Computer Methods (MAE 3403)

Ordinary Differential Equations
(ODE)

Motivation

- Differential equations describe relationships between a function and its derivatives
- Widely used in modelling systems in every engineering and science field
  - Car’s motion, pendulum, spacecraft, air vehicle, HVAC
- Finding exact solutions to a differential equation is hard.
  - Numerical solutions are critical

Differential equations

- Describe the relationships of f(x) and its derivatives.
- Ordinary differential equations (ODE): single independent variable (x)
- An nth order ODE:

$$F\left(x, f(x), \frac{df(x)}{dx}, \frac{d^2 f(x)}{dx^2}, \frac{d^3 f(x)}{dx^3}, \ldots, \frac{d^{n-1} f(x)}{dx^{n-1}}\right) = \frac{d^n f(x)}{dx^n},$$

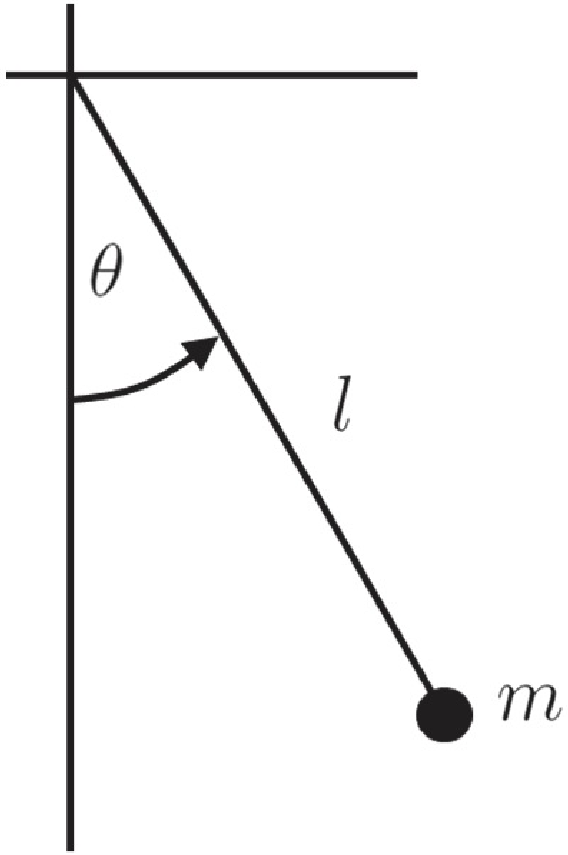
Examples

- The motion of the angle in the presence of gravity can be described as

$$ml\frac{d^2\theta(t)}{dt^2} = -mg\sin(\theta(t)).$$

- Second-order acceleration model

$$\frac{d^2p(t)}{dt^2} = a(t)$$



Two main problems

- Initial value problems (IVP)

- Boundary value problems (BVP)

Initial value problems

- For an nth order ODE, the **initial value** is the known value for the 0^{th} to (n-1)th derivatives at x = 0, i.e., $f(0), f^{(1)}(0), \ldots, f^{(n-1)}(0)$.

$$F\left(x, f(x), \frac{df(x)}{dx}, \frac{d^2 f(x)}{dx^2}, \frac{d^3 f(x)}{dx^3}, \ldots, \frac{d^{n-1} f(x)}{dx^{n-1}}\right) = \frac{d^n f(x)}{dx^n},$$

- IVP: finding a solution to the ODE given an initial value.
- Notation: $f'(t) = f^{(1)}(t) = \dot{f}(t) = \frac{df(t)}{dt}$

Rewrite the ODE to "first-order"

- Numerical methods designed for first-order DEs.

$$f^{(n)}(t) = F\left(t, f(t), f^{(1)}(t), f^{(2)}(t), f^{(3)}(t), \ldots, f^{(n-1)}(t)\right)$$

$$S(t) = \begin{bmatrix} f(t) \\ f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ \cdots \\ f^{(n-1)}(t) \end{bmatrix} \Rightarrow \frac{dS(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ F\left(t, f(t), f^{(1)}(t), \ldots, f^{(n-1)}(t)\right) \end{bmatrix}$$

$$= \begin{bmatrix} S_2(t) \\ S_3(t) \\ S_4(t) \\ S_5(t) \\ \cdots \\ F\left(t, S_1(t), S_2(t), \ldots, S_{n-1}(t)\right) \end{bmatrix}$$

- First-order ODE for S(t)!
- nth order ODE => n first-order coupled ODEs

Examples

$$ml\frac{d^2\theta(t)}{dt^2} = -mg\sin(\theta(t)).$$

$$S(t) = \begin{bmatrix} \Theta(t) \\ \dot{\Theta}(t) \end{bmatrix}$$

$$\frac{dS(t)}{dt} = \begin{bmatrix} S_2(t) \\ -\frac{g}{l}S_1(t) \end{bmatrix}$$

$$=$$

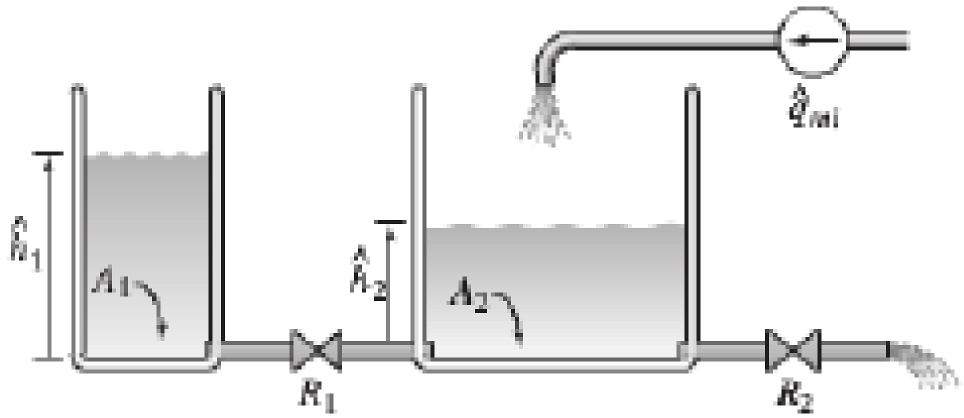
Simple model to describe population of rabbits r(t) and wolves w(t)

$$\frac{dr(t)}{dt} = 4r(t) - 2w(t)$$
$$\frac{dw(t)}{dt} = r(t) + w(t)$$

$$S(t) = \begin{bmatrix} r(t) \\ w(t) \end{bmatrix}$$

$$\frac{dS(t)}{dt} =?$$

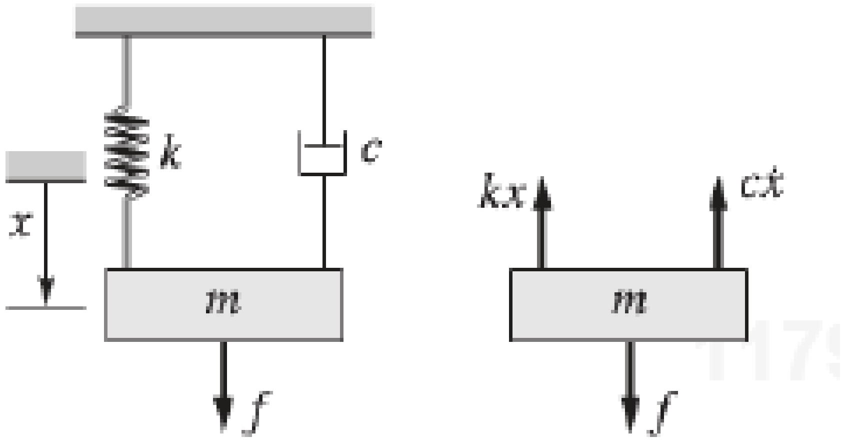
More examples: two coupled 1st-order ODE



$$A_1 \frac{dh_1}{dt} = -\frac{g}{R_1}(h_1 - h_2)$$

$$\rho A_2 \frac{dh_2}{dt} = q_{mi} + \frac{\rho g}{R_1}(h_1 - h_2) - \frac{\rho g}{R_2} h_2$$

