The STIFF ODE
Backward Euler and implicit ODE solvers

MATH1902: Numerical Solution of Differential Equations

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

A zigzag path is a sign of trouble!

Stiff equations and implicit methods

So far, our ODE solvers have used an explicit formula for the next approximation. But such methods can fail if they encounter a "stiff" differential equation, which results in an enormous errors unless a small step size is used. We develop implicit ODE solvers that can handle such problems.

1 Vocabulary: *explicit* versus *implicit*

The word "explicit" comes from Latin, meaning *unfolded* or *unwrapped*, and "implicit" naturally has the opposite meaning. In mathematics and computation, we are used to explicit formulas, which describe some quantity of interest in terms of the values of other, known quantities. The area of a circle is an explicit function of its radius:

$$A = \pi r^2$$

However, there are many situations in which we express the relationship among a set of variables in a way that makes it difficult, or impossible, to unfold the information into an explicit formula. Consider this example:

$$x^2 + xy - y^2 = 19$$

If we are given a pair of values (x, y), we can determine whether they satisfy this relationship. The values (4,3) work. But if $x = 5$, what is the value of y? It is not possible to rearrange these terms to have an explicit form, along the lines of $y =$ formula involving only x.

Implicit equations occur in various applications, and we will see such an example in this topic, which involves methods of dealing with what are known as "stiff" differential equations.

2 *stiff_deriv.m* defines a surprisingly difficult ODE

Consider the following differential equation, which we will nickname the *"stiff ODE"*:

$$y' = 50(\cos(t) - y)$$

to be solved over the interval $0 \le t \le 1$ with initial condition $y(0) = 0$.

We can write a MATLAB function *stiff_deriv(t,y)* which returns the value of this right hand side function, and hence defines the stiff ODE.

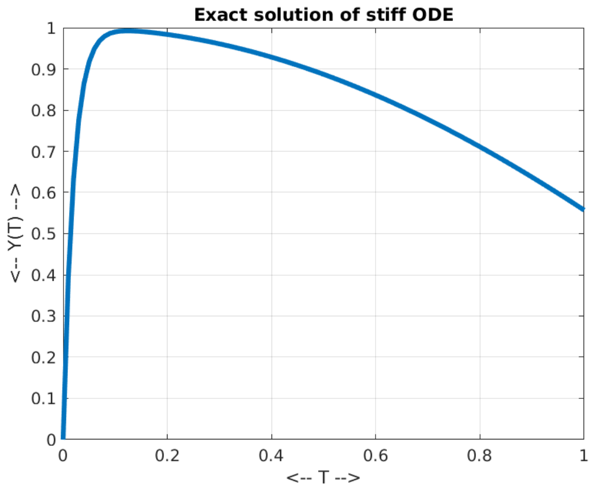
3 *stiff_solution.m* evaluates the exact solution

The exact solution of this stiff ODE is;

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

The appearance of the exponential function in this formula is a bit of a surprise, and, it turns out, will cause us some problems very shortly. However, we can easily write a MATLAB function *stiff_solution(t)* to evaluate this formula. (Remember that an exponential expression like $-50e^{-50t}$ becomes `-50 * exp(-50*t)` in MATLAB.)

If we plot this function, it might at first not look difficult or dangerous, although you might notice that the slope is very steep at $t = 0$, and there is a sharp bend shortly afterwards.



The exact solution of the stiff ODE.

Let's see how well we can approximate this solution.

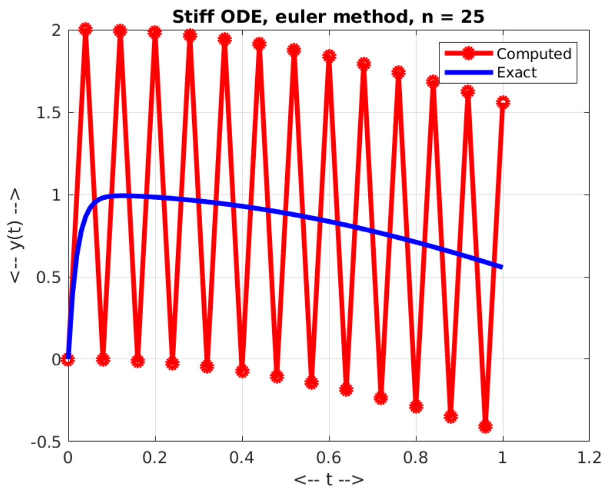
4 *stiff_euler.m* tries to approximate the solution

