# Python for Machine Learning

# Solving ODEs

# CE 5331 Machine Learning for Civil Engineers

Venki Uddameri, Ph.D. , P.E.

# Recap and Goals

- Installed Python and Anaconda Environments
- Introduction to Python
  - Setting working directory
  - Adding comment lines
    - Docstrings
- Introduction to Pandas
  - Reading a csv
  - Extracting columns (attributes)
  - Extracting rows
  - Obtaining summary measures
- Basic graphing and charting in Python
  - Matplotlib package
- Scipy
  - Interpolation
  - Kernel Density Functions
  - Integration (1D)
  - Integration (2D)

- Control Statements
  - If, if-elif-else, if-else
  - For loop
  - While loop
  - Use of Boolean operators
- Functions
  - Passing inputs
  - Lambda functions
  - Pass by object reference
- Numpy
  - Matrix Calculations
  - Vectorization
- Optimization
  - Unconstrained optimization
  - Constrained linear programming

Goal of this module is to explore Solution of ODEs

# ODEs

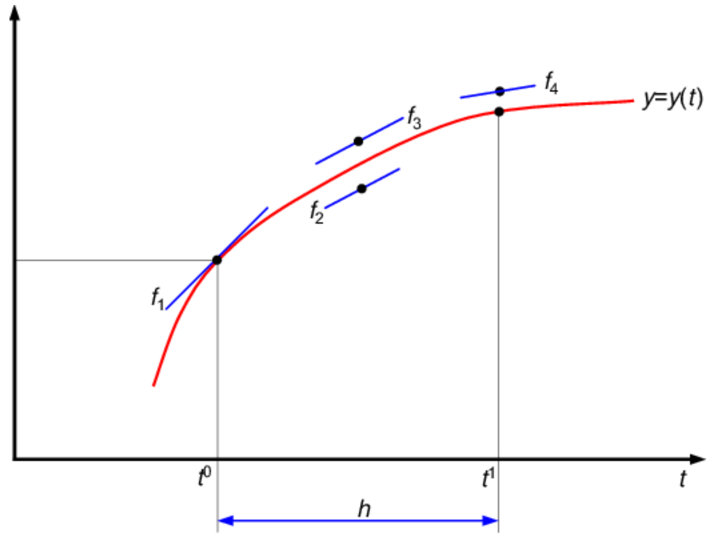
Ordinary Differential Equations (ODEs) are common in many civil engineering applications

First-Order ODEs describe the variations of a parameter with respect to either time or along an axis

Second-Order ODEs can be written as a system of 2 first-order ODEs

# ODE Solution Methods

- A system of first-order ODEs can be solved using numerical methods
  - Runge-Kutta-Fahlberg Methods are very popular
- Numerical methods essentially approximate the ODE into a set of algebraic equations
  - Solution of algebraic equations is easier
  - The derivative is approximated using a linear slope in a piece-meal manner
- Stiffness is an issue when solving some ODEs
  - Over some ranges small variations in parameters can cause large variations in the differential equation
    - Algebraic approximation becomes untrustworthy
    - Needs special handling




There is a generalized algorithm called ‘lsoda’ that handles both stiff and non-stiff equations well

It is an adaptive method that changes the step-size and solution method when it encounters stiffness

While original ‘lsoda’ algorithm was programmed FORTRAN it is available in Python (as well as R)

# ODE Solution

- We shall use the built-in lsoda implementation in scipy.integrate library
  - The function to solve ODE is called **odeint**
  - This solver can solve both a single and a system of ODEs

ODEs to solved must be in this form

$$\frac{dy}{dt} = af(y,t) + bf(t) + c$$

There can be multiple state variables (y)

$$\left.y(t=0) = y_o\right\} \text{Initial Condition}$$

An initial condition must be specified for each state variable

# Setting up to solve ODE

- The module must be imported from the scipy.integrate
  - Also import numpy
- We need to write a function for the RHS of ODE
  - This is called the model
- We need to specify initial conditions and time-steps where we need solution
  - Referred to as yo and t respectively
- We call odeint and solve the ODEs

```
import numpy as np
from scipy.integrate import odeint
```

