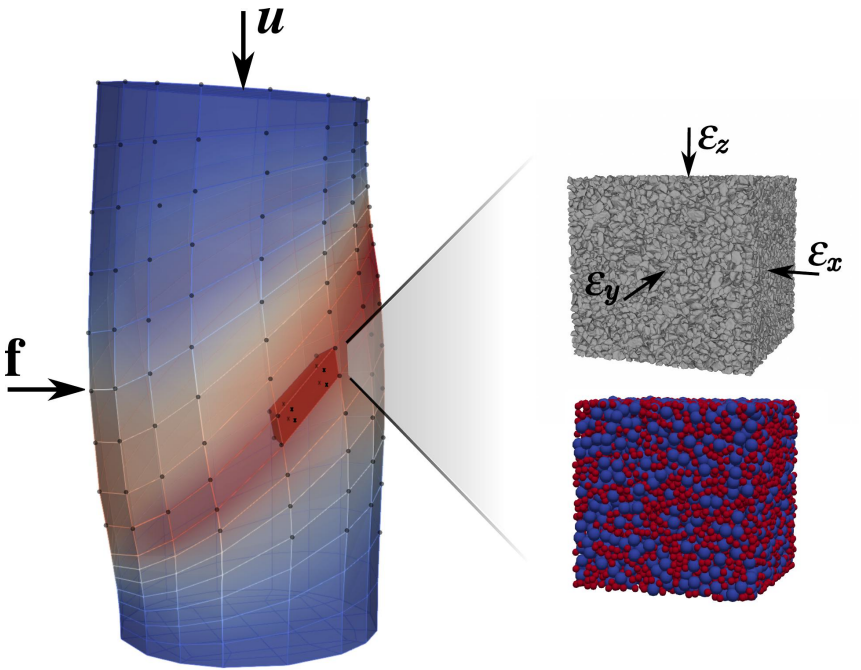


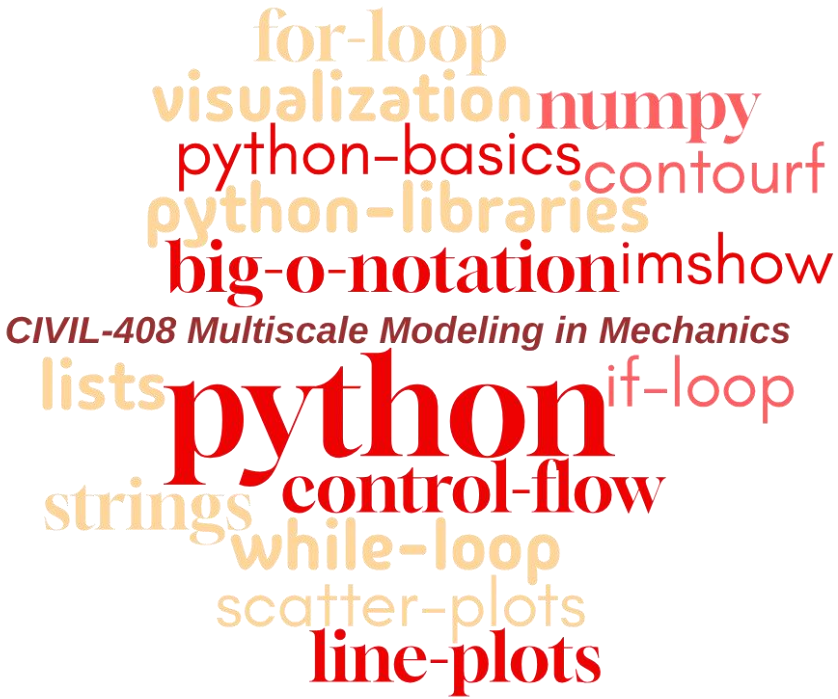
CIVIL-408 Multiscale Modeling in Mechanics

Prof. Kostas Karapiperis

Dr. Govinda Anantha Padmanabha

Exercises - Week 3 *Advanced Python Topics*

Recap


for-loop
visualization
numpy
python-basics
contourf
python-libraries
big-o-notation
imshow
CIVIL-408 Multiscale Modeling in Mechanics
lists
python
if-loop
strings
control-flow
while-loop
scatter-plots
line-plots

