SIAM student workshop on Matlab and differential equations

Mike Sussman

December 1, 2012
Outline

Introduction

Ordinary Differential Equations (ODEs)
   Options for controlling ode solvers

Partial Differential Equations (PDEs)
   Heat equation
   Burgers’ equation
Who am I?

- Mike Sussman
- **email:** sussmanm@math.pitt.edu
  There is an “m” at the end of “sussman”.
- Thackeray 622
- **Web page:** [http://www.math.pitt.edu/~sussmanm](http://www.math.pitt.edu/~sussmanm)
- Retired from Bettis Laboratory in West Mifflin.
- Part-time instructor at Pitt: 2070, 2071, 3040
- Interests: numerical partial differential equations, particularly the Navier-Stokes equations and applications
Objectives

- Matlab Ordinary Differential Equation (ODE) solvers and application
  - Solving ODEs with default options
  - Writing m-files to define the system
  - Advanced options
- Solving time-dependent Partial Differential Equations (PDEs) using Matlab ODE solvers.
  - Finite-difference discretizations
  - One and two space dimension, one time dimension
Non-objective

- Will not discuss the Matlab PDE toolbox
- GUI for creating complicated 2D mesh
- Limited set of differential equations, not including Navier-Stokes.
- Limited choice of finite element.
Start up Matlab

- Log in
- Start up Matlab
Outline

Introduction

Ordinary Differential Equations (ODEs)
   Options for controlling ode solvers

Partial Differential Equations (PDEs)
   Heat equation
   Burgers’ equation
Initial Value Problem (IVP)

\[ \dot{u} = f(t, u) \]

\[ u(t_0) = u_0. \]

- \( u \in \mathbb{R}^n \)
- \( \dot{u} \) is shorthand for the derivative \( du/dt \)
- *Explicit* because \( \dot{u} \) can be written explicitly as a function of \( t \) and \( u \)
- *First-order* because the highest derivative that appears is the first derivative \( \dot{u} \)
- Higher-order equations can be written as first-order systems
- *IVP* because \( u_0 \) is given and solution is \( u(t) \) for \( t > t_0 \)
Boundary Value Problems (BVP)

- Values specified at both initial and final times
- No special support for BVP in Matlab
- Can use shooting methods
- Can use finite element or finite difference methods
Solutions

- An analytic solution is a formula $u(t) \in C^p$ for some $p$
- A numerical solution of an ODE is a table of times and approximate values $(t_k, u_k)$, possibly with an interpolation rule
- In general, a numerical solution is *always wrong*, and numerical analysis focusses on the error.
Steps for basic solution

\[ \dot{u} = f(t, u) \]
\[ u(t_0) = u_0. \]

1. Write a Matlab m-file to define the function \( f \).
2. Choose a Matlab ODE solver
## Matlab ODE solvers

<table>
<thead>
<tr>
<th>Matlab ODE solvers and support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode23</td>
</tr>
<tr>
<td>ode113</td>
</tr>
<tr>
<td><strong>ode15s</strong></td>
</tr>
<tr>
<td>ode23s</td>
</tr>
<tr>
<td>ode23t</td>
</tr>
<tr>
<td>ode23tb</td>
</tr>
<tr>
<td><strong>ode45</strong></td>
</tr>
<tr>
<td>odeset</td>
</tr>
<tr>
<td>odeget</td>
</tr>
</tbody>
</table>
What is a DAE?

\[ G(t, u, \frac{du}{dt}) = 0 \]

or

\[ M \frac{du}{dt} = f(t, u) \]

where \( M \) is not an invertible matrix or where \( G \) cannot be solved for \( du/dt \).
Learning strategy

1. Do simple and “known good” examples
2. Pick a simple example similar to the objective but that has a known, steady solution. Be sure the steady solution holds for both very short and long times. Check that $f(t, u) = 0$.
3. If there is a known unsteady solution, test it.
4. If there is no known unsteady solution, change conditions and see if the solution is well-behaved. Check that all known theoretical conditions are reflected in the solution.
5. Go on to progressively harder and more realistic examples.
Debugging strategy

What if my steady example won’t work?