Put y first
Put t next
Put other arguments

$$\frac{dy}{dt} = -kt$$

```
def model(y,t,k):
    dydt = -k * y
    return dydt
```

Return RHS of the ODE

Begin
End
# steps

```
t = np.linspace(0,5,101)
k = 0.3
yo = 5.0
y = odeint(model, y0, t,args=(k,))
```

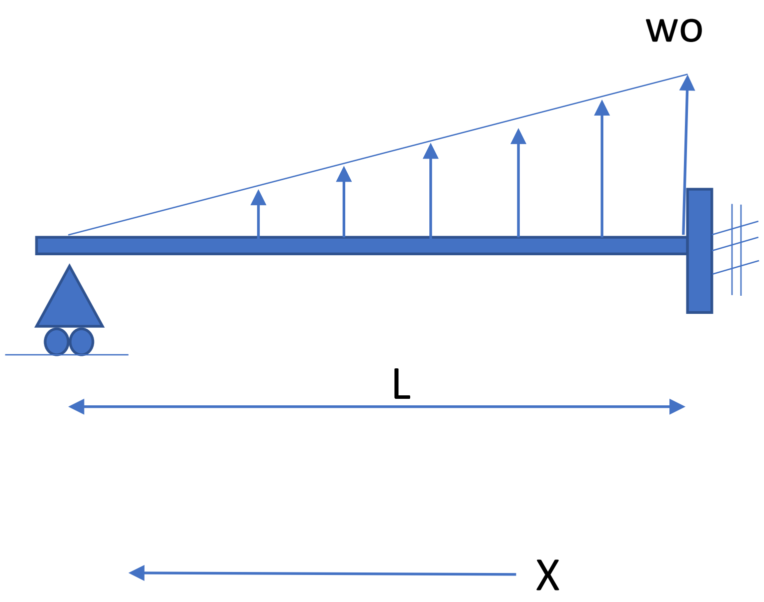
Times we need solution
Initial Condition
Trailing comma for tuple

# Illustrative Example 1

- Consider a simply supported beam with an uniformly increasing load
- Compute the deflection along the Y-axis

$$\frac{dy}{dx} = \frac{w_o}{120EIL}\left(-5x^4 + 6L^2x^2 - L^4\right)$$

$$\frac{dy}{dx} = 0 \, (\forall \, x = 0 \; and \, x = L)$$




| **Parameter** | **Value** |
|---|---|
| I = Moment of Inetia ($m^4$) | 0.0003 |
| E = Elastic Modulus ($N/m^2$) | $200 \times 10^9$ |
| wo = Max Linear load (N/m) | $2.5 \times 10^5$ |
| L = Maximum Length (m) | 3.0 |
| | |

# Illustrative Example

- The differential equation can be integrated by separation of variables to yield the following expression

$$\boxed{\begin{aligned}&\frac{dy}{dx} = \frac{w_o}{120EIL}\left(-5x^4 + 6L^2x^2 - L^4\right)\\&\frac{dy}{dx} = 0\,(\forall\, x = 0\, and\, x = L)\end{aligned}}$$

Separating the Variables and Integrating

$$\int dy = \frac{w_o}{120EIL}\int\left(-5x^4 + 6x^2L^2 - L^4\right)dx$$

Which yields the following solution

$$y = \frac{w_o}{120EIL}\left(-x^5 + 2x^3L^2 - L^4x\right)$$

Result from the numerical solution can be checked against this analytical solution

