is there some way to build some confidence in our result?

Let's consider a few approaches to this problem:

- If you only have one ODE solver, recompute with a smaller stepsize;
- Use solutions of different stepsize to extrapolate;
- Use a more accurate solver to monitor a weaker one;

2 Compare big and small step approximations

Previously, we considered the error made by the Euler method, when using n equal steps of size dt to march from an initial condition (t_0, y_0) to compute an approximate solution (t_{stop}, y_n). We then considered using twice as many steps of half the size, computing a new estimate which we can write as (t_{stop}, y_{2n}). We were able to claim that, at least in the limit, the error in the new estimate would tend to be half as large as before.

So suppose we have just computed n equal Euler steps from $t0$ to $tstop$, giving us a solution vector of length $n + 1$ which we will have to call $y1$. We are interested in estimating, for each time t_i, the value of e_i, the error between our estimate $y1_i$ and the exact solution $y(t_i)$:

```
  e_i = y1_i - y(t_i);
```

Unfortunately, in real life we don't know the exact solution, so now we will try to see how we can estimate the error.

So now let's generate our second solution approximation $y2$ using twice as many steps. Then every other entry of $y2$ will contain a second (and more accurate) estimate of a value we've already estimated in $y1$. Therefore, we can use the following estimate for MATLAB indicees `i = 1, n + 1`:

$$e_i \approx y1_i - y2_{2i-1}$$

The crazy indexing on `y2` accounts for the fact that MATLAB starts indexing at 1, and only the odd entries of `y2` are to be used:

```
y1(1)         y1(2)       y1(3)             y1(n)             y1(n+1)
y2(1)  y2(2)  y2(3) y2(4) y2(5) y2(6) ...   y2(2*n-1) y2(2*n) y2(2*n+1)
e(1)          e(2)        e(3)              e(n)              e(n+1)
```

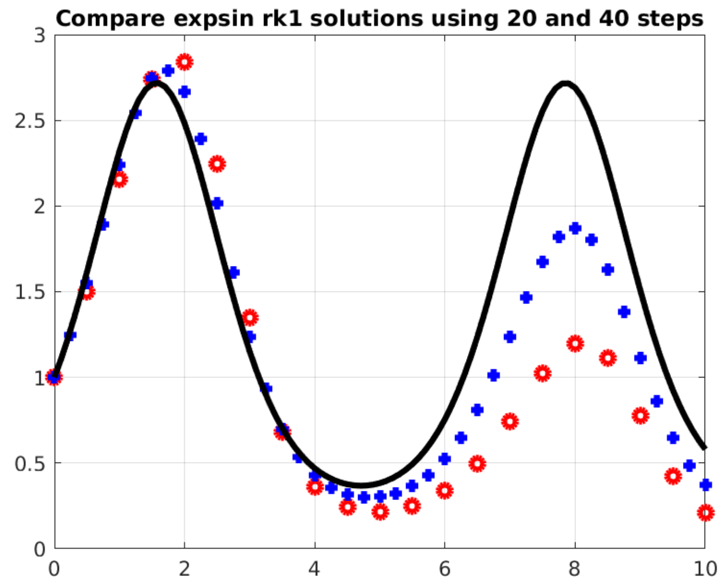
So even though we don't know the exact solution, and even if our ODE solver is the least accurate one available (Euler), we still can not only estimate the solution, but also estimate our error! This only requires that we are willing to do some extra calculations.

3 *expsin_rk1_double.m* uses the step doubling technique

```
function expsin_rk1_double ( n1 )

  dydt = @ expsin_deriv;
  t0 = 0.0;
  tstop = 10.0;
  tspan = [ t0, tstop ];
  y0 = 1.0;

  [ t1, y1 ] = rk1 ( dydt, tspan, y0, n1 );

  n2 = 2 * n1;
  [ t2, y2 ] = rk1 ( dydt, tspan, y0, n2 );

  y = expsin_solution ( t1 );

  e =  y1(1:n1+1) - y(1:n1+1);
  e2 = y1(1:n1+1) - y2(1:2:n2+1);

  e_norm  = rms ( e );
  e2_norm = rms ( e2 );

  return
end
```

Listing 1: Error estimate by step doubling.