SciPy

- SciPy = **Scientific Python**
- Built on top of NumPy, it provides a wide range of **advanced numerical algorithms**.
- Provides:
  - **Linear algebra** routines & factorizations (scipy.linalg)
  - **Optimization** & curve fitting (scipy.optimize)
  - **Interpolation** & splines (scipy.interpolate)
  - Signal processing, statistics, sparse matrices, and more

- Why SciPy?
  - **Extends NumPy** with specialized scientific functions.
  - Trusted, well-tested algorithms from numerical libraries (e.g., LAPACK, BLAS).
  - Saves time vs. writing numerical methods from scratch.
  - Widely used in **engineering, mechanics, data science, and scientific research**.
- https://github.com/scipy/scipy

SciPy: Interpolation

- NumPy → only 1D linear interpolation
- SciPy (scipy.interpolate) → **flexible** 1D, 2D, 3D, and N-D interpolation (regular & scattered data)
- Provides a callable interpolation function, so you can reuse it for multiple queries.
- **Supports many interpolation** types:
  - "linear" (default)
  - "nearest"
  - "cubic" (spline-based)
- Higher-order splines (e.g., "quadratic")
- Works seamlessly with **irregularly spaced data**.

```python
import numpy as np
from scipy.interpolate import interp1d

x = np.linspace(0, 10, 5)
y = np.sin(x)

# Linear vs cubic interpolation
f_lin = interp1d(x, y, kind='linear')
f_cub = interp1d(x, y, kind='cubic')

x_new = np.linspace(0, 10, 50)
y_lin = f_lin(x_new)  # linear interpolation
y_cub = f_cub(x_new)  # cubic interpolation
```

SciPy: Optimization

- SciPy: minimize
  - General-purpose function minimization.
  - Supports **constraints and bounds**.
- SciPy: curve_fit
  - **Fits data to a model function** using least squares.
  - Useful for calibration in mechanics.

Syntax:

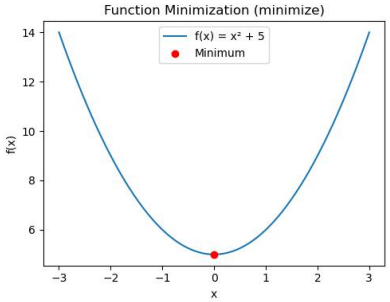
minimize(fun, x0, ...)

curve_fit(model, xdata, ydata)

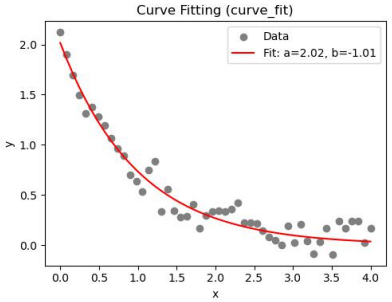
```python
from scipy.optimize import minimize, curve_fit
import numpy as np

# Minimize f(x) = x^2 + 5
f = lambda x: x**2 + 5
res = minimize(f, x0=2)

# Fit y = a*exp(bx)
def model(x, a, b): return a*np.exp(b*x)
xdata = np.linspace(0, 4, 50)
ydata = model(xdata, 2, -1) + 0.1*np.random.randn(50)
params, _ = curve_fit(model, xdata, ydata)
```



SciPy: minimize



SciPy: curve_fit

SciPy: Linear Algebra

- SciPy: solve
- Solves **linear systems** Ax=b.
  - More stable than writing elimination manually.
- Matrix factorizations:
  - LU (lu), QR (qr), SVD (svd)
  - Used in finite element & numerical methods.
- NumPy: basic linear algebra (good for small problems).
- SciPy: **advanced routines** & factorizations (needed in real **scientific computing**).

```python
from scipy.linalg import solve, lu
import numpy as np

A = np.array([[3, 2], [1, 4]])
b = np.array([6, 8])

# Solve Ax = b
x = solve(A, b)

# LU factorization
P, L, U = lu(A)
```

