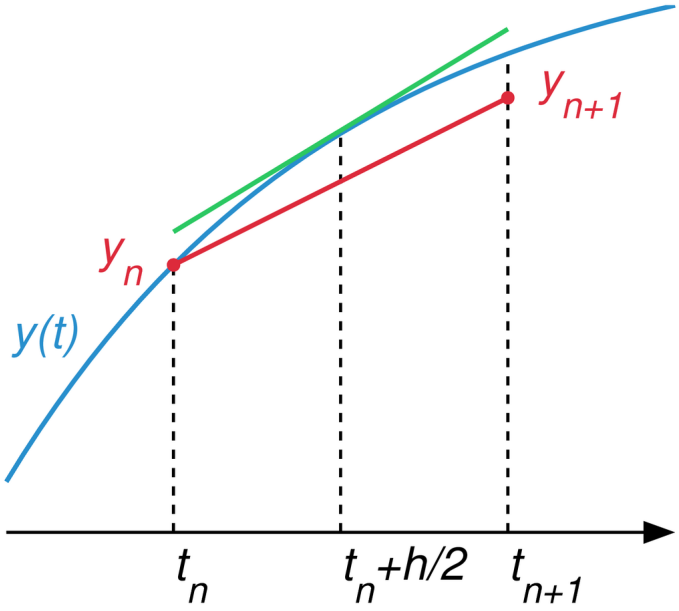
Midpoint and ODE45
Two more ODE Solvers

MATH1902: Numerical Solution of Differential Equations

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

---



The midpoint method approximates the slope in the middle.

More ODE Solvers

Our current choices for ODE solvers include the fixed stepsize versions of the Euler method, Runge Kutta solvers rk1, rk2, rk4, the implicit Euler method, and some simple adaptive stepsize solvers based on rk1 and rk2. We introduce the midpoint method, a more powerful alternative to the backward Euler method, and ode45(), an ODE solver provided with MATLAB, which has high accuracy, powerful error estimation and stepsize adaptation.

1 The Midpoint Method "improves" Backward Euler

We introduced the backward Euler method as a technique that could handle stiff differential equations. The special characteristic of the method is that it is implicit, so that each step requires setting up and solving an implicit equation. This extra work was easy to set up using MATLAB's *fsolve()* function.

Thus the backward Euler method was able to get a good approximation to the solution to our stiff ODE example, while the usual Euler method produced a ridiculously zigzagging result. However, both these methods are only first order - that is, the error decreases like dt. We also saw, in the discussion on conservation, that for some ODE's, it may be important to use a solver that can conserve a quantity such as energy or mass. The backward Euler method can only do this for a conservation quantity that is a linear function of the variables, whereas many physical systems have a conservation law that is a quadratic function.

The midpoint method is an enhanced version of the backward Euler method. It also involves solving an implicit equation, but it has second order accuracy (the error now descreases like dt^2, and it can capture quadratic conservation laws. We have already seen the midpoint method in action in the previous class, but now we will explain how it is defined, and try some more examples.

Mathematically, the midpoint method for solving $y'(t) = f(t, y)$ using a fixed stepsize dt produces a sequence of approximate values, whose $i + 1$-th member should satisfy the following equation:

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

To see where this came from, rewrite it as

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

We can interpret this as the statement that $y'(t_{i+1/2})$ should be well approximated by the average of the derivatives at t_i and t_{i+1}. In fact, this approximation is of one higher degree of accuracy than we had for the forward or backward Euler methods.

To figure out the value of y_{i+1}, we have to solve this implicit equation, so in general we will need to use something like MATLAB's *fsolve()* to crack the implicit equation that results. However, if the ODE is linear, then we can actually work out the solution for this special case. Let's start by looking at such a simpler example.

