Part 1: Crystal structure foundation

1.1 Understanding Crystal Structures as Data

To human, a crystal is a periodic arrangement of atoms.
To an AI model, a crystal is a periodic graph or a high-dimensional tensor.
Before we can train a model, we must understand how to "tokenize" a crystal structure into variables that a computer can process.

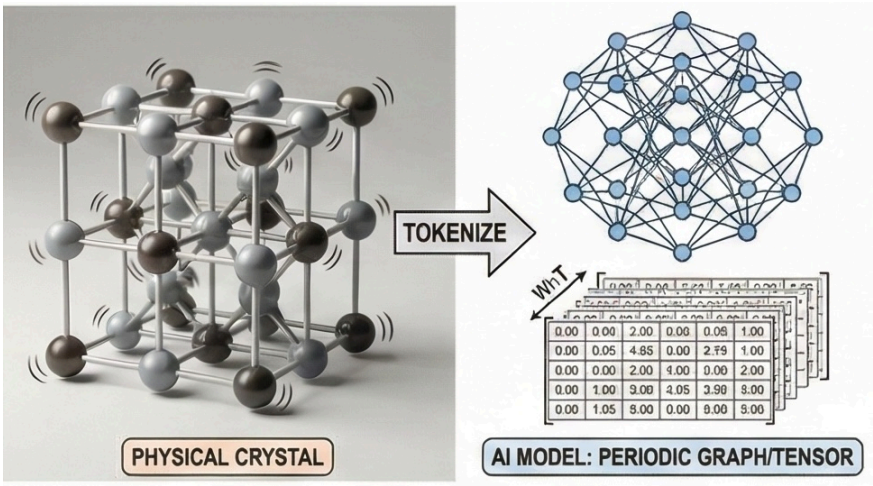

Fig. 1: Crystal to data

1.2 The Fundamental Components

Any crystal structure is defined by two primary pieces of information:

- **The Lattice (The "Container"):** A 3×3 matrix of lattice vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ that defines the repeating unit cell and its periodicity.
- **The Basis (The "Content"):** A list of atomic species (elements) and their corresponding positions within that unit cell.

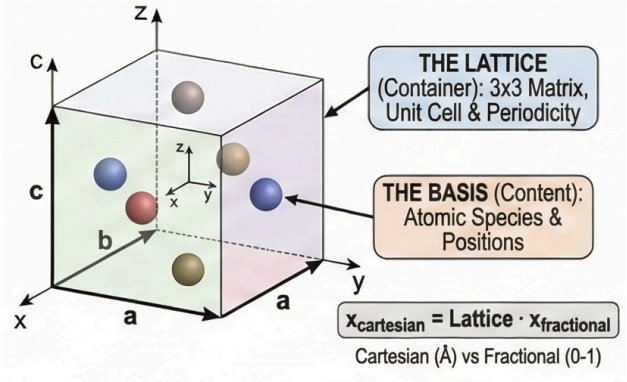

Fig. 2: Crystal structure

Note on Coordinates: While physicists often use Fractional Coordinates (ranging from 0 to 1), many ML models (like SchNet or DeepMD) require Cartesian Coordinates (Ångströms). The conversion is:

$$\vec{X}_{cartesian} = \text{Lattice} \cdot \vec{x}_{fractional}$$

1.3 The POSCAR Format: the standard for periodic systems

While `.xyz` files are common for molecules, the **POSCAR** format (originally from the **VASP** software) is the standard representation for **periodic crystals** in condensed matter physics. One of its key advantages is that it **explicitly separates the lattice geometry from the atomic positions**, making it ideal for crystalline systems.

Format structure

- **Line 1:** Comment line (usually the system name)
- **Line 2:** Universal scaling factor (usually `1.0`)
- **Lines 3–5:** The three lattice vectors
  (\mathbf{a}, \mathbf{b}, \mathbf{c}) defining the unit cell
- **Line 6:** Atomic species labels
  (optional in older formats, standard in VASP 5+)
- **Line 7:** Number of atoms for each species
- **Line 8:** Coordinate system
  (`Direct` for fractional coordinates, or `Cartesian`)
- **Line 9+:** Atomic positions

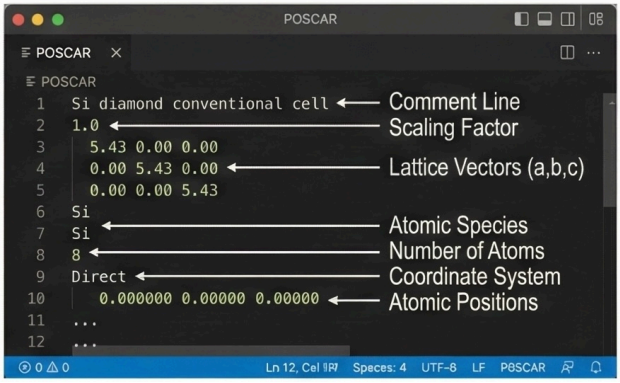

Fig. 3: POSCAR

Example A: Silicon (Diamond Structure)

Silicon in its ground state has a cubic unit cell with 8 atoms. Here is how that unit cell looks in a standard Cartesian coordinate list ($a = 5.43$ Å):

```
Si diamond conventional cell
1.0
  5.43  0.00  0.00
  0.00  5.43  0.00
  0.00  0.00  5.43
Si
8
Direct
  0.00000  0.00000  0.00000
```

```
  0.25000  0.25000  0.25000
  0.00000  0.50000  0.50000
  0.25000  0.75000  0.75000
  0.50000  0.00000  0.50000
  0.75000  0.25000  0.75000
  0.50000  0.50000  0.00000
  0.75000  0.75000  0.25000
```