- Choose a ridiculously short time and see if it works.
- Try one Euler explicit time step with a very small $\Delta t$
  \[ u^{n+1} = u^n + \Delta t \, f(t^n, u^n) \]
  If this doesn’t work, $f(t, u) \neq 0$.
- Try several Euler explicit steps.
- If Euler explicit works but Matlab does not, you are probably using Matlab wrong. Check everything.
- **Warning:** Matlab will call your $f(t, u)$ many times in order to compute an approximate Jacobian matrix. If that matrix is not a good approximation, the Matlab solvers will not work.
A first example

\[ \frac{du}{dt} = \sin t - u \]

\[ u(0) = 1 \]

The Matlab m-file is **ex1_ode.m**.

```matlab
function udot=ex1_ode(t,u)
% udot=ex1_ode(t,u)
% computes the right side of the ODE du/dt=sin(t)-u
% t,u are scalars
% udot is value of du/dt

udot=sin(t)-u;

Use this command line:

ode45(@ex1_ode,[0,15],1)
```
More first example

If you want to get access to the solution values, use the following command line

\[ [t, u] = \text{ode45}(\text{@ex1_ode}, [0, 15], 1); \]

You can then plot it using the normal plot commands

\[ \text{plot}(t, u) \]

or compare it with other solutions

\[ \text{plot}(t, u, t, \text{sin}(t)) \]
A stiff example

- Widely-different time scales
- Modify example 1 to be \( \dot{u} = 1000 \times (\sin(t) - u) \)
- Changing name of function requires changing name of file!
- \texttt{ex2_ode.m}
- Looks like \( \sin t \).
  \[
  [t,u] = \text{ode45}(@\text{ex2}_{\text{ode}},[0,15],1);
  \text{plot}(t,u,t,\sin(t))
  \]
- But different for first 100 time steps!
  \[
  \text{plot}(t(1:100),u(1:100),t(1:100),\sin(t(1:100)))
  \]
How to tell stiffness

- Easiest way is to time a non-stiff solver and a stiff solver
  \[
  \text{tic;} \ [t45,u45]=\text{ode45}(@\text{ex2\_ode},[0,15],1); \text{toc}
  \]
  \[
  [t15s,u15s]=\text{ode15s}(@\text{ex2\_ode},[0,15],1);
  \text{tic;} \ [t15s,u15s]=\text{ode15s}(@\text{ex2\_ode},[0,15],1); \text{toc}
  \]
  Never time the first use of a function!

- Solution is same.
  \[
  \text{plot}(t45,u45,t15s,u15s)
  \]

- Time difference comes from number of steps
  \[
  \text{length}(u45)
  \]
  \[
  \text{length}(u15s)
  \]

- Visual depiction of number of steps
  \[
  \text{plot}(t45,u45,'b*',t15s,u15s,'y*')
  \]
High order ODEs as systems

Consider a fourth-order differential equation

$$5 \frac{d^4 u}{dt^4} + 4 \frac{d^3 u}{dt^3} + 3 \frac{d^2 u}{dt^2} + 2 \frac{du}{dt} + u = \sin(t)$$

The first step is to define new variables

$$v_1 = u$$
$$v_2 = \frac{du}{dt}$$
$$v_3 = \frac{d^2 u}{dt^2}$$
$$v_4 = \frac{d^3 u}{dt^3}$$

and write

$$\dot{v}_1 = v_2$$
$$\dot{v}_2 = v_3$$
$$\dot{v}_3 = v_4$$
$$\dot{v}_4 = \frac{(\sin x - v_1 - 2v_2 - 3v_3 - 4v_4)}{5}$$
van der Pol's equation

\[ \ddot{u} + a(u^2 - 1)\dot{u} + u = 0 \]

