Stepsize

MATH1902: Numerical Solution of Differential Equations

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

---

Can we anticipate the problem areas ahead?

Stepsize

Small steps are more accurate; big steps solve the problem faster. We know how to estimate the error we make on each step. Can we use that knowledge to adjust our stepsize as we go along?

1 Adapting your stepsize

During our discussion of error, we computed two estimates for the solution of an ODE, $y1$ and then $y2$, using n steps of the *rk1()* and *rk2()* methods. The difference vector $e = y1 - y2$ was our estimated error at the $n+1$ solution points (although we could also use extrapolation to sharpen this estimate). This information was only available after we had completed both computations. At that time, we could decide whether the estimated error was small enough that we could accept the solution, or required us to try again with a smaller stepsize.

All this time, we have been using a fixed stepsize dt for every step. We know that the stepsize has a strong influence on the size of the error. We also know that, as soon as we have taken a step from t_i to t_{i+1}, we only need to compare y_i and y_{i+1} to decide whether the single step we have just take resulted in too much error.

But suppose that we somehow compute the $y1$ and $y2$ values simultaneously, as we go along. After computing our next pair of values, we have enough information to estimate the error we have just made. Depending on the size of the error, we might increase, decrease or maintain the current stepsize. Bigger steps get us a solution faster; smaller steps, if necessary, get us a more accurate solution.

Let's consider what tools we already have, and how we can combine them to create an ODE solver with an adjustable stepsize.

2 *adapt0.m* is a framework for an adaptive stepsize code

We have already implemented the step-doubling + extrapolation algorithm as part of the discussion of error estimation. Now we want to use those ideas to allow us to vary the stepsize. But varying the stepsize means that we can no longer use the input quantity n to specify how many steps to take. We need a different strategy to get us from $t0$ to *tstop*.

Because we plan to use a variable stepsize, we can't know how many steps to take, so we can't use a MATLAB `for` loop. Instead, a `while()` loop is used, which tells us to repeat the operations until the given test is true. The format is

```
while (TEST )
  things to do
end
```

Listing 1: A MATLAB while statement

In our case, `TEST` will ask whether we have reached or passed `tstop` yet.

Here is a sample code *adapt0.m* which has this form. We will use this code in the same way we used codes like *euler()* and *rk2()* to solve an ODE. The *adapt0()* code doesn't actually change the stepsize yet, but we will start from this code and then add instructions that allow us to do so.

```
function [ t, y ] = adapt0 ( dydt, tspan, y0, dt, tol )

  t0 = tspan(1);
  tstop = tspan(2);

  i = 1;
  t(i) = t0;
  y(i) = y0;

  while ( t(i) < tstop )

    y1 = y(i) + dt * dydt ( t(i), y(i) );

    th = t(i) + dt / 2.0;
    yh = y(i) + dt / 2.0 * dydt ( t(i), y(i) );
    y2 = yh   + dt / 2.0 * dydt ( th, yh );

    t(i+1) = t(i) + dt;
    y(i+1) = 2.0 * y2 - y1;
    i = i + 1;

    e = y2 - y1;
%
%  Do you want to change the next stepsize?
%
%   dt =?
%
  end

  return
end
```

Listing 2: adapt0.m source code.

Notice the the usual input n is missing, and that two new inputs have been added:

1. `dt`, the initial stepsize to use;
2. `tol`, an error tolerance.

We will make three versions of this program. Each time, we will only be adding or changing statements that follow the comment about adjusting the stepsize. So you should be able to build your own versions of *adapt1.m*, *adapt2.m* and *adapt3.m* by simple modifications, after copying *adapt0.m* from the web page.