Example B: Perovskite ($SrTiO_3$)

Perovskites follow the ABO_3 formula. In a simple cubic unit cell ($a \approx 3.90$ Å), the atoms are positioned at the corners, center, and face-centers:

```
SrTiO3 cubic perovskite
1.0
  3.905  0.000  0.000
  0.000  3.905  0.000
  0.000  0.000  3.905
Sr Ti O
1  1  3
Direct
  0.00000  0.00000  0.00000
  0.50000  0.50000  0.50000
  0.50000  0.50000  0.00000
  0.50000  0.00000  0.50000
  0.00000  0.50000  0.50000
```

1.4 The CIF Format: a crystallographic standard

The **CIF (Crystallographic Information File)** format is a **community standard** developed by the International Union of Crystallography (IUCr). It is the most widely used format for **experimentally determined crystal structures**, especially those obtained from **X-ray, neutron, or electron diffraction**.

Unlike POSCAR, which is optimized for **first-principles simulations**, CIF is designed to be **self-describing**, human-readable, and suitable for **archival and data exchange**.

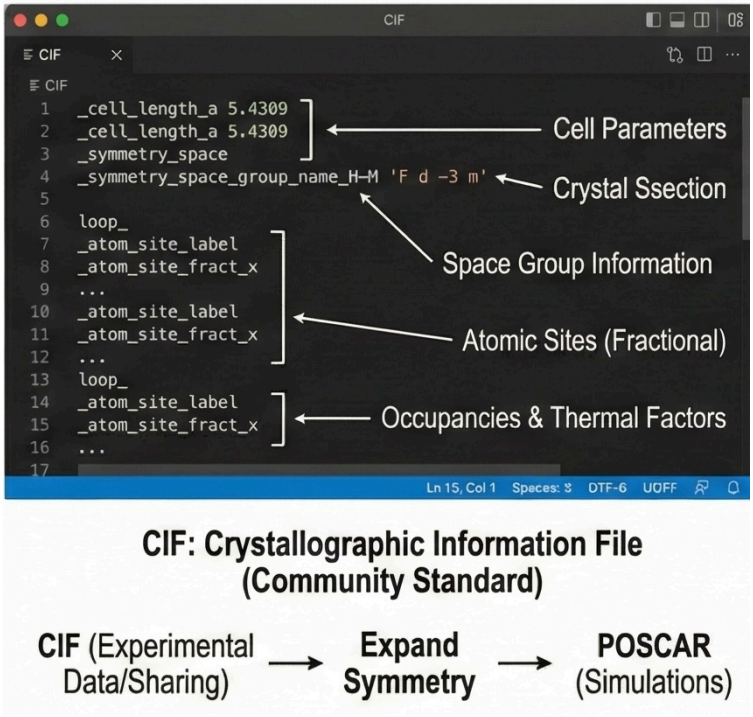

Fig. 4: CIF

Key characteristics of CIF

- Text-based, keyword–value format
- Strongly standardized (dictionary-driven)
- Commonly used by:
    - ICSD
    - COD
    - CCDC
    - Journals and crystallographic databases
- Native format for **space group symmetry** and **Wyckoff positions**

Typical CIF contents

- **Lattice parameters:**
  (a, b, c, α, β, γ)
- **Space group information:**
  Hermann–Mauguin symbol and symmetry operations
- **Atomic sites:**
  fractional coordinates ((x, y, z))
- **Occupancies and thermal factors (B-factors)**

CIF structure (conceptual)

- Cell parameters block
- Symmetry block
- Atomic position loop

CIF **always uses fractional coordinates**, and symmetry operations are applied implicitly via the space group, rather than listing every atom explicitly.

POSCAR vs CIF (conceptual comparison)

- **POSCAR**
  - Optimized for simulations (VASP)
  - Explicit lattice vectors
  - Explicit atomic positions
  - No symmetry information required
  - Minimal, fast, and deterministic
- **CIF**
  - Optimized for crystallography and data sharing
  - Uses lattice parameters and space groups
  - Atomic positions defined by symmetry
  - Rich metadata (occupancy, uncertainty, references)

Practical workflow (what you usually do)

1. **Download CIF** from a database (Materials Project, ICSD, COD)
2. **Expand symmetry → full structure**
3. **Convert to POSCAR** for DFT or atomistic simulations

1.5 From Lattice Geometry to Graphs

As mentioned in our first lecture, we often treat matter as a Graph:

- **Nodes (V):** Atoms (Features include atomic number Z, mass, electronegativity).
- **Edges (E):** Bonds or interatomic distances r_{ij} (calculated from the XYZ coordinates above).

By representing crystals this way, we can use Message Passing to let atoms "talk" to their neighbors, allowing the AI to learn how the local environment determines global properties like the band gap.

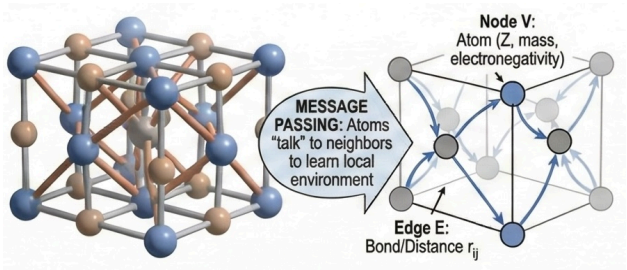

Fig. 5: Graph

1.6 More about Representations: From Atomic Coordinates to Feature Vectors

To bridge the gap between physics and machine learning, we must convert raw atomic data into a format that a neural network can process. This is known as **Representation Learning** or **Feature Engineering**.

