CMPS 251 - Assignment 9

Kinan Dak Al Bab

November 7, 2014

1 ODE models

1.1 Question

Describe an ODE that arises in some context you are familiar with.

1.2 Answer

The Time-independent Schrödinger Equation in one Dimension of a particle.

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x)$$

The equation is a second degree ODE, in which We find both $\psi(x)$ as well as $\psi''(x)$, Where the function $\psi(x)$ gives us the space component of the probability of finding the particle if we looked at it in position x.

This equation is a simplified version of the Time-independent Schrödinger equation in 3 Dimensions, which has partial derivatives, also The space-independent Schrödinger equation is a differential equation that gives the space-independent component of the probability, the values from the two equations can be merged together to give us the full wave (probability) function:

$$\Psi(x) = \psi(x).\phi(x)$$

2 Second-order Runge-Kutta method

2.1 Question

- Derive the second order Runge-Kutta method that uses an average of two slopes at the beginning (t_i) and end (t_{i+1}) of a step to take the actual step as described in section 16.3
- Show that the *local* error of this method is $O(h^3)$, and therefore its *global* error is $O(h^2)$. Hint: Shows that a Taylor series expansion of $y(t)$ and the formula obtained above for $y(t_{i+1})$ match up to the quadratic term. See Exercise 6 on p. 532.

2.2 Solution

So Let us write the formulas of the forward and backwards Euler (respectively):

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + O(h^2) \ (1)$$

$$y(t_{i+1}) = y(t_i) + hf(t_{i+1}, y(t_{i+1})) + O(h^2) \ (2)$$

By adding the two Equations and dividing by 2 we get:

$$y(t_{i+1}) = y(t_i) + h\frac{f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))}{2} + Error \ (3)$$

Notice That we have

$$\frac{f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))}{2} = \frac{y'(t_i) + y'(t_{i+1})}{2}$$

which is the average of the two slopes (as requested). Notice that (3) is the same as the trapezoidal rule. therefore it should have $O(h^3)$ locally and $O(h^2)$ globally.

Let us look at the taylor expansion of $f(t_{i+1}, y(t_{i+1})) = y'(t_{i+1})$:

$$y'(t_{i+1}) = y'(t_i + h) = y'(t_i) + hy''(t_i) + O(h^2)$$

Plug in in formula (3)

$$y(t_{i+1}) = y(t_i) + h\frac{y'(t_i) + y'(t_{i+1})}{2} + Error$$

$$y(t_{i+1}) = y(t_i) + h\frac{y'(t_i) + y'(t_i) + hy''(t_i) + O(h^2)}{2} + Error$$

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + h\frac{hy''(t_i) + O(h^2)}{2} + Error$$

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + h^2\frac{y''(t_i)}{2} + O(h^3) + Error$$

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + h^2\frac{y''(t_i)}{2} + E \ (4)$$

Now let us take the Taylor expansion of $y(t_{i+1})$:

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + h^2\frac{y''(t_i)}{2} + O(h^3)(5)$$

Now by comparing (4) to (5) we get that $E \sim O(h^3)$ so the total local error we have in our method is $O(h^3)$. Remember that $h = \frac{C}{n}$ where C is the length of the interval we are trying to solve in. Therefore $n = \frac{C}{h}$, so The global error would be $Tot = E \times n = E \times \frac{C}{h}$ But $E \sim O(h^3)$ therefore $Tot \sim \frac{C}{h} \times O(h^3) \sim O(h^2)$

3 Implementation and error behavior

3.1 Question

- Write a function
  $[t, y] = ode2(f, interval, y0, h)$ hat solves the ODE: $y' = f(t, y); y(t_0) = y_0$ using the trapezoid method in the problem above.
- Test it on the following ODE in the interval $1 \leq t \leq 10$