2 *stiff_midpoint_explicit()*: The Midpoint Method for a Linear ODE

Our stiff ODE example has the form:

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

with initial condition:

$$y(0) = 0$$

Suppose we wish to apply the midpoint method to this problem, using a stepsize dt. Then we have

$$\begin{aligned}
y_{i+1} &= y_i + dt * \frac{y'(t_i, y_i) + y'(t_{i+1}, y_{i+1})}{2} \\
y_{i+1} &= y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * (\cos(t_{i+1}) - y_{i+1})}{2} \\
y_{i+1} + \frac{dt}{2}(50 * y_{i+1}) &= y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * \cos(t_{i+1})}{2} \\
(1.0 + 25 * dt) * y_{i+1} &= y_i + dt * \frac{50 * (\cos(t_i) - y_i) + 50 * \cos(t_{i+1})}{2} \\
y_{i+1} &= \frac{y_i + 25 * dt * (\cos(t_i) - y_i + \cos(t_{i+1}))}{1.0 + 25 * dt}
\end{aligned}$$

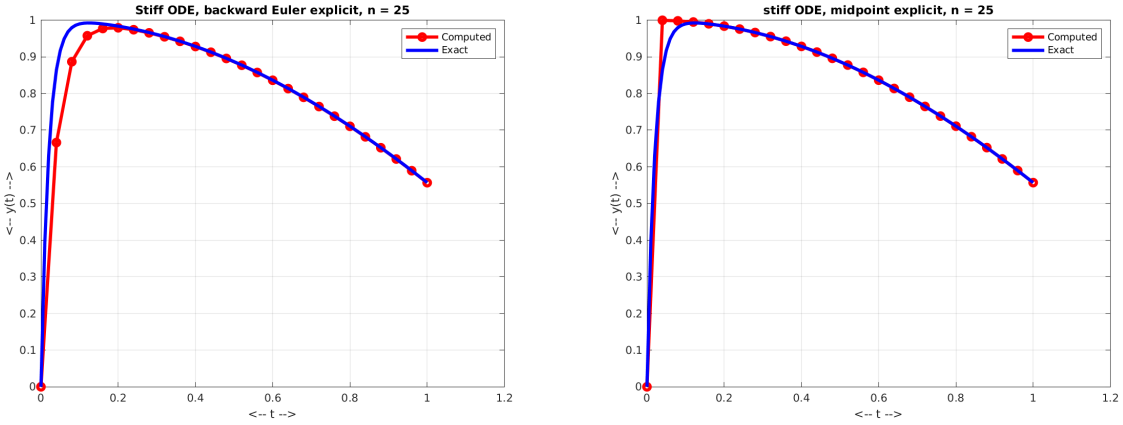
It's actually tricky to write this update formula correctly in MATLAB, getting all the parentheses in the right place. So I will write out the solver routine in full, and show the update formula spread out over several lines, so that I can force the open and close parentheses to line up clearly.

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

```
5
6     t0 = 0.0;
7     tstop = 1.0;
8     dt = ( tstop - t0 ) / n;
9
10    t(1) = t0;
11    y(1) = 0.0;
12
13    for i = 1 : n
14      t(i+1) = t(i) + dt;
15      y(i+1) = ...
16        ( ...
17          y(i) + 25.0 * dt * ( ...
18                                 cos ( t(i) ) - y(i) + cos ( t(i+1) ) ...
19                               ) ...
20        ) ...
21        / ( 1.0 + 25.0 * dt );
22    end
23
24    return
25  end
```

Listing 1: midpoint_explicit.m solves the stiff problem.

This solver is actually part of the file *stiff_midpoint_explicit.m*, which uses it to to compute and plot a solution. We can compare the result from backward Euler result to our new midpoint calculation:



Explicit backward euler and midpoint solutions with $n = 25$ 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 *midpoint()*: A general midpoint solver using *fsolve()*

Except when the right hand side is linear in the variable, most ODE's don't allow us to convert the midpoint method into an explicit formula. So, just as we did for the backward Euler method, we are interested in creating a MATLAB function that can apply the midpoint method to a general problem, while looking

"simple" to the user (same interface as for *euler*, *rk2()* and so on), while taking care of the hard work of solving the implicit equation by calling the MATLAB function *fsolve()*.

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

Listing 2: Signature for the midpoint ODE solver.