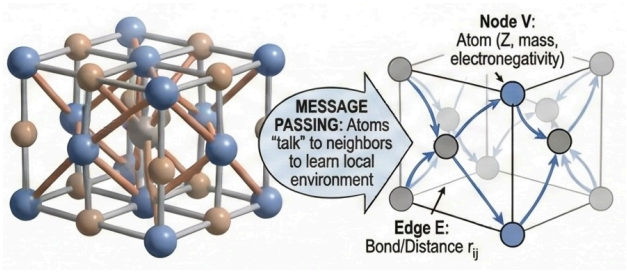

Fig. 6: Representations

1.6.1 The Raw Input

The fundamental description of a crystal system consists of:

- **Atomic Species (Z):** The chemical identity of each atom (e.g., H, Li, Fe).
- **Atomic Positions ($\mathbf{R}$):** The Cartesian coordinates (x, y, z) of every atom in the unit cell.
- **Lattice Vectors ($\mathbf{h}$):** The vectors defining the periodic boundary conditions (the "box").

1.6.2 The Symmetry Challenge

A "naive" input of XYZ coordinates fails because physical properties are independent of how we label the atoms or how we orient the coordinate system. A robust ML representation must satisfy three symmetries:

1. **Translational Invariance:** Moving the entire crystal in space shouldn't change its energy.
2. **Rotational Invariance/Equivariance:** Rotating the crystal shouldn't change scalar properties (like energy), while vector properties (like forces) must rotate with the system.
3. **Permutational Invariance:** Swapping the index of two identical atoms (e.g., Atom 1 and Atom 5 are both Oxygen) must not change the output.

1.6.3 Descriptors vs. Message Passing

There are two primary ways to handle these representations:

- **Fixed Descriptors:** Using mathematical transforms like **Smooth Overlap of Atomic Positions (SOAP)** or **Behler-Parrinello Symmetry Functions**. These transform coordinates into invariant vectors before they enter the ML model.

- **Graph Representations:** Treating the crystal as a graph where atoms are **nodes** and chemical bonds/neighbors are **edges**. Modern **Graph Neural Networks (GNNs)** "learn" the best representation by passing messages between neighboring atoms.

1.7 Physical Targets: T_c, Bandap, Energy, Potential, and Forces....

In the "Physics-ML" workflow, our goal is to train a model that acts as a proxy for the Schrödinger equation. In the following attached example, we will lean the band gap with the datasets obtained from Materials Projects.

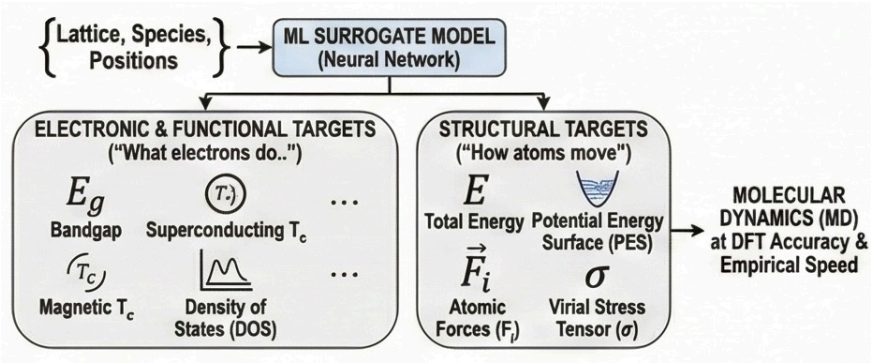

Fig. 7: Target

1.7.1 Electronic & Functional Targets ("What electrons do..")

These targets define the material's performance in technology, from semiconductors to quantum computers.

(1). Electronic Bandgap (E_g)

The energy difference between the Valence Band Maximum (VBM) and Conduction Band Minimum (CBM).

- **Workflow Example:** Using datasets from the Materials Project, we can train a regression model to predict if a material is a metal ($E_g = 0$) or a wide-bandgap semiconductor.

(2). Superconducting Transition Temperature (T_c)

A high-level target deciding when materials have zero resistivity.

- **ML Challenge:** Unlike energy, T_c is highly sensitive to minute structural and doping changes (like moiré twisting or high T_c cuprates). Models often use Compositional Embeddings (MatScholars) or Crystal Graphs (CGCNN) to find new high-T_c candidates.

(3). Magnetic Transition Temperature (T_c)

Temperature under which materials become magnetic.

(4). Density of States (DOS)

Instead of a single scalar, some modern models (like PhREDS) predict the entire DOS curve as a distribution, providing a "fingerprint" of the material's electronic nature.

1.7.2 Structural Targets ("How atoms move")

In phase II, instead of solving for wavefunctions, we want to predict the **Potential Energy Surface (PES)** directly from the atomic coordinates.

(1) The Total Energy (E)

The total energy of a crystal is the fundamental scalar property we seek. In DFT, this is the ground-state energy of the system for a given set of atomic positions $\{\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_N\}$ and lattice vectors.

- **ML Perspective:** This is a **Global Property**. The model takes the entire crystal graph and outputs a single scalar value.
- **Units:** Typically reported in **eV** (electron-volts) or **eV/atom**.

(2) The Potential Energy Surface (PES)

The PES is a high-dimensional landscape where the energy E is a function of the atomic positions $\mathbf{R}$.

- **Stability:** Local minima on this surface correspond to stable or metastable crystal structures.
- **Dynamics:** The "topography" of the PES determines how atoms move, vibrate, and react.