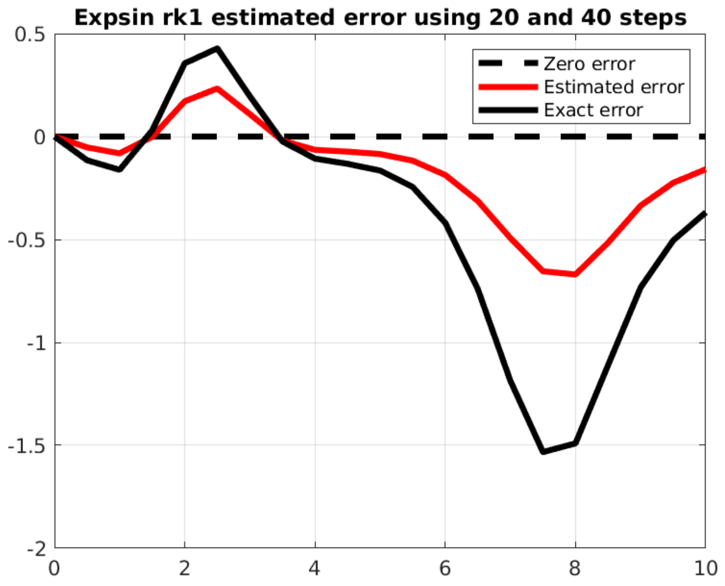
If we plot the `y1` and `y2` solutions, you can see how `y2` is computed on twice as many time values as `y1`, and is clearly closer to the exact solution. In a real world problem, the exact solution curve cannot be constructed, and so our error estimate must be formed from data that we can compute, namely the distance between the pairs of `y1` and `y2` solutions.



Solution estimates using 20 (red) and 40 (blue) steps to exact solution (black).

4 Using the error estimate

Consider a plot of the exact and estimated error vectors for the EXPSIN ODE, what the program calls the vectors `e` and `e2`. Remember that the blue line (exact error) is important to us, but invisible, while the red line can be computed. The plot suggests that the error estimate is doing a reasonable job. It goes up and down, with the right sign and shape, although the magnitude of the estimate seems to be noticeably smaller than the actual value.



Estimated error comparing 20 and 40 steps (red) compared to exact error (black).

Now in a real problem, there is no blue line, but we can compute the data that appears on the red line. Even though it isn't exact, the variations and size of this error approximation is a clear indication of how well we are doing in solving the problem. That means that we can use this approach in order to decide whether to accept our computation with the current number of steps, or to redo it with more steps because our error estimate has exceeded some tolerance that we impose.

Later, we will look at how to do the error estimates step by step, so that instead of having to redo the whole calculation, we may simply reject the current step and retry it with a smaller stepsize. But now we will see how to sharpen our error estimate so that its magnitude is much closer to the true value.

5 Extrapolation squeezes information out of a pair of solutions

It turns out that if the unknown solution $y(t)$ is smooth enough to allow a Taylor expansion up to third degree, then there is a formula $Y(t, dt)$ that can theoretically compute the Euler estimate for the solution of an ODE with given initial data, at a range of later times t and timesteps dt:

$$Y(t, dt) = y(t) + c_1 dt + c_2 dt^2 + O(dt^3)$$

This formula is completely "theoretical", in that we would never want to try to work out the coefficients involved in the formula. On the other hand, the formula allows us to compare estimates two estimated solutions `y1` and `y2` made with stepsizes `dt` and `dt/2` respectively:

$$\begin{aligned} y1 = Y(t, dt) &= y(t) + c1dt + c2dt^2 + O(dt^3) \\ y2 = Y(t, dt/2) &= y(t) + c1dt/2 + c2(dt/2)^2 + O(dt^3) \\ 2 * y2 &= 2y(t) + c1dt + c2dt^2/2 + O(dt^3) \\ 2 * y2 - y1 &= y(t) - c2dt^2/2 + O(dt^3) \end{aligned}$$