The midpoint function will look similar to our other ODE solvers, except that when we need to approximate the solution at the next time, we have to call a hidden function which solves the implicit midpoint equation:

```
function [ t, y ] = midpoint ( 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 )';  %  Initial guess is forward Euler estimate.

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

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

  end

  return
end
```

Listing 3: Text of midpoint.m

Meanwhile *midpoint_residual()* defines a function whose value would is zero if *yp* is the correct midpoint prediction. It is simply this:

```
function value = backward_euler_residual ( f, to, yo, tp, yp )
  value = yp - yo - ( tp - to ) * ( f ( t0, y0) + f ( tp, yp ) )' / 2.0;
  return
end
```

Listing 4: Code to evaluate the midpoint residual.

You don't need to understand why we have to write things this way, but you should be able to see that if *fsolve()* finds a value *yp* for which *midpoint_residual()* returns a value of 0, then we have found our desired result.

4 *stiff_midpoint()* repeats our computation

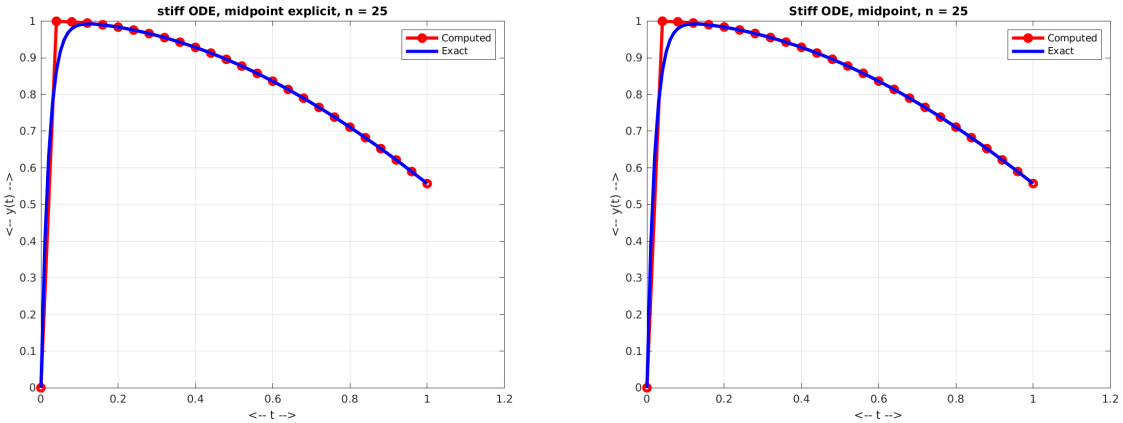
Let's use our general midpoint method to solve the stiff ODE again, and make sure that the explicit and implicit results are identical for this problem.

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

```
function stiff_midpoint ( n )

  dydt = @ stiff_deriv;
  t0 = 0.0;
  tstop = 1.0;
  tspan = [ t0, tstop ];
  y0 = 0.0;
%
%  Same format as if we were calling euler() or rk2() or other of our solvers...
%
  [ t1, y1 ] = midpoint ( dydt, tspan, y0, n );

  return
end
```

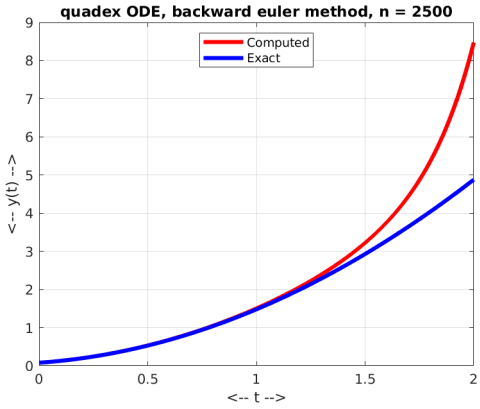


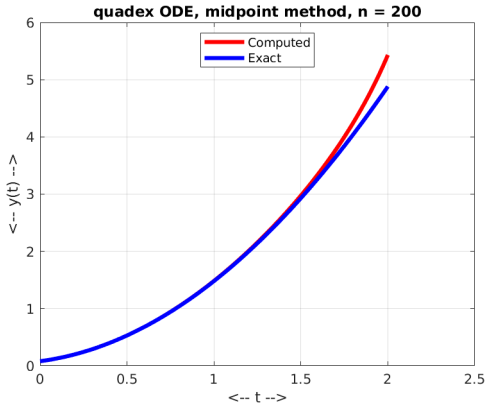
Explicit and implicit midpoint solutions with 25 steps.