(3) Atomic Forces ($\mathbf{F}$)

While energy is a scalar, forces are vector fields. The force acting on an atom i is defined as the negative gradient of the total energy with respect to its position $\mathbf{R}_i$:

$$\mathbf{F}_i = -\nabla_{\mathbf{R}_i} E = -\left(\frac{\partial E}{\partial x_i}, \frac{\partial E}{\partial y_i}, \frac{\partial E}{\partial z_i} \right)$$

- **ML Requirement (Equivariance):** This is a critical point. If you rotate the crystal structure, the energy (scalar) should remain the same (**Invariance**), but the force vectors must rotate accordingly (**Equivariance**).
- **Targeting Forces:** Modern AI models like Deep Potential (DP) or Equivariant GNNs are trained on both energy and forces simultaneously. This "force-matching" provides significantly more information per training structure than energy alone.

(4) The Virial Stress Tensor (σ)

To predict how a lattice expands or contracts (e.g., during a phase transition), we need the Stress Tensor. This is the derivative of the energy with respect to the lattice strain ϵ:

$$\sigma_{\alpha\beta} = \frac{1}{\Omega}\frac{\partial E}{\partial \epsilon_{\alpha\beta}}$$

where Ω is the volume of the unit cell.

Summary: The ML proxy Model

In the coming weeks, we will build models that perform the following mapping:

$$\{\text{Lattice}, \text{Species}, \text{Positions}\} \xrightarrow{\text{Neural Network}} \{E, \mathbf{F}_i, \sigma\}$$

By mastering this mapping, we can perform **Molecular Dynamics (MD)** simulations at the speed of empirical potentials but with the accuracy of first-principles DFT.

Setting up the Materials Project API

Now that we understand the data structure, let's fetch real structures. We will use the Materials Project (MP), which contains over 150,000 calculated structures and is the primary data source for this course.

Action Required:

1. Go to `materialsproject.org`.
2. Log in and navigate to your **Dashboard**.
3. Locate the API Key section and click "Generate" or "Copy".

> **Important:** Keep this key private. You will need to paste it into the code cell below to authenticate your data queries.

With the API set up, we will use the `mp-api` or `pymatgen` libraries to pull crystal structures directly into this notebook.

The MaterialsProject database

The MaterialsProject (MP) database is a massive amount of material science data that was generated using density functional theory (DFT). Have a look at the database here: https://materialsproject.org.

As discussed in our lecture, the primary bottleneck in condensed matter physics is the expensive experimental and computational screening of new materials. To build AI models that bypass these limits, we need high-quality standardized data.

The **Materials Project (MP)** is a massive open-access database of materials science data generated via high-throughput DFT calculations. It serves as the "ImageNet" for

our field, providing the training data necessary for the "Physics-ML" intuition we aim to build.

Database Statistics & Scope

Explore the database at `materialsproject.org`. Currently, the ecosystem includes:

- **~154,000 Inorganic Crystals:** 3D structures with calculated electronic, magnetic, and thermodynamic properties.
- **~530,000 Nanoporous Materials:** Critical for catalysis and gas storage.
- **Calculated properties:** Band structures, elastic tensors, and X-ray absorption spectra.

Signing up and the API key

You will need to **sign up** in materialsproject.org to be able to access the database. Signing up there is free.

Once you sign up, you can obtain an **API key** that will enable you to access the database using python. Will discuss this further shortly.

A look at MaterialsProject

Let's have a look at the inorganic crystals data. Each one of these crystals is a 3D crystal that includes: a number of elements, arranged in a lattice structure. Check, for example, the MP page for diamond: https://materialsproject.org/materials/mp-66/.

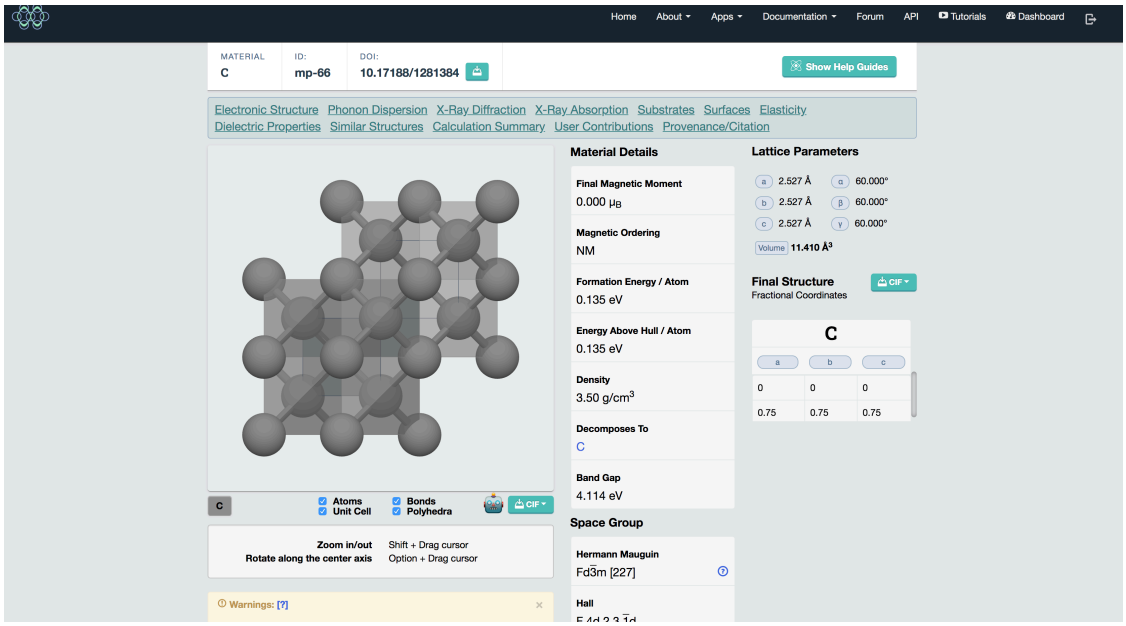

Fig. 7: MP page for diamond

Note that each material on MP is identified by an ID that goes like `mp–X` where `X` is a number. The ID of diamond is `mp–66`. People use these identifiers when referring to MP materials in papers, and we will use them soon when we start querying materials from MP using python.