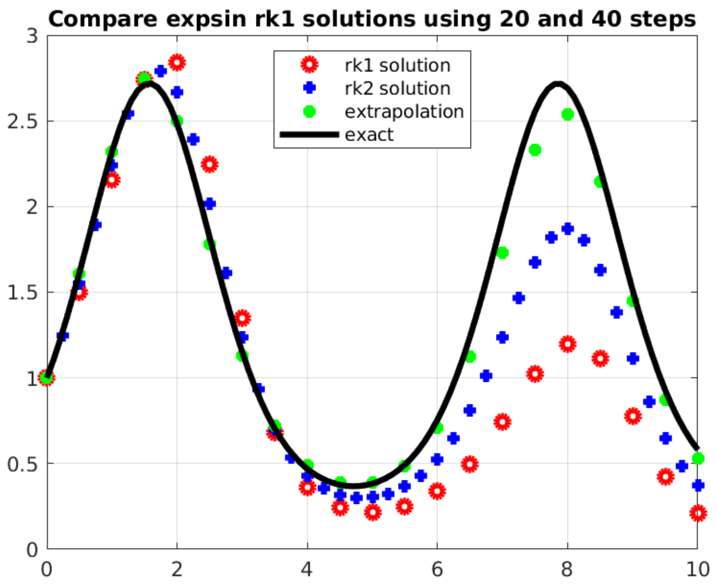
In other words, given these two solution estimates, we can form a new estimate $y3 = 2\,y2 - y1$ where the leading error term is the signifiantly smaller quantity of order $O(dt^2)$.

This is an example of a process called *extrapolation*. We have data (the estimated solution) at several parameter values (dt and $dt/2$) and want to combine this data to produce an estimate at a new parameter value (which for us is essentially $dt = 0$).

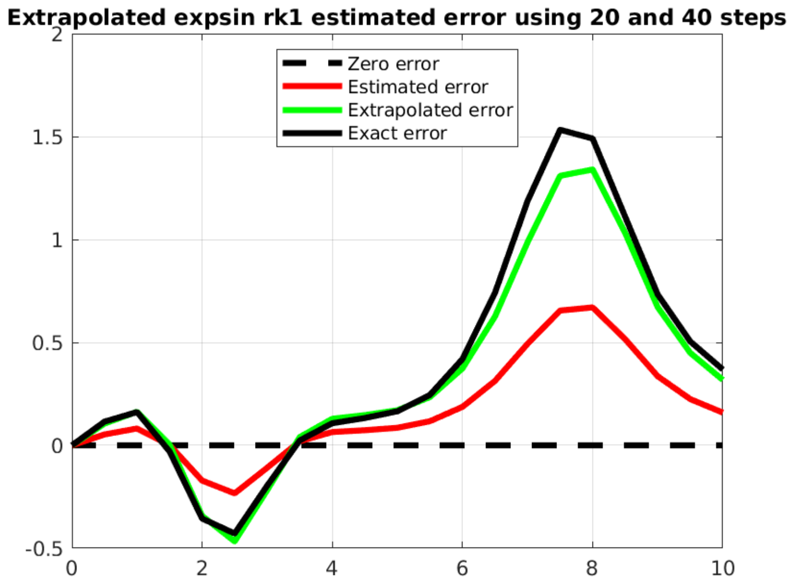
We have already computed almost everything we need for extrapolation, so here I will just show you the interesting part of the extrapolation code for our problem:

```
function expsin_rk1_extrapolate ( n1 )

%...Lots of code omitted...

  y3 = 2.0 * y2(1:2:n2+1) - y1(1:n1+1);

  y = expsin_solution ( t1 );
%
%  Evaluate the errors at the t1 values.
%
  e =   y(1:n1+1)    - y1(1:n1+1);
  e2 = y2(1:2:n2+1) - y1(1:n1+1);
  e3 = y3(1:n1+1)    - y1(1:n1+1);

  e_norm  = rms ( e );
  e2_norm = rms ( e2 );
  e3_norm = rms ( e3 );

  return
end
```

Listing 2: Extrapolation for the expsin ODE.