Syntax:

solve(A, b)

lu(A), qr(A), svd(A)

Functions

- Functions are **reusable blocks of code** that perform a specific task.
- They improve **modularity** by breaking large programs into smaller, manageable units.
- They enhance **clarity and readability** by giving descriptive names to operations.
- They **support reusability**, so the same code can be applied to different inputs.
- They **reduce errors and make debugging easier** in complex simulations.

```
# Compute stress for steel
strain = 0.001
E = 210e9
stress_steel = E * strain

# Compute stress for aluminum
strain = 0.001
E = 70e9
stress_al = E * strain
```

```
def compute_stress(strain, E):
    return E * strain

stress_steel = compute_stress(0.001, 210e9)
stress_al    = compute_stress(0.001, 70e9)
```

Function Syntax

- Functions in Python are defined with the **def** keyword.
- They can take **input arguments** and **return** outputs.
- A **docstring** can be added to explain the purpose of the function.
  - They can include information on **arguments** and **return values**.
- **Indentation** is required to define the function body.
- The **return** keyword passes results back to the caller.

```python
def compute_stress(strain: float, E: float) -> float:
    """
    Compute stress from Hooke's law.

    Args:
        strain (float): Strain.
        E (float): Young's modulus (Pa).

    Returns:
        float: Stress (Pa).
    """
    return E * strain
```

```python
steel = compute_stress(0.001, 210e9)
al    = compute_stress(0.001, 70e9)
```

Function Features

- Functions can have **default arguments**, allowing fallback values when inputs are not specified.
- They can be called with **keyword arguments**, improving clarity in simulation codes.
- They can accept **variable arguments** (*args, **kwargs) for flexible input handling.
- They support returning **multiple values**, useful for mechanics calculations with multiple outputs.
- They can be **nested or higher-order**, enabling advanced workflows and code reuse.

```python
def stress_energy(*strain_values, E=210e9, V=1.0, **kwargs):
    """Return (stresses, energies) for given strain values."""
    stresses = [E * strain for strain in strain_values]
    energies = [0.5 * V * E * strain**2 for strain in strain_values]
    return stresses, energies

stresses, energies = stress_energy(0.001, 0.002, E=70e9, V=2e-3)

print("Stresses:", stresses)
print("Energies:", energies)
```

Iterators & Generators

- **List-return computes and stores all results at once**, which is simple but memory-intensive and only practical for small datasets.
- Iterators allow looping over large datasets **without storing** everything in memory.
- Generators are functions that use **yield** instead of return.
- They produce values one at a time, pausing execution until the next call.
- They are efficient for simulations where results are computed step by step.

```python
def strain_list(n, step=0.001):
    return [i*step for i in range(n)]
```

```python
def strain_gen(n, step=0.001):
    for i in range(n):
        yield i*step
```

Objects

- Python supports **many different kinds of** data:
  - 210e9 7850 "Steel" [stress_xx, stress_yy, stress_xy]
  - {"E": 210e9, "v": 0.3}
- Each is an object, and every object has:
  - A **type** (Material, StressTensor, Geometry, …)
  - An **internal data representation** (scalar, vector, tensor, …)
  - A set of **procedures for interaction** with the object (e.g., compute stress, update strain, assemble stiffness)
- An object is an **instance** of a type
  - 210e9 is an instance of a Young's modulus (float)
  - "Steel" is an instance of a Material name (string)
  - [100, 80, 20] is an instance of a Stress state (list)

Object oriented programming (OOP)

- Everything in python is an object (and has a type)
- We can **create new objects** (*e.g.*, Materials, Elements, Models)
- We can manipulate objects (*e.g.*, compute stress, update strain, assemble stiffness)
- We can **destroy objects** (*e.g.*, remove an element, discard an RVE)

Advantages of OOP:

- Bundle data and procedures together into a **single package** (class).
- Divide-and-conquer development:
  - Implement and test **each class separately**
  - Increased modularity reduces complexity
- **Code reuse** through classes:
  - Many Python modules define new classes
  - Each class has its own namespace (no collisions in function names)
  - Inheritance allows subclasses to redefine or extend behavior