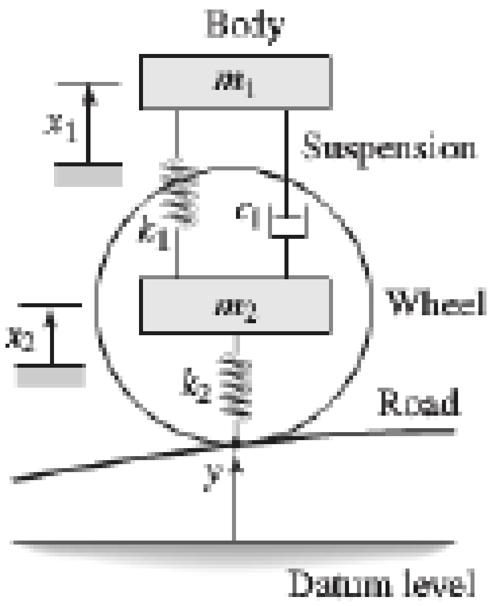
One 2nd-order ODE



$$m\ddot{x} + c\dot{x} + kx = f$$

Two 2nd-order ODE

$$m_1\ddot{x}_1 = c_1(\dot{x}_2 - \dot{x}_1) + k_1(x_2 - x_1)$$



$$m_2\ddot{x}_2 = -c_1(\dot{x}_2 - \dot{x}_1) - k_1(x_2 - x_1) + k_2(y - x_2)$$

Solving ODE: The Euler Method

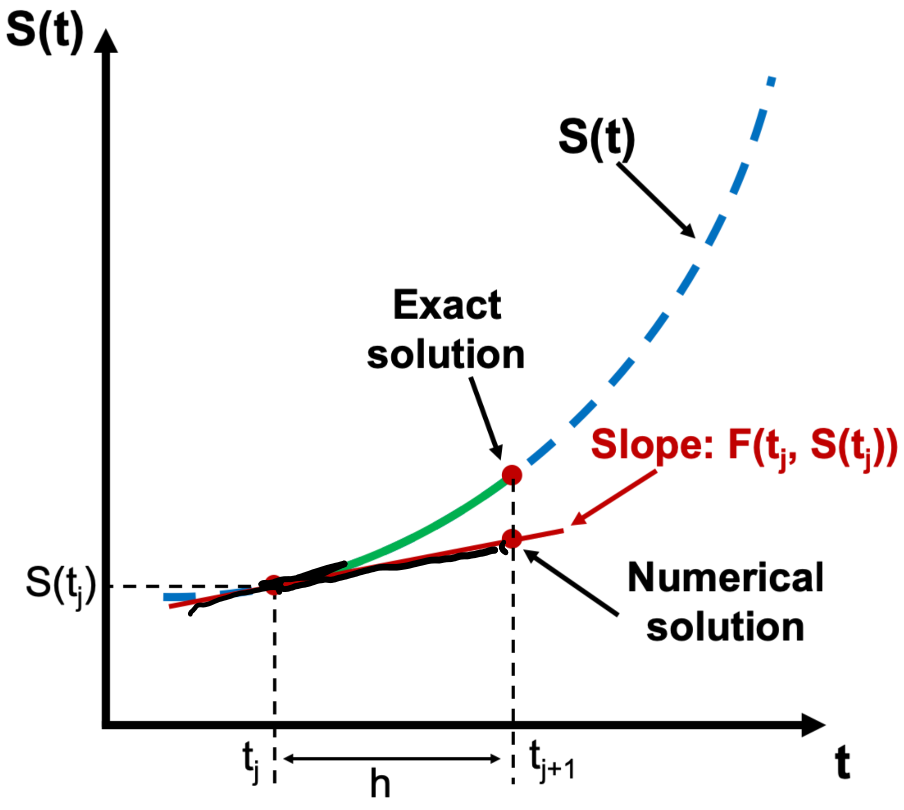
- Suppose we have an ODE system explicitly given

$$\dot{S}(t) = \mathbf{F}(t, S(t))$$

- Also given is the initial condition *$S(t_0)$*
- Define a numerical grid $[t_0, t_f]$ with spacing h. Let the ith grid point t_i = ih and t_f = Nh.
- Explicit Euler Method: starting with *$j=0$ and $S(t_0)$*

$$S(t_{j+1}) = S(t_j) + h\mathbf{F}(t_j, S(t_j))$$

What’s happening?



