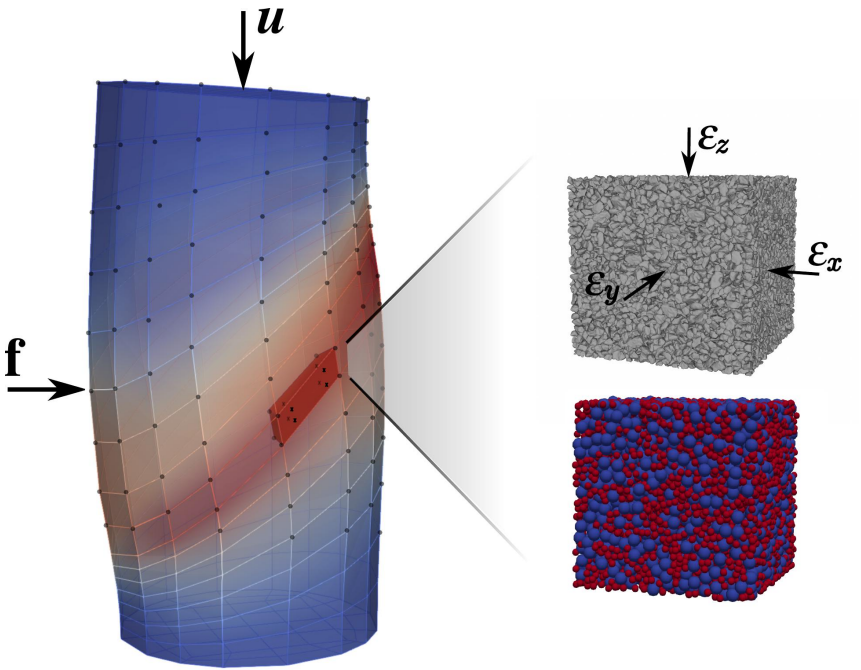


CIVIL-408 Multiscale Modeling in Mechanics

Prof. Kostas Karapiperis

Dr. Govinda Anantha Padmanabha

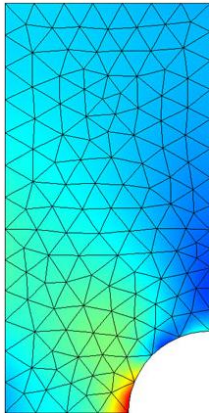
Exercises - Week 2

Intro to Python

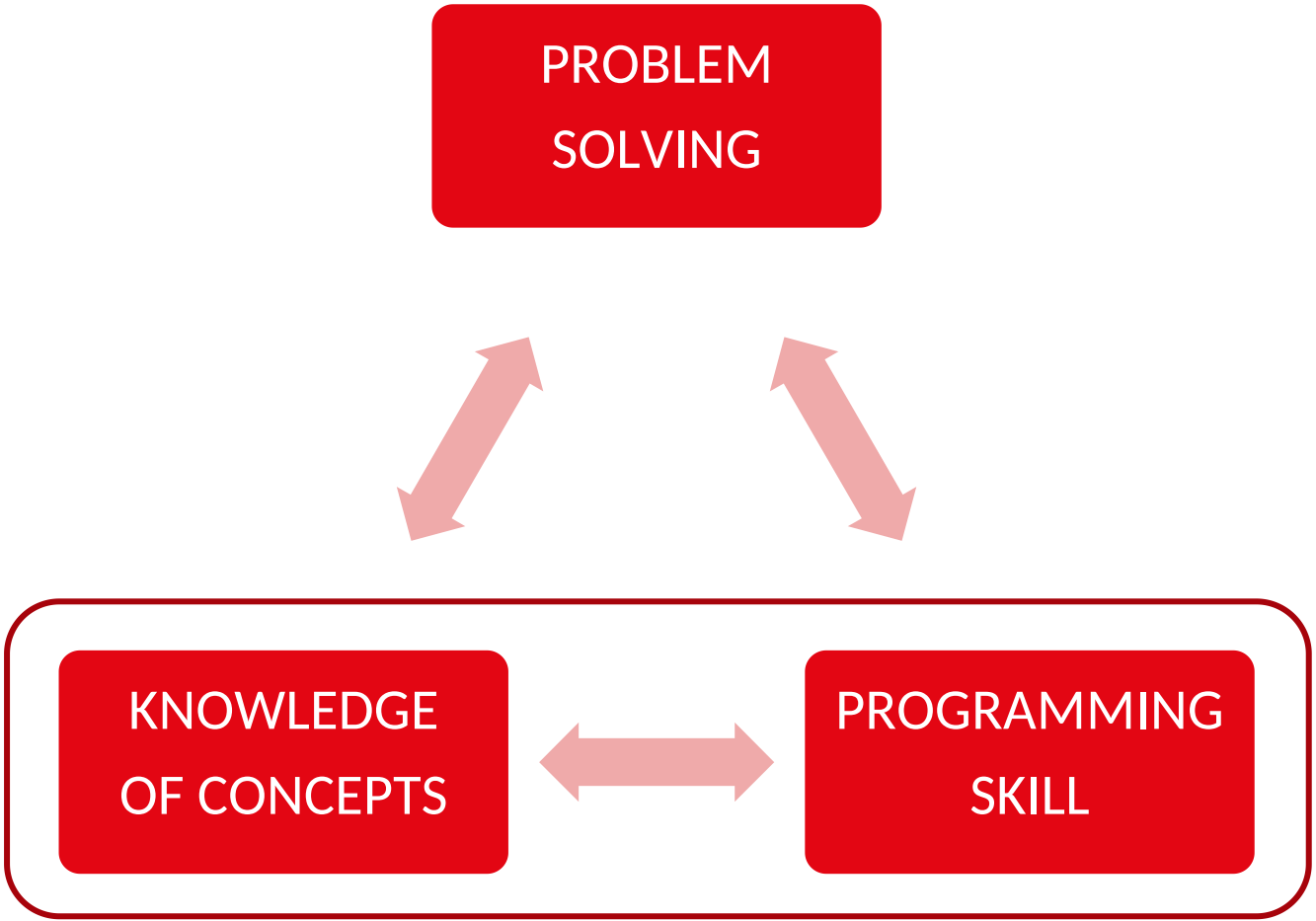
Python

PROBLEM SOLVING

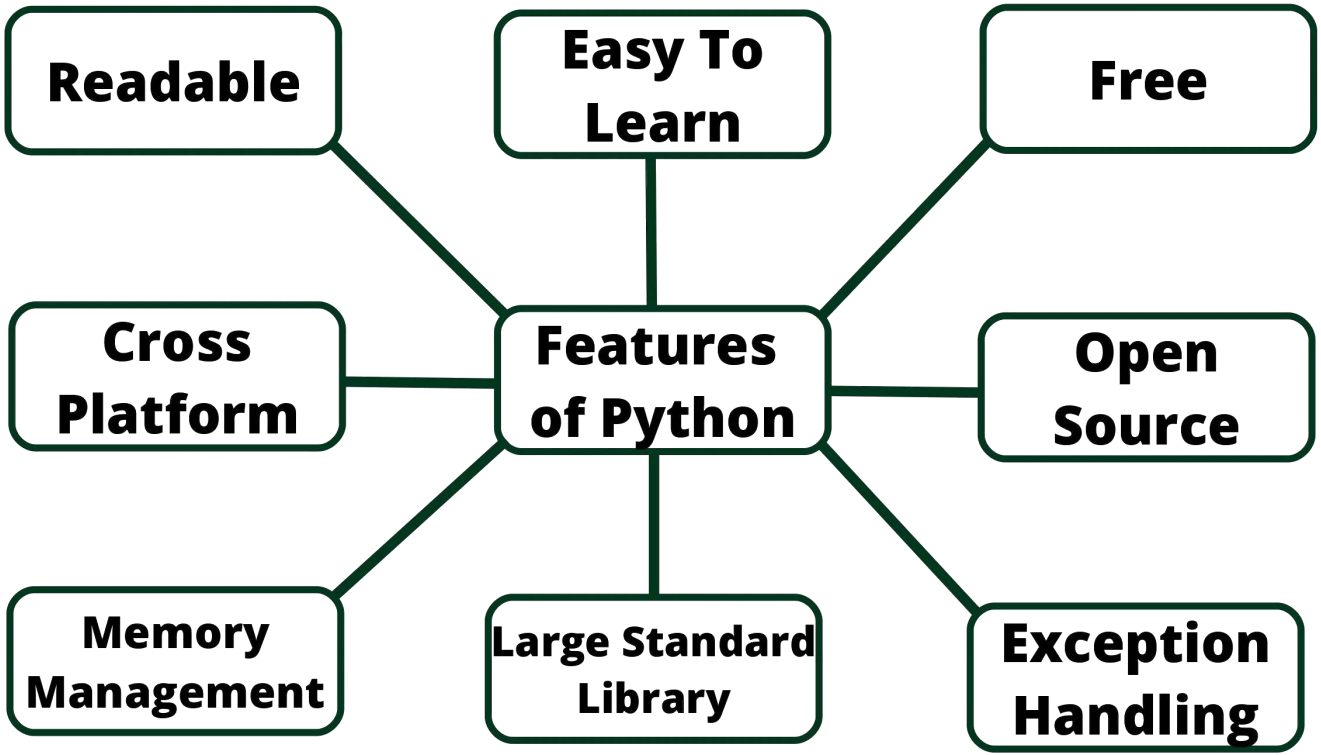
ANSYS/Abaqus	Python
• **Expensive**, licensed software. • Mostly used in industry and academia with institutional **licenses**.	• **Free, open-source**. • Large community support.
• **User-friendly GUI** (drag-and-drop, meshing tools, predefined materials). • Less coding required (though scripting possible).	• **No GUI** for FEA by default. • Needs libraries (NumPy, SciPy, PyAnsys, Abaqus Python scripting, FEniCS, etc.). • **Flexible** but **requires programming skills**.