  $$y' = -y^2;\ y(1) = 1$$

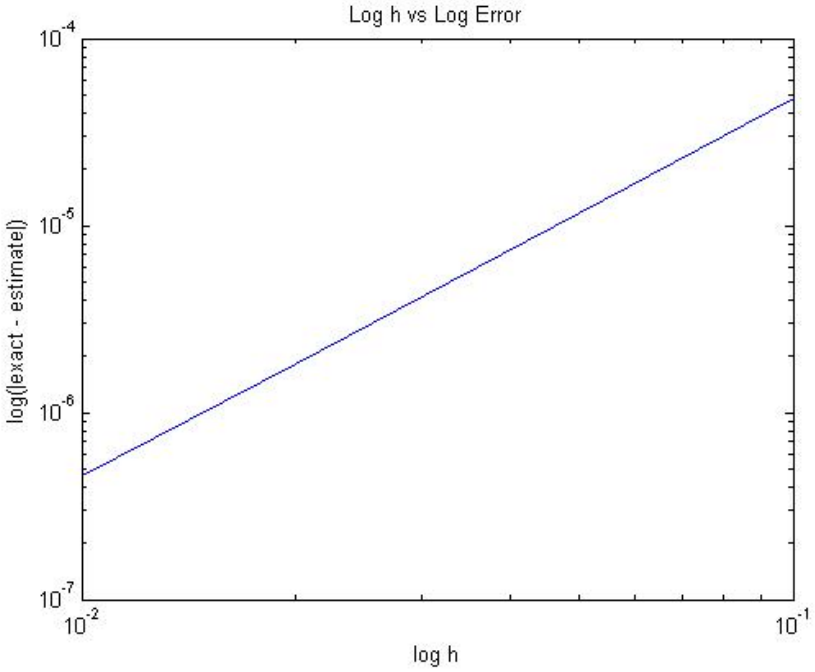
  This equation has the closed-form solution $y = -\frac{1}{t}$. Compute $y(10)$ with $h = 0.1, 0.05, 0.02, 0.01$ and plot the error in it vs h on an appropriate plot. Comment.
- What h should we use in the example above if the desired global error at $y(10)$ is to be less than 10^{-8}.

3.2 Code

```
%% Function
function [t, y] = ode2(f, interval, y0, h)
    a = interval(1);
    b = interval(2);

    n = (b - a)/h;

    t = zeros(1, (b - a)/h);
    y = zeros(1, (b - a)/h);

    t(1) = a;
    y(1) = y0;

    for i = 1:n
        t(i+1) = t(i) + h;
        Y = y(i) + h * feval(f, [t(i), y(i)]);
        y(i+1) = y(i) + ((feval(f, [t(i), y(i)]) + feval(f, ...
            [t(i+1), Y]))/2) * h ;
    end

end

%%Test Script
f = @(x)(-x(2)^2);
exact = 1/10;

interval = [1 10];
y0 = 1;

h = [0.1 0.05 0.02 0.01];
y = zeros(1, 4);
E = zeros(1, 4);

for i = 1:4
    [~, x] = ode2(f, interval, y0, h(i));
    y(i) = x(length(x));
    E(i) = abs(exact - y(i));
```

```
37  end
38
39  loglog(h, E);
40  title('Log h vs Log Error');
41  xLabel('log h');
42  yLabel('log(|exact - estimate|)');
43
44  lh = log(h);
45  lE = log(E);
46  slope = (lE(4) - lE(1))/(lh(4) - lh(1))
47
48  %% Error Discovery
49  ErrorBound = 0.5 * 10^(-8);
50  NeededH = sqrt(ErrorBound / E(1)) * h(1);
51  NeededH = round(NeededH*10000) / 10000 % Round the required h. ...
        to get n to be an integer.
52  [~, X] = ode2(f, interval, y0, NeededH);
53  Y = X(length(X));
54  abs(exact - Y) % Error with the targeted h
```

3.3 Plot and Output




```
slope =

   2.020791479251348


NeededH =

    1.000000000000000e-003
```

```
ans =

    4.502474795775591e-009
```

3.4 Comment

As expected the slope of the line is nearly 2, indicating $O(h^2)$ Global Error Behavior.

The NeededH as we noticed is around 0.001, we round it to the closest 3rd decimal place to avoid having n being not an integer, The error resulting from it is in order 10^{-9} which is what we want. By changing the h used as reference inside the NeededH formula from $h(1)$ to $h(2), h(3) \ldots$ We get the same results.

4 ode23

4.1 Question

Matlab has a rich suite of ODE solvers that are adequate for the solution of a broad range of IVPs. In this exercise, you are asked to use one of the simplest methods, *ode*23(), that uses an adaptive step size in the integration.

- Solve the following first-order ODE: $y' = 3 + 5sin(t) + \sqrt{y}; y(0) = 0$ in the interval $[0, 8]$ using *ode*23().
- Compare the solution with the one obtained using the fixed-step trapezoid method above.