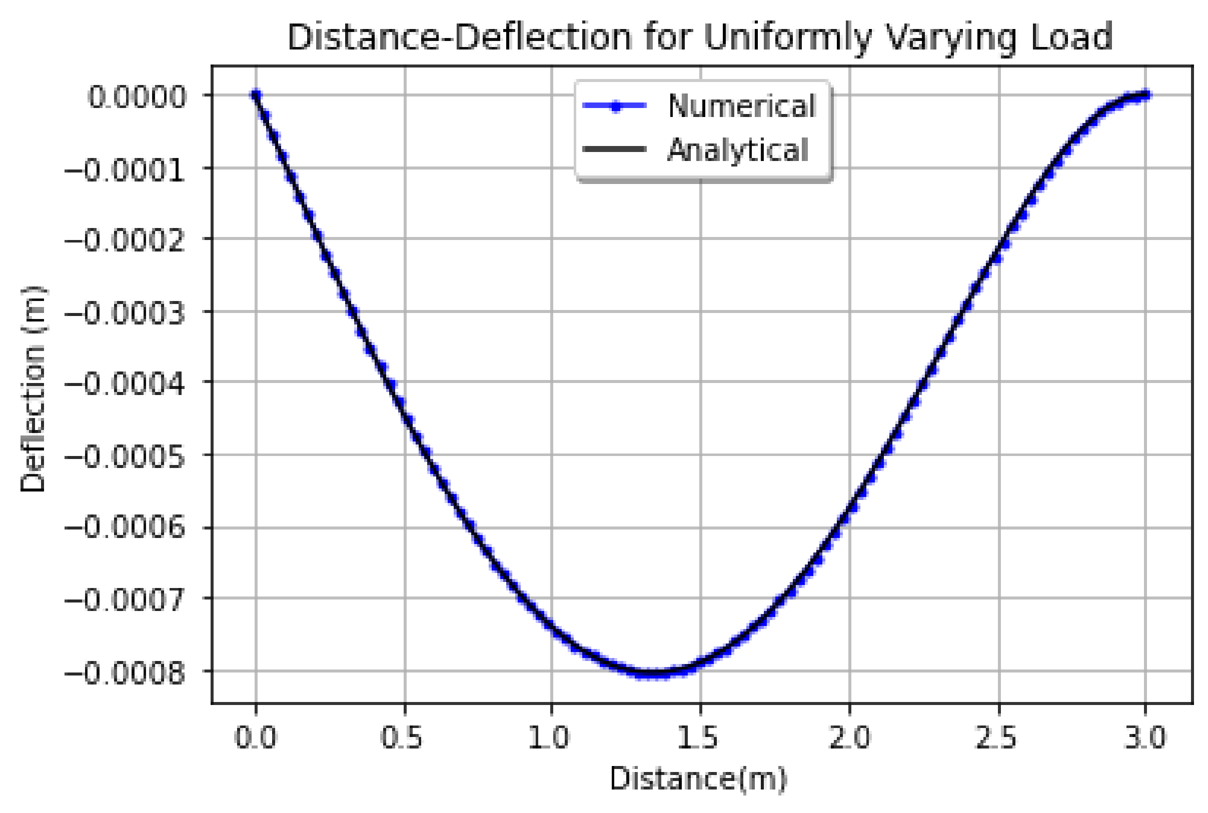
Note constant of integration is zero;
Which can be obtained from either substituting either y = 0 at x = 0 or y = 0 at x = L