3 *adapt1.m* reduces the stepsize after big errors

Our first attempt to adjust the stepsize involves reducing the next stepsize if we observe a big estimated error on the current step. The input quantity `tol` will control what we think is a big error. The idea of the error tolerance is that we hope to make a total global error of no more than `tol` over the whole interval $t_0 \leq t \leq t_{stop}$. When we are working in an interval $[t, t+dt]$, this represents a proportion $dt/(tstop - t0)$ of the global interval. Therefore, a reasonable goal is to keep the local error e in this interval bounded by not much more than $tol * dt/(tstop - t0)$. Thus, we will assume that our stepsize is not too big if

$$|e| \leq \frac{tol * dt}{tstop - t0} \quad \text{Error is not too big.}$$

and correspondingly, if e is bigger than this limit, we need to cut back the stepsize. If we see that our error exceeds this value, we will simply halve the value of dt for the next step.

Note that this adjustment is not very sensitive. Our error could be a little more than our goal, or a lot more, but we always simply cut the stepsize in half. If this idea seems to work, then we can probably improve it by producing a more sensitive stepsize adjustment factor.

This is all we need to do to create *adapt1.m*. The lines that we have changed look as follows:

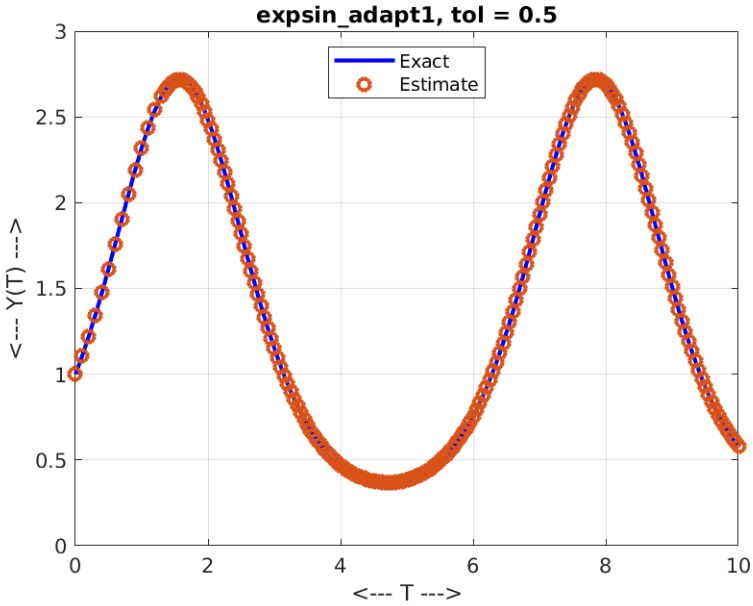
```
1  function [ t, y ] = adapt1 ( dydt, tspan, y0, dt, tol )
2
3    ***
4  %
5  %  Do you want to change the next stepsize?
6  %
7      if ( tol * dt / ( tstop - t0 ) < abs ( e ) )
8        dt = dt / 2.0;
9      end
10
11   ***
12
13   return
14 end
```

Listing 3: adapt1.m source code.

Let's try out this code on the EXPSIN ODE. Our interval is $0 \leq t \leq 10$, so it might seem reasonable to specify an initial stepsize of 0.1, which would correspond to $n = 100$ if we were using a fixed stepsize. We will start out with an terror tolerance of $tol = 0.5$, and do a few smaller values. Here's what we see:

```
tol    rms(y-ye)   n
---    --------  ---
0.5    0.00059   187
0.25   0.00041   364
0.125  0.00006   731
```

Here is a plot of the results for the first case:



Solving EXPSIN with adapt1.