There you will find the crystal structure, the lattice parameters, the basic properties (in a column to the right of the figure that displays the crystal), and then a range of DFT-calculated properties.

The computational datasets

These are quantities that are calculated for each crystal in MP. In fact, every thing you see on the MP page for diamond was calculated using DFT.

- For a given elemental composition, the lattice parameters and the positions of the atoms within the lattice are all obtained using DFT.
- For the obtained crystal structure, the `Final Magnetic Moment`, `Formation Energy / Atom`, `Energy Above Hull / Atom`, `Band Gap` are calculated. The `Density` is derived from the obtained crystal structure.
- Further DFT calculations are performed to obtain the band structure as well as other properties that you can find as you scroll down the structure page on MP.

Some of the crystals on MP correspond to crystals that exist in nature, and some are purely hypothetical. The hypothetical crystals have been generated by some algorithm that uses artificial intelligence, or probably by simple elemental substitution.

Why is MP great for AIS 5204

Machine Learning requires a Dataset ($\mathbf{X}$) and Target Properties ($\mathbf{Y}$). MP provides:

- **Scale:** Enough data to train Deep Neural Networks.
- **Consistency:** Properties are calculated using a uniform set of DFT parameters, reducing noise in the training set.
- **Discovery:** Many "hypothetical" crystals in MP were generated by AI algorithms or elemental substitution, waiting for us to validate their properties.

The PyMatGen python library

To programmatically access these 150,000+ structures, we use **pymatgen** (Python Materials Genomics). This is the industry-standard library for materials analysis and is a core tool for this course.

Remember: to be able to run the codes in this section, you must obtain an API key from the MP website: https://materialsproject.org/

The first thing we do here is to install PyMatGen in Colab.

```
!pip3 install pymatgen mp-api sklearn pandas
```

Show hidden output

This will download PyMatGen into your Golab environment. Now, we are going to use PyMatGen to do two things: open a CIF crystal file to view its content, and query MP for crystals that satisfy certain properties.

By the way, I used the `pip3` command to install PyMatGen. On your computer, if you have python installed, you can install PyMatGen by typing the same command with the `!`:

```
pip3 install pymatgen
```

Structure file formats

One of the most common file formats that describe crystal structure is the CIF format (Crystallographic Information File). The official definition of this formnat is here: https://www.iucr.org/resources/cif.

But we are not going to learn the details of the format. We will just learn how to open a CIF with pythton. Here is how we can do this.

```
from pymatgen.io.cif import CifParser
from urllib.request import urlopen
import io # Import the io module

request = urlopen("https://raw.githubusercontent.com/sheriftawfikabba
cifFile = request.read().decode('utf-8')
parser = CifParser(io.StringIO(cifFile)) # Fix: Use io.StringIO to cr
```

In the above code, we imported a **class** from the PyMatGen library: the `CifParser` class. It allows us to create a new CIF file **object**. This object will then represent the CIF structure, and can be used to access its information.

Next, let's extract some information from the `CifParser` object.

```
structure = parser.get_structures()
# Returns a list of Structure objects
# #http://pymatgen.org/_modules/pymatgen/core/structure.html
# Let's print the first (and only) Structure object
print(structure[0])
```

Show hidden output

Here we have the details of the CIF structure in a human-readable format, which include the formula, the lattice parameters and the positions of the atoms in the

crystal.

```
structure = structure[0]

print(structure.lattice)
print(structure.species)
print(structure.charge)
print(structure.cart_coords)
print(structure.atomic_numbers)
print(structure.distance_matrix)
```

Show hidden output

Querying structures using PyMatGen

Now let's use PyMatGen to query structures from MP. To be able to do that, we need first to create a `MPRester` with the API key that we receive from MP.

```
from mp_api.client import MPRester
from pymatgen.ext.matproj import MPRestError # Keep for MPRestError i
from google.colab import userdata
import os
#from google.colab import drive
#drive.mount('/content/drive')

# Ensure your Materials Project API key is saved as a secret named 'MI
m = MPRester(userdata.get('MP_API_KEY'))

#os.getcwd()
#os.listdir(".")
#print(os.listdir("/content/drive/MyDrive/NUS/AI4S-5204/nb-byweek/"))
#with open("/content/drive/MyDrive/NUS/AI4S-5204/nb-byweek/MP_API_KEY'
#    api_key = f.read().strip()

#m = MPRester(api_key)
```

**Note: I am hiding my API key here. You'd better not share you API key...

Then we can use the object variable, `m`, to access MP. For today, I willl just show you a simplle query: getting MP IDs for all materials with bandgap larger than 1, band_gap=(1, None) means from 1eV to +∞:

```
results = m.materials.summary.search(band_gap=(1, 100), fields=['mate
```

Retrieving SummaryDoc documents: 100% 55164/55164 [00:11<00:00, 4230.34it/s]

You can notice that this operation takes time: there are 918 such materials.

Finally, let's have a look at the MP IDs we got. This code shows the total count, and the first 10 MP IDs.

```
print(len(results))
results[0:10]
```

Show hidden output

One last thing: let's query a crystal from MP using its MP ID.

```
results=m.materials.summary.search(material_ids=['mp-27971'], fields=
structure_obj = results[0].structure
cifFile = structure_obj.to(fmt="cif")
parser = CifParser(io.StringIO(cifFile)) # Corrected: Use io.StringIO

structure = parser.parse_structures(primitive=True) # Updated to use
print(structure[0])
```

Show hidden output

Part 2: Practical machine learning: poor-man approach

Descriptors

Before diving into the ML code, we must address the fundamental question at the heart of AI-driven material discovery: **How exactly do we translate a physical crystal into a mathematical format that a machine learning model can understand?**

To answer this, we first define our **Target Properties (y)**. These are physical quantities—such as total energy, bandgap, or elastic constants—that are either measured through experiments or calculated using first-principles methods like Density Functional Theory (DFT).