OOP: Class

- A class is a **blueprint** for **creating objects**.
- Use the class keyword to define a new type.
- Classes **bundle data** (properties) and **procedures** (methods) together.
- Similar to def, indent code to indicate which statements are part of the class definition.
- The word object means that Material is a Python object and inherits all attributes from the base class object.
  - Material is a subclass of object.
  - Object is a superclass of Material.

```
class Material(object):   # class definition
    """Define a material with properties"""
```

OOP: Creating and using classes

- Make a **distinction between** creating a class and using an instance of the class.
- **Creating the class** involves:
  - Defining the class name (Material)
  - Defining class attributes (Young's modulus)
  - For example, writing code to represent a material model
- **Using the class** involves:
  - Creating new instances of objects
  - Doing operations on the instances
  - For example, L=[1,2] and len(L)

```python
class Material:
    def __init__(self, name, youngs_modulus):
        """Initialize a material with name and Young's modulus."""
        self.name = name
        self.youngs_modulus = youngs_modulus
```

OOP: Class Attributes

- **Data attributes**
  - Think of these as properties that describe the object.
  - For example, a Material is defined by its Young's modulus.
- **Methods** (procedural attributes)
  - Think of methods as functions that only work with this class.
  - They describe how to interact with the object.
  - For example, a Material object may have a method to compute stress, but a plain number does not.

Data attributes:

```
steel.youngs_modulus
```

Methods :

```
steel.compute_stress(strain_value)
```

OOP: Instance of a class

- First have to define **how to create an instance** of a class
- Use a special method called __init__ to initialize data attributes.
- __init__ → special method (double underscores) called when a new object is created.
- self → reference to the object being constructed.
- Input arguments → values required to initialize the object (e.g., name, youngs modulus).
- Inside __init__, assign inputs to attributes

  (e.g. self.name, self.youngs_modulus).
- **Result:** each new object stores its own material properties.

```python
class Material:
    def __init__(self, name, youngs_modulus):
        """Initialize a material with name and Young's modulus."""
        self.name = name
        self.youngs_modulus = youngs_modulus

    def compute_stress(self, strain):
        """Return stress using Hooke's law."""
        return self.youngs_modulus * strain

# Create instances for Steel and Aluminum
steel = Material("Steel", 210e9)
aluminum = Material("Aluminum", 70e9)
```

OOP: Method

- A method is a **function defined inside a class** → works only with that class.
- Python automatically passes the object itself as the first argument (self).
- Use the . operator to access:
  - data attributes (steel.youngs_modulus)
  - methods (steel.compute_stress(strain_value))

```python
class Material:
    def __init__(self, name, youngs_modulus):
        self.name = name
        self.youngs_modulus = youngs_modulus

    def compute_stress(self, strain):
        return self.youngs_modulus * strain

steel = Material("Steel", 210e9)
print(steel.compute_stress(0.001))  # method call
```

OOP: Inheritance

- Inheritance → **allows one class to build upon another.**
- Avoids rewriting code for similar entities.
- The new class (child/subclass) inherits attributes and methods from the parent (base/superclass).
- The child can also add new attributes/methods or override existing ones.
- For example:
  - Material → base class (elastic model).
  - PlasticMaterial → subclass (adds yield stress).

Inheritance:

```python
class Material:
    def __init__(self, youngs_modulus):
        self.youngs_modulus = youngs_modulus

    def compute_stress(self, strain):
        return self.youngs_modulus * strain

# Subclass: extends Material
class PlasticMaterial(Material):
    def __init__(self, youngs_modulus, yield_stress):
        super().__init__(youngs_modulus)  # inherit base init
        self.yield_stress = yield_stress

    def compute_stress(self, strain):
        # override: elastic up to yield, then capped
        stress = super().compute_stress(strain)
        return min(stress, self.yield_stress)
```