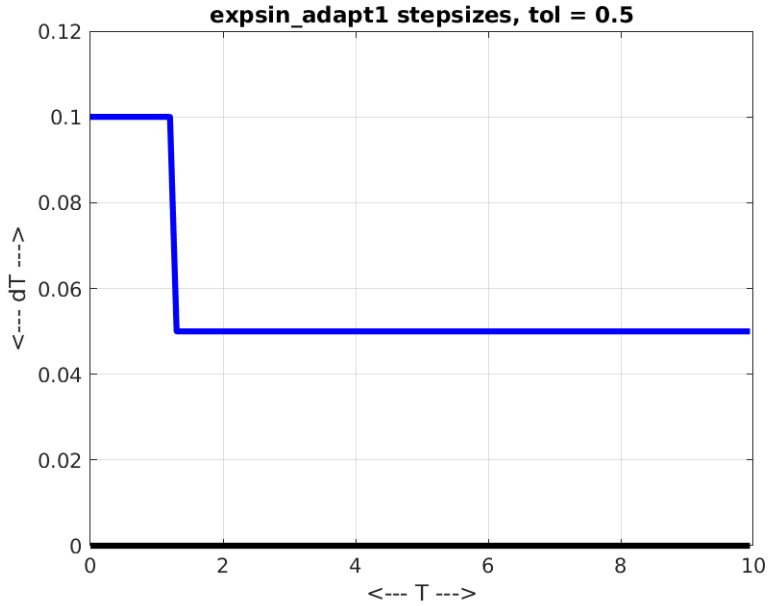
This is disappointing. In fact, if we use the fixed step *rk1()* method with $n = 100$ for this problem, we get an RMS error of 0.16; in other words, if we had just used a fixed step method, we would have gotten a satisfactory answer with much less work. We aren't complaining that the adaptive computations are too accurate, we're complaining that they are too expensive, because the number of steps n is higher than for the fixed point approach, rather than lower.

But let's try to look more closely at the results. In particular, what stepsizes did the program use? We got back the $n + 1$ vector t, so we can create a vector of stepsizes by computing $(t(2 : n + 1) - t(1 : n)$ and plotting that. The coding would be something like this:

```
1  plot ( t(1:n), t(2:n+1)-t(1:n), 'b-', ...   % plot t versus dt, blue line
2         [t(1),t(n)], [0.0, 0.0], 'k-', ...   % plot the X axis, black line
3         linewidth', 3 )                      % make the lines 3 pixels thick
```

Listing 4: Plotting the stepsizes.

Look what we get for our stepsize behavior for the first computation, when $tol = 0.5$.



Adaptive stepsizes for EXPSIN with adapt1.

Here we have an "OOPS" moment! The stepsize started out at $dt = 0.1$ just like we asked. Then it had to be reduced to 0.05 to get around the curved portion of the function. It never went up again...because our program does not include any way for the stepsize to increase. We correctly cut down the stepsize when the error was too high, but we never allowed the stepsize to rise again thereafter. We need to fix this, so that the program doesn't waste small stepsizes on easy problems.

4 *adapt2.m* also doubles the stepsize after tiny errors

So clearly, if the computed error is rather small, we should be justified in increasing the stepsize. To keep things simple, let's just double the stepsize in such cases. How do we decide when to do this? Let e be the error on this step. We are assuming that if we take another step of side dt, we will get another error of size e. Roughly speaking, the local error grows quadratically with the stepsize, so we are also assuming that if we take a bigger step of size $2 * dt$, the error might tend to increase to $4 * e$.

So if e is small, then taking the next step of size $2 * dt$, with an error of $4 * e$ might be safe as long as

$$4 * |e| \leq \frac{tol * 2 * dt}{tstop - t0} \quad \text{We can problably double the next stepsize}$$

We can make a new code, *adapt2.m*, which adds this step doubling feature. The only change comes at the bottom of the loop:

```
1  function [ t, y ] = adapt2 ( dydt, tspan, y0, dt, tol )
2
3    ***
4  %
5  %  Do you want to change the next stepsize?
6  %
7      if ( tol * dt / ( tstop - t0 ) < abs ( e ) )
8        dt = dt / 2.0;
9      elseif ( 2.0 * abs ( e ) < ( tol * dt / ( tstop - t0 ) ) )
10       dt = dt * 2.0;
```

```
11      end
12
13    ***
14
15    return
16  end
```