In this course, we focus on **Crystals**, which are defined by their **Periodicity**. Consider the classic example of **Diamond**. While we see a macroscopic gemstone, at the atomic scale, diamond consists of a repeating arrangement of Carbon (C) atoms. If we were to "zoom in" using advanced techniques like Scanning Tunneling Microscopy (STM) or X-ray diffraction, we would observe a highly ordered, infinite lattice similar to the figure below.

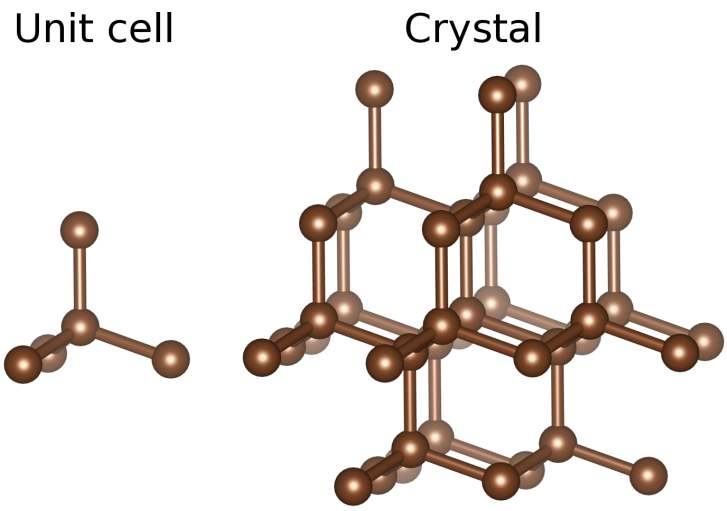

Fig. 8: Left: the diamond unit cell. Right: infinitely repeating the unit cell in the three directions gives the diamond crystal.

The figure above illustrates how a macroscopic diamond crystal is constructed from a **unit cell**—a fundamental building block containing a specific arrangement of atoms (in this case, 4 to 8 Carbon atoms depending on the convention). In a crystal, this unit cell is repeated **infinitely** along the three lattice vectors, creating a continuous 3D pattern.

Molecules vs. Crystals

Before we discuss crystals further, consider molecules for a minute. Molecules are **fundamentally different** from crystals because of the pattern (periodicity) bit: a molecule is just that one single molecule, sitting on its own, in isolation, whereas a crystal is really composed of an infinite number of molecules.

- **Molecules**: These are isolated, finite systems. A water molecule (H_2O), for example, exists in a vacuum and has a unique, fixed geometry.
- **Crystals**: These are infinite periodic systems. A crystal is not merely a large molecule; it is a symmetry-governed tiling of space.
- **Unit Cells**: Crystals are constructed from a fundamental "unit cell"—a building block containing a specific atomic arrangement (e.g., 4 to 8 Carbon atoms for a diamond lattice).
- **Infinite Tiling**: This cell repeats infinitely along three lattice vectors to form a continuous 3D pattern.

The Challenge of Atomic Representation

In molecular machine learning, descriptors like the **Coulomb Matrix** (see Week1-Coulomb matrix.ipynb) are used to represent the relative positions of atoms. Because molecules are finite, it is straightforward to ensure these descriptors are invariant to global translations and rotations.

Crystals, however, present a **non-uniqueness problem**:

- **Arbitrary Selection**: The choice of a unit cell is arbitrary. There are an infinite number of ways to define the "box" that, when repeated, creates the exact same crystal.
- **Coordinate Variance**: Shifting the origin of the box or selecting a different motif results in entirely different atomic coordinates (x, y, z), even though the physical material remains identical.

The key thing here in the molecular descriptors is that they are **based on the atomic positions**. In crystals, **we can't really use atomic positions** like we did with molecules to obtain descriptors. Why?

Think about the diamond crystal pattern above. Because it is a pattern, it is **symmetric in all three directions**. Now let's say we are going to calculate the Coulomb matrix for diamond, so we start with the positions of the atoms in the unit cell of diamond, that figure on the left. But wait: even though these four atoms do form a valid unit cell for diamond, we can also come up with another valid unit cell, as shown below.

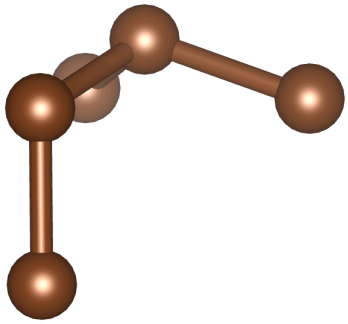

Fig. 9: Another possible structure for the diamond unit cell.

The figure above is also valid. How to obtain it? Look at the diamond crystal in Figure 8, and take a different repeating unit from it. As long as this repeating unit can also form the same pattern, it is a valid unit cell!

So, there are many different ways we can represent the unit cell of a crystal. Therefore, we cannot use the atomic coordinates to derive descriptors for crystals, otherwise the derived descriptors, such as the eigenvalues of the Coulomb matrix in the case of molecules, will **change dramatically for the same crystal**. That is, **the descriptor vector is not invariant with respect to translation of the unit cell**. What do we do then?