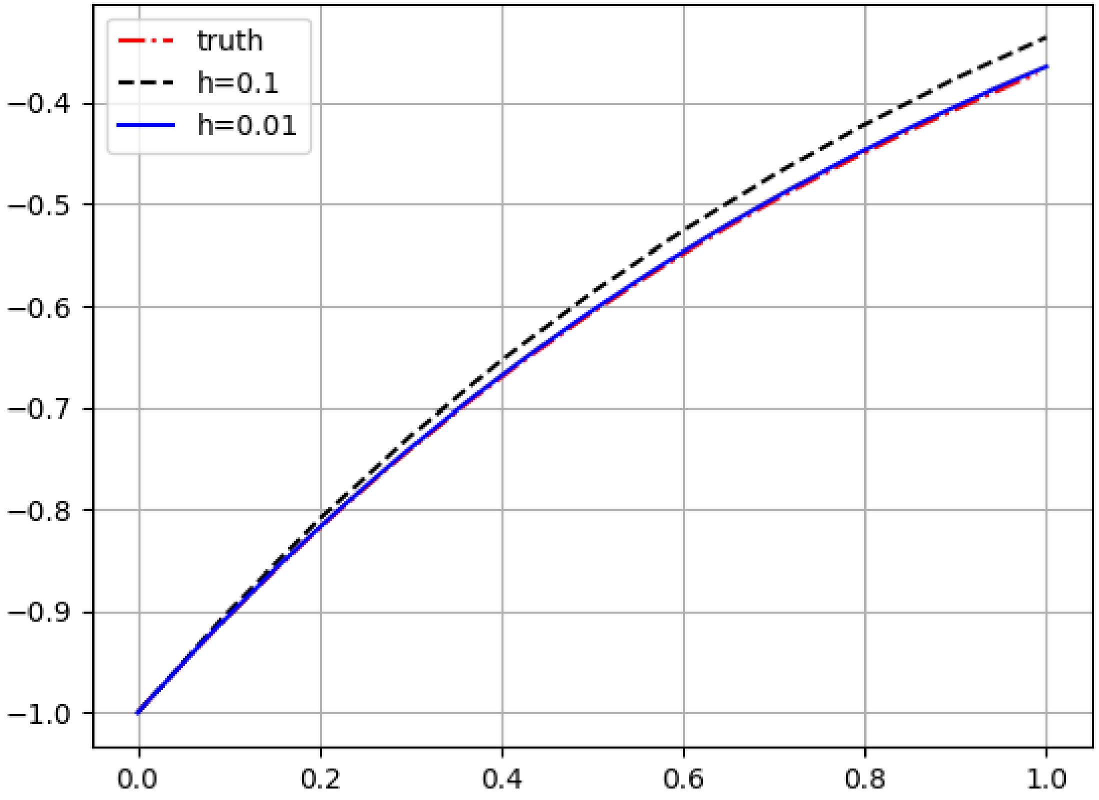
1. Store $S_0 = S(t_0)$ in an array, S.
2. Compute $S(t_1) \approx S_0 + hF(t_0, S_0)$.
3. Store $S_1 = S(t_1)$ in S.
4. Compute $S(t_2) \approx S_1 + hF(t_1, S_1)$.
5. Store $S_2 = S(t_2)$ in S.
6. ……
7. Compute $S(t_f) \approx S_{n-1} + hF(t_{n-1}, S_{n-1})$.
8. Store $S_n = S(t_f)$ in S.
9. S is an approximation of the solution to the IVP.

Example

$df(t)/dt = e^{-t}$ with $f(0)=-1$. Explicit solution: $f(t) = -e^{-t}$

Can you use the explicit Euler method to compute the solution of $f(t)$ from $t=0$ to $t = 1$ with $h = 0.1$? Compare the solution to the explicit solution in a plot. How about $h = 0.01$?

What conclusions can you draw from this example?




```python
import matplotlib.pyplot as plt
import numpy as np
# author: HB Oct-16-24

def euler_int(f, f0, a, b, h):
    F = [f0]
    i = 0
    while True:
        x = a + i * h
        if x > b - h:
            break
        i = i + 1
        F.append(F[-1] + h * f(x))
    return F
x = np.arange(0, 1, 0.01)
y = -np.exp(-x)

plt.plot(x, y, 'r-.', label='truth')
# euler integration
h = 0.1
f = lambda x: np.exp(-x)
F1 = euler_int(f, -1, 0, 1, h)
plt.plot(np.arange(0, 1+h, h), F1, 'k--', label=f'h={h}')
h = 0.01
F2 = euler_int(f, -1, 0, 1, h)
plt.plot(np.arange(0, 1+h, h), F2, 'b-', label=f'h={h}')
plt.legend()
plt.grid()
plt.show()
```

http://tpcg.io/_A5NKPC

Implicit Euler Method

- Explicit method: only requires information at t_j to compute the state at t_{j+1}

$$S(t_{j+1}) = S(t_j) + h\mathbf{F}(t_j, S(t_j))$$

- Implicit method:

$$S(t_{j+1}) = S(t_j) + h\mathbf{F}(t_{j+1}, S(t_{j+1}))$$

- Another relevant method: trapezoidal formula

$$S(t_{j+1}) = S(t_j) + \frac{h}{2}(\mathbf{F}(t_j, S(t_j)) + \mathbf{F}(t_{j+1}, S(t_{j+1})))$$

Example

$$\frac{dS(t)}{dt} = \left[\begin{array}{cc} 0 & 1 \\ -\frac{g}{l} & 0 \end{array} \right] S(t) \qquad \mathbf{F}(t_j, S(t_j)) = \left[\begin{array}{cc} 0 & 1 \\ -\frac{g}{l} & 0 \end{array} \right] S(t_j).$$

- Explicit formula

$$S(t_{j+1}) = S(t_j) + h \left[\begin{array}{cc} 0 & 1 \\ -\frac{g}{l} & 0 \end{array} \right] S(t_j) = \left[\begin{array}{cc} 1 & h \\ -\frac{gh}{l} & 1 \end{array} \right] S(t_j)$$

- Implicit formula

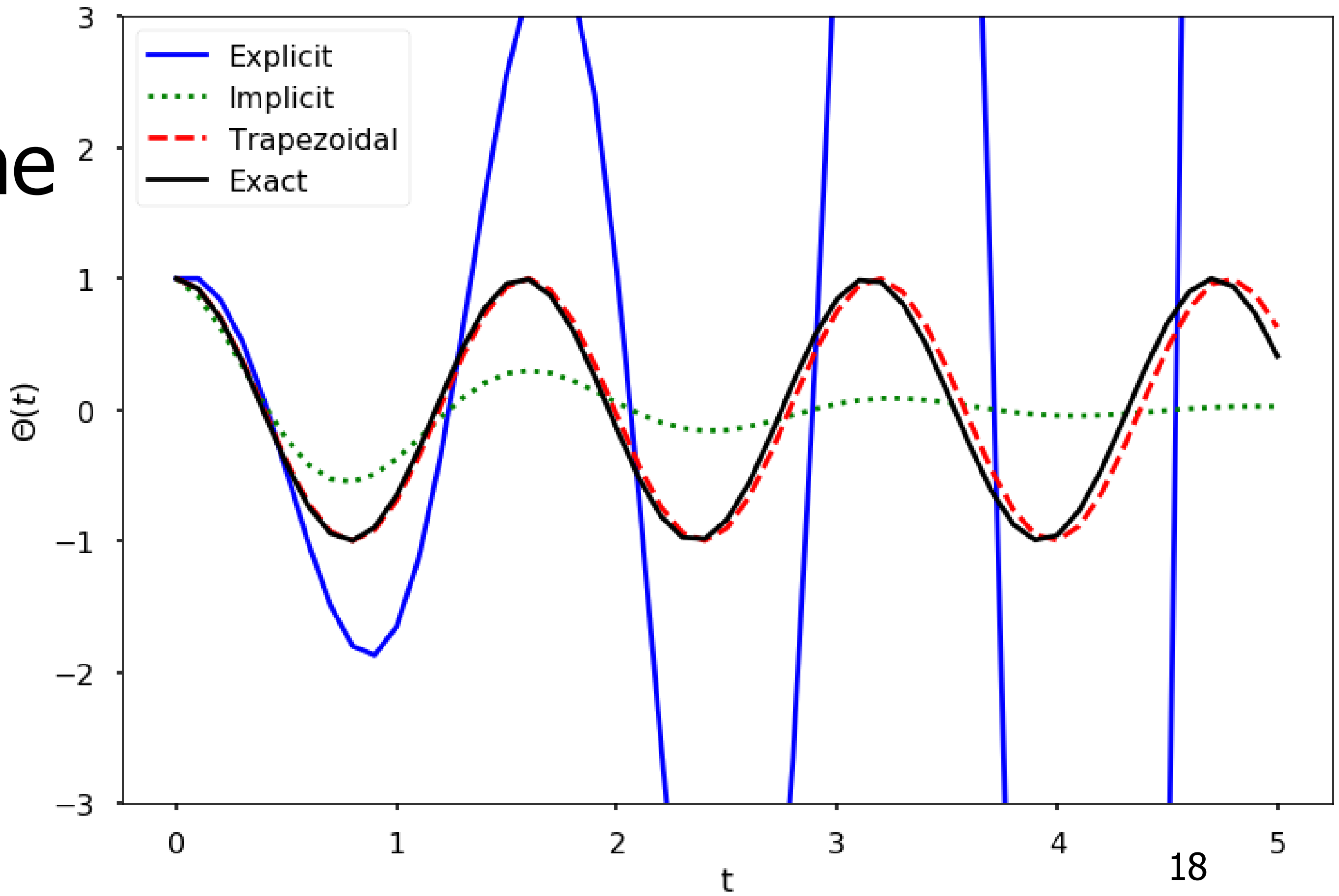
$$\left[\begin{array}{cc} 1 & -h \\ \frac{gh}{l} & 1 \end{array} \right] S(t_{j+1}) = S(t_j), \quad S(t_{j+1}) = \left[\begin{array}{cc} 1 & -h \\ \frac{gh}{l} & 1 \end{array} \right]^{-1} S(t_j)$$

