Implicit ODE Methods
Backward Euler and stiff equations

MATH1090: Directed Study in Differential Equations

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

---

A zigzag path is a sign of trouble!

Implicit methods and stiff equations

A stiff differential equation can cause an ODE solver to produce oscillatory errors unless a very small step size is used. Using an implicit method instead may kill the oscillations.

1 A surprisingly difficult ODE

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

$$y' = 50(\cos(t) - y)$$
$$y(0.0) = 0.0$$

for which the exact solution is

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

and suppose we want estimate the solution over the interval $[0, 1]$ using the Euler method.

We could write the solution as a single program, that allows us to choose the number of steps n, and returns the resulting set of t, y values for tabulating or plotting:

```
1  function [ t, y ] = stiff_euler ( n )
2
3    t = zeros ( n + 1, 1 );
4    y = zeros ( n + 1, 1 );
5
```

```
  a = 0.0;
  b = 1.0;
  dt = ( b - a ) / ( n - 1 );

  t(1) = a;
  y(1) = 0.0;
%
%  We can call a function for dydt, or write the formula.
%
  for i = 1 : n
    t(i+1) = t(i) + dt;
    y(i+1) = y(i) + dt * stiff_prime ( t(i), y(i) );
%   y(i+1) = y(i) + dt * 50.0 * ( cos ( t(i) ) - y(i) );

  end

  return
end
```

Listing 1: Forward Euler method for stiff equation.

For convenience, we might write the derivative as a function:

```
function dydt = stiff_prime ( t, y )
  dydt = 50.0 * ( cos ( t ) - y );
  return
end
```

Listing 2: stiff_prime.m evaluates the right hand side.

and the exact solution:

```
function value = stiff_exact ( t )
  value = 50.0 * ( sin ( t ) + 50.0 * cos(t) - 50.0 * exp ( - 50.0 * t ) ) / 2501.0;
  return
end
```

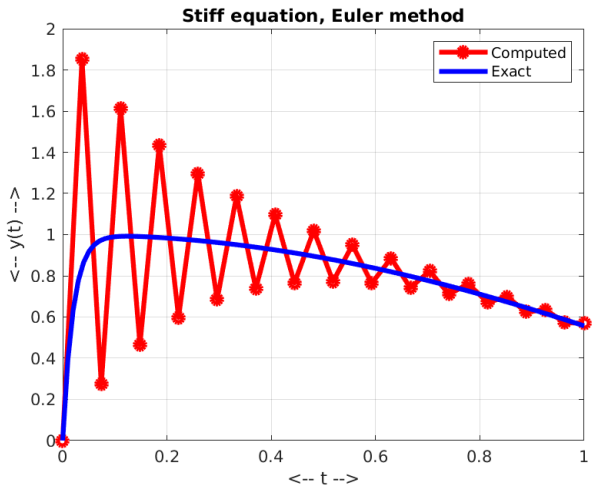
Listing 3: stiff_exact.m evaluates the exact solution.

A value of $n = 27$ might seem reasonable to get from $t = 0.0$ to $t = 1.0$. We can compute the solution, and plot it versus the true solution (on a finer grid) as follows:

```
function stiff_euler_test ( n )
  [ t1, y1 ] = stiff_euler ( n );

  t2 = linspace ( 0.0, 1.0, 101 );
  y2 = stiff_exact ( t2 );

  plot ( t1, y1, 'ro-', t2, y2, 'b-' )
  return
end
```

Listing 4: stiff_euler_test.m computes a forward Euler solution.

The resulting approximate zigzag curve (red) looks nothing like the true smooth solution:



Explicit Euler solution with 27 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. However, this stepsize reduction may be quite severe, even though the shape of the solution curve doesn't give us any warning about how difficult it will be for

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`, we put all the ODE derivative information on the right hand side, evaluated at the previous time `t(i)` and previous solution `y(i)`. And that's what 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.

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" step $i + 1$. In other words, for our stiff example, we replace

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

Listing 5: Explicit Euler update step.

by

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

Listing 6: Implicit Euler update step.

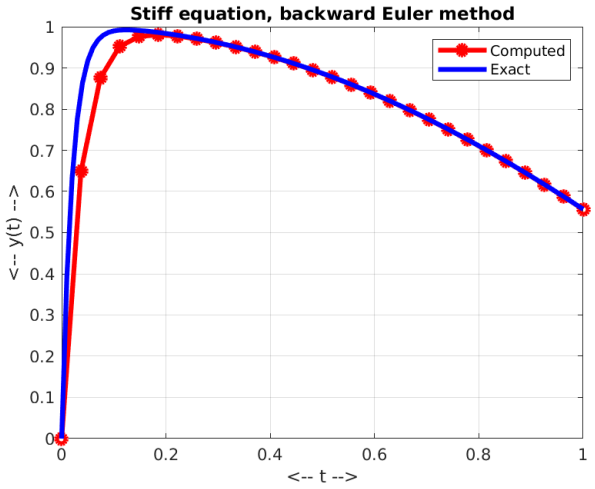
Once we rewrite the update step, we have to rearrange it so that it tells us how to solve for `y(i+1)`. For this case, it's not too hard to work out what the implicit Euler update state should be:

```
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 )
```

```
function stiff_euler_backward_test ( n )
  [ t1, y1 ] = stiff_euler_backward ( n );

  t2 = linspace ( 0.0, 1.0, 101 );
  y2 = stiff_exact ( t2 );

  plot ( t1, y1, 'ro-', t2, y2, 'b-' )
  return
end
```

Listing 7: stiff_euler_backward.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:



Implicit Euler solution with 27 steps.