5 *quadex_midpoint()* handles the QUADEX ODE

The homework for the class on stiff ODE's asked you to apply the backward Euler method to the QUADEX ODE. This is a difficult problem, and we know that small errors can cause the calculation to go bad. Even with 2,500 steps, the backward Euler method can't hang onto the solution for the whole test range.

The midpoint method is also designed to handle stiff problems, but it has the extra advantage of greater accuracy. Using only $n = 200$ steps over the same interval, we get a much more accurate picture of this tricky problem.





QUADEX solution by backward Euler (2500 steps) and midpoint (200 steps).

6 The Degree of Accuracy

We have claimed that the midpoint method has degree 2, or order of accuracy 2, meaning that the global error should behave like dt^2. It is more clear to say that if we double n or equivalently, cut dt in half, the error should go down by a factor of 1/4. The forward and backward Euler methods have degree 1, meaning halving the stepsize only halves the error.

To get a feeling for what this looks like, and how to investigate the order of accuracy of any ODE solver, we will consider the EXP ODE:

$$\begin{aligned} y' &= y; && \text{ODE} \\ y(0) &= 1; && \text{Initial condition} \\ y(t) &= e^t; && \text{Exact solution} \end{aligned}$$

We will solve this problem over the interval $0 \leq t \leq 5$, and report the RMS error in our solution, using $n = 10, 20, 40, 80$. You can see that the error goes down by roughly a factor of 2 for the Euler codes, but by a factor of 4 for the midpoint rule:

Method	n=10	n=20	n=40	n=80
euler()	33.15	20.21	11.44	5.14
backward_euler()	301.01	52.64	18.16	7.72
midpoint()	6.07	1.27	0.29	0.07

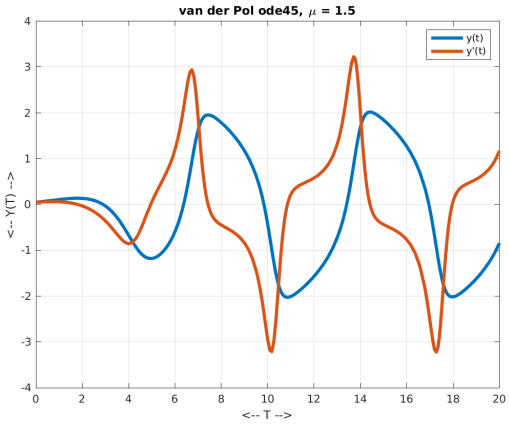
This chart is evidence that our accuracy claim is correct.

7 The Midpoint Method and Research Questions

The midpoint method for solving an ODE is of special interest to Professor Trenchea. Here are a few points of interest.

- People disagree about what method should be called "the midpoint method". Professor Trenchea very firmly believes in the method we have used here. Textbooks and other researchers have used this same name for methods that look similar, differ in small or large details;

- Because the midpoint method has a good conservation and accuracy properties, it is preferred by researchers in areas such as chemistry, physics, magnetism, and even glacial modeling; Professor Trenchea is trying to demonstrate its advantages to other scientists who are using inferior methods;
- Professor Trenchea is very interested in using an adaptive stepsize scheme (more powerful than the ones we looked at) with the midpoint method. This becomes a difficult research problem when the ODE switches rapidly between smooth and varying behavior. You can ask Professor Trenchea about the van der Pol ODE, or look it up on Wikipedia!



An "easy" version of the van der Pol ODE.