- Assume \( a > 0 \). We will use \( a = 3 \).
- \( u < 1 \), system behaves as negatively-damped oscillator
- \( u > 1 \), system behaves as damped oscillator
- tunnel diodes, beating heart
van der Pol solution

- **ex3_ode.m**
  
  ```matlab
  function udot=ex3_ode(t,u)
  % udot=ex3_ode(t,u)
  % van der Pol ode with A=1

  A=3;
  % udot MUST be a column vector
  udot=[ u(2)
        -A*(u(1)^2-1)*u(2)-u(1)];
  ```

- There are other ways to make column vectors.
- Not stiff. Use ode45
  
  ```matlab
  [t u]=ode45(@ex3_ode,[0,75],[1;0]);
  ```

- First component of solution is $u$, second is $\dot{u}$
  
  ```matlab
  plot(t,u(:,1))
  ```
Options: odeset

<table>
<thead>
<tr>
<th>Option name</th>
<th>value</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsTol</td>
<td>positive scalar or vector</td>
<td>1e-6</td>
</tr>
<tr>
<td>RelTol</td>
<td>positive scalar</td>
<td>1e-3</td>
</tr>
<tr>
<td>OutputFcn</td>
<td>function_handle</td>
<td></td>
</tr>
<tr>
<td>OutputSel</td>
<td>vector of integers</td>
<td></td>
</tr>
<tr>
<td>Stats</td>
<td>on</td>
<td>off</td>
</tr>
<tr>
<td>InitialStep</td>
<td>positive scalar</td>
<td></td>
</tr>
<tr>
<td>MaxStep</td>
<td>positive scalar</td>
<td></td>
</tr>
<tr>
<td>MaxOrder</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Jacobian</td>
<td>matrix</td>
<td>function_handle</td>
</tr>
<tr>
<td>JPattern</td>
<td>sparse matrix</td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td>on</td>
<td>off</td>
</tr>
<tr>
<td>Mass</td>
<td>matrix</td>
<td>function_handle</td>
</tr>
<tr>
<td>MvPattern</td>
<td>sparse matrix</td>
<td></td>
</tr>
<tr>
<td>MassSingular</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>InitialSlope</td>
<td>vector</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>function_handle</td>
<td></td>
</tr>
</tbody>
</table>
Using options

- **Use all default options:**
  
  ```matlab
  ode45(@ex3_ode,[0,15],[1;0]);
  ```

- **Change one option:**
  
  ```matlab
  opt=odeset('OutputSel',1);
  ode45(@ex3_ode,[0,15],[1;0],opt);
  ```

- **Change several options:**
  
  ```matlab
  opt=odeset('OutputSel',1,'RelTol',1.e-5);
  ode45(@ex3_ode,[0,15],[1;0],opt);
  ```

- **Alternative**

  ```matlab
  opt=odeset('OutputSel',1);
  opt=odeset(opt,'RelTol',1.e-5);
  ```
A word about tolerance

- **RelTol** and **AbsTol**:

  \[ e_i \leq \max\{(\text{RelTol})|y_i|, (\text{AbsTol})_i\} \]

- If **NormControl** is on, then tolerance is done using norms, not componentwise.
Some other options

- **Vectorized** if ODE function is coded so that $F(t, [y_1 \ y_2 \ \ldots])$ returns $[F(t, y_1) \ F(t, y_2) \ \ldots]$
- **Events** is discussed below
- **Refine** to interpolate between points
- **Stats** for printed statistics
- **Jacobian** is critical when extremely stiff
Extra parameters

- **ex4_ode.m**

  ```matlab
  function udot=ex4_ode(t,u,a)
  \% udot=ex4_ode(t,u,a)
  \% van der Pol ode with parameter = a
  \% default value of a is 1
  
  if nargin < 3
    a=3;
  end
  
  udot=[ u(2)
        -a*(u(1)^2-1)*u(2)-u(1)];
  ```