Building a simple descriptor vector for crystals

The Invariance Crisis

This non-uniqueness leads to a representation crisis in ML. If a descriptor is derived purely from coordinates within a specific unit cell:

1. The resulting vector changes depending on where the cell boundaries are drawn.
2. The descriptor lacks **translation invariance** regarding the unit cell boundary.

To solve this, specialized models like **Graph Neural Networks (GNNs)** and **Symmetry Functions** are used. Instead of relying on absolute coordinates, these methods focus on **local atomic environments**, creating a consistent mathematical "fingerprint" regardless of the chosen unit cell. We will discuss

---

A possible **poor-man solution for now** to this problem is to use some statistics of atomic properties as the descriptor vector. For example:

- Average of the atomic numbers of all the elements in the crystal. For example, in silicon carbide, SiC, the average value would be the average of 14 (for Si) and 6 (for C), which is (14 + 6)/2 = 10. So that's now one number in the descriptor vector.
- The average of the ionization potential of the atoms
- The average of the electron affinity of the atoms
- And more averages

So we can keep adding averages of properties to this list, to expand the descriptor vector. This vector will not suffer from the lack of invariance issue pointed out above, because these are average values of quantities that do not depend on the geometry of the crystal.

Average is just one statistic. We can also add other statistics, such as the standard deviation and the variance. Adding those will triple the number elements in the descriptor vector above.

```
import numpy as np
from pymatgen.core.structure import Structure # Import Structure for
print(structure[0])
# Ensure 'structure' refers to the Pymatgen Structure object.

# Extract atomic numbers using the appropriate property for a Structu
atomic_numbers_list = structure[0].atomic_numbers

mean_atomic_number = np.mean(atomic_numbers_list)
max_atomic_number = np.max(atomic_numbers_list)
min_atomic_number = np.min(atomic_numbers_list)
std_atomic_number = np.std(atomic_numbers_list)

print(mean_atomic_number, max_atomic_number, min_atomic_number, std_a
```

```
Full Formula (Ac2 Cl6)
Reduced Formula: AcCl3
abc   :   4.569003   7.695636   7.695635
angles: 119.999998  90.000000  90.000000
pbc   :       True       True       True
Sites (8)
```

```
  #  SP       a         b         c
---  ----  ----  --------  --------
  0  Ac    0.25  0.333333  0.666667
  1  Ac    0.75  0.666667  0.333333
  2  Cl    0.25  0.389595  0.085802
  3  Cl    0.75  0.085802  0.696207
  4  Cl    0.75  0.610405  0.914198
  5  Cl    0.25  0.914198  0.303793
  6  Cl    0.25  0.696207  0.610405
  7  Cl    0.75  0.303793  0.389595
35.0 89 17 31.176914536239792
```

However, there is a **problem**. A lot of materials exist in **various phases**. That is, for the same atomic composition, let's say SiC, there are several possible structures. Right now, there are 27 possible structures for SiC on MaterialsProject.org.

So, the above descriptors won't work. For example, for the case of SiC, all of the 27 SiC phases in MP will have the same values for the statistical values above.

To solve this problem, we have to add descriptors **based on the geometrical arrangement of atoms**. A simple such descriptor is to average the bond lengths (a bond is formed between two atoms).

```
mean_distance_matrix = np.mean(structure[0].distance_matrix)
max_distance_matrix = np.max(structure[0].distance_matrix)
min_distance_matrix = np.min(structure[0].distance_matrix)
std_distance_matrix = np.std(structure[0].distance_matrix)

print(mean_distance_matrix, max_distance_matrix,
      min_distance_matrix, std_distance_matrix)
```

```
2.9494274703814507 4.995986576538145 0.0 1.173431851403311
```

Building a data set

Now it's time to build our dataset, before we can do machine learning on it. We will do this in 3 steps:

- Step 1: collecting the structures
- Step 2: pre-processing the data

Step 1: Collecting the structures

We want to predict the bandgaps of structures, so we need to collect the structures (dataset) along with their corresponding bandgaps (target vector).

For this exercise, let's focus on stoichiometric perovskites: these are materials of the form ABC3. The followiing query will collect the CIFs and bandgaps for these materials from MP.

```
results = m.materials.summary.search(formula="ABC3", fields=["structu
```