Python



Why Python?


Readable
Easy To Learn
Free
Cross Platform
Features of Python
Open Source
Memory Management
Large Standard Library
Exception Handling

Why Python?

Python

```
print("Hello world.")
```

VS.

Java

```
public class HelloWorld {
    public static void main (String[]args) {
      System.out.println.("Hello world");
    }
}
```

Why Python?


Top 10 Python Libraries
Pandas
Data analysis and manipulation
NumPy
Mathematical functions
Matplotlib
Data visualisations
SeaBorn
Data visualisations
Tensorflow
Machine Learning
Keras
Deep Learning
SciPy
Scientific computing
PyTorch
Machine Learning
Scrapy
Web crawling
SQLModel
Interact with SQL databases

Why Python?


GARBAGE COLLECTION IN PYTHON
Garbage Collection
MEMORY MANAGEMENT IN C++
C++
Manual Memory Mgmt

Python basics

Variables

Binding Variables And Values

- Equal sign is an assignment of a value to a variable name

```
# Problem parameters
num_samples = 200
volume_fraction = 0.5

# Discretization
n_x = 10
n_y = 10

print(type(volume_fraction))    # float
print(type(num_samples))    # int
```

✓ 0.0s

```
<class 'float'>
<class 'int'>
```

Scalar objects

- **int** – represents integers
  e.g. N_atoms = 500
- **float** – represents real numbers
  e.g. a_lattice = 0.36
- **bool** – represents Boolean values True and False
  e.g. is_yielding = sigma > sigma_y
- **NoneType** – special and has one value, None
  e.g. result = None when no calculation is done
- Can use type() to see the type of an object

