Instructors: Prof. Christopher Bergevin (cberge@yorku.ca)

Schedule: Lecture: MWF 11:30-12:30 (CLH M)

Website: http://www.yorku.ca/cberge/2030W2018.html
To the best of our ability, determine the time according to this clock....

How about now?
How about now?

7:39

07:41:06

Keep in mind this notion of error associated w/ our estimate of the current time...
Runge-Kutta (RK)

- ‘Higher order’ methods improve in a similar fashion to Riemann sums, for example:
  - Euler $\rightarrow$ LEFT
  - Modified Euler $\rightarrow$ MID
  - Improved Euler $\rightarrow$ TRAP

- Most popular RK method is the ‘fourth order’ (RK4) and is equivalent to SIMP:

\[
\begin{align*}
  f_0 &= f(x_0, y_0), \\
  f_1 &= f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2} f_0), \\
  f_2 &= f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2} f_1), \\
  f_3 &= f(x_0 + h, y_0 + hf_2).
\end{align*}
\]

\[
y(x_0 + h) = y(x_0) + \frac{h}{6} (f_0 + 2f_1 + 2f_2 + f_3)
\]

(More advanced) RK methods

- Basic gist: ‘adapt’ the step size and see if error ($\varepsilon$) increases or decreases. Leads to the Runge-Kutta-Fehlberg method:

When the smoke clears....

\[ f_0 = f(x_0, y_0), \]
\[ f_1 = f(x_0 + \frac{h}{4}, y_0 + \frac{h}{4} f_0), \]
\[ f_2 = f(x_0 + \frac{3h}{8}, y_0 + \frac{3h}{32} f_0 + \frac{9h}{32} f_1), \]
\[ f_3 = f(x_0 + \frac{12h}{13}, y_0 + \frac{1932h}{2197} f_0 - \frac{7200h}{2197} f_1 + \frac{7296h}{2197} f_2), \]
\[ f_4 = f(x_0 + h, y_0 + \frac{439h}{216} f_0 - 8hf_1 + \frac{3680h}{513} f_2 - \frac{845h}{4104} f_3), \]
\[ f_5 = f(x_0 + \frac{h}{2}, y_0 - \frac{8h}{27} f_0 + 2hf_1 - \frac{3544h}{2565} f_2 + \frac{1859h}{4104} f_3 - \frac{11h}{40} f_4). \]

- Can directly assess error (and set some sort of tolerance)

\[
Err = \hat{y} - y = h \left( \frac{1}{360} f_0 - \frac{128}{4725} f_2 - \frac{2197}{75240} f_3 + \frac{1}{50} f_4 + \frac{2}{55} f_5 \right).
\]

\[
h_{\text{new}} = 0.9h \sqrt[4]{\frac{|h|\varepsilon}{|y(x_0 + h) - \hat{y}(x_0 + h)|}}.
\]

→ At the most basic level, at least there is a clear numerical recipe one could cook up here!

→ allows step-size to be adjusted on the fly!
Visual summary

Euler method

‘Modified’ Euler method

‘Improved’ Euler method

Runge-Kutta

Devries (1994)
Kutz (2013)
Stability

- When we ‘broke’ dfield, why did some solutions completely miss the mark?

→ We can reproduce this divergence by making step-size too big
Error types

- **Truncation error** (also called discretization error) arises by virtue of approximating an infinite series with a finite number of terms

\[
y(x_0 + \Delta x) = y(x_0) + y'(x_0)\Delta x + \frac{1}{2!} y''(x_0)(\Delta x)^2 + \frac{1}{3!} y^{(3)}(x_0)(\Delta x)^3 + \cdots
\]
Truncation vs. Rounding error

- **Truncation error** (also called discretization error) arises by virtue of approximating an infinite series with a finite number of terms

\[ y(x_0 + \Delta x) = y(x_0) + y'(x_0)\Delta x + \frac{1}{2!} y''(x_0)(\Delta x)^2 + \frac{1}{3!} y^{(3)}(x_0)(\Delta x)^3 + \cdots \]

- **Rounding error** (also called quantization or representation error) stems from the finite memory used to represent a number