We start our exploration of this problem by using our simplest ODE solver, the Euler method. (Yes, we have been calling this method *rk1()* for a while, but now we are going to back to calling it *euler()*!) I am warning you that we are going to have trouble with this problem, so it makes sense to write our solver program *stiff_euler.m* in a way that allows us to set the number of steps `n` as an input quantity, so we can easily retry our solution if we are not satisfied.

```
function stiff_euler ( n )

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

  [ t1, y1 ] = euler ( dydt, tspan, y0, n );

  t2 = linspace ( t0, tstop, 101 );
  y2 = stiff_solution ( t2 );

  plot ( t1, y1, 'r-o', ...
         t2, y2, 'b-', ...
         'linewidth', 3 );

  return
end
```

Listing 1: stiff_euler.m Euler method for the stiff ODE.

Let's try the value $n = 25$, which should be enough to capture the shape of the solution curve. Unfortunately, the approximation does not seem to agree well with the true solution. In particular, it does not just drift away, it distinctly wobbles up and down. The resulting approximate zigzag curve (red) looks nothing like the true smooth solution (blue):



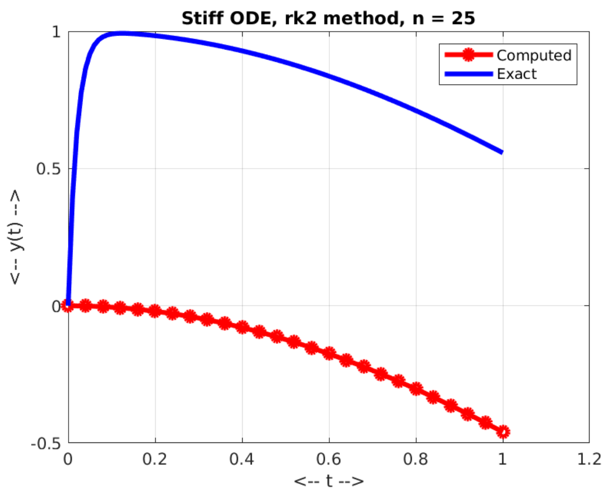
Explicit Euler solution with 25 steps.

It doesn't just seem like inaccuracy, but a wild variation in the direction. This equation is just one example of a stiff differential equation. Such equations include a feature that gives a typical ODE solver make a staggering, zigzag approximation to the solution. Even if we don't know the true solution, we can usually recognize that something is wrong. In general, if we reduce the stepsize dt or equivalently increase n, eventually the solution will look reasonable.

Exercise: Investigate the behavior of the Euler solution on this problem for various values of n. Compare the solutions for $n = 20, 25, 30, 35$. Convince yourself that the problem will go away if our stepsize becomes small enough.

5 *stiff_rk2.m* tries a more accurate approach

We saw earlier that the Euler method can be thought of as *rk1*, the weakest member of the Runge-Kutta family of ODE solvers. It is natural to repeat our calculation using *rk2*, a method which is one step up in accuracy. If we simply make a copy of our solver, but replace the word *euler* with *rk2*, we should be able to easily create the appropriate MATLAB solver code. Here's what I get when I repeat my calculation with $n = 25$:



The rk2 solution with 25 steps.

What a catastrophe! Now the approximate solution seems to be heading towards $-\infty$. Suppose that you had started out to solve this equation using the Euler method with $n = 25$, got a suspiciously bad answer, and tried to make things better by using $n = 25$ with *rk2()*. You would have expected that the higher accuracy solver would provide a better picture, but instead, it is in some ways much worse.

Again, I need to remind out that we can assume that if we keep increasing n, the *rk2()* solution will settle down and start tracking the exact solution. It's just these strange misbehaviors are something we have not seen before in our efforts to solve ODEs.

But consider yourself very lucky that we actually know what the true solution is! In a real life situation, we would simply have two crazy, unbelievable answers to our problem. We could suspect there was something terribly wrong with our program. Or we might think this is simply an impossible problem to solve. And although a small enough stepsize will cure the problem, it might be the case that we can't afford to use the necessary small steps because it will take too long for us to solve our problem.