The extrapolated estimate (green) deduced from red and blue estimates.



The extrapolated error estimate (green) compared to previous estimate (red).

It's important to realize that the secret to the extrapolation method was knowing the relationship between the stepsize dt and the leading term in the error. Roughly speaking, the error at any point was of order $c_1 dt$. This made is possible to find a linear combination of our two solutions such that the leading error term was canceled out.

Suppose we had approximated the EXPSIN ODE using *rk2()*, with a solution `y1` that used `n=10` and a second solution `y2` that used `n=20` steps. For *rk2()*, the formula for the value at any time t using any timestep dt would look like

$$Y(t, dt) = y(t) + c_2 dt^2 + c_3 dt^3 + O(dt^4)$$

What linear coefficients a and b could we use in the extrapolation

$$y3 = a * y2 + b * y1$$

that would cancel out the leading error term associated with the coefficient c_2?

6 *expsin_rk1_higher.m* compares rk1() and rk2() solutions

The Runge-Kutta ODE family provides a sequence of ODE solvers of increasing order of accuracy. Thus, the global error for *rk1()* (AKA "the Euler method") is $O(dt)$, while *rk2()* global error behaves like $O(dt^2)$ and so on. This suggests an alternative procedure for estimating our approximation error: solve twice, with a pair of Runge-Kutta methods of order k and $k+1$. Call these solutions y^k and y^{k+1}. Treat the y*k+1 solution as though it were exact, and estimate the error at any time t_i as

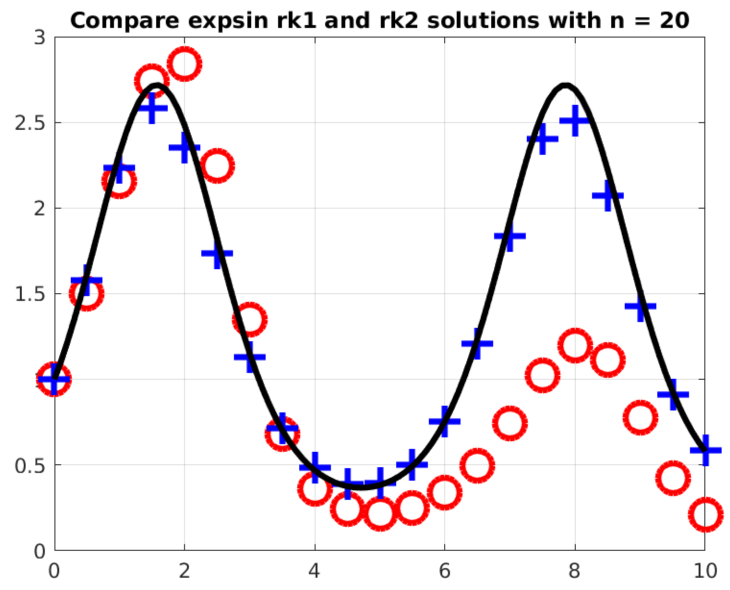
$$e_i \approx y^k - y^{k+1}$$

It's natural to try this with $k = 1$, using Euler and *rk2()*:

```
function expsin_rk1_higher ( n )

  dydt = @ expsin_deriv;
  t0 = 0.0;
  tstop = 10.0;
  tspan = [ t0, tstop ];
  y0 = 1.0;

  [ t, y1 ] = rk1 ( dydt, tspan, y0, n );
  [ t, y2 ] = rk2 ( dydt, tspan, y0, n );
  y = expsin_solution ( t );

  e =  y1 - y;
  e2 = y1 - y2;

  e_norm  = rms ( e );
  e2_norm = rms ( e2 );

  return
end
```