Just as there are many explicit ODE solvers, there are many implicit ODE solvers. The backward Euler method has only first order accuracy, so if we think an ODE is stiff, and we want high accuracy, we might look for an implicit method of higher order.

2 The Midpoint Method

The midpoint method for solving an ODE is an implicit method. When describing programs, we write `t(i)` and `y(i)`, but mathematically, we write t_i and y_i, and we think of the derivative or ODE right hand side as a function $y'(t, y)$. Using this notation, we can mathematically describe the midpoint method as seeking values that satisfy the following equation:

$$y_{i+1} = y_i + dt * \frac{y'(t_i, y_i) + y'(t_{i+1}, y_{i+1})}{2}$$

In a way, this looks like an explicit Euler method, except that instead of using $y'(t_i, y_i)$ in the update, we use an average of the derivative at the current and future points. Of course, since we don't actually know the future point, this is really an implicit method. If we are lucky, for a particular problem we can rewrite this update formula so that it is simple to see how to determine y_{i+1}.

For our stiff example, we start with the definition, and then solve for y_{i+1}:

$$y_{i+1} = y_i + dt * \frac{y'(t_i, y_i) + y'(t_{i+1}, y_{i+1})}{2}$$

$$y_{i+1} = y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * (\cos(t_{i+1}) - y_{i+1})}{2}$$

$$y_{i+1} + \frac{dt}{2}(50 * y_{i+1}) = y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * \cos(t_{i+1})}{2}$$

$$(1.0 + 25 * dt) * y_{i+1} = y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * \cos(t_{i+1})}{2}$$

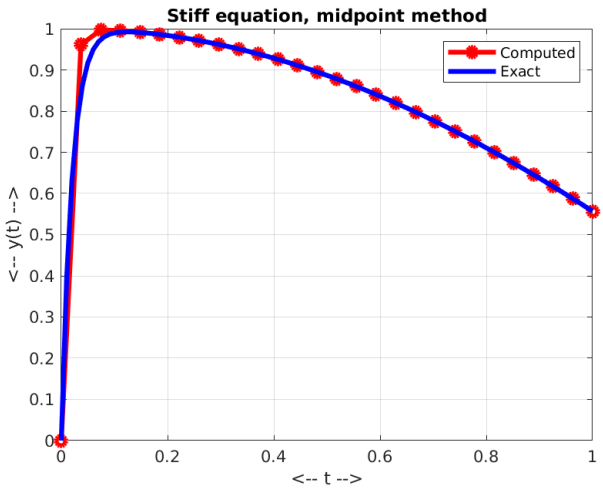
$$y_{i+1} = \frac{y_i + 25 * dt * (\cos(t_i) - y_i + \cos(t_{i+1}))}{1.0 + 25 * dt}$$

It's actually tricky to write this update formula correctly in MATLAB, getting all the parentheses in the right place. So I will write out the solver routine in full, and show the update formula spread out over several lines, so that I can force the open and close parentheses to line up clearly.

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

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

  a = 0.0;
  b = 1.0;
  dt = ( b - a ) / n;

  t(1) = a;
  y(1) = 0.0;

  for i = 1 : n
    t(i+1) = t(i) + dt;
    y(i+1) = ...
      ( ...
        y(i) + 25.0 * dt * ( ...
                               cos ( t(i) ) - y(i) + cos ( t(i+1) ) ...
                             ) ...
      ) ...
      / ( 1.0 + 25.0 * dt );
  end

  return
end
```

Listing 8: stiff_midpoint.m solves the stiff problem.

Now by calling the function *stiff_midpoint_test(n)*, we can compute a new estimate and plot it:



Midpoint solution with 27 steps.

We can guess from the plot that the midpoint method has better accuracy than the backward Euler method. In fact, while the backward Euler method has accuracy $O(dt)$, the midpoint method's accuracy is $O(dt^2)$. This means that, for a given stepsize, we are likely to get better results with the midpoint rule, and if we cut the stepsize in half, the backward Euler error is divided by 2, but the midpoint error is divided by 4. The midpoint method has some other features that make it very attractive for research computing, and so we will be discussing it more in the future.

3 Report: Convergence Comparison

Since we know the exact solution of the *stiff* problem, we can determine the error we make at each step. We summarize the error using a norm. A good norm to use is the RMS norm, which is just the ℓ^2 norm adjusted for the vector size.

We want to verify that the forward Euler and backward Euler methods have an accuracy that is $O(dt)$, whereas the midpoint method's accuracy is $O(dt^2)$. To do this, we will solve the *stiff* problem repeatedly, increasing the number of steps from $n = 20$ to 40, 80, 160, 320. Each time we will compute the RMS norm of the error, and make a table. Next week we will talk about how to analyze the results of the table.

You can modify the codes *stiff_euler_test.m*, *stiff_euler_backward_test.m*, and *stiff_midpoint_test.m* to solve the equation, returning values `t1` and `y1`. Evaluate the exact solution `y2` at the same set of `t1` values, and then compute `e`, the RMS norm of `y2-y1`. For each value of `n` and each ODE solver, record the value of `e`.

n	Euler	Backward Euler	Midpoint
20	.....	.....	.....
40	.....	.....	.....
80	.....	.....	.....
160	.....	.....	.....
320	.....	.....	.....

Bring your completed table to our next meeting, at 2:00pm, Thursday, 13 February, room Thackery 624.