- Trapezoidal formula

$$S(t_{j+1}) = \left[\begin{array}{cc} 1 & -\frac{h}{2} \\ \frac{gh}{2l} & 1 \end{array} \right]^{-1} \left[\begin{array}{cc} 1 & \frac{h}{2} \\ -\frac{gh}{2l} & 1 \end{array} \right] S(t_j).$$

Solving ODE

- Accuracy: ability to get close to the true solution
- Stability: ability to keep the error from growing as it integrates over time.
- We solve the pendulum equation using Euler explicit, implicit and trapezoidal formula.



Better schemes to solve ODEs

- Predictor-correct methods: improve the accuracy by querying the F multiple times at different locations (predictions) and using a weighted average of the results (correction) to update the state.
- Midpoint method:

$$\text{Predictor step: } S\left(t_j + \frac{h}{2}\right) = S(t_j) + \frac{h}{2}\mathbf{F}(t_j, S(t_j))$$

$$\text{Corrector step: } S(t_{j+1}) = S(t_j) + h\mathbf{F}\left(t_j + \frac{h}{2}, S\left(t_j + \frac{h}{2}\right)\right)$$

Runge Kutta Methods (RK methods)

- Better accuracy if we use higher-order of derivatives

$$S(t_{j+1}) = S(t_j + h) = S(t_j) + S'(t_j)h + \frac{1}{2!}S''(t_j)h^2 + \cdots + \frac{1}{n!}S^{(n)}(t_j)h^n$$

- RK methods avoid computing higher-order derivatives:

2nd order RK with $c_1 = c_2 = 0.5$, $p = q = 1$:

$$S(t+h) = S(t) + c_1\mathbf{F}(t, S(t))h + c_2\mathbf{F}[t + ph, S(t) + qh\mathbf{F}(t, S(t))]h$$

4th order RK method: $O(h^4)$

Python ODE solvers

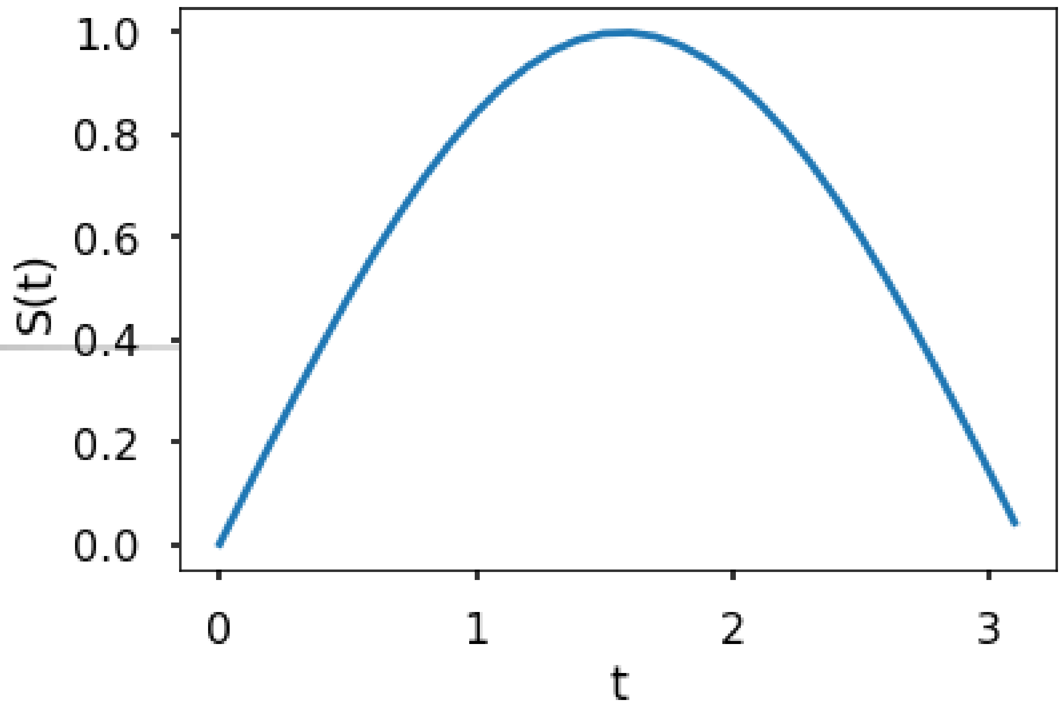
- scipy.integrate.solve_ivp or scipy.integrate.odeint

