Integrals compute area

Calculus in Simulation

• We’ll commonly have a derivative of some function (or instantaneous values of the derivative) and need to construct values for the function

• Example: We know velocity, need to compute position

• Example: We know acceleration, need to compute velocity and position
Time Slicing

- Simulation is updated at regular interval in time
  - $dt = \text{time step/slice}$

- To approximate integration:
  - compute derivative at time steps
  - make assumption about derivative value at other times
  - accumulate area to approximate integration
**Time slice approximation**

- $\Delta t = 1$, assume current value held for previous time slice

![Graph showing time slice approximation](image)

- **Blue**: correct derivative function
- **Red**: approximated derivative function

- under-estimate of positive area
- over-estimate of positive area
- under-estimate of negative area
- over-estimate of negative area
Time slice approximation

\[ \Delta t = 1 \]

**blue** → actual velocity

**red** → approx velocity

**green** → actual position

**magenta** → computed position
Smaller $\Delta t \rightarrow$ better approximation

euler_pos_vel.py
Smaller $\Delta t \rightarrow$ better approximation

Definition of a derivative:

$$f'(a) = \lim_{\Delta t \to 0} \frac{f(a + \Delta t) - f(a)}{\Delta t}$$

As $\Delta t$ gets smaller, we get closer to the true value of the derivative.
Time slice approximation

Definition of derivative:

\[ f'(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \]

Forward Euler Method:

\[ f(t + \Delta t) = f(t) + \Delta t \cdot f'(t) \]

Backward Euler Method:

\[ f(t + \Delta t) = f(t) + \Delta t \cdot f'(t + \Delta t) \]

(we’re using backward Euler)
Euler’s Method

- Taylor Series:

\[ f(t + \Delta t) = f(t) + \Delta t \cdot f'(t) + \Delta t^2 \cdot f''(t) + \Delta t^3 \cdot f'''(t) + \ldots \]

as \( \Delta t \to 0 \), the higher order terms quickly approach 0, giving Euler’s approximation:

\[ f(t + \Delta t) = f(t) + \Delta t \cdot f'(t) \]

- The higher order terms are the error in the approximation
Instantaneous Derivative Values

- In interactive simulations, we often do not have a function for the derivative, rather we have instantaneous values for the derivative.
- Example: driving simulation
  - acceleration value is set by driver’s joystick
- Euler’s method can still be applied (twice) to compute velocity and position
  - assumes that current value of acceleration was constant through previous time step
Example: Vehicle Dynamics

- Acceleration is set by user
- Function updates velocity, then position

```cpp
void Vehicle::update(double elapsed_time) {
    double dt = elapsed_time/1000.0;
    velocity += acceleration*dt;
    double vx = velocity*sin(heading);
    double vy = velocity*cos(heading);
    double x = position.getX() + vx*dt;
    double y = position.getY() + vy*dt;
    position.moveTo(Point(x,y));
}
```
Numeric Integration Methods

(explicit or forward) Euler Integration
2\textsuperscript{nd} order Runge Kutta Integration (Midpoint Method)
4\textsuperscript{th} order Runge Kutta Integration

Three slides borrowed from:
www.cse.ohio-state.edu/~parent/BookWebMaterial/Chapter07/Powerpoint/Ch7_1_Integration.ppt
Runge Kutta Integration: 2\textsuperscript{nd} order
Aka Midpoint Method

For unknown function, $f(t)$; known $f'(t)$

\[ f(t_{i+\frac{1}{2}}) = f(t_i) + \frac{1}{2} f'(t_i) \cdot \Delta t \]

\[ f(t_{i+1}) = f(t_i) + f'(t_{i+\frac{1}{2}}) \cdot \Delta t \]
Runge Kutta Integration: 4\textsuperscript{th} order

For unknown function, $f(t)$; known $f'(t)$

$$f(t_{i+1}) = f(t_i) + h \left( \frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4 \right)$$
Numerical Integration Techniques

- Euler:
  - \( x(t+\Delta t) = x(t) + f'(t, x)\Delta t \)

- Runga-Kutta 2:
  - \( k1 = f'(t, x)\Delta t \)
  - \( k2 = f'(t+\frac{\Delta t}{2}, x+k1/2)\Delta t \)
  - \( x(t+\Delta t) = x + k2 \)