Python basics

Comparison/relational operations in Python

Operator	Meaning	Example	Result
>	Greater than	stress > sigma_y → 300 > 250	True
>=	Greater or equal	strain >= strain_y → 0.002 >= 0.002	True
<	Less than	a_lattice < b_lattice → 0.36 < 0.41	True
<=	Less or equal	T <= T_melt → 900 <= 1085	True
==	Equality test	phase == "solid"	True
!=	Inequality test	crystal != "BCC"	True

Python basics

Arithmetic operations in Python

Operator	Meaning	Example	Result
+	Addition	stress1 + stress2 → 120 + 80	200
-	Subtraction	length_initial - length_final → 10 - 9.8	0.2
*	Multiplication	force * distance → 50 * 0.1	5 (Work, J)
/	Division	force / area → 1000 / 50	20 (Stress, MPa)
//	Floor division	atoms // unit_cells → 105 // 4	26
%	Remainder	atoms % unit_cells → 105 % 4	1 (extra atom)
**	Power	radius ** 2 → 5 ** 2	25 (area factor)

Strings

- A **string** can contain letters, digits, spaces, and special characters
- Always enclosed in **quotation marks** (" ") or **single quotes** (' ')

```
material = "Steel"
crystal_structure = "FCC"
phase = "Solid"
experiment = "Tensile Test"

print("Material:", material)
print("Crystal:", crystal_structure)
```

✓ 0.0s

```
Material: Steel
Crystal: FCC
```

Lists

- An ordered sequence of information.
  - Access elements by index (**0-based**).
  - Defined using **square brackets** [].
- Contains elements:
  - Often **homogeneous**

    e.g. all integers.
  - Can **mix data types** (less common).
  - **Mutable** → elements can be changed after creation.

```
# list of Young's moduli (MPa)
E_values = [210e3, 70e3, 110e3]

# access first element
print(E_values[0])    # 210000.0

# change an element
E_values[1] = 75e3
print(E_values)       # [210000.0, 75000.0, 110000.0]

# list of mixed info
material = ["Steel", 210e3, "MPa"]
```

✓ 0.0s

Lists in Python

Iterating over a list

- Compute the **sum of elements** of a list
- **Common pattern**, iterate over list elements

Operations on lists

- Add elements to end of list with **L.append(element)**
- Mutates the list

```
# Iterating over a List
E_values = [210e3, 70e3, 110e3]

print(E_values[0])   # first element
print(E_values[1])   # second element
print(E_values[2])   # third element
```

```
E_values = [210e3, 70e3]    # MPa
E_values.append(110e3)      # add Cu
print(E_values)             # [210000.0, 70000.0, 110000.0]
```

Control Flow: IF Loop

- <condition> has a value *True* or *False*
- Evaluates expressions in that block if <condition> is *True*
- **Indentation**:
  - Defines blocks of code (no { } like in other languages).
  - Wrong indentation → IndentationError

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```python
stress = 300     # MPa
sigma_y = 250    # MPa

if stress < sigma_y:
    phase = "Elastic"
elif stress == sigma_y:
    phase = "Yielding"
else:
    phase = "Plastic"