> see http://mathworld.wolfram.com/RoundoffError.html

- Two types: Local vs Global (i.e., cumulative) error

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Local error $\epsilon_k$</th>
<th>Global error $E_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler</td>
<td>$O(\Delta t^2)$</td>
<td>$O(\Delta t)$</td>
</tr>
<tr>
<td>Second-order Runge–Kutta</td>
<td>$O(\Delta t^3)$</td>
<td>$O(\Delta t^2)$</td>
</tr>
<tr>
<td>Fourth-order Runge–Kutta</td>
<td>$O(\Delta t^5)$</td>
<td>$O(\Delta t^4)$</td>
</tr>
<tr>
<td>Second-order Adams–Bashforth</td>
<td>$O(\Delta t^3)$</td>
<td>$O(\Delta t^2)$</td>
</tr>
</tbody>
</table>

- Euler $\Rightarrow$ LEFT
- Modified Euler $\Rightarrow$ MID
- Improved Euler $\Rightarrow$ TRAP
- RK4 $\Rightarrow$ SIMP

> Euler
> Modified Euler
> Improved Euler
> RK4
Rounding error & Precision

- When we specify a variable in which to store a variable, we must tell the computer how much memory (i.e., precisely how many bits) to allow for such

- For binary representation:
  - Single precision (32 bits)
  - Double precision (64 bits)
  - long double (128 bits)
  - ...
  - ASCII (7-8 bits)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Representation</th>
<th>Approximation</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/7</td>
<td>0.142 857</td>
<td>0.142 857</td>
<td>0.000 000 142 857</td>
</tr>
<tr>
<td>ln 2</td>
<td>0.693 147 180 559 945 309 41...</td>
<td>0.693 147</td>
<td>0.000 000 180 559 945 309 41...</td>
</tr>
<tr>
<td>log10 2</td>
<td>0.301 029 995 663 981 195 21...</td>
<td>0.3010</td>
<td>0.000 029 995 663 981 195 21...</td>
</tr>
<tr>
<td>3/2</td>
<td>1.259 921 049 894 873 164 76...</td>
<td>1.25992</td>
<td>0.000 049 894 873 164 76...</td>
</tr>
<tr>
<td>√2</td>
<td>1.414 213 562 373 095 048 80...</td>
<td>1.41421</td>
<td>0.000 035 623 730 95 048 80...</td>
</tr>
<tr>
<td>e</td>
<td>2.718 281 828 459 045 235 36...</td>
<td>2.718 281 828 459 045</td>
<td>0.000 000 000 000 000 235 36...</td>
</tr>
<tr>
<td>π</td>
<td>3.141 592 653 589 793 238 46...</td>
<td>3.141 592 653 589 793</td>
<td>0.000 000 000 000 000 238 46...</td>
</tr>
</tbody>
</table>

8 bits = 1 byte
10^6 bytes = 1 MB

See also http://www.mathworks.com/help/symbolic/digits.html
Error types

What “type” of error is at play here?
Stability (for numerically solving ODEs)

Consider a simple example:

$$\frac{dy}{dt} = \lambda y \quad y(0) = y_0$$

(known) solution:

$$y(t) = y_0 \exp(\lambda t)$$

Euler’s method:

$$y_{n+1} = y_n + \Delta t \cdot \lambda y_n = (1 + \lambda \Delta t)y_n$$

$$y_N = (1 + \lambda \Delta t)^N y_0$$  value after N steps

Due to rounding error \((e)\), we in fact will have

$$y_N = (1 + \lambda \Delta t)^N (y_0 + e)$$

Leaving us with total error

$$E = (1 + \lambda \Delta t)^N e$$
Stability

Consider the case: \[ \lambda < 0 \]

As \[ N \to \infty \] Then \[ y_N \to 0 \]

But.....

I: \[ |1 + \lambda \Delta t| < 1 \] then \[ E \to 0 \]

II: \[ |1 + \lambda \Delta t| > 1 \] then \[ E \to \infty \]

→ So it’s possible for the solution to diverge (even though it should converge)!

\[ z = \lambda \Delta t \]

let \( y (=z) \) be complex

**Figure 7.3**: Regions for stable stepping (shaded) for the forward Euler and backward Euler schemes. The criteria for instability are also given for each stepping method.

**Note**: Simply decreasing step-size is not a solution (as such is what lead to truncation error)
### Built-in solvers

- **Warning:** Beware the black box!