# Python Code

```
# Step 0: Document the code
# -*- coding: utf-8 -*-
"""
Created on Mon Sep  7 22:06:21 2020
An example of ODE of beam deflection
@author: Venki
"""
# Step 1: Import Libraries
import numpy as np # for numerical calculation
from scipy.integrate import odeint # to solve ODE
from matplotlib import pyplot as plt # for graphing

# Step 2: Write Functions
# function to compute dy/dx
def dydx(y,x,L,wo,E,I):
    num1x = (wo)/(120*L*E*I)
    num2x = -5*x**4+6*x**2*L**2-L**4
    dydx = num1x * num2x
    return dydx

# function to compute analytical solution
def yanal(x,L,wo,E,I):
    num1x = (wo)/(120*L*E*I)
    y = num1x * (-x**5+2*x**3*L**2-L**4*x)
    return y

# Step 3: Obtain all inputs
# Write input parameters
E = 200. * 10**9
I = 0.0003
L = 3.0
wo = 2.5 * 10**5
Nx = 101 # 100 spaces
x = np.linspace(0,L,Nx) # split the length into 100 points
yo = 0  # initial conditions

# Step 4: Call Functions to solve ODE
ynum = odeint(dydx,yo,x,args=(L,wo,E,I)) # solve using lsoda numerical
yana = yanal(x,L,wo,E,I)

# Step 5: Make plots
# Create plots with pre-defined labels.
fig, ax = plt.subplots()
ax.plot(x, ynum, color ='blue', marker=".", markersize=6, label='Numerical')
ax.plot(x, yana, 'k-', label='Analytical')
plt.title('Distance-Deflection for Uniformly Varying Load')
plt.xlabel('Distance(m)')
plt.ylabel('Deflection (m)')
legend = ax.legend(loc='upper center', shadow=True)
plt.grid()
plt.show()
```

# Results

- Exact match between analytical and numerical solutions
- ‘lsoda’ algorithm in odeint works well



# Illustrative Example 2

- SEIR (susceptible-exposed-infectious-recovered) is a epidemiological modeling approach used to study the transmission of infectious diseases such as COVID-19. The model is given the following system of ODEs

$$\frac{dS}{dt} = \mu N - \mu S - \beta \frac{SI}{N}$$

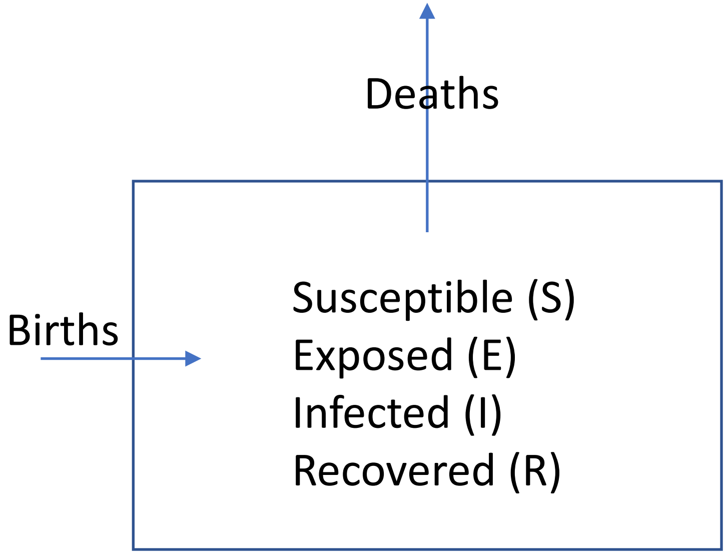
$$\frac{dE}{dt} = \beta \frac{SI}{N} - (\mu + \epsilon) E$$

$$\frac{dI}{dt} = \epsilon E - (\gamma + \mu + \alpha) I$$

$$\frac{dR}{dt} = \gamma I - \mu R$$

$$N = S + E + I + R$$

Deaths are implicitly calculated by the model




**Assume:**
Natural Birth rate ~ Natural Death Rate
Everyone is susceptible to infection
OK Assumption for places like Lubbock

Carcione JM, Santos JE, Bagaini C, Ba J. A simulation of a COVID-19 epidemic based on a deterministic SEIR model. arXiv preprint arXiv:2004.03575. 2020 Apr 7.