4.2 Code

```
f = @(t, y)(3 + 5 * sin(t) + sqrt(y));
f2 = @(x)(3 + 5 * sin(x(1)) + sqrt(x(2)));
y0 = 0;
interval = [0 8];

[T, Y] = ode23(f, interval, y0);
mFinal = Y(length(Y));

kFinal= [];
for h = [0.1 0.05 0.01 0.005 0.001]
    [t, y] = ode2(f2, interval, y0, h);
    kFinal = [kFinal y(length(y))];
end

diff_ode23 = abs(mFinal - kFinal)

%% COMMENT SECTION VERIFICATION
[T, Y] = ode45(f, interval, y0);
m2Final = Y(length(Y));

```

```
21 diff_ode45 = abs(m2Final - kFinal)
22 ode23_v_ode45 = abs(mFinal - m2Final)
```

4.3 Output

```
diff_ode23 =

   0.052979097834296   0.015985335381487   0.000373846348253   0.001267916630496
  0.001683429189356


diff_ode45 =

   0.054673264710132   0.017679502257323   0.001320320527583   0.000426250245340
  0.000010737686480


ode23_v_ode45 =

   0.001694166875836
```

4.4 Comment

Notice that when we start with h = 0.1 we find that the difference between the two methods is big, as we decrease h this difference increases for a while, then the difference increases again. We expect our trapezoidal solution to converge to that of matlab but it is not, Therefore either one of the two solutions is converging to the correct solution with smaller error, It is very unlikely that it is my method.

However, In science we don't disregard things without verifying, so I tested my method against ode45 which is very accurate. (diff_ode45), My solution is converging to that of ode45, further more the difference between ode23 and ode45 is in 2 order magnitude. Therefore my method is actually outdoing ode23 for small h.

I am pretty sure that ode23 would outdo my method if we provide it with the a designated error as an option, because by doing so we will be forcing it to take smaller and smaller h on most sub-intervals because we will be decreasing the error it is comparing to.

In MATLAB doc for ode23 they mentioned that its accuracy is very low

5 Higher-order ODEs

5.1 Question

An n-th order scalar differential equation may be written as a system of n first-order equations together with n initial conditions. Consider the following second-order IVP

$$y'' = 2t - y' - y^2; y(0) = 0; y'(0) = 0.1;$$

- Convert the problem to a set of two first-order equations.
- Describe (in Matlab pseudo-code only, no need to implement) how Euler's method may be used to solve the problem. **MEH, I WILL IMPLEMENT!**
- Derive the local and global errors of Eulers method.
- Use $ode23()$ to solve the IVP in the interval [0 2].

5.2 Solution

Take $y = g_1;\ y' = g_2;$

$$\frac{d}{dt}\begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} g_2 \\ 2t - g_2 - {g_1}^2 \end{bmatrix}$$

We also get:

$$y(0) = \begin{bmatrix} g_1(0) \\ g_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$$

5.3 Derivation of Algorithm (Pseudo-code

The original Forward Euler for solving one linear differential equation is:

$$y_{i+1} = y_i + hf(t_i, y_i)$$

But here we have a set of two linear equations, so we have to "vectorize" our euler formula:

$$y_{i+1} = \begin{bmatrix} g1_{i+1} \\ g2_{i+1} \end{bmatrix} = \begin{bmatrix} g1_i \\ g2_i \end{bmatrix} + h \begin{bmatrix} g2_i \\ 2t_i - g2_i - g1_i^2 \end{bmatrix}$$

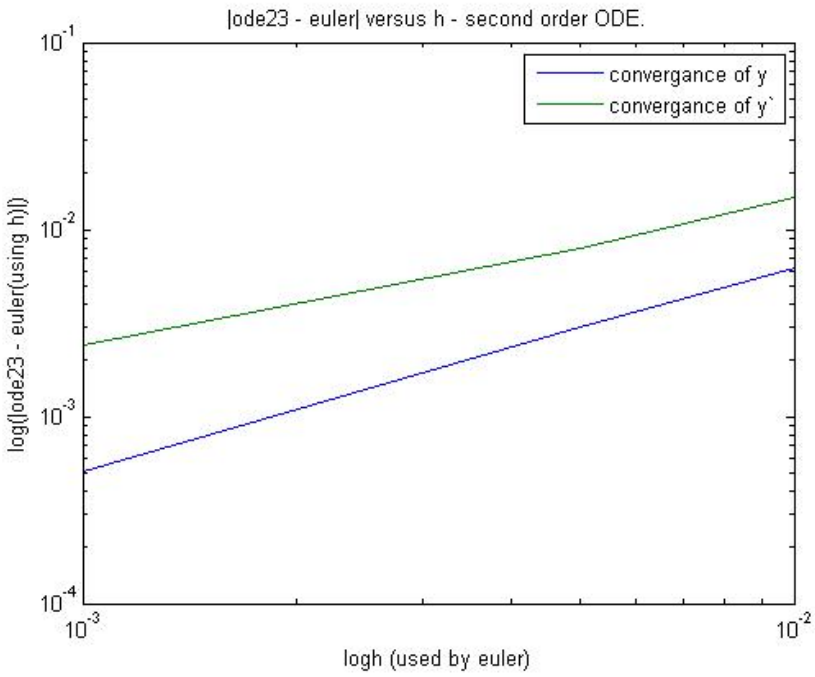
This forumla is explicit and rather easy to code.