- Runga-Kutta 4:
  - \( k1 = f'(t, x)\Delta t \)
  - \( k2 = f'(t+\frac{\Delta t}{2}, x+k1/2)\Delta t \)
  - \( k3 = f'(t+\frac{\Delta t}{2}, x+k2/2)\Delta t \)
  - \( k4 = f'(t+\Delta t, x+k3)\Delta t \)
  - \( x(t+\Delta t) = x + k1/6 + k2/3 + k3/3 + k4/6 \)

Better methods use more points on \( f' \) to better approximate results.
A hand-coded Euler integration of $x' = x + 3$

```python
from pylab import *

# initial value for x
x = 1.0

# initial value for time
# initial value for time
# time step for integration
# initial value for time
# time step for integration

dt = 0.1

dx = x + 3
x = x + dx*dt

# store for plotting
# initial value for time
# initial value for time
# time step for integration
# store for plotting
# store for plotting
# store for plotting

x_data = []
t_data = []

while t < 5.0:
    # one step of integration
    dx = x + 3
    x = x + dx*dt
    # store for plotting
    x_data.append(x)
t_data.append(t)
    # increment time
    t = t + dt

xlabel("t = time")
ylabel("df/dt")
plot(t_data, x_data, "b")
show()
```

hand-coded_euler.py
The attached Python class implements Euler, RK2 and RK4 techniques. It operates on a set of first order differential equations. Class constructor receives diffeqs and ∆t. Calls to integrate function move integration forward by ∆t time.
Example: one diffeq

- Solve $x' = x + 3$

```python
# function to compute f'(t,x)
def diff_eqs(t, x):
    # create return vector
dx = [0.0]*num_eqs
    # solve diff eqs
dx[0] = x[0] + 3
    # return answer
return dx
```

first_example.py
Example: one diffeq

- Solve $x' = x + 3$

```python
# setup the simulation

# initial values for x
x = [1.0]

# initial value for time
t = 1.0

# time step for integration
dt = 0.1

# set up the integrator
integrator = Integrator(num_eqs, dt, diff_eqs)
```

first_example.py
Example: one diffeq

• Solve $x' = x + 3$

```python
# lists to store results
x_data = []
t_data = []

# run the simulation
while t < 5.0:
    # one step of integration for current time
    x = integrator.integrateEuler(x, t)
    # store current values for plotting
    x_data.append(x)
t_data.append(t)
    # increment time
    t = t + dt
```

first_example.py
Example: one diffeq

- Result:

```
first_example.py
```
Example: two diffeqs

- $x_1' = k_1 x_1 - k_2 x_1 x_2$  # prey population change
- $x_2' = k_3 x_1 x_2 - k_4 x_2$  # predator population change

```python
num_eqs = 2
k1 = 0.1    # prey birth
k2 = 0.001  # predator/prey interaction
k3 = 0.001  # predator/prey interaction
k4 = 0.3    # predator death

# define two differential equations
def diff_eqs(t, x):
    dx = [0.0]*num_eqs
    dx[0] = k1*x[0] - k2*x[0]*x[1]
    dx[1] = k3*x[0]*x[1] - k4*x[1]
    return dx
```

`predator_prey_euler.py`
Example: two diffeqs

- \( x_1' = k_1 x_1 - k_2 x_1 x_2 \)  # prey population change
- \( x_2' = k_3 x_1 x_2 - k_4 x_2 \)  # predator population change

```python
# setup simulation
x = [100.0, 100.0] # initial populations
dt = 0.1
t = 0.0
integrator = Integrator(num_eqs, dt, diff_eqs)

# run simulation
while t < 1000:
    x = integrator.integrateEuler(x, t)
    p1_data.append(x[0])
    p2_data.append(x[1])
    t_data.append(t)
    t = t + dt
```

one call solves both equations

current values of each equation are in array

predator_prey_euler.py
Example: two diff eqs

- \( x_1' = k_1 x_1 - k_2 x_1 x_2 \)  \# prey population change
- \( x_2' = k_3 x_1 x_2 - k_4 x_2 \)  \# predator population change

blue = population 1
red = population 2
Euler integration

predator_prey_euler.py
Example: two diffeqs

- $x_1' = k_1 x_1 - k_2 x_1 x_2$  # prey population change
- $x_2' = k_3 x_1 x_2 - k_4 x_2$  # predator population change

blue = population 1
red = population 2

Runga-Kutta 2 integration

```
predator_prey_rk2.py
x = integrator.integrateRK2(x, t)
```
Example: two diffeqs