# Data

Table 1: Model Parameters

| Parameter | Value | Remarks |
|---|---|---|
| $\mu$ (1/d) | 3.81e-05 | Natural Death Rate; Based on life-expectancy of 72 years (Birth rate ~ Death Rate) |
| $\alpha$(1/d) ; | 0.006 | Virus-induced fatality rate |
| $\beta$ (1/d) | 0.75 | Probability of disease transmission per contact * Number of contacts / day (exposure rate) |
| $\varepsilon$ (1/d) | 1/3 | Progression from exposed to infection (reciprocal of Average incubation time) |
| $\gamma$ (1/d) | 1/15 | Recovery rate (reciprocal of average time of recovery) |

Table 2: Initial Conditions

| Param | Value | Remarks |
|---|---|---|
| $N_o$ | 225000 | Initial Population |
| $E_o$ | 2250 | 1% Initial exposure |
| $I_o$ | 1 | Patient zero |
| $S_o$ | 222749 | $S_o = N_o – E_o - I_o$ |
| $R_o$ | 0 | No initial recovery |

Data is for illustrative purposes only

# Python Code

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Sep  8 00:31:00 2020
SEIR Model for Covid-19 propogation
@author: Venki Uddameri
"""
# Step 1: Import libraries
import numpy as np # for numerical calculation
from scipy.integrate import odeint # to solve ODE
from matplotlib import pyplot as plt # for graphing
```

```python
# Step 2: Define Function
def seir(y,t,alpha,beta,gamma,eps,mu):
    S = y[0]
    E = y[1]
    I = y[2]
    R = y[3]
    N = S + E + I + R
    dsdt = mu*N - mu*S - beta*(S*I)/N
    dedt = beta*(S*I)/N - (mu + eps)*E
    didt = eps*E - (gamma + mu + alpha)*I
    drdt = gamma*I - mu*R
    zz = [dsdt,dedt,didt,drdt]
    return(zz)
```