<table>
<thead>
<tr>
<th>Solver</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ode23</strong></td>
<td>An explicit, one-step Runge-Kutta low-order (2–3) solver. Suitable for problems that exhibit mild stiffness, problems where lower accuracy is acceptable, or problems where $f(t, y)$ is not smooth (e.g., discontinuous).</td>
</tr>
<tr>
<td><strong>ode45</strong></td>
<td>An explicit, one-step Runge-Kutta medium-order (4–5) solver. Suitable for nonstiff problems that require moderate accuracy. <em>This is typically the first solver to try on a new problem.</em></td>
</tr>
<tr>
<td><strong>ode113</strong></td>
<td>A multistep Adams-Bashforth-Moulton PECE solver of varying order (1–13). Suitable for nonstiff problems that require moderate to high accuracy involving problems where $f(t, y)$ is expensive to compute. Not suitable for problems where $f(t, y)$ is not smooth (i.e., where it is discontinuous or has discontinuous lower-order derivatives).</td>
</tr>
<tr>
<td><strong>ode23s</strong></td>
<td>An implicit, one-step modified Rosenbrock solver of order 2. Suitable for stiff problems where lower accuracy is acceptable, or where $f(t, y)$ is discontinuous. <em>Stiff problems are generally described as problems where the underlying time constants vary by several orders of magnitude or more.</em></td>
</tr>
<tr>
<td><strong>ode15s</strong></td>
<td>An implicit, multistep numerical differentiation solver of varying order (1–5). Suitable for stiff problems that require moderate accuracy. <em>This is typically the solver to try if ode45 fails or is too inefficient.</em></td>
</tr>
</tbody>
</table>
Built-in solvers

What is a numerically ‘stiff’ problem?

“Stiffness is a subtle, difficult, and important - concept in the numerical solution of ordinary differential equations.”

“An ordinary differential equation problem is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results.”

“Stiffness is an efficiency issue. If we weren't concerned with how much time a computation takes, we wouldn't be concerned about stiffness. Nonstiff methods can solve stiff problems; they just take a long time to do it.”

ode45

- Uses a ‘one-step’ 4\textsuperscript{th} (5\textsuperscript{th}?!) order Runge-Kutta method

- Requires a bit more convoluted syntax → Typically uses two files

\[
\frac{dP}{dt} = kP \left( 1 - \frac{P}{L} \right)
\]

**Logistic equation**

**ODErkEX1.m**

```matlab
k= 1;   % intrinsic growth rate (const.)
L= 5;   % carrying capacity (const.)
Pinit= L/15;   % initial condition at tI(1)
tI= [0 10];  % time boundaries

% ************************
[tM,PM] = ode45(@(t,P) logistic(t,P,k,L),tI, Pinit);
plot(tM,PM,'kx','LineWidth',2);
```

**logistic.m**

```matlab
function Pdot=logistic(t,P,k,L)
% Logistic equation
Pdot= k*P*(1-P/L);
```

→ define equation via external function, then call that when invoking ode45
% numerically solve the Logistic equation
% \( \dot{P}(t) = kP(1-P/L) \)

% Program calculates in four ways: 1&2. solve explicitly using Euler and RK4,
% 3. solve via ode45 (via external function call) and 4. actual solution

clear; figure(1); clf

% User Inputs
k= 1;   % intrinsic growth rate (const.)
L= 5;   % carrying capacity (const.)
Pinit= L/15;   % initial condition at tI(1)
tI= [0 10];  % time boundaries
stepsize= 0.05;     % for RK4

% Runge-Kutta 4 (and Euler’s method)
m=1;  % counter
for j=tI(1):stepsize:tI(2)
    if j == tI(1)
        P(m)= Pinit;    Peuler(m)= Pinit;
    else
        P0= k*P(m-1)*(1-P(m-1)/L);  % deriv. at y=y0 (last meas.)
P1= k*(P(m-1)+(stepsize/2)*P0)*(1-(P(m-1)+(stepsize/2)*P0)/L);
P2= k*(P(m-1)+(stepsize/2)*P1)*(1-(P(m-1)+(stepsize/2)*P1)/L);
P3= k*(P(m-1)+(stepsize)*P2)*(1-(P(m-1)+(stepsize)*P2)/L);
P(m)= P(m-1) + (stepsize/6)*(P0+ 2*P1+ 2*P2+ P3);   % RK4 solution
        Peuler(m)= P(m-1) + stepsize* P0;      % also store away Euler’s method value
    end
    t(m)= j;  % keep track of t for plotting
    m= m+ 1;  % increment counter
end

