Homework 1: Introduction to Python Computations

CHEM4050/5050 Fall 2025

due Friday, September 5, 2025, at 11:59 PM Central

1. **Problem Statement**: You are given a dataset that records the relationship between volume and pressure (in Hg) in a closed system. Your task is to analyze this data and explore the nature of this relationship using Python, NumPy, Pandas, and Matplotlib.

   Dataset: The dataset is provided as a CSV file (`volume_pressure_data.csv`) containing the following columns:

   - **Volume**: The volume of the gas in the system (in^3).
   - **Pressure**: The pressure the gas exerts (in Hg).

   Tasks

   (a) **Data Import and Exploration**

   - ☐ Use **Pandas** to import the CSV file and explore the data.
   - ☐ Display the first few rows of the dataset.
   - ☐ Use Pandas to calculate volume and pressure's mean, median, and standard deviation.

   (b) **Data Visualization**

   - ☐ Using **Matplotlib**, plot the volume vs. pressure data to visualize the relationship between the two variables.
   - ☐ Label the axes appropriately and give the plot a title.
   - ☐ Include grid lines for clarity.

   (c) **Curve Fitting and Modeling**

   - ☐ Use **NumPy** to fit a polynomial curve to the data. Start by trying a second-degree polynomial (quadratic). You can use `numpy.polyfit` for this purpose.
   - ☐ Plot the original data and the fitted polynomial curve on the same graph.
   - ☐ Use Matplotlib to create a legend indicating which curve is the fit.

   (d) **Deriving a Mathematical Model**

   - ☐ Based on the polynomial fit, write down the curve equation.
   - ☐ Determine if this relationship makes physical sense, given what you know about gases (hint: think of Boyle's Law, $P \times V =$ constant for an ideal gas).

   (e) **Error Analysis**

   - ☐ Calculate the root mean square error (RMSE) between the fitted polynomial curve and the actual data points. Use NumPy for this calculation.
   - ☐ Report the RMSE value and briefly discuss whether the polynomial fit represents the data well.

   Submission Guidelines:

   - Push a Jupyter notebook named `homework-1-1.ipynb` that implements all the above tasks to your GitHub repository named `chem-4050-5050` for this course.
   - Include comments in your code explaining each step.
   - Your plots should be clear, well-labeled, and properly formatted.

2. **Problem Statement**: In this exercise, you will use the **Computational Chemistry Comparison and Benchmark DataBase (CCCBDB)** by NIST to retrieve molecular geometry data and compute geometric properties of selected molecules using Python. You will also practice using **if/elif/else** statements, **loops**, and **functions** in Python.

   Dataset Access: To get started, follow these steps to retrieve molecular data from the CCCBDB website:

   (a) Go to the CCCBDB website: https://cccbdb.nist.gov/.

   (b) Hover over the "Experimental" menu and click on "One molecule all properties."

   (c) Enter the chemical formula of a molecule (e.g., H_2O, CO_2, NH_3) and click Submit.

   (d) If you encounter an error message or the page doesn't load, try refreshing the page, using a different browser, or using private browsing mode. Structure files will be posted on Canvas if you continue to have trouble.

   (e) Once the molecule's data page loads, scroll to the Geometric Data section and find the Cartesian coordinates of the atoms (labeled as "Cartesians").

   Tasks

   Part 1: Importing and Exploring Data

   ☐ Collect the Cartesian coordinates of H_2, H_2O, and benzene from the CCCBDB website.

   ☐ Store the Cartesian coordinates of each molecule in Python dictionaries. For example:

```
molecule = {
    "H1": [x1, y1, z1],
    "O": [x2, y2, z2],
    "H2": [x3, y3, z3]
}
```

   ☐ Print the coordinates of each molecule to verify that they are stored correctly.

   Part 2: Bond Length Calculation: Write a Python function to compute the bond length between two atoms using their Cartesian coordinates.

   ☐ Implement a function `compute_bond_length(coord1, coord2)` that takes two 3D coordinate lists (*e.g.*, `[x1, y1, z1], [x2, y2, z2]`) as input and returns the bond length d between the two atoms.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

   ☐ Use an `if` statement to check if the calculated bond length falls within a reasonable range for covalent bonds (*e.g.*, less than 2 Å). If the bond is too long, print a warning.

   Part 3: Bond Angle Calculation: Write a Python function to compute the bond angle between three atoms using their Cartesian coordinates.

   ☐ Implement a function `compute_bond_angle(coord1, coord2, coord3)` that takes the coordinates of three atoms as input and returns the bond angle in degrees. Use the dot product of vectors to compute the bond angle:

$$\cos(\theta) = \frac{\vec{AB} \cdot \vec{BC}}{|\vec{AB}||\vec{BC}|}$$

   Where vectors $\vec{AB}$ and $\vec{BC}$ are the vectors between the atom pairs.

   ☐ Use an `if/else` block to classify the bond angle as acute, right, or obtuse based on the calculated value.

   Part 4: Automating the Calculation of Unique Bond Lengths and Angles: Write a loop that automatically calculates all unique bond lengths and bond angles for each molecule without specifying which atoms are bonded manually.

Task 4.1: Calculating All Unique Bond Lengths: Write a loop to compute the bond lengths for every unique pair of atoms in a molecule.