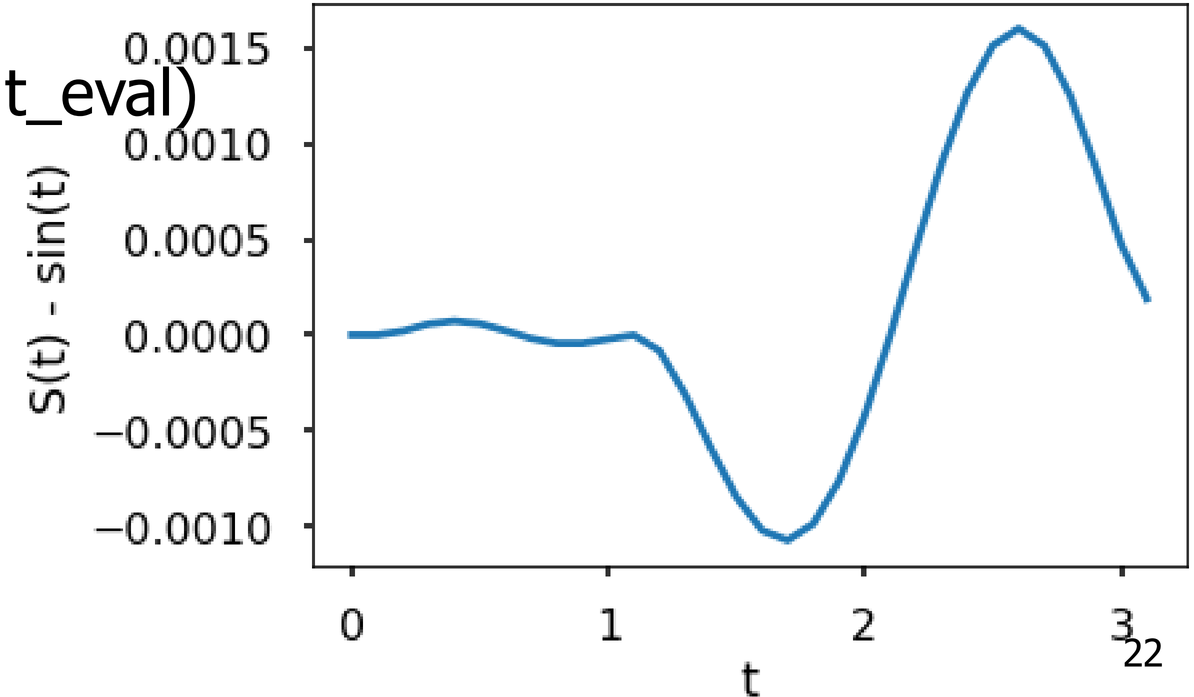
solve_ivp(fun, t_span, s0, method = 'RK45', t_eval=None)

- fun: takes the function **F**(t, S(t)), t_span: integration interval [t0, tf], s0: initial state, method: different integration methods, t_eval takes in the times to store the computed solution, must be sorted and lie in t_span.
- Also can set tolerances atol, rtol (default 1e-6, 1e-3)
- **odeint**: works similarly, check the documentation on its use

Example

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
F = lambda t, s: np.cos(t)
t_eval = np.arange(0, np.pi, 0.1)
sol = solve_ivp(F, [0, np.pi], [0], t_eval=t_eval)
plt.figure(figsize = (12, 4))
plt.subplot(121)
plt.plot(sol.t, sol.y[0])
plt.xlabel('t’)
plt.ylabel('S(t)’)
plt.subplot(122)
plt.plot(sol.t, sol.y[0] - np.sin(sol.t)) …
```

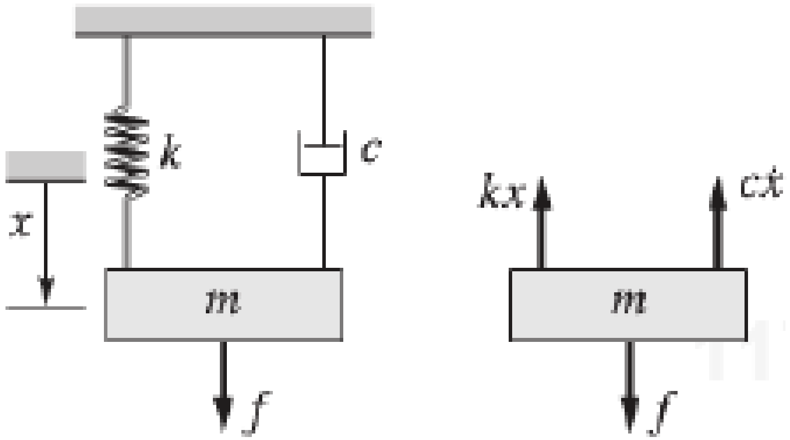




solve_ivp vs. odeint

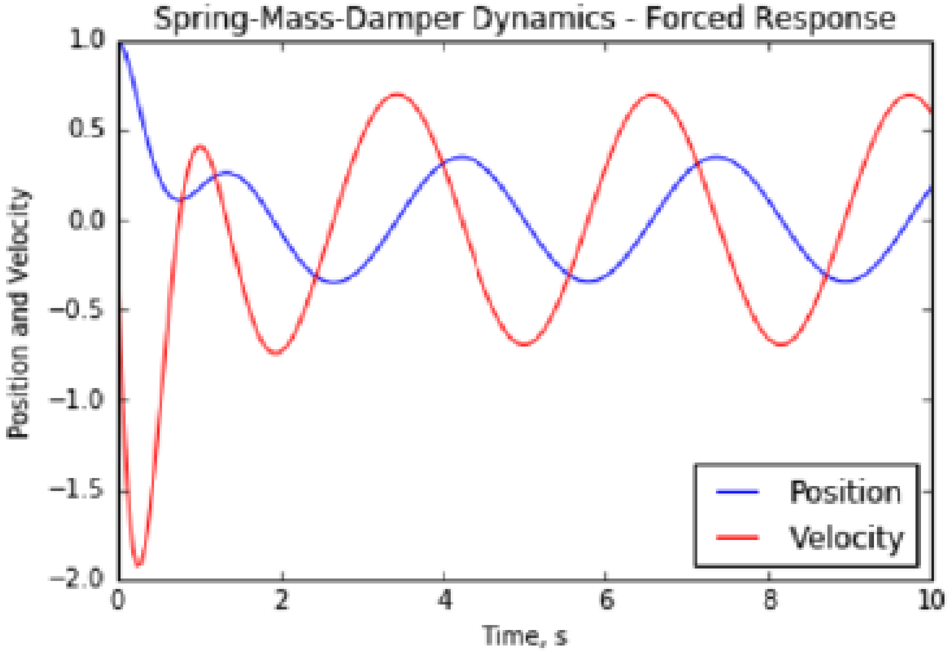
- odeint uses lsoda from Fortran package to solve ODEs
- solve_ivp is more general, containing multiple methods, including lsoda, but also others like BDF.
- solve_ivp is reported slower than odeint.
- Recent Python release suggests using solve_ivp

One 2nd-order ODE



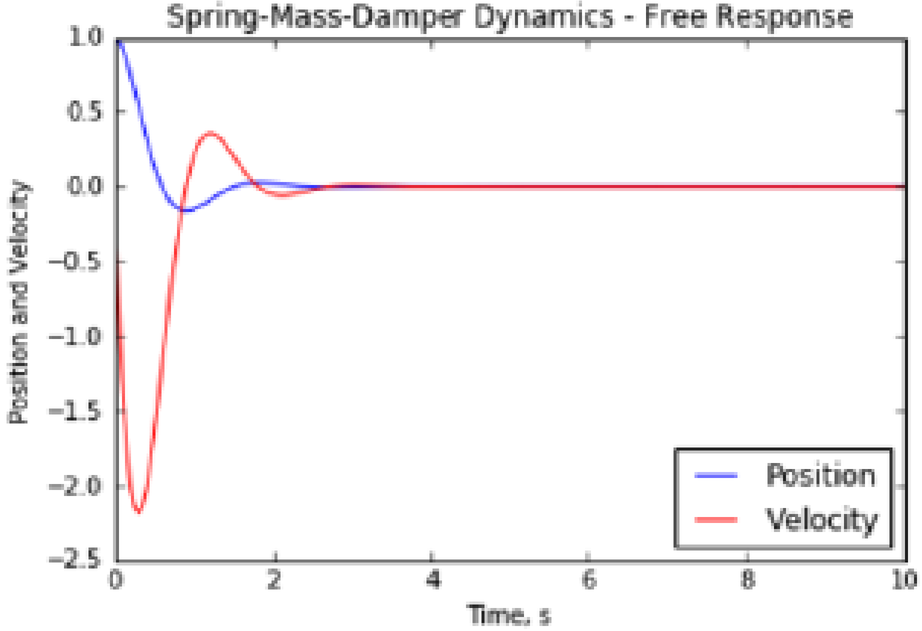
$$m\ddot{x} + c\dot{x} + kx = f$$

```
def ode_system(X, t, m,c,k,fmag ):
    #define any numerical parameters (constants)
    # these params were stored in a list, and must be passed in the correct order!