- **Timing test ...**

  ```matlab
  tic;[t,u]=ode45 (@ex4_ode,[0,750],[1;0],[],3);toc
  tic;[t,u]=ode15s(@ex4_ode,[0,750],[1;0],[],3);toc
  tic;[t,u]=ode45 (@ex4_ode,[0,750],[1;0],[],50);toc
  tic;[t,u]=ode15s(@ex4_ode,[0,750],[1;0],[],50);toc
  ```
Solutions must be critically evaluated!
In this case, default options are too coarse to pick up the nonzero initial value!

```matlab
opt=odeset('AbsTol',1.e-14,'RelTol',1.e-10);
[td,ud]=ode15s(@ex4_ode,[0,200],[1.e-5;0],[],100);
[to,uo]=ode15s(@ex4_ode,[0,200],[1.e-5;0],opt,100);
plot(td,ud(:,1),to,uo(:,1))
```
Critical events

- **ex4_event.m** (look for peak)

  ```
  function [value, isterminal, direction] = ex4_event(t, u, dummy)
  % [value, isterminal, direction] = ex4_event(t, u, dummy)
  
  % event is when value becomes zero
  value = u(2);  % event is when derivative becomes 0
  direction = -1; % event is when derivative is decreasing
  isterminal = 1; % terminate at event
  ```

- Use any of the integrators with it

  ```
  opt = odeset('OutputSel', 1, 'Events', @ex4_event);
  ode45(@ex4_ode, [0, 25], [2.5; 0], opt);
  ```

- To pick up and continue for one more cycle:

  ```
  [t0, u0] = ode45(@ex4_ode, [0, 25], [2.5; 0], opt);
  [t1, u1] = ode45(@ex4_ode, [t0(end), 25], u0(end,:), opt);
  plot(t0, u0(:,1), 'b', t1, u1(:,1), 'r');
  ```
Outline

Introduction

Ordinary Differential Equations (ODEs)
Options for controlling ode solvers

Partial Differential Equations (PDEs)
Heat equation
Burgers’ equation
Heat equation: continuous

- Imagine a rod of some sort of metal,
  - Part of it might be heated in some manner
  - Its ends are kept at a constant temperature
  - It starts out with some distribution of temperature

- Temperature is $u(x, t)$, $x_{\text{left}} \leq x \leq x_{\text{right}}$, and $t \geq t_{\text{initial}}$.

$$ \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( k(u, x, t) \frac{\partial u}{\partial x} \right) + f(x, t) $$

- Boundary conditions $u(x_{\text{left}}, t) = u_{\text{left}}(t)$, $u(x_{\text{right}}, t) = u_{\text{right}}(t)$.
- Initial condition $u(x, t_{\text{initial}}) = u_{\text{initial}}(x)$. 
Spatial discretization

\[ u_0 = 0 \quad u_1 \quad u_2 \quad u_3 \quad \ldots \quad u_{N-1} \quad u_N = 0 \]

\[ x_0 = 0 \quad x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_{N-1} \quad x_N = 1 \]

\[ x_{\text{left}} = 0, \quad x_{\text{right}} = 1, \quad t_{\text{initial}} = 0, \quad k(x, t) = 2x + t + 1 \]

\[ u_{\text{left}} = u_{\text{right}} = 0 \]