print("Material state:", phase)
```

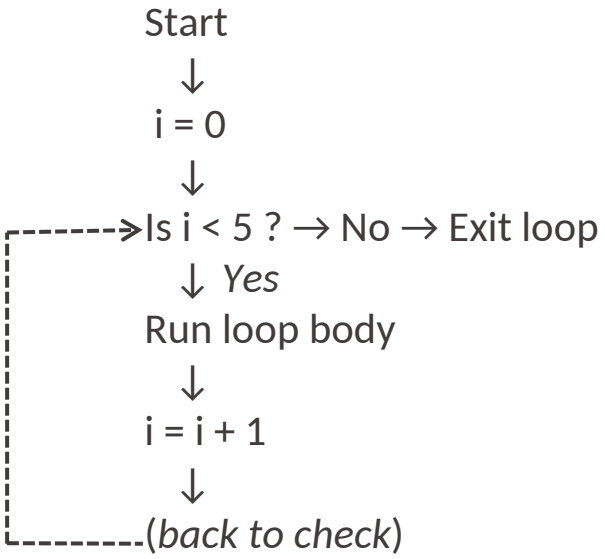

K.Karapiperis

Control Flow: For Loop

- **Each time** through the loop, <variable> takes a value.
- First time → <variable> starts at the **smallest value** (**0 by default**).
- Next times → <variable> **increases by 1**.
- **Stops** when it reaches **<some_num> - 1**.

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

```
# Generate Strain Values
for i in range(5):
    strain = i * 0.001
    print(strain)
```



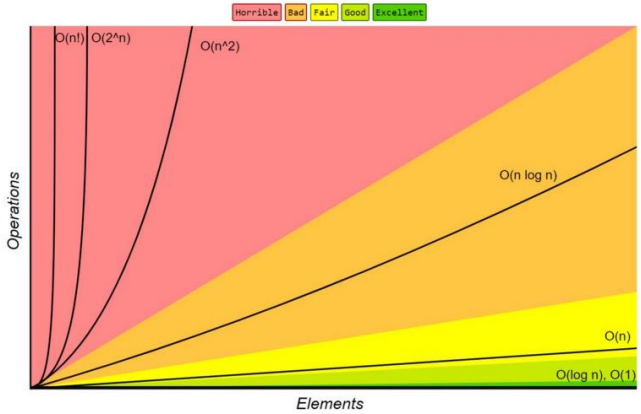
Control Flow: While Loop

```
while <condition>:
    <expression>
    <expression>
    ...
```

- While is useful when you don't know **how many steps in advance**
- Continues **until physics-based condition fails** (strain limit, grain growth cap, failure threshold, etc.)
- **<condition>** → evaluates to a Boolean (*True / False*).
- If <condition> is *True* → run all the steps inside the block.
- After finishing → check <condition> again.
- Repeat until **<condition> becomes *False***.

```
strain = 0.0
stress = 200  # MPa
creep_rate = 0.0005

while strain < 0.01:   # loop until strain reaches 1%
    strain += creep_rate
    print(f"Strain = {strain:.4f}")
```

Big O notation



Complexity	Notes
O(1)	Constant-time operation, independent of system size
O(log n)	Very efficient, grows slowly with system size
O(n)	Linear growth, scales proportionally to data size
O(n log n)	Common in sorting and multiscale algorithms
$O(n^2)$	Quadratic growth, quickly expensive for large n
$O(2^n)$	Exponential growth, impractical beyond small n
O(n!)	Factorial growth, impossible for large n

Example

Critical Stress Combination (Failure Criterion):

- In materials, two local stress concentrations can interact such that their combined effect exceeds the strength limit.
- Finding stress[i] + stress[j] == target identifies which two regions together trigger failure.

Example:

Input:

stress = [120, 80, 200, 150]

target = 350

Output:

[2, 3]

Explanation:

stress[2] + stress[3] = 200 + 150 = 350, which matches the strength limit.

Python Libraries

- What is a library?
  - **A collection** of pre-written code (functions, classes, modules).
  - Helps you **avoid reinventing** the wheel.
- Why use libraries?
  - **Extend** Python's capabilities.
  - Provide specialized tools for tasks (math, data, graphics, ML, etc.).
  - **Save time, increase reliability**.

- How to **use** a library?