Show hidden output

Step 2: Pre-processing the data

Here we will extract the data we need from the structures, put them in a pandas DataFrame and then apply normalization.

```
import io
import numpy as np
from pymatgen.io.cif import CifParser

# atomic number ranges roughly covering alkali/alkaline earth + trans
_METAL_RANGES = [
    range(3, 5),    # Li–Be
    range(11, 13),  # Na–Mg
    range(19, 31),  # K–Zn
    range(37, 49),  # Rb–Cd
    range(55, 81),  # Cs–Hg (incl. lanthanides)
    range(87, 113), # Fr–Cn (actinides +)
]

# element attributes to aggregate (mean, max, min, std) over species
_ELEM_ATTRS = [
    "van_der_waals_radius",
    "electrical_resistivity",
    "velocity_of_sound",
    "reflectivity",
    "poissons_ratio",
    "molar_volume",
    "thermal_conductivity",
    "melting_point",
    "critical_temperature",
    "superconduction_temperature",
    "liquid_range",
    "bulk_modulus",
    "youngs_modulus",
    "brinell_hardness",
    "rigidity_modulus",
    "vickers_hardness",
    "density_of_solid",
    "coefficient_of_linear_thermal_expansion",
    "average_ionic_radius",
    "average_cationic_radius",
    "average_anionic_radius",
]

def _stats(values):
    """Return [mean, max, min, std] for a 1D array-like."""
    v = np.asarray(values, dtype=float)
    return [float(np.mean(v)), float(np.max(v)), float(np.min(v)), fl
```

```
def _safe_attr(specie, name, default=0.0):
    """Get specie.<name>, replace None with default."""
    val = getattr(specie, name, None)
    return default if val is None else float(val)

def descriptors(cif_str: str):
    # parse CIF from string
    structure = CifParser(io.StringIO(cif_str)).parse_structures(prim

    Zs = np.asarray(structure.atomic_numbers, dtype=float)
    n_atoms = len(Zs)

    # metals fraction (count per site)
    n_metals = sum(any(sp.Z in r for r in _METAL_RANGES) for sp in st
    metals_fraction = n_metals / n_atoms if n_atoms else 0.0

    # space group one-hot (1..230)
    spg_num = structure.get_space_group_info()[1]
    spg_onehot = [1 if i == spg_num else 0 for i in range(1, 231)]

    # basic lattice / geometry
    a, b, c = structure.lattice.abc
    alpha, beta, gamma = structure.lattice.angles

    dist = structure.distance_matrix
    dist_stats = _stats(dist.ravel())  # flatten matrix

    # element-level attributes stats over species
    attr_stats = []
    for name in _ELEM_ATTRS:
        vals = [_safe_attr(sp, name) for sp in structure.species]
        attr_stats.extend(_stats(vals))

    # your original "Density" definition (volume per atom)
    V = a * b * c
    density_like = V / n_atoms if n_atoms else 0.0

    # assemble feature vector (same overall ordering intention, but c
    feat = []
    feat += _stats(Zs)                 # mean, max, min, std of atomi
    feat += [density_like]
    feat += [alpha, beta, gamma]
    feat += [metals_fraction]
    feat += dist_stats
    feat += attr_stats
    feat += spg_onehot

    return feat

# usage:
# features = descriptors(cifFile)   # cifFile is a CIF string
```

Now let's iterate through the list of results and extract our descriptors into the above lists. This will take a few minutes.

```
band_gaps = []
dataset = []

counter =0
for r in results:
    structure_obj = r.structure # Access the Structure object
    cif = structure_obj.to(fmt="cif") # Convert Structure object to C
    bg = r['band_gap']
    parser = CifParser(io.StringIO(cif)) # Corrected: Use io.StringIO

    structure = parser.parse_structures(primitive=True)[0] # Updated

    dataset += [descriptors(cif)]

    band_gaps += [bg]
    print(counter)
    counter +=1

dataset_df = pd.DataFrame(dataset)
```

Show hidden output

Now that we created our dataset, we need to have a bird's eye view of the data. For now, let's have a look at how the features and bandgap values look like.

```
print(dataset[0],band_gaps[0])
```

Show hidden output

```
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 20})

plt.figure(figsize=(10, 10))
plt.hist(band_gaps, bins=100)
plt.xlabel('band gap(eV)')
plt.savefig('Histogram_PDF', bbox_inches='tight')
```

Show hidden output

This plot shows that amost half of our structures are metals (zero bandgap). The bandgaps around 7 eV could be outliers, but we can deal with those in a later lecture.

How about a scatter plot?

```
band_gaps_sorted=sorted(band_gaps)

# Scatter plot
plt.figure(figsize=(10,10))
plt.plot(band_gaps_sorted)
plt.ylabel('band gap(eV)')
plt.xlabel('material count')
plt.savefig('ScatterPlot', bbox_inches='tight')
```

Show hidden output

Next, we split the dataset into training and test sets, and we use the 80/20 split ratio.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    dataset_df, band_gaps, test_size=.2, random_state=None)
```

Then we normalize our dataset using the normalization applied on the training set.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# We need to normalize the data using a scaler

# Define the scaler
scaler = StandardScaler().fit(X_train)

# Scale the training and test sets
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Next, we create a pandas DataFrame object
```

The machine learning task

Now it's time to actually do machine learning. We will try two machine learning models: the random forests and the XGBOOST models. We will quantify the prediction accuracy using two measures: goodness of fit (R2) and the mean squared error (MSE).

```
from sklearn.metrics import mean_absolute_error, r2_score
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
regr = RandomForestRegressor(n_estimators=400, max_depth=400, random_
regr.fit(X_train_scaled, y_train)
y_predicted = regr.predict(X_test_scaled)

print('RF MAE\t'+str(mean_absolute_error(y_test, y_predicted))+'\n')
print('RF R2\t'+str(r2_score(y_test, y_predicted))+'\n')

xPlot=y_test
yPlot=y_predicted
plt.figure(figsize=(10,10))
plt.plot(xPlot,yPlot,'ro')
plt.plot(xPlot,xPlot)
plt.ylabel('RF')
plt.xlabel('DFT')
plt.savefig('RF_Correlation_Test', bbox_inches='tight')


regr = XGBRegressor(objective='reg:squarederror', max_depth=10, n_est
regr.fit(X_train_scaled, y_train)
y_predicted = regr.predict(X_test_scaled)

print('XGBOOST MAE\t'+str(mean_absolute_error(y_test, y_predicted))+'
print('XGBOOST R2\t'+str(r2_score(y_test, y_predicted))+'\n')


xPlot=y_test
yPlot=y_predicted
plt.figure(figsize=(10,10))
plt.plot(xPlot,yPlot,'ro')
plt.plot(xPlot,xPlot)
plt.ylabel('XGBOOST')
plt.xlabel('DFT')
plt.savefig('XGBOOST_Correlation_Test', bbox_inches='tight')
```

Show hidden output

Achieving R^2 > 0.71 and MAE < 0.5 a promising start, demonstrating that our basic structural features capture significant physical trends. However, to bridge the gap between "good" approximation and "publication-quality" prediction, we need deeper descriptors.

That concludes this introduction—you are now ready to tackle high-fidelity property prediction!