Problem Set 5

Reminder: No AI Tools

According to MSSE Department policy, use of AI tools is not permitted in Chem 274A. Do not use generative AI tools (e.g., ChatGPT, Claude, GitHub Copilot, or similar) for any part of this assignment, including planning, coding, or writing.

Showing your work

Your homework assignments should show incremental development with at least two different commits over two different days of work. Assignments that do not show incremental development will incur a 5% penalty.

Python – Residue Templates and CIF Parsing

For this homework, you will create a Python class to represent residues. Residues are the building blocks of proteins and nucleic acids. You will then add functionality to read residue information from Crystallographic Information Files (CIFs) using the `gemmi` library.

You can install gemmi using

```
conda install -c conda-forge gemmi
```

This assignment builds directly toward your final project, where you'll use these residue templates to construct polymer chains.

What is a residue?

In structural biology, a **residue** is a single building block in a polymer chain. Both proteins and nucleic acids are made up of repeated building blocks.

- In proteins: one amino acid (like alanine, lysine, etc.)
- In nucleic acids: one nucleotide (like adenine, thymine, etc.)

Each residue is described by:

- All of its constituent atoms

- The bonds connecting those atoms
- Identifying codes (like "ALA" for alanine or "A" for its one-letter code)
- The 3D coordinates of each atom in an idealized geometry

Think of a residue template as a "blueprint" that tells you everything about that particular building block before it's placed into a larger structure.

Requirements and Constraints

This assignment uses test-driven development (TDD). Tests are provided in `test_residues.py` that define the interface of your `Residue` class. However, please read this entire assignment explanation before jumping into the test file. Understanding the big picture will help you make better design decisions.

Library restrictions: You will use gemmi to read CIFs for this assignment. You may also consider using NetworkX for molecule representation, like in previous assignments, but this is not required. You are strongly encouraged to consider using NumPy for storing and managing coordinate data to optimize for the geometric operations required in the final project. In this assignment, you may use pytest, gemmi, networkx, numpy, matplotlib, and the Python Standard Library.

Part 1 – Designing the Residue Class

Your first task is to design and implement a `Residue` class that stores all the information about a residue template. The tests guide you through this incrementally.

What Your Residue Must Store

The tests in `test_residues.py` specify exactly which fields your residue class needs and how it should behave. In general, your residue template class will need to store:

- **Residue name** (the full name, like "ALANINE")
- **One-letter code** (like "A" for alanine)
- **Three-letter code** (like "ALA" for alanine)
- **Residue type** (either "amino_acid" or "nucleic_acid")
- **Atom IDs** (the names of each atom, like "CA", "CB", "N")
- **Element symbols** (what element each atom is, like "C", "N", "O")
- **Ideal 3D coordinates** (x, y, z positions for each atom)
- **Bond information** (which pairs of atoms are connected, and what the bond order is)

Consider whether you'd like to store this information as Python data types, or if you would like to use NetworkX like in previous assignments.

Implementation Strategy

The test file is organized into **stages** that guide your incremental development. You can run tests for specific stages using:

```
 pytest -vm stage1  # Run only stage 1 tests
pytest -vm "stage1 or stage2"  # Run stages 1 and 2
```

Running Tests Incrementally

Start with Stage 1 tests and work your way forward:

```
pytest -vm stage1
```

Once Stage 1 passes, move to Stage 2, and so on. Don't wait until everything is implemented to run tests—run them frequently as you work. Not that Stage 4 requires CIF files, so read `Part 2 - Reading Data from CIF` before completing this part.

Part 2 – Reading Data from CIF

This is `Stage 4` in testing.

Now that you have a working `Residue` class, you'll add the ability to create residues from Crystallographic Information Files (CIFs).

What is a CIF?

Residue CIF files (for example, `ALA.cif`, `LYS.cif`, `DA.cif`, `DT.cif`) are structured text files that contain information about residues. They contain three main categories of information:

1. **Residue-level information** – the names and codes that identify this residue (e.g., "alanine", "ALA", "A")
2. **Atom-level information** – for each atom in the residue: atom IDs (like "CA", "CB"), element symbols (like "C", "N"), and ideal 3D coordinates
3. **Bond-level information** – which atoms are bonded to which other atoms, and what type of bond connects them (single, double, etc.)

Exploring CIF Structure

Before implementing CIF parsing, explore the structure of a CIF file:

First, open `ALA.cif` in your IDE. The first few lines look like:

```
 data_ALA
#
_chem_comp.id                                   ALA
_chem_comp.name                                 ALANINE
_chem_comp.type                                 "L-PEPTIDE LINKING"
_chem_comp.pdbx_type                            ATOMP
_chem_comp.formula                              "C3 H7 N O2"
```

Further down (around line 51), you'll see coordinate data:

```
 ALA N   N   N 0 1 N N N Y Y N 2.281  26.213 12.804 -0.966 0.493  1.500  N   A
ALA CA  CA  C 0 1 N N S Y N N 1.169  26.942 13.411 0.257  0.418  0.692  CA  A
ALA C   C   C 0 1 N N N Y N Y 1.539  28.344 13.874 -0.094 0.017  -0.716 C   A
```

Next, use the starter code with gemmi to examine how this data is structured:

1. **Run the starter code** and examine the dictionary structure gemmi creates
2. **Insert a `breakpoint()` or print statements** to explore the dictionary interactively
3. **Locate the `_chem_comp` block** – contains residue-level information
4. **Locate `_chem_comp_atom`** – contains atom information (IDs, elements, coordinates)
5. **Locate `_chem_comp_bond`** – contains bond information

Take time to understand the structure. What keys are present? How are lists of atoms represented? How are bonds specified?