```
import numpy as np
import matplotlib.pyplot as plt

# Seed for reproducibility
np.random.seed(2)
```

Python Libraries

Examples of useful libraries in mechanics:

- math (standard library) math.sqrt(), math.exp()
- random (standard library): Generate **random** grain orientations
- numpy (external): Arrays for **stress/strain** data, **linear algebra** for FEM
- scipy (external): **Curve fitting** for stress–strain models, **optimization** of parameters
- matplotlib (external): **Plot** stress–strain curves, grain size distributions
- pandas (external): Manage experimental **datasets** (e.g., tensile test results)

NumPy

- NumPy = **Numerical Python**
- Core package for **scientific computing** in Python.
- Provides:
  - **Multidimensional** arrays (ndarray)
  - **Mathematical functions** for linear algebra, statistics, Fourier transforms, random numbers
  - Tools for handling large datasets efficiently
- Why NumPy?
  - Arrays are implemented in C → **faster than Python lists**.
  - Supports **vectorized operations** (no need for slow **Python loops**).
  - **Foundation** of SciPy, Pandas, Matplotlib, Scikit-learn, TensorFlow.
- *https://github.com/numpy/numpy/tree/v2.0.0*

NumPy

Feature	Python List	NumPy Array (ndarray)
Data type	Can mix types (e.g., int + str)	Homogeneous (all floats, all ints, etc.)
Operations	Element by element (need a loop)	Vectorized (no loop needed)
Speed	Slow for large data	Optimized, very fast
Convenience	Manual operations	Built-in math, linear algebra, FFT, etc.
Mechanics Example	Store stress values as list of floats	Compute stress-strain curve in one line

```python
E = 210e3
strain_list = [0.0, 0.001, 0.002, 0.003, 0.004]
stress_list = []
for eps in strain_list:
    stress_list.append(E * eps)

print(stress_list)   # [0.0, 210.0, 420.0, 630.0, 840.0]
```

```python
import numpy as np
E = 210e3
strain = np.linspace(0, 0.004, 5)  # array of strains
stress = E * strain                # vectorized operation
print(stress)   # [   0.  210.  420.  630.  840.]
```

NumPy

- NumPy: **np.arange()**
- Function to create **regularly spaced** values.
- Similar to Python's built-in range(), but returns a **NumPy array** instead of a list.
- With lists, **you'd need a loop**.
- With NumPy + np.arange, **one line creates** your entire dataset.

```
# Problem parameters
num_samples = 10
volume_fraction = 0.5

# Grid sizes to test (L/l)
n_x = np.arange(5, 35, 5)
n_y = np.arange(5, 35, 5)
```

Syntax:

np.arange(start, stop, step, dtype=None)

- start → first value (**default = 0**)
- stop → generate numbers up to, but not including, this value
- step → spacing between numbers (**default = 1**)
- dtype → optional, specify data type (*e.g.*, float)

NumPy

np.zeros(array)

- Creates an array filled with zeros.
- Useful for initializing arrays, placeholders, or matrices.

np.where

- Returns indices or elements that satisfy a condition.
- Works like a vectorized “if” statement.

np.mean(array)

- Computes the average of array elements.
- Can also compute mean along rows or columns.

np.random.seed

- Sets the random seed (starting point for the random number generator).
- Ensures reproducibility → every time you run the code, you get the same “random” numbers.
- Very important in scientific computing (so others can reproduce your results).

K.Karapiperis

NumPy

np.linspace(start, stop, num)

- Returns num evenly spaced points between start and stop (inclusive).
- Very useful for stress–strain curves, time steps, or sampling domains.

np.random.randint

- np.random.randint(n) → returns a random integer from 0 to n-1.
- Useful for “random sampling” of data points (e.g., stress values, grains, etc..).

np.sum(array)

- Returns the sum of all elements in the array.
- Much faster and cleaner than looping through a Python list.

np.std(array)

- Computes the standard deviation of array values.
- Standard deviation = how much the values vary around the mean.
- Useful in mechanics for microstructural variability or uncertainty in measurements.

NumPy Linear Algebra

K.Karapiperis

np.linalg.eig(A)

- Computes eigenvalues and eigenvectors of a square matrix.
- Useful in mechanics: principal stresses, vibration modes.

np.linalg.det(A)

- Computes the determinant of a square matrix.

np.trace(A)

- Computes the trace (sum of diagonal elements).
- Related to invariants of stress/strain tensors.

np.linalg.inv(A)

- Computes the inverse of a square matrix.
- Only valid if det(A) ≠ 0.

NumPy

- NumPy arrays can **represent vectors, matrices, and tensors** (*e.g.*, stress, strain, stiffness).
- **Element-wise operations** apply to each component of the array, similar to operating on tensor components in mechanics.
- Useful when handling field quantities at multiple points/elements in multiscale models.

Stress Superposition (Addition):

$\sigma_{total} = \sigma_{mech} + \sigma_{thermal}$