    #define the forcing function equation
    f=fmag*np.sin(2*t)

    x=X[0]; xdot=X[1]  # copy from the state array to nicer names

    #write the non-trivial equatin
    xddot= (1/m) * (f-c*xdot-k*x)

    return [xdot,xddot]
```




```python
t = np.linspace(0, 10, 200)    #time goes from 0 to 10 seconds
ic=[1,0]

#define the model parameters
m=1  # the mass
c=4  # damping (shock absorber)
k=16 # the spring
fmag = 5 # the magnitude of the forcing function

x = odeint(ode_system, ic, t,args=(m,c,k,fmag))

plt.plot(t, x[:,0], 'b-', label = 'Position')
plt.plot(t, x[:,1], 'r-', label = 'Velocity')
plt.legend(loc = 'lower right')
plt.xlabel('Time, s')
plt.ylabel('Position and Velocity')
plt.title('Spring-Mass-Damper Dynamics - Forced')
plt.show()
```

```python
def ode_system(X, t, m,c,k,fmag ):
    #define any numerical parameters (constants)
    # these params were stored in a list, and must be passed in the

    #define the forcing function equation
    f=fmag*np.sin(2*t)

    x=X[0]; xdot=X[1]  # copy from the state array to nicer names

    #write the non-trivial equatin
    xddot= (1/m) * (f-c*xdot-k*x)

    return [xdot,xddot]
```



Schedule

- Exam on Oct. 30
  - 2 Problems
  - Useful materials
- HW 5 due on Nov. 5
- The week of Nov. 4

- Final exam: Friday, Dec. 13, 8-9:50am, ATRC 102

Template for using solve_ivp

- Write down your ODE

$$f^{(n)}(t) = F\left(t, f(t), f^{(1)}(t), f^{(2)}(t), f^{(3)}(t), \ldots, f^{(n-1)}(t)\right)$$

$$\frac{dS(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ F\left(t, f(t), f^{(1)}(t), \ldots, f^{(n-1)}(t)\right) \end{bmatrix}$$

$$= \begin{bmatrix} S_2(t) \\ S_3(t) \\ S_4(t) \\ S_5(t) \\ \cdots \\ F\left(t, S_1(t), S_2(t), \ldots, S_{n-1}(t)\right) \end{bmatrix}$$

```python
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.integrate import solve_ivp
4
5 # define your ODE_system
6
7 def ode_system(t, S, param1, param2):
8     # S should be the n by 1 state vector given the differential equation
9
10     # if you need the passed parameters
11     p1 = param1
12     p2 = param2
13     # assignment of x, xdot, etc for easy reading and writing, assuming n = 2
14     x = S[0]
15     xdot = S[1]
16     # you can have more, xddot, xdddot,... depending on n
17     # now write the last equation in dotS
18     xddot = # F(t, x, xdot)
19     # return dotS, the derivative of S, another n by 1 vector
20     # dot S[0] = dot x = xdot (see line 13, 14)
21     # dot S[1] = dot xdot = xddot
22     return [xdot, xddot]
```

```python
25 # solving the ODE
26 # preparation
27 ic = [0,0] #initial condition for S[0], S[1],..., S[n-1],
28 t0 = 0 # initial sim time
29 tf = 3 # final sim time
30 t_ = np.linspace(t0, tf, 100) # evaluate the ODE solution on this grid
31 param1 = 1
32 param2 = 2
33 #ready to solve
34 sol = solve_ivp(ode_system, [t0, tf], ic, t_eval = t_, args =(param1, param2))
35
36 #decode the "sol" variable
37 t = sol.t # same as t_
38 S = sol.y # corresponding S[0], S[1], ... over time, row-wise
39
40 plt.plot(t, S[0,:], label='this is the first state (x) in S over time')
41 plt.plot(t, S[1,:], label='this is the second state (xdot) in S over time')
42 plt.legend()
43 plt.grid()
44 plt.show()
```

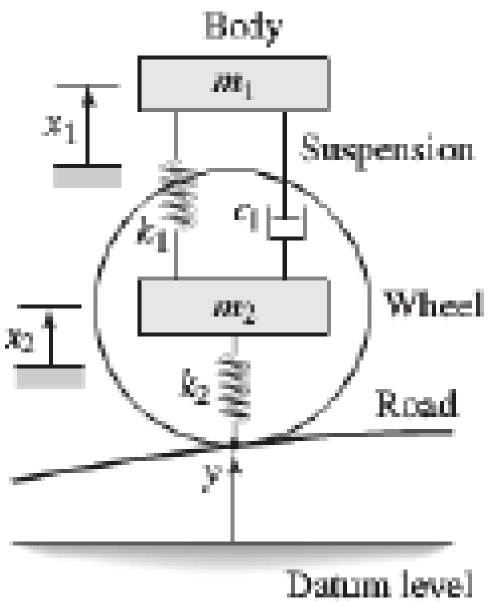
Python code for defining, solving, plotting ODE

http://tpcg.io/_XY8RO7

```python
def ode_system_ivp(t, S, carparams, roadparams):
    # returns the dotS
    # these params were stored in two lists, and must be passed in the correct order!
    m1 = carparams[0]
    m2 = carparams[1]
    c1 = carparams[2]
    k1 = carparams[3]
    k2 = carparams[4]
    ymag = roadparams[0]

    # define the forcing function equation
    y = ymag * np.sin(2 * t) if t < np.pi/2 else 0

    x1 = S[0]
    x1dot = S[1]
    x2 = S[2]
    x2dot = S[3]  # copy from the state array to nicer names