Hint: Distinguishing Residue Types

The `_chem_comp.type` field contains information about what kind of residue this is:

- Amino acids typically contain `"PEPTIDE"` in the type field
- Nucleic acids typically contain `"DNA"` or `"RNA"` in the type field

You can use this field to determine whether a residue is an amino acid or nucleic acid.

You will need to get all information required to construct a residue. You should be getting the ideal coordinates from these files.

Retrieve information from the CIF for constructing a residue. Note that the tests require you to also write a `from_cif` method on your class.

Part 3 – Factory Function for Convenience

Finally, you'll create a convenience function that makes it easier to create residues without knowing file paths.

Stage 5: Factory Function (Tests: `test_create_residue_exists`, `test_create_residue_returns_residue`, `test_create_residue_loads_correct_residue`)

Create a function called `create_residue(code)` that:

- Takes a three-letter residue code (like "ALA", "LYS", "DA")
- Determines the appropriate file path for that residue's CIF file
- Calls `Residue.from_cif(path)` and returns the result

This is an example of the **factory function** design pattern—a function that encapsulates the logic of creating objects.

Part 4 – Documentation and Discussion

As always, your submission must include a `README.md` file with installation instructions and usage examples.

What to Include in Your README

Your README should contain:

1. **Installation instructions** – How to set up the environment and install dependencies
2. **Usage examples** – How to create residues both manually and from CIF files
3. **Design discussion** – Answers to the reflection questions below

Reflection Questions

Please provide answers to the following questions in your README:

Answering Reflection Questions

Your written answers should be *complete explanations*, not short phrases. A complete answer:

- References specific Python concepts, data types, or library features relevant to the question.
- Explains the reasoning and constraints, not just what you did (say *why* a particular approach works or doesn't work).

- Uses correct terminology from Python programming and relevant domains where appropriate.
- Is written in full sentences and paragraphs.

Grading: Full credit requires answers that demonstrate understanding of underlying concepts and explain *why*. Answers that only describe what you did without explanation will receive partial credit.

1. Data Representation

- What data structures did the tests require you to use for returning atom information (e.g., atom IDs, coordinates, bonds)? Why are these data structures appropriate for this data?
- Did you use NetworkX for your residue template? Why or why not?

2. Properties vs. Stored Attributes

- What is a decorator in Python, and how does the @property decorator work?
- The tests require n_atoms to be a @property rather than a regular attribute. Explain why a property is more appropriate here than computing and storing the count during `__init__`.
- Did you use @property for any other attributes? If so, which ones and why? If not, why not?

3. Factory Function

- What is a factory function, and how does it differ from a regular function or a classmethod?
- Explain the design decisions you made in implementing `create_residue()`. Where does it look for CIF files? How did you decide on this approach?
- Compare the user experience of `create_residue("ALA")` versus `Residue.from_cif("path/to/ALA.cif")`. What are the advantages and disadvantages of each approach?

4. Test-Driven Development

- How did having tests before implementation change your development process compared to previous assignments?
- The tests are organized into stages. How did this staged approach help (or not help) you develop your code incrementally?

5. Test Decorators

- Identify at least two different decorators used in the test file. What does each decorator do, and why might it be useful in organizing and running tests?

C++ - Eigen and Principal Moments of Inertia

In this repo, the `molecule.cpp` file contains some of the Molecule class from the last problem set. I have given you a version of the code that calculates the inertia tensor. You have the following two tasks - convert the code to use Eigen rather than nested `std::array`, and then to use Eigen to compute the principal moments of inertia, from which you will determine the molecular rotor type.

We talked about the Eigen library in lecture. I have included a version of the library in this repo, so you do not have to install it. Use `-I./include` when compiling the `molecule.cpp` file.

Use the functions you create in this section on the two molecules in the `main` function - one representing water, and the other methane.

Makefile: Create a makefile that compiles and runs your code

Converting the code to use Eigen

In the current code, the `inertia_tensor` function returns a 3x3 matrix stored in nested arrays. This is kind of annoying, and it should really be stored in a matrix.

Modify the code so that Eigen matrices are used rather than nested arrays.

Principal Moments of Inertia

The principal moments of inertia are an important quantity which can be used to determine some things about rotational behavior of the molecule, as well as for aligning molecules in a standard orientation.

The principal moments of inertia are calculated by diagonalizing the inertia tensor, forming its eigenvectors and eigenvalues.

For an example of how to do this in Eigen, see here.

Write a class method that returns the eigenvalues from the above diagonalization.

Note: The inertia tensor is represented by *symmetric* matrices. The eigenvalues and eigenvectors of a matrix are complex unless the matrix is hermitian (or self-adjoint), in which case they are real. A real, symmetric matrix is hermitian/self-adjoint, so we can use the eigensolver from Eigen that handles self-adjoint matrices. If you used the regular Eigensolver, you would have to allow for complex eigenvalues and eigenvectors.

Molecular Rotor Type

The eigenvalues obtained from the previous step can be used to determine the molecular rotor type.

- If all moments are equal (to within some tolerance), then the rotor type is a "spherical top"
- If one moment is small (close to zero) and the others are equal, the rotor type is "linear"
- If two moments are equal (but different from the third) then the rotor type is is a "symmetric top"

- If all three moments are different, the rotor type is "asymmetric top"

Write a class method that returns the molecular rotor type (as a string).

Testing your code

In the `main` function in the `molecule.cpp` file, I have given you the atomic masses and coordinates for a water molecule and methane molecule. Using your additions to the Molecule class, calculate (and print out) the inertia tensor, principal moments of inertia, and molecular rotor type for each molecule.