```
import numpy as np
sigma_mech = np.array([100, 50, 25])      # MPa
sigma_thermal = np.array([10, 10, 10])    # MPa
sigma_total = sigma_mech + sigma_thermal
```

Stress Increment (Subtraction):

$\Delta\sigma = \sigma_{new} - \sigma_{old}$

```
sigma_new = np.array([120, 55, 30])
sigma_old = np.array([100, 50, 25])
delta_sigma = sigma_new - sigma_old
```

NumPy

- **Tensor** Operations with NumPy
- NumPy extends element-wise operations to **matrices/tensors**

→ works naturally for stress, strain, stiffness fields in mechanics.

$$\sigma_{\text{total}} = \sigma_{mech} + \sigma_{thermal}$$

$$\sigma_{mech} = \begin{bmatrix} 100 & 20 & 0 \\ 20 & 80 & 0 \\ 0 & 0 & 60 \end{bmatrix}$$

$$\sigma_{thermal} = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

$$\sigma_{\text{total}} = \begin{bmatrix} 110 & 20 & 0 \\ 20 & 90 & 0 \\ 0 & 0 & 70 \end{bmatrix}$$

```python
import numpy as np

sigma_mech = np.array([[100, 20, 0],
                       [20,  80, 0],
                       [0,    0, 60]])

sigma_thermal = np.array([[10, 0, 0],
                          [0, 10, 0],
                          [0,  0, 10]])

sigma_total = sigma_mech + sigma_thermal
print(sigma_total)
```

Solution of Linear Systems

- **np.linalg.solve(A, b)** is preferred over computing $\mathbf{A}^{-1}$ because it is faster and numerically more stable.

Consider a 2D truss with 3 unknown nodal displacements u_1,u_2,u_3

Global stiffness matrix:

$$\mathbf{K} = \begin{bmatrix} 12 & -6 & 0 \\ -6 & 12 & -6 \\ 0 & -6 & 12 \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} 100 \\ 0 \\ 50 \end{bmatrix}$$

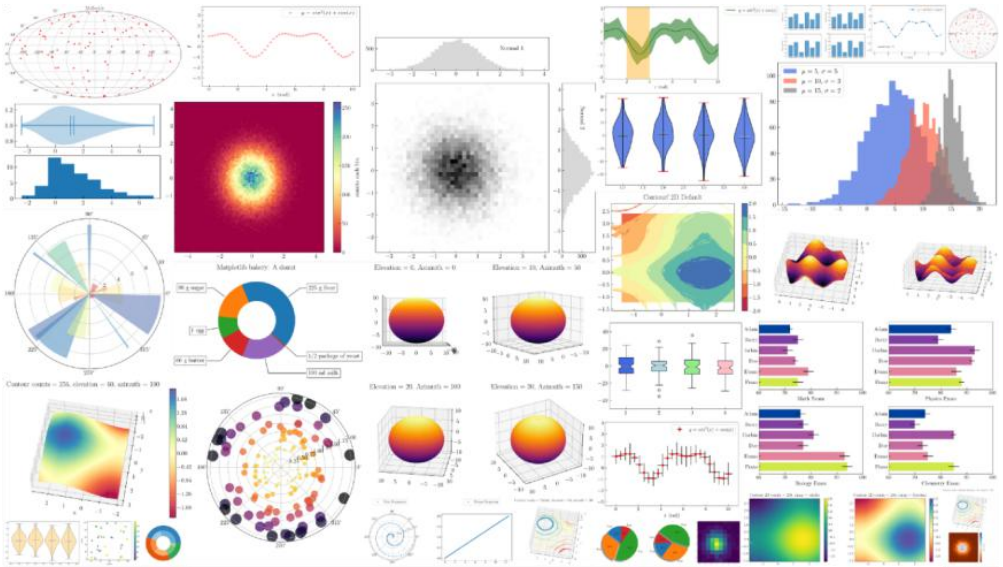
$$\mathbf{u} = \mathbf{K}^{-1}\mathbf{F} = \begin{bmatrix} 11.11 \\ 16.67 \\ 18.89 \end{bmatrix}$$

