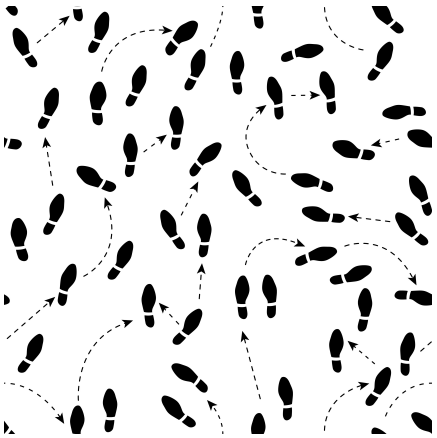
Ordinary Differential Equations: The Euler method

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/ode_euler/ode_euler.pdf



A discrete differential equation takes many little steps.

ODE's by the Euler method

Given an ODE: $y' = f(t,y), y(t_0) = y_0$, compute a sequence of sample values (t_i, y_i) that approximate the solution.

1 Euler's method

An initial value problem (IVP) is a common example of an ordinary differential equation. We are interested in a quantity that varies with time, which we denote by $y(t)$. We are told that, at time t_0, the function has the value y_0. Moreover, we are told that the function changes over time with a derivative $f(t,y)$. In some cases, it is possible to determine an exact formula for $y(t)$, but in general this is not possible. Instead, we will seek to estimate the value of y at a sequence of times $t_0, t_1, ..., T = t_{final}$.

We already know y_0. Euler's method is based on the idea that, since we know y_0, a good approximation for y_1 is simply

$$y(t_1) \approx y_1 = y_0 + dt\, f(t_0, y_0)$$

where $dt = t_1 - t_0$, the stepsize in time. If we have estimated y_1, we can then make an estimate for y_2, and proceed in this way for as many steps as we like. It's also possible that we will change the value of dt as we move along, although for now we will assume dt is a fixed quantity. A general step of the method can be written

$$y_{i+1} = y_i + dt\, f(t_i, y_i)$$

Denote $e_i \equiv y(t_i) - y_i$. The error in computing y_i has two parts: there is already an error in our computation of y_{i-1}. In fact, that's e_{i-1}. On top of that is the new error we make in predicting y_i using the update $dt\, f'(t_{i-1}, y_{i-1})$. It was shown in class that this second part of the error behaves like dt^2.

Now by the time we reach $T = t_{final}$, the error in y_T is affected by all the errors that came before it. It turns out that, if we hold T fixed, and seek more accurate solutions by reducing dt, then the error at y_T will behave like dt.

Thus, roughly speaking, the error behavior is quadratic in dt on a single step, and linear in dt when we consider the solution at some fixed time such as T.

In this lab, we will look at how to:

- implement Euler's method;
- set up and solve a differential equation;
- compare the computed and exact solutions;
- plot the computed and exact solutions;
- access MATLAB's ODE solvers;
- computed solution improves as we reduce dt;
- apply Euler's method to a higher-order differential equation.

2 Example: Approximate solution of $y' = 2t - 2$

Let us consider the IVP:

$$y'(t) = 2t - 2$$
$$y(0) = 10.0$$

and suppose we want to estimate $y(1.0)$. If we use a fixed stepsize $dt = 1/4$, then we have

```
y_0                                                = 10.000
y_1 = y_0 + dt * y'(t_0) = 10.000 + 0.25 * -2.0 =  9.500
y_2 = y_1 + dt * y'(t_1) =  9.500 + 0.25 * -1.5 =  9.125
y_3 = y_2 + dt * y'(t_2) =  9.125 + 0.25 * -1.0 =  8.875
y_4 = y_3 + dt * y'(t_3) =  8.875 + 0.25 * -0.5 =  8.750
```

In other words, to compute the sequence of values of y, we repeatedly loop over the following statement:

```
y(i+1) = y(i) + dt * f(t(i),y(i))
```

Thus, a simple Euler computation might be:

```
n = 4;
a = 0.0;
b = 1.0;
dt = 0.25;
t = linspace ( a, b, n + 1 );
y = zeros ( n + 1, 1 );
y(1) = 10.0;
for i = 1 : n
  y(i+1) = y(i) + dt * ( 2.0 * t(i) - 2.0 );
end
```