    # write the non-trivial equations
    x1ddot = (1 / m1) * (c1 * (x2dot - x1dot) + k1 * (x2 - x1))
    x2ddot = (1 / m2) * (-c1 * (x2dot - x1dot) - k1 * (x2 - x1) + k2 * (y - x2))
    # return the derivitaves of the state vector S
    return [x1dot, x1ddot, x2dot, x2ddot]
```

$$m_1\ddot{x}_1 = c_1(\dot{x}_2 - \dot{x}_1) + k_1(x_2 - x_1)$$

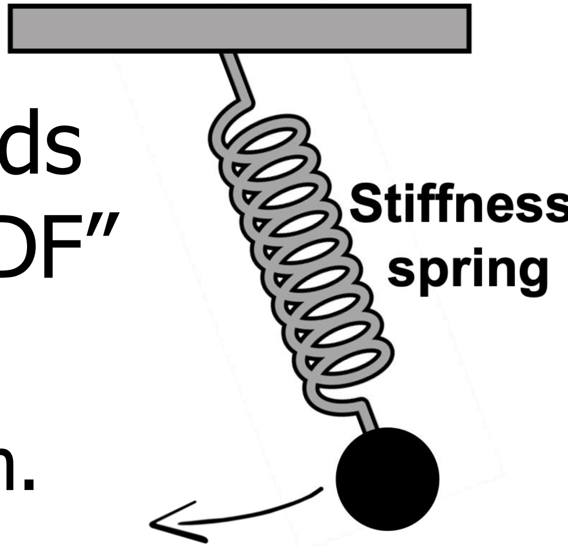
$$m_2\ddot{x}_2 = -c_1(\dot{x}_2 - \dot{x}_1) - k_1(x_2 - x_1) + k_2(y - x_2)$$



http://tpcg.io/_4Z5GG9

More about ODE solvers

- Stiffness of an ODE: A stiff ODE is difficult to solve numerically (takes longer, not stable, small steps).
- Particularly for systems with very different time/spatial scales, e.g., a very stiff spring
- In solve_ivp, use "RK45" or "RK23" methods for non-stiff problems, use "Radau" or "BDF" methods for stiff problems.
  - Try "RK45". If it fails, it's likely a stiff problem.



Boundary value problems

- An ODE with a set of constraints (boundary conditions)

$$\frac{d^2 f(x)}{dx^2} = \frac{df(x)}{dx} + 3$$

- IVP: specify f(0) and f'(0) and find f(x) for x > 0 given the ODE.
- BVP: specify f(0) and f(20) and find f(x) for x > 0 given the ODE. *Would be easy if we were given f'(0) as in IVP*
- In general, *n*th order ODE requires *n* constraints.

Generic formulation

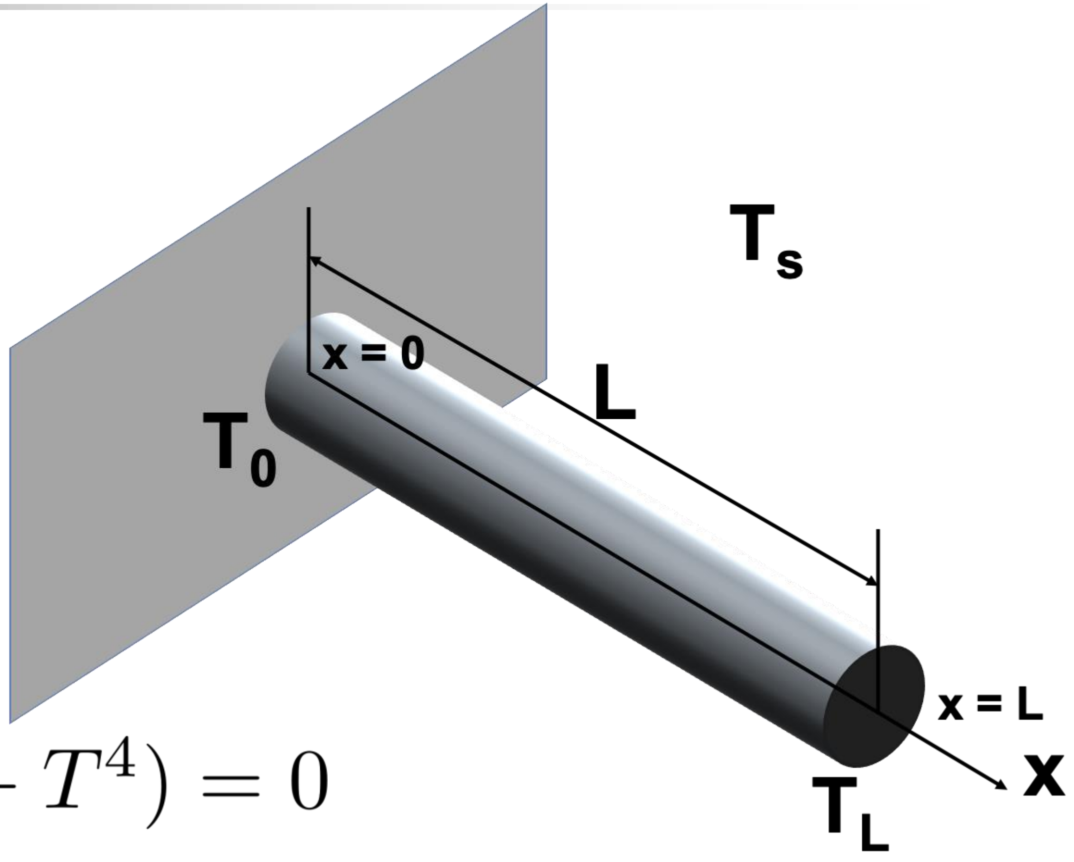
$$F\left(x, f(x), \frac{df(x)}{dx}, \frac{d^2 f(x)}{dx^2}, \frac{d^3 f(x)}{dx^3}, \ldots, \frac{d^{n-1} f(x)}{dx^{n-1}}\right) = \frac{d^n f(x)}{dx^n},$$

- x in a region [a,b], we need n boundary conditions at value a and b.
- For 2nd order case, we have different cases
  - f(a) and f(b) are given
  - f'(a) and f'(b) are given
  - f(a) and f'(b) are given or f(b) and f'(a) are given

Two-point BVP

Example

- Design of a cooling pin fin
- Consider both convection and radiation
- Steady state temperature distribution T(x)

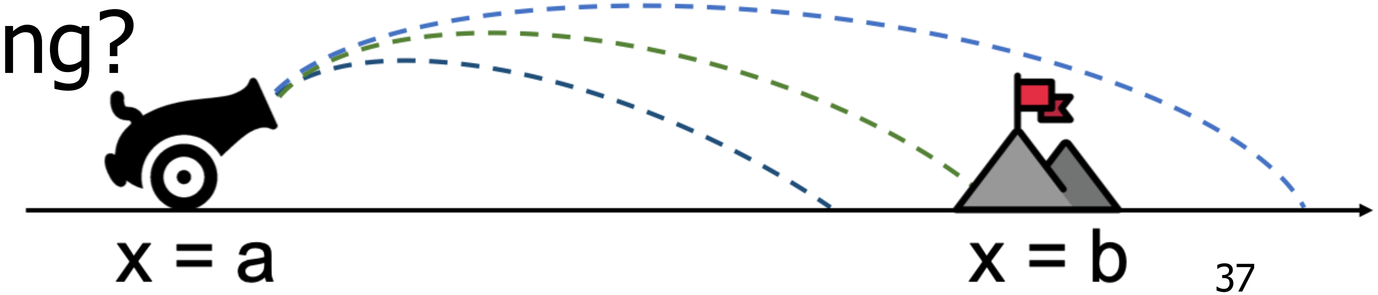


$$\frac{d^2T}{dx^2} - \alpha_1(T - T_s) - \alpha_2(T^4 - T^4) = 0$$

- $T(0)=T_0$, $T(L) = T_L$

The shooting methods

- Transform the BVP to an IVP and solve it.
- Iterative method: trial and error, enhanced with root finding. Say we are given $f(a)=f_a$ and $f(b)=f_b$.
  - Guess $f'(a)=d$. Together with $f(a) = f_a$, solve the IVP.
  - Obtain $f(b)=g$, which may not equal to f_b.
  - Adjust the initial guess and repeat (Goal: $f_b = g$)
- Last step: root finding?



Example

- Launch a rocket so that it reaches 50 m at 5 seconds. What should be the velocity at launching (no drag)?
- System: $d^2y(t)/dt^2=-g$, $y(0)=0$ and $y(5) = 50$. Need to find $y'(0)$?
- Analytically we can solve it $y'(0)=34.5$.
- Numerically, using the shooting method with root finding (e.g., secant method).

Python