```python
import numpy as np

K = np.array([[12, -6, 0],
              [-6, 12, -6],
              [0, -6, 12]])

F = np.array([100, 0, 50])

u = np.linalg.solve(K, F)
print(u)   # [11.11 16.67 13.89]
```

Visualization

Matplotlib: the **main plotting library** in Python.
Allows creation of **figures, subplots, axes, labels, legends**.

- Line plot (sine wave)
- Histogram (horizontal)
- Bar chart
- Violin plots
- Heatmap (2D colormap)
- Scatter plot (black density points)
- Radar / polar chart
- Pie chart
- 3D surface plots (multiple angles)
- Contour plot (filled contour levels)
- *Etc.…*



Visualization

Scatter Plots

- Show discrete data points (measurements, samples).
- Syntax: plt.scatter(x, y, marker='o')

Line Plots

- Show trends, averages, or continuous curves.
- Syntax: plt.plot(x, y, lw=2)

Field / Spatial Plots: Used for stress, strain, temperature fields in mechanics.

Contourf

- Smooth filled contours of a field.
- Syntax: plt.contourf(X, Y, field, cmap="viridis")

Imshow

- Displays 2D data as a pixel/heatmap image.
- Syntax: plt.imshow(field, cmap="", origin="lower")

Visualization

Scatter Plot – *Discrete Measurements*. Use for experimental data points

```python
import matplotlib.pyplot as plt

strain = [0.0, 0.001, 0.002, 0.003]
stress = [0, 210, 420, 630]

plt.scatter(strain, stress, marker='o')
plt.xlabel("Strain")
plt.ylabel("Stress [MPa]")
plt.title("Stress-Strain (measured points)")
plt.show()
```

Line Plot – *Continuous Curves*. Use for theoretical models or simulation results.

```python
import numpy as np
strain = np.linspace(0, 0.003, 50)
E = 210e3
stress = E * strain

plt.plot(strain, stress, lw=2)
plt.xlabel("Strain")
plt.ylabel("Stress [MPa]")
plt.title("Stress-Strain (linear elastic model)")
plt.show()
```

Contourf – *Field Visualization*. Use for stress or temperature fields in 2D.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 50)
y = np.linspace(-2, 2, 50)
X, Y = np.meshgrid(x, y)
stress_field = X**2 + Y**2   # toy stress distribution

plt.contourf(X, Y, stress_field, cmap="viridis")
plt.colorbar(label="Stress [MPa]")
plt.title("Stress Field")
plt.show()
```

Imshow – *Heatmap / Microstructure*. Use for pixel-based microstructure or FEM mesh visualization.

```python
import numpy as np
import matplotlib.pyplot as plt

micro = np.random.randint(0, 3, (10, 10))  # 3 grain types

plt.imshow(micro, cmap="viridis", origin="lower")
plt.colorbar(label="Grain type")
plt.title("Synthetic Microstructure")
plt.show()
```

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/)