A better Euler computation would separate the data for our problem, the Euler computation itself, and the derivative into separate bits of code:

```
n = 4;
a = 0.0;
b = 1.0;
y0 = 10.0;
[ t, y ] = euler ( @(t,y)quadratic_prime(t,y), n, a, b, y0 );
plot ( t, y );
```

Listing 1: Specifying the data for the quadratic problem.

```
function [ t, y ] = euler ( yprime, n, a, b, y0 )

  t = linspace ( a, b, n + 1 );
  t = t';
  dt = ( b - a ) / n;

  m = length ( y0 );
  y = zeros ( n + 1, m );
  y(1,:) = y0(:);
  for i = 1 : n
    y(i+1.:) = y(i,:) + dt * yprime ( t(i), y(i,:) );
  end
  return
end
```

Listing 2: A general Euler code.

```
function value = quadratic_prime ( t, y )
  value = 2.0 * t - 2.0;
  return
end
```

Listing 3: The derivative code for the quadratic problem.

3 MATLAB conventions for ODE's

Later, we will be interested in using some of MATLAB's built-in ODE solvers. In order to do so, it will be necessary to follow a few conventions when writing the codes that define our ODE's. Primarily, we need to create a function that evaluates the derivative, using a template similar to what we did above for the quadratic problem:

```
function value = function_name ( t, y )
  value = function formula
  return
end
```

We will also be dealing with situations in which the solution y is a vector, in which case the derivative y' is also a vector. To follow MATLAB conventions, the derivative function must return a **row vector** result. This can be guaranteed by separating the individual components by commas in the vector notation:

```
function value = function_name ( t, y )
  u = y(1);
  v = y(2);
  w = y(3);
  dudt = u.^2 + u*v;
  dvdt = 2.0 + t;
  dwdt = sin ( w * t );
  value = [ dudt, dvdt, dwdt ];
  return
end
```

4 Exercise: A problem with quadratic solution

A formula for the exact solution of our quadratic problem, for any initial condition $y0$, is

$$y(t, y0) = t^2 - 2t + y0$$

We can use the `quadratic_euler(n,a,b,y0)` function, with various values of `n`, and fixed arguments `a=0.0`, `b=2.0`, `y0=10.0` to estimate the solution at points between 0 and 2, using a stepsize $dt = \frac{2}{n}$. We claimed that the error in a single step is proportional to dt^2, and the error at a fixed location is proportional to dt (smaller steps, but more of them.)

Define the first-step error as:

```
1 e(2) = y(2)-quadratic_exact(t(2),y0)
```

and verify the first claim:

```
  n   t(2)  y(2)  e(2)
  2
  4
  8
 16
 32
 64
128
```

Define the last-step error as:

```
1 e(n+1) = y(n+1)-quadratic_exact(t(n+1),y0)
```

and verify the second claim:

```
  n   t(n+1)    y(n+1)  e(n+1)
  2
  4
  8
 16
 32
 64
128
```

5 Some test problems

- **Quadratic:**

$$y'(t) = 2t - 2$$
$$y(0.0) = 10.0$$
$$y_{exact}(t) = t^2 - 2t + 10$$

- **Exponential:**

$$y'(t) = y$$
$$y(0.0) = 1.0$$
$$y_{exact}(t) = e^t$$

- **Negative exponential: ($\lambda = 1$ easy, 100 very hard)**

$$y'(t) = -\lambda y$$
$$y(0.0) = 1.0$$
$$y_{exact}(t) = e^{-\lambda t}$$

- **Predator/prey (system of two equations)**

$$\begin{aligned} r'(t) &= 2r - 0.001 * r * f \\ f'(t) &= -10f + 0.002 * r * f \\ r(0.0) &= 5000 \\ f(0.0) &= 100 \end{aligned}$$

- **Linear pendulum (second order system):**

$$\begin{aligned} y"(t) &= -y(t) \\ y(0.0) &= u \\ y'(0.0) &= v \\ y_{exact}(t) &= u\cos(t) + v\sin(t) \end{aligned}$$

- **Nonlinear pendulum with air resistance (second order system)**

$$\begin{aligned} y"(t) &= -\alpha y'(t) - \sin(y(t)) \\ y(0.0) &= u \\ y'(0.0) &= v \end{aligned}$$

6 Handling a system of first order ODE's

The Euler method can easily apply to a system of several first order ODE's. We are given initial values for several variables, and the rate of change for each. We use the Euler method to advance each variable to the next time, and continue in this way. The main new feature is that our unknown y and our derivative y' are now vectors.