```
def objective(v0):
        sol = solve_ivp(F, [0, 5], [y0, v0], t_eval = t_eval)
        y = sol.y[0]
        return y[-1] - 50

v0, = secant(objective, 10, 11)
print(v0)
```

Python BVP solver

- scipy.integrate.solve_bvp

solve_bvp(*fun*, *bc*, *x*, *y*, *p*=*None*, *S*=*None*, *fun_jac*=*None*, *bc_jac*=*None*, *tol*=*0.001*, *max_nodes*=*1000*, *verbose*=*0*, *bc_tol*=*None*)

- fun: similar to ivp, fun(x,y) or fun(x,y,p)
- bc: boundary conditions
- x: initial mesh
- y: initial guess at the mesh nodes

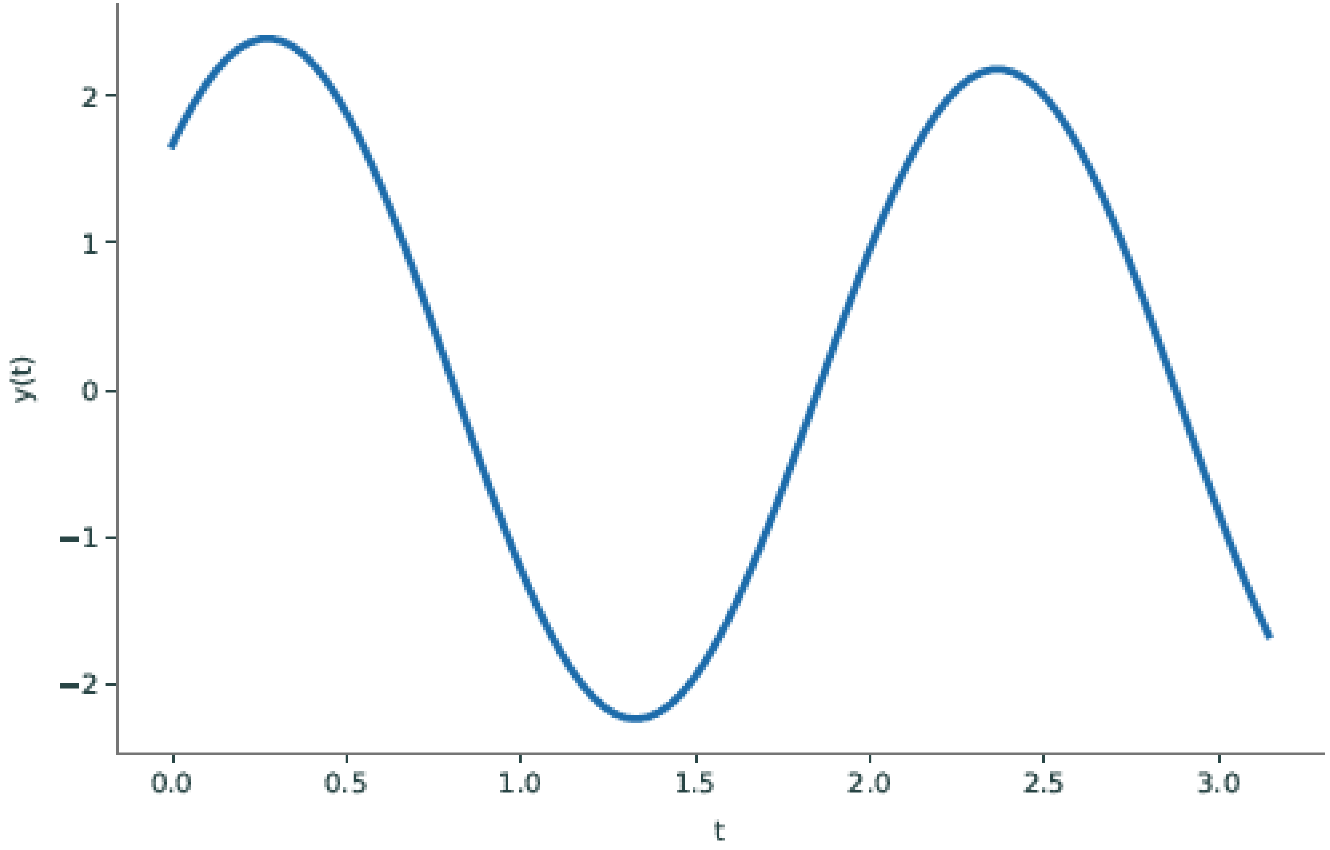
Example

$y'' + 9y = \cos(t)$,
$y'(0) = 5$,
$y(\pi) = -5/3$

http://tpcg.io/_DNZFYR

```
from scipy.integrate import solve_bvp
import numpy as np
# element 1: the ODE function
def ode(t,y):
''' define the ode system '''
return np.array([y[1], np.cos(t) - 9*y[0]])
# element 2: the boundary condition function
def bc(ya,yb):
''' define the boundary conditions '''
# ya are the initial values
# yb are the final values
# each entry of the return array will be set to zero
return np.array([ya[1] - 5, yb[0] + 5/3])
```

```
# element 3: the time domain.
t_steps = 100
t = np.linspace(0,np.pi,t_steps)
# element 4: the initial guess.
y0 = np.ones((2,t_steps))
# Solve the system.
sol = solve_bvp(ode, bc, t, y0)

import matplotlib.pyplot as plt
# here we plot sol.x instead of sol.t
plt.plot(sol.x, sol.y[0])
plt.xlabel('t')
plt.ylabel('y(t)')
plt.show()
```



- fun remains the same as ivp problem
- Must provide bc: 2 arrays representing initial and final values. bc evaluate to zero.
- Pass a linspace of $[t_0, t_f]$
- Pass an initial guess for all values

Notes

- sol.sol is a callable function. Plug in any value or numpy array, e.g., sol.sol(np.linspace), sol.sol(float), sol.sol(list).

- Pay attention to the initial values. Small changes can lead to large difference in the final approximations.

- BVP with free parameters can also be addressed.