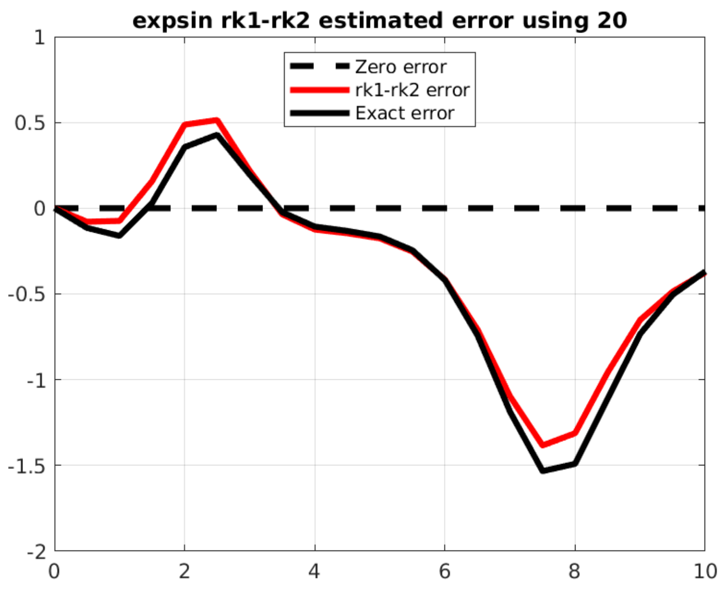
Listing 3: Estimate error by comparing two Runge-Kutta solutions.

For this example, the *rk1()* solution estimate is far from the exact solution, while the *rk2()* solution is rather close. For other problems, stepsizes, or ODE solver pairs, the difference might not be so dramatic. However, this plot suggests why we can assume that the higher-order approximation can be used as a stand-in for the exact solution.



The rk1 estimate (red) and rk2 estimate (blue).

Comparing our error estimate to the exact error shows very good agreement. At least for this problem, if we didn't know the exact solution, we expect that we could still use this approach to estimate the error very closely.



The error estimated from rk1-rk2 (red) versus exact rk1-exact (black).

A common error estimation approach uses Runge-Kutta solvers of order 4 and 5, giving a very accurate solution and a very accurate error estimate as well.

7 The NORMAL ODE example

In probability, there is a normal probability distribution function (PDF) defined over $-\infty < t < +\infty$ as follows:

$$y = \frac{1}{\sqrt{2\pi}} e^{-t^2/2}$$

which satisfies the ODE

$$y' = -t \frac{1}{\sqrt{2\pi}} e^{-t^2/2}$$

We are interested in solving this ODE over the interval $-5 \leq t \leq +5$, with the initial condition

$$y0 = y(-5.0) = \frac{1}{\sqrt{2\pi}} e^{-25/2} \approx 0.0000014867$$

Write the necessary MATLAB functions *normal_solution(t)* and *normal_deriv(t,y)*.

Using just `n=10` steps and the Euler *rk1()* method, carry out the ODE solution.

1. Use the step doubling technique with $n1 = 10$ and $n2 = 20$ to estimate the RMS error you have made.
2. Use the extrapolation technique to give an improved estimate of the RMS approximation error.

8 Homework #5

Submit the values of your two RMS error estimates to `trenchea@pitt.edu`

9 Extra Credit

You can also define an ODE for the normal PDF as follows:

$$y' = -t * y$$

However, all ODES are not the same! If you solve this ODE with $n1 = 10$ and $n2 = 20$ steps, you will not get the same approximation as you did before. Using the extrapolation technique, try increasing $n1$ and $n2$ as necessary, until you can estimate that your RMS error is less than 0.04. Report $n1$, $n2$ and your error.