5.4 Code (Code is the best Pseudo-Code)

```
ff = @(t, x)([x(2); 2*t - x(2) - x(1)^2]); % Solving using ode23
interval = [0 2];
[¬, Y] = ode23(ff, interval, [g1(1) g2(1)]);

hv = [0.01 0.005 0.001]; % solve using my euler with different ...
    values of h
E = [];
for h = hv
    n = 2 / h;

    t = zeros(1, n);
    g1 = zeros(1, n);
    g2 = zeros(1, n);

    t(1) = 0;
    g1(1) = 0;
    g2(1) = 0.1;

    for i = 1:n
        t(i+1) = t(i) + h;

        vect = feval(ff, t(i), [g1(i) g2(i)]);
        g1(i+1) = g1(i) + h * vect(1);
        g2(i+1) = g2(i) + h * vect(2);

        solution = [g1(i+1) g2(i+1)]; % save the last solution
    end


    E = [E; abs(solution - Y(length(Y),1:2))];
end

E
loglog(hv, E)
```

5.5 Plot and Output

```
E =

   0.006169512323520   0.014880540062800
   0.002996988226483   0.007943121328268
   0.000507185332668   0.002410872106131
```



5.6 Comment

As you can see from the errors printed on the screen, the solution for both y, y' using forward Euler converges towards that solution using built-in *ode*23.

We also care about the speed of convergence, So let us take the error between Euler and buit-in *ode*23, We don't have a given solution for the equation, therefore I am treating the solution from *ode*23 as an "accurate" solution (since i am assuming matlab did their work right). The plot indeed verifies that our solution converges to *ode*23's solution

We can see that the error in our solution of y behaves linearly (slope is nearly 1). which is expected since Euler's method gives us global error behavior $O(h)$, However, y' converges with a line of slope less than 1 (nearly 0.79 if you would calculate it), Which is also not surprising because y''s equation is more complex (so harder to solve) and also because **when solving for y' we are using the values we got for y, and these values already have error in them so the error accumulates**. The general behavior is still close to linear.

5.7 Error Derivation

Let us look at the general form of Euler's Method:

$$Y_{i+1} = Y_i + hf(t_i, Y_i)$$

Where Y can be a vector (System of equations) or a single value (One equation).

We want to know how the error behaves and estimate it.

We know from the definition of ODE that:

$$Y'(t) = f(t, Y(t))$$

By Integrating both sides between t_i and t_{i+1} we get

$$Y(t_{i+1}) - Y(t_i) = \int_{t_i}^{t_{i+1}} f(t, Y(t))$$

To obtain a numerical solution, we can apply a numerical method of calculating the integral in the right side, for Euler we use the rectangular method (left rectangular gives us forward euler).

$$Y(t_{i+1}) - Y(t_i) = hf(t_i, Y(t_i)) + IntegrationError$$

$$Y(t_{i+1}) = Y(t_i) + hf(t_i, Y(t_i)) + IntegrationError$$

And Thus We get the Euler's forward rule

$$Y_{i+1} = Y_i + hf(t_i, Y_i)$$

So we established that the error in the Euler's method is the same as the Integration Error from the rectangle rule. But we have an upper bound on that error (By taking the taylor expansion of f truncated after the first (constant) term, then integrating the constant term plus the error):

$$IntegrationError \leq \frac{||f||_\infty}{2} h^2$$

So the local error behaves $\sim O(h^2)$. Therefor the global error behaves $\sim n \times O(h^2) \sim \frac{b-a}{h} \times O(h^2) \sim O(h)$

□