8 *ode45()*: MATLAB's ODE Solver

The user interface to *ode45()* is almost the same as I have been using for our example solvers like *euler()* and *rk2()*. In particular, calling *ode45()* for a solution looks almost the same as we are used to:

```
1   [ t, y ] = ode45 ( f, tspan, y0 );
```

where `f` is the name of the user derivative function, `tspan` is a vector of length 2 containing the initial and final times, and `y0` is an ny-vector containing the values of the dependent variables at time `t0`.

Our previous ODE solvers included a final argument `n`, the number of steps to take. But *ode45()* does not accept that value; this is because it uses an adapative stepsize algorithm, and so the number of steps to be taken cannot be determined in advance. But by allowing the stepsize to vary, `ode45()` aims to produce a solution that is as accurate as desired, as efficiently as possible. After the solution is returned, you can determine the number of steps that were taken by a command like this:

```
1   n = length ( t ) - 1;
```

Here, we are subtracting 1 because we want n to count the number of steps, and so we don't wish to include the initial value as a step.

Another thing to keep in mind is how `ode45()` handles a problem involving a system of ny equations. First, it is important that the derivative routine returns a *column* vector; *ode45()* will refuse to proceed if a row vector is returned instead. Thus, for our pendulum ODE problem, the derivative function looks like this:

```
1   function dydt = pendulum_deriv ( t, y )
2
3     [ g, l, m, t0, y0 ] = pendulum_parameters ( );
4
5     u = y(1);
```

```
  v = y(2);

  dudt = v;
  dvdt = -( g / l ) * u;

  dydt = [ dudt; dvdt ];   % <-- the semicolon makes dydt a COLUMN vector.

  return
end
```

Secondly, note that the output of *ode45()* will be the $nt+1$-vector `t` and the $(nt+1) \times ny$-array `y`. The pendulum solution values at time i are `y(i,1)` and `y(i,2)`, or in general `y(i,1:ny)`. To plot the angular deflection of the pendulum problem over time, we want to plot `y(1:nt+1,1)` or `y(:,1)`.

Be sure you know the shape of your data! MATLAB can make a mess of things if you give it a row vector when it wants a column vector; sometimes it prints an error message, and sometimes it thinks you are doing a very fancy linear algebra computation and computes a result you didn't want!

The *ode45()* code adapts the stepsize by estimating the error it has made in the most recent step. It does this by comparing the predictions of Runge Kutta methods of degrees 4 and 5, which is where the "45" comes from in the name. Note that, if you cut the stepsize in half, the error in a solver of degree 5 is divided by 32. This high accuracy means the solver usually does not have to take very small steps to get good results.

The *ode45()* code uses a sophisticated error measurement. After computing the difference of two solution estimates to get an error estimate, it decides whether the error is small or large using a relative scale based on the size of the solution. Our simple error estimation ideas would fail such a test.

Because *ode45()* uses high degree Runge Kutta solvers, it is expecting that the derivative function and solution curve will be smooth, with at least 5 continuous and bounded derivatives. Some ODE's do not have such smooth solutions, and will cause the algorithm to be inefficient or to fail. MATLAB provides alternatives that use lower degree solvers, such as *ode23()* for such cases.

Because *ode45()* uses explicit Runge Kutta solvers, it may perform poorly on stiff problems. When a problem seems to be stiff, MATLAB provides several alternatives that can work better, including *ode23s()*.

9 Homework #9

Set up the pendulum and predator-prey problems using *ode45()*, using the same data that we had for the conservation class. For both problems, count n, the number of steps taken, compute and plot the conserved quantities $h()$, and print the last value $h(n+1)$. (For the pendulum problem, $h()$ is computed by the function *pendulum_energy()*.)
Fill in the chart to compare the *ode45()* results against *midpoint()*. What does this chart suggest about the *ode45()* efficiency and conservation?

Problem	initial h(1)	midpoint h(n+1)	midpoint n	ode45 h(n+1)	ode45 n
pendulum	5.37	5.37	400	.....	.....
predator	3.22	3.25	400	.....	.....

Submit your completed table via email to **trenchea@pitt.edu**