Choose \( N \), and set \( \Delta x = 1/N \)

\[ x_n = n\Delta x \text{ for } n = 0, 1, \ldots, N \]

\[ u_n(t) \approx u(x_n, t). \]

\[ \ddot{u}_n = \frac{1}{\Delta x} \left[ k(x_{n+1/2}, t) \left( \frac{u_{n+1} - u_n}{\Delta x} \right) - k(x_{n-1/2}, t) \left( \frac{u_n - u_{n-1}}{\Delta x} \right) \right] \]
Matlab spatial discretization

\[
\dot{u}_n = \left[ k(x_{n+1/2}, t) \left( \frac{u_{n+1} - u_n}{\Delta x} \right) - k(x_{n-1/2}, t) \left( \frac{u_n - u_{n-1}}{\Delta x} \right) \right] / \Delta x
\]

dx = 1/N;
for n = 1:N-1
  kright = (2*(x(n)+dx/2)+t+1);
  kleft = (2*(x(n)-dx/2)+t+1);
  if n == 1 %left
    udot(n,1) = (kright*(u(n+1)-u(n))-kleft*(u(n)-uleft))/dx^2;
  elseif n < N-1 %interior
    udot(n,1) = (kright*(u(n+1)-u(n))-kleft*(u(n)-u(n-1)))/dx^2;
  elseif n == N %right
    udot(n,1) = (kright*(uright-u(n))-kleft*(u(n)-u(n-1)))/dx^2;
  else %impossible
    error('Error in ex5_ode: bad value of n')
  end
end

Full code is in \texttt{ex5_ode.m}
Heat equation results

\[ [t,u] = \text{ode15s}(@\text{ex5}\_\text{ode},[0,.1,.2,.3,.4,.5],100\times\text{ones}(99,1)); \]
figure(1)
for \( k=1:6 \)
    plot(u(k,:))
    hold on
end
hold off
title('Temperature distribution at several times')
figure(2)
plot(u(:,50))
xlabel('time')
ylabel('temperature')
title('Temperature vs. time in the middle')
Burgers’ equation

\[ \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \]

With \( u_{\text{left}} = 1 \) and \( \frac{\partial u}{\partial x}\bigg|_{\text{right}} = 0 \) and \( u \in [0, 1] \).

\[ \frac{du_n}{dt} = \nu \frac{u_{n+1} - 2u_n + u_{n-1}}{\delta x^2} - u_n \frac{u_{n+1} - u_{n-1}}{2\Delta x} \]

With \( u_{\text{left}} = 1 \) and \( \frac{\partial u}{\partial x}\bigg|_{\text{right}} = 0 \) approximated by \( u_{\text{right}+} = u_N \).
Note: Now there is an $x_N$ because of the new boundary condition.

dx=1/N;
x=(1:N)*dx;
for n=1:N
  if n==1 %left
    udot(n,1)=n*(u(n+1)-2*u(n)+uleft )/dx^2- ...
    u(n)*(2*u(n)+uleft )/(2*dx);
  elseif n<N %interior
    udot(n,1)=nu*(u(n+1)-2*u(n)+u(n-1))/dx^2- ...
    u(n)*(u(n+1)-u(n-1))/(2*dx);
  else n==N %right
    % approximate Neumann b.c.
    uright=u(n);
    udot(n,1)=nu*(uright-2*u(n)+u(n-1))/dx^2- ...
    u(n)*(uright-u(n-1))/(2*dx);
  else %impossible
    error('Error in ex6_ode: bad value of n')
  end
end
Running ex6

- $\nu = .001$ solution is relatively smooth.
  
  ```matlab
  N=500;
  init=(1-linspace(0,1,N)).^3;
  nu=.001;
  [t,u]=ode45(@ex6_ode,0:.01:1.5,init,[],nu);
  ```

- See the wave steepen as it moves
  
  ```matlab
  for k=1:length(u(:,1));plot(u(k,:));
  axis([0,N,0,2]);pause(.1);end
  ```

- $\nu = .0001$ causes numerics to break down: need much finer mesh
  
  ```matlab
  nu=.0001;
  [t,u]=ode45(@ex6_ode,0:.01:1.5,init,[],nu);
  ```

- See the oscillations grow
  
  ```matlab
  for k=1:length(u(:,1));plot(u(k,:));
  axis([0,N,0,2]);pause(.1);end
  ```