```python
# Step 3 Define Model parameters
mu = 3.81e-05
alpha = 0.006
beta = 0.75
gamma = 1/15
eps = 1/3
```

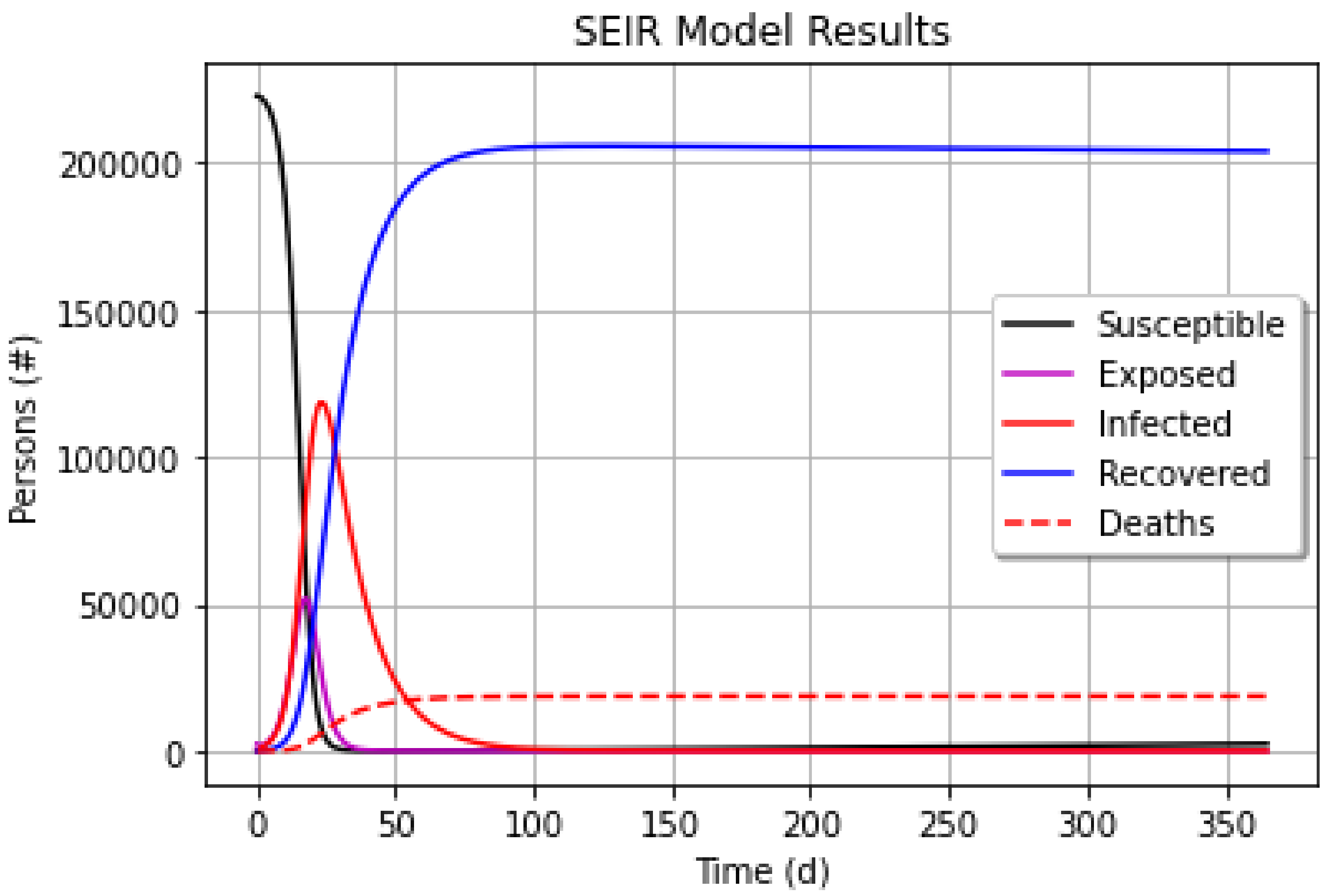
```python
# Step 4 Define Initial conditions & timesteps
No = 225000
Eo = 2250
Io = 1
So = 222749
Ro = 0
yo = [So,Eo,Io,Ro]
t = np.linspace(0,365,366) # Time Step 1 day
```

```python
# Step 5: Solve the model
SEIR = odeint(seir,yo,t,args=(alpha,beta,gamma,eps,mu))
D = No - (SEIR[:,0] + SEIR[:,1] + SEIR[:,2] + SEIR[:,3])
```

```python
# Step 6: Make plots
fig, ax = plt.subplots()
ax.plot(t, SEIR[:,0], 'k-', label='Susceptible')
ax.plot(t, SEIR[:,1], 'm-', label='Exposed')
ax.plot(t,SEIR[:,2],'r-',label='Infected')
ax.plot(t,SEIR[:,3],'b-',label='Recovered')
ax.plot(t,D,'r--',label='Deaths')
plt.title('SEIR Model Results')
plt.xlabel('Time (d)')
plt.ylabel('Persons (#)')
legend = ax.legend(loc='right', shadow=True)
plt.grid()
plt.show()
```

# Results

- Infection spreads early
- Nearly 50% of the people are infected at the peak (1 month)
- Death rate rise and become constant
- Recovered population increases




Use Pandas to get summary statistics or write a csv file to process in R

```
import pandas as pd
# Create a pandas data frame
seir.pd = pd.DataFrame(SEIR,columns=('S','E','I','R'))
seir.pd['D'] = D # add deaths to the data-frame
seir.pd.describe() # get summary statistics
seir.pd.to_csv('seir.csv',index_label = 'Time') #write to csv
```

# You Should Know

- How ODEs can be solved in Python
  - LSODA algorithm
- How to write the ODE function
- How to specify model parameters
- How to write the time/space steps
- How to call odeint
  - How to pass the function
  - How to pass the initial conditions
  - How to pass other arguments

How to make plots and summary statistics to analyze data