% visualize
plot(t,Peuler,'g+'); grid on;   hold on
plot(t,P,'o--');
xlabel('t'); ylabel('P(t)');

% can also solve via Matlab’s builtin ode45, but need to use an external
% function to define the ODE
[tM,PM] = ode45(@(t,P) logistic(t,P,k,L),tI, Pinit);
plot(tM,PM,'kx','LineWidth',2);

% also plot analytic solution
A= (L-Pinit)/Pinit;
sol= L./(1+A*exp(-k*t));
plot(t,sol, 'r-')
legend('Euler','RK4','ode45 solution','Exact solution','Location','SouthEast')
title('Solution to logistic equation using various numerical methods')
Solution to logistic equation using various numerical methods

- Euler
- RK4
- ode45 solution
- Exact solution

What if we decreased the step-size?
What is the spacing for ode45 different?

Adaptive step-size at work here!

Warning: Beware the black box!

Golden rule 1 - The computer only does what you tell it to do

Caveat: Tricky when you are using code someone else wrote!
ode45

tI = [0 10];  % time boundaries
stepsize = 0.05;  % for RK4

➤ Syntax matters! Note subtle difference between following two lines of code:

[tM,PM] = ode45(@(t,P) logistic(t,P,k,L), tI, Pinit);

[tM,PM] = ode45(@(t,P) logistic(t,P,k,L), [0:stepsize:10], Pinit);

→ Latter doesn’t force fixed step-size, just interpolates the solution(!!)

➤ ode45 also allows a lot of flexibility to specify quantities such as step-size or error tolerance

% tell it to actually use the specified step-size
options = odeset ('MaxStep',stepsize);
[tM,PM] = ode45(@(t,P) logistic(t,P,k,L), tI, Pinit, options);

% allow step-size to vary based upon specified error tolerance
options = odeset ('RelTol',0.1);
[tM,PM] = ode45(@(t,P) logistic(t,P,k,L), tI, Pinit, options);

→ Actually fixes the step-size
So far, we have limited ourselves to a single first order ODE. But what about ‘systems’ of equations?

Lorenz equations
\[
\begin{align*}
\frac{dx}{dt} &= \sigma (y - x) \\
\frac{dy}{dt} &= rx - y - xz \\
\frac{dz}{dt} &= xy - bz
\end{align*}
\]

SIR model
\[
\begin{align*}
\frac{dS}{dt} &= -\beta IS \\
\frac{dI}{dt} &= \beta IS - \gamma I \\
\frac{dR}{dt} &= \gamma I
\end{align*}
\]

Predator-Prey
(Lotka-Volterra equations)
\[
\begin{align*}
\frac{dx}{dt} &= x(\alpha - \beta y) \\
\frac{dy}{dt} &= -y(\gamma - \delta x)
\end{align*}
\]

- What does each term physically represent?
- Are these equations linear? Is there an exact solution?
- What is the ‘atto-fox’ problem?
Systems of ODEs

- Solve in the exact same way as before, we just have one (or more) additional equation(s) to solve for each time step

```matlab
% User input (Note: All parameters are stored in a structure)
P.y0(1) = 3.0;   % initial prey population
P.y0(2) = 3.0;   % initial predator population
P.a= 1;         % prey pop. growth rate const.
P.b= 0.5;       % predation upon prey rate const.
P.c= 5;         % predator pop. growth rate const. (negative means loss)
P.d= 0.5;       % predator pop. growth rate const. due to prey consumption

% Integration limits
P.t0 = 0.0;     % Start value
P.tf = 10.0;    % Finish value
P.dt = 0.001;   % time step

% use built-in ode45 to solve
[t y] = ode45('LVfunction', [P.t0:P.dt:P.tf],P.y0,[],P);

figure(1); clf;
plot(t,y(:,1)); hold on; grid on;
plot(t,y(:,2),'r');
xlabel('t'); ylabel('Population size'); legend('Prey','Predator')

figure(2); clf;
plot(y(:,1), y(:,2)); hold on; grid on;
xlabel('Prey'); ylabel('Predator')
```

```matlab
function [out1] = LVfunction(t,y,flag,P)
% y(1) ... prey
% y(2) ... predator
out1(1)= y(1)*(P.a-P.b*y(2));
out1(2)= -y(2)*(P.c-P.d*y(1));
out1= out1';
```