Exercise: Investigate the behavior of the *rk2()* solution on this problem for various values of n. Compare the solutions for $n = 20, 25, 30, 35$. Convince yourself that the problem will go away if our stepsize becomes small enough.

6 Implicit ODE methods reference data we don't know yet

The ODE solvers we have looked at so far have all been of the **explicit** or **forward** type. That is, when we write the formula for the next solution estimate `y(i+1)` or "ynew", we put all the ODE derivative information on the right hand side, evaluated at the previous time `t(i)` or "told" and previous solution `y(i)` or "yold". And that's why the Euler method can be thought of as drawing the tangent line at the current solution, and following it forward in time a small distance. We write an expression that approximates the derivative

$$\text{dydt(told,yold)} \approx \frac{\text{ynew - yold}}{\text{tnew - told}} = \frac{\text{ynew - yold}}{\text{dt}}$$

and then solve for `ynew`:

```
  ynew = yold + dt * dydt ( told , yold );
```

A different approach to solving ODE's is known as the family of **implicit** or **backward** methods. To make an implicit version of the Euler method, we start out by writing the Euler update equation again, except that we evaluate the right hand side of the ODE at the "future" time `tnew`. In general, this would be something like:

```
  ynew = yold + dt * dydt ( tnew , ynew );
```

where we do have the value of `tnew`, but we do not know the value of `ynew`. It can be difficult to visualize what is happening, because essentially we are trying to approximate the tangent line by using a value that is "in the future".

It's easy to express the backward Euler step algebraically, but now we have to figure out some way to actually solve it. For our first example, we will see that sometimes, if the right hand side is actually linear in the unknown function y, there is a trick to work out the solution.

7 Applying the backward Euler method to our stiff ODE

Let us try using the backward Euler approach for our stiff example. Instead of the forward Euler approximation:

```
  y(i+1) = y(i) + dt * 50.0 * cos ( t(i) ) - dt * 50 * y(i)  % evaluate at (told,yold)
```

Listing 2: Explicit Euler update step.

we now have to pose the following backward Euler approximation, and solve it for `y(i+1)`:

```
  y(i+1) = y(i) + dt * 50.0 * cos ( t(i+1) ) - dt * 50 * y(i+1) % evaluate at (tnew,ynew)
```

Listing 3: Implicit Euler update step.

This is an implicit formula for `y(i+1)`. However, for this particular ODE, it turns out that we can "unfold" the formula into an explicit version by doing the appropriate algebra. This only happens for special cases, and we will later have to deal with what to do when the implicit equation can't be solved.

To crack this particular nut, we will try to rearrange the formula so that it tells us how to evaluate `y(i+1)`. For this case, it's not too hard to work out what the implicit Euler update should be:

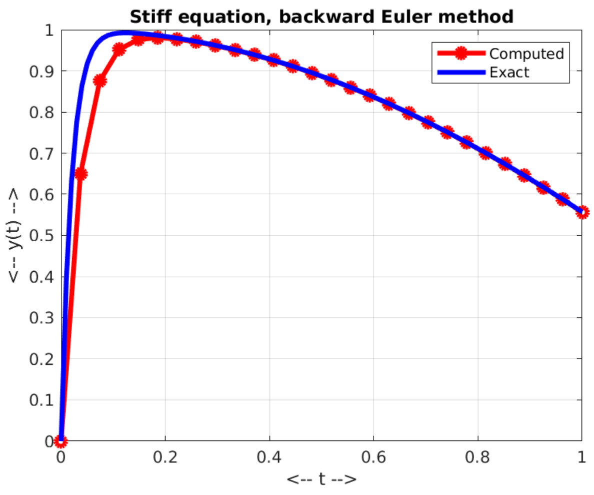
```
y(i+1)                       =   y(i) + dt * 50.0 * cos ( t(i+1) ) - dt * 50 * y(i+1)
y(i+1) + dt * 50   * y(i+1) =   y(i) + dt * 50.0 * cos ( t(i+1) )
(  1.0 + dt * 50 ) * y(i+1) =   y(i) + dt * 50.0 * cos ( t(i+1) )
y(i+1)                       = ( y(i) + dt * 50.0 * cos ( t(i+1) ) ) / ( 1.0 + 50.0 * dt )
```