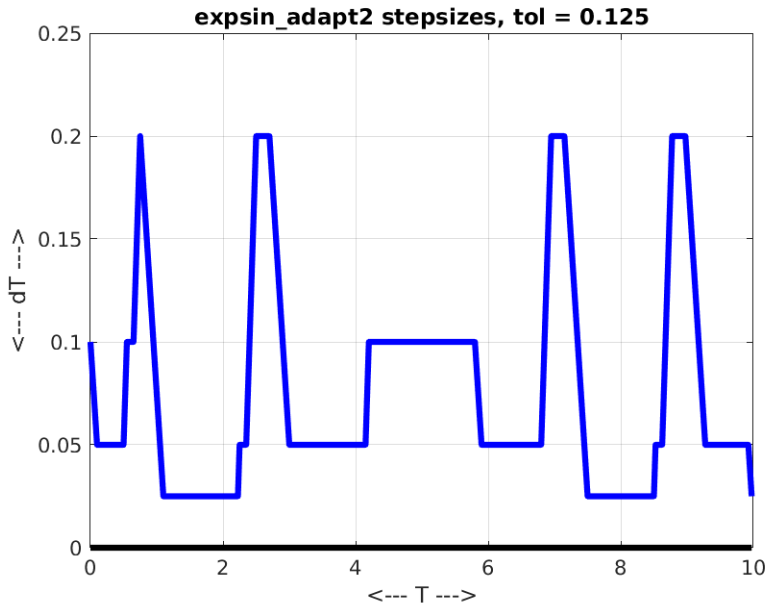
Listing 5: Portion of adapt2.m that includes step doubling.

Now let's repeat our earlier calculations:

```
        <--adapt1--->     <--adapt2--->
tol     rms(y-ye)    n    rms(y-ye)    n
---     --------   ---    --------   ---
0.5     0.00059    187    0.010       46
0.25    0.00041    364    0.017       87
0.125   0.00006    731    0.0018     196
```

The *adapt2* results are certainly an improvement, in that the number of steps n is much reduced, while the error is still well under our tolerance.

To get a better picture of what is going on, let's look at the stepsize changes. The stepsize plot for $tol = 0.125$ verifies that we move back and forth between smaller and larger steps as we proceed along the interval, with the stepsizes corresponding to difficult and easy portions of the calculation



Adaptive stepsizes for EXPSIN with adapt2.

If you think for a moment, you can also explain what is going on at the very first and last stepsizes, where the graph goes sharply down.

5 *adapt3.m* does a careful stepsize adjustment

It turns out that our estimated error e, produced by the step doubling + extrapolation method, has behavior that can be approximated by a quadratic term in dt with unknown coefficient c which should be roughly

constant over the interval of interest:

$$e \approx c\frac{dt^2}{2}$$

If we take this formula seriously, we can solve for c:

$$c = 2e/dt^2$$

Therefore we can figure out the maximum stepsize DT we could have taken, which would have resulted in an error of tol exactly:

$$tol = c\frac{DT^2}{2}$$
$$DT^2 = 2\,tol/c$$
$$DT = dt\sqrt{\frac{tol}{e}}$$

This suggests that if our step of size dt yielded an error of size e, our next step should be computed by multiplying the old stepsize by the factor $\alpha = \sqrt{\frac{tol}{e}}$. This will take care of increasing or decreasing the stepsize.

In order to avoid extreme stepsize changes, it is recommended that the stepsize factor be limited, as for example by the requirement $0.3 \leq \alpha \leq 2.0$, and that it be cautiously reduced by a factor of 0.9. In other words,

1. Compute the error e;
2. Compute the stepsize factor $\alpha = \sqrt{\frac{tol}{e}}$;
3. Force $0.3 \leq \alpha \leq 2.0$;
4. Set next stepsize $dt = 0.9 * \alpha * dt$;

Here's how that might be programmed in MATLAB:

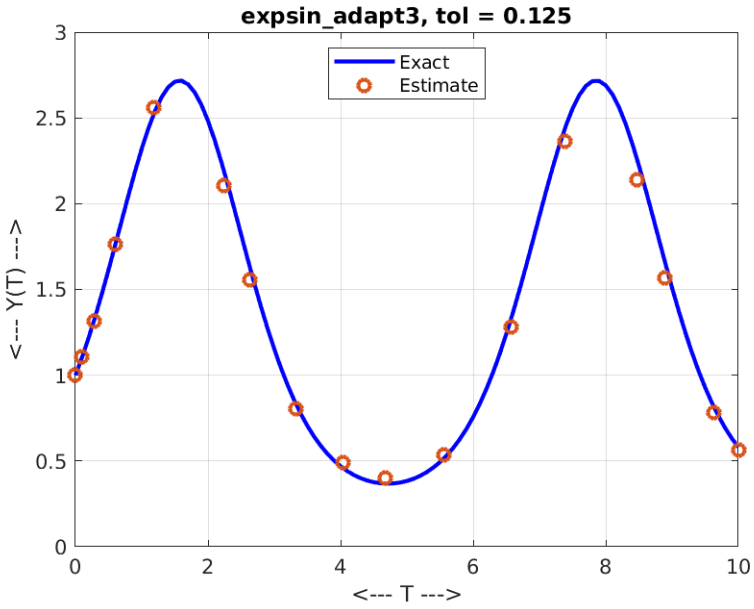
```
function [ t, y ] = adapt3 ( dydt, tspan, y0, dt, tol )

  ***
%
%  Do you want to change the next stepsize?
%
    alpha = sqrt ( tol / norm ( e ) );
    alpha = max ( alpha, 0.3 );
    alpha = min ( alpha, 2.0 );
    dt = dt * 0.9 * alpha;

  ***

  return
end
```

Listing 6: The stepsize formula in adapt3.m

Now let's try our example a third time:

```
       <--adapt1--->    <--adapt2--->   <--adapt3--->
tol    rms(y-ye)   n    rms(y-ye)   n    rms(y-ye)   n
---    --------  ---    --------  ---    --------  ---
0.5    0.00059   187    0.010      46    0.30       12
0.25   0.00041   364    0.017      87    0.08       14
0.125  0.00006   731    0.0018    196    0.05       16
```

Here, it's worth comparing the approximate and exact solutions for the toughest tolerance, since there aren't that many points to plot:



EXPSIN solution by adapt3.

Look at this plot carefully. In particular, notice how the code gets around the first "hill" in the function. This might seem impossible. There's an uphill point, and then a downhill point. Since the Euler method uses the slope to estimate the next point, how is it possible that we switch so easily from going up to going down? Something seems to be missing. (Yes, something is missing from the plot. What is it?)

6 Things to worry about

It looks like we've had a success in using adaptive stepsizes for an ODE. However, here are some things to keep in mind.

- The user has to specify a starting stepsize `dt`. What happens if the user makes a bad choice (much too small, or much too big). Could your program somehow try to estimate a good stepsize?
- Suppose we take a step, and our error estimate is very bad. Right now, we nonetheless accept the result, and simply reduce the stepize before computing the next step. Would it be possible to actually reject the step, and redo it? If the problem is very hard, can you redo the step several times if necessary?
- With a fixed number of steps `n`, we were able to guarantee that the last step ended exactly at $t = tstop$. With a variable step, we only stop when we reach or pass *tstop*. Is there some way of guaranteeing that our last step ends exactly at *tstop*?
- This adaptive stepisze calculation can be modified to work with a vector ODE, such as the predator prey ODE. However, there are some important changes that have to be made to the code to correcly handle vector data. If you don't do this correctly, your code will fail or print nonsense. What changes must be made?

7 The STIFF ODE defines a difficult problem.

We will soon look closely at a new example called the "stiff ODE". For now, you can assume that "stiff" just means the example can be difficult to solve. The ODE can be described as follows:

$$\begin{aligned} y' &= 50(\cos(t) - y) \\ y(0) &= 0 \end{aligned}$$

to be solved over the interval $0 \leq t \leq 1$.

The exact solution to the stiff ODE is

$$y(t) = 50(\sin(t) + 50\cos(t) - 50e^{-50t})/2501$$

I would suggest putting these two computations into MATLAB files *stiff_deriv.m* and *stiff_solution.m*.

8 Homework #6

Use *adapt3.m* to approximate a solution to the stiff ODE, using a tolerance `tol` = 0.05. You will need to choose an appropriate value of the initial stepsize `dt`. Create plots of the solution and the stepsizes. Report the number of steps taken, and the RMS error between your approximation and the exact solution.

Submit your results to `trenchea@pitt.edu`