Without Inheritance:

```python
class Material:
    def __init__(self, youngs_modulus):
        self.youngs_modulus = youngs_modulus

    def compute_stress(self, strain):
        return self.youngs_modulus * strain

# Separate PlasticMaterial (no inheritance)
class PlasticMaterial:
    def __init__(self, youngs_modulus, yield_stress):
        # repeated fields from Material
        self.youngs_modulus = youngs_modulus
        self.yield_stress = yield_stress

    def compute_stress(self, strain):
        # repeated formula from Material
        stress = self.youngs_modulus * strain
        return min(stress, self.yield_stress)
```

OOP: Polymorphism

K.Karapiperis

- Polymorphism → **same method name, different behavior depending on the object**.
- Allows using a common interface (compute_stress) across many material models.
- **Reduces code duplication:** no need to check material type manually.
- Makes simulation code more **flexible** (can swap in new materials without changing solver code).

Without Polymorphism:

```python
def compute_stress(material_type, youngs_modulus, strain, yield_stress=None):
    if material_type == "elastic":
        return youngs_modulus * strain
    elif material_type == "plastic":
        stress = youngs_modulus * strain
        return min(stress, yield_stress)

# Must manually check type each time
print(compute_stress("elastic", 210e9, 0.002))
print(compute_stress("plastic", 210e9, 0.002, yield_stress=250e6))
```

Polymorphism:

```python
class ElasticMaterial:
    def __init__(self, youngs_modulus):
        self.youngs_modulus = youngs_modulus
    def compute_stress(self, strain):
        return self.youngs_modulus * strain

class PlasticMaterial:
    def __init__(self, youngs_modulus, yield_stress):
        self.youngs_modulus = youngs_modulus
        self.yield_stress = yield_stress
    def compute_stress(self, strain):
        stress = self.youngs_modulus * strain
        return min(stress, self.yield_stress)

# Same interface
materials = [
    ElasticMaterial(210e9),
    PlasticMaterial(210e9, 250e6)
]

for mat in materials:
    print(mat.compute_stress(0.002))
```

OOP: Encapsulation

- Encapsulation → **bundles data** (attributes) and **methods** (behavior) inside a class.
- **Protects internal details** from unintended external changes.
- External code interacts only through well-defined interfaces (methods).
- Improves safety → **prevents overwriting** or **corrupting data accidentally**.

Encapsulation:

```python
class Material:
    def __init__(self, youngs_modulus):
        self.__youngs_modulus = youngs_modulus  # private attribute

    def compute_stress(self, strain):
        return self.__youngs_modulus * strain

steel = Material(210e9)
print(steel.compute_stress(0.001))   # allowed
print(steel.__youngs_modulus)        # error: not accessible
```

Without Encapsulation:

```python
class Material:
    def __init__(self, youngs_modulus):
        self.youngs_modulus = youngs_modulus  # public attribute

steel = Material(210e9)

# Directly changing internal data from outside
steel.youngs_modulus = -1000   # unrealistic, unsafe
print(steel.youngs_modulus)
```

OOP: Abstraction

- Abstraction → show only **essential features** and hide complex details.
- Users interact with a **simple interface (methods)** instead of internal steps.
- Reduces complexity for the user.
- Allows internal implementation to change without breaking external code.

Abstraction:

```python
class RVE:
    def solve_model(self):
        # Internally: assemble stiffness, apply BCs, iterate...
        print("Model solved")

# User side
micro_model = RVE()
micro_model.solve_model()   # simple call, details hidden
```

Without Abstraction:

```python
# User must handle everything manually
def assemble_stiffness():
    print("Assembling stiffness matrix...")

def apply_boundary_conditions():
    print("Applying BCs...")

def iterate():
    print("Iterating until convergence...")

# User side
assemble_stiffness()
apply_boundary_conditions()
iterate()
print("Model solved")
```

Suggested reading

Textbooks:

- **Python from Scratch**: Programming for absolute beginners with Python by Nilo Ney Coutinho Menezes
- **Python Crash Course**, Third Edition by Eric Matthes

Online:

- https://docs.python.org/
- **Python for Everybody** (https://www.py4e.com/book)
- https://www.python.org/ - The Python Tutorial — Python 3.9.6 documentation
- **W3 Schools** - Python Tutorial (https://www.w3schools.com/)