- ☐ Write a function `calculate_all_bond_lengths(molecule)` that takes a dictionary of Cartesian coordinates as input.
- ☐ Use a **nested for loop** to iterate through every unique pair of atoms in the molecule (you can skip duplicate pairs). A nested for loop is a loop within another loop. The inner loop (the one inside) executes completely each time the outer loop runs once. In other words, for every iteration of the outer loop, the inner loop will run from start to finish.
- ☐ For each unique pair, call your `compute_bond_length` function to calculate the bond length.
- ☐ Store the calculated bond lengths in a list and print them. You can store values in a list by creating an empty list where values can be added (*e.g.*, `bond_lengths = []`) and use the `.append()` method to add values to the list one at a time.

Hint: You can avoid duplicate pairs by looping over atom pairs to ensure you only consider the pairs where the second atom comes after the first atom in the dictionary keys. Example:

```
for atom1 in molecule:
    for atom2 in molecule:
        if atom1 != atom2:
            # Compute bond length only if it's a unique pair
```

In Python, `!=` is a **comparison operator** that stands for **"not equal to."**

Operator	Name	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 10`	`True`
`>`	Greater than	`10 > 5`	`True`
`<`	Less than	`3 < 7`	`True`
`>=`	Greater than or equal to	`7 >= 7`	`True`
`<=`	Less than or equal to	`4 <= 4`	`True`

Table 1: Common Comparison Operators in Python

Task 4.2: Calculating All Unique Bond Angles: Write a loop to compute bond angles for every unique set of three atoms in a molecule.

- ☐ Write a function `calculate_all_bond_angles(molecule)` that takes a dictionary of Cartesian coordinates as input.
- ☐ Use a nested for loop to iterate over all unique sets of three atoms in the molecule.
- ☐ For each set of three atoms, call your `compute_bond_angle` function to calculate the bond angle.
- ☐ Store the calculated bond angles in a list and print them.

Hint: You can generate unique triplets of atoms similar to bond pairs, ensuring that the second and third atoms always come after the first atom.

Submission Guidelines:

- Push a Jupyter notebook named `homework-1-2.ipynb` that implements all the above tasks to your GitHub repository named `chem-4050-5050` for this course.
- Include comments in your code explaining each step.
- For each molecule, your script should now:
  - Calculate all **unique bond lengths** between pairs of atoms.
  - Calculate all **unique bond angles** between triplets of atoms.
  - Store and print the results.

Graduate Supplement

3. **Problem Statement**: In this problem, you will solve the time-independent Schrödinger equation for a particle in a one-dimensional infinite potential well using Python and a real-space grid method. The time-independent Schrödinger equation is:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} = E\psi(x)$$

where $\hbar$ is the reduced Planck's constant, m is the mass of the particle, E is the energy of the particle, and $\psi(x)$ is the wavefunction of the particle. We will solve this equation numerically by discretizing the second derivative using a finite difference method and constructing the Hamiltonian matrix for the system. The eigenvalues of the Hamiltonian matrix will give us the energy levels, and the eigenvectors will give the corresponding wavefunctions.

Tasks

(a) **Define Constants and Discretize the System**

- ☐ Use **atomic units**, where $\hbar = 1$ and $m = 1$.
- ☐ The width of the infinite potential well is $L = 1.0a_0$ (Bohr radii).
- ☐ Discretize the space using a real-space grid of **2000 points** from $-L/2$ to $L/2$.

(b) **Construct the Laplacian Matrix**

- ☐ The second derivative $\frac{d^2\psi(x)}{dx^2}$ can be approximated using a **finite difference** method. The Laplacian matrix for the finite difference method is given by:

$$\text{Laplacian} = \frac{1}{(\Delta x)^2}\left(-2\mathbf{I} + \mathbf{I}_{\text{off-diagonal}}\right)$$

  where $\mathbf{I}$ is the identity matrix, $\mathbf{I}_{\text{off-diagonal}}$ contains ones on the sub- and super-diagonal, and Δx represents the **spacing between adjacent points** on a discretized grid.
- ☐ Write a Python function that constructs this Laplacian matrix using NumPy. For a system with 5 points in space, the Laplacian matrix looks like this:

$$\text{Laplacian} = \frac{1}{(\Delta x)^2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

(c) **Construct the Hamiltonian Matrix**

- ☐ The Hamiltonian operator in the discretized form is given by:

$$H = -\frac{\hbar^2}{2m} \cdot \text{Laplacian}$$

  In atomic units, this simplifies to:

$$H = -\frac{1}{2} \cdot \text{Laplacian}$$

- ☐ Calculate this Hamiltonian matrix, which will be used to compute its eigenvalues and eigenvectors, which correspond to the energies and wavefunctions of the system.

(d) **Solve for Eigenvalues and Eigenfunctions**

- ☐ Use **NumPy's linear algebra** package (`np.linalg.eig`) to compute the eigenvalues (energy levels) and eigenfunctions of the Hamiltonian.
- ☐ Sort the eigenvalues in increasing order and extract the first seven energy levels.

(e) **Plot the Results**

- ☐ Plot the first five wavefunctions along with their corresponding energy levels.
- ☐ Label the axes appropriately and include the wavefunctions in the correct units.

Submission Guidelines:

- Push a Jupyter notebook named `homework-1-grad.ipynb` that implements all the above tasks to your GitHub repository named `chem-4050-5050` for this course.
- Include comments in your code explaining each step.
- Your Jupyter notebook should include:
  - The constants and the Laplacian matrix construction.
  - The Hamiltonian matrix and the eigenvalue/eigenfunction computation.
  - A plot of the first five energy levels and wavefunctions.