Thus, the predator prey problem can be restated in terms of the vector $y = [r, f]$:

$$y'(t) = \begin{bmatrix} 2y_1 - 0.001y_1y_2 \\ -10y_2 + 0.002y_1y_2 \end{bmatrix} \quad y(0.0) = \begin{bmatrix} 5000 \\ 100 \end{bmatrix}$$

This problem has a nice solution which we can approximate well if we take a small enough time step. Note that the same `euler()` code can be used for a system as for a single equation.

7 Handling a second order ODE

A linear second order ODE might have the form:

$$\begin{aligned} w'' + \alpha w' + \beta w &= g(t, w, w') \\ w(t_0) &= \gamma \\ w'(t_0) &= \delta; \end{aligned}$$

We make the substitutions:

$$\begin{aligned} y_1 &\leftarrow w \\ y_2 &\leftarrow w' \end{aligned}$$

and noting that $y_1' = y_2$, we have the equivalent system of two equations:

$$y' = \begin{bmatrix} y_2 \\ -\alpha y_2 - \beta y_1 + g(t, y) \end{bmatrix} \quad y(t_0) = \begin{bmatrix} \gamma \\ \delta \end{bmatrix}$$

and we already know how to solve this using Euler's method.

8 Example: Nonlinear pendulum

For the second order system represented by the nonlinear pendulum with air resistance, assuming that $\alpha = 0.05$, we could set up the Euler computation as follows:

```
function pendulum_nonlinear_euler ( )

  n = 100;
  a = 0.0;
  b = 6.0 * pi;
  u = 1.0;
  v = 0.0;
  y0 = [ u, v ];

  [ t, y ] = euler ( @(t,y)pendulum_nonlinear_prime(t,y), n, a, b, y0 );

  return
end
```

with the derivative specified by:

```
function value = pendulum_nonlinear_prime ( t, y )

  alpha = 0.05;
  u = y(1);
  v = y(2);

  dudt = v;
  dvdt = - alpha * v - sin ( u );

  value = [ dudt, dvdt ];

  return
end
```

9 The Backward Euler method

The Euler method discretizes the ODE by replacing the derivative by a forward difference estimate. The backward Euler method is derived by using a backward difference. The i-th step of the backward Euler method can be written:

$$y_{i+1} = y_i + dt\, f(t_{i+1}, y_{i+1})$$

This is actually an implicit equation for y_{i+1}. Unless $f(t, y)$ is very simple, every step of the backward Euler method involves approximately solving this equation.

For our negative exponential test problem, however, f is in fact very simple. So we can rewrite the backward Euler method in this case as

$$\begin{aligned} y_{i+1} &= y_i + dt\,(-\lambda y_{i+1}) \\ y_{i+1} + dt\,\lambda\, y_{i+1} &= y_i \\ y_{i+1} &= \frac{y_i}{1.0 + dt\lambda} \end{aligned}$$

Assuming $0 < \lambda$, our solution will decay monotonically, and always remain between 0 and 1, unlike the behavior of the forward Euler solution.

If we don't have a simple way of simplifying the backward Euler step, we may have to use fixed point iteration or some kind of nonlinear equation solver to advance our solution on every step.

10 Computing Assignment #10

Consider the predator test problem, with the given initial conditions. We wish to solve this equation over the time interval [0,5]. Find a value n for the number of time steps so that the solution doesn't blow up, and in fact behaves almost periodically.

You might start from copies of *pendulum_euler.m* and *pendulum_prime.m*, and make the necessary changes.

Since the solution y is a vector, the MATLAB command `plot(t,y)` will make a standard plot that shows both components together. Call the plot file that is created `hw10.png`.

Turn in: your program *hw10.m* and your plot *hw10.png* by Friday, November 1.