- \( x_1' = k_1 x_1 - k_2 x_1 x_2 \) # prey population change
- \( x_2' = k_3 x_1 x_2 - k_4 x_2 \) # predator population change

Common problem with Euler: accumulating errors
Aside: Chaotic Attractors

- Lorenz system:
  - $x_1' = a(x_2 - x_1)$
  - $x_2' = (1 + d - x_3)x_1 - x_2$
  - $x_3' = x_1x_2 - bx_3$
  - blue $\rightarrow$ Euler
  - red $\rightarrow$ RK4

lorenz.py
Aside: Chaotic Attractors

- Silnikov system:
  - $x_3' = -a x_3 - x_2 + b x_1 (1 - c x_1 - d x_1^2)$
  - $x_2' = x_3$
  - $x_1' = x_2$
  - blue $\rightarrow$ Euler
  - red $\rightarrow$ RK4

silnikov.py
Higher-order Systems

- Our solution solves systems of first order equations.
- To deal with higher-order systems, we can make substitutions to convert to first-order equations.
Higher-order Systems

- Example: projectile
  \( y(0) = 0, \ y'(0) = 10, \ y''(t) = -9.8 \)

  \( x(0) = 0, \ x'(0) = 10, \ x''(t) = 0.0 \)
Higher-order Systems

- Example: projectile
  - $y(0) = 0, y'(0) = 10, y''(t) = -9.8$
  - $x(0) = 0, x'(0) = 10, x''(t) = 0.0$

- To get a system of 1st order equations, introduce new variables:
  - $A_0 = y$
  - $A_1 = y'$
  - $A_2 = x$
  - $A_3 = x'$

- Derivatives of new variables:
  - $A_0' = y' = A_1$
  - $A_1' = y'' = -9.8$
  - $A_2' = x' = A_3$
  - $A_3' = x'' = 0.0$
Higher-order Systems

• projectile:

#======================== # system constants
# g = -9.8 # meters/second
m = 2    # kilograms
y0 = 100 # meters

#======================== # number of equations
num_eqs = 4

# function to compute f'(t,x)
def diff_eqs(t, A):
    # create return vector
dA = [0.0]*num_eqs
    # solve diff eqs
dA[1] = g
dA[3] = 0
    # return answer
return dA
Higher-order Systems

- projectile result:
Harmonic Oscillation

- A spring and block system (no friction) will exhibit harmonic oscillation

\[ mx'' = -kx \]
Harmonic Oscillation

- $mx'' = -kx$
  $x'' = (-k/m)x$

- New variables:
  - $A_0 = x$
  - $A_1 = x'$
  - $A_0' = A_1$
  - $A_1' = (-k/m)x = (-k/m)A_0$

- Initial values:
  - $A_0(0) = 5$
  - $A_1(0) = 0$

```
spring_harmonic.py
```
Harmonic Oscillation

- Pendulum
  \[ md\theta'' = -mg\sin(\theta) - b\theta' - k\theta \]
  \[ \theta'' = (-g/d)\sin(\theta) + (-b/md)\theta' + (-k/md)\theta \]

- \( m \) = mass of bob
- \( g \) = gravity
- \( d \) = length of string
- \( b \) = damping (friction)
- \( k \) = spring (string)
Pendulum

- Pendulum
  \[md\ddot{\theta} = -mg \sin(\theta) - b\theta' - k\theta\]
  \[\ddot{\theta} = \left(-\frac{g}{d}\right) \sin(\theta) + \left(-\frac{b}{md}\right)\theta' + \left(-\frac{k}{md}\right)\theta\]

- new variables:
  \[A_0 = \theta\]
  \[A_1 = \theta'\]
  \[A_0' = \theta' = A_1\]
  \[A_1' = \ddot{\theta} = \left(-\frac{g}{d}\right) \sin(A_0) + \left(-\frac{b}{md}\right)A_1 + \left(-\frac{k}{md}\right)A_0\]
Pendulum

- Pendulum, initial values:
  \( A_0(0) = 1.5 \) radians
  \( A_1(0) = 0.0 \)
Pendulum

positions vs. time
blue $\rightarrow$ x position
green $\rightarrow$ y position

angle vs. time

`pendulum.py`