And look at that! Now we have an explicit formula for `y(i+1)` that we can use in our solution strategy.

```
function stiff_backward_euler_explicit ( n )

  t0 = 0.0;
  tstop = 1.0;
  tspan = [ t0, tstop ];
  y0 = 0.0;

  dt = ( tstop - t0 ) / n;
  t1(1) = t0;
  y1(1) = y0;

  for i = 1 : n
    t1(i+1) = t1(i) + dt;
    y1(i+1) = ( y1(i) + dt * 50.0 * cos ( t1(i+1) ) ) / ( 1.0 + 50.0 * dt );
  end

  return
end
```

Listing 4: stiff_backward_euler_explicit.m computes a backward Euler solution.

Using the same number of steps, our backward Euler solver does a much better job of using the ODE to approximate the true curve:



Backward Euler solution with 25 steps.

8 Exercise: Find explicit form of a backward Euler step

Practice this technique of dealing with the backward Euler method for an equation whose right hand side is linear in the y variable. Consider the following ODE, nicknamed *"trigger"*, which is linear in y:

$$y' = c\,(-y + \sin(t))$$
$$y(0) = 0$$

where the parameter $c = 10$. The equation is to be integrated over $0 \leq t \leq 2\pi$.

The exact solution is

$$y(t) = \frac{ce^{-ct} + c^2 \sin(t) - c\cos(t)}{1 + c^2}$$

Suppose you wanted to solve this problem using a backward Euler method. Start by writing the implicit backward Euler step `y(i+1) = ...`, which will also include the value of `y(i+1)` on the right hand side. Use algebra to determine an explicit formula for `y(i+1)`, as was done for the stiff equation. Once you have this formula, you could implement the backward Euler method for this problem and compare your result to the exact solution.

9 Solving an implicit equation

Turning the implicit backward Euler step into an explicit formula work for the special case where the right hand side is linear in the unknown y. Of course, we can expect that most ODE's will not have this special structure, which means we are stuck unless we can solve the implicit algebraic equation involving `y(i+1)`. In order to use the backward Euler method for a wider range of problems, we need to know how to approximately solve an implicit equation.

Luckily, computational methods are available that can usually provide a good estimated solution to implicit equations. The key to this approach is to use an implicit equation solver such as MATLAB's *fsolve()* function, which is part of the Optimization Toolbox. A similar function, with the same name, is also a built-in part of the Octave program.

In its simplest form, *fsolve()* can try to solve an implicit equation by a command like

```
x = fsolve ( @func, x0 );
```

Listing 5: Solving an implicit equation using fsolve().

Here:

- `x0` is an initial guess for the solution,
- `func()` is the name of a MATLAB code which defines the implicit equation,
- `x` is the approximate solution.

Assuming that the implicit relationship has a form like $\cos(x) = x$, it must be rewritten as a function $f(x)$, where all the terms are on one side, so that we are trying to solve $f(x) = 0$. For this example, the corresponding MATLAB code might be

```
function value = func ( x )
  value = cos ( x ) - x;
  return
end
```

and we could now try to find a solution:

```
>> x0 = 0.0
>> x = fsolve ( @func, x0 )
   x = 0.7391
>> func ( x )
   ans = -2.8480e-10
```

Listing 6: Example of using fsolve().

We have found an approximate solution to the problem $f(x) = 0$, that is,

$$f(0.7391) = \cos(0.7391) - 0.7391 = 0$$

or, in our original form

$$\cos(0.7391) = 0.7391$$

We could visualize this problem by looking for the crossing point of the graphs $y = cos(x)$ and $y = x$.

Note that *fsolve()* can also handle systems of equations, which would be necessary if we are dealing with a system of ODEs.

10 A backward Euler solver using *fsolve()*

When we used the forward Euler method, we were able to put the ODE solver into a separate function that would work with any ODE. Now, with our stiff example, we started with the right hand side, and had to do the algebra on our own, and then set up a code that only works for that problem. We really want to be able to write a general backward Euler solver, with the signature

```
function [ t, y ] = backward_euler ( dydt, tspan, y0, n )
```

Listing 7: Signature for backward Euler code.

which we can use just like the *euler()* code, so that we only have to write a function that defines the right hand side. We can create such a general backward Euler ODE solver thanks to the existence of the MATLAB *fsolve()* function.

The code will look the same, except that we replace the forward Euler explicit formula by a very weird call to a function that sets up the backward Euler formula and calls fsolve() to solve it.

```
function [ t, y ] = backward_euler ( f, tspan, y0, n )

% Ask fsolve() not to print messages.

  options = optimoptions ( 'fsolve', 'Display', 'off' );

  m = length ( y0 );
  t = zeros ( n + 1, 1 );
  y = zeros ( n + 1, m );

  dt = ( tspan(2) - tspan(1) ) / n;

  t(1,1) = tspan(1);
  y(1,:) = y0(:);

  for i = 1 : n

    to = t(i,1);
    yo = y(i,:);

    tp = to + dt;
    yp = yo + dt * f ( to, yo )';  % We use the forward Euler approximation
                                   % as our starting estimate.

    yp = fsolve ( @(yp)backward_euler_residual(f,to,yo,tp,yp), yp, options );

    t(i+1,1) = tp;
    y(i+1,:) = yp;

  end

  return
end
```

Listing 8: backward_euler.m

You can see that my use of *fsolve()* is more complicated than for the simple example we discussed a moment ago. That's because the backward Euler formula involves lots of extra quantities whose values must be passed in. Secondly, notice that we need a starting estimate for the solution, and we have used the forward Euler approximation for that. We could instead have used the value `yo` for this purpose.

I don't mean to torture you with MATLAB details, but I don't want to pretend that we are doing magic, so here is the function that actually does the solutions step:

```
function value = backward_euler_residual ( f, to, yo, tp, yp )
  value = yp - yo - ( tp - to ) * ( f ( tp, yp ) )';
  return
end
```

Listing 9: Code to evaluate the backward Euler relation.

11 *stiff_backward_euler()* repeats our computation

Even though we've already solved our stiff ODE with an explicit formula, it's a good idea to test out our new code and see if its solution matches our previous result.

The important thing to note is that, by using fsolve() to handle the implicit equation, we now have a backward Euler ODE solver that can be used in the exact same way as our familiar codes like *euler()* and *rk2()*:

```
function stiff_backward_euler ( n )

  dydt = @ stiff_deriv;
  t0 = 0.0;
  tstop = 1.0;
  tspan = [ t0, tstop ];
  y0 = 0.0;
%
%  Same format as if we were calling the Euler code...
%
  [ t1, y1 ] = backward_euler ( dydt, tspan, y0, n );

  return
end
```

The moral is, if you have a stiff equation for which you need to use the backward Euler method, then you can go ahead and try to rewrite the method as an explicit formula, which will avoid the cost of an implicit function solution procedure. But if you can't rewrite it, or don't want to, you can use a version like this which will deal with the implicit function automatically.

12 The QUADEX ODE

The "quadex" ODE has a solution that looks like a quadratic (a parabola). However, getting this parabolic solution is difficult because the equation is stiff. Small computational errors can blow up exponentially.

The ODE is

$$y' = 5 * (y - t^2)$$
$$y(0) = \frac{2}{25}$$

to be solved over the interval $0 \le t \le 2$.

The exact solution is

$$y(t) = ce^{5t} + t^2 + 2t/5 + 2/25$$

where, for our initial condition, $c = 0$. Thus, our solution is actually a parabola, but for any other initial condition, the exponential term would be active. These nearby exponential solutions will give us trouble.

Try to solve this problem twice, getting an approximately parabolic answer each time. The first time, use the Euler method, and the second time use the backward Euler method. For each solution attempt, try the values of $n = 25, 250$, and then $n = 2500$.

13 Homework #7

Send the final plots ($n = 2500$) for your two solutions to the quadex problem via email to **trenchea@pitt.edu**

14 Extra credit

Try to solve the "hockeystick" ODE, which has the form:

$$\begin{aligned} y' &= \frac{-y}{c+y} \\ y(1) &= 1 \end{aligned}$$

where the parameter $c = 0.0001$. The equation is to be solved over the interval $0 \le t \le 4$.

I don't have a formula for the exact solution. But what I do know is that the exact solution is always positive. If you don't use enough steps when trying to solve this problem, you will get a plot that looks like a straight line. But the problem